# La shell di Unix

## *Uso interattivo e scripting*

Lucidi per il corso di Laboratorio di Sistemi Operativi tenuto da Paolo Baldan presso l'Università Ca' Foscari di Venezia, anno accademico 2004/2005.

# Manipolazione di testi

# Linguaggi formali

- Dato un alfabeto $A = \{a_1, \ldots, a_n\}$, l'insieme delle stringhe (finite) sull'alfabeto $A$ è indicato con $A^*$, ovvero

$$A^* = \{s_0 \ldots s_k : s_i \in A\}.$$

  Un linguaggio $L$ sull'alfabeto $A$ è un insieme di stringhe su $A$, ovvero $L \subseteq A^*$.

- Operazioni sui linguaggi
  - Unione: $L + L' = \{\alpha : \alpha \in L \ \vee \ \alpha \in L'\}$
  - Concatenazione: $L \cdot L' = \{\alpha\alpha' : \alpha \in L \ \wedge \ \alpha' \in L'\}$
  - Potenza: $L^n$ definito induttivamente

$$L^0 = \{\epsilon\} \qquad L^{n+1} = L \cdot L^n$$

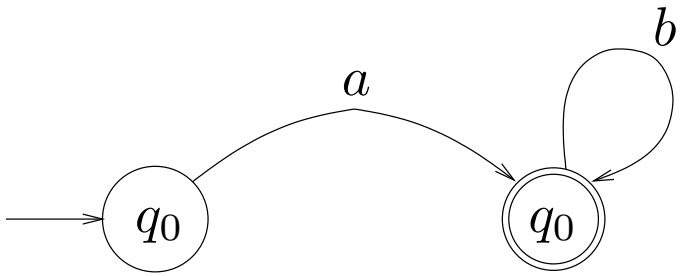
  - Stella di Kleene: $L^* = \bigcup_{n \in \mathbb{N}} L^n$

# Linguaggi Regolari

- Classificazione dei linguaggi (Noam Chomsky)

| Linguaggi | Grammatiche $[\alpha \rightarrow \beta]$ | Automi |
|---|---|---|
| tipo 0 | generali | automi di Turing |
| tipo 1 | monotone / context sensitive | automi limitati linearm. |
| tipo 2 | context-free | automi a pila |
| tipo 3 | regolari | automi a stati finiti |

- I linguaggi regolari sono molto semplici da trattare (ad es. il problema dell'appartenenza al linguaggio ha complessità lineare).




$$L = \{ab^n : n \geq 0\}$$

# Espressioni regolari

Teorema (Kleene) Un linguaggio $L$ sull'alfabeto $A$ è regolare

se e solo se

è ottenibile dai linguaggi $\emptyset$ e $\{a\}$ con $a \in A$, utilizzando gli operatori di unione $+$, concatenazione $\cdot$ e stella $(_)^*$.

Espressioni Regolari su di un alfabeto $A = \{a_0, \ldots, a_n\}$ sono

$$re := \emptyset \mid \mathbf{a_0} \mid \ldots \mid \mathbf{a_n} \mid re + re \mid re \cdot re \mid re^*$$

A ciascuna espressione regolare $re$ è associato un linguaggio $\mathcal{L}(re)$:

$$\mathcal{L}(\emptyset) = \emptyset \qquad \mathcal{L}(\mathbf{a_i}) = \{a_i\} \qquad \mathcal{L}(re_1 \cdot re_2) = \mathcal{L}(re_1) \cdot \mathcal{L}(re_2)$$

$$\mathcal{L}(re_1 + re_2) = \mathcal{L}(re_1) + \mathcal{L}(re_2) \qquad \mathcal{L}(re^*) = \mathcal{L}(re)^*$$

Per il teorema di Kleene, ogni linguaggio regolare è denotato da un'espressione regolare e viceversa.

# Espressioni Regolari in Unix

- Patterns (modelli) che individuano insiemi di stringhe (linguaggi) e rappresentano uno strumento piuttosto potente e sofisticato di UNIX.

- Ispirate alle espressioni regolari della teoria dei linguaggi formali, ma risultano molto più potenti.

- Utilizzate da vari programmi di elaborazione di testi (ad es. `grep`, `sed`, `awk`, `vi`, `emacs`) per:
  - elencare/eliminare righe che contengono un'espressione regolare
  - sostituire un'espressione regolare (~ find/replace)
  - ...

# Espressioni Regolari in Unix/A

- insiemi di caratteri: pattern elementari che specificano la presenza un carattere appartenente ad un certo insieme.
- ancore: legano il pattern a comparire una posizione specifica della riga (es. inizio, fine).
- gruppi: “racchiudono” l’espressione regolare che può quindi essere riferita come una singola entità.
- modificatori: specificano ripetizioni dell’espressione che precede il modificatore stesso.

# Espressioni Regolari: Sintassi

```
[abc]        match any of the characters enclosed
[a-d]        match any character in the enclosed range
[^set]       match any character not in the following set

 .           match any single character except <newline>

[:alpha:]    some predefined character sets
[:digit:]

\            treat the next char literally. Normally used to
             escape special characters such as "." and "*"
```

Esempi

```
[^[:alpha:]]  any non alphabetic char (same as [^a-zA-Z])
[^a-c]        any char except 'a', 'b', 'c'
[\.a]         chars 'z' and '.'
```

# Espressioni Regolari: Sintassi / A

```
exp1 | exp2  match either exp1 or exp2

exp1 exp2    match exp1 followed by exp2

^exp         exp must start at the beginning of the line
exp$         exp must end at the end of the line
```

Esempi:

```
a|the        either the string 'a' or the string 'the'
cat          the string 'cat'
.at          any char followed by 'at', such as 'cat', 'rat',
             'mat', 'bat', 'fat', 'hat'
^cat         'cat' at the beginning of the line
cat$         'cat' at the end of the line
[cC]at       'cat' or 'Cat'
[0-9]$       any digit ending a line
```

# Espressioni regolari: Sintassi / A

```
exp{n}        match the regular expression exp exactly n times
exp{n,m}      match exp a minimum of n times and a maximum
              of m times
exp{m,}       match exp a minimum of n times
              (0 <= n,m <= 255)

exp*          match exp zero or more times
exp+          match exp one or more times
exp?          match exp zero or one times
```

Esempi:

```
xy*z          any occurrence of an 'x', followed by zero or
              more 'y's, followed by a 'z'
xy{3,5}z      an 'x', followed by 3 to 5 'y' and then 'z'
[A-Z][A-Z]*   one or more upper case letters (same as [A-Z]+)
[A-Z]*        zero or more upper case letters.
```

# Espressioni regolari: Sintassi / B

```
\<exp\>  match exp as long as it is a separate word

(exp)    saves the matching exp in a buffer, for future reference.
         Up to nine patterns can be saved for each line.

\n       where n is a digit (1-9). Matches the nth expression
         previously saved for this line.
         Expressions are numbered  starting from the left.
```

Esempi:

```
\<cat\>  'cat' as a separate word (e.g. 'concatenate' not ok)

(.*)\1   strings of the kind s s, where s is any string
         (not regular nor context-free)
```

Questa è la sintassi per le espressioni regolari estese accettata da `egrep`, `sed`, `awk`, ma non da grep (vedere `man` per ulteriori informazioni).

# Cercare espressioni regolari

Unix fornisce i comandi `grep` ed `egrep` (acronimo per [extended] generalized regular expression parser) per cercare espressioni regolari (estese) nei file.

```
egrep [-ivnwx] <regexpr> [<filename1> ...  <filenamen>]
```

Il modo più semplice di usarlo è:

```
egrep <regexpr> <filename>
```

Esempio:

```
/home/rossi$ cat ugolino
La bocca sollevo' dal fiero pasto
quel peccator, forbendola a'capelli
del capo ch'elli avea di retro guasto.

/home/rossi$ egrep pasto ugolino
La bocca sollevo' dal fiero pasto
```

# Cercare espressioni regolari / A

```
/home/rossi$ egrep '(p|gu)asto' ugolino
La bocca sollevo' dal fiero pasto
del capo ch'elli avea di retro guasto.
```

```
/home/rossi$ egrep '\<d.l\>' ugolino
La bocca sollevo' dal fiero pasto
del capo ch'elli avea di retro guasto.
```

Si osservi che non compare la riga

```
quel peccator, forbendola a'capelli
```

```
/home/rossi$ egrep '^.a|li$' ugolino
La bocca sollevo' dal fiero pasto
quel peccator, forbendola a'capelli
```

# Cercare espressioni regolari / B

- Alcune opzioni utili di `egrep`
  - `-i` non distingue tra maiuscolo e minuscolo
  - `-n` mostra anche il numero di linea
  - `-v` mostra le linee che non contengono l'espressione
  - `-w` cerca parole intere
  - `-x` cerca linee intere
- Se non si specifica `<filename>`, `egrep` esamina lo standard input.
- Una versione semplificata di `egrep` è `fgrep`

```
fgrep [-ivn] <text> <filename>
```

è utilizzato per cercare stringhe in file di tipo testo. Le opzioni `-i`, `-n` e `-v` funzionano come per `egrep`.

# Sostituire caratteri

Il comando `tr` converte un testo, carattere per carattere, secondo una corrispondenza specificata.

```
tr '<sourceCh>' '<destCh>' < <file>
```

Esempio:

```
/home/rossi$ tr 'aeiuo' 'eiuoa' < ugolino
Le bacce salliva' del fuira pesta
qoil piccetar, farbindale e'cepillu ...
```

trasforma `a` → `e`, ..., `o` → `a`.

Se `'<destCh>'` è più corto di `'<sourceCh>'`, si usa l'ultimo carattere in `'<destCh>'`.

Opzioni

- `-c` trasforma i caratteri non in `'<sourceCh>'` (complement)
- `-s` elimina caratteri uguali vicini (squeeze)
- `-d` cancella i caratteri in `'<sourceCh>'` (delete)

# Sostituire caratteri / A

Esempi:

- trasforma tutti i caratteri maiuscoli in minuscoli
  `tr 'A-Z' 'a-z' < filename`
- semplice encryption (`a` $\longrightarrow$ `b`, `b` $\longrightarrow$ `c` ..., `z` $\longrightarrow$ `a`)
  `tr 'a-zA-Z' 'b-zA-Za' < filename`
- rimuove tutte le cifre dall'input
  `tr -d '0-9' < filename`
- scompone l'input in parole, sistemandole una per riga
  `tr -cs 'a-zA-Z0-9' '\n' < filename`

# sed - editor di stream

Il comando `sed` permette di scandire uno o più file, modificando le linee che soddisfano una particolare condizione (nello stile dell'editor `vi`).

```
sed -e '<edit commands>' <filename>
sed -f <edit command file> <filename>
```

Nel primo caso le azioni da eseguire sono inserite nella linea di comando, nel secondo caso in un file, ed hanno forma `<selettore><comando>`.

Alcuni comandi di Editing

| | |
|---|---|
| `p` | stampa la linea corrente |
| `d` | cancella la linea corrente |
| `s/RegExp/Repl/` | sostituisce la prima occorrenza di `RegExp` con `Repl` |
| `s/RegExp/Repl/g` | sostituisce ogni occorrenza di `RegExp` con `Repl` |

# sed - editor di stream / A

I comandi di editing possono essere qualificati con un pattern che seleziona le linee alle quali tali comandi devono essere applicati.

Alcuni selettori di Linea

| | |
|---|---|
| `n` | linea numero `n` |
| `n,m` | le linee dal numero `n` ad `m` |
| `/RegExp/` | le linee che corrispondono a `RegExp` |
| `/RegExp1/,/RegExp2/` | tutte le linee tra ogni occorrenza di `RegExp1` e la successiva occorrenza di `RegExp2` |

Un'opzione interessante è

```
-n
```

sopprime ogni linea non esplicitamente selezionata (si ricordi che, altrimenti, `sed` copia in output ogni linea, possibilmente modificata).

# sed - editor di stream / B

- Rimuove tutte le linee che contengono la stringa `guasto`
  `sed -e '/guasto/d' ugolino`

- Stampa solo le prime dieci linee di un file (simile a `head`)
  `sed -n -e '1,10p' ugolino`

- Mostra il file delle password, fornendo solo le linee da `root` a `nobody`, ovvero gli account di sistema.
  `sed -n -e '/^root/,/^nobody/p' /etc/passwd`

- Aggiunge due spazi all'inizio di ogni riga.
  `sed -e 's/^/ /' filename`

- Elimina l'indentazione in un file, ovvero elimina tutti gli spazi ad inizio riga.
  `sed -e 's/^ *//' filename`

# sed - Esercizi

Scrivere un comando `sed` che

- visualizzi un file, cancellando le linee da 3 a 5;
- visualizzi solo la parte di un file che inizia alla prima linea che contiene “start” e finisce alla prima linea che contiene “stop”;
- visualizzi solo le linee non vuote di un file (cioè le linee che contengono almeno un carattere);
- visualizzi solo le linee non vuote e non costituite di soli spazi;
- dato un file lo visualizzi sostituendo ogni linea contenente la parola `bomba` con `OMISSIS`;
- dato un file con riferimenti bibliografici del tipo “si veda [Kant12]”, visualizzi solo i riferimenti, uno per riga.

# Manipolare testi con awk

- `awk` è un utility per la manipolazione di testi, il cui nome deriva dalle iniziali degli autori (Aho, Weinberger, Kerningham)

- La sintassi, nelle forme più semplici, è la seguente:

  ```
  awk '<programma>' <file>
  awk -f <progfile> <file>
  ```

  scorre un file (o più) ed esegue alcune azioni sulle linee che soddisfano una data condizione, concordemente con quanto specificato nel programma.

- La sintassi dei programmi è simile a quella del linguaggio C.

- Si tratta di un'utility estremamente flessibile e potente. Qui ci limiteremo a vedere solo alcuni esempi (→ Perl).

# Manipolare testi con `awk` - campi

- Ogni linea di testo letta viene scomposta in campi (i separatori di campo di default sono `<space>` e `<tab>`).

- Il primo campo di una linea può essere riferito con `$1`, il secondo con `$2`, .... L'intera linea è riferita da `$0`. Il numero di campi è riferito dalla variabile predefinita `NF`.
- Ad esempio

```
awk '{print NF,$0}' pippo
```

  mostra il file `pippo` sullo schermo premettendo ad ogni linea il numero `NF` di campi (parole) che la compongono.

# Manipolare testi con `awk` - script

- Un programma `awk` è costituito da una o più linee della forma

```
[ <condizione> ]   { <azione> }
```

- `<condizione>` può essere
  - un'espressione che coinvolge operatori logici e relazionali, oppure espressioni regolari.
  - uno dei token speciali `BEGIN` e `END`, che risultano attivati all'inizio del file (prima della prima linea) ed alla fine del file (dopo l'ultima linea)
- `<azione>` è un insieme di istruzioni C-like.

# Manipolare testi con awk - script / A

L'elenco di istruzioni C-like può contenere, ad esempio:

- `if (conditional) statement [ else statement]`
- `while (conditional) statement`
- `for (expr; conditional; expr) statement`
  `for (var in array) statement`
- `break`
- `continue`
- `variable = expression`
- `print list-of-expressions`
- `printf format list-of-expressions`
- `statement; statement`
- `{ statement }`

# Manipolare testi con awk - script / B

Il programma `awk` viene eseguito secondo i passi seguenti:

- il testo viene analizzato linea per linea e per ogni regola del tipo `[ <condizione> ] { <azione> }`
  - si valuta `<condizione>` e se questa è vera viene eseguita `<azione>`
  - se `<condizione>` è assente allora `<azione>` viene eseguita in ogni caso.
  - se `<azione>` è assente allora la riga corrente è stampata immutata.

# Manipolare testi con awk - script / C

Si supponga che il file `wordcount.awk` contenga

```
BEGIN { print ‘‘Inizio ad elaborare il file ...’’ }
{ for (i = 1; i <= NF; i++)
        num[$i]++
}
END { for (word in num)
            print word, num[word];
        print ‘‘Fine del file:’’, FILENAME
    }
```

allora il comando

```
awk -f wordcount.awk file
```

restituisce la lista delle parole che compaiono nel file `file` e la loro frequenza (`word in num` permette di scorrere gli indici di un array).

# Manipolare testi con awk - variabili

- `awk` supporta variabili definite dall'utente;
- le variabili non devono essere dichiarate (tipico dei linguaggi di scripting);
- inizialmente ogni variabile ha valore null o 0

```
File: lfcount.awk
BEGIN { print ``Inizio ad elaborare il file ...'' }
NF == 5 {
   printf ``Linea %d: %s \n'', NR, $0;
   lineCount++;
   wordCount += NF;
}
END { printf ``Linee: %d, Campi: %d'', lineCount, wordCount) }
```

Seleziona le linee che contengono 5 campi, le visualizza con il numero di riga, ne conta il numero ed il numero dei campi e lo visualizza.

# Manipolare testi con awk - regexp

La condizione può essere

- un'espressione regolare (estesa):

  ```
  /RegExp/
  ```

  valida per ogni riga che corrisponda all'espressione;

- una coppia di espressioni regolari

  ```
  /RegExp1/, /RegExp2/
  ```

  in questo caso l'azione `<action>` si applica ad ogni linea tra la prima che corrisponde a `/RegExp1/` e la linea successiva a questa che corrisponde a `/RegExp2/`.

- Esempio:

  ```
  awk '/start/, /stop/ { printf ''line %d: %s'', NR, $0}'
      filename
  ```

# awk - pattern matching

Potenti funzioni di pattern matching su espressioni regolari:

- Condizioni: `str ~exp` oppure `str !~exp`
  Esempi:
  - `''abbc''  /a[b]+c/`: ok
  - `''ac''  /a[b]+c/`: no
- Operatori su stringhe:
  - `gensub(exp, str, h [, target])`:
    Sostituisce la prima/tutte le occorrenze di `exp` in `target` con `str` (a seconda del valore di `t`). Quando `target` è assente usa `$0`. La stringa `str` può contenere `\n`.
  - `split(str, array [, exp])`:
    Suddivide la stringa `str` e mette i frammenti in un array. Usa come separatore `exp` (default `FS`)

# Manipolare testi con `awk` - esercizi

- Data un agenda con righe:

  <cognome, nome, indirizzo, città, numero>

  scrivere un programma `awk` per stampare le righe con nome uguale a Giovanni e città uguale a Milano.

- Scrivere un programma `awk` che stampi le righe di un file dall'ultima alla prima.

- scrivere un programma `awk` per sostituire in un testo la parola `sempre` con la parola `mai` e contare le sostituzioni effettuate.

- Scrivere un programma `awk` che spezzi una riga subito dopo il 20mo carattere, appenda uno  come avvertimento e scriva a capo il resto della riga, giustificato a destra (la funzione `substr(str,i,j)` fornisce la sottostringa di `str` dall'indice `i` all'indice `j`).

# Eliminare doppioni

L'utility `uniq` permette di modificare un file, sostituendo righe identiche adiacenti con un'unica occorrenza della riga ripetuta.

```
uniq [-c] <infile> <outfile>
```

Con l'opzione `-c` ogni riga è preceduta dal numero di occorrenze.

```
/home/rossi$ cat uniqfile
  questa e' una riga doppia
  questa e' una riga doppia
  questa e' una riga singola
  questa e' una riga tripla
  questa e' una riga tripla
  questa e' una riga tripla

/home/rossi$ uniq -c uniqfile
  2 questa e' una riga doppia
  1 questa e' una riga singola
  3 questa e' una riga tripla
```

# Confrontare due file

Due utility permettono di confrontare file:

- `cmp <file1> <file2>`
  Verifica l'uguaglianza di `<file1>` e `<file2>` e
  - nel caso siano diversi fornisce la riga e l'offset (numero di byte dall'inizio del file) del primo mismatch
  - se uno è un prefisso dell'altro si ottiene un messaggio di EOF sul file più corto.

- `diff <file1> <file2>`
  Simile a `cmp`, ma fornisce una lista di differenze tra i due file (cambiamenti necessari per convertire il primo file nel secondo).

# Un sommario dei comandi visti

```
awk      - search for and process patterns in a file,
cat      - display, or join, files
cd       - change working directory
chgrp    - change the group associated to a file
chmod    - change the access mode of a file
cmp      - compare files
chown    - change the owner of a file
compress - compress files
cp       - copy files
date     - view/set the date of the system
diff     - show differences between files
file     - display file classification
find     - find files
grep     - search for a pattern in files (egrep, fgrep)
gzip     - compress files
head     - display the first few lines of a file
lp       - print files (System V)
lpr      - print files (Berkeley)
```

# Un sommario dei comandi visti / A

```
lprm    - remove a  job from the print queue
ls      - list information about files
mkdir   - create a directory
more    - display a file one screen at a time (System V)
mv      - move and/or rename a file
pwd     - print the working directory
rm      - remove (delete) files
rmdir   - remove (delete) a directory
sed     - stream editor
sort    - sort and/or merge files
tail    - display the last few lines of a file
tar     - store or retrieve files from an archive file
tr      - transliterate charaters
umask   - set file creation permissions
uniq    - remove duplicates
wc      - counts lines, words and characters in a file
whereis - show where executable is located in path
which   - locate an executable program using "PATH"
```

# Altri comandi

```
df      - display the amount of available disk space
du      - display information on disk usage
fsck    - check and repair a file system
groups  - show user groups
ln      - create a link to a file
spell   - check a file for spelling errors
split   - split files
cut     - cut columns from a file
paste   - join coulumns in a single file
whatis  - list man page entries for a command
```