Tools and Commands

Commands:

- **find**
- **tar, gzip, gunzip**

Filters:

- **grep**
- **sort**
- **sed**
- **awk**

Searching Subdirectories

- **find <dir> [-opt]**
- Some options:
  - **-name pattern**
    -type [b c d l r]

if pattern is a regular expression
it must be quoted between **'…'**
- **Example: find / -name '*.c'**

find

- **find** not only finds filenames in the directory sub-trees, but also can execute a command (or a script) on every file matching the previous expressions using the option

  -exec command \;
- In **command**, **{}** is the name of the current file matching the previous expressions
- Example:

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

find

- The search expression can be composed of more than one conditions that can be connected by means of logical operations
  - AND operation: list of conditions
  - OR: list of conditions connected by the operator `-o`
  - NOT: the negation operator is `!`

find

Use parentheses to enclose logical expressions

Example:

```
find . \( -name core -o -size 10b \)

find . \( -name test -o ! \( -size 10c \) \)
```

`\` protects `(` that would be otherwise interpreted by the shell

10c means 10 bytes

10b means 10 blocks

grep

- Searches the occurrence of **`pattern`** in the list of **`files`**

```
grep [-options] pattern files
```

grep

- Options:
  - `-v` shows the lines that do not match `pattern`
  - `-n` shows the line number of the lines that match `pattern`
  - `-c` counts the lines that match `pattern`
  - `-i` ignores lower/upper case difference
  - `-l` shows only the file names (not also the lines) that match `pattern`

Regular Expressions in grep

- A `pattern` can be a regular expression
- In a regular expression some characters have special meaning unless they are preceded by `\`
  - `.` a single character
  - `^` beginning of line
  - `$` end of line

Regular Expressions in grep

- `*` repeat (zero or more times)
- `+` repeat (one or more times)
- `[ ]` one among the characters in parentheses
- `[^ ]` a character excluding the ones in parentheses
- `\<` beginning of word
- `\>` end of word

Use of `find` and `grep`

```
find . -name '*' -exec grep pattern {} \;
```

- Searches all the files in the directory sub-tree rooted at the current directory (`.`)
- For each filename matching the first condition (`-name '*'`) the command argument of the `–exec` condition is executed
- The command is `grep 'pattern' {}` terminated by `\;`
- `{}` is the notation for the matching filename

Archiving files: `tar`

- `tar [option] tarfile.tar files`
- Creates or appends a list of files to a single tarfile `tarfile.tar`, and performs all the typical archiving operation (list of content, extraction of files, etc.

`tar` : Creation options

`-c` creates a new tarfile
`-f file` specifies the name of the tarfile
`-v` verbose mode

- Example:

  `tar -cvf /tmp/tarfile.tar /usr/bin`

  Archives in `tarfile.tar` all the files (including the directories) of the file system sub-tree rooted at `/usr/bin`

`tar` : Extract options

- `-x` extracts files (keeping the sub-directory structure) archived in the tarfile
- `-t` type the list of files in the tarfile
- `-f file` specifies the tarfile name
- `-v` verbose mode

- Examples:

```
tar -tvf /tmp/tarfile.tar .
tar -xvf /tmp/tarfile.tar
```


`gzip` and `gunzip`

```
gzip file
```

Compress a file appending the suffix extension `.gz`

```
gunzip file.gz
```

Decompress `file.gz`

Filters Programs

- A **filter** is a program that receives its input data from `stdin` and produces its results on `stdout`
- Filters are a typical paradigm of Unix programming because they allow to
  - easily redirect `stdin` and `stdout` to a file
  - build applications by concatenating simple programs that redirect their `stdin` and `stdout` to `pipes`
- Examples:
  - `more, less` **pagers**
  - `head, tail` **selectors**

Sorting

- `sort [-options] [file ...]`
- Options:
  - `-n` numeric sorting, not alphabetic
  - `-r` reverse sorting
  - `-f` ignores upper/lower case difference

Sorting

`sort [-options] [file ...]`

Options:

- `-k` field selector
- `-t`*`char`* field separator character

Example:

`sort –k2.2,2.3 filename`

Sort `filename` using as key the second and third character of the second field of each line

`sed` - Stream text EDitor

`sed` is a filter that is also able to edit the content of file according to commands specified in a script

`sed [-n] script [files]`

`sed [-n][-e script][-f script_file][files]`

filters `stdin` according to the commands in `script` and `script_file`

`-n` does not repeat `stdin` on `stdout`; prints only what is required by the `script` but does not repeat on `stdout` the rest of the file lines

sed - Stream text EDitor

Script syntax

`[address[,address]] function [args]`

- `address`: line number or regular expression
- `function`: command executed on matching line
- `args`: function arguments

sed Functions

`p` prints current line

`d` delete current line

`q` quit `sed`

`s/p1/p2/`

`y/orig/subs/`

replaces the characters in `orig` by those in `subs`

sed Functions

`s/regexp/replace/flags`

replaces patterns matching `regexp` by `replace`

`flags:`

- `num` replaces `num` occurrences only
- `g` replaces all the line occurrences
- `p` prints the line if a substitution occurred in that line

sed Examples

`sed '1,3 d' filename`

prints `filename` deleting rows 1 to 3

`sed '3,$ d' filename`

prints `filename` deleting rows 3 to the EOF

`sed -n '/^foo/ p' filename`

prints only the lines of `filename` beginning by `foo` (`-n` inhibits printing the not matching lines)

`sed -f sedfile filename`

`sedfile` is the command script

sed: examples

```
1,1  {
     s/^/Begin:/
     s/$/ -- End/
     }
/\/\*.*\*\// d
```

`sed -f sedfile filename`

sed: esempi

```
1,1  {
     s/^/Begin:/
     s/$/ -- End/
     }
/\/\*.*\*\// d
```

- `sed -f sedfile filename`

```
One Two Three
A B C
/* Comment */
```

sed Example

```
1,1  {   [for the first row only]
     s/^/Begin:/
     s/$/ -- End/
     }
/\/\*.*\*\// d    [delete comments]
```

```
sed -f sedfile filename
```

```
One Two Three
A B C
/* Comment */
```

```
Begin: One Two Three -- End
A B C
```


awk

- awk proposed in 1977 by
  - A. V. Aho
  - P. J. Weinberger
  - B. W. Kernighan
- awk is a text processing language based on pattern matching

Introduction

- Searches each record of the argument file that matches a given pattern
  - No need for `fopen`, or `fgets` cycles
- For each matching pattern an action is executed
- The `awk` syntax is similar to the syntax of the C language

Introduction

- Smart **grep**
  - All the functionality of **grep** with added processing abilities
- File conversion
  - Quickly write format converters for text files
- Spreadsheet
  - Easy use of columns and rows

Input file

- The input text file is described as a sequence of **records**
  - default record separator is **\n**
- A record is composed by a sequence of **fields**
  - default field separator is **" "**

Predefined variables

RS	record separator
FS	field separator
$0	the whole record
$1	the first field of a record
$n	the n-th field of a record
NR	current record number
NF	current number of fields
FILENAME	the name of the currently processed file

Running awk

- Command line: `awk 'command'`
  - `gawk '(pattern){action}' file`
  - `gawk '(pattern){action}' < file`
  - `cat file | gawk '(pattern) action}'`
- Using a script file: `awk -f scriptFile.awk file`

```
#!/usr/bin/gawk –f
# This is a comment
pattern {action}
. . .
. . .
```


Command structure

`pattern {action}`

`pattern` decides when the action can be executed

`action` is a sequence of instructions executed only in case of pattern matching

Example:

```
awk '/dollar/{++n; print $3, n}' filename
```

prints the third word of lines in `filename` that contains an occurrence of string `dollar` followed by the count of matching lines

Programming with awk

- Programming is done by building a list of rules
- The rules are applied sequentially to each record in the input file or stream
- The rules have two parts, a `pattern` and an `action`
- If the input record matches the `pattern`, then the `action` is applied

```
pattern1 { action }
pattern2 { action }
```

Command Structure

- The match is TRUE if there is no pattern
  - the action is executed for each record
- `awk '{print $1}' inputFile`
  - Prints the first field of each line in `inputFile`

Pattern Format

- Patterns can be:
  - Empty
  - Regular expressions
    `/ expr /`
  - Logical expressions
    `($1 == "Ciao" && $2 == "Bye")`
  - Range operators
    `( pattern1, pattern2 )`
  - Special
    - `BEGIN`
    - `END`

Regular Expressions

- \ escape character
- ^ beginning of line
- $ end of line
- . a single character
- [abc] one of a, b and c
- [a-z] character a to z
- [^abc] a character excluding a, b,and c

Regular Expressions

- `one|two` matches `one` or `two`
- `*` 0 or more occurrences of the previous symbol
- `+` 1 or more occurrences of the previous symbol

Regular Expressions: Examples

```
/^((may)|(MAY)|(May))$/ { print ”Maggio”}

/^[Tt]itle.*/ { print ”\nNew title.” }
```


Logical and Comparison Operators

- `!` , `&&` , `||` , `==` , `< >` , `<=` , `>=` , `!=`
  - The usual C operators
- `~` matches a regular expression
- `!~` does not match a regular expression

Comparison Operators: Examples

- `$1 == "Bob" { print "Bob stuff"}`
- `($1 !~ /[Mm]aggio/) { print "Is not May"}`
- `(($1 == "Bob") && ($2 ~ /[mM]*xy?RR$/) { print "The first field is Bob"; print "The second field is", $2; }`

Range Operators

- **`cond1, cond2`**
  - The patten is true when a record matches **`cond1`**, and remains true until a record matched **`cond2`**

```
/1/, /CIAO/ { print $2}
```

```
one
1 number one
2 number two
CIAO bye
3 number three
```

→

```
number
number
bye
```


Range operators

- $1=="1", $1=="CIAO" { print $2; }

```
try ciao
1 number uno
2 number due
CIAO End
3 number tre
```

→

```
number
number
End
```

Predefined Patterns

- **BEGIN** the actions in **{ }** are executed before processing the input file
- **END** the actions in **{ }** are executed after processing the input file

Examples

```
BEGIN { FS=":"}
$1 ~ /[0-9]/ { print $3 }
```

```
one
1:number:one
2:number:two
CIAO:bye
3:number:three
```

→

```
one
two
three
```


Examples

```
BEGIN {
min=2000000;
}

$1 > max {max = $1}
$1 < min {min = $1}

END{
print max, min;
}
```

Actions

- Variables
- Strings
- Arrays
- Operators
- Conditional flow
- Loops
- Input from multiple files
- Functions
- Interaction with shell

Variables

- Variables are:
  - Predefined
    - Ex. `NF`
  - Fields of a record prefixed by `$`
    - `$0` , `$i` , `$(i+1)`, `$NF`
  - User defined
    - No need for declaration
    - Implicitly set to 0 and Empty String
    - i = -1
    - string = "main"
    - s1 = "main program"

Variables

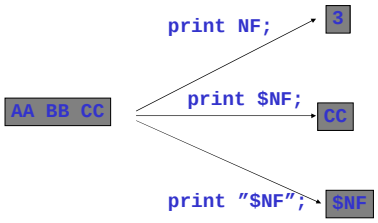
- There is only one type in `awk`
  - Combination of a floating-point and string
  - The variable is converted as needed
    - Based on it's use
  - No matter what is in `x` you can always do
    - `x++`
    - `length(x)`
- A variable can be set externally from command line

```
awk –v var=value –v var1=value1 ...
```

Predefined Variables

- **RS** record separator
- **FS** field separator
- **$0** the whole record
- **$1** the first field of a record
- **$n** the n-th field of a record
- **NR** current record number
- **NF** current number of fields
- **FILENAME** the name of the currently processed file
- **ENVIRON** array of the environment variables Ex. **ENVIRON["PATH"]**

Examples



String Operators

- Appending strings is easy:

```
BEGIN{
s1 = "Pietro";
s2 = "Laface";
s3 = s1 s2;
print s3;        # prints PietroLaface
print s1 s2;     # prints PietroLaface
print s1, s2;    # prints Pietro Laface
print s1 " " s2; # prints Pietro Laface
}
```

String functions

- `n_of_subs = gsub (reg, string, target)`
  - For each substring matching the regular expression `reg` in the string `target`, substitute `string`, and return the number of substitutions. If `target` is not supplied, use `$0`.
- `sub (reg, string, target)`
  - Like `gsub` but replaces only the first matching substring
- `length (s)`
  - Return the length of string `s`
- `position = match (string, reg)`
  - Return the position in `string` of the first occurrence of a substring matching the regular expression `reg; 0` if `reg` is not present

String functions

- `printf (s, …), sprintf (s, …)`
  - Like in C – parentheses can be omitted
- `n_of_fields = split (string, vec, delim`)
  - Splits the string `string` into the array `vec` on the basis of the regular expression `delim`, and returns the number of fields.
- `s = substr (string, position, len)`
  - Returns an at most `len`-character substring of `string` starting at `position`.
- `tolower (s), toupper (s)`
  - Like in C

Arrays

- Arrays in are associative
  - Implemented through a hash table
- Arrays can be sparse, they automatically resize, auto-initialize, and are fast (unless they get huge)

Arrays

- An array can, thus, be indexed by a string:
  - `days["january"] = 31;`
  - `days["february"] = 28;`
- An number is a special case of a string interpreted by `awk`
  - `v[5]`
  - `i=3; v[i]= "CIAO"`
  - `matrix[1,2,3] = "yes";`
    - `# matrix` is an associative array, `"1,2,3"` is a string, but we see `matrix` as a three-dimensional matrix

Arrays

- The arrays in `awk` can be used to implement almost any data structure
  - Set:

```
myset["a"]=1; myset["b"]=1;
if ("b" in myset )
```

  - Multi-dimensional array:

```
myarray[1,3] = 2;
myarray[1,"happy"] = 3;
```

  - List:

```
mylist[1,"data"]=2;
mylist[1,"next"] = 3;
```


Arrays

- To check if there is an element in the array use keyword `in`:

```
myset["a"]=1; myset["b"]=1;
if ("b" in myset )
```

returns TRUE

- `delete myset["b"]`

remove the entry `"b"` from associative array `myset`

Operators and Built-in Functions

- `x+y, x-y, x*y, x/y`
- `x^y, x%y`
- `++x, x++, --x, x—`
- `int(x) truncates to integer`

- `sqrt(x), sin(x), cos(x), log(x), exp(x)` etc.
- `rand( )` (random number between 0 and 1)
- `srand(expr)` (seed initializer)
- `systime( )` (number di seconds from january 1st 1970)

Functions

- Functions were not part of the original spec
  - Rule variables are global
  - Function variables are local

```
function MyFunc(a,b,c) {
   return a+b+c;
}
```


Conditional Flow

- ```
  if (cond) {
      true block of instructions
   } else {
       false block of instructions
   }
  ```
- `condition ? True_instr : False_instr;`
  ```

Loop Constructs

- `while (condition) { . . .}`
- `do {. . .} while (condition)`
- `for (init; cond; op) {. . .}`
- `for (i in array) {block of instructions}`

Reading Multiple Lines

- `exit`
  - Stop file processing
    - if pattern `END` exists, it is executed
- `getline`
  - reads next record filling the fields `$0, $1,…, $NF`
  - returns 0 if EOF

Interaction with the shell

- Redirection

```
getline  < ”filename”

print ”Ciao” > ”nomeFile.txt”;
print myVar | ”more”;

k = 3
filename = ”file-” k;
print $3 >> filename;
```

- `system (shellCommand);`
  - Forks a shell, that executes `shellCommand`
  - Ex. `system(”ls –la”)`

Example

The file **Student_ID** contains a sequence of line

<First_Name > <Last_Name> : <ID>

Ex.

```
John White:12345
Peter Green:12367
. . . . . : . .
. . . . . : . .
```


Example

The file **Student_ID** contains a sequence of line

<First_Name > <Last_Name> : <ID>

Ex.

```
John White:12345
Peter Green:12367
. . . . . .
```


Example

Another file **Scores** has the following format:

<First_Name > <Last_Name> : <Exam> : <Score>

```
John White:Operating Systems:27
Peter Green:Information Theory:28
John White:Mobile Networks:24
. . . . . .
. . . . . .
```

Example

Implement an **awk** script **publish_scores.awk** that reads the content of the two files **Student_ID** and **Scores**, and produces a set of files – one per Exam -

Id_scores-<Exam> with format:

<ID> <Score>

Example

```
#!/bin/gawk –f

BEGIN {
system(”rm –f Id_scores-*”); # delete old files
FS = ":";                    # set field separator
while (getline < "Student_ID" ){
  student_id[$1] = $2;
}
}
{
sub(" ", "_", $2);          #replace blank
filename = "Id_scores-"$2; #creates filename
print student_id[$1], $3 >> filename
}
```


Example

The script **publish_scores.awk** is run with the following command line:

```
awk –f publish_scores.awk Scores
```

and produces the files

Id_scores-Information_Theory

Id_scores-Mobile_Networks

Id_scores-Operating_Systems