Unix per Linguisti

Marco Baroni

7 ottobre 2004

1 Preambolo: interfaccia grafici e righe di comando

1.1 Interfaccia grafici

- Un *sistema operativo*,[1] tra l'altro, offre all'utente degli interfaccia[2] per interagire con il computer (leggere un documento, rimuovere una directory, ecc.)
- Il tipo di interfaccia di gran lunga più popolare si basa su "metafore visive" e sulla simulazione grafica di varie operazioni:
  - Apriamo una "cartella" facendo "clic" o doppio clic;
  - Spostiamo un documento sul "desktop";
  - "Trasciniamo" un file da una "cartella" al "cestino";
  - Apriamo un programma in una nuova "finestra";
  - E via dicendo...
- Gli interfaccia grafici sono molto facili ed intuitivi da usare, e sono certamente responsabili, almeno in parte, dell'enorme diffusione dei computer negli ultimi vent'anni.
- Tuttavia, ci sono certe operazioni per le quali gli interfaccia grafici non sono molto adatti, e gran parte delle cose utili che un linguista può fare con un computer rientra in questa categoria.

[1] Windows 2000, Windows XP, Mac Os 9, Mac Os X, le varie distribuzioni Linux, i vari BSD, Solaris sono esempi di sistemi operativi, nel senso molto lato in cui uso questo termine qui.

[2] Nel mio italiano, la parola *interfaccia* è maschile e rimane immutata al plurale.

- Caso tipico: c'è una cartella che contiene 300 files, e vorremmo calcolare la frequenza complessiva con cui una lista di parole capita in questi files: provate a farlo con un interfaccia grafico!

1.2 Righe di comando

- Quasi tutti i sistemi operativi offrono un interfaccia alternativo tramite la riga di comando (command line).
- Si interagisce con il computer tramite istruzioni che vengono battute al "prompt"; il computer a sua volta comunica attraverso messaggi scritti mandati all'output (che oggi tipicamente si presenta come una console dentro una finestra).
- Nei sistemi operativi della Microsoft, esiste una riga di comando con capacità piuttosto rudimentali (MS-DOS prompt).
- Nei sistemi operativi della famiglia Unix (vedi sotto) l'interazione tramite riga di comando gioca un ruolo molto più importante, e permette un'amplissima gamma di operazioni che sarebbero impossibili o scomodissime con un interfaccia grafico.
- La command line come Weltanschauung: Neal Stephenson, "In The Beginning was the Command Line" (disponibile in rete: cercate con Google).

1.3 Nota Bene

- Non bisogna dimenticarsi che, sullo stesso computer, l'interfaccia grafico e l'interfaccia via riga di comando vi fanno interagire con gli stessi files.
- Per esempio, su un computer con sistema operativo Windows:
  - potete aprire una certa directory facendo doppio click, aprire e leggere un file di testo nella directory con l'applicazione *Bloc notes* (per esempio), poi spostarlo nel cestino e svuotare il cestino, oppure...
  - potete aprire la console MS-DOS, andare nella stessa directory con il comando `cd`, leggere il file con `more` e rimuoverlo con il comando `del`.

- Le operazioni compiute sono esattamente le stesse, e hanno lo stesso effetto sulla stessa directory e sullo stesso file!

2 Il setup per il corso di linguistica computazionale

- Architettura cliente/server:
  - Clienti Windows nella sala informatica.
  - Server centrale Linux (newton).
  - Ci connettiamo in remoto a newton tramite il programma Putty.
  - Una volta connessi, Putty ci fa interagire con newton attraverso la riga di comando Unix.
  - Il computer *locale* che stiamo usando ha sistema operativo Windows, ma dentro la console di Putty siamo all'interno del mondo Unix e stiamo lavorando con programmi e files che sono su computer remoto (usando le risorse di memoria del computer remoto).
  - Gli stessi files si possono manipolare attraverso interfaccia grafico (provate a visitare le vostre home utenti su newton con WinSCP).
- Non fare confusione tra architettura cliente/server e interfaccia grafico vs. command line: se i computer locali avessero Linux o Mac OS X, potremmo sperimentare tutte le combinazioni possibili (lavorare sia in remoto che in locale su command line, lavorare sia in remoto che in locale via interfaccia grafico, command line in remoto e interfaccia grafico in locale, command line in locale e interfaccia grafico in remoto).[3]
- Putty si può scaricare gratuitamente (cercatelo con Google o nella pagina dei miei links) e richiede pochissime risorse in termini di memoria e velocità di connessione.[4]
- Programmi analoghi permettono di connettersi a newton da computer con altri sistemi operativi (io ho un computer con Mac Os X e uno con Linux, e lavoro quotidianamente su newton da entrambi).

[3] Resta il fatto che la riga di comando, richiedendo meno risorse, è spesso l'interfaccia migliore per lavorare in remoto.

[4] Putty è anche disponibile nei computer della sala Heilmann, del piano ammezzato e del laboratorio di terminologia.

- Da casa, bisogna connettersi a `newton.sslmit.unibo.it` via *protocollo* `ssh`.
- Per copiare files da un computer locale a uno remoto (e vice versa), ci sono molti programmi disponibili in rete e spesso gratuiti, per riga di comando o grafici, basati sui protocolli `ftp` o `scp/ssh`, e per tutti i sistemi operativi (su Windows, consiglio il cliente gratuito WinSCP: cercate con Google o tra i miei links).[5]
- NB: Putty è semplicemente lo strumento che permette di connettersi al computer remoto; i programmi che useremo per manipolare dati *non* fanno parte di Putty, ma sono programmi installati nel computer remoto.
- Confondere Putty con tali programmi (come talvolta gli studenti fanno) equivale a confondere un telefono con la persona che si chiama, o un modem con il world wide web.

3 La famiglia di sistemi operativi Unix

- Una famiglia di sistemi operativi scritti da "power users" per "power users" (a Berkeley, ai Bell Labs, e altrove).
- Enfasi su potenza e flessibilità.
- Più di recente: user friendliness (interfaccia grafici).
- Alcuni membri della famiglia Unix:
  - Linux (molte distribuzioni: Debian, Red Hat, Mandrake, Suse, Slackware...)
  - Darwin (Mac OS X)
  - Solaris
  - FreeBSD, OpenBSD, NetBSD
- Unix is Not Linux (but Linux IS Unix)
- Unix su un pc con Windows:
  - Installare Linux

[5] Anche WinSCP è disponibile anche in sala Heilmann, nel piano ammezzato e in terminologia.

  - Connessione a server Unix via Putty o altro cliente ssh/telnet
  - Knoppix (e Knorpora!)
  - Cygwin

3.1 Perché imparare ad usare la riga di comando Unix

- "Steep learning curve", ma i benefici valgono decisamente lo sforzo iniziale.
- La linguistica computazionale richiede l'analisi di corpora e liste di parole o altre unità testuali.
- In concreto, un corpus (o una lista di parole) è un grosso file di testo...
- ...o un insieme di molti piccoli files di testo.
- Unix offre molte utilities "generiche" per manipolare ed analizzare files di testo, oltre a molti programmi (spesso liberi, ancor più spesso gratuiti) pensati per linguisti.[6]
- La command line permette la manipolazione efficiente e flessibile di dati testuali.
- In particolare:
  - Non serve far partire un pesante interfaccia grafico per manipolare i dati.
  - Ideale per batch (pre-)processing: "You can go a long way before actually reading the text" (Lou Burnard).
  - La filosofia del "box degli attrezzi":
    * Ciascun mini-programma (comando) fa una cosa sola, ma la fa bene.
    * I mini-programmi vengono combinati per compiere operazioni complesse in maniera flessibile.
    * Spesso un programma da command-line compie un'operazione (mettere una lista in ordine, mostrare le prime righe di un file, ecc.) che in altri sistemi sarebbe al massimo un'opzione nel menu di un applicativo complesso.

[6] Uso i termini utility e comando in maniera essenzialmente intercambiabile. Le utilities/comandi sono programmi, ma non tutti i programmi sono dei comandi (solo quelli che si possono invocare sulla riga di comando!)

- Do-it-yourself: l'utente si può costruire gli strumenti giusti per quello che deve fare.
- Il confine tra utente e programmatore nel mondo Unix non è netto: chiunque usa Unix finisce per costruirsi dei semplici tool per le proprie esigenze.
- Il comando della command line Unix permette al non-informatico di svolgere compiti che, su interfaccia grafici, potrebbero essere gestiti soltanto da informatici (o programmi sviluppati da informatici).
- "It is much better to solve the right problem naively than the wrong problem expertly" (Hamming, citato da Church in "Unix for Poets").
- Filosofia del codice sorgente libero (e spesso dei binari gratis): free as in freedom, e spesso free as in free beer.
- Una comunità estremamente amichevole.[7]

4 Istruzioni per la sopravvivenza e funzioni demiurgiche

- Prima di tutto, bisogna fare login (per es., via Putty, ma lo stesso varrebbe su sistema locale).
- Per fare login, scrivete il vostro username seguito da invio, e poi la password anch'essa seguita da invio.
- NB: mentre scrivete la password, il cursore rimane immobile. Non preoccupatevi, e date l'invio comunque.[8]
- Una volta fatto il login, vi trovate nella vostra home.[9]
- Vi si presenta (o, su alcuni sistemi, dovete far partire via interfaccia grafico) una finestra/terminale/console con un prompt. Per esempio:

[7] A Forlì, questa comunità si manifesta soprattutto nell'associazione FoLUG, e simili gruppi esistono un po' ovunque, oltre che in maniera virtuale in un gran numero di mailing lists internazionali.

[8] Si tratta di una misura di sicurezza (se sullo schermo si vedesse il cursore muoversi, e magari degli asterischi comparire ad ogni battuta, un agente segreto alle vostre spalle verrebbe a sapere da quanti caratteri è composta la vostra password...)

[9] Molte idee ed esempi presentati da qui in poi vengono da "Unix for Poets" di Kenneth Church.

```
[newton ~] compling2$
```

- Chi sono? Dove sono?[10]

```
[newton ~] compling2$ whoami
compling2
[newton ~] compling2$ pwd
/home/compling2
```

- Che cosa c'è in questa directory?

```
[newton ~] compling2$ ls
cl_shared_data
```

- Per ora, le vostre home directories contengono soltanto un collegamento a una directory comune (`cl_shared_data`) con vari dati che useremo nel corso (e che si trova nella mia home su newton).[11]

- Muoversi da una directory all'altra:

```
[newton ~] compling2$ cd cl_shared_data
[newton ~/cl_shared_data] compling2$ ls
brown.txt       genesis.txt  novelle
europarl_it_de  lob.txt      unsorted_repubblica_words.txt
[newton ~/cl_shared_data] compling2$ cd ..
```

- Per uscire:

```
[newton ~/cl_shared_data] compling2$ exit
```

- Ricordatevi che gli stessi dati che manipolate via riga di comando nella vostra home su newton possono anche essere gestiti tramite interfaccia grafico: provate a visitare la vostra home con WinSCP.

[10] Naturalmente, quando scrivete sulla command line ogni comando va seguito da un invio.

[11] Quasi di sicuro, quando leggerete queste note i contenuti della directory `cl_shared_data` saranno cambiati, rispetto a quelli che compaiono negli esempi in questo handout.

4.1 Cose che rendono la vita più facile

- `Ctrl + c` ferma qualsiasi programma in corso (quasi sempre...)
- Command/filename completion con la tab.
- Andare avanti e indietro nella command-line history con up-/down-arrows.
- File globbing:

```
[newton ~/cl_shared_data/novelle] compling2$ ls
01_scial.txt  05_la_mo.txt  09_donna.txt  13_cande.txt  indice.txt
02_la_vi.txt  06_in_si.txt  10_il_ve.txt  14_berec.txt  info.txt
03_la_ra.txt  07_tutt_.txt  11_la_gi.txt  15_una_g.txt  manuzio.txt
04_l_uom.txt  08_dal_n.txt  12_il_vi.txt  16_appen.txt
[newton ~/cl_shared_data/novelle] compling2$ ls *_*
01_scial.txt  05_la_mo.txt  09_donna.txt  13_cande.txt
02_la_vi.txt  06_in_si.txt  10_il_ve.txt  14_berec.txt
03_la_ra.txt  07_tutt_.txt  11_la_gi.txt  15_una_g.txt
04_l_uom.txt  08_dal_n.txt  12_il_vi.txt  16_appen.txt
```

- `man` is your friend! (Avanti con *spazio*, indietro con *b*, fuori con *q*)
- Non c'è ragione di limitarsi ad un solo terminale: potete tranquillamente fare molteplici login (per es., facendo partire Putty varie volte), per compiere più operazioni in parallelo o monitorare l'andamento di un programma che sta girando su una console da un'altra console.
- Taglia-e-incolla con Putty: seleziona-e-right-mouse-click (su terminali Linux, di solito: seleziona-e-center-mouse-click).

4.2 Percorsi

- Percorsi relativi:[12]

```
[newton ~] compling2$ cd cl_shared_data/
```

- Percorsi assoluti (il primo / nel percorso sta per la directory "radice"):

```
[newton ~/cl_shared_data] compling2$ cd /home/compling2
```

[12] Ricordatevi la tab completion!

- Dalla home:

```
[newton ~] compling2$ cd ~/cl_shared_data/novelle/
```

- In su di una directory:

```
[newton ~/cl_shared_data/novelle] compling2$ cd ..
[newton ~/cl_shared_data] compling2$ cd \
../../compling2/cl_shared_data
```

- (Notate che \ permette di andare a capo sulla command line: questo non serve quasi mai in pratica, ma serve a me ai fini di far stare i comandi dentro i margini di questo documento!)

- Il modo più rapido per tornare a casa:

```
[newton ~/cl_shared_data] compling2$ cd
```

- Confondersi con i percorsi è, nella mia esperienza, di gran lunga l'errore più comune dei principianti: quando un comando non funziona, date subito un'occhiata ai percorsi che avete specificato...

4.3 Creare, spostare, rimuovere files e directories, e ispezionare il contenuto dei files

- Creare una directory, rimuovere una directory vuota:[13]

```
$ mkdir blah
$ rmdir blah
```

- Copiare un file, rinominare/spostare un file,[14] copiare una directory, rimuovere un file, rimuovere directories non vuote (usate `ls` per controllare cosa succede):

[13] Da qui in poi, riporto solo l'ultimo simbolo del prompt ($), seguendo la consuetudine delle guide a Unix. Dovrebbe essere chiaro dagli esempi dove ci troviamo.

[14] Anche se questo è un fatto che viene un po' oscurato dagli interfaccia grafici, un attimo di riflessione dovrebbe convincervi che rinominare un file e spostarlo in un'altra directory sono esattamente la stessa cosa.

```
$ cp cl_shared_data/novelle/01_scial.txt copia1
$ cp cl_shared_data/novelle/02_la_vi.txt .
$ mv copia1 copia2
$ cp copia2 copia3
$ mkdir tempdir
$ mv copia2 tempdir
$ cp -r tempdir tempdir2
$ mkdir tempdir3
$ mv 02_la_vi.txt tempdir3
$ cp tempdir3/02_la_vi.txt tempdir3/copia4
$ rm tempdir3/02_la_vi.txt
rm: remove regular file ‘tempdir3/02_la_vi.txt’? y
$ rm -f tempdir3/copia4
$ rm -r tempdir
rm: descend into directory ‘tempdir’? y
rm: remove regular file ‘tempdir/copia2’? y
rm: remove directory ‘tempdir’? y
$ rm -fr tempdir*
$ rm -f copia3
```

- Leggere un file schermata per schermata (stesse regole di navigazione di `man`):

  ```
  $ more cl_shared_data/brown.txt
  ```

- Unix gives you enough rope to hang yourself.
- Per esempio, un comando come `rm -fr *` non è praticamente mai una buona idea.
- Ma sistema di permessi, se computer ha buon amministratore, dovrebbe bloccare le azioni più pericolose:

  ```
  $ ls
  brown.txt       genesis.txt  novelle
  europarl_it_de  lob.txt      unsorted_repubblica_words.txt
  $ rm -fr *
  rm: cannot remove ‘brown.txt’: Permission denied
  rm: cannot remove ‘europarl_it_de’: Permission denied
  rm: cannot remove ‘genesis.txt’: Permission denied
  rm: cannot remove ‘lob.txt’: Permission denied
  ```

```
rm: cannot remove ‘novelle’: Permission denied
rm: cannot remove ‘unsorted_repubblica_words.txt’: Permission denied
$ ls
brown.txt       genesis.txt  novelle
europarl_it_de  lob.txt      unsorted_repubblica_words.txt
```

4.4 Struttura di un comando Unix

- *NOME_DEL_COMANDO OPZIONI ARGOMENTI*
- Per esempio: `rm -rf tempdir`
- Scrivere il nome del comando fa partire il comando in questione: è come scegliere il nome di un programma da un menu nell'interfaccia grafico.
- Le opzioni sono precedute da "-" e specificano, appunto, varie opzioni su come il comando vada eseguito: nell'esempio di sopra, `r` specifica che la rimozione deve essere *ricorsiva* e `f` di forzare la rimozione (ossia di non chiedere conferma all'utente).
- Come il nome suggerisce, le opzioni sono quasi sempre opzionali (ma non sempre).
- Gli argomenti sono gli "oggetti" a cui applicare il comando: spesso, si tratta del nome di files e directories, ma non solo (per esempio, in un programma che cerca una parola in un file, gli argomenti potrebbero essere la parola in questione e il nome del file in cui cercare).
- Alcuni comandi hanno argomenti di default: per esempio, se invocate `ls` senza argomenti esso listerà i files nella directory in cui vi trovate.
- Le opzioni possono, a loro volta, avere degli argomenti (per esempio, un comando potrebbe richiedere un'opzione `-f file`, dove `file` è il nome di un file da cui leggere dei dati).
- Non è sempre ovvio cosa vada passato come opzione e cosa vada passato come argomento, ma dopo un po' si impara, almeno per i comandi usati più spesso (per gli altri, basta dare un'occhiata a `man`).
- Un'altra cosa che si acquisisce solo con la pratica e l'abitudine è l'uso appropriato delle virgolette (quali comandi richiedono quali virgolette per quali opzioni e quali argomenti).

4.5 Primi comandi utili per l'analisi files di testo

- `wc`:

```
$ wc genesis.txt
   1534   39628  207930 genesis.txt
$ wc *txt
  92291 1015440 5998341 brown.txt
   1534   39628  207930 genesis.txt
  99932 1005539 5817523 lob.txt
 880111 1760204 11116406 unsorted_repubblica_words.txt
1073868 3820811 23140200 total
```

- `head` e `tail`:

```
$ head -5 brown.txt
$ tail -2 lob.txt
```

- Numero di righe mostrate per *default* (senza opzioni specificate): 10.

4.6 Piping e redirection: la "colla" che lega i comandi Unix

- Comandi come filtri applicati al testo, o come le componenti di un programma più complesso assemblato dall'utente.
- Metafore popolari: box degli attrezzi, coltellino svizzero.
- Un esempio semplice e tipico:

```
$ tail -100 brown.txt | more
```

- (Prima, provate senza `more`!)
- Leggi le dieci righe da 3991 a 4000 (per esempio, per avere un'idea dei contenuti di un file...):

```
$ head -4000 brown.txt | tail -10
```

- Controlla di aver fatto la cosa giusta (e scopri l'utile comando `nl`):

```
$ nl brown.txt | head -4000  | tail -10
```

- Salvare l'output in un file nella tua home (in unixese: redirect standard output to a file):[15]

```
$ nl brown.txt | head -4000  | tail -10 > ~/ten_lines_of_brown
```

- Processo "incrementale" nella costruzione di pipes: prova un comando e ispeziona output con `more` (o `head` e `tail`), aggiungi un altro comando, ispeziona output, e via dicendo.
- Quando tutto sembra a posto, "ridirigi" standard output in un file.
- Salva il contenuto di tutte le novelle in un singolo file:

```
$ cat cl_shared_data/novelle/*_* > novelle.txt
```

- Ricordarsi che l'input da file o da altre fonti va specificato solo per il primo comando della pipe: gli altri comandi riceveranno come input l'output del comando precedente.
- Come sempre, prestare attenzione ai percorsi: avete specificato dove si trova il file d'input? Avete specificato dove volete creare il file d'output?
- (Un errore classico: se siete dentro a `cl_shared_data`, non occorre specificare un percorso per il Brown, ma in compenso dovete specificare un percorso che conduca alla vostra directory per il file d'output, e viceversa se siete nella vostra directory.)
- Un esempio di piping realistico (reale):

```
$ ../scripts/get_connector_grams.pl 4 candidate_uniterms \
ac_sites.tok | grep -v _ | sort | uniq -c | perl -ane \
'if ($F[0]>2){print join " ",@F[2...($#F-1)];print "\n";}' \
| sort | uniq -c | \ gawk '{print $2,$3,$1}' | sort -nrk3 \
> bi_connect_all
```

[15] Da notare: in linea di massima le utilities Unix non *modificano* il file a cui applicano una trasformazione, ma mandano all'output (schermo o altro file) il risultato della trasformazione. A (s)proposito: ricordatevi dell'up-arrow mentre fate queste prove...

5 Un primo incontro con `egrep`

- `egrep` trova tutte le righe in un file o insieme di files che contengono un certo *pattern*:[16]

```
$ egrep amarezza novelle/* | more
$ egrep bitterness brown.txt | more
```

- Ci sono moltissime opzioni, vale la pena di leggere il `man`.
- Per esempio:

```
$ egrep -C1 bitterness brown.txt | more
$ egrep -nC1 bitterness brown.txt | more
```

- Provate la seguente ricerca:

```
$ egrep anger lob.txt | more
```

- Il risultato è quello che vi aspettavate?
- Date un'occhiata al manuale, ripetete la ricerca con opzioni che la rendano case-insensitive e limitata a parole intere, e salvate il risultato in un file.
- *bitterness* e *anger* sulla stessa riga:

```
$ egrep -win bitterness lob.txt | egrep -wi anger
```

5.1 Le espressioni regolari

- La vera potenza di `egrep` (e molte altre utilities) deriva dalla possibilità di usare le *espressioni regolari*.[17]
- Il pattern che viene ricercato tramite `egrep` può contenere dei caratteri speciali che permettono di rappresentare *insiemi* di stringhe, oltre che semplici stringhe.

[16] Esiste anche un comando `grep`, che fa quasi la stessa cosa. Tuttavia, alcune espressioni regolari (vedi sezione 5.1) funzionano solo in `egrep`, dunque tanto vale usare direttamente quest'ultimo.

[17] Non cerco qui di definire che cosa sia una espressione regolare dal punto di vista formale, ma mi limito ad illustrarne l'uso tramite esempi. Per un'introduzione più rigorosa, che mette in relazione espressioni regolari e automi a stati finiti, leggete il capitolo 2 del Jurafsky & Martin.

- Un pattern come `bitterness` rappresenta un'espressione regolare molto semplice: come abbiamo visto, usando questo pattern troviamo semplicemente tutte le righe che contengono la sequenza di caratteri `bitterness`.
- I simboli `^` e `$` permettono di *ancorare* la stringa all'inizio e alla fine della riga. Provate per esempio:

```
$ egrep -i the brown.txt | head
$ egrep -i "^the" brown.txt | head
$ egrep -i "the$" brown.txt | head
```

- Tra parentesi quadre possiamo specificare un insieme di caratteri.
- Per esempio, così troviamo tutte le righe che iniziano per vocale:

```
$ egrep -i "^[aeiou]" brown.txt | head
```

- Basandoci sull'ordine dei caratteri nella tabella ASCII, possiamo anche usare il dash ("-") all'interno delle quadre per specificare un range.
- Per esempio, così cerchiamo tutte le righe che contengono una sequenza di una lettera minuscola immediatamente seguita da una lettera maiuscola:

```
$ egrep "[a-z][A-Z]" brown.txt | more
```

- Il simbolo `^` subito dentro una parentesi quadra *nega* i simboli che seguono. Dunque, con la seguente ricerca troveremo tutte le righe che iniziano con sequenze di tre consonanti, o meglio (come si inferisce dai risultati) di tre simboli non-vocalici:[18]

```
$ egrep "^[^aeiouAEIOU][^aeiouAEIOU][^aeiouAEIOU]" brown.txt | more
```

- I simboli `*`, `+` e `?` specificano il numero di volte che l'elemento *immediatamente precedente* può capitare; rispettivamente: 0 o più volte, una o più volte, zero o una volta.[19]

[18] Notare i due significati distinti del simbolo `^`: fuori da parentesi quadra è un'ancora, all'interno della parentesi quadra è una negazione.

[19] Non confondersi con il significato che * ha nel file globbing (e in molti programmi che supportano "wildcard characters" e simili). Se listiamo dei file con `ls`, un'espressione quale `file*` vuol dire: tutti i files che cominciano con la stringa `file`. Ma in una espressione regolare (per esempio in `egrep`), la stessa espressione vuol dire: tutte le righe che contengono la sequenza `fil` e zero o più `e`. In un mondo dominato da forze razionali non esisterebbero ambiguità di questo genere.

- Spesso questi simboli sono usati in combinazione con il simbolo ".", che vuol dire: qualsiasi carattere.
- Cosa cerchiamo con le seguenti espressioni regolari?

```
$ egrep "patterns?" brown.txt | more
$ egrep "^....$" brown.txt | more
$ egrep "^....?$" brown.txt | more
$ egrep "^....+$" brown.txt | more
$ egrep "^....*$" brown.txt | more
$ egrep -i "^[^aeiou]*[aeiou][^aeiou]*$" brown.txt | more
```

- Il simbolo \ ha la funzione di forzare un'interpretazione letterale di quello che segue (altrimenti, non potremmo mai cercare un punto o un punto di domanda!)

```
$ egrep "\.\.\." lob.txt | more
$ egrep "^[^\.\?]*$" brown.txt | more
```

- Occorre molta pratica, ma le espressioni regolari (che, ripeto, non sono limitate a `egrep`) sono uno strumento potentissimo per estrarre informazioni da dati testuali.
- Notate che le espressioni regolari si applicano alle *righe* di un file.
- Tuttavia, dal punto di vista linguistico le righe sono raramente unità interessanti.
- Dopo che avremo imparato come *tokenizzare* corpora, ossia dividere il testo in modo che ci sia una parola per riga, le espressioni regolari ci torneranno utili per investigare proprietà delle parole, anziché delle righe.
- Segue una tabella riassuntiva.[20]

[20] Da "Unix for Poets" di Kenneth Church, con alcune modifiche.

Esempio	*Spiegazione*
`a`	la lettera "a"
`[a-z]`	qualsiasi lettera minuscola non accentata
`[a-zàèéìòù]`	qualsiasi lettera minuscola in italiano
`[A-Z]`	qualsiasi lettera maiuscola non accentata
`[A-ZÀÈÉÌÒÙ]`	qualsiasi lettera maiuscola in italiano
`[0123456789]`	qualsiasi numero
`[0-9]`	qualsiasi numero
`[aeiouAEIOU]`	qualsiasi vocale non accentata
`[^aeiouAEIOU]`	qualsiasi carattere tranne le vocali non accentate
`.`	qualsiasi carattere
`^`	inizio riga
`$`	fine riga
`x*`	zero o più x
`x+`	uno o più x
`x?`	zero o un x
`x|y`	o x o y

5.2 A quick note on kwic

- `egrep` ha un'opzione `--color` che produce risultati dove la stringa ricercata è evidenziata in rosso.
- Questo può essere utile quando il vostro fine ultimo è ispezionare il contesto in cui la stringa in questione capita nel corpus.
- Allo stesso fine, potete usare `kwic`, che produce risultati in formato *keyword in context*:

```
$ kwic -c35 friendship lob.txt | more
```

- Anche `kwic` supporta le espressioni regolari.

6 La tokenizzazione con `sed` e `egrep -v`

- Comandi come `egrep` diventano davvero utili al linguista una volta che *tokenizziamo* un corpus, ossia lo dividiamo in tokens.[21]

[21] Dico "tokens" invece che "parole" perché spesso vogliamo trattare come unità indipendenti stringhe che normalmente non considereremmo parole, come i numeri e la punteggiatura.

- Anche se altri formati sono possibili, tokenizzeremo i nostri corpora mettendo un token per riga – questo li rende particolarmente facili da gestire con le command line utilities.
- "Tokenizzare bene" è complicato.
- Pensate per esempio al problema di distinguere tra
  `Studio all'U.C.L.A.`
  dove vorremmo che `U.C.L.A.` fosse trattato come un solo token, punti inclusi, e
  `Sono nella sezione A.`
  dove vorremmo che `A` fosse separato dal punto che segue.
- Tuttavia, spesso un certo margine di errore è tollerabile (e questo è una delle massime fondamentali per chi fa linguistica computazionale basata sui corpora!)

6.1 Mettere le parole una per riga con `sed`

- `sed` è un *non-interactive stream editor*: prende una o più istruzioni, e le applica in sequenza a ciascuna riga del file specificato (come al solito, il risultato viene mandato allo standard output, e può essere ridiretto in un file).
- Con il seguente comando, sostituiamo tutti i caratteri che non sono lettere non accentate con il simbolo di a capo:

```
$ sed "s/[^a-zA-Z]/\n/g" brown.txt | more
```

- L'esempio appena mostrato si interpreta così:
  - `sed "x" y`
    significa: applica l'istruzione x al file y.
  - All'interno dell'istruzione, `s/x/y/g` significa: sostituisci tutte le occorrenze dell'espressione regolare x che trovi in una riga con la stringa y.[22]
  - `[^a-zA-Z]`, come già sappiamo, si riferisce all'insieme di tutti i caratteri non alfabetici o alfabetici con diacritici (per tokenizzare l'italiano, dovremmo aggiungere a questo set le vocali accentate).

[22] Provate a vedere cosa succede senza la `g` finale – che richiede che un cambio sia eseguito "globalmente".

  - `\n` è un carattere speciale che `sed` interpreta come l'istruzione di inserire un "a capo".

- La tokenizzazione appena mostrata dividerà `it's` in due tokens `it` e `s`. Come si fa a mantenere l'apostrofo e tokenizzare `it's` come un singolo token?
- Per l'italiano, le cose sono un po' più complicate (soprattutto se ci interessa mantenere l'apostrofo ma separare le parole in, e.g., `un'amica`.)
- In `sed`, possiamo dare molteplici istruzioni (che si applicheranno alla stessa riga una dopo l'altra) separandole con il punto e virgola.
- Dunque, possiamo fare così:

```
$ cat cl_shared_data/novelle/*_* \
| sed "s/'/' /g; s/[^a-zA-ZàèéìòùÀÈÉÌÒÙ']/\n/g" | more
```

- Cosa abbiamo fatto?
- Per alcuni testi o alcuni scopi, schemi di tokenizzazione così semplici possono creare dei problemi seri.
- Tokenizzare numeri, sigle, indirizzi web, punteggiatura, ecc. diventa immediatamente complicato: `U.C.L.A.`, `http://www.google.com`, `...`, `ecc.`, `100.000` e via dicendo.
- Se necessario, al posto di `sed` si può usare un tokenizzatore "vero" (e.g., il mio *regexp_tokenizer.pl*: vedi informazioni sul mio sito)...
- ... ma per molti progetti `sed` può bastare.

6.2 Eliminare righe con `egrep -v`

- Laddove `egrep` stampa[23] tutte le righe che contengono un pattern, `egrep -v` stampa tutte le righe che *non* contengono un pattern.
- Per esempio, lo possiamo usare per cancellare tutte le righe vuote create da `sed` nei comandi che abbiamo appena visto:

```
$ sed "s/[^a-zA-Z]/\n/g" brown.txt | egrep -v "^$" | more
```

[23] Qui e altrove, uso il verbo *stampare* nel senso informatico di *mandare all'output*, e non nel senso di: *mandare a una stampante*. Quando i computer non avevano monitor, i due sensi erano equivalenti.

- Possiamo anche usarlo per togliere i separatori di documento dal Brown e dal LOB prima di invocare `sed`:

```
$ egrep -v "^CURRTEXT" brown.txt |sed "s/[^a-zA-Z]/\n/g" \
| egrep -v "^$" | more
```

6.3 Maiuscole e minuscole

- Spesso, ha senso convertire tutte le maiuscole in minuscole prima di analizzare un corpus.[24]

- Per questo corso, ho preparato `lc`, un mini-programma che converte le maiuscole in minuscole:[25]

```
$ egrep -v "^CURRTEXT" cl_shared_data/brown.txt \
|sed "s/[^a-zA-Z]/\n/g" | lc | egrep -v "^$" | more
```

7 Lavorare con corpora tokenizzati

- Una volta che un corpus è stato tokenizzato, diventa molto più facile estrarre informazioni utili con le command line utilities, poiché abbiamo un'equivalenza tra parole e righe.

- Tokenizzate il Brown, il LOB e le novelle salvando gli output nella vostra home: `brown.tok`, `lob.tok` e `novelle.tok`.

- Per le novelle, ricordatevi di includere solo i files che contengono un underscore (_), perché gli altri files nella directory novelle sono documenti informativi che non fanno parte dell'opera di Pirandello.

- La prima cosa da fare è provare `wc`:

```
$ wc *tok
1024374 1024374 5769743 brown.tok
1016404 1016404 5619864 lob.tok
 861095  856293 4857355 novelle.tok
2901873 2897071 16246962 total
```

[24] L'ideale sarebbe lasciare le maiuscole in nomi propri e simili ma convertire le maiuscole in inizio di frase. Tuttavia, fare ciò con `sed` sarebbe estremamente complicato, e spesso i vantaggi di un'operazione del genere non valgono davvero lo sforzo.

[25] Un'alternativa più standard sarebbe `tr`, vedi "Unix for Poets".

- Adesso il numero di righe corrisponde al numero di tokens, ed è un conto più affidabile di quello che otteniamo se lasciamo a `wc` il compito di decidere il numero di parole (notate la differenza tra la prima e la seconda colonna per `novelle.tok`).
- Usate `egrep` e `wc` per contare il numero di occorrenze della parola `scialle` nelle novelle.
- Cercate le parole di almeno dieci caratteri nei tre corpora.
- Contate il numero di parole che finiscono per `t` nei tre corpora.

7.1 Lessici e liste di frequenza con `sort` e `uniq`

- Il lessico più semplice possibile è una lista delle parole distinte (o *tipi*) che capitano in un corpus.
- Per estrarre un lessico da un corpus tokenizzato, mettiamo le parole in ordine alfabetico con `sort` e poi per ciascuna sequenza di parole identiche rimuoviamo tutte le istanze tranne una con `uniq` (provate a guardare gli output intermedi con dei `more`, e dovrebbe essere ovvio cosa succede):

```
$ sort brown.tok | uniq > brown.lex
```

- Ripetete l'operazione per gli altri corpora.
- Contate e ispezionate le parole che finiscono per `t` nel lessico delle novelle.
- Contate e ispezionate le parole di meno di 4 lettere nel Brown.

7.1.1 Incrociare liste con `comm`

- Date due liste ordinate, `comm` stampa tutte le righe che capitano solo nella prima lista (prima colonna), solo nella seconda (seconda colonna), o in entrambe (terza colonna).
- Utilissimo per paragonare lessici!

```
$ comm brown.lex lob.lex | more
$ comm -23 brown.lex lob.lex | more
$ comm -13 brown.lex lob.lex | more
$ comm -12 brown.lex lob.lex | more
```

- Nelle opzioni, si specificano i campi che *non* si vogliono vedere (-23: mostra righe solo in primo file; -13: mostra righe solo in secondo file; -12: mostra solo righe comuni).

7.1.2 Liste di frequenza

- `uniq`, specificando l'opzione `-c`, stampa il numero di righe ripetute che ha trovato.
- Dunque, in un corpus tokenizzato e ordinato, ci dà la frequenza di occorrenza (*token frequency*) di ciascuna parola (*tipo*) nel corpus:

```
$ sort brown.tok | uniq -c > brown.fq
```

- Possiamo usare `sort` con le opzioni giuste per ispezionare questa lista in ordine di frequenza: l'opzione `-n` ordina numericamente, `-r` inverte l'ordine (vogliamo che 100 venga prima di 50) e `-kN` ordina sulla base dei dati nel campo `N`.
- Dunque:

```
$ sort -nrk1 brown.fq | more
```

- Mettiamo che non ci interessi salvare i dati in ordine alfabetico. Rimuovete `brown.fq` e, partendo da `brown.tok`, create direttamente una lista ordinata per frequenza decrescente.
- Ripetere l'operazione con `lob.tok` e `novelle.tok`.
- Salvate le 100 parole più frequenti del Brown e le 100 parole più frequenti del LOB in due files.
- Cercate le 100 parole più frequenti trattando Brown e LOB come fossero un solo corpus.

7.2 Operare sulle liste con `gawk`

- `gawk` è un linguaggio di programmazione molto adatto per lavorare su files di testi strutturati (e.g., le liste di frequenza).[26]

[26] `gawk` è una variante evoluta di `awk`: in linea di massima, la documentazione su `awk` si applica anche a `gawk`.

- Come con `sed`, qui useremo soltanto un piccolissimo sottoinsieme delle funzioni offerte da `gawk`.
- Un esempio, per cominciare: vogliamo estrarre dalla lista di frequenza del Brown le parole che hanno una frequenza maggiore di 100.

```
$ gawk '$1>100{print $2}' brown.fq | more
```

- `gawk 'x' y` applica l'espressione x a ciascuna riga del file (o files) y.
- All'interno dell'espressione, abbiamo una *condizione* seguita da un'*azione* (quest'ultima tra parentesi graffe).
- `gawk` applica l'azione a tutte le righe che soddisfano la condizione.
- In questo caso, la condizione è `$1>100`, l'azione è {`print $2`}.
- Per capire queste espressioni, bisogna sapere che `gawk` considera ogni riga come divisa in *campi* separati da uno o più spazi, e si riferisce ad ogni campo con la sintassi dollaro-numero (`$N`), dove il numero è il numero del campo.
- Per esempio, il file `brown.fq` conterrà due campi: `$1` (frequenza) e `$2` (parola).
- Dunque, la condizione `$1>100` significa: il primo campo (quello con le frequenze) deve avere un valore maggiore di 100.
- L'azione {`print $2`} significa: stampa (manda all'output) il secondo campo (quello con le parole).
- Ci si può riferire al numero dei campi con `NF` (che sta per *Number of Fields*) e all'ultimo campo con `$NF`.
- Le condizioni basate su comparazioni numeriche usano gli operatori `>`, `<`, `>=`, `<=`, `==`, `!=` (l'ultimo vuol dire: diverso da).
- Si possono combinare più condizioni con `&&`.
- Se non c'è una condizione, `gawk` applica l'azione a tutte le righe (comodo per eliminare o invertire colonne e simili).
- Il comando print può prendere più di un argomento (nel qual caso gli argomenti sono separati da virgole) o nessun argomento (nel qual caso stampa tutta la riga).

- Ecco un po' di esempi da provare:

```
$ gawk '$1==1{print}' brown.fq | more
$ gawk '$1==1{print $1,$2}' brown.fq | more
$ gawk '{print $2,$1}' brown.fq | more
$ gawk '$1>=100 && $1<=200 {print}' brown.fq | more
```

8 Bigrammi

- Spesso, oltre alle frequenze di parole singole, siamo interessati alla frequenza di due o più parole (*bigrammi*, *n-grammi*).
- Per creare una lista di bigrammi a partire da un corpus tokenizzato, prima creiamo una copia del file togliendo la prima riga, e poi combiniamo ciascuna riga del corpus originale con ciascuna riga del nuovo file.
- Per esempio, mettiamo di avere un file con queste parole:

```
la
linguistica
computazionale
mi
esalta
```

- Possiamo rimuovere la prima riga, ottenendo:

```
linguistica
computazionale
mi
esalta
```

- Adesso "incolliamo" i due files riga per riga, e otteniamo la lista dei bigrammi (rimane una parola in più alla fine, ma si può eliminare facilmente.)

```
la linguistica
linguistica computazionale
computazionale mi
mi esalta
esalta
```

- Per creare una copia di un file togliendo una riga all'inizio possiamo usare `tail` in una nuova maniera: invece di usare `-N` usiamo `+N`, così `tail` stampa tutte le righe *a partire* dalla N-esima riga, che è quello che vogliamo:

```
$ tail +2 novelle.tok > novelle.2
```

- Adesso dobbiamo incollare i due files riga per riga – cosa che possiamo fare con il comando `paste`:

```
$ paste novelle.tok novelle.2 | head
$ paste novelle.tok novelle.2 | tail
```

- Come ci aspettavamo, l'ultima riga non è un bigramma – potremmo toglierla a mano con un text editor, ma è più comodo chiedere a `gawk` di stampare solo le righe in cui il numero di campi è uguale a 2 (cioè tutte le righe tranne l'ultima):

```
$ paste novelle.tok novelle.2 | gawk 'NF==2{print}' > novelle.bigrams
$ tail novelle.bigrams
$ wc novelle.bigrams
```

- A questo punto, create una lista di frequenza per i bigrammi, e ordinatela per frequenza decrescente (non leggete oltre finché non l'avete fatto).

- Un modo di fare tutto ciò in una sola pipe (senza creare files intermedi):

```
$ tail +2 novelle.tok | paste novelle.tok - | gawk 'NF==2{print}' \
| sort | uniq -c | sort -nrk1 > novelle.bigrams.fq
```

- Nella pipe che precede, c'è una cosa nuova: quando un comando richiede più di un argomento, si può specificare che uno degli argomenti sia l'output del comando immediatamente precedente mettendo "-" al suo posto.

- Dunque, `paste novelle.tok -` nell'esempio appena presentato vuol dire: applica il comando `paste` usando `novelle.tok` come primo file e l'output del comando precedente (`tail +2 novelle.tok`) come secondo file.[27]

[27] Quando un comando richiede un solo argomento, non c'è bisogno di "-" perché non ci sono ambiguità: è quando un comando richiede più di un argomento che diventa obbligatorio specificare *dove* va inserito l'output della pipe.

9 `pico`, un text editor d'emergenza

- Con i comandi che abbiamo studiato, possiamo modificare i files in vari modi, e con `more` possiamo esplorarne i contenuti.
- Talvolta, però, ci farebbe comodo aprire un file e fare dei cambiamenti a mano.
- Unix offre due text editors molto potenti: `emacs` e `vi`.
- Mentre alla lunga imparare `emacs` o `vi` semplifica enormemente la vita, i primi passi con entrambi sono piuttosto difficili.[28]
- Per i principianti, l'editor da riga di comando più semplice da usare è `pico` (o il suo clone open source `nano`).
- Per far partire *pico*, date il comando:

```
$ pico file
```

- Se `file` è il nome di un file esistente, `pico` aprirà quel file. Se invece nella directory in cui vi trovate non esiste un file con quel nome, la prima volta che salvate `pico` vi chiederà se volete salvare i dati in un file con quel nome.
- La semplicità di `pico` deriva in gran parte dal fatto che in basso avete costantemente una lista dei comandi più comuni.
- In questa lista di comandi, notate che `^X` vuol dire: premere insieme il tasto `Ctrl` e il tasto `X`.

10 Esercizi

- Create una lista di frequenza dal LOB direttamente dal file lob.txt (nella cartella comune) in una sola pipe.[29]
- Raccogliete le 100 parole più frequenti nel Brown e le 100 parole più frequenti nel LOB, e incrociate queste liste usando `comm`, trovando le parole che capitano solo nella top list del Brown e le parole che capitano solo nella top list del LOB.

[28] Tipicamente, uno impara o `emacs` o `vi`, ne diventa fanatico, e poi guarda in cagnesco i sostenitori dell'altro editor. Naturalmente, noi utenti di `emacs` abbiamo ragione ;-)

[29] Quando fate questo e gli altri esercizi, ricordatevi di costruire le pipes un passo alla volta, controllando gli output intermedi con `more`.

- Ripetete questa operazione con i bigrammi (trovate i top 100 bigrammi del LOB e i top 100 bigrammi del Brown, e incrociate queste liste).
- Create una lista con le frequenze dei *trigrammi* (sequenze di tre parole) che capitano nelle novelle.

10.1 Spettri di frequenze

- In questo esercizio preparate dati utili per studiare le distribuzioni di frequenza delle parole in un corpus, un tema importante negli studi di *statistica lessicale*.
- A questo fine, ci interessa raccogliere *spettri di frequenze*.
- Uno spettro di frequenze riporta, per ciascuna frequenza nel corpus, il numero di parole che hanno quella frequenza.
- Dati di questo genere servono a studiare la distribuzione delle parole in un corpus (per esempio, ponendosi domande quali: qual è la proporzione di parole che capitano meno di tre volte in questo corpus?)
- Costruite lo spettro di frequenze del Brown, del LOB e delle novelle di Pirandello.
- Per esempio, una spettro di frequenze potrebbe avere il seguente formato:

```
18209 1
 6675 2
 3627 3
 2366 4
 1607 5
   ...
   ...
    1 17884
    1 18348
    1 20000
    1 21591
    1 26498
```

- La prima riga ci dice che ci sono 18209 parole con frequenza 1, la seconda riga che ci sono 6675 parole con frequenza 2, e l'ultima riga che c'è una sola parola con frequenza 26498.

- Pensateci bene – all'inizio può sembrare una cosa difficile, ma partendo dalle liste di frequenza delle parole e con un po' di `gawk`, `sort` e `uniq` non è difficile ricavare liste di questo genere.
- Visto che ci siete, create anche spettri di frequenze per i bigrammi e i trigrammi nelle novelle.
- Cosa si nota confrontando la proporzione di parole/bigrammi/trigrammi che capitano una volta sola?