PERLinguisti v0.7.7

Manuale di programmazione in Perl
per umanisti
(linguisti, giuristi, scrittori, pubblicitari, giornalisti...)

Adriano Allora

8 maggio 2006

Indice

Elenco delle tabelle

Capitolo 1

Introduzione

Le idee banali possono essere assimilate,
mentre quelle che richiedono una
riorganizzazione della propria immagine
del mondo suscitano ostilità.
James Gleick, *Caos*, BUR 1997

1.1 Come è nato questo manuale

Di solito, agli umanisti che devono studiare informatica (gli umanisti, per definizione, non vogliono ma devono studiare informatica), le lezioni di informatica insegnano molte (troppe?) cose inutili: cos'è una memoria di massa, se il plotter è input oppure output, le somme con i numeri binari, i registri e la cpu. Cioè quel che studiano gli informatici nelle prime lezioni di un corso generale introduttivo[1].

I docenti più "umanizzati" forniscono nozioni di storia dell'informatica[2], rispondendo ad una richiesta degli studenti stessi, che in questo modo studiano sì qualcosa di simile ad altre materie a loro più affini, però, intascato il voto dell'esame, rimangono in possesso di un sapere che è appunto storia dell'informatica, e non informatica.

L'informatica è una porta che apre la strada a nuovi modi di agire e di pensare, anche fuori ed indipendentemente dai calcolatori; è prima di tutto un sapere performativo[3], e deve essere utile: qualsiasi insegnamento di questa materia che ignori tale imperativo categorico fallisce miseramente.

[1] Ed è quanto accade in [Ham03]: un manuale di Perl ottimo nelle intenzioni ma che, a parte il titolo e la lentezza con la quale sono date le informazioni, non ha nulla che lo qualifichi veramente come "per umanisti".

[2] Come avviene ad esempio in [Gig97], che però riesce anche ad arricchire con descrizioni di programmi e concetti teorici.

[3] La teoria dell'informazione, della programmazione, di tutto quel che è *computer science*, è meglio lasciarla a specialisti che hanno basi matematiche e tecniche solide.

Nella direzione della prassi si collocano quei docenti che insegnano come usare specifici programmi, i quali spesso però non forniscono alcuna formazione su cosa è e come funziona davvero un programma[4].

La strada dell'addestramento all'uso di specifici programmi è relativamente facile (per studenti e docenti), ma rischiosa soprattutto perché nella maggior parte dei casi non tiene conto del contesto nel quale tali programmi potrebbero, o dovrebbero, trovare applicazione: è sempre presente il pericolo che il discente rimanga legato a quei programmi, a quelle versioni, a quelle procedure senza essere in grado di fare astrazione, di elaborare strade alternative. Il discente, allora, diventa uno studente dell'avere – per riutilizzare una celebre opposizione – piuttosto che dell'essere e nel lungo periodo non gli resta che essere costretto a "ricomprare" ogni volta che diventa davvero necessario usare il calcolatore.

Questo testo nasce, nelle intenzioni, da un corso di Informatica applicata alla comunicazione che ho tenuto nell'a.a. 2004-2005 presso l'Università degli Studi di Torino, Facoltà di Lingue e letterature straniere[5].

Alla realizzazione del manuale sono intervenute però diverse esperienze: in maniera sostanziale il dottorato di ricerca[6] e poi altre docenze e assegni di ricerca, lezioni per colleghi di dottorato, peregrinazioni in ambiti informatici che non mi erano propri ma nei quali ho trovato comunque (quasi) sempre quella straordinaria energia che a suo tempo già il libro di Levy [Lev94] e quello di Singh [Sin99] mi avevano trasmesso.

Spero che un poco di quella energia, e passione, possa per tramite del presente manuale essere travasata nei lettori. Inoltre, mi sono divertito a scriverlo: spero che possiate divertirvi leggendolo.

1.2 A chi serve?

Questo manuale serve a chi serve il Perl – cioè a chiunque, per iniziare, abbia da fare analisi quantitative sui testi, ed eventualmente su testi non etichettati – e sia intimorito o non abbia voglia di leggere manuali scritti da esperti informatici per apprendisti esperti informatici. Inoltre il presente manuale può rivelarsi piuttosto utile a chiunque voglia gettare uno sguardo non troppo impegnativo sul mondo della programmazione in generale,

[4] Al limite estremo della dis-formazione troviamo la patente europea del computer, che finisce per trasmettere agli utenti liturgie vuote ed incomprensibili, anche se efficaci.

[5] L'ultima parte del corso, attraverso l'insegnamento degli elementi fondamentali del Perl, mirava a fornire gli studenti di qualcosa che potessero riutilizzare come fruitori, studiosi o professionisti della lingua, ma anche, più banalmente, qualcosa che fornisse loro un quadro di riferimento per capire la "roba tecnica" con la quale poi capita di scontrarsi nel lavoro d'ufficio (anche solo delle macro che qualcuno ha scritto per ottimizzare il lavoro e che improvvisamente iniziano a rallentarlo o delle formule esoteriche che un tecnico impiega per spiegare cosa non va).

[6] Il Dottorato di ricerca in Linguistica, Linguistica applicata, Ingegneria linguistica dell'Università degli Studi di Torino, coordinatore prof. Carla Marello.

a qualsiasi livello (anche se non è con questo manuale che arriverete a programmare sistemi operativi tridimensionali a comando vocale).

Peraltro, questa non è una guida di riferimento: manca molto di Perl, gli argomenti trattati sono tutti e sempre presentati in funzione dell'analisi e manipolazione dei testi e i programmi descritti privilegiano smaccatamente il fine didattico-esplicativo rispetto a velocità ed efficienza. Da tutti i non tecnici ci si attende, tuttavia, che abbiano qualche minima competenza di informatica, o fantasia, a sufficienza per capire:

- la differenza tra computer, monitor, mouse e tastiera (iniziamo con le cose più facili...);
- la differenza tra sistema operativo e applicativi (ma non da un punto di vista tecnico! Basta sapere e avere sempre ben chiaro in testa che il sistema operativo contiene gli altri programmi);
- la differenza tra input ed output (come azioni e come strumenti – o periferiche – che si usano per compiere l'azione[7]);
- una nozione almeno intuitiva di interfaccia (ma se volete approfondire potete sempre leggervi qualcosa di Raskin, come [Ras00]);
- la differenza tra se stessi come programmatori e se stessi come fruitori di un applicativo.

Inoltre, esiste anche un rozzo requisito linguistico: sarebbe auspicabile (per motivi che trascendono in realtà il presente testo) che il lettore avesse almeno un'idea generale e intuitiva di cosa sono semantica, sintassi e morfologia di una lingua (cos'è un morfema, cos'è una parola, cos'è una frase...).

Per il resto, non serve nient'altro, solo voglia di sperimentare e pazienza.

Dodici tonnellate di pazienza.

1.3 Perché Perl?

Se parlate con qualche informatico, potrebbe chiedervi perché proprio il Perl[8]. In fondo C è più veloce (ma più difficile), Java è multipiattaforma (ma lentissimo[9]), Python è più facile (ma meno potente sui testi, anche se ammetto che è proprio un bel linguaggio).

Non solo, il Perl ha un grosso difetto: *può essere* piuttosto oscuro.
Se partite dal presupposto che il maggior pregio di un programmatore è la pigrizia[10], e

[7] DOMANDA: la tastiera permette di fare input od output? E il video? E i touch-screen? E la stampante?

[8] Ah, già, Perl sta per: Practical Extraction and Report Language.

[9] E poi è multipiattaforma solo se si ha la Java Virtual Machine. A queste condizioni è multipiattaforma anche Python... hey, e anche Perl! – di sicuro lo sarà Perl 6.

[10] Un programmatore pigro ottimizzerà e automatizzerà tutto quel che può.

che in Perl una cosa può essere scritta in modi più chiari e verbosi o in modi più oscuri e stringati, ne otterrete che i programmi più interessanti sono brevissimi e scritti in modo assolutamente incomprensibile. Ma questo non significa che, anche senza i forti vincoli all'indentazione, imposti ad esempio da un linguaggio come Python, non si possa scrivere del Perl chiaro.

Il motivo per cui è stato scelto il perl è che, per alcuni versi, è molto simile alle lingue storico-naturali. Sarà quindi intuitivo fare riferimento a quella materia che con perfetta padronanza manipoliamo fin dai primi anni di vita: la lingua.

L'idea di un linguaggio tanto libero, però, va chiarita. Facciamo un esempio. Come in italiano posso dire:

È antipatico? Allora ammazzalo.
Se ti è antipatico ammazzalo.
Ammazzalo, se ti è antipatico.

Così con il Perl posso scrivere:

```
$altro eq "antipatico"? ammazzare($tu,$altro): exit;
if($altro eq "antipatico") {ammazzare($tu,$altro);}
ammazzare($tu,$altro) if $altro eq "antipatico";
```

E tutta questa discrezionalità non rende le cose infinitamente più difficili? Certo, noi scriveremo dei programmi leggibili e lo faremo con un linguaggio da grimorio di necromanzia. Ma sarà il primo linguaggio informatico che non vi darà l'orticaria. E farete cose meravigliose sui testi[11].

1.4 Come è strutturato questo manuale

Il presente manuale è costruito nel seguente modo:

capitolo 1 introduzione con le prime informazioni sul Perl;

capitolo 2 istruzioni per installare Perl sulla propria macchina. ATTENZIONE: leggere questo manuale senza usarlo per fare esperimenti è assolutamente inutile: questo libro è stato scritto per giocare col Perl e fare cose con le parole (del Perl, e della vostra lingua);

[11] Fino al 2001 ogni anno si è svolto un concorso di poesia in Perl: le poesie devono essere grammaticali e funzionanti, oltre che belle (http://www.perlguy.com/contest.html). A quelli che vi suggeriscono di imparare un altro linguaggio, chiedete se in altri linguaggi si fa ANCHE questo.

capitolo 3 iniziamo a vedere le basi del Perl, i tipi di parole che si incontrano programmando o leggendo programmi, da dove entrano ed escono informazioni; primi dettagli sulla punteggiatura;

capitolo 4 qualcosa di sensato: un approfondimento sui sostantivi e sui verbi, in maniera da avere gli strumenti di base per scrivere qualsiasi applicazione;

capitolo 5 in questo capitolo lo strumento più potente a disposizione dei programmatori Perl: le espressioni regolari, cioè quello straordinario linguaggio nel linguaggio che permette di descrivere un insieme di possibili realizzazioni linguistiche per fare operazioni su di esse (come fate a far sostituire tutte le parole più lunghe di sette lettere che iniziano con "va" con parole che iniziano con "fa" senza cambiare il resto di ogni singola parola?);

capitolo 6 in questo capitolo useremo il Perl per scrivere una serie di semplici programmi che non mancheranno di mostrare una certa utilità: un estrattore di liste di frequenza, un giocattolo con la legge di Zipf, un estrattore di concordanze, un correttore automatico (imparate a fare queste cose, e potrete fare qualsiasi altra cosa);

appendice A ecco perché smettere di usare MSWord o altri editor di testo WYSIWYG (What You See Is What You Get, cioè: quel che vedi è quel che fai) a vantaggio di uno strumento utile e multipiattaforma come LaTeX (un editor WYGIWYT: What You Get Is What You Think, cioè: quel che fai è quel che pensi). E, ancora, un paio di casi in cui Perl è utile e non lavora sui testi: menzioni di GIMP! e di ImageMagick;

appendice B quattro argomenti sui quali vale la pena di dire qualcosa, anche solo di sfuggita: la scrittura di pagine CGI in Perl, le interfacce grafiche per applicazioni *stand alone*, la programmazione orientata agli oggetti in Perl e il futuro di Perl.

1.5 Usi e costumi

1.5.1 Licenza

Penso che la licenza GPL e le Creative Common rappresentino il senso della collaborazione per il conseguimento di scopi comuni, scientifici e no.

Quindi il presente testo è licenziato sotto Creative Commons: *Attribuzione - Non Commerciale - Condividi allo stesso modo 2.0*.

Tale licenza permette la modifica e l'uso non commerciale di quest'opera (con l'eccezione della casa editrice Aracne[12] entro i limiti espressi dal contratto con l'autore), purché sia sempre attribuita la paternità dell'opera all'autore e purché non ne venga cambiata la licenza.

[12] Aracne editrice S.r.l., rappresentata dall'amministratore unico Gioacchino Onorati.

Se traducete questo libro, o se lo modificate, sarebbe *davvero* cortese comunicarlo all'autore.

Se desiderate ringraziare finanziariamente l'autore, potete acquistare una copia del libro pubblicato da Aracne[13] (che ha una copertina bellissima). Grazie!

1.5.2 Contatti

Per (gradite) segnalazioni di errori, imprecisioni e consigli, scrivetemi pure a: all.adr@e-allora.net
Vi risponderò in ritardo.

Una copia aggiornata del presente manuale, in versione LaTeX o PDF, potrebbe essere disponibile da qualche parte nel mio sito. Non ne sono sicuro ma cercarla non costa quasi nulla.

1.5.3 Ringraziamenti

Doverosi e da profondersi a piene mani alla professoressa Carla Marello, per un numero tale di motivi che qui sarebbe troppo lungo spiegare. E per la stessa quantità di motivi (quantunque non della stessa natura) alla dottoressa Manuela Manera.

Un sentito ringraziamento anche alla professoressa Marie Berthe Vittoz, il cui Corso di Laurea specialistica in Comunicazione Internazionale 43/S è il motore primo del presente manuale.

Un ringraziamento, ancora, alle studentesse che hanno letto, testato, suggerito correzioni (in rigoroso ordine alfabetico): Federica Anfossi, Alice Lanzone, Patrizia Mazzucato.

Infine, duratura gratitudine a tutti i membri delle varie comunità Perl sparse per il mondo cui ho fatto riferimento via mailing list in questi anni.

Grazie a tutti, senza di voi, semplicemente, questo libro non esisterebbe.

1.5.4 Convezioni tipografiche

Autori più qualificati e pazienti di me distinguono la parola *Perl* (il linguaggio) dalla parola *perl* (il programma che traduce i testi in Perl in qualcosa che una macchina possa comprendere, l'interprete perl).
Io però sono notoriamente rozzo, e tale notoria rozzezza mi permette di non dover andare ad alcune feste alle quali sarei certamente invitato altrimenti.

Oltre a questo indiscutibile vantaggio, mi permette di applicare il minimo sindacale delle convenzioni tipografiche: `testo a spaziature fisse` per gli esempi di codice perl;

[13] ISBN 88-548-0550-5, formato 17 x 24 cm, 184 pag., 9,00 euro.

corsivo, o italico per parole straniere (e solo quelle che io reputo tali) o per esempi di linguistica (parole in quanto materia verbale, oggetti della lingua, e non per il loro ruolo di veicoli di un significato); **grassetto** quando devo introdurre una definizione importante (ma non sempre, ché non mi piace).

Inoltre, per definire la pressione di un singolo tasto lo scriverò tra parentesi quadre, come in questo esempio: [invio]; e la stessa norma verrà applicata per l'esplicitazione del significato di una parola (il significato della parola *io* è [io]).

Le espressioni regolari saranno scritte, conformemente alla sintassi perl, tra barre, /in questo modo/. Me ne scuso con i fonetisti che potrebbero sviluppare dei tic nervosi a causa delle espressioni regolari che leggeranno laddove la loro *forma mentis* richiederebbe invece trascrizioni fonetiche.

Infine, identificherò il prompt di shell, di terminale o di ms-dos[14] con la coppia di simboli `$>`, e un comando colà digitato avrà la forma:
`$> comando` [invio]

Credo sia tutto.

[14] *Prompt* significa [suggerimento, sollecitazione] ed indica l'inizio della linea sulla quale scriverete i comandi per il computer.

Capitolo 2

A me gli occhi!

2.1 Installare Perl

Installare Perl è un'operazione che non si dovrebbe mai fare, ma che non è impossibile portare a termine.

Se avete una macchina Linux, Unix o MacOSX, perl è già installato, e il problema non si pone. Aprite una finestra di Terminale, una shell[1] o *switchate*[2] su un'altra shell e digitate:
`$> perl -v` [invio]
e saprete quale versione di perl è già stata installata. Se la vostra versione è sopra la 5.0 va tutto bene, altrimenti proseguite la lettura del presente paragrafo, e in particolare del paragrafo intitolato "lavorare con X".
Ad ogni modo, al momento in cui scrivo sulla mia macchina gira la versione 5.8*, quindi non credo che avrete problemi di sorta...

Se invece siete annoverati tra i disgraziati utenti di MS-Windows[3], proseguite la lettura al paragrafo intitolato "lavorare con win".

[1] Una **shell** è una interfaccia testuale del computer, nella quale tutto si fa per mezzo di comandi digitati su tastiera (anche guardare dentro una cartella!).

[2] Orribile, vero? Deriva dal verbo inglese *to switch* [slittare, passare a...]. Per passare alla shell su linux basta premere i tasti F (F1, F2, F3...) fino a sei, infatti lo schermo F7 è di solito quello usato dall'interfaccia grafica.
IMPORTANTE: se siete puristi della lingua o intregralisti della Crusca (non in senso dietetico), interrompete immediatamente la lettura del presente testo! Io ricorro ad un idioletto italiano piuttosto corrotto: diversi puristi sono stati colti da attacchi epilettici soltanto vedendomi scrivere.

[3] Nel senso che lavorare con quel sistema operativo è una disgrazia. Per carità: non ho nulla contro di voi!

2.2 Lavorare con X

2.2.1 Installare su X

Scaricate perl dall'archivio CPAN (Comprehensive Perl Archive Network): http://www.cpan.org.

È un archivio (tar, ovvero *tape archive*) compresso (con un programma che si chiama gzip); quindi ha estensione `.tar.gz`.

Per decomprimere aprite una shell e spostatevi nella cartella nella quale avete scaricato perl. Potete spostarvi da una cartella all'altra con il comando `cd`. Con `$> cd ~`[invio] vi spostate alla vostra cartella utente, con `$> cd ..`[invio] nella cartella di livello immediatamente superiore, con `$> cd /`[invio] nella cartella radice che contiene tutte le altre cartelle.

Per spacchettare e decomprimere perl digitate:
`$> tar -zxvf nome.della.cartella.da.decomprimere` [invio]
Mandare in esecuzione:
`$> configure` [invio]
e rispondere affermativamente quando lo script shell `configure` chiede se si vuole il caricamento dinamico.

Leggete il file README e procedete a tutte le esecuzioni di
`$> make` [invio]
richieste.

Queste indicazioni sono un po' sbrigative, ma lavorando su sistemi operativi X-like non ne avrete bisogno nella maggior parte dei casi (non ho mai visto una macchina X-like senza perl).

Inoltre, se qualche comando di quelli che menziono (la prima parola subito dopo i digramma `$>`) vi è oscuro, sarà sufficiente usare il comando per la visualizzazione dei manuali:
`$> man nome.programma` [invio]

2.2.2 Eseguire script perl su X

Per eseguire script perl bisogna renderli eseguibili. Sempre da shell:
`$> chmod +x nome.dello.script` [invio]
il programma `chmod` cambia la modalità di funzionamento di un file; l'opzione `+x` rende eseguibile il file nominato di seguito.

Se vi trovate nella cartella in cui si trova anche il listato, potete avviarlo semplicemente invocando direttamente l'interprete perl e spiegandogli che deve interpretare il testo contenuto nel listato:
`$> perl nome.del.listato` [invio]
Ma si può anche velocizzare la procedura.

Innanzitutto dovete capire dove sta l'interprete, con il comando:
`$> which perl` [invio]

e tenere a mente l'indirizzo che vi verrà restituito (potrebbe essere qualcosa tipo /usr/bin/perl o /usr/local/bin/perl).

Come prima linea del listato dovete quindi scrivere l'indirizzo che vi ha restituito l'interrogazione con `which perl`; nel resto del manuale immagineremo che sia `#!/usr/bin/perl`.

Reso eseguibile lo script e specificato l'indirizzo dell'interprete, si avvia il programma con:
`$> ./nome.listato` [invio]
Attenzione alla sequenza punto-slash: se li usate non state più invocando direttamente l'interprete perl, ma segnalate che c'è un programma (il vostro listato, nella cui prima linea è scritto che deve essere interpretato dal perl) e che il programma va cercato nella directory nella quale vi trovate.

Sembra macchinoso, lo ammetto, ma alla lunga darà i suoi frutti. Il "problema" deriva dal fatto che X cerca i programmi in cartelle speciali (il che contribuisce a renderlo più sicuro contro i virus). Se noi non mettiamo nelle cartelle speciali i nostri programmi, lui non li trova. La sequenza punto-slash significa, nel linguaggio della shell, [qui]; specificando che si deve eseguire un programma [qui] e, nel programma, che deve essere letto dall'interprete perl (che si trova in /usr/bin/perl), risolviamo il problema delle cartelle speciali.

2.3 Lavorare con win

2.3.1 Installare su win

Cercate su Google "perl for windows": il primo risultato dovrebbe essere quello del sito ActiveState. Non scaricate nulla che abbia nel nome il morfema commerciale *pro* (dovrebbe esserci un semplice link "Perl language" o qualcosa di simile).

Quando lo avete sulla vostra macchina, cliccate due volte sull'icona con estensione .exe. Fatto.

2.3.2 Eseguire script perl su win

Per eseguire script perl su win, dovete usare il DOS (che in winXP ha un nome diverso... tipo esecuzione comandi... non so, non ricordo, ho rimosso)[4].

Appena aprite il DOS, nella riga col cursore lampeggiante vi viene segnalato dove vi trovate, in quale cartella; potreste essere nella vostra cartella utente – in winXP – oppure in una cartella di C: che contiene anche win. Cercate di scoprire dove vi trovate con il comando `dir`, che elenca il contenuto della cartella corrente[5], poi cercate di spostarvi sul desktop, o dove sta il vostro script perl, con il comando
`$> cd nome.cartella` [invio]

[4] Dovrebbe sempre essere possibile aprire il DOS anche da "Esegui..." digitando nel campo di testo `command`.

[5] È il corrispettivo del comando `ls` in X.

Per spostarvi invece in una cartella in alto (che contiene quella nella quale vi trovate adesso[6]), potete invece digitare:
`$> cd ..` [invio]
Dovrete fare qualche esperimento, è inevitabile, ma è tutto esercizio che si tesaurizza.
Quando sarete nella cartella che contiene il file perl, eseguitelo digitando:
`$> perl nome.script` [invio]

2.4 Scorciatoie per X e win

Ricordatevi che su X per ripetere l'invio di un comando già inviato basta premere il testo [freccia in alto] finché non compare il comando che vi interessa (se l'avete appena fatto, basta una pressione), e poi premete [invio] come al solito.

Su win la cronologia è ridotta (ma a partire da winME, e quindi anche su winXP, hanno migliorato l'interfaccia dei comandi e dovrebbe funzionare come su X, fate qualche prova), quindi se volete ripetere l'ultimo comando, premete il tasto [freccia a destra] per riscriverlo.

2.5 L'inevitabile pistolotto su Linux

2.5.1 I like X-like

È necessario chiarirlo subito: penso che iniziare lavorando su Windows sia indispensabile, perché naturale, soprattutto finché Linux continuerà a porre problemi di compatibilità e accessibilità. Ma ad un certo punto, se non si usa il computer esclusivamente per videogiocare, rimanere su quel sistema operativo è come avere una palla di acciaio incatenata alle caviglie. E trovarsi su una zattera in mezzo al mare giusto al di qua di un fronte tempestoso[7].

Le nuove versioni di Windows, per quanto esteticamente belle siano (e comunque sempre un passo dietro ai sistemi operativi Apple), rimangono ricche di buchi, di virus, di nemici, di limitazioni, di controlli, di lentezze, di spese. Le macchine con quel sistema operativo fanno parte di un circolo vizioso: se vuoi il nuovo sistema operativo devi comprarti una macchina nuova, se vuoi una macchina nuova devi comprarti anche il nuovo sistema operativo. E dopo due anni devi ricominciare.

I sistemi operativi X-like richiedono una curva di apprendimento sensibilmente più ripida all'inizio, ma una volta che avrete ingranato la marcia, qualsiasi cosa dobbiate fare, su piattaforme X-like vi verrà meglio, sarà più facile, più sicura, più economica, non si

[6] Sembra strano dire che ci si trova in una cartella, ma è tutta colpa delle interfacce grafiche e di quanto esse siano svianti nella comprensione della struttura e del funzionamento del computer.

[7] Le opinioni espresse dall'autore su ogni programma o *software-house* menzionati si intendono personali e non supportate da alcuna motivazione diversa dalla propria esperienza.

bloccherà e potrete farla senza paura che qualcuno in quel momento stia usando il vostro sistema operativo per fare altro (come spiarvi o danneggiare la vostra rete locale).
E, con poche eccezioni, potrete ripetere l'esperienza senza sostanziali problemi di compatibilità con macchine di dieci anni fa e con macchine che si useranno ancora tra dieci anni.

2.5.2 Inter-facce toste

Chi si avvicina al mondo Linux scopre abbastanza in fretta che ne esistono molte versioni, chiamate *distribuzioni* o *distro*, e che Linux non è che un membro di una famiglia più numerosa: esistono anche BSD e FreeBSD, Unix, Solaris, MacOSX e probabilmente molto altro che io ignoro.

In questo paragrafo mi limiterò a fornire qualche informazione pratica su alcune distribuzioni di Linux.

Anche se negli anni Linux è via via diventato sempre più accessibile, credo che le distribuzioni più "grosse" rimangano, sia per l'installazione che nell'uso e nella configurazione, le più facili: Mandriva, Red Hat, Suse.
Personalmente vi consiglio però Ubuntu.

Alcuni vi suggeriranno di provare altri distribuzioni, come Knoppix (magari in versione *live*, non è una brutta idea) o Crux; il mio consiglio è: se vi promettono di restare con voi fino a che non avete finito l'installazione e risolto tutti i possibili problemi di configurazione iniziale, usate quella che vi consigliano. Un amico preparato vale più della più facile interfaccia pensabile (be', un amico preparato è la più facile interfaccia pensabile).
Attenti a quelli che sembrano preparati e non lo sono.

E badate al fatto che alcune pregevolissime distribuzioni (Gentoo, Slackware o Debian, per fare alcuni nomi) potrebbero darvi problemi più avanti, se volte cambiare qualcosa: sono più solide e performanti delle altre, ma anche meno facili.

Se avete un amico esperto che vi permette anche di scegliere la distribuzione (tanto, se se ne conosce bene una, le altre sono simili) e vi promette assistenza post-installazione, provate Debian oppure Gentoo.

Se volete avere accesso alle meraviglie di Unix, di cui Linux e molti altri sono figli, ma non volete rinunciare a Word, potete sempre lavorare su Mac che, al momento, è l'unica piattaforma che accetta contemporaneamente programmi di Microsoft, Adobe, Macromedia, Oracle (per citarne alcuni), ottimi programmi proprietari (Keynote fa mangiare la polvere a PowerPoint) ma anche software GPL (come GIMP! ed OpenOffice), shell-scripting, perl e cosette di questo tipo.

Si tratta di un sistema operativo che non dà problemi di configurazione, è molto facile da usare, bello a vedersi, sicuro e performante. Inoltre, se vi occupate di lingue, supporta perfettamente un notevole numero di lingue, e il passaggio dall'una all'altra è sempre indolore e veloce.

Certo, è necessario comprare l'hardware Apple[8], ma alcuni lo fanno: non dovrebbe essere una catastrofe.

2.5.3 Due titoli per seguire strade

A macchina perfettamente installata, vi consiglio di consultare Stutz [Stu01], per agire su X: si tratta di un libro di "ricette" agile e semplice che non mancherà di mostrarvi qualche piccola magia che si può eseguire con X. Invece, se avete tempo e voglia di scoprire nei dettagli il sistema operativo, vale proprio la pena di leggere [War04]. Sono entrambi tradotti in italiano ed entrambi reperibili in edizione economica.

2.6 I moduli Perl

Uno dei numerosi punti di forza di perl sono i suoi moduli (raccolti e liberamente disponibili su CPAN, citata a pag. 10): files che contengono funzioni e variabili che potete riutilizzare. Non perdo tempo a spiegarvi qual è la loro utilità: se continuerete a programmare in perl la scoprirete da soli, ma è opportuno segnalarvi che esistono cose simili.

Fate un giro si www.cpan.org prima di buttarvi a capofitto su un nuovo progetto. Probabilmente troverete qualcosa di buona da riciclare e risparmierete tempo e risorse.

[8] Che con il passaggio ai processori Intel potrebbe alla lunga porre problemi di controllo e privacy.

Capitolo 3

Studiare il nemico

3.1 L'anima della lingua

Ripetete dieci volte: "un linguaggio di programmazione è una lingua che devo usare per parlare con il mio computer". Fatelo ad alta voce, anche se siete in un luogo pubblico, anzi: l'imbarazzo stimola la produzione di ormoni che aiutano a imprimere bene nella memoria certi dettagli.

Il segreto per non lasciarsi intimorire da un linguaggio di programmazione è tutto in quella formula, perché come umanisti avrete studiato lingue infinitamente più complesse: il perl, al confronto, è un'idiozia. Non ci credete?

Pensate a questo:

- nei linguaggi di programmazione ci sono due modi, l'indicativo e l'imperativo; inoltre non esiste una flessione verbale: vengono gestite solo le radici (che sono quasi tutte in inglese americano);
- la morfologia nominale non è ambigua, avete sostanzialmente tre morfemi monografematici che indicano quello che devono indicare: se ci sono, significano una cosa e solo quella; se non ci sono significano altro;
- anche se in perl una cosa può essere detta in più modi; si tratta nella sostanza di modi più espliciti, analitici, descrittivi oppure di modi più impliciti, sintetici, performativi – oppure di perifrasi sinonimiche – ma in perl non piovono cani e gatti e non ci sono altre espressioni idiomatiche;
- ogni proposizione si conclude con un punto e virgola e in perl standard praticamente tutte le subordinate sono racchiuse in diversi tipi di parentesi, che significa: rarissime ambiguità sintattiche;
- avete un dizionario di un centinaio di parole con il quale potete dire quasi tutto quello che vi interessa. Quante ne servono in inglese?

- quello che scrivete non viene letto direttamente dalla macchina, ma da un interprete perl che le traduce in codice binario da eseguire sul momento. E l'interprete vi avvisa quando fate un errore: non vi capiterà mai di scrivere un testo, farvi capire con qualche madornale errore grammaticale e ottenere risposta con un sorrisetto di superiorità.

3.2 L'interprete (con chi userete la lingua)

Anche se non è notevole quanto Nicole Kidman nel film intitolato come questo paragrafo, l'interprete perl non può essere ignorato, anche perché rappresenta l'unico vero interlocutore del programmatore.

Potete immaginare quello che avviene in questo modo: il signor Stefano Lavori[1] è un programmatore perl che scrive dei piccoli programmi per svolgere più in fretta dei semplici compiti con il computer, come rinominare automaticamente dei files o fare delle ricerche su certi documenti. Di lavoro fa qualcosa di molto più utile e divertente, come il pizzaiolo o il pilota di formula uno, ma nel tempo libero programma ed usa i propri programmi.

Stefano Lavori scrive i propri programmi, che sono liste di istruzioni in lingua perl, e li invia a Nicole Kidman. Nicole legge il testo scritto da Stefano, lo trasforma in codice binario e lo invia alla macchina. La macchina legge il codice binario e lo esegue.

Come **programmatore** Stefano deve tenere ben presente la funzione di Nicole, anche se non la vede né incontra mai direttamente (eccetto quando verifica che i propri programmi funzionino, prima di usarli per davvero, ma si tratta di incontri molto formali e veloci, che non lasciano tempo per alcuna forma di corteggiamento).
D'altronde, in veste di **utente** dei propri programmi, Stefano chiama Nicole e richiede le sue prestazioni di interprete, ma ancora una volta si tratta di fugaci incontri durante i quali i due non hanno neppure il tempo di guardarsi negli occhi, perché per la maggior parte del tempo Stefano interagisce in realtà con la macchina, e non con Nicole.

E alla fine la bella Nicole esce sempre con qualcun altro – fino a qualche tempo fa sapevamo trattarsi di Tommaso Crociera. Peccato, a Stefano sarebbe piaciuta una cena fuori.

3.3 Verbi

3.3.1 Il verbo attaccapanni

Per introdurre i verbi del perl, inizierò col raccontarvi qualcosa dei verbi in generale, e per farlo rievocherò la figura di un linguista francese del secolo scorso (1893-1954): Lucien Tesnière.

[1] O Guglielmo Cancelli, se preferite.

A scanso di equivoci va detto che Tesnière e l'informatica sono elementi di due universi paralleli e non hanno mai avuto a che fare l'uno con l'altra, cito apposta questo umanista per non essere accusato di far passare sottobanco figure di loschi e ambigui linguisti e filosofi del linguaggio – come Bertrand Russel – che maneggiavano disinvoltamente anche la matematica[2].

Per il linguista Lucien Tesnière il centro della frase, ciò a cui tutto il resto generalmente si appoggia, è il verbo[3].

Tesnière diceva: guardate il verbo *piovere*: se ne sta da solo, e non può stare vicino a nient'altro, eccetto qualche informazione sulle circostanze in cui piove (oggi, a Vicenza) o sui modi (per tre ore, a catinelle).

Ma prendete ad esempio il verbo *parlare*: questo verbo "chiede" di essere completato da almeno un'informazione: chi parla? Anche tutte le informazioni **circostanziali** sono possibili, ma il soggetto della frase è in qualche misura necessario al completamento dell'espressione di un significato da parte del verbo.

E il verbo *mangiare*?
Due **argomenti** gli sono indispensabili: chi mangia e cosa mangia. Gli *optional*, i circostanziali, sono a parte.

Esistono verbi con tre argomenti? Certo: *dare*: qualcuno dà qualcosa a qualcun altro.

E con quattro argomenti? Qualcuno *sposta* qualcosa da un punto ad un altro punto.

E con cinque? Con cinque no, quattro è il limite concesso alla lingua italiana.

Il numero di argomenti che un verbo può/deve reggere è chiamato da Tesnière **valenza**[4].

Badate che quando non tutte le valenze di un verbo sono saturate, spesso il verbo cambia un po' significato. Per esempio il verbo *dare*, trivalente, con due soli argomenti – soggetto e complemento oggetto– può significare [produrre], come nella frase: *il melo non ha dato i suoi frutti*. E i verbi *andare* e *venire*, divalenti, hanno significati diversi se sono usati come monovalenti[5].

Potete dare un'occhiata anche solo a [SC05], per avere qualche conferma.

[2]Per quanto proprio questi altri linguisti siano invece stati utili all'informatica, anche riproponendo il concetto di struttura argomentale e poi di calcolo proposizionale. Se vi interessano i linguisti loschi, in odor di scienze dure, un libro bellissimo, curato dal filosofo del linguaggio Bonomi, è [Bon73].

[3]Cfr. [([Tes59]) 2001:32].

[4]Il termine è mutuato dalla chimica, nella quale le valenze descrivono il fatto che un atomo di ossigeno con valenza 2 si può legare a due atomi di idrogeno con valenza 1 ciascuno e dare origine ad una molecola di acqua. La *valenza* – chimica e linguistica – ha un corrispettivo informatico: nel ProLog si chiama *arity*.

[5]I verbi di movimento, in quanto intransitivi, sono generalmente classificati come monovalenti, con il soggetto come argomento unico. Se però valutiamo la differenza tra gli enunciati: *Giangiacomo è andato* (senza argomenti sottintesi) e *Giangiacomo è andato via*, risulta evidente che l'avverbio nel secondo esempio è un argomento e non un semplice circostanziale. Lo stesso discorso vale per: *Pierluca è venuto* e *Pierluca è venuto qui*.

3.3.2 Gli argomenti del verbo perl

Anche il perl ha dei verbi. Ma, poiché i creatori di perl hanno più in comune con i matematici che con i linguisti, hanno preferito chiamare i verbi **funzioni**.

Quel che interessa a noi è che anche le funzioni del perl hanno degli argomenti; a differenza dei verbi italiani, però, non cambiano significato, neanche un poco, quando non sono presenti tutti gli argomenti.

Prendiamo il verbo `print()` [stampa]. Come è logico che sia, questa funzione regge due argomenti: cosa stampare e dove (ma non chi: la macchina fa tutto, quindi si dà per scontato che sia sempre la macchina il soggetto dei verbi. Anche questa è una semplificazione). È meno intuitivo l'ordine di questi argomenti: nel perl si dice prima dove stampare e poi cosa.

Scriviamo il nostro primo programma.

Aprite un qualsiasi editor di testo (rigorosamente blocco note se lavorate su Win[6]; potete scegliere invece tra nvi, vi, vim, elvis, emacs, textedit, bbedit se lavorate su X), e scrivete:

```
print(STDOUT "ecce script!");
```

osservate la struttura del listato:

verbo(argomenti);

Il punto e virgola serve ad esplicitare il fatto che la proposizione finisce lì. Tra le parentesi tonde ci sono due argomenti. Il primo è una parola speciale, *STDOUT*, che significa [standard output], il secondo è un messaggio tra virgolette.

Lo standard output per il perl è la finestra di terminale o di DOS.
Come vedremo, con questo argomento si passa all'interprete perl (che trasforma il vostro messaggio in una sequenza di uno e zero che il computer possa eseguire) la conferma che deve stampare il messaggio proprio nello standard output.

Quanto alle virgolette, va fatta una considerazione utile. Nelle lingue storico naturali che possiedono una forma scritta, le virgolette hanno una precisa funzione di esplicitamento di slittamento testuale (sono degli swichatori di livello testuale): servono per forzare una interpretazione non letterale – un tipo "simpatico" è al contrario insopportabile – oppure per aprire un testo nel testo, come nel discorso riportato.

Nel perl è quasi la stessa cosa: con l'apertura delle virgolette si dice all'interprete: "guarda che quel che segue non è scritto in perl, è scritto in un'altra lingua e tu non devi interessartene, riproducilo e basta". Cioè: qui avviene uno slittamento di livello testuale. E infatti, se noi scrivessimo:

[6] Ovviamente, potete lavorare anche con Word, o con Wordpad, salvando come solo testo, ma è più macchinoso e ci sono più tentazioni.

```
print(STDOUT "print(STDOUT ecce script!)");
```

il programma scriverebbe semplicemente:

print(STDOUT ecce script!)

nel suo standard output (su questo argomento torneremo sia a pag. 23 che a pag 37).

Ma magari tutte queste informazioni per ora vi paiono inutili e volete vedere all'opera il perl. Allora salvate il file con la riga di testo che abbiamo appena scritto come prova1.pl e in formato solo testo[7].

Poi aprite il DOS o la shell, e digitate
`$> perl prova1.pl` [invio]

Funziona? Dovrebbe. Se non funziona assicuratevi di non aver dimenticato il punto e virgola. Se non l'avete dimenticato, controllate di essere nella stessa cartella di prova1.pl. Se ci siete, verificate di aver salvato il file con estensione `pl` e in formato solo testo. Se avete fatto anche questo digitate nella shell (o prompt di DOS, d'ora in poi per mia comodità mi riferirò a questi due referenti sempre con il nome *shell*):
`$> perl -v` [invio]
Se ottenete un avvertimento di errore non avete installato correttamente perl: bisogna ricominciare da lì. Se invece vi risponde con la versione corrente di perl, ricontrollate che il vostro script sia identico al mio.

Ricordatevi che gli utenti di X possono abbreviare il processo. Devono sapere dove si trova l'interprete perl[8], e poi far iniziare tutti i loro programmi con la linea: `#!/usr/bin/perl`, o l'indirizzo dell'interprete perl che è stato loro rivelato dalla shell.

Non so da dove derivi, ma a me serve per non dimenticarmi il dettaglio: il cancelletto e il punto esclamativo si chiamano, in gergo, *she bang*.

Se il programma inizia con quella linea, poi vi basterà digitare:
`$> ./prova1.pl` [invio]

[7] Non dimenticate che in X non è l'estensione del file che ne definisce il funzionamento, ma le sue caratteristiche implicite, quindi dovrete rendere eseguibile il file con il comando:
`$> chmod +x prova1.pl` [invio]
Può sembrare noioso e lungo, ma è sicuro. D'altronde, se su X non esistono praticamente virus ci saranno dei motivi, no?

[8] Basta chiederlo alla shell:
`$> which perl` [invio]
e la risposta sarà qualcosa tipo /usr/bin/perl o /usr/local/bin/perl.

Argomenti inespressi

Perché dover dire all'interprete perl che deve stampare il nostro testo nello standard output, dove stamperebbe comunque? Infatti possiamo non dirglielo: il nostro primo programma avrebbe potuto avere questa forma:

```
print("ecce script!");
```

E, in realtà, neppure le parentesi sono necessarie, servono a me e a voi, ma non sono indispensabili per l'interprete, il quale ha ben chiaro il fatto che il verbo print regge due argomenti seguiti dalla chiusura della proposizione con un punto e virgola. Quindi il nostro script avrebbe potuto avere anche queste forme:

```
print STDOUT "ecce script!";
print "ecce script!";
```

E non ci sarebbero stati fraintendimenti di sorta.

Ma non fate l'errore di ritenere sempre le parentesi inutili: esse rivestono infatti la fondamentale funzione di rendere il testo più leggibile, soprattutto quando ad una funzione sono **passati** numerosi argomenti (immaginate il caso di un ordine di stampa che preveda come argomento/complemento oggetto una stringa di cento parole: avere una parentesi chiusa ad identificarne la stringa può essere utile).

E non fate mai l'errore di pensare che anche il punto e virgola sia opzionale: è obbligatorio. Esiste una versione alternativa delle Tavole della Legge che impone (cito testualmente): "sesto: non dimenticare il punto e virgola"[9].

Siete avvisati.

3.4 Nomi e/o pronomi

La natura dei sostantivi dei quali ci occuperemo noi è... originale. Perché i nostri sostantivi hanno qualcosa dei pronomi: un nome in perl non ha quasi mai un referente preesistente: nel momento in cui viene creato il nome, di solito, viene anche creato il suo referente[10].

Questa coincidenza ha delle conseguenze, la più importante delle quali è che il nome delle cose che creiamo lo scegliamo noi. E la stessa cosa, in testi/programmi diversi o scritti da persone diverse, potrebbe avere nomi diversi. Per questo, i nomi si chiamano **variabili**.

Esistono diversi tipi di variabile (be'... tre), ma noi per ora ci occuperemo del più semplice: le **variabili non strutturate**. Una variabile non strutturata può essere quasi qualsiasi cosa: una parola, una frase, un testo di quaranta milioni di parole, un numero.

[9] Si tratta della versione non bacchettona.

[10] NOTA CHE GENERA CONFUSIONE: a volte viene prima generato il nome, e solo in un secondo momento il suo referente; altre volte, addirittura, viene generato ed impiegato un nome che ha molti referenti diversi nessuno dei quali direttamente creato dal programmatore.

Proviamo a modificare il nostro primo listato[11]:

```
$testo = "ecce script!";
print $testo;
```

Osservate con attenzione il primo morfema[12] che incontriamo: il segno del dollaro indica che la parola attaccata è un nome di variabile.
Il nome di variabile può avere quasi qualsiasi forma, ma è saggio che sia costituito solo da lettere dell'alfabeto o da numeri, che inizi con una lettera minuscola e che non sia troppo lungo né troppo breve. Inoltre non deve includere degli spazi o degli altri segni di dollaro (logico, no?)[13].

Torniamo allo script: segno del dollaro più nome della variabile, il segno di uguaglianza (=) può essere tranquillamente interpretato come [è uguale a], poi c'è il referente della nostra parola *$testo*, cioè la stringa *ecce script* tra virgolette; infine il punto e virgola.

Dopo aver definito una stringa di testo (*ecce script!*) ed averle attribuito un nome (la variabile *$testo*), posso aggiungere l'ordine di stampare `$testo`.

Tutto chiaro?
Se qualcosa vi sfugge nella mia spiegazione, provate a fare esperimenti: la pratica a volte è illuminante.

È il momento di un altro esempio: copiatelo così com'è e mandatelo in esecuzione come il primo, osservate l'output e cercate di capire che cosa fa ogni linea; dopo lo commentiamo.

```
$ciccio = "parolona";
$num = 3;
print "ecco la stampa di una stringa: ".$ciccio."\n";
print "ecco la stampa di un numero: $num \n";
$somma = $num + 1;
print "stampa la somma del numero $num piu' 1 : $somma\n";
$gatto = $ciccio;
print "stampa della variabile \$gatto $gatto\n";
$num ++;
print "nuovo numero $num\n";
$num += 5;
print "variabile \$num sommato a 5 = $num\n\n";
```

[11] Listato, script, programma... per chi non l'avesse ancora capito li uso come se fossero sinonimi perfetti. E per voi non farà alcuna differenza.

[12] Il morfema è l'unità minima di senso in una lingua. Per esempio la parola *programma* è costituita da due morfemi: uno lessicale (*programm-*) che contiene il significato di [insieme di istruzioni] e l'altro grammaticale, o flessionale, (*-a*) che in questo caso porta i "significati" di [maschile, singolare]. La sinossi linguistica è un po' rozza, ma nella sostanza corretta. Perdiana, se non sapete queste cose che umanisti siete?

[13] Sulle convenzioni battesimali delle variabili torneremo comunque a pag. 43.

L'output di questo listato dovrebbe essere grosso modo:

ecco la stampa di una stringa: parolona
ecco la stampa di un numero: 3
stampa la somma del numero 3 piu' 1 : 4
stampa della variabile $gatto parolona
nuovo numero 4
variabile $num sommato a 5 = 9

È uguale? Beene.

Commenti 1: variabili

Ecco il codice dell'ultimo listato, numerato per comodità.

```
1  $ciccio = "parolona";
2  $num = 3;
3  print "ecco la stampa di una stringa: ".$ciccio."\n";
4  print "ecco la stampa di un numero: $num \n";
5  $somma = $num + 1;
6  print "stampa la somma del numero $num piu' 1 $somma\n";
7  $gatto = $ciccio;
8  print "stampa della variabile \$gatto $gatto\n";
9  $num ++;
10 print "nuovo numero $num\n";
11 $num += 5;
12 print "variabile \$num sommato a 5 = $num\n\n";
```

Ho introdotto alcune novità, ma confido che non ci daranno problemi. Le prime due linee dovrebbero essere perfettamente comprensibili: abbiamo **istanziato** (cioè creato) delle variabili, una stringa e un numero e nelle linee successive le abbiamo stampate.

Ma nella riga 3 compare un nuovo segno di interpunzione, il punto. Il punto perl ha la funzione opposta del punto nelle lingue storico-naturali: invece di dividere unisce. È un segno di **concatenazione** di stringhe. Il codice:

```
"ecco la stampa di una stringa: ".\$ciccio."\n"
```

è uguale a:

```
"ecco la stampa di una stringa: "."parolona"."\n"
```

perché, come è scritto nella prima linea di codice, $ciccio è uguale a "parolona".

In:

```
"ecco la stampa di una stringa: "."parolona"."\n"
```

leggiamo tre diverse stringhe concatenate, giustapposte, che verranno stampate una di seguito all'altra senza soluzione di continuità.

Commenti 2: l'insostenibile invisibilità di \n

Ma c'è un'altra novità alla terza linea e nelle ultime tre frasi d'esempio: un carattere invisibile. Non proprio invisibile, o, meglio, non invisibile nello script ma che diventa invisibile una volta stampato. Provate a confrontare il listato con il suo output presentato nel paragrafo precedente. Lo vedete? Esatto: è la stringa $\backslash n$. Ma se è una stringa, perché l'ho chiamata *carattere*?

Bene, se state leggendo il presente testo siete abbastanza grandi per un'orrenda verità: nel computer, anche lo spazio è un carattere. E anche l'a capo. E pure la tabulazione.

Anche se non li vedete, nel file di testo che leggete gli spazi, le interruzioni di linea, le tabulazioni sono codificate: si tratta di indicazioni che spiegano alla macchina – cioè, al programma che legge il documento e lo mostra sullo schermo – che deve visualizzare uno spazio, incominciare la nuova linea oppure mostrare la parola che segue dopo un certo intervallo di spazi.

Backslash-enne significa [nuova linea], [new line].
Stampando la stringa "ecco la stampa di un numero: $num \n" scrivo tutto il testo che serve e, alla fine, aggiungo un a capo. L'a capo, nell'output lo vedo, ma non è codificato, è realizzato, quindi non mi rendo conto del fatto che è un carattere codificato (da una stringa di due caratteri) come qualsiasi altro.

Anche lo spazio è così. Lo spazio si scrive $\backslash s$, backslash-esse. E la mia stringa potrebbe essere:

```
"ecco\sla\sstampa\sdi\sun\snumero:\s\$num\s\n"
```

senza che il carattere di spaziatura sia visibile in quanto tale nell'output[14].
Ora provate a modificare il listato, cambiate la stringa in questo modo:

```
"ecco\nla\nstampa\ndi\nun\nnumero:\n\$num\n"
```

Guardate che cosa succede.

Magia!

Commenti 3: le barre retroverse non sono contro natura

Ancora un dettaglio sui testi: alle linee 8 e 12 dei segni di dollaro sono preceduti da dei backslash (tasto [\] in alto a sinistra). Perché? Anche questi funzionano come dei segni di slittamento testuale.

[14] Ehm, ehm... per motivi al momento ignoti, su alcune versioni di Windows questo carattere, tra virgolette, non funziona. Il motivo non è facilmente intuibile (con il `\n` funziona!).

Le virgolette, abbiamo detto[15], sono degli switchatori testuali. In realtà accettano almeno un paio di eccezioni: la prima è quella delle sequenze precedute dalla barra retroversa – questo il nome italiano del backslash –, \n, \s, \t eccetera; la seconda è quella delle variabili.

Una variabile sarà trascritta non come nome di variabile ma come valore di variabile, referente del nome; per questo la linea 4 restituisce come output:

ecco la stampa di un numero: 4

e non

ecco la stampa di un numero: $num

Che cosa devo fare, allora, se desidero che invece la variabile compaia nell'output come nome di variabile e non come valore di variabile? Faccio precedere il segno del dollaro da una barra retroversa, il cui significato è: "il carattere che segue va interpretato letteralmente". L'interprete perl, a questo punto, sa che non deve più ammettere eccezioni, che quel segno di dollaro va interpretato come segno di dollaro e non come morfema nominale, il resto del nome di variabile diventa per lui una parola qualsiasi, e non è più una variabile perché non è preceduta da un simbolo di variabile.

Affascinante, non trovate?

Commenti 4: agire sui numeri

Adesso concentriamoci invece su alcune manipolazioni prettamente numeriche che compaiono alle linee 5, 9 e 11. Ecco, le riporto insieme con le linee che ordinano la produzione dell'output così non dovete andare a cercarvele indietro:

```
5  $somma = $num + 1;
6  print "stampa la somma del numero $num piu' 1 $somma\n";
9  $num ++;
10 print "nuovo numero $num\n";
11 $num += 5;
12 print "variabile \$num sommato a 5 = $num\n\n";
```

La linea 5 è di facile interpretazione: la variabile `$num` prima aveva un valore (3, la variabile è stata istanziata alla linea 2), nella linea in esame questo valore viene usato come addendo. Viene creata una nuova variabile, `$somma`, il cui valore è uguale alla somma di `$num` più uno, come potete vedere dall'output generato dalla linea 6[16].

[15] Ma ritorneremo sulla questione in maniera più approfondita a pag. 37.

[16] Sforzatevi di capire qual è l'output della linea 6, rilanciando lo script e controllando.

La somma che avete visto è una delle tre possibili formulazioni di una somma in perl. Cioè, potete dire all'interprete perl di aggiungere una quantità ad un'altra quantità (immagazzinata in una variabile) in tre diversi modi.

Il primo (quello della linea 5) è il più semplice. Tuttavia, quando si tratta di aggiungere una unità, è il meno usato. Esiste una scorciatoia, che potete leggere alla linea 9: `$num++;` è uguale in tutto e per tutto a `$num = $num + 1;`

Questa scorciatoia non può essere imboccata quando si aggiunge più di una unità al valore della variabile (ad esempio quando si aggiunge 5 o 100.000). In questi casi è necessario imboccare una scorciatoia meno rapida, che potete vedere alla linea 11. `$num += 5;` è perfettamente uguale a `$num = $num + 5`. Potete leggere la coppia di segni `+=` come un "aggiungi a quel che c'è", se vi aiuta a ricordarlo.

Commenti 6: perché?

A questo punto i miei studenti reagivano più o meno invariabilmente con una certa confusione: perché dobbiamo sapere questa cosa? A cosa serve? A cosa servono le variabili?

Allora diventavo io confuso: quando studiano che in inglese *utterance* significa [enunciato], si chiedono perché lo devono studiare o lo studiano e basta? Ma in fondo anche loro avevano le loro ragioni: per quanto rara nel linguaggio comune una parola può sempre capitare di usarla, e quindi vale la pena di conoscerla. Senza contare che nella maggior parte dei casi impariamo nuove parole traducendo, o ascoltando, e quindi la loro utilità è immediata, oltre che immediatamente percepibile.

Il salto concettuale da fare a questo punto – ma si tratta del più importante e più delicato, fatto il quale tutti gli altri verranno come necessarie conseguenze – è che il perl non è solo costituito da ordini scritti impartiti ad un calcolatore, o meglio: gli ordini che possiamo impartire richiedono una articolazione interna che generalmente non è necessaria nelle lingue storico-naturali.

Non possiamo dire ad una macchina: "fai questo lavoro 3 volte". Non possiamo farlo in questo modo. Abbiamo bisogno di dire alla macchina:

1. devi iniziare (quindi non hai svolto il tuo lavoro neanche una volta);
2. hai già svolto il tuo lavoro 3 volte? Se sì, smetti; altrimenti prosegui;
3. svolgi il tuo lavoro;
4. ricordati che hai svolto il tuo lavoro una volta più di prima;
5. torna al punto 2;

Il punto quattro, in perl potrebbe essere scritto in questo modo:

```
$volte++;
```

È anche più breve che in italiano.

Si verifica un bizzarro fenomeno: nei linguaggi di programmazione in generale, e quindi ovviamente anche in perl, abbiamo bisogno di dire più cose, esplicitare un maggior numero di dettagli rispetto a quanto è utile fare parlando o scrivendo[17], ma lo facciamo con una quantità di materia verbale notevolmente minore rispetto a quella necessaria per dire o scrivere le stesse cose in una lingua storico-naturale.

3.4.1 Tipi di variabili

Ho usato qualche volta, fino ad ora, il termine **stringa** per riferirmi ad una non meglio precisata sequenza di caratteri[18], ed ho detto che, insieme ai numeri, questo tipo di "materia informatica" costituisce la classe delle variabili non strutturate, delle variabili, cioè, prive di struttura interna.

Esistono delle variabili strutturate. Pensateci: come potrebbero essere fatte?

Ci state davvero pensando?

Bravi!, bravi, (è bello avere lettori intelligenti), esatto: come elenchi.

[17] Ma vi invito a valutare le differenze tra i due modi di comunicare. Anche in italiano nello scritto – causa la mancata condivisione di un contesto d'enunciazione comune e l'impossibilità di ricorrere ad ostensioni e deittici – dobbiamo comunicare più informazione di quella che sarebbe sufficiente nell'orale.

[18] Ma un testo scritto in lingua storico-naturale è una sequenza di caratteri, alfabetici come quelli che compongono le parole, e non alfabetici come la punteggiatura, anche se non tutte le sequenze di questi tipi di caratteri sono testi.

Esistono due tipi di variabili strutturate: gli array e gli hash.

Gli array

Gli **array** sono delle liste ordinate: serie di elementi che l'interprete sa essere in ordine numerato a partire da zero. Gli array hanno un loro morfema identificativo: `@`, la chiocciolina, [at].

Un array potrebbe quindi avere la seguente forma[19]:

```
@arrayDiEsempio = ("Frodo", "Merry", "Pipino", "Sam");
```

in cui, come vedete, viene creato l'arrayDiEsempio come una lista di quattro stringhe tra virgolette distinte da una virgola. Leggendo la chiocciolina, l'interprete perl sa che il primo elemento è Frodo, il secondo Merry e così via, e questo, se ci pensate, rende la lista manipolabile, ben più di una sequenza di nomi dentro una stringa (per esempio, posso dirgli di stamparmi il terzo nome).

Una domanda viene naturale[20]: dentro un array ci sono delle stringhe, ma l'array può includere anche variabili? Certo!

Per esempio potreste avere una formulazione del tipo:

```
$baggins="Frodo";
@arrayDiEsempio = ($baggins, "Merry", "Pipino", "Sam");
```

Addirittura, visto che un array può includere una qualsiasi variabile, può anche includere un altro array:

```
$baggins="Frodo";
@aggiuntiDopo = ("Merry", "Pipino");
@arrayDiEsempio = ($baggins, @aggiuntiDopo, "Sam");
```

È il momento di qualche nuovo script, che chiarirà quel che intendevo dire nel paragrafo precedente sul perché è utile conoscere i trucchi di manipolazione dei numeri.

Posso usare gli array in tre modi diversi:

1. posso fare riferimento diretto all'array, con un programma del tipo:

   ```
   $baggins="Frodo";
   @arrayDiEsempio = ($baggins, "Merry", "Pipino", "Sam");
   print @arrayDiEsempio;
   ```

[19] Avete letto "Il Signore degli Anelli"? Guardate che l'abominevole film non conta.

[20] Come mi è stato suggerito, forse solo a persone affette da gravi disturbi della personalità.

in questo modo, però, l'output risultante sarà una sola parola costituita dai quattro elementi dell'array incollati:

FrodoMerryPipinoSam

Per avere gli elementi divisi, devo fare qualcosa di più raffinato, per esempio selezionare ogni singolo elemento e concatenarlo ad un carattere di nuova linea.

2. si fa così: per fare riferimento ad ogni singolo elemento dell'array, devo ricorrere a questa struttura morfologica: `$nomearray[numero d'ordine]`. Prestate attenzione al fatto che se faccio riferimento ad un singolo elemento dell'array, uso il morfema delle variabili non strutturate, il segno del dollaro. Tra parentesi quadre segue il numero d'ordine, a partire da zero. Per esempio potrei scrivere un programma con questa forma:

```
$baggins="Frodo";
@arrayDiEsempio = ($baggins, "Merry", "Pipino", "Sam");
print $arrayDiEsempio[0]."\n";
print $arrayDiEsempio[1]."\n";
print $arrayDiEsempio[2]."\n";
print $arrayDiEsempio[3]."\n";
```

Copiato e lanciato?

L'output è già meglio, no? Si può fare ancora meglio. Vedremo tra poco come, introducendo una struttura di controllo; per ora accontentatevi.

3. l'ultimo modo in cui posso fare riferimento ad un array, è come se fosse una variabile non strutturata, in tal caso l'interprete perl mi restituisce il numero degli elementi dell'array. Per esempio:

```
$baggins="Frodo";
@arrayDiEsempio=($baggins, "Merry", "Pipino", "Sam");
$quantiHobbit = @arrayDiEsempio;
print "Il numero degli hobbit nella Compagnia
dell'Anello e': $quantihobbit \n";
```

Controllate l'output.

A conclusione di questo ricco paragrafo sugli array, anticiperò la struttura di controllo: *for* [per].

Cercate di capire la logica di questa struttura di controllo, che è la più difficile nel perl, ma non preoccupatevi troppo se qualche dettaglio vi sfugge, perché alle strutture di controllo dedicherò molto più spazio nel prossimo capitolo.

Tornate per un attimo al penultimo script che abbiamo visto, quello in cui gli elementi della lista degli hobbit che fecero parte della Compagnia dell'Anello vengono richiamati uno per uno e stampati. Considerate anche il fatto che quella lista è piccola (anche considerando tutti i membri della Compagnia, non siamo che a nove), ma esistono liste molto più lunghe e ne esistono alcune, addirittura, delle quali i programmatori non conoscono il numero né l'identità degli elementi; è il caso, ad esempio, di una lista di tutte le parole di un testo, che viene creata automaticamente da poche righe di codice.

Ad ogni modo, anche questa semplice lista di quattro elementi può essere riprodotta nell'output ordinatamente e in modo più agevole[21]. Osservate il codice, poi lo commentiamo insieme:

```
$baggins="Frodo";
@arrayDiEsempio = ($baggins, "Merry", "Pipino", "Sam");
$max = @arrayDiEsempio;
    for($volte=0;$volte<$max;$volte++)
    {
    print $arrayDiEsempio[$volte]."\n";
    }
```

Ci interessano le ultime cinque linee. Eccole numerate

```
1 $max = @arrayDiEsempio;
2 for($volte=0;$volte<$max;$volte++)
3    {
4    print $arrayDiEsempio[$volte]."\n";
5    }
```

Alla linea 1 abbiamo istanziato una variabile, `$max`, che corrisponde al numero di volte che dovrà essere stampato il nome di un hobbit, cioè corrisponde al numero degli elementi dell'array (modo 3 di riferimento ad un array, pag. 28).

La linea 2 è quella sintatticamente più complessa: la parola *for* è una parola grammaticale, dovete impararla a memoria così com'è, punto. Di buono c'è che è trasparente. Il testo all'interno delle parentesi può essere tradotto come segue: [avendo lavorato zero volte; finché il numero di volte che hai lavorato è minore di `$max`; aumentando il numero di volte di una unità ogni volta che lavori].

Si tratta di tre proposizioni subordinate.

Si vede che sono subordinate perché sono racchiuse tra parentesi tonde, similmente a quanto accade per gli argomenti di un verbo che in qualche misura gli sono subordinati.

[21] Ho già detto che il maggior pregio di un programmatore è la pigrizia?

Il vincolo delle parentesi tonde è talmente forte, che all'ultima delle tre subordinate non è neppure chiesto il punto e virgola.

Spendiamo qualche parola sulla sintassi, sfruttando l'occasione che la struttura di controllo `for` ci offre.
Ecco due tra le più importanti regole sintattiche del perl (la prima l'abbiamo già vista):

1. usa il punto e virgola alla fine di una proposizione;

2. racchiudi tra parentesi gli insiemi di proposizioni coordinate. In generale, le parentesi graffe racchiudono le proposizioni principali, le tonde le proposizioni subordinate. Le proposizioni principali di livello più alto, quelle che non sono sottoposte ad alcuna struttura di controllo, possono comparire senza parentesi graffe[22].

La seconda regola è più importante della prima[23]; infatti, nelle situazioni non ambigue (con una sola proposizione) se si omette il punto e virgola non viene segnalato l'errore. Come accade in:

```
{print "ciao\n"}
```

Tenete ben presenti queste due regole, perché ci accompagneranno per il resto del manuale ed anche oltre, se inizierete a programmare autonomamente.

Ed ora torniamo alla nostra struttura di controllo e alle tre subordinate.

La prima definisce una situazione iniziale: il numero di volte che sono state eseguite le istruzioni scritte tra parentesi è zero; in latri termini: `$volte = 0;`
La seconda definisce un limite: le istruzioni scritte tra parentesi graffe devono essere eseguite fino a quando il valore della variabile `$volte` è inferiore al valore della variabile `$max`.
La terza definisce un'istruzione di mutamento tra la condizione iniziale e quella finale: perché `$volte` inizialmente uguale a zero diventi almeno uguale a `$max`, ad ogni esecuzione delle istruzioni tra le parentesi graffe il valore di `$volte` deve aumentare di uno; detto altrimenti: `$volte++` oppure: `$volte=$volte+1` (si tratta di due forme perfettamente sinonimiche, come abbiamo visto).

Allora, la sintassi del ciclo `for` è chiara? Se lo è a grandi linee proseguite pure, altrimenti provate a rileggere eseguendo con attenzione tutto il codice – o, se preferite, provate a leggere una descrizione più dettagliata di questo tipo di ciclo a pagina 62 –.

Prima di concludere voglio ancora segnalarvi il fatto che la variabile `$volte`, alla linea 2 e alla linea 4 cambia il proprio valore piuttosto rapidamente nel giro di poche frazioni di

[22] E infatti noi non abbiamo mai scritto le parentesi graffe all'esterno dei cicli `for`. Ma se provate a modificare i vecchi script aggiungendo parentesi graffe all'inizio e alla fine dei listati, vedrete che continuano a funzionare correttamente, mentre se aggiungete parentesi tonde avrete la prima esperienza di messaggi di errore da parte dell'interprete perl.

[23] È anche vero che neppure la seconda regola è esente da eccezioni: quando vengono creati degli array, variabili che contengono liste, tali liste sono comunque racchiuse tra parentesi tonde.

secondo, ma questo è il vantaggio di lavorare con le variabili: se si riesce a controllare il loro mutare, possono rivelarsi molto produttive, infatti il codice eseguito all'interno della struttura `for` è in tutto e per tutto una sequenza di quattro chiamate individuali ad altrettanti elementi dell'array.

Ma anche questo concetto sarà più chiaro procedendo.

In the mood for Perl

Ora attenzione, seguitemi bene perché quel che sto per spiegare sembrerà complicato e inutile, ma è molto importante per entrare nell'ottica dell'informatico, capire cosa è veramente importante nella programmazione.

Il linguaggio Python deve il proprio nome al fatto che il suo autore, Guido van Rossum, ama molto i Monty Python.
I Monty Python, in un loro spettacolo, ripetono alla nausea la parola *spam* ed è grazie a quell'uso che tale parola ha assunto anche il significato di [massa di inutili dati inviati per posta elettronica].
Ma il significato originario della parola ha a che fare con della carne di maiale in latta.
Un'altra parola che originariamente designava della carne, nella fattispecie tritata, e che in seguito è stata impiegata in ambito informatico – seppure non con la stessa fortuna della parola *spam* – è *hash*.
Di *hash* (non nell'accezione alimentare) ci occuperemo in questo paragrafo ma, ed ecco un motivo di profondo sgomento, nonostante tutte queste mirabolanti coincidenze nel linguaggio Python gli hash non si chiamano *hash* ma *dizionari*.

Dico: proprio nel Python.

È semplicemente incredibile.

Eppure esiste una ragione per chiamare gli hash *dizionari*[24]: un hash è una lista non ordinata, ma nella quale ad ogni elemento della lista è associato un nome. Immaginate appunto un dizionario in cui ogni elemento della lista è una definizione (o un traducente) a cui sia associato un nome (la parola definita o il lemma in L1).

L'hash ha un proprio morfema: il segno percentuale. Un semplice hash ha questa forma:

```
%eroiBonelliani = ("Tex" => "Willer", "Dylan" => "Dog",
"Martin"=>"Mystere", "Nick"=>"Raider", "Nathan"=>"Never");
```

Ovviamente, quando bisogna elicitare il valore di un elemento dell'hash, è necessario segnalare anche la parola chiave che lo definisce, per esempio il comando:

```
print $eroiBonelliani{Martin}."\n";
```

[24] Come accade nel Python. Incredibile.

restituirà l'output:

$> Mystere

Come è evidente, se si richiede la stampa di tutti gli elementi dell'hash, quel che c'è da scrivere è infinitamente più lungo di quanto non accadeva per l'array (tanto che io, per generare il seguente script, mi sono scritto un listato che lo facesse):

```
%eroiBonelliani = ("Tex" => "Willer", "Dylan" => "Dog",
"Martin"=>"Mystere", "Nick"=>"Raider", "Nathan"=>"Never");
print $eroiBonelliani{Tex}."\n";
print $eroiBonelliani{Dylan}."\n";
print $eroiBonelliani{Nick}."\n";
print $eroiBonelliani{Nathan}."\n";
print $eroiBonelliani{Martin}."\n";
```

Una gran noia, come potete immaginare (anzi, sarebbe educativo trascrivere il listato ed eseguirlo).

E d'altronde, non avendo qui un ordine con delle chiavi numeriche ad identificare ogni elemento, e non potendo quindi usare usare degli **incrementatori** con `++`, non è possibile usare una struttura di tipo `for`. Useremo, allora, una struttura di controllo che si chiama *foreach* [perogni]. La vedremo in due listati diversi, uno un poco più raffinato e preciso dell'altro.

Il primo foreach su un hash

Visto che la struttura `foreach` è molto più semplice della struttura `for`, iniziamo subito con il listato, dopo lo discuteremo.

Considerate che con questo listato vogliamo una lista di tutti gli elementi dell'hash, cioè di tutti i cognomi dei personaggi che fanno parte dell'hash (mentre i nomi – *Tex, Dylan, Nathan, Nick* – sono le parole chiave che identificano gli elementi, ma non elementi).

```
%eroiBonelliani = ("Tex" => "Willer", "Dylan" => "Dog",
"Martin"=>"Mystere", "Nick"=>"Raider", "Nathan"=>"Never");
foreach $eroe (%eroiBonelliani)
    {
    print $eroe."\n";
    }
```

Naturalmente ci interessano soprattutto le ultime quattro linee. La parola *foreach* è una parola grammaticale, dovete impararla a memoria così com'è, punto. Di buono c'è che è trasparente. Il resto è sintassi, infatti potete tradurre la prima linea in questo modo: [per ogni] [variabile1] [in] [variabile2] [fai...]. Dove:

- il nome di [variabile1] è libero, io ho scritto *$eroe*, ma poteva essere *$puffo* o qualsiasi altra cosa. L'importante è che sia chiaro che questa variabile indica ogni singolo elemento dell'hash (notate che la variabile viene creata prima dei suoi referenti, prima dei valori che può assumere);

- la preposizione [in] è resa dalle parentesi tonde. Mi piace pensare che possiamo immaginare la seconda regola sintattica del perl (quella secondo la quale le proposizioni subordinate dovrebbero essere racchiuse da parentesi tonde) manifestata nella lingua da uno di quei morfemi circonfissi, che stanno sia davanti che dietro le parole cui si attaccano[25], e, in questo caso, ha anche un senso che il circonfisso stia intorno alla parola che denota un hash: è come due mani che tengono l'hash e significano [dentro questo];

- la seconda variabile è l'hash nel quale si pesca. Non mi pare che ci sia altro da dire;

- il quarto elemento, in realtà, non sta sulla stessa linea degli altri tre. È un altro circonfisso: le parentesi graffe. Come per il `for`, tra le parentesi graffe stanno tutte le operazioni sottoposte all'istruzione *foreach*, cioè tutte le operazioni che devono essere ripetute per ogni elemento dell'hash; ma di questo parleremo più avanti.

  Badate anche al fatto che gli spazi sono una faccenda personale: io preferisco, per chiarezza di lettura, andare a capo dopo la chiusura delle parentesi (ma non è obbligatorio) e, all'apertura di ogni nuova parentesi, spostarmi con il tasto [tabulatore] (grosso modo in alto a sinistra sulla tastiera, rappresentato da una freccia verso destra che punta una linea verticale) in modo da isolare i blocchi logicamente connessi – per esempio tutte le proposizioni subordinate ad una determinata struttura di controllo –, ma neppure questo è obbligatorio. Questo listato potrebbe stare scritto tutto su un'unica linea, ma sarebbe d'impiccio per noi utenti umani senza essere un gran vantaggio per l'interprete perl[26].

Eseguite il programma. L'output non va bene: non ci sono errori – infatti il codice viene eseguito correttamente – ma l'output non è quello che volevamo, cioè una lista di tutti gli elementi dell'hash (i cognomi degli eroi bonelliani): in questo output parole chiave ed elementi sono indistinti.

Per ottenere quello che volevamo dobbiamo scrivere un listato più ricco.

[25] SUFfisso: il morfema sta dopo; PREfisso: il morfema sta prima; CIRCONfisso: il morfema sta prima e dopo.

[26] NOTA CHE GENERA CONFUSIONE: nel dettaglio, se lo script fosse scritto tutto su un'unica linea, e senza spazi tra il punto e virgola e l'inizio della proposizione successiva, l'interprete perl potrebbe leggere cinque segni di nuova linea in meno, con un vantaggio nell'esecuzione del programma di una manciata di millesimi di secondo.

Il secondo foreach su hash

Il listato che ci serve è praticamente identico: esistono due sole differenze. Osservate:

```
%eroiBonelliani = ("Tex" => "Willer", "Dylan" => "Dog",
"Martin"=>"Mystere", "Nick"=>"Raider", "Nathan"=>"Never");
foreach $eroe (keys(%eroiBonelliani))
    {
    print $eroiBonelliani{$eroe}."\n";
    }
```

La prima differenza si vede tra le parentesi tonde del `foreach`, dove compare un nuovo verbo: *keys*.

`keys()`, come potete desumere dal fatto che non ha morfemi all'inizio, è una funzione[27]. Come funziona la funzione `keys()`? Moolto semplice: `keys()` accetta un argomento soltanto, che deve essere un hash; prende l'argomento, ne estrae tutte le parole chiave e le immagazzina in un array.

Se immagazzina in un array tutte le chiavi dell'hash, la linea `foreach $eroe (keys(%eroiBonellia` può essere tradotta come:

perogni elemento (dell'array di tutte le parole chiave dell'hash eroiBonelliani).

Notate il dettaglio: l'array con le chiavi di `%eroiBonelliani`, che risulta dall'azione di `keys(%eroiBonelliani)`, io non lo vedo neppure, non so come si chiama e non mi interessa. Ma lo uso. Se mi interessasse, potrei scrivere un listato come:

```
%eroiBonelliani = ("Tex" => "Willer", "Dylan" => "Dog",
"Martin"=>"Mystere", "Nick"=>"Raider", "Nathan"=>"Never");
@chiaviEroi = keys(%eroiBonelliani);
foreach $eroe (@chiaviEroi)
    {
    print $eroiBonelliani{$eroe}."\n";
    }
```

Potrei farlo se desiderassi poter anche avere, ad esempio, il numero degli elementi che fanno parte della lista[28].

[27] In realtà, la forma più regolare del perl prevede, come morfema dei verbi/funzioni, il segno & [e commerciale]. Tuttavia, dato che non sono presenti casi ambigui, e che praticamente nessuna delle funzioni non create dai programmatori è segnalata con la &, accoglieremo la *lectio* che prevede le funzioni senza morfemi distintivi.

[28] ESERCIZIO: fate un bell'esperimento mentale alla Galileo – un *Gedankenexperiment* –: per avere il numero degli elementi della lista, mi basta inserire nel listato una variabile incrementatore (tipo: `$numEroi++;`); riuscite ad immaginare dove? Dopo l'esperimento mentale, fate un po' di esperienza pratica: provate a mettere l'incrementatore dove pensate che potrebbe essere utile e vedete che effetto fa sull'output. Ricordatevi che senza un `print $numEroi;` non si verificherà comunque alcun effetto!

Ma io sono abbastanza pigro, e non vedo perché dovrei fare del lavoro in più.

Ora il programma stampa nell'output quel che volevo, ovvero la lista di tutti i cognomi degli eroi dei fumetti della casa editrice Bonelli.
E se volessi un output che include anche i nomi, magari impaginati/formattati così:

nome: nome; cognome: cognome

cosa dovrei fare?
Riscrivere il listato in questo modo:

```
%eroiBonelliani = ("Tex" => "Willer", "Dylan" => "Dog",
"Martin"=>"Mystere", "Nick"=>"Raider", "Nathan"=>"Never");
foreach $eroe (keys(%eroiBonelliani))
    {
    print "nome: $eroe; cognome: $eroiBonelliani{$eroe}\n";
    }
```

In cui la variabile `$eroe` corrisponde alle parole chiave dell'hash, e quindi ai nomi dei personaggi, mentre la variabile `$eroiBonelliani{$eroe}` è uguale al cognome corrispondente.

Io non ve lo spiego oltre, dovreste aver capito come funziona. Se non vi risulta immediatamente chiaro, trascrivetelo ed eseguitelo. Se neppure dopo averlo trascritto e visto all'opera vi risulta chiaro, rileggete il capitolo fin dall'inizio. Se `$volteCheAveteLetto` > `1` e ancora non capite, provate a mandarmi una e-mail o, meglio, andate a sgattare in rete.

Posso usare un foreach su array?

Ovviamente sì.

Un primo esempio lo avete visto come forma "esplosa" dello script con la funzione `keys()` integrata nel ciclo `foreach` (a pag. 34); il secondo ripropone gli hobbit.

```
$baggins="Frodo";
@arrayDiEsempio = ($baggins, "Merry", "Pipino", "Sam");
foreach $hobbit (@arrayDiEsempio)
    {
    print $hobbit."\n";
    }
```

Mi dite che il ciclo `foreach` è infinitamente più semplice del ciclo `for` e che sono stato inutilmente crudele nell'accanirmi contro di voi? È vero, grazie, grazie[29].

[29] In verità, la mia religione mi impedisce di spiegare `foreach` prima di aver spiegato `for`, quindi prendetevela con il mio dio, se ne avete il coraggio.

3.5 Comunicazione

Tutto molto bello, certo, ma siamo nell'era dell'interattività e scrivere dei listati per averne degli output, almeno all'inizio, è un'esperienza demoralizzante.

Allora vi tiro subito su il morale, perché in questa fulminea sezione ci occuperemo di input (ma non di output, che abbiamo già incontrato e sul quale non vale la pena, adesso, di perdere altro tempo), cioè: interagire con i propri programmi.

3.5.1 L'input

Esistono diversi tipi di input.
Il primo, quello demoralizzante ma anche più semplice, lo abbiamo già visto: una istruzione scritta nel listato (per esempio il valore della variabile `$testo` nel listato a pag. 21 o il valore della variabile `$baggins` nell'ultimo listato).

Ma è possibile inserire dinamicamente dei dati, e un modo abbastanza esaltante all'inizio – con il quale forse avrete l'impressione di avere a che fare con un programma vero – consiste nell'istanziare una variabile e attribuirle come valore l'input dell'utente (e non del programmatore come è avvenuto fino ad ora)[30] [31].

Più facile da spiegare che da fare. Provate a trascrivere il seguente programma:

```
print "prova a scrivere qualcosa:\n";
$ecco_cosa = <STDIN>;
$ecco_cosa = reverse($ecco_cosa);
print $ecco_cosa."\n";
```

salvatelo come *ribalta.pl* in formato di solo testo[32]. Poi lanciatelo nel solito modo:
`$> perl ribalta.pl` [invio]
compare come output:
`$> prova a scrivere qualcosa` [invio]
e il cursore lampeggia nella linea sotto. Provate a scrivere qualcosa, qualsiasi cosa, per esempio:
`$> qualcosa, qualsiasi cosa` [invio]

Magia!

Lo standard input

E così abbiamo fatto la conoscenza dello standard input.

[30] Rileggete il paragrafo che questa nota conclude. So che avete capito, ma rileggetelo lo stesso. Per favore.

[31] Fatto? Bene. Grazie.

[32] Riuscite ad immaginare che cosa fa la funzione `reverse()`? Una volta visto l'output, provate a descriverne il funzionamento usando le espressioni: "accetta come argomento" e "che manipola invertendo".

Desidero portare la vostra attenzione su due fatti, uno meramente grammaticale (anche se della grammatica del perl), l'altro più generale e informatico.

Lo standard input, così come lo standard output che abbiamo già visto (a pag. 18), è scritto tutto a caratteri maiuscoli e non ha morfemi distintivi. Come vedremo nel prossimo capitolo, anche alcune variabili speciali, i *filehandles*, hanno queste caratteristiche. Tenetelo a mente, solo questo.

Quanto alla nota più generale, invece, è importante che sia chiaro il fatto che il programma ribalta.pl è esplicativo di cosa sono tutti i programmi per computer e del modo in cui funzionano. Un programma (come Word, Internet Explorer, Photoshop), anche se non ha la forma del testo leggibile ma è un file binario, di zero ed uno giustapposti, è esattamente un testo con cui noi interagiamo per creare, distruggere o modificare altri testi. A differenza di questi secondi testi, il programma è interattivo e dispone di controlli sia per "funzionare"[33] sia per presentare il risultato del suo lavoro.

Naturalmente i programmi che ho citato, e la maggior parte degli altri programmi esistenti, svolgono numerose operazioni e praticamente tutte assai più complesse di quella svolta da ribalta.pl, ma questo non cambia la sostanza dei fatti: avete appena (tra)scritto il vostro primo, vero, programma completo.

3.6 Slittamenti testuali

3.6.1 Le virgolette

Ho già detto che le virgolette implicano degli slittamenti testuali; permettetemi di aggiungere qualche particolare.

Fino ad ora abbiamo usato i doppi apici (tasti: [[maiuscolo][2]] su un buon 90% delle tastiere), ma potrebbero anche essere usati gli apici semplici – singoli o scempi, chiamateli come preferite. Esiste, tuttavia, una differenza abbastanza importante: gli apici singoli non accettano interpolazione, cioè quel che è scritto è scritto e va letto: se ho due righe di codice del tipo:

```
$potter = "Harry";
print 'ciao $potter\n';
```

Il mio output sarà invariabilmente:

ciao $potter\n

[33] Come il verbo/funzione `reverse()` nell'esempio, che è la parte di codice che svolge veramente il compito del programma, anche se da solo non sarebbe sufficiente.

Gli apici singoli rendono leggermente più veloce l'esecuzione del programma, ma costringono alla continua concatenazione tramite punto[34]; diciamo che vanno molto bene per descrivere del codice, come in questo esempio:

```
print 'per ottenere questo codice ho usato le variabili
$testo1, $testo2, e $x, $y, $z e poi ho fatto spesso
ricorso a codici di escape come \n \s \t \f...';
```

in cui altrimenti avrei dovuto scrivere un mucchio di barre retroverse in più e, nel caso di disattenzioni avrei fatto errori difficilmente identificabili[35].

Naturalmente è possibile concatenare stringhe tra diversi tipi di virgolette; posso scrivere il listato precedente come:

```
print 'per ottenere questo codice ho usato le variabili
$testo1, $testo2, e $x, $y, $z'."\n".' e poi ho fatto
spesso ricorso ai codici di escape \n \s \t \f...'."\n";
```

per vederne l'output con un a capo in mezzo e uno al fondo. Fate voi la prova! Non fidatevi di me!

E se invece desidero scrivere una stringa con diversi tipi di virgolette, mi basta usarne un tipo all'interno dell'altro, per esempio:

```
print 'in questo esempio "complicato" ricorro a diversi
tipi di apici '."gli uni 'dentro' gli altri\n";
```

oppure, se ho la necessità di usare un apostrofo tra apici singoli oppure virgolette doppie dentro ad altre virgolette doppie, posso sempre ricorrere al caro vecchio backslash:

```
print 'ecco l\'apostrofo che dicevo '."ed ecco \"un\" paio
di virgolette (o le virgolette sono due paia?)\n";
```

In alcuni casi, capita per i testi molto lunghi, sarebbe tuttavia utile evitare di scrivere tutte le barre retroverse, e poter scrivere più liberamente. Naturalmente dipende dai vostri testi, ma se per esempio avete molte virgolette e poche parentesi tonde il perl dispone di una funzione molto utile[36].

La funzione `qq()` sostituisce le virgolette. Ecco un esempio:

[34] Vi ricordate? A pag. 22. Sforzatevi di capire perché è necessaria la concatenazione.

[35] Il codice sarebbe stato comunque eseguito, ma chi vede uno spazio in più o in meno tanto facilmente? Ad esempio potrei digitare $\backslash s$ invece di $/\backslash s$, nel primo caso, fallace, l'output contiene uno spazio in più, difficile da individuare, nel secondo l'output contiene i caratteri: $\backslash s$.

[36] In realtà è un'intera famiglia di funzioni – `q()`, `qq()`, `qw()`, `qx()`, `qr()` – ognuna adatta ad uno specifico uso. La *q* sta per *quoting*. Per approfondimenti vi consiglio la lettura di [SSP99] oppure potete provare a digitare nella shell:
`$> man perlop` [invio]
Nella sezione intitolata "Quote and Quote-like Operators" troverete pane per i vostri denti.

```
$signor = "rossi";
$var = qq(io penso che "insomma", si potrebbe anche
rinverdire l'alluce del signor $signor \n);
print $var;
```

Naturalmente, se volete mettere delle parentesi tonde, dovete backslashare (ops... giustapporre barre retroverse) dove opportuno.

La cosa positiva è che potete usare al posto delle parentesi anche, per esempio, delle barre, o slash, sempre a patto di scriverle, se ce ne sono all'interno della stringa, precedute da una barra retroversa:

```
$signor = "rossi";
$var = qq/io penso che "insomma", si potrebbe anche
rinverdire l'alluce del signor $signor \n esempio
di slash: \/ \n/;
print $var;
```

3.6.2 I commenti

Esiste un altro tipo di slittamento di livello testuale: i commenti. Cioè esiste la possibilità di inserire nel vostro listato del testo senza che l'interprete perl lo interpreti. Qualcosa di simile alle virgolette, ma che l'interprete perl ignora invece di prendere così com'è.

Prendete ad esempio l'ultimo listato. Potete inserivi dentro qualsiasi cosa, purché sia preceduta da un cancelletto (e tutto quel che si trova tra il cancelletto e l'invisibile carattere di nuova linea, \n, è un commento che ha lasciato un programmatore per altri programmatori).

```
# sto per istanziare la variabile $signor
$signor = "rossi"; # sto istanziando la variabile $signor
# ho appena istanziato la variabile $signor
# ed ecco un altro commento
# e un altro
# e un altro
$var = qq/io penso che "insomma", si potrebbe anche
rinverdire l'alluce del signor $signor \n esempio di
slash: \/ \n/;
print $var; # e un altro!
```

A cosa servono i commenti? Sono fondamentali, la cosa più importante dopo il codice (e talvolta, sono più importanti di certi pezzi di codice): servono a farvi capire che cosa fa e come funziona un programma anche sei mesi dopo che lo avete scritto.

E serve a farlo capire ad altri che potrebbero doverlo leggere[37].

Tenete però conto del fatto che esistono almeno due eccezioni che l'interprete perl applica nella lettura dei cancelletti, ma state tranquilli: si tratta di eccezioni sempre segnalate.

La prima eccezione si verifica proprio con le funzioni del tipo `qq()`: le parentesi che isolano gli argomenti di questa famiglia di funzioni possono anche essere sostituite da dei cancelletti. L'ultimo script che abbiamo visto può indifferentemente avere la seguente forma:

```
# sto per istanziare la variabile $signor
$signor = "rossi"; # sto istanziando la variabile $signor
# ho appena istanziato la variabile $signor
# ed ecco un altro commento
# e un altro
# e un altro
$var = qq#io penso che "insomma", si potrebbe anche
rinverdire l'alluce del signor $signor \n esempio di
slash: / \n#;
print $var; # e un altro!
```

In tal caso, il primo cancelletto subito dopo il nome della funzione `qq()` e quello successivo non vengono interpretati letteralmente a causa della funzione.

Il secondo caso riguarda le espressioni regolari, e ce ne occuperemo più avanti (a pag 76), ma vale la pena di anticipare che anche in questo caso l'interpretazione non letterale del cancelletto è suggerita da un particolare segno.

3.7 Stile: programmatori e gentiluomini

I listati che abbiamo visto fino ad ora funzionano e sono corretti, ma non sempre sono formalmente perfetti.

Ora, bisogna distinguere una perfezione formale umana ed una macchinica. Quella umana non è facilmente raggiungibile, e si identifica con la leggibilità del codice. A pag. 33 ho introdotto le mie convenzioni per uno stile leggibile:

- a capo dopo ogni punto e virgola (eccetto nelle subordinate dei cicli `for`!);
- un livello di indentazione aggiuntivo ad ogni apertura di parentesi graffe;

A queste vale la pena di aggiungerne almeno altre due:

- i nomi di variabili dovrebbero essere il più perspicui possibile;

[37]Per esempio, se ad un esame di informatica applicata qualcuno dovesse chiedervi di commentare per iscritto un listato, si aspetterebbe che voi lo facciate per mezzo di questi commenti.

- la giusta misura di commenti: non troppi, ma neppure troppo pochi: il tempo che si perde nello scrivere i commenti mentre si scrive codice è meno di quello che si perde a capire il proprio codice tre mesi più tardi.

Essendo però la leggibilità tipografica un parametro soggettivo e culturale, per quanto desiderabile è difficile da catturare. Se vi trovate meglio lavorando in modo diverso (e tendenzialmente siete gli unici a dover leggere i vostri stessi listati), fate pure come preferite.

Quanto alla perfezione macchinica, esistono almeno due metodi per ottenerla (ma non sempre è la soluzione preferibile: la perfezione macchinica impedisce qualche strada): l'opzione -w e il modulo `strict`[38].

L'opzione -w (che sta per *warnings!*) va scritta, su X, nella prima linea, dopo `#!/usr/bin/perl`. Su win, quando si avvia un programma, si scrive
`$> perl -v programma` [invio]
Una alternativa, valida sia per win che per X, consiste nello scrivere all'inizio del codice – ma dopo al linea con lo *she bang*, se lavorate con X – l'indicazione: `use warnings;` che ha esattamente lo stesso valore.

In sostanza, se attivate questa opzione l'interprete perl è un po' più pistino e avverte se una variabile non è stata usata o è usata non correttamente. Qualche volta interrompe l'esecuzione del listato, altre volte no. In generale l'opzione -w è cosa buona e giusta – ed evita errori.

Il modulo *strict* permette di ottenere codice a prova di bomba. Si usa scrivendo tra le prime righe del codice `use strict;` ed è molto più rigido dell'opzione -w. Io lo uso raramente, nella maggior parte dei casi sarebbe doveroso ricorrervi, in alcuni è soprattutto una perdita di tempo.

Se vi interessa provatelo, e fatevi i vostri esperimenti. Io non mi dilungherò oltre sulla questione.

[38] Tecnicamente parlando `strict` non è un modulo ma una pragma. Ma noi faremo finta di niente.

Capitolo 4

Iniziamo a domare la bestia

4.1 Diversi tipi di variabili - approfondimento

Nel bel volume [WCO00] (ottimo per iniziare, in inglese, di ampio respiro), si suggerisce che le variabili non strutturate siano associate a nomi singolari e le variabili strutturate a nomi plurali[1].

Non ho seguito questa associazione perché non mi convince, è legittima ma arbitraria – il numero è, in italiano, connesso (ed espresso d)alla morfologia, e mi pareva che si potesse generare della confusione – e quindi è altrettanto legittimo ignorarla.

Poiché sono possibili innumerevoli associazioni e parallelismi, tutti più o meno arbitrari, i riferimenti alla lingua qui presenti – morfologia e tipi di parole, valenze verbali ed argomenti delle funzioni – sono relativamente pochi e sempre in qualche modo verificabili, almeno nella misura di un confronto diretto tra lingua naturale e linguaggio di programmazione, e fuori da schematismi linguistici troppo astratti.

Per dirla con Eco, e forse anche con Kant, un morfema è un morfema è un morfema...

4.1.1 Variabili non strutturate

Giusto un po' di terminologia, per evitare futuri fraintendimenti. Con **scalare** e con **stringa** d'ora in poi mi riferirò a qualsiasi sequenza di caratteri (numeri, lettere, segni di interpunzione) in quanto tali. Testi nel senso materico[2] del termine.

Ed ora qualche altra regola sui nomi di variabile.

La regola generale, che permette sempre la creazione di nomi di variabile corretti, è che un nome di variabile deve essere composto solo da lettere minuscole. Ma questa regola è un poco limitante: poter inserire qualche altro carattere potrebbe essere utile e

[1] Per la precisione quel manuale parla di *singular and plural data. Strings and numbers are singular pieces of data, while lists of strings or numbers are plural.*

[2] Ecco un altro termine orribile, mutuato questa volta non dall'ambito informatico ma da quello della critica dell'arte. Ognuno ha i propri scheletri nell'armadio.

giovare alla leggibilità del codice. Ad esempio, tra le due variabili `$nomefilecorretto` e `$nome_file_corretto` la seconda è decisamente più chiara.

Un nome di variabile può:

- essere lungo 255 caratteri;
- iniziare con una lettera o un carattere di underscore[3] (ma è meglio evitare di farlo iniziare con un carattere di underscore);
- iniziare anche con altri caratteri, ma i nomi di variabili che iniziano con una cifra possono essere costituiti solo da cifre e quelli che iniziano con un altro carattere possono essere costituiti solo da quel carattere, e anche in questo caso sarebbe meglio evitare, visto che la maggior parte delle variabili predefinite del perl[4] sono costituite solo da segno del dollaro e al massimo due o tre simboli non alfabetici.

Detto questo, siete liberi di battezzare le variabili come preferite, rimanendo veri i principi di leggibilità del codice cui ho accennato a pag. 40.

Le stringhe che diventano numeri e viceversa

Togliamoci questo sassolino dalla scarpa che prima o poi salta sempre fuori: per l'interprete perl è importante sapere se quando si parla di una variabile ci si riferisce ad un numero o ad una stringa?
La risposta è no.

L'interprete legge il valore della variabile, guarda come lo usiamo e capisce se ci riferiamo a un numero o a un testo (si basa soprattutto sul modo in cui lo usiamo, il che rappresenta comunque la migliore approssimazione possibile a quel che intendevamo farne).

Pressoché inutile perdersi in minuziose descrizioni: copiate il codice che segue e lanciatelo, poi fate attenzione all'output e provate – con l'output sotto gli occhi – a rispondere alle domande che seguono (non fate quella faccia: sono domande facili e basta un cucchiaino di attenzione per rispondere correttamente).

```
$testo = "ciccio";
$numero = 5;
$testumero = "6";

$somma1 = $testo + $numero;
$somma2 = $testo + $testumero;
$somma3 = $testumero + $numero;

$conca1 = $testo . $numero;
```

[3] Trattino basso.

[4] Per ora immaginatele come parole grammaticali del perl, ne riparleremo a pag. 53.

```
 $conca2 = $testo . $testumero;
 $conca3 = $testumero . $numero;

print "
 $somma1 = $testo + $numero;\n
 $somma2 = $testo + $testumero;\n
 $somma3 = $testumero + $numero;\n
\n
 $conca1 = $testo . $numero;\n
 $conca2 = $testo . $testumero;\n
 $conca3 = $testumero . $numero;\n
";
```

Dunque, ecco le domande:

1) la prima variabile istanziata è un numero o una stringa?

2) oltre che dal nome, da cosa si capisce che la seconda variabile è un numero?

3) la terza variabile è un numero o una stringa? E da cosa si capisce?

4) qual è il valore di una stringa quando viene sommato ad un valore numerico?

5) esiste differenza tra una stringa esclusivamente testuale ed una stringa composta da un numero? L'interprete si comporta diversamente con l'una e l'altra in una addizione?

6) come viene trattato un numero quando è concatenato ad una stringa?

e ora la più difficile:

7) il comportamento della concatenazione è sempre coerente, nel senso che indipendentemente dai tipi di oggetto che concatena l'interprete concatena sempre nello stesso modo, senza fare differenze tra stringhe, numeri, stringhe composte da numeri. Sapreste dire se l'addizione gode della stessa coerenza? E se no, sapreste spiegare perché?

Vi aiuto[5].

Se avete una macchina X ed usate la flag `-w` oppure scrivete nel vostro codice (all'inizio, meglio): `use warnings;`, l'interprete, al quale con quella formula esoterica è stato detto di essere un po' più severo negli errori, vi avverte con un messaggio del tipo:

Argument ciccio isn't numeric in addition (+) at ./provaperl21.pl line 9.

Avverte cioè del fatto che avete tentato di sommare un testo. Perché non lo fa anche con la variabile `$testumero`?

[5] O forse no.

4.1.2 Variabili strutturate

Ancora un poco di terminologia.

Ho parlato[6], a proposito degli hash, di parole chiave – e a proposito degli array di chiavi numeriche – contrapponendo questo tipo di dato a dei generici *elementi* della lista (array o hash).

La terminologia corretta prevede che per le parole chiave e le chiavi numeriche, gli indirizzi degli elementi delle liste, si parli più genericamente di **chiavi** e che per gli elementi si parli di **valori** della lista.

Una distinzione importante, tra chiavi e valori, riguarda la loro unicità: le chiavi sono sempre uniche, mentre i valori possono essere uguali. Perché ci interessa?

Pensateci: se dovete creare una lista di frequenza[7], verosimilmente questa sarà immagazzinata in una variabile strutturata, in una lista, appunto. Poiché per ogni parola avremo DUE dati – la parola e il numero di volte che occorre –, avremo la nostra lista di frequenza in un hash[8]: quale di questi dati sarà la chiave e quale il valore del nostro hash?

Non vi presento subito la soluzione; vedremo più tardi che cosa serve per una lista di frequenza e come va amministrato.

Potenza delle variabili strutturate

Le variabili strutturate sono davvero una gran bella cosa, soprattutto perché il perl dispone di alcuni strumentini mica male per la manipolazione e la creazione (sì, avete letto bene: la *creazione*) di variabili strutturate. Creazione di liste a partire, per fare un esempio qualsiasi, da variabili non strutturate come delle stringhe... cioè dei testi!

In questo paragrafo risponderemo, con un fine biecamente esemplificativo[9] a due domande:

- come faccio a scoprire qual è la cinquantacinquesima parola di un testo senza contarle tutte?
- come faccio a mettere in ordine alfabetico tutte le parole di un testo?

Per rispondere a queste domande introdurremo qualche nuova funzione: `split()` (sì, come *banana split*[10]) e `sort()`. Pronti? Via!

Iniziamo con un listato davvero banale, non lasciatevi impressionare dalla sua consistenza: la maggior parte delle linee è costituita da una variabile di testo:

[6] Cfr. pag. 31.

[7] Una lista che raccoglie tutte le parole presenti in un testo, e di ognuna specifica quante volte è usata.

[8] Infatti gli array sono portatori di un solo tipo di dato, i valori, e di un altro dato intrinseco, che è l'ordine dei valori, ma che in quanto intrinseco non è manipolabile, è perció utile fino ad un certo punto.

[9] Della potenza e bellezza delle variabili strutturate, e degli array in particolare.

[10] Ma senza la banana, chiaramente.

```
$testo = "Ciao, il mio nome e' Arild Ovesen.
Soffro di malattie rare e mortali, cattivi risultati
scolastici, estrema verginita', e senso di colpa per non
aver inoltrato 50 miliardi di catene di Sant'Antonio
mandatemi da persone che pensano davvero che se uno le
inoltra la povera piccola bambina di 6 anni a Foligno
con un capezzolo in fronte riuscira' a procurarsi
abbastanza denaro per toglierlo prima che i genitori
la vendano al Circo Orfei.
Prima di tutto devi mandare questa lettera a 7491
amici entro i prossimi 5 secondi, altrimenti inciamperai
camminando e cadrai su una collina di escrementi animali.
Se nella tua citta' tutti i cani sono stati privati
chirurgicamente dell'ano, a causa di uno strano virus la
ventola dentro il pc si mettera' a girare al contrario
e ti risucchiera' il processore.
Dopo una serie di lampi di colore blu, dal tuo lettore cd
uscira' il totem (o tantra) della buona sorte
che ha gia fatto il giro del mondo tre volte
(e mi ha confidato di essersi rotto le palle
perche' vorrebbe starsene cinque minuti a casa)
e ti picchiera' a sangue.
Per ogni messaggio che manderai all'indirizzo
boccaloni@cheregalo.it
un'associazione donera' un quarto di dollaro per comprare
un aereo all'aviazione americana che servira' per tirare
giu' un'altra funivia in Italia.
E' tutto vero!!!
Erode Scannabelve non mando' questa mail a nessuno:
dei suoi tre figli uno comincio' a drogarsi,
il secondo entro' nel Partito Umanista e il terzo si
iscrisse a Ingegneria dei Materiali.
Turiddu Von Wasselvitz, un allenatore di farfalle da
combattimento austro-siculo, si fece beffe di questa
mail ad alta voce, e in quello stesso istante gli
esplose la testa.
Meo Smazza, pornodivo shakespeariano, non diede
alcun peso a questa mail: ignoti gli riempirono un
profilattico di azoto liquido, e lui se ne accorse solo
dopo averlo indossato.
Un tizio che conosco non ha diffuso questa mail
```

```
e ha disimparato ad andare in bicicletta.";

@parole = split(" ", $testo);

print " $parole[12]\n $parole[56]\n";
```

Non pretendo che ricopiate tutta la stringa; il mio output (cioè la tredicesima[11] e la cinquantasettesima parola) è:
mortali,
di
Adesso concentriamoci sulla penultima ed ultima riga di codice, che sono quelle che ci interessano (dopo affronteremo il problema del perché diavolo si dovrebbero cercare la tredicesima e la cinquantasettesima parola di un testo). Allora:

```
@parole = split(" ", $testo);
```

Iniziamo col dire che viene istanziata una variabile, un array. Questo array – una lista di elementi, ricordatevelo bene – è il risultato di una operazione che si chiama *split* [spacca, dividi]. È legittimo pensare che il qualcosa che viene diviso sia un testo e che quindi la lista risultante sia costituita da frammenti di testo.

Osservate tra parentesi: divisi da un virgola, possiamo identificare due argomenti. Il secondo è la variabile `$testo`; sappiamo anche qual è il valore di quella variabile[12] e possiamo quindi decidere che il *$testo* è ciò che è stato diviso.

Notate che il complemento oggetto sta in seconda posizione, come per il verbo *print*.

Fate questo esercizio mentale: l'interprete perl sa che deve dividere (l'informazione gli viene fornita dal tipo di funzione); sa cosa deve dividere (il testo, specificato dal secondo argomento); sa infine che il risultato della sua operazione deve essere immagazzinato in una lista (dichiarata a sinistra del segno di uguale). Di quale altra informazione avrà ancora bisogno per portare a termine il proprio lavoro?

Al primo posto tra gli argomenti di *split* (li vedete fra parentesi, come di norma per le funzioni), è esplicitato il modo in cui la stringa va suddivisa, quel che nel testo viene sostituito da una cesura – lo spazio – tra i frammenti del testo stesso.

In altre parole: l'azione espressa dalla funzione `split()` consiste nella segmentazione in parti di un testo quindi, quando si usa questo verbo è, bene specificare il testo da dividere e come – o, meglio, dove: dove c'è uno spazio, in questo caso – dividere la stringa, il testo.

Cosa fa l'interprete perl quando incontra questo verbo? Legge la stringa e la spezzetta in stringhe più corte, introducendo una cesura ogni volta che incontra lo spazio. E cosa fa di questi pezzetti? Li infila in una lista di pezzetti. Questa lista è l'array.

[11] Ricordatevi che gli elementi di un array sono numerati a partire da 0, e quindi l'elemento numerato come 12 è in realtà il tredicesimo.

[12] Una delle tante e-mail di catene di sant'Antonio che sono arrivate nella mia casella di posta elettronica.

Il resto è semplice: dato un array, il programma stampa gli elementi della lista alla posizione 12 e alla 56.

Parentesi: un'importante distinzione...

...in seno alle funzioni.

Le funzioni, diciamo, si distinguono in due grandi categorie: quelle che agiscono su materiale e non richiedono l'esplicitazione di un "contenitore" del risultato delle loro azioni (tipicamente una variabile), e quelle che richiedono una esplicitazione della variabile che potremo poi usare per fare riferimento al risultato delle azioni.

Prendiamo ad esempio tre funzioni che abbiamo visto fino ad ora: `print(), reverse()` – vista a pag. 36 – e `split()`.

`print()` non richiede l'esplicitazione di una variabile, ma prevede un output, e quando non viene selezionato un particolare output seleziona quello predefinito; `reverse()` e `split()` richiedono invece, rispettivamente, la stringa rovesciata e l'array dei pezzi di testo (che al limite può essere costituito di un solo pezzo, quando l'interprete non trova nel testo occorrenze del carattere che noi vorremmo usare per dividere il testo).

In realtà, in entrambi i casi è tecnicamente possibile non fornire variabili con le quali fare riferimento ai risultati delle azioni, ricorrendo alle variabili predefinite, ma: a) la pratica è deprecata dallo stesso interprete perl; b) si tratta di codice difficilmente leggibile, e noi ci stiamo impegnando per produrre codice che invece sia leggibile.

Ora vi mostrerò il caso di una variabile che non richiede alcunché.

Riprendiamo il listato con la funzione `reverse()`. Ve lo ricordate?

```
print "prova a scrivere qualcosa:\n";
$ecco_cosa = <STDIN>;
$ecco_cosa = reverse($ecco_cosa);
print $ecco_cosa."\n";
```

Fate attenzione ad un dettaglio nell'output:

$> ./pq.pl
prova a scrivere qualcosa:
qualcosa

asoclauq

Il programma ha scritto qualcosa in più rispetto a quel che doveva: la stringa che doveva essere rovesciata è costituita dalla sola parola *qualcosa*; io, prima della parola *qualcosa*, vedo un carattere in più. Lo vedete anche voi? Guardate con attenzione.

L'a capo.

Certo, quello spazio non dovrebbe esserci: io l'ho digitato, ma per segnalare al programma che avevo finito di scrivere la stringa che volevo rovesciasse, non come parte della stringa stessa. Invece l'interprete ha capito che il segno $\backslash n$ faceva parte della stringa, e l'ha messo all'inizio della stringa stessa[13].

La prossima volta dovrò scrivere meglio il programma, per esempio così:

```
print "prova a scrivere qualcosa:\n";
$ecco_cosa = <STDIN>;
chomp($ecco_cosa);
$ecco_cosa = reverse($ecco_cosa);
print $ecco_cosa."\n";
```

nel quale la funzione `chomp()` elimina il carattere di avanzamento di riga alla fine della nostra stringa di output.

Non conosco l'etimologia del verbo *chomp* e non saprei fornirne un traducente, e neppure ho mai usato tale funzione in contesti diversi da quello dell'interazione con l'utente, ma funziona, e infatti il mio nuovo output è:

```
   $> ./pq.pl
prova a scrivere qualcosa:
qualcosa
asoclauq
```

Vedete? adesso non c'è più lo spazio.

Il motivo per cui vi ho parlato della funzione `chomp()` è che, come potete vedere nella terza linea di codice, questo verbo non richiede risultati, agisce sul posto[14].

Ma si potrebbe, volendo, istanziare una variabile nella quale sia immagazzinato il risultato dell'operazione svolta da `chomp()`? Certo, anche se l'utilità di questo lavoro aggiuntivo per il processore e l'interprete va valutata di volta in volta.

Il fatto di poter fare delle copie di variabili può rivelarsi piuttosto utile nel corso della programmazione. Immaginate di voler fare delle ricerche su un testo sia prima che dopo una correzione automatica, ad esempio per verificare automaticamente quali correzioni sono state fatte; in tal caso potreste istanziare una variabile uguale alla variabile iniziale, effettuare le modifiche sulla copia, e infine confrontare l'originale e la copia modificata[15].

Torniamo alle due domande

A questo punto dovrei rispondere alla domanda sul perché può essere interessante sapere qual è la ventesima parola di un testo. Non perderò troppo tempo; vi basti sapere che in

[13] Gli ultimi saranno i primi!

[14] È come il signor Wolf di Pulp Fiction, per intenderci.

[15] Naturalmente, si tratta di una possibile procedura. ESERCIZIO: ricorrendo allo standard input riuscite ad immaginare un correttore che vi permette di verificare cosa va corretto?

alcuni casi, quando bisogna isolare o cancellare un frammento di un testo molto lungo e non si può ricorrere a dispositivi di altra natura[16], il modo migliore di agire consiste nel dire all'interprete di cancellare o mostrare tutte le parole dalla numero 16 alla numero 321 (i numeri sono casuali).

La seconda domanda prevedeva che si disponessero in ordine alfabetico tutte le parole di un testo[17].

Ho ripreso il codice dell'esempio a pag. 46, accorciando un po' la stringa da analizzare. Osservate il codice, poi ne discutiamo.

```
$testo = "Ciao, il mio nome e' Arild Ovesen.
Soffro di malattie rare e mortali, cattivi risultati
scolastici, estrema verginita', e senso di colpa per non
aver inoltrato 50 miliardi di catene di Sant'Antonio
mandatemi da persone che pensano davvero che
se uno le inoltra la povera piccola bambina di 6 anni
a Foligno con un capezzolo in fronte riuscira' a
procurarsi abbastanza denaro toglierlo
prima che i genitori la vendano al Circo Orfei.
Prima di tutto devi mandare questa lettera a 7491
persone entro i prossimi 5 secondi, altrimenti inciamperai
passeggiando e cadrai su una collina di escrementi animali.
E' tutto vero!!!
Erode Scannabelve non mando' questa mail a nessuno:
dei suoi tre figli uno comincio' a drogarsi,
il secondo entro' nel Partito Umanista e il terzo si
iscrisse a Ingegneria dei Materiali.
Turiddu Von Wasselvitz, un allenatore di farfalle da
combattimento austro-siculo, si fece beffe di questa
mail ad alta voce, e in quello stesso istante gli esplose
la testa.
Meo Smazza, pornodivo shakespeariano, non diede peso a
questa mail: ignoti gli riempirono un profilattico di azoto
liquido, e lui se ne accorse solo dopo averlo indossato.
Un tizio che conosco non ha diffuso questa mail e ha
disimparato ad andare in bicicletta.";
```

[16] Come le espressioni regolari, che vedremo nel prossimo capitolo.

[17] Anche questa operazione può apparire bizzarra, ma se vi interessa avere un unico documento che vi mostri se date parole sono presenti o meno nel testo, senza dover ogni volta fare una ricerca sul vostro editor, fate attenzione alla manciata di linee di codice che sto per mostrarvi.

```
@parole = split(" ", $testo);

@aelopr = sort(@parole);

foreach $parola (@aelopr)
    {
    print $parola."\n";
    }
```

Dovrebbe essere tutto chiaro fino alla creazione dell'array *@parole*[18].

E dopo? Viene creato un secondo array, *@aelopr*, con la funzione `sort()` che accetta come argomento una lista – eventualmente una stringa – e la riordina in ordine alfabetico.

E dopo ancora?
(Guardate l'output).

ESERCIZIO: secondo voi `sort()` è una di quelle funzioni che richiedono l'esplicitazione di una variabile che contenga il risultato dell'operazione che svolgono? Cioè: era davvero necessario creare una variabile `@aelopr` o avrei potuto fare altrimenti?
Provate, in ogni caso, a modificare il listato senza creare quell'array: quando l'interprete perl avrà smesso di segnalare errori di sintassi avrete la risposta a questa domanda[19].

ESERCIZIO: sapendo che `sort()` richiede come argomento un array, se io volessi disporre in ordine alfabetico le chiavi di un hash che cosa dovrei scrivere come argomento della funzione? Pensateci.

Vi aiuto: serve una funzione che abbiamo già visto e che genera un array. Ancora non vi viene in mente? Pensateci. Io non ho fretta.

Infatti, non ne ho.

Se non vi viene in mente, non preoccupatevi, lo vedremo più avanti.

[18] Se non lo è tornate a rileggere a partire da pag. 48.
[19] Ma sarebbe meglio provare a dare la risposta prima. Per esercizio, intendo.

Questo codice ha il piacevole effetto secondario di funzionare come una rudimentale lista di frequenza: poiché le parole compaiono nella lista tutte le volte che occorrono nel testo, basta contarle nella lista[20].

4.1.3 Variabili predefinite

Abbiamo visto che in certi casi, per determinate funzioni, possiamo omettere di esplicitare determinati argomenti. Ad esempio la funzione `print()` può comparire anche senza il complemento di luogo (ovvero dove scrivere il complemento oggetto dell'azione).

Provate ad immaginare: come lavora – verosimilmente, non è detto né necessario che lavori veramente in questo modo – l'interprete perl?

Dunque, sa di avere un certo numero di argomenti per ogni verbo, una determinata valenza, ma può trovare o meno tutte le valenze saturate. È lecito ipotizzare che esista un costrutto condizionale per ogni argomento: *se questo argomento è stato esplicitato, usa il contenuto dell'argomento in questo modo, altrimenti...*

Quando in un testo italiano qualcosa manca, ma noi siamo comunque in grado di comprenderne il significato, diciamo che la parte mancante è sottintesa, oppure che è inferibile a partire dal contesto verbale o situazionale. Ma l'interprete non ha un contesto situazionale di riferimento, dispone invece di **variabili predefinite**: variabili con un nome ed un valore che non cambia mai, che non vengono create dall'utente ma fanno parte della grammatica dell'interprete perl.

Non descriverò tutte le variabili predefinite: a noi ne serve una manciata e mi limiterò a quella. Si chiamano: `$_` `$.` `$^I` `$!` `@ARGV` ed è a causa loro che i nomi di variabile non dovrebbero iniziare con altri segni che lettere o numeri.

Di queste variabili non vi fornirò adesso esempi in contesti, perché dovrei anticipare funzioni e strutture di controllo che non è indispensabile (potrebbe anzi essere deleterio) descrivere adesso. Invece vi presenterò una definizione per ognuna di esse e nei prossimi esempi saprete che per un ridotto numero di variabili esistono questi valori[21].

`$_` identifica l'elemento attualmente focalizzato. È difficile da spiegare perché è un elemento eminentemente deittico[22]. Diciamo semplicemente che, ad esempio, in una struttura di controllo iterativa, `$_` significa: [questo].

Immaginate un listato che legge un testo, una frase per volta, e lo stampa nello

[20] Anche se migliore del conteggio manuale nel testo non formattato, questo metodo è piuttosto rudimentale (e inutile con testi molto grandi). Nel seguito vedremo come costruirsi una lista di frequenza vera, comoda, semplice. E che fa anche il caffè.

[21] Cioè, non iniziate adesso a preoccuparvi: quando reincontreremo delle variabili predefinite, farò riferimento alle spiegazioni che seguono.

[22] Spiegare il significato di parti del discorso deittiche è facile, ma solo a patto di esplicarne il contesto d'enunciazione, cosa che io, qui ed ora, preferisco evitare.

standard output[23]. In quel listato `$_` significa: [la linea che sto leggendo in questo momento, a questa ripetizione delle istruzioni];

$. più facile da spiegare. Tornate all'ultimo esempio che ho fatto, quello del programma che legge una riga per volta e la stampa sul monitor.
`$.` tiene a mente a quale numero di linea ci si trova per ogni `$_`. Facile, no? Ha qualcosa in comune con la variabile contatore che si trova nei cicli `for`;

**$!** slittiamo ad un altro livello. Questa variabile predefinita memorizza il tipo di errore che si fa, se se ne fa uno. Torniamo all'ipotetico listato che legge un file; tale programma dovrà prima – ovviamente – aprire il file. Il programmatore, saggiamente, adopererà una funzione che apre i file con questa sintassi, grosso modo: *apri($file) oppure print "non ci sono riuscito perché: $!"*, cioè: apri il file `$file`, se non ci riesci stampa un avvertimento che includa anche qual è l'errore secondo l'interprete perl[24];

$ ^ I ora, immaginate che il programma non solo legga e stampi sul monitor riga per riga un intero documento, ma immaginate anche che lo modifichi. Bene, la *I* nel nome di questa variabile sta per *inplace edit*, cioè: stampa sul posto, modifica direttamente. In realtà questa variabile ha una sintassi un po' particolare, e significa quel che ho scritto solo in un caso, quando cioè è impostata come uguale a zero:

```
$^I = '';
```

Se invece ha un qualsiasi altro valore, questo valore verrà giustapposto all'estensione dei file generati dal programma, che risultano essere quindi delle copie. Complicato? Non preoccupatevi, vedremo `$ ^ I` all'opera tra qualche script;

@ARGV questo array predefinito contiene tutti gli argomenti passati dalla linea di comando. Ops. Ecco qualcosa che non avevo ancora spiegato. È presto detto: avete presente quando si usavano `print()` e lo `STDIN` per mettere nello script dei valori di variabile? Ebbene, quegli stessi valori – con una sintassi differente – potevano essere inseriti nello script già da linea di comando.

Facciamo un altro passo indietro: avete presente la shell e l'avvio di uno script perl? Quelli con:
`$> ./nomeprogramma.pl` [invio]
oppure con
`$> perl nomeprogramma.pl` [invio]
Invece di aspettare che lo script mi chieda, ad esempio, la parola da rivoltare, è possibile scrivere la parola immediatamente dopo e modificare il programma in modo

[23] Per quanto sia incredibile, è esattamente grazie ad un dispositivo analogo che riuscite letteralmente a vedere qualcosa sullo schermo della vostra macchina.

[24] L'errore potrebbe anche essere il fatto che l'interprete non riesce a trovare il file che deve leggere.

che funzioni anche con:
`$> perl nomeprogramma.pl ‘‘rivoltamicomeuncalzino’’` [invio]
Chiaro?

4.1.4 I filehandle

Un *filehandle*[25] è un tipo di variabile un po' particolare.

Innanzitutto, morfologicamente, è sottoposto a norme piuttosto rigide: va scritto tutto maiuscolo e non vuole alcun segno distintivo. Ne abbiamo visti già due, ulteriormente particolari perché si tratta di gestori di file predefiniti. Riuscite a ricordare quali parole speciali che rispettano le norme ortografiche appena menzionate abbiamo già visto?

Si tratta di *SDTIN* ed *STDOUT*.

Esiste un altro filehandle predefinito, che si chiama ARGV e che ha molto in comune con l'omonima variabile predefinita, nel senso che in certi cicli, in certe strutture di controllo iterative, si usa ARGV: per esempio, quando viene passato dalla linea di comando il nome di una directory e si desidera poter leggere il contenuto di tutti i file contenuti in quella directory (vedremo come si fa).

La cosa divertente è che le variabili e i filehandle predefiniti, proprio perché sono predefiniti, si vedono in giro assai meno spesso di quel che si potrebbe immaginare, infatti possono essere omessi nel codice – in determinati casi ben normati[26] – con la certezza che l'interprete perl saprà colmare la mancanza di un argomento con l'argomento (predefinito) corretto.

Badate bene, però: i filehandle non sono solo predefiniti: fra poco vedremo la funzione `open()` che apre un file (il suo complemento oggetto) e ricorre ad un filehandle per gestirne il contenuto[27]. Potete pensare che funzioni come `print()`, che legge una stringa e la trascrive nel filehandle `STDOUT`.

Ma... se `print()` scrive in un filehandle, e quando apro un file uso un filehandle per gestirlo... evidentemente... posso aprire un file e poi scriverci dentro specificando il filehandle con il quale gestisco il file appena aperto...

Magia!

4.2 I giocatori di Rami

In questa sezione ci occuperemo di strutture di controllo. Non spaventatevi per il nome: abbiamo già visto molte strutture di controllo: `foreach` e `for` (sono solo due, certo, ma costituiscono il 50% di tutte le strutture di controllo disponibili!).

[25] Gestore di file: *handle*, in inglese, significa [manico].
[26] Ad esempio nel caso del verbo *print* e del filehandle *STDOUT*.
[27] E in questo caso, esattamente come per le variabili, è il programmatore a decidere il nome del filehandle.

Potete immaginare le strutture di controllo come delle griglie semantiche e sintattiche che permettono di mettere insieme i verbi, o funzioni. In altri termini, potete pensare al fatto che un numero ridottissimo di testi, e nessuno di una certa complessità, funziona se è semplicemente costituito da predicati coordinati per asindeto. Alla medesima maniera, un numero ridotto di programmi, e nessuno di una certa complessità, funziona per semplice giustapposizione di funzioni.

Possiamo dire che le strutture di controllo sono quei dispositivi della lingua che permettono l'esistenza di subordinate. Periodi ipotetici e subordinate temporali/causali[28] vengono prodotte grazie (dentro) alle strutture di controllo.

Le strutture di controllo controllano un flusso di dati e operazioni dirigendolo verso l'output che il programmatore desidera. Non so quanti di voi ricordano Rami[29], un gioco per bambini degli anni Ottanta nel quale bisognava comporre dei disegni con delle palline colorate facendole scendere attraverso percorsi con numerosi bivi. Il gioco consisteva nel chiudere per ogni pallina una delle due possibilità che si presentavano ad ogni bivio in maniera da far andare la pallina al posto giusto.
Usare le strutture di controllo significa esattamente inviare la pallina al posto giusto. E, come accadeva in Rami, se si sbaglia bisogna annullare e ricominciare daccapo (ma è di gran lunga meno noioso che in Rami!).

Ho anticipato che esistono quattro strutture di controllo. Si tratta di:

if la struttura condizionale per eccellenza. Ne esistono tre tipi[30], a seconda della complessità:

- se si verifica una condizione, agisci di conseguenza;
- se si verifica una condizione, agisci di conseguenza, altrimenti fai un'altra cosa;
- se si verifica una condizione, agisci di conseguenza, se se ne verifica un'altra, fai una azione diversa, se... (il numero di alternative è illimitato), altrimenti fai un'altra cosa.

Come potete vedere, si tratta di strutture molto semplici e comuni alla nostra lingua.

while è in un certo senso una struttura condizionale diversa, che potrebbe essere tradotta come: *fintantoché* si verifica una data condizione, agisci di conseguenza. *while* non dispone della ricchezza di *if*, ma ha un contrario: **unless**, cioè: finchè una data condizione non si verifica, agisci in questo modo. Si tratta di una struttura piuttosto utile e comune, soprattutto nella lettura dei testi.

[28] Non esistono altri tipi di subordinate.

[29] Non ho trovato tracce di Rami in rete: la Quercetti produce una serie di giochi simili – Reflex, DigiColors – ma non identici. Potete comunque andare sul sito a vedere, per capire bene quel che intendo.

[30] La cui distinzione è di gran lunga più motivata della distinzione che le grammatiche tradizionali descrivono per il periodo ipotetico italiano.

for l'abbiamo già vista; si tratta di un'altra struttura condizionale o, meglio, di una struttura iterativa e quindi, indirettamente, condizionale, esattamente come *while* (mentre *if* è una struttura puramente condizionale e nient'affatto iterativa). La sua struttura è: data una condizione iniziale, data una condizione finale, e sapendo che esiste un meccanismo di trasformazione della condizione attuale da condizione iniziale a condizione finale, agisci in un certo modo.

Come vedete, anche in questa descrizione il ciclo `for` è complicato da spiegare.

foreach è una struttura di controllo (un ciclo puramente iterativo) di una disarmante semplicità: per ogni elemento di una lista, agisci di conseguenza.

Chissà perché lo spiegano sempre per ultimo.

Di questi quattro costrutti inizio col descrivervi la forma più ortodossa, introducendo di volta in volta nel proseguio forme alternative, più gergali e oscure.

4.2.1 E se...

Sono quasi certo di averlo già detto, ma ripeterlo costa poco e può giovare: il periodo ipotetico in perl è uguale al periodo ipotetico in italiano:

protasi – apodosi
subordinata – principale
condizione – conseguenza
se ... – allora...

Ma se per uno straniero imparare l'italiano può essere difficile, e può esserlo anche perché non sempre sono chiari i confini di una proposizione, questo accade con meno frequenza con il perl.

Conosciamo tutti quanti la regola sintattica[31] per la quale le subordinate sono raccolte tra parentesi tonde e le principali tra parentesi graffe: vale anche in questo caso.

Ormai sarete diventati bravini, quindi guardiamo subito un listato, poi lo commenteremo. Voi nel frattempo copiatelo e lanciatelo dalla shell.

```
$terzo = "ananas";
@lista = ("avocado","mela","pera",$terzo);

print "scrivi un numero da 0 a 3, per favore\n";
$numero = <STDIN>;
if($numero==1)
    {
```

[31] Valida per le strutture di controllo, in effetti, e non per le funzioni né per le dichiarazioni dei valori delle liste. Tenetelo a mente.

```
    print "il frutto numero uno: $lista[$numero]\n";
    }
elsif($numero==2)
    {print "il frutto numero due: $lista[$numero]\n";}
elsif($numero==3){
    print "il frutto numero tre: $lista[$numero]\n";}
else
{print "hey, dovevi scrivere un dannato numero uguale a 1,
a 2 oppure a 3, non i fatti tuoi!\n\n";}

print "\n\n";
```

Questo listato ha una particolarità, che va segnalata ma è ininfluente ai fini della sua esecuzione, quindi ve la illustro subito e passiamo oltre.

Osservate l'amministrazione delle parentesi graffe: nella prima opzione (se la variabile *numero* è uguale ad uno) potete vedere il modo di trattare le graffe generalmente applicato nel presente manuale: molto chiaro, esplicito, ingombrante. Nella seconda opzione (la variabile *numero* è uguale a due) compare un trattamento delle graffe applicato talvolta anche in questo manuale: ignora gli a capi dopo l'apertura e prima della chiusura della parentesi: può andare nei casi come questo nei quali la principale è una sola, altrimenti è più facile dimenticarsi di chiudere o aprire qualcosa. Nella terza opzione (la variabile è uguale a tre) vedete la forma tipicamente applicata dai programmatori C: la parentesi va aperta sulla prima linea della struttura e chiusa sull'ultima.
Tutti questi modi vanno bene, ne potete inventare ed usare anche altri, se avete tempo da perdere. Adesso pensiamo alle cose importanti.

La coppia *se(protasi){apodosi}* è il caso più semplice, ma in casi come quello preso in considerazione[32] sono espresse altre possibilità oltre a quella espressa nella protasi. Infatti *numero* può essere uguale ad uno, a due, a tre oppure può non essere uguale a nessuno di questi numeri.

Lessico e sintassi di queste possibilità alternative sono piuttosto semplici: la prima possibilità è sempre espressa dall'*if* semplice, con la sua struttura che abbiamo già visto. Le possibilità successive sono espresse da *elsif*, con la stessa identica struttura sintattica dell'*if*.

L'ultima opzione[33], quella che esprime il significato [in tutti gli altri casi possibili] oppure [altrimenti], non ha ovviamente una protasi, ma solo una principale.

[32] Avete capito cosa fa e come funziona il listato? Badate bene: NON ci sono novità, siete assolutamente in grado di tradurlo in italiano.

[33] L'aggettivo non è casuale! Questa deve essere l'ultima opzione, altrimenti l'interprete sbrocca.

Non dovrebbero esserci perplessità riguardo al funzionamento del periodo ipotetico, ma se ancora ve ne fossero, riprovate a trascrivere il listato contenuto in questo paragrafo e ad analizzarlo PRIMA di lanciarlo; poi lanciatelo – anche più volte, per sperimentare le sue risposte a seconda delle vostre risposte –, osservate l'output.
Dovrebbe bastare.

L'uguale non è sempre uguale

Un altro dettaglio, del listato che abbiamo appena visto, che potrebbe lasciarvi perplessi è la presenza del doppio uguale all'interno delle protasi.

Si tratta, ovviamente, di un operatore di confronto che verifica l'uguaglianza tra quanto sta alla sua destra e quanto sta alla sua sinistra. La geminazione (il raddoppiamento) dei segni di uguaglianza dipende dalla differente modalità nella quale troviamo il segno uguale usato qui rispetto alla modalità nella quale lo troviamo quando viene attribuito un valore ad una variabile, come nel caso:

$variabile = 5;

L'uguale significa in entrambi i casi [è uguale a], ma nel caso della verifica si tratta di una eventualità, una possibilità da verificare e quindi non necessariamente realizzata; nel secondo si tratta di una attribuzione, necessariamente vera.

Per distinguere la modalità della possibilità[34] da quella della realtà, è stato escogitato il trucco di scrivere il primo uguale raddoppiato.

Come dispositivo di memotecnica per ricordare che la modalità della possibilità richiede il raddoppiamento, potete ricordarvi i segni <= (minore o uguale), >= (maggiore o uguale) e != (diverso, non uguale), che non sono grammaticali se impiegati nell'attribuzione di un valore di variabile[35].

4.2.2 Fintantoché...

La struttura di controllo `while` è molto potente, bella e usata. Si tratta di un ciclo iterativo condizionale: *se/finché permane una data condizione, esegui il blocco di codice contenuto nella principale.*

Ma tradotto in questo modo risulta ovviamente ostico. Vi farò un esempio, poi vedremo un listato e infine lo commenteremo.

L'esempio: la subordinata temporale introdotta da *while* viene usata per leggere un file. Si scrive all'interprete perl di aprile un file, poi si aggiunge: *finché c'è file...* fai qualcosa, ad esempio leggilo con attenzione e copialo.

[34] Che forse Amedeo Conte definirebbe deontica.

[35] Esattamente come il singolo segno di uguale è agrammaticale – e illogico, in quanto sempre vero – se usato in una verifica.

Ecco, visto che ho fatto un esempio in cui si legge e si copia un file, adesso vediamo il listato che compie l'operazione. Ma prima tornate all'ultima espressione in corsivo che ho scritto: *finché c'è file*, cioè: fino a quando esiste qualcosa che tu possa leggere, fino a quando il file non finisce. Chiaro?
Bene, eccovi lo script:

```
print "scrivi il nome del file che devo copiare: ";
$file = <STDIN>;
chomp($file);
open(ORIGIN, "<$file");
$nuovo = $file."_copia";
open(COPIA, ">$nuovo");

while(<ORIGIN>)
    {
    print COPIA $_;
    }

close(ORIGIN);
close(COPIA);

print STDOUT "ho copiato: $file in $nuovo\n";
```

Le cose potevano essere scritte in maniera informaticamente più sintetica, ma così si capisce bene quello che accade veramente.

Le prime linee non dovrebbero contenere sorprese: alla prima si richiede il nome del file da copiare; nella seconda si memorizza tale nome in una variabile; nella terza si ciompa la variabile, nella quarta si apre il file in lettura con filehandle `ORIGIN`.
Si capisce che il file è aperto in lettura grazie al segno di minore (`<$file`). In questo contesto, nel quale non può significare [minore di], immaginatelo come un occhio aperto sul testo, visto in sezione, oppure come un imbuto che "raccoglie" il testo e lo indirizza nel filehandle.
Molte parole, ma la sintassi è abbastanza semplice.

Neanche la quinta riga dovrebbe comportare sforzi cerebrali da sistema di equazioni non lineari: si crea il nome del nuovo file prendendo il nome del file originale e giustapponendovi la stringa *_copia*. Nella riga successiva, poi, si apre in scrittura il nuovo file con il nuovo nome.
Che il file è aperto in scrittura lo si capisce dal segno di maggiore (`>$nuovo`), che potete immaginare di nuovo come un imbuto oppure come la punta di una matita pronta a scrivere.

Se non lo avete fatto prima, provate a lanciare questo programma adesso, così per vedere che succede.

Alla settima riga di codice iniziano le cose interessanti: la frase perl può essere tradotta con: *finché riesci a leggere ORIGIN....* Ci sono due fatti notevoli: il primo sono le parentesi angolari all'interno delle parentesi tonde. Potete immaginarle come due occhi molto stilizzati che guardano – leggono – il contenuto del filehandle, visto che il loro significato è esattamente questo.
Il secondo fatto notevole è la presenza del filehandle, e non della variabile, nella subordinata temporale. Perché lì compare il filehandle e non la variabile?

Dipende dalla funzione del filehandle, che è uno strumento dell'interprete perl, il quale legge i file in due modi diversi. Allora, innanzitutto l'interprete è in grado di una visione generale impressionante: quando apre il file, lo acquisisce con un solo sguardo (se l'ha aperto in lettura, è ovvio), ma è come se avesse fra le mani una materia senza rendersene conto. Per poter iniziare a manipolare questa materia che già possiede deve nominarla, battezzarla come Adamo nel guardino dell'Eden. Finché ciò che ha visto non ha un nome (il filehandle), l'interprete prova quella sensazione quasi spiacevole che proviamo noi quando abbiamo capito qualcosa ma non siamo ancora in grado di razionalizzare e verbalizzare quello che abbiamo capito. È un'intuizione che non permette ulteriori elaborazioni. Dunque questa intuizione è il risultato della funzione che apre in lettura in file. Dopo che il file ha ricevuto un gestore di file, può essere letto con attenzione e consapevolezza dall'interprete e tale lettura attenta viene svolta dalla riga `while{<>}`.

Quindi attenzione, sono entrati in gioco quattro elementi: il file, il suo nome, la percezione del file da parte dell'interprete, il nome che si dà a tale percezione al fine di congetturare o lavorare a partire da tale percezione.

Kant avrebbe parlato di fenomeni (i nomi) e noumeno (il file), forse. Saussure di referente (il file), significato (forse la sua percezione) e significante (il nome della percezione), mentre il principio di arbitrarietà delle lingue non gli permetteva di presupporre un'entità noumenica e formale ad un tempo come il nome del file. Pierce avrebbe magari usato il concetto di interpretante per riferirsi all'insieme di percezione e nome della percezione. E potrebbe venire alla mente la quadripartizione di Hjelmslev: sostanza del contenuto (file), sostanza dell'espressione (percezione), forma del contenuto (nome del file), forma dell'espressione (nome della percezione); ma, come potete vedere, si tratterebbe anche in questo caso di un accostamento improprio.

Il punto è che l'interprete ha un proprio modo di agire, e che ci serve capirlo a grandi linee indipendentemente da come agiamo noi, anche se è legittimo provare ad interpretare il suo modo di agire alla luce di come agiremmo noi.

Ma torniamo al codice.

Dunque la linea con `while(<ORIGIN>)` legge riga per riga il testo contenuto nel file, e poi cosa fa?
Scrive, con *print*, sul filehandle *COPIA*. Osservate il secondo argomento della funzione `print()`: si tratta della variabile predefinita *$_* che abbiamo già visto e che significa [quello che sto leggendo adesso]. Quindi legge e mentre legge copia in un altro file. Quando ha finito chiude i due filehandle con la funzione `close()` e avverte nello standard output di

avere finito.
Tutto sommato piuttosto intuitivo, no?

E questo – con l'eccezione dell'avvertimento nell'ultima linea di codice – è il modo in cui in qualsiasi sistema operativo un file viene copiato.

Non avrete davvero creduto che windows semplicemente prendesse l'icona e copiasse? E come faceva a sapere che cos'era quell'icona?

4.2.3 Per

Spiego il ciclo `for` prima di altre cose[36] come battesimo del fuoco: i sopravvissuti al `for`, oltre a ritrovarsi in locali per veterani a ricordare il loro primo contatto con questo costrutto, possono tranquillamente affrontare anche diversi mesi di addestramento nella Tana delle Tigri[37].

Parlando di hobbit ho già menzionato il ciclo `for`. Da allora sono passate numerose pagine, e vale la pena di fare un esempio in cui il ciclo iterativo in questione sia impiegato in maniera un po' più ragionevole. La locuzione *un po'* è indispensabile, perché questa struttura di controllo viene usata raramente per la ricerca o l'analisi dei testi, e quindi il nostro esempio non potrà che essere soltanto *un po'* ragionevole.

Vediamolo subito:

```
print "scrivimi il nome della cartella nella quale devo
guardare: ";
$cartella = <STDIN>;
chomp($cartella);

opendir(GUARDONE, $cartella);
@tuttifiles = readdir(GUARDONE);
$quantifiles = @tuttifiles;
for($c=0;$c<$quantifiles;$c++)
    {
    print "$c) $tuttifiles[$c] \n";
    }

closedir(GUARDONE);
```

Le prime tre linee dovrebbero essere ormai roba vecchia: il listato chiede un input, lo memorizza in una variabile, la ciompa.

[36] Chi mi ha seguito fin qua dovrebbe sapere perché, ma se non lo ricordasse può leggere la nota a pag. 35.

[37] E se siete così giovani da ignorare di cosa si tratti, dovreste essere costretti a cantare la sigla dell'Uomo Tigre ogni domenica mattina, verso le nove.

Poi compare la prima nuova funzione, `opendir()`, il cui significato e funzionamento sono comprensibili: il verbo *opendir* apre una directory – o cartella – e ha due parametri: un dirhandle (un filehandle per directory) e il nome della cartella da aprire. È molto simile al verbo *open* che apre i singoli documenti, ma è necessario conoscere una differenza tra i set di argomenti di entrambi i verbi[38], una differenza i cui effetti, eminentemente sintattici, si vedranno a partire dalla prossima linea di codice.

Però, prima di individuare e spiegare questa differenza (se già non l'avete notata voi), facciamo un passo indietro e pensiamo a cosa distingue un file da una cartella. A livello intuitivo la differenza è abbastanza evidente: una cartella contiene dei files, un file non contiene cartelle o altri files. Ma questa è una visione fortemente mutuata dalle interfacce grafiche: in realtà – come qualsiasi utente smaliziato di sistemi operativi X-like sa – una cartella non è altro che un tipo particolare di file, che contiene i riferimenti a locazioni di memoria che corrispondono a documenti o altre cartelle. Non sono il livello intuitivo o lo sguardo "grafico" che ci interessano. Pensate, invece, alla struttura di un file-file e a cosa lo distingue da un file-cartella. Cosa può contenere un documento e che cosa una cartella? Il primo può contenere qualsiasi cosa, la seconda no. Il primo è simile ad una variabile non strutturata (una stringa), la seconda è più simile ad una variabile strutturata: un array che contiene tutti gli altri file o cartelle[39].

È chiara la differenza? Una cartella contiene un insieme, un elenco di cose che sono state memorizzate sotto una comune etichetta (il nome della cartella, appunto).

Poiché le due materie (documenti e cartelle) si prestano a differenti tipi di manipolazioni, sono state pensate specifiche funzioni per le une e le altre e specifiche strutture sintattiche.

Torniamo alla line di codice: `opendir(GUARDONE, $cartella)`. La cosa che, rispetto alla funzione sinonima `open()`, manca nell'esplicitazione del set degli argomenti è la specificazione di come la cartella debba essere aperta: in lettura, in scrittura, entrambe. Non c'è, non viene detto: lo vedete? c'è solo il nome della cartella.

Nella linea successiva l'interprete perl scopre come deve usare la cartella aperta: leggendola con `readdir()` e immagazzinando il risultato della lettura in un array chiamato *@tuttifiles*.

È legittimo domandarsi perché farlo in questo modo e non con una struttura di controllo come `while`. La differenza è essenzialmente dovuta al fatto che il numero di operazioni svolte sul contenuto di una directory è relativamente basso e ben definito: la si apre, la si chiude, al si legge, vi si fanno ricerche. I nomi dei file possono essere modificati, ma a quel punto si tratta di altre e diverse operazioni su specifici elementi. Il lavoro sul contenuto di

[38] Notate che il numero di valenze è sempre lo stesso, due, e che gli argomenti che saturano tali valenze sono uguali.

[39] Che sono, certo, a propria volta array. Ma come abbiamo visto, il perl permette che un array contenga altri array che contengono altri array che contengono altri array...

un file è infinitamente meno prevedibile e soprattutto può essere più complesso e richiedere quindi soluzioni meno rapide (ovviamente una funzione che svolge uno specifico compito è più veloce da usare di una struttura di controllo) ma più duttili e personalizzabili.

La sesta linea di codice è un trucco magico del perl, che abbiamo già visto: se si crea una variabile scalare e le si attribuisce come valore un array, il valore dello scalare corrisponde al numero degli elementi dell'array[40]. Questo valore mi serve per sapere quando il mio ciclo `for` si deve fermare.

Rileggiamo le subordinate temporali (coordinate per asindeto) rette dalla congiunzione *for*:

- avendo un numero `$c` uguale, all'inizio, a zero;
- continuando ad agire finché il numero `$c` è inferiore al numero degli elementi della lista `@tuttifiles` (cioè al numero dei files nella cartella);
- e aumentando di uno il numero `$c` ad ogni ripetizione del ciclo...

(aggiungo per completezza:) *scrivi il numero* `$c` *adatto seguito dal corrispondente nome del documento e vai a capo.*

Adesso tutto va spiegato meglio.

Immaginate che nella cartella `ciccio` siano presenti quattro files: `la_cosa, l_uomo_elastico, la_torcia_umana, la_donna_invivibi- le`.

Alla fine delle prime tre linee di codice, il listato avrà una variabile `$cartella` uguale a *ciccio*.
Nella quarta linea apre la cartella e nella quinta ne legge il contenuto e lo memorizza nella variabile `@tuttifiles`.
Nella sesta linea scopre che il numero degli elementi contenuti nella cartella *ciccio* è uguale a quattro e memorizza questo valore nella variabile `$quantifiles`.

Nella settima linea inizia il ciclo e viene istanziata una variabile `$c` con valore uguale a zero. L'interprete verifica che `$c` sia inferiore a `$quantifiles` (che sappiamo essere uguale a 4), ed esegue.

Scrive `$c`, che adesso è uguale a zero (lo abbiamo appena visto), poi scrive una parentesi chiusa, uno spazio e il valore dell'elemento zero dell'array `@tuttifiles`. Perché? Guardate, scrive: `$tuttifiles[$c]`, cioè, poiché `$c` è uguale a zero, scrive il valore di `$tuttifiles[0]`: il primo elemento dell'array, *la_cosa*.

Poi incrementa `$c` di uno (`$c++`, ve ne ricordate?) e ripete la verifica: il nuovo valore di `$c` è ancora inferiore a `$quantifiles`? Certo: uno è inferiore a quattro. Allora esegue di nuovo: scrive *$c) $tuttifiles[$c] \n*, cioè: *1) $tuttifiles[1] \n*, ovvero: *1) l_uomo_elastico \n*.

[40] Quindi la variabile `$quantifiles` corrisponde al numero di documenti che la funzione `readdir()` legge nella directory.

Poi incrementa `$c` di uno e ripete la verifica: il nuovo valore di `$c` è ancora inferiore a `$quantifiles`? Certo: due è inferiore a quattro. Allora esegue di nuovo: scrive *$c) $tuttifiles[$c] \n*, cioè: *2) $tuttifiles[2] \n*, ovvero: *2) la_torcia_umana \n*.

Poi incrementa `$c` di uno e ripete la verifica: il nuovo valore di `$c` è ancora inferiore a `$quantifiles`? Certo: tre è inferiore a quattro. Allora esegue di nuovo: scrive *$c) $tuttifiles[$c] \n*, cioè: *3) $tuttifiles[3] \n*, ovvero: *3) la_donna_invivibile \n*.

Poi incrementa `$c` di uno e ripete la verifica: il nuovo valore di `$c` è ancora inferiore a `$quantifiles`? No: quattro è uguale a quattro: esce dal ciclo e chiude il dirhandle.

Un programma che avete già

Il meccanismo iterativo del `for` dovrebbe esservi chiaro, adesso.

Se provate a lanciare il vostro programmino avrete un risultato strano, però: i primi due elementi della lista non risultano essere nomi di documenti o cartelle ma rispettivamente un punto e un doppio punto.

Ho già detto che le directory contengono dei nomi e degli indirizzi nella memoria, e all'inizio di questo manuale ho spiegato come muoversi attraverso le cartelle (ricordate il comando `cd`? Ne ho parlato nella parte sull'esecuzione degli script).

Nel linguaggio della shell il doppio punto significa [sali di un livello], mentre il punto significa [qui], ed è questo che significano le prime due voci del risultato del vostro listato.

Ma quello che con sudore e sangue avete ottenuto digitando direttamente il codice perl era già presente sulle vostre macchine. Se nella shell di X digitaste:
`$> ls -al ciccio` [invio]
oppure nella shell di Windows digitaste:
`$> dir ciccio` [invio]
otterreste:

```
.
..
la_cosa
l_uomo_elastico
la_torcia_umana
la_donna_invivibile
```

Con tutte le differenze del caso (sia `ls` che `dir` restituiscono risultati con molta più informazione), entrambi i programmi funzionano così; leggono e scrivono grosso modo seguendo questa stessa logica.

E, non illudetevi, anche la più puffosa delle interfacce grafiche in fondo legge e stampa nello stesso modo.

4.2.4 Perogni

La struttura di controllo `foreach` è mooolto più usata di `for` da chi, come noi, si occupa di analizzare/archiviare/fare-altro dei testi.

Ad esempio, riprendiamo l'ultimo listato che abbiamo visto, quello che legge il contenuto delle cartelle. Sta a pagina 62, ma lo riporto per comodità:

```
print "scrivimi il nome della cartella nella quale devo
guardare: ";
$cartella = <STDIN>;
chomp($cartella);

opendir(GUARDONE, $cartella);
@tuttifiles = readdir(GUARDONE);
$quantifiles = @tuttifiles;
for($c=0;$c<$quantifiles;$c++)
    {
    print "$c) $tuttifiles[$c] \n";
    }

closedir(GUARDONE);
```

Non dovreste avere bisogno di spiegazioni. Ora osservate come cambia se invece di `for` usiamo `foreach`:

```
$c = 0;
print "scrivimi il nome della cartella nella quale devo
guardare: ";
$cartella = <STDIN>;
chomp($cartella);

opendir(GUARDONE, $cartella);
@tuttifiles = readdir(GUARDONE);
foreach $elemento (@tuttifiles)
    {
    print "$c) $elemento \n";
    $c++;
    }
closedir(GUARDONE);
```

Visto che abbiamo già trattato il ciclo `foreach`, non dovrei rispiegarlo, ma preferisco abbondare e se davvero vi sentite sicuri su questo argomento potete saltare il resto del paragrafo e procedere a partire dal paragrafo successivo.

La creazione della variabile `$elemento` nel ciclo `foreach` è qualcosa di simile ai filehandle: un nome che è necessario dare ad un ente perché l'interprete possa manipolare quell'ente. L'ente in questione è un elemento dell'array `@tuttifiles`. Già, ma quale? Il valore della variabile `$elemento` va interpretato alla luce della struttura sintattico-semantica all'interno della quale si trova: *foreach*.
È come se la lista venisse letta un elemento alla volta, in modo da permettere una o più operazioni su questo elemento (anche una semplice stampa dell'elemento stesso). Allora la prima iterazione `$elemento` significherà [.], la seconda significherà [..], la terza [la_cosa], la quarta [l_uomo_elastico] e così via.

Due conseguenze di questa differente struttura:

1. poiché il ciclo `foreach` lavora direttamente sui valori dell'array, e non su numeri che possono essere usati come chiavi dell'array, non abbiamo bisogno di scrivere *$tuttifiles[$c]* menzionando esplicitamente l'array: ci basta ricorrere a `$elemento`. Questa è una buona semplificazione;

2. però noi il numero d'ordine di ogni elemento continuiamo ad usarlo e volerlo, quindi abbiamo bisogno di un incrementatore. Nel ciclo `for` stava fra le condizioni del ciclo, ma `foreach` implica una diversa sintassi, e l'incrementatore compare all'interno della principale (fra le parentesi graffe). Inoltre, l'ho istanziato e gli ho attribuito un valore nullo all'inizio dello script, per chiarezza e semplicità.

ESERCIZIO: anche se continuiamo a volere l'incrementatore, il primo elemento uguale a zero è bruttino. Ora, con la struttura `for` era necessario partire da zero, perché si agiva direttamente sull'array e il primo elemento di un array è quello con il numero zero, ma adesso possiamo farne a meno. Immaginiamo di volere che il primo elemento sia uguale a uno: cosa bisogna fare? Pensateci, non è difficile.

AIUTINO: basta fare uno spostamento. Sì, avete letto bene: basta spostare una linea di codice da un punto ad un altro. Provate a rileggere il codice e a capire che cosa bisogna spostare...

RI-AIUTINO: e va bene: bisogna spostare l'incrementatore. Riuscite ad immaginare dove? È quasi un gioco; immaginate cosa accadrebbe se l'incrementatore stesse in ogni altro punto del listato. Ed escludete tutti i punti del listato nei quali non ha senso che stia l'incrementatore.

SOLUZIONE: guardate dove si trova l'incrementatore. Adesso riuscite a capire perché l'output cambia?

```
$c = 0;
print "scrivimi il nome della cartella nella quale devo
guardare: ";
$cartella = <STDIN>;
chomp($cartella);
```

```
opendir(GUARDONE, $cartella);
@tuttifiles = readdir(GUARDONE);
foreach $elemento (@tuttifiles)
    {
    $c++;
    print "$c) $elemento \n";
    }
closedir(GUARDONE);
```

Serve davvero una spiegazione?

ESERCIZIO: immaginate di NON volere assolutamente vedere i primi due elementi contenuti nella directory. Ovvero, immaginate di desiderare di vedere tutti gli elementi che non sono uguali a [.] e a [..]. Come potete fare? Pensateci, prima di andare a sbirciare la soluzione.

AIUTINO: serve un costrutto condizionale, evidentemente.

SOLUZIONE (VERGOGNA!, se avete letto subito la soluzione): Se l'elemento di turno è uguale a [.] oppure a [..] bisogna passare all'elemento successivo.
Questa risposta bastava.

Ne esistono due possibili implementazioni:

1. se l'elemento di turno è uguale a [.] oppure a [..] bisogna passare al successivo;
2. se l'elemento di turno è diverso da [.] oppure da [..] bisogna scriverlo.

Ecco la prima implementazione:

```
$c = 0;
print "scrivimi il nome della cartella nella quale devo
guardare: ";
$cartella = <STDIN>;
chomp($cartella);

opendir(GUARDONE, $cartella);
@tuttifiles = readdir(GUARDONE);
foreach $elemento (@tuttifiles)
    {
    if($elemento eq ".") {next;}
    if($elemento eq "..") {next;}
    $c++;
    print "$c) $elemento \n";
    }
closedir(GUARDONE);
```

Le prime due proposizioni della principale retta dal *foreach* sono due periodi ipotetici molto semplici, osservateli: nella protasi vediamo la parola *eq* [equal], usata per confrontare le stringhe di testo.

Diverso, invece, si scrive *!=* per i numeri, e *ne* [not equal] per le parole. Vale la pena di mostrare una semplice tabellina riassuntiva:

	numeri	parole
uguale	==	eq
diverso	!=	ne

Tabella 4.1: Numeri uguali e diversi

Grazie a questa tabella potete provare a scrivere (prima di leggerlo qui sotto) il listato in modo da realizzare la seconda implementazione.

Fatto?

Bene, potete allora controllare di aver scritto correttamente:

```
$c = 0;
print "scrivimi il nome della cartella nella quale devo
guardare: ";
$cartella = <STDIN>;
chomp($cartella);

opendir(GUARDONE, $cartella);
@tuttifiles = readdir(GUARDONE);
foreach $elemento (@tuttifiles)
    {
    if($elemento ne "." && $elemento ne "..")
        {
        $c++;
        print "$c) $elemento \n";
        }
    }
closedir(GUARDONE);
```

In questo esempio ho preferito comprimere del codice: dovevano esserci due cicli condizionali uno dentro l'altro:

```
se ($elemento e' diverso dal punto)
    {
    se($elemento e' diverso dal doppio punto)
```

```
    {scrivi l'elemento}
}
```

e invece ho riformulato come: *se ($elemento è diverso dal punto E dal doppio punto)*. La congiunzione all'interno della parentesi è data dalle due e commerciali[41].

Per tornare a quel che più ci interessa, la sintassi della struttura di controllo `foreach` è la seguente:

definitore_del_ciclo elemento_variabile (array_da_percorrere)

Dove l'array da percorrere deve sempre essere un array, e deve essere racchiuso tra parentesi semplici.

Come abbiamo visto, la struttura `foreach` lavora sui valori della lista che legge uno alla volta; se desiderate farla lavorare sulle chiavi[42], dovete generare un array con tutte le chiavi. Abbiamo visto come si fa a pag. 34 e a pag. 35; potete tornare a controllare quegli esempi.

Adesso dovrebbe essere tutto molto più facile.

4.3 Sala operatoria

Gli operatori sono una delle cose più noiose di tutto il linguaggio. Ma si usano, servono, funzionano bene e non sono ambigui, quindi bisogna conoscerli.
Tiriamoci su le maniche (non crediate che descriverli sia più divertente che leggerne) e finiamo il capitolo.

Cos'è esattamente un operatore? Qualcosa che fa delle operazioni. Come le funzioni? No, non esattamente.

Come avremo modo di vedere, esistono vari tipi di operatori: operatori matematici, logici, di confronto, di assegnamento, di corrispondenza pattern eccetera eccetera. Esistono anche, però, diversi tipi di funzioni: numeriche, per le espressioni regolari, per l'elaborazione di scalari, o di array, o di hash, di input ed output, di filehandle eccetera eccetera.

Ma se le funzioni sono i verbi della lingua perl, gli operatori rappresentano un aspetto originale dei linguaggi di programmazione rispetto alle lingue storico-naturali: si tratta di un insieme di verbi, avverbi, interi sintagmi verbali e nominali e addirittura costrutti sintattici (come nel caso dell'operatore condizionale, che non abbiamo ancora visto).

[41] *ne, eq, ==, !=, &&* sono operatori, e li vedremo nel prossimo paragrafo.

[42] Di un hash, ovviamente. Per lavorare sulle chiavi di un array – che sono una lista ordinata di numeri (vale la pena di ricordarlo) – vi conviene usare un incrementatore o la struttura di controllo `for`.

Un analogo realistico nella grammatica italiana sono gli avverbi: una classe di parole eterogenea, chiusa, costituita da forme invariabili, non meglio definibile[43]. Rispetto agli avverbi, gli operatori sono funzionalmente più eterogenei (hanno anche funzione predicativa).

Nei seguenti sottoparagrafi vedremo solo gli operatori di uso più frequente e quelli che sono presenti in questo manuale.

Operatori aritmetici e di autoincremeneto e autodecremento

Via, li conoscete già[44]: sono quelli che vi permettono di fare operazioni aritmetiche. Più qualcos'altro di carino che vediamo subito.

segno	traduzione	esempio
+	più	5+6 (=11)
-	meno	6-5 (=1)
*	per	3*5 (=15)
/	diviso	20/5 (=4)
**	elevato alla...	3**4 (=81)
–	meno uno	$c– (=$c-1)
++	più uno	$c++ (=$c+1)

Tabella 4.2: Operatori di autoincremento e autodecremento

Giusto una nota sull'elevamento a potenza: 2**3**3 viene letto come 2**27 e non come 8**3. Magari vi faceva piacere saperlo.

Operatori di assegnazione

Abbiamo già visto un operatore di assegnazione (circa ottocentomila volte), e ne abbiamo descritto le differenze rispetto ad un operatore di confronto che ha la stessa traduzione in lingua corrente ma una diversa modalità d'uso.

Avete capito di cosa sto parlando?

Negli esempi della tabella a pag. 72, attenti al valore della variabile `$c` nella prima linea e tutte le sue trasformazioni ad ogni passo.

[43] Una vera indecenza, come classe. Se la descrizione della lingua fosse una casa signorile, la classe degli avverbi sarebbe l'armadio pieno di scheletri: da quando esiste un'idea di grammatica, ogni volta che un grammatico non sa dove ficcare qualche parte del discorso, oplà, via negli avverbi.

[44] E potete ripassarli nella tabella a pag. 71.

segno	traduzione	esempio
=	uguale a	$c = 5;
+=	(numero) se stesso più	$c += 6; (*ergo* $c=11)
-=	se stesso meno...	$c -= 3; (*ergo* $c=8)
*=	se stesso per...	$c *= 3; (*ergo* $c=24)
/=	se stesso diviso...	$c /= 2; (*ergo* $c=12)
.=	(stringa) se stesso più...	$c.= “fuffa” (*ergo* $c=“12fuffa”)

Tabella 4.3: Operatori di assegnazione

Operatori di confronto

Gli operatori di confronto[45] sono qui riassunti per valori numerici e per stringhe; usarli a sproposito è un errore grammaticale.

stringa	numero	significato
eq	==	uguale a
ne	!=	diverso da
cmp	<=>	confronto (usi particolari, vedrete)
gt	>	maggiore di
ge	>=	maggiore o uguale a
lt	<	minore di
le	<=	minore o uguale a

Tabella 4.4: Operatori di confronto

La struttura sintattica degli operatori di confronto è abbastanza ordinaria: il confronto avviene tra quel che sta a destra dell'operatore e quel che sta alla sua sinistra.

Operatori logici

Gli operatori logici[46] sono [e] e [oppure] e sono usati per valutare la veridicità di una affermazione.

Ad esempio quando voglio sapere se l'affermazione *A e B* (il maglione è giallo E rosso) è vera oppure falsa, oppure se lo è l'affermazione *A oppure B* (il maglione è giallo O rosso). Avete visto un confronto del primo tipo nell'ultimo listato del paragrafo precedente: *if($elemento ne . && $elemento ne ..)*. In questa protasi si verificava che il file preso in esame fosse diverso dal punto E che fosse diverso dal doppio punto.

[45] Tabella a pag. 72.
[46] Tabella a pag. 73.

Se avessi voluto comprimere nello stesso modo (con un solo periodo ipotetico) anche l'implementazione alternativa, quella nella quale si procedeva oltre se `$elemento` era uguale al punto oppure se era uguale al doppio punto, avrei dovuto scrivere, invece di:

```
if($elemento eq ".") {next;}
if($elemento eq "..") {next;}
```

```
if($elemento eq "." || $elemento eq "..") {next;}
```

Cioè: *se l'elemento è uguale al punto OPPURE l'elemento è uguale al doppio punto, passa all'elemento successivo.*

Ecco dunque gli operatori logici:

segno	traduzione	esempio	risultato
&&	e	$a && $b	vero solo se entrambi sono veri
||	o	$a || $b	vero se almeno uno dei due è vero

Tabella 4.5: Operatori logici o Booleani

Operatori di confronto pattern e altri operatori

Gli operatori di confronto pattern sono molto importanti; li si poteva lasciare insieme ad altri operatori di confronto, ma preferivo che li aveste ben chiari in mente. Sono facili: ce ne sono due, significano [uguale] e [diverso] e sono usati nelle espressioni regolari che vedremo nel prossimo capitolo.

Guardateli nella tabella a pag. 73 e fingete che non vi lascino indifferenti, per ora sarà più che sufficiente.

segno	traduzione	esempio
=~	uguale	$testo =~ /sesso/
!~	diverso	$testo !~ /sesso/

Tabella 4.6: Operatori per le espressioni regolari

Abbiamo quasi finito, siamo stati bravi.

Esistono diversi altri tipi di operatori: gli operatori di test di file, l'operatore *range*, l'operatore condizionale, l'operatore virgola, l'operatore freccia, l'operatore stringa[47]. E

[47] L'operatore stringa non è altri che il punto come segno di concatenazione. Come in *$c .= "mento"* e come in *print $testo. "\n";*.

degli stessi tipi di operatori che abbiamo visto, ad esempio gli operatori di assegnazione, alcuni sono stati omessi a vantaggio di una maggiore rapidità ed essenzialità del manuale.

Ma se aveste voglia di approfondire sia la comunità perl che i manuali in rete (che quelli cartacei, ovviamente) sono aperti e disponibili ad ulteriori e più approfondite consultazioni.

4.4 In principio era la funzione

Dunque, abbiamo iniziato con una funzione, quindi in principio era il verbo *scrivi*: da allora ho scritto moltissimo (e voi avete letto immagino almeno altrettanto).

In questo brevissimo paragrafo non vi fornirò nuove nozioni: quelle che avete acquisito fin qui sono più che sufficienti per tutto quel che segue, ma desidero far baluginare di fronte ai vostri occhi almeno una possibilità, farvene avere un sentore in modo da, eventualmente, spingervi ad approfondire al di là della rozza introduzione che il testo che state leggendo rappresenta.

Esistono numerose funzioni, come ho avuto modo di scrivere, che possiamo distinguere in grandi famiglie a seconda dell'oggetto sul quale lavorano (testo, numero, file, directory, processi). Ma se anche fossero molte di più, probabilmente non basterebbero per scrivere tutto quel che si può scrivere con il perl.

Esiste una procedura per creare funzioni definite dall'utente: frammenti di codice che poi, esattamente come le normali funzioni, possono essere richiamati e dotati di argomenti e che danno dei risultati.

È davvero utile? Oppure è solo una inutile complicazione? Pensateci.

Capitolo 5

Lode al Signore: le regexp

5.1 Esprimere le regolarità

Benvenuti nel quinto capitolo, che tratta quel linguaggio nel linguaggio che per molte persone è il vero motivo per il quale vale la pena di studiare il perl. Intendiamoci, le espressioni regolari[1] sono ormai comuni a tutti i linguaggi di programmazione e di scripting – anche javascript le ha[2] –, ma si sono sviluppate fino allo stato dell'arte in perl, ed è in questo linguaggio che continuano ad esprimersi al loro meglio.

Ma cosa sono le espressioni regolari? Esattamente quel che dice il loro nome: *espressioni* cioè descrizioni, testi, *regolari*. Ecco, questo secondo aggettivo richiede qualche spiegazione.

Una espressione regolare è una descrizione in un linguaggio formale (una espressione) di una regolarità di un testo (a livello di significante, grafematico, ovviamente).

Cosa sono, allora, le regolarità di un testo? Ce ne sono di vari tipi. Ad esempio, costituiscono una regolarità del presente manuale tutte le parole che finiscono in -are, o tutte quelle che iniziano in pre-. Ma sono ugualmente regolarità tutte le occorrenze di una qualsiasi parola o di un qualsiasi numero di parole una vicina all'altra caratterizzate in qualche modo.

Desiderate trovare la parola *carne* e tutte le sue forme e i suoi derivati in un testo? State cercando una regolarità all'interno di quel testo.

State cercando tutte le preposizioni o tutti gli indirizzi di e-mail o tutti i verbi all'infinito seguiti da una preposizione presenti in una raccolta di testi? Forse non senza errori ed omissioni, ma verosimilmente una espressione regolare fa al caso vostro.

Le espressioni regolari sono usate per cercare all'interno di un testo e per sostituire all'interno di un testo.

In questo capitolo, diversamente da quanto accade nel resto del manuale, non troverete

[1] *Regular expressions*, da cui il termine gergale *regexp*.
[2] E ho detto tutto.

molti esempi di codice; si tratta di un capitolo di riferimento, utile per capire quello che sarà scritto nel codice del prossimo capitolo. Tuttavia, se volete fare qualche esperimento, potete usare uno dei due script presenti nei prossimi due paragrafi.

Per usare i listati dovete scrivere:
`$> perl nomelistato.pl file_nel_quale_cercare` [invio]
dove il `file_nel_quale_cercare` NON deve contenere spazi[3].

Per sperimentare, se non avete un vostro corpus (i vostri articoli, un codice legislativo, il romanzo nel cassetto) in forma digitale e pronto ad essere brutalizzato dagli esperimenti con perl, vi consiglio di scaricare materiale da due siti che sono anche meravigliose iniziative in rete[4]: www.liberliber.org e www.projectgutemberg.org.

5.2 Riconoscimento

La funzione per il riconoscimento di un pattern, di una sequenza di caratteri, si chiama `m//` (da *matching*).

Come vedete non ha le parentesi, ma due barre tra le quali bisogna scrivere quello che si vuole cercare. Il motivo di questa bizzarria risiede nel semplice fatto che le parentesi hanno un ampio uso nelle espressioni regolari, e usarle quindi sia per contenere l'espressione regolare che al suo interno avrebbe causato confusione o costretto ad altri e più confusionari artifici. Questo artificio è tutto sommato sopportabile.

Provate ad usare il seguente script per i vostri esperimenti:

```
print "scrivi quel che devo cercare: ";
$pattern = <STDIN>;
chomp($pattern);
while(<>)
    {
    if($_=~ m/$pattern/g)
        {print STDOUT $_;}
    }
```

Come vedete, dopo la seconda barra c'è una *g*. La strana sintassi delle funzioni delle espressioni regolari prevedono che all'interno delle barre sia compreso il pattern da cercare nel testo – l'espressione regolare – mentre all'esterno, dopo, si trovino alcuni specificatori che definiscono come il riconoscimento deve avvenire.

Gli specificatori, che possono essere semplicemente giustapposti, sono:

g che significa [global], cioè si verifica un riconoscimento ogni volta che la sequenza (il pattern) compare nel testo analizzato. Se non specificassimo che la ricerca deve essere globale, il programma si fermerebbe alla prima occorrenza del pattern;

[3] Gli spazi nei nomi di file sono una maledizione peggiore della confusione delle lingue di biblica memoria.
[4] Paragonabili alla Wikipedia per la meraviglia che suscitano.

i che significa [case insensitive], cioè insensibile alle maiuscole. Se avessimo specificato questa opzione di confronto, la ricerca non avrebbe distinto tra *Franco* (il nome proprio) e *franco* (l'aggettivo o il nome comune);

m che significa [multiple]. Tratta la stringa come linee multiple. Non so precisamente in quali casi è conveniente usare questa opzione; a me non è mai capitato, ma magari potrebbe, un giorno, servirvi;

o che significa [once], cioè il pattern è compilato una sola volta. Rende il codice insensibilmente più veloce, tutto qui;

s che significa [single], cioè la stringa viene trattata come una linea singola. Serve quando si desidera cercare tutto ciò che sta dopo un carattere di a capo;

x che significa [extended]. Ebbene sì, esistono delle espressioni regolari estese, che complicano ed arricchiscono le espressioni regolari[5].

Alcune note sul codice

Solo appunti, giacché, per ora, non è necessario approfondire.

Tra le parentesi del `while` si trova il cosiddetto operatore diamante[6].
In realtà tra le parentesi angolari c'è un filehandle, solo che si tratta di un filehandle predefinito (come lo *STDOUT* nelle funzioni `print()`) e quindi, se lo omettiamo, l'interprete perl immagina che lì ci debba stare proprio quel filehandle, e che quel filehandle debba essere impiegato per leggere con attenzione dentro ai file.

Devono essere poste altre due domande: 1) come si chiama il filehandle predefinito? 2) a che cosa corrisponde, cioè: quale file apre e legge questo filehandle predefinito?

Se non sapete già rispondere ad entrambe le domande, potete dare un'occhiata a pag. 55, nella quale si parla dei filehandle predefiniti, ed eventualmente alla pagina precedente, nella quale si parla dell'array predefinito `@ARGV`.

Ma per amore della ridondanza[7] fornirò anche qui due risposte veloci:

1. il filehandle predefinito in questione si chiama ARGV: potete provare ad aggiungere la parola ARGV tra le parentesi angolari;

2. quando si lancia il programma si accoda al nome del programma stesso anche il nome del file nel quale cercare. L'interprete perl riceve questa informazione e, non vedendo alcunché tra le parentesi angolari laddove si aspetterebbe un filehandle aperto precedentemente, decide di immagazzinare l'informazione ricevuta in ARGV. Potrà così

[5] Ma qui non ne parleremo: è uno di quegli argomenti che potete approfondire in altra sede, magari parlando direttamente con dei programmatori.

[6] Oh, un operatore del quale non vi ho detto prima.

[7] Chissà che non favorisca una maggiore chiarezza...

aprire e processare automaticamente il o i file per cercare il pattern, la sequenza che ci interessa[8].

Una seconda particolarità del listato riguarda un'altra variabile predefinita. Anzi, in una certa misura la variabile predefinita per eccellenza, foss'anche solo per frequenza d'uso.

Il programma verifica che `$_` corrisponda al pattern. Come abbiamo visto, la variabile predefinita `$_` significa [la linea in esame] o [quel che sto guardando adesso], quindi l'output del programma sarà costituito dalle linee nelle quali almeno una volta compare il pattern.

Il resto del codice dovrebbe essere chiaro.

5.3 Sostituzione

Il secondo uso più frequente delle espressioni regolari è la sostituzione. La funzione della sostituzione è `s///`. Come vedete, compaiono ben TRE barre. Tra la prima e la seconda sta quel che si cerca, tra la seconda e la terza ciò con cui si sostituisce.

Di seguito è riportato uno script di sostituzione che crea una copia del file originario rinominandolo come *nomefile_vecchio* (cosicché nessun danno possa essere irreparabile) e poi modifica il file originario secondo le correzioni richieste dall'utente.

```
$^I ='_vecchio';
print "cosa devo cercare: ";
$pattern = <STDIN>;
chomp($pattern);
print "con cosa lo correggo: ";
$nuovo = <STDIN>;
chomp($nuovo);

while(<ARGV>)
    {
    $_=~ s/$pattern/$nuovo/gi;
    print $_;
    }
```

Come vedete non solo sono chieste entrambe le informazioni necessarie all'interprete per le sue sostituzioni, ma in più viene istanziata la variabile predefinita `$^I` con valore uguale a `_vecchio`.

In questo listato non è più necessario il periodo ipotetico dello script di *pattern matching*: la sostituzione deve essere eseguita sempre. Piuttosto, vi invito a meditare sull'uti-

[8] Ricordate quando avete letto che l'interprete perl, pur non manifestando l'aspetto di Nicole Kidman presenta comunque notevoli caratteristiche? Eccone una, e, forse, Nicole Kidman non conosce neanche le espressioni regolari.

lità e potenza, nel caso della sostituzione, dell'opzione g (global).

Ma se non foste sicuri di dover sostituire tutte le occorrenze di un pattern con un altro pattern, è sufficiente mettere insieme quello che abbiamo fatto negli ultimi due listati, in questo modo:

```
$^I ='_vecchio';
print "cosa devo cercare: ";
$pattern = <STDIN>;
chomp($pattern);
print "con cosa lo correggo: ";
$nuovo = <STDIN>;
chomp($nuovo);
print "\n!!!!!!!! Per confermare una sostituzione,
devi premere \"s\" e il tasto di invio; per impedirla
devi premere \"n\" e il tasto di invio\n\n";
while(<>)
    {
    $scelta = "";
    if($_=~m/$pattern/gi)
        {
        print STDOUT "sostituisco questo? => ". $_;
        $scelta = <STDIN>;
        chomp($scelta);
        }
    $_=~ s/$pattern/$nuovo/gi if $scelta eq "s";
    print;
    }
```

Dopo aver ripulito l'ultimo input dell'utente, compare un avviso (in più rispetto al codice precedente). Successivamente, all'interno del ciclo di `while` compaiono, allo stesso livello, quattro linee; osservate: l'attribuzione del valore nullo alla variabile `$scelta`; un ciclo condizionale; un altro ciclo condizionale (non lo sembra, ma poi vedrete che lo è); un ordine di stampa.

Per capire la funzione della variabile `$scelta` dobbiamo capire cosa accade dentro al primo ciclo condizionale.

Il primo ciclo condizionale si domanda se nella linea che sta guardando compaia il pattern. Se il pattern non compare, tutto il blocco di istruzioni sottoposto alla condizione verificata nella protasi viene saltato; anche il secondo ciclo condizionale (quello sulla linea che inizia con *$_ =~ s/$pattern*) salta; è invece eseguita l'istruzione *print*. Se, al contrario, il pattern compare, il programma chiede all'utente: *devo effettuare la sostituzione nella linea*

che ho appena letto (e che ti mostro)?. La risposta che l'utente, attraverso lo standard input (STDIN) fornisce, è memorizzata nella variabile `$scelta` e ciompata.

Poi è il turno del secondo ciclo condizionale. So che non lo sembra, ma leggete la terzultima linea con attenzione: ad un certo punto compare la parola *if*. Questa è una struttura sintattica del ciclo condizionale alternativa a quella vista finora. Tale struttura si può usare solo quando si vuole verificare se una data condizione è vera oppure no – ciò significa che non accetta `else` o `elsif`.
Questa struttura sintattica vede l'apodosi prima della particella *if* e la protasi dopo, come nelle frasi italiane: *finiscila pure, se davvero vuoi sbronzarti* oppure *sostituisci, se trovi la parola che va sostituita*.

L'effetto di questo secondo ciclo condizionale è dunque quello di effettuare la sostituzione se e soltanto se la variabile `$scelta` è uguale ad *s*[9].

Infine, qualsiasi siano gli esiti dei vari costrutti condizionali, il programma stampa la linea che ha appena letto. Sembra semplice, no? Ma cosa stampa? E dove?

Cosa stampa è facile: stampa `$_`, cioè quello che ha appena letto. Non è esplicitato, perché si tratta di una variabile predefinita e quando l'interprete non trova nient'altro sa che in quella posizione, con quella funzione, ci deve essere una e una sola variabile predefinita.

La risposta alla seconda domanda, dove, è un poco più difficile. Tornate indietro, a leggere nel blocco di istruzioni del primo periodo ipotetico: ci sono sia un *print* che uno *STDOUT*; badate bene al fatto che lo standard output qui è necessario, perché all'inizio del codice abbiamo dichiarato non nulla la variabile `$^I` (per la precisione abbiamo dichiarato che il suo valore è uguale a `_vecchio`).
La variabile predefinita `$^I` (*Inplace edit*) fa sì che l'interprete scriva automaticamente dove legge; quindi, in questo caso, fa sì che l'interprete legga e scriva automaticamente non nello STDOUT ma in ARGV. In una parola, se `$^I` è dichiarata e non è disabilitata, sovrascrive. Il valore di questa variabile predefinita indica se deve effettivamente sovrascrivere il file in lettura (`$^I = '';`) oppure se prima deve anche fare una copia identica (per esempio una copia con l'estensione *_vecchio* se `$^I = '_vecchio';`).

In definitiva, la forma esplicita della linea con il comando print sarebbe: `print ARGV $_`[10].

Adesso ipotizziamo un'applicazione del nostro listato di sostituzione mirata.
Partiamo dal *while*, supponendo quindi che l'interprete conosca il pattern, la sua sostituzione e il file da aprire.

Il programma apre il file (inizio del `while(<>)`), istanzia come uguale a nulla il valore della variabile `$scelta`, inizia a leggere la prima linea di codice. Trova il pattern? Immaginiamo di no; allora salta il blocco dentro le parentesi graffe del primo `if`. La variabile

[9] Sappiamo, avendolo letto nell'avviso prima del ciclo *while*, che la scelta dell'utente deve essere *s* (per: "*s*ostituisci") oppure *n* (per: "*n*on sostituire").

[10] NOTA CHE GENERA CONFUSIONE E SCONFORTO: benvenuti nel mondo dei programmatori perl malvagi, dove la conoscenza delle variabili predefinite, delle sintassi alternative e dei meccanismi di ellissi permette di scrivere codice illeggibile.

`$scelta` è uguale a *s*? Evidentemente no, visto che il suo valore non è cambiato; quindi non avviene alcuna sostituzione. Infine il programma stampa (scrive su ARGV) senza modifiche la linea che ha appena letto.

Seconda linea di testo letto dal programma: `$scelta` diventa di nuovo uguale a niente, perché le viene di nuovo attribuito questo valore (prima linea di codice tra le graffe del `while(<>)`). Trova il pattern? Immaginiamo di sì. Allora il programma chiede: *sostituisco questo?* => (segue la linea in questione). L'utente vuole che la sostituzione abbia luogo, quindi digita [s] e [invio]. A questo punto il programma si domanda: la variabile `$scelta` è uguale a *s*? Evidentemente sì, visto che tale valore è stato introdotto tramite standard input dall'utente e memorizzato proprio nella variabile `$scelta`. Quindi il programma procede alla sostituzione, modificando la linea che ha appena letto. Infine il programma stampa la linea che ha appena letto e modificato.

Terza linea del testo letto dal programma. `$scelta` diventa di nuovo uguale a niente, ormai dovreste aver visto come si fa. Trova il pattern? Immaginiamo di sì. Allora il programma chiede: *sostituisco questo?* => (segue la linea in questione). L'utente NON vuole che la sostituzione abbia luogo, quindi digita [n] e [invio]. A questo punto il programma si chiede: la variabile `$scelta` è uguale a *s*? Evidentemente no, visto che il valore introdotto dall'utente è *n*; quindi non avviene alcuna sostituzione. Infine il programma stampa senza modifiche la linea che ha appena letto.

Quarta linea di testo. `$scelta` diventa di nuovo uguale a niente. Il programma trova il pattern nella quarta linea? Immaginiamo di no, quindi salta il blocco dentro le parentesi graffe del primo `if`. La variabile `$scelta` è uguale a *s*? Evidentemente no, visto che il suo valore non è cambiato, quindi non avviene alcuna sostituzione. Infine il programma stampa senza modifiche la linea che ha appena letto.

E via discorrendo fino alla fine del testo da correggere, linea per linea.

Aaah! un baco! una baco! uccidilo!

Il programma che abbiamo appena finito di vedere ha un baco: se su una stessa linea di testo compare, mettiamo, quattro volte un'occorrenza del pattern da noi cercato, sia che io decida di correggere che decida di non correggere, le conseguenze della mia scelta si applicano a tutte le occorrenze di quella linea.

È un problema: e sei io volessi correggere solo la seconda occorrenza? Bisogna procedere in un altro modo.
Voi come fareste?

Per ora pensateci, più avanti vedremo.

5.4 Coda di rospo, ali di pipistrello...

Il cuore delle espressioni regolari non sta, ovviamente, nelle funzioni, che sono condizione necessaria ma non sufficiente. Il cuore delle espressioni regolari è una lingua formale che

descrive il testo.

Per chiarezza, distingueremo diversi tipi di strumenti di descrizione, gli ingredienti dei nostri incantesimi in perl: caratteri, quantificatori, raggruppatori, connettivi, ancore ed altro.

5.4.1 Caratteri

Un carattere, preceduto dalla barra retroversa, sta per una classe di caratteri. Per esempio, se voglio indicare un numero qualsiasi, posso scrivere *d* che sta per *digit* [numero]. Potete vederne un elenco quasi completo nella tabella a pag 82.

segno	nome	traduzione	esempi
\w	word	carattere alfanumerico	a,b,c,d,E,Z,H,5,9,0
\d	digit	numero	1,2,3,4,5,6,7,8,9,0
\s	space	spazio	[spazio]
\n	newline	nuova linea	[invio]
\f	formfeed		
\t	tab	tabulazione	
\r	return	l'a capo (solo in win)	
.	dot	ogni carattere eccetto \n	un \s o un \w
\b	border	limite di parola	

Tabella 5.1: I caratteri

I segni \w, \d, \s hanno ognuno un corrispettivo negativo scritto in maiuscolo: \W significa [tutto ciò che non è una lettera]; \D significa [tutto ciò che non è un numero]; \S significa [tutto ciò che non è uno spazio].

Il segno \b è una cosa strana: lo si usa per segnalare un limite di parola. A volte serve[11].

5.4.2 Quantificatori

I quantificatori sono abbastanza intuitivi, visto che i significati dei loro codici richiamano il significato originario dei codici stessi: si capisce in fretta perché il punto interrogativo significa zero o uno (equivale alla domanda: *il carattere seguito dal punto interrogativo c'è o non cè?*).

Due note importanti:
1) un quantificatore, da solo, non significa nulla; la sua funzione è sempre attributiva, per questo un'espressione regolare come: /?/ non solo è agrammaticale, ma è pure priva di

[11] Lo vedremo alla fine di questo capitolo.

segno	traduzione	esempi
?	sì o no (0 o 1)	/cass?a/ = *casa* e *cassa*
*	da 0 a infinito (∞)	/ca\w*/ = parole che iniziano per *ca*
+	almeno 1 (da 1 a ∞)	/can\w+/ = *canto* ma non *can*
{x,y}	(da x a y)	/\w{3,5}/ = parole di 3, 4, 5 lettere

Tabella 5.2: I quantificatori

significato;
2) i quantificatori si applicano esclusivamente a quel che li precede. L'espressione regolare /cass?a/ riconosce *casa, cassa* e non *cass* (il morfema flessivo *a* deve essere presente);
3) *x* e *y* nell'ultimo esempio sono segnaposto: potete mettere qualsiasi numero al loro posto.

5.4.3 Raggruppatori

I raggruppatori mettono insieme gli altri segni.

segno	traduzione
[]	gruppo di alternative
()	unità

Tabella 5.3: I raggruppatori

Le parentesi quadre sono un potente raggruppatore: tutti i caratteri al loro interno sono percepiti dall'interprete perl come alternative possibili. Per esempio, con /p[aieou]zza/ trovereste *pazza, pizza, pezza, pozza, puzza*. Attenzione: ogni singolo carattere è alternativo a tutti gli altri e tutto quel che sta nelle parentesi indica comunque un solo carattere; quell'espressione regolare non trova, ad esempio, *piazza*. Se voleste poter trovare anche *piazza* dovreste riformulare l'espressione regolare come: /p[aieou]+zza/. Eppure /p[aieou]+zza/ troverebbe anche *spiazzare*; per essere sicuri che ci sia un matching solo di parole intere e non anche di pezzi di parole l'espressione nella sua forma migliore è: /\bp[aieou]+izza\b/. Riuscite a leggere e capire l'espressione regolare? Dovrebbe essere abbastanza facile: riunisce solo caratteri che abbiamo già visto.

Scrivere tutte le vocali entro un paio di parentesi quadre è tutto sommato pratica accettabile, ma ci sono casi nei quali le espressioni regolari non forniscono scorciatoie (almeno tra quelle già viste) e agire direttamente tra le parentesi quadre potrebbe diventare un po' troppo noioso. Ad esempio, se volessimo effettuare un riconoscimento di tutte le parole che iniziano con lettera maiuscola, il segno `\w` non ci servirebbe, perché non è in grado di distinguere tra maiuscole e minuscole, e scrivere tutte le lettere maiuscole tra le parentesi

sarebbe semplicemente da masochisti informatici (e pessimi programmatori, che è peggio). Si può, in tal caso, usare il trattino per indicare degli intervalli all'interno delle parentesi quadre. Per esempio [A-Z] indica tutte le lettere maiuscole, mentre [a-zA-Z] indica tutte le lettere e [a-zA-Z0-9] è uguale a \w.

Un altro uso interessante delle parentesi quadre dipende dalla possibilità di esprimere una negazione: se il primo carattere che si trova all'interno delle parentesi quadre è un accento circonflesso[12], allora l'espressione regolare troverà tutto quello che NON corrisponde ai caratteri tra parentesi quadre.

Per esempio /\bca[^nr]ta\b/ troverà *casta* ma non *carta* e neppure *canta*.
E se cercate tutti gli allomorfi*I morfemi con lo stesso valore. -zion-, sion-, -gion- sono ad esempio allomorfi.* di *-zione*, il primo tentativo può essere quello di cercare /\w+[^z]ione/, che troverà tutte le parole terminanti in *-sione, -gione* (come *fissione, libagione*, ma anche come *mattacchione* che non c'entra nulla).

Mettete insieme la negazione e gli intervalli all'interno delle parentesi quadre, ed ecco questa curiosa tabella di equivalenze:

segno	espressione
\w	[a-zA-Z0-9]
\W	[^a-zA-Z0-9]
\d	[0-9]
\D	[^0-9]
\s	[\n \t \f \r]
\S	[^\n \t \f \r]

Tabella 5.4: Carrateri e parentesi quadre

Le parentesi tonde hanno diverse funzioni, la più immediata ed evidente delle quali è quella di raggruppare frammenti di espressione regolare per poterli trattare con dei quantificatori.

L'espressione regolare /avvocato?(essa)?/ è un po' rozza, ma lampante nel significato: un suo utilizzo con la funzione `m//`[13] restituirà esito positivo con *avvocato* oppure con *avvocatessa* escludendo nel contempo, ad esempio, *avvocatura*.

Nel proseguio vedremo un altro possibile uso delle parentesi, ma per ora può essere sufficiente quanto detto.

5.4.4 Connettivi

Dei tre connettivi che vedrete qui elencati, solo il terzo è davvero nuovo; abbiamo infatti già visto sia le parentesi quadre (qui menzionate per il loro valore di [oppure]) che l'accento

[12] A scanso d'equivoci: l'accento circonflesso è quello a forma di tetto, la tilde è quella a forma di onda.
[13] Cfr. pag. 76.

circonflesso (menzionato per il valore di negazione).

Il terzo si chiama *pipe*, sta sotto il tasto [esc], lo vedete? Significa [oppure] ma viene usato per le alternative tra interi frammenti di espressione regolare – e non per singoli caratteri come accade per le parentesi quadre. Ad esempio, l'espressione regolare sugli avvocati che abbiamo visto prima può essere riformulata come: /\bavvocat(o|essa)\b/ in cui i termini dell'alternativa sono la *o* a sinistra del *pipe* e il blocco *essa* alla sua destra.

segno	traduzione	limite
[]	oppure	caratteri
^	non	caratteri
|	oppure	non caratteri

Tabella 5.5: I connettivi

5.4.5 Ancore

Le ancore servono per definire, all'interno del testo analizzato, il punto in cui deve trovarsi l'espressione regolare. Le posizioni definibili sono l'inizio e la fine del testo in esame, ma attenti: il testo analizzato non è, nella maggior parte dei casi, il testo nella sua interezza, il testo come lo analizzerebbe un utente umano. Pensate ai listati visti finora che aprono un file e lo leggono con `while(<>)` o `while(<FILEHANDLE>)`: i testi sono letti linea per linea, quindi l'inizio e la fine sono l'inizio e la fine della linea, non del testo[14].

segno	traduzione	esempi
^	inizio	/^gatto/
$	fine	/gatto$/

Tabella 5.6: Le ancore

Le ancore non solo indicano la posizione, ma pure dipendono dalla posizione: guardate gli esempi nella tabella a pag. 85: dove sono ancorati i segni che ci interessano? Inoltre, si tratta di segni che dispongono anche di altri usi, talvolta piuttosto interessanti – come l'accento circonflesso.

5.4.6 ...e altri

Già, direte voi, ma se io voglio cercare delle parentesi oppure l'accento circonflesso o il punto, come faccio?

[14] E, in realtà, poiché il calcolatore riconosce la fine di una linea solo quando incontra il carattere di nuova linea, non è detto che quello che per noi umani è una linea lo sia anche per il calcolatore.

Esiste il carattere di *escape*, la barra.
Avete già visto alcuni casi di barre seguite da caratteri normali, come `\n`, che assumono per questo un valore speciale.
La barra modifica il funzionamento del carattere che precede: davanti alla lettera *w* – una comunissima u doppia – forza un'interpretazione speciale: [un qualsiasi carattere alfanumerico]. Davanti al punto – il carattere speciale delle espressioni regolari che sta per [qualsiasi carattere] – forza una interpretazione letterale (ovvero normale, ai nostri occhi: il punto è interpretato come punto).

segno	significato
\	carattere di escape
$[numero]	riferimento

Tabella 5.7: Altri segni speciali

Il segno del dollaro seguito da un numero, invece, serve a fare un riferimento ad un frammento di espressione regolare. (Eeh?)

Immaginiamo che vogliate oscurare degli indirizzi di e-mail per salvaguardare la *privacy* di quanti sono esplicitamente citati in un epistolario elettronico. Vi interessa che gli indirizzi di e-mail siano irriconoscibili, ma non volete perdere i dominî[15]. Ad esempio, volete che questi indirizzi:

giacomo.leopardi@recanati.it
umberto.saba@trieste.it
dante.alighieri@firenze.it
francesco.petrarca@arezzo.it
rocco.siffredi@ortonamare.it

risultino essere rispettivamente:

INDIRIZZO@recanati.it
INDIRIZZO@trieste.it
INDIRIZZO@firenze.it
INDIRIZZO@arezzo.it
INDIRIZZO@ortonamare.it

Come fare? Come fareste con Word?[16]

[15] La parte di indirizzo che sta al di là della chiocciolina. Potrebbe trattarsi di un'informazione utile.
[16] Immagino che cerchereste tutte le chioccioline e fareste la sostituzione a mano. Ecco a cosa serve il perl.

Con le espressioni regolari si potrebbe scrivere:

```
s/[^ ]+(@[^ ]+)/INDIRIZZO$1/g;
```

Priviamo a tradurre, prima pezzo per pezzo e poi tutto insieme.

Abbiamo visto cosa significa l'accento circonflesso, in quella posizione dentro le parentesi quadre. Insieme all'accento, tra le parentesi, c'è uno spazio, quindi: `[^ ]` significa [qualsiasi cosa eccetto uno spazio].

Il segno del più significa [almeno uno]: almeno un carattere diverso dallo spazio.

Poi si trova la chiocciolina; di nuovo la sequenza di caratteri che significa [qualsiasi cosa eccetto uno spazio] e di nuovo il più. La chiocciola, la sequenza di caratteri e il secondo più sono racchiusi tra parentesi. Tonde.

Il significato complessivo del testo da sostituire potrebbe pertanto essere parafrasato con: almeno un carattere qualsiasi, seguito da una chiocciola, seguito da almeno un altro carattere. Questo è quello che deve trovare il nostro codice. In fondo, un indirizzo di e-mail è esattamente una chiocciola con almeno un carattere prima ed uno dopo.

Vediamo adesso con cosa deve sostituirlo. Nello spazio del testo con cui sostituire possiamo leggere la parola *INDIRIZZO* e una variabile, la variabile `$1`. La variabile `$1` è uguale al contenuto della prima parentesi tonda che si trova nell'espressione regolare da cercare; cioè è uguale alla chiocciolina più tutto quello che segue fino al primo spazio escluso.
Chiaro, no?

5.5 Un altro esempio con le parentesi tonte

5.5.1 Risultato desiderato

ATTENZIONE: il programma di sostituzione che abbiamo appena visto non funziona con i riferimenti alle parentesi tonde! Se, cioè, chiedete al programma di sostituire in un documento tutte le occorrenze dei titoli *dottore/i* e *dottoressa/e* con le loro forme abbreviate, userete come pattern di ricerca: /dottor[ei]?(ss[ae])?/ e per la sostituzione: dott.$1. In questo modo:

```
$stringaDaSostituire =~ s/dottor[ei]?(ss[ae])?/dott.$1/gi;
```

Questa espressione regolare, però, sostituisce ad ogni occorrenza che incontra *dott.$1* e non, ove opportuno, *dott.*, *dott.ssa* o *dott.sse*. Ciò dipende dal fatto che alcuni caratteri "non passano" attraverso lo standard input, vengono cioè valutati letteralmente e non interpretati: è esattamente quello che accade al nostro segno del dollaro quando lo digitiamo nello STDIN.

Dunque il listato deve essere un poco più ricco e complicato[17]. Eventualmente, date un'altra occhiata all'ultima versione del listato di sostituzione e alla sua spiegazione[18], visto che nella descrizione che seguirà non saranno ripetuti tutti i concetti.

5.5.2 Codice

```
my($scelta,$s1,$rimpiazzo);
$^I ='_vecchio';
print "cosa devo cercare: ";
my $pattern = <STDIN>;
chomp($pattern);
print "con cosa lo correggo: ";
my $nuovo = <STDIN>;
chomp($nuovo);
print "\n!!!!!!!! Per confermare una sostituzione,
premi \"s\" e il tasto di invio; altrimenti
premi solo il tasto di invio\n";

while(<>)
    {
    $scelta = "";
    $rimpiazzo = $nuovo;
    $s1 = "";
    if($_=~m/$pattern/gi)
        {
        print STDOUT "trovato!:\n\t\ \$1 = $1
                      \n\t pattern: $pattern\n";
        $s1 = $1 if $1;
        if($s1) {$rimpiazzo =~ s/\$1/$s1/g;}
        else {$rimpiazzo =~ s/\$1//g;}
        print STDOUT "\t sostitituz. = $rimpiazzo ($s1)\n";
        print STDOUT "sostituisco questo? => ". $_;
        $scelta = <STDIN>;
        chomp($scelta);
        }
     if($scelta eq "s")
        {$_=~ s/$pattern/$rimpiazzo/gix;}
```

[17] Il listato d'esempio è fatto per lavorare con una sola variabile/parentesi; è cioè in grado di interpretare correttamente solo la variabile `$1` e non delle eventuali variabili `$2, $3` eccetera. Ma sono certo che saprete opportunamente modificare il listato in maniera opportuna, una volta compreso il suo funzionamento.

[18] A pag. 79.

```
print;
}
```

5.5.3 Descrizione del codice

Il codice

Il listato è diviso in due blocchi, potete vederli divisi da una linea bianca.

Il primo blocco si apre con la funzione `my()` che serve, in questo caso, a dichiarare preventivamente quali saranno alcune delle variabili impiegate all'interno del ciclo `while`[19]. A parte quella prima linea, che presenta una nuova ma non difficile funzione, il resto del blocco non presenta novità strabilianti: avete già visto tutto almeno una volta.

Il secondo blocco, interno alla struttura di controllo `while`, è più interessante.

Come vedete, oltre alla variabile `$scelta` che avevamo già incontrata nella versione precedente del listato, adesso vengono attribuiti dei valori alle variabili `$s1` (valore nullo) e `$rimpiazzo` (uguale alla variabile `$nuovo`). I motivi dell'attribuzione di questi valori saranno più chiari nel seguito.

Dopo l'attribuzione di questi valori, potete leggere la protasi di un periodo ipotetico che già avete visto e conoscete, e finalmente iniziano le danze.
Innanzitutto è presente uno strano periodo ipotetico di una sola linea, `$s1 = $1 if $1;` che significa: *attribuisci alla variabile $s1 il valore della variabile $1 se esiste una variabile $1.*

Ma la variabile `$1` esiste? E, se esiste, a cosa è uguale?
Iniziamo dall'ultima domanda: (immaginate che esista) è uguale a quanto, nell'espressione regolare di ricerca, sta tra la prima coppia di parentesi tonde. Il nostro pattern di ricerca sarà: `/dottor[ei]?(ss[ae])?/`. Dunque, se c'è, la variabile `$1` è uguale a *ssa* oppure a *sse*[20].
E... c'è? È presente? Questo dipende dal testo: se nel testo compare la stringa *dottoressa*, per esempio, allora avremo un `$1`, altrimenti no[21].

È utile porre altre due domande su questo periodo ipotetico: è davvero necessario? E: che senso ha creare una seconda variabile quando potremmo usare direttamente `$1`?

Il periodo ipotetico è necessario formularlo così perché non siamo certi che ad ogni match corrisponderà un match che implica anche un valore per la variabile `$1`.
Ad esempio, incontrando la parola *dottor* il programma effettuerà un riconoscimento (visto che quella parola viene descritta dall'espressione regolare) senza attribuire alcun nuovo valore alla variabile `$1`.

[19] Potete immaginarle come un promemoria.
[20] Entrambi i trigrammi corrispondono ale' espressione regolare /ss[ae]/.
[21] Più correttamente potremmo dire che avremo un valore di `$1`, altrimenti no, ma in pratica cambia poco.

Quanto al senso di creare una seconda variabile invece di usare direttamente `$1`, va considerato che `$1` è una variabile *sui generis*: è una variabile globale che entra in gioco solo quando ci sono delle espressioni regolari; il suo valore come abbiamo visto non è garantito né si può arbitrariamente impostare come uguale a zero prima che sia "chiamata" da una espressione regolare. È perciò in generale una buona strategia quella di metterla da parte il prima possibile, per esempio ricorrendo ad un'altra variabile con lo stesso valore (iniziale, visto che poi potremmo decidere di modificarlo come accade in questo caso).

Immaginiamo che finalmente il programma avverta di aver trovato un'occorrenza del pattern e la mostri insieme al pattern di ricerca.
Se il match non implica una variabile `$1` (ad esempio viene trovata la parola *dottor*), noterete che il programma restituirà un errore, avvertendo che all'interno del ciclo di `while` è stata impiegata una variabile non inizializzata. Si sta solo lamentando del fatto che `$1` non ha un valore, ma lavorerà lo stesso.

Un altro periodo ipotetico ma con una sintassi regolare: *se esiste $s1 sostituisci nel valore della variabile $rimpiazzo i caratteri* `$1` *con il valore di $s1, altrimenti cancella i due caratteri* `$1` *dalla variabile $rimpiazzo.*
Il motivo di questa sostituzione dovrebbe essere abbastanza intuitivo: se abbiamo un valore di `$1` (e quindi di `$s1`), allora questo valore dovrà prendere il posto dei caratteri *$1* nella variabile `$rimpiazzo` (che equivale al pattern di sostituzione introdotto dall'utente), altrimenti è necessario cancellare i caratteri *$1* perché non vengano stampati come accadeva nella vecchia versione del listato.

A questo punto entrano in gioco due linee molto utili nella correzione degli errori di codice: si tratta di due `print()` nello standard output, che diventa così a tutti gli effetti l'interfaccia del programma di sostituzione.

Il primo verifica che la sostituzione sia corretta; se per esempio il programma trova il nome *dottoressa Brückenmeier*, su questa linea comparirà l'avviso:
`$> sostituzione = dott.ssa (ssa)` [invio]
che ci permette di capire che a quel punto del programma, a quella iterazione, il valore della variabile `$s1` è uguale a *ssa* e che la sostituzione finale sarà: *dott.ssa.*

Il resto del programma è sostanzialmente uguale alla vecchia versione.

Il programma

Ora proviamo ad immaginare cosa accadrebbe se effettivamente usassimo il programma per sostituire /dottor[ei]?(ss[ae])?/ con /dott.$1/.

Escludiamo per semplicità tutte le linee del testo da correggere in cui i titoli non compaiono o sono già abbreviati (non c'è matching, riscontro: il programma riscrive semplicemente la linea) e quelli in cui ci sono almeno due occorrenze in una linea (in tal caso, se i titoli non hanno la stessa forma, se cioè non c'è accordo di genere e caso, con lo script che abbiamo è meglio non sostituire nulla e risolvere specifici problemi successivamente a mano).

Immaginiamo, per maggior semplicità, che il nostro programma incontri quattro linee di codice interessanti, una consecutiva all'altra:
dottor Sivago
dottoressa Brückenmeier
dottori Jackill e Hide[22]
dottoresse Materassi
I quattro blocchi che seguono simulano la trasformazione delle variabili e del testo ad ogni iterazione[23].

iterazione del dottor Sivago $scelta = “”; $rimpiazzo = “dott.$1”; avviene il riconoscimento ma con $1 uguale a nulla, quindi $rimpiazzo diventa uguale a “dott.” (visto che i caratteri *$1* vengono cancellati); $scelta = “s” per la risposta dell'utente quindi *dottor* viene sostituito con *dott.* Ricomincia;

iterazione della dottoressa Brückenmeier $scelta di nuovo uguale a niente; $rimpiazzo di nuovo uguale a “dott.$1”; riconoscimento avvenuto con $1 uguale a *ssa*, quindi $rimpiazzo diventa uguale a “dott.ssa”; $scelta = “s” per la risposta dell'utente quindi *dottoressa* viene sostituito con *dott.ssa*. Ricomincia;

iterazione dei dottori Jackill e Hide $scelta di nuovo uguale a niente; $rimpiazzo di nuovo uguale a “dott.$1”; match avvenuto ma con $1 uguale a nulla, quindi $rimpiazzo diventa uguale a “dott.” (visto che i caratteri *$1* vengono cancellati); $scelta = “s” per la risposta dell'utente quindi *dottori* viene sostituito con *dott.* Ricomincia;

iterazione delle dottoresse Materassi $scelta di nuovo uguale a niente; $rimpiazzo di nuovo uguale a “dott.$1”; match con $1 uguale a *sse*, quindi $rimpiazzo diventa uguale a *dott.sse*; $scelta = “s” per la risposta dell'utente quindi *dottoresse* viene sostituito con *dott.sse*. Ricomincia.

A qualcuno potrebbe non essere chiara la faccenda dei riconoscimenti di /dottor[ei]?(ss[ae])?/; quindi, visto che siamo nel capitolo sulle espressioni regolari, vale la pena di riprendere l'argomento.

Innanzitutto: come fa a riconoscere *dottor*? Osservate l'espressione regolare: possiamo dividerla in tre parti: la stringa *dottor*; il carattere *[ei]*[24] che può esserci o meno, come dichiarato dal punto interrogativo che lo segue; e il trigramma *ss[ae]* tra parentesi tonde e anch'esso seguito da un punto interrogativo. Nel caso in cui né il secondo elemento dell'espressione regolare (il carattere singolo), né il terzo elemento (il trigramma) compaiano, il programma potrà comunque riconoscere la sola sequenza /dottor/.

[22] Essendo la stessa persona, dovrebbero essere laureati entrambi, se lo è uno dei due.

[23] In questa sequenza di iterazioni i nomi di variabile non saranno scritti con testi a spaziatura fissa.

[24] Dite che sono DUE lettere? Oh, no, sono due possibili forme di UN solo carattere: ricordatevi che le parentesi quadre, per quanta roba contengono, rappresentano UN solo carattere, a meno che dei quantificatori non dicano altrimenti.

Alla luce di questa spiegazione, provate a rispondere alle domande: 1) come fa a riconoscere *dottoresse*?; 2) come fa a riconoscere *dottori*? Vedrete che non vi sarà difficile iniziare a proporre anche altre espressioni regolari non meno interessanti ed utili.

Capitolo 6

Labora et ora

6.1 Introduzione

A questo punto dovreste essere diventati dei piccoli Larry Wall[1] in calzamaglia e mantello, che hanno capito le logiche e sono interessati ad applicarle; in fondo la programmazione è proprio questo: logica applicata.

Nel resto del capitolo troverete alcuni listati che "fanno cose con le parole" (del perl) e sulle parole (della lingua storico naturale che decidete voi). Ognuno di questi listati avrà un bagaglio minimo di spiegazioni, soprattutto per il funzionamento dei meccanismi meno intuitivi e le nuove funzioni. Se dovessero sorgere problemi che il testo scritto non risolve, la pratica colmerà certo le lacune del presente manuale.

Ogni sezione sarà così strutturata: un paragrafo introduttivo che dichiara il risultato atteso da un programma; un paragrafo con il codice (ampiamente commentato[2], da copiare e far girare sulla propria macchina); un paragrafo con dei commenti più lunghi e articolati; un paragrafo con dei suggerimenti per mettersi alla prova e modificare i listati.

6.2 Lista di frequenza

6.2.1 Risultato desiderato

Una lista di frequenza è una lista di tutte le parole contenute in un testo e chespecifica quante volte ogni parola è stata usata.

Le prime volte che si usa una lista di frequenza è abbastanza divertente vedere quale parola è stata usata di più e quale di meno. Giusto per togliervi il divertimento: le parole grammaticali sono usate sempre tantissimo, il primo verbo di solito compare intorno al settimo posto in ordine di frequenza, il primo sostantivo tra il decimo e il quindicesimo se

[1] Il creatore del perl.

[2] Cfr. pag. 22 per un promemoria sui commenti.

il testo analizzato è piccolo, anche oltre il quindicesimo o il ventesimo se è grande o davvero grande.

Dopo i primi esperimenti per curiosità, si tende a ricorrere alle liste di frequenza solo quando ce n'è un effettivo bisogno, per esempio per provare ad indovinare di che cosa parla un testo senza leggerlo o per dare un'occhiata al lessico di un autore – e che ci crediate o no c'è chi ha usato delle liste di frequenza per attribuire scientificamente la paternità di un'opera ad un determinato autore.

Il risultato che vorremmo per il listato di questo paragrafo sarà dunque un elenco di tutte le parole di un testo di nostra scelta che, a fianco alle parole estratte, mostri anche il numero di occorrenze di ogni parola. Inoltre, ci piacerebbe poter ordinare la lista, a piacimento, sia per numero di occorrenze che alfabeticamente.

6.2.2 Codice

```
# seleziona la punteggiatura
my $punteggiatura = '(\.|\,|\;|\:|\-|\?|\!|\"|\=|\)
|\(|\||\_|\<|\>|\'\w|\~+)';
# seleziona il nome del file
print "scrivi il nome del file:\n";
$nome_del_file = <STDIN>;
chomp($nome_del_file);

# apre il file
open(INPUT, $nome_del_file);
while (<INPUT>)
    {
    # trasforma maiuscole in minuscole
    tr/A-Z/a-z/;
    # aggiunge uno spazio dopo le lettere apostrofate
    s/(\w\')/$1 /g;
    # normalizza tutti gli indirizzi e-mail
    s/(\S|\.)+@(\S|\.)+/EMAIL/g;
    # rimuove la punteggiatura
    s/$punteggiatura//g;
    # rimuove gli spazi multipli
    s/ +/ /g;
    # trasforma/divide il file-stringa in un array
    @words = split;
    # conta le occorrenze di ogni parola
    foreach $word (@words)
        {
```

```
        $wordcount{$word}++;
        $numero++;
        }
    }

# ordina alfabeticamente
foreach $word (sort keys(%wordcount))
# ordina per occorrenze
#foreach $word
# (sort {$wordcount{$b}<=>$wordcount{$a}} keys(%wordcount))

# stampa
    {
    printf "%s %d\n", $word, $wordcount{$word};
    }
# stampa il numero di elementi
print "Nel documento ci sono $numero parole \n";
# chiude il file
close(INPUT);
```

6.2.3 Commenti

Il codice è diviso in quattro parti: acquisizione di informazioni; apertura e lettura del file; conteggio parole; ordine delle parole; output e chiusura file.

acquisizione informazioni nella prima linea definiamo, con una espressione regolare, la punteggiatura. Ci servirà perché i segni di interpunzione non vengono contati, di solito. Il resto è chiaro.

apertura file i singoli passaggi sono già commentati nel codice. Vanno segnalate però la funzione `tr///` e l'uso di `split()` senza argomenti. Il resto sono delle sostituzioni non troppo fantasiose.

La funzione `tr///`[3] è una funzione delle espressioni regolari (per questo non ha parentesi ma barre) affine ad `s///`. La maggiore differenza tra `tr///` ed `s///` sta nel fatto che la prima lavora carattere per carattere, mentre la seconda no, o non necessariamente. Il fatto di operare carattere per carattere permette – quando è esattamente sui caratteri che si deve lavorare – delle espressioni regolari particolarmente stringate. Per esempio in questo caso non sono necessarie le parentesi quadre (ovviamente tutto quello che occupa lo spazio del pattern di ricerca descrive un solo carattere), e nel pattern di sostituzione basta specificare [lettere minuscole]: sarà poi l'interprete

[3] *tr* sta per *translate*, [traduci].

a chiarire che le sostituzioni riguardano precisamente la lettera riconosciuta e non un'altra.

La funzione `split()` senza argomenti richiede due precisazioni: quando l'interprete non trova argomenti, gli argomenti predefiniti che colloca a saturare le proprie valenze sono: `$_`, il testo da segmentare (in questo caso la linea di testo in lettura in ogni momento) e [spazio], lo strumento con il quale segmentare (per questo è importante prima rimuovere tutta la punteggiatura ed eliminare gli spazi multipli).

conteggio parole il cuore del listato è probabilmente tutto qui: con lo *split* appena visto il programma genera un array per ogni linea di testo, e questo array è costituito da tutte le parole della linea stessa. A questo punto cosa deve fare? La soluzione ha qualcosa di geniale, davvero, giuro che non lo dico perché sta scritto in questo script[4]. Pensateci: abbiamo bisogno di un modo per raccogliere in maniera un minimo sistematica delle coppie parola-occorrenze.
Se non ci fosse bisogno della sistematicità, ci potremmo accontentare di coppie nome-valore di una variabile per ogni parola, ma ne abbiamo bisogno. D'altronde, se usassimo solo delle variabili non strutturate, dovremmo risolvere il problema di richiamare e stampare ogni variabile.
Ecco la soluzione: si crea un hash, si attribuisce come chiave ad ogni elemento dell'hash una delle parole che compaiono, si attribuisce come valore di ogni elemento il numero delle occorrenze. Per aggiornare la nostra lista di frequenza, ad ogni iterazione del codice si richiama la variabile `$nomehash{$parola}` e la si incrementa con un operatore di autoincremento `++`. Se la parola è nuova, il suo valore viene impostato ad 1 – zero incrementato di uno –, se invece esiste già un elemento con quella chiave (il listato ha già incontrato la parola), il valore viene incrementato di uno.
Magia!

ordinamento la fase di ordinamento è probabilmente la più complessa. Valutate innanzitutto due cose: il vero codice è solo di due linee, di queste due linee una deve sempre essere commentata (quindi ne funziona una per volta). Riprendiamo le due linee e cerchiamo di dare una spiegazione del loro funzionamento:

```
foreach $word (sort keys(%wordcount))
foreach $word (sort {$wordcount{$b}<=>$wordcount{$a}}
                                    keys(%wordcount))
```

La prima, in realtà è abbastanza facile: iniziamo a leggerla da sinistra: vedete subito dopo le parentesi tonde chiuse il nome di un hash, rielaborato dalla solita funzione `keys()`, a propria volta "lavorato" dalla funzione `sort()`.

[4]Che non ho neanche inventato io, è... una soluzione standard. Si fa così, si sa.

La funzione `sort()` mette in ordine degli array[5], in questo caso l'array delle chiavi dell'hash %wordcount. Le chiavi sono ovviamente in ordine alfabetico; una volta che sono state messe in ordine alfabetico, possono essere processate al solito modo di `foreach`, una per volta. Chiaro, no?

La seconda linea[6] è forse più impegnativa: vedete che tra la funzione `sort()` e il suo argomento, l'array, esiste un altro argomento, tra parentesi graffe. È un argomento piuttosto complicato, lo ammetto. Ma se lo osservate con attenzione, vedrete che è costruito in maniera ordinata: ci sono due variabili (sono elementi di un hash, ma in questo momento si tratta di un dettaglio irrilevante) e in mezzo un operatore di confronto che si chiama *compare*[7] e che serve per fare dei confronti tra numeri.

Il nuovo argomento fornisce a `sort()` un modo per ordinare: non più quello predefinito, alfabetico, ma un altro definito dall'utente. In realtà `sort()` più che ordinare alfabeticamente ordina in modo decrescente, solo che per questa funzione la *a* è in qualche modo più della *b*, che è più della *c* e così via. Questo per dire che in effetti *sort()* non ci rimane male, se gli si dice di ordinare non alfabeticamente ma in qualche altro modo.

Torniamo però al nuovo argomento. Il nuovo argomento si basa su un confronto tra numeri: li vedete tra le graffe più esterne, si tratta dei valori dell'hash, cioè il numero di occorrenze di ogni parola. `sort()` effettua questo confronto e mette in ordine decrescente per occorrenza le parole che fanno parte del corpus di riferimento.

Già, ma quali elementi dell'hash vengono confrontati? Cosa significano `$a` e `$b`?
Vengono confrontati TUTTI i valori, `$a` e `$b` sono elementi della sintassi particolare dell'operatore di confronto <=> che identificano ogni elemento della lista.
Non c'è altro da capire.

Non si poteva fare altrimenti? Certo, ad esempio si sarebbe potuto creare esplicitamente un array di tutti i valori di %wordcount, e, una volta ordinato con *sort* tale array, si sarebbero stampati tutti gli elementi dell'hash %wordcount il cui valore corrispondeva ad ognuno dei valori dell'array creato prima. Quindi per ogni elemento dell'hash si sarebbe verificato se corrispondeva al valore corrente dell'array ordinato e lo si sarebbe stampato.

Ma mi pare assai più macchinoso[8].

output la stampa della lista poteva essere più semplice, ma ho voluto introdurre una nuova funzione, `printf()`, perché so che avete intelligenze esuberanti e temevo che

[5] Che va letto: questa funzione accetta un argomento, un array, che ordina, e tendenzialmente il risultato del suo lavoro è lo stesso array, ma ordinato.

[6] Nel listato è spezzata, ma sui vostri calcolatori non avrete problemi a scriverla su una sola linea.

[7] In inglese!

[8] ESERCIZIO: se non ci credete, provate a implementare questa soluzione.

poteste annoiarvi. E poi potrebbe esservi utile averla già incontrata.
Questa nuova funzione costituirà tuttavia un balocco di effimero interesse, visto che si tratta di una funzione dai meccanismi quasi banali.

printf significa [stampa formattando] e semplicemente stampa molte variabili in un ordine. Si usa per descrivere all'interprete perl cosa deve stampare e come, senza dover mettere troppe variabili nel complemento oggetto del verbo *print*.

Guardate come abbiamo fatto noi:

```
printf "%s %d\n", $word, $wordcount{$word};
```

Tradotto: *stampa "stringa numero a capo", la stringa è uguale a $stringa, il numero a $wordcount{$word}*.
Con due variabili sembra avere poco senso, ma quando già iniziano a diventare cinque o sei vedrete che è molto più comodo mettere tutte le variabili in fondo (così è subito evidente cosa viene stampato da ogni istruzione `printf()`) e spiegare in sede separata al programma come le cose devono essere impaginate: rispettando l'ordine nel quale sono presentate le variabili in coda e ricordando che `%d` indica un numero e `%s` indica una stringa.

6.2.4 Orizzonti (di gloria)

Allora, cose che potete fare come esercizio per mettervi alla prova[9] ce ne sono ancora molte; ve ne illustro quattro, giusto per sgranchirvi le meningi.

Mettiamo i puntini, punto

Come dovrebbe essere modificato il listato per contare, separatamente, anche la punteggiatura? Immaginate, cioè, di desiderare che nella vostra lista di frequenza compaiano anche i segni di interpunzione, come se fossero parole, per capire ad esempio se sono usate quanto le parole grammaticali o le parole non grammaticali.

Provate ad agire come dei giovani scienziati del linguaggio: formulate un'ipotesi riguardo alla distribuzione della punteggiatura riguardo alle altre parti del discorso nel vostro corpus di riferimento – spiegando anche perché la distribuzione dovrebbe essere quella –, poi estraete la lista di frequenza e verificate se l'ipotesi di partenza è corretta oppure no, infine escogitate qualche metodo per verificare la validità delle vostre spiegazioni.

Messi i puntini, buttare via il resto

Diciamo che i segni di interpunzione sono l'unica cosa che vi interessa: come modificare il listato in modo da avere una lista che riguardi esclusivamente la punteggiatura del vostro corpus di riferimento?

[9] Altro che sudoku, questo sì che vi farà ottenere l'ammirazione di tutti gli altri avventori del bar!

È più impegnativo, soprattutto se desiderate tenere insieme tutti gli agglomerati multisegnici, come *?!* o *!!!*, e tutte le icone emotive, come `:)` `;-)` `:D` oppure `^___^`. Tuttavia non è un compito impossibile: dovete solo pensare che il tipo e l'ordine delle operazioni che il vostro programma deve svolgere potrebbero essere diverse.
Be', almeno un po'.

Contare le forme uniche

Essendovi vantati a destra e a manca di essere divenuti eccezionali programmatori di perl, un vostro amico vi chiede un estrattore di liste di frequenza, perché nel tempo libero ha registrato e sbobinato tutti i discorsi del suo personaggio politico preferito ed è curioso di sapere che tipo di lessico usa (eventualmente per acquisirlo ed iniziare a parlare come il personaggio politico, ahivoi).
Di fronte ad una simile richiesta ostentate un sorriso superiore e memorizzate su una penna USB il vostro listato. State per consegnargliela quando un lampo di genio illumina la vostra mente: sarebbe veramente *chic* affiancare ad ogni parola anche il numero d'ordine, per permettergli di considerare che la parola *sociale*, ad esempio, è la novantaduesima più usata.

Ecco quel che potete fare: create un contatore che fornisca il numero d'ordine di ogni parola (iniziando ovviamente dal numero 1) e fate in modo di stamparlo.

Ridirezionare l'output

A dirla veramente tutta, il vostro amico non è così entusiasta di dover lavorare con lo STDOUT nella shell (o, peggio, nel DOS), e amerebbe invece avere un singolo file con la sua bella lista di frequenza. Sapete fare anche questo?[10]

6.3 Legge di Zipf

6.3.1 Risultato desiderato

Nell'esercizio "contare le forme uniche" abbiamo immaginato di associare ad ogni forma della lista di frequenza un numero d'ordine: 1 alla prima forma, 2 alla seconda e così via. In effetti, quel numero c'è quasi sempre nelle liste di frequenza ed è così utile che ha anche un nome: si chiama **rango**.

Una lista di frequenza è infatti costituita da tre serie di dati correlati: il rango, cioè il numero d'ordine di una forma, la forma stessa e la sua frequenza, cioè il numero di volte che questa parola occorre.

[10] La provocatoria richiesta del vostro amico è irresistibile. Un po' come le allusioni alla mancanza di coraggio di Marty McFly in "Ritorno al futuro" 1, 2 e 3. Ma io so che ne siete capaci... So addirittura che potreste fare in modo che via STDIN il nome del file sia deciso dal vostro amico.

Per definizione il rango e la frequenza di una parola sono inversamente proporzionali: tanto più grande è il primo, tanto minore è la seconda e viceversa[11].

Ma rango e frequenza sono anche correlati da una considerazione di più ampio respiro: la legge di Zipf. La legge di Zipf[12] descrive la frequenza delle occorrenze di un qualche evento (P_i) in funzione del rango (i), essendo quest'ultimo definito come, appunto, il numero d'ordine decrescente sulla frequenza stessa. In altre parole[13], e in riferimento al linguaggio e alle liste di frequenza, secondo questa legge il prodotto di rango e frequenza è grosso modo sempre uguale ad una costante reale positiva.

Facciamo un esempio. Prendiamo un corpus di 32228 parole, di cui 5937 forme uniche (ranghi), come questo manuale fino alla presente frase esclusa. Se voi moltiplicaste il rango[14] di ogni parola per la sua frequenza, dovreste ottenere più o meno sempre lo stesso numero.
A partire dalla parola con rango 10 (*in*, 409 occorrenze) inizio ad avere un valore uguale, grosso modo, a 4000. Questo valore rimane compreso tra 3500 e 4200 fino al rango 1399. Il che significa che tutte le oscillazioni di questo valore rappresentano comunque circa 25 mila parole su 32 mila e un insieme di ranghi sulla lista compreso tra un cinquantesimo e un terzo di tutti i ranghi della lista. L'oscillazione è ampia e caotica, ma tale caoticità implica anche una sistematicità[15], e questo è... interessante.

Funziona, con l'eccezione dei primissimi ranghi e degli ultimi.

Perché è bene conoscere la legge di Zipf?
Perché è stata verificata con le letterature di moltissime lingue (la maggior parte di quelle che hanno una tradizione scritta), anche non umane (be', con trascrizioni del linguaggio dei delfini), ed ha sempre funzionato. Ma non funziona con testi casuali; come se in qualche modo potesse dimostrare, appunto, che i testi non sono casuali.
Se un giorno dovessimo rinvenire, nel sottosuolo o nello spazio, qualcosa che potrebbe essere un testo (un monolito nero ricoperto di incisioni, forse non naturali), la legge di Zipf potrebbe rappresentare il primo strumento che abbiamo per capire se si tratta di un testo oppure no.
Subito dopo dovremmo ingaggiare Martin Mystère.

Perché è bene che noi conosciamo la legge di Zipf?
Ma perchè ci permette di fare un entusiasmante gioco di programmazione per verificare la legge di Zipf su qualsiasi corpus!

6.3.2 Codice

```
my(%parole,@forme,$forma,$tot,$dist,$zipf);
```

[11] Infatti, la parola con il rango più basso – uno – è quella con la frequenza maggiore.
[12] George Kingsley Zipf (1902-1950), filologo e matematico.
[13] Un primo approfondimento di buon livello ve lo può fornire la WikiPedia: `http://it.wikipedia.org/`.
[14] Bloccato, cioè uguale nel caso di uguale frequenza. Lo vedremo più avanti.
[15] L'idea matematica del caos: una sequenza di numeri sempre diversi ma sempre compresi tra un minimo e un massimo. Uhm, be', più o meno.

```
my $punteggiatura = '[.,;:\-?!"=\)\(\|_<>~+]';
my $c = 1;

while (<>)
    {
    tr/A-Z/a-z/;
    s/(\w\')/$1 /g;
    s/$punteggiatura//g;
    @forme = split;
    foreach $forma (@forme)
        {
        $parole{$forma}++;
        $numero++;
        }
    }
# ordina per occorrenze
foreach $forma
       (sort {$parole{$b} <=> $parole{$a}} keys(%parole))
    {
    $zipf = $parole{$forma}*$c;
    print "$c) $zipf \t\t $forma \t\t $parole{$forma}\n";
    $c++;
    }

$tot = keys(%parole);
print "Nel documento ci sono $numero parole
($tot forme non ripetute)\n";
```

6.3.3 Commenti

Avete notato che ancora una volta la struttura `foreach` è scritta su due linee per motivi di spazio? Bene, abituatevici.

Questo listato non è altro che una versione modificata del precedente; quindi vi segnalo solo tre differenze:

- la variabile `$punteggiatura` è meno ingombrante, ma uguale, per effetti, alla omonima variabile nel listato precedente: basti considerare il numero di segni preceduti da barra retroversa;
- non mettiamo in ordine alfabetico, perché non ci serve;
- nell'output, come secondo valore, viene scritto il valore della variabile `$zipf`, che come potete vedere nella linea precedente è il risultato della moltiplicazione (il segno

asterisco) del rango (la variabile `$c`) per il numero di occorrenze (il valore di hash corrispondente alla chiave `$forma`).

6.3.4 Orizzonti

Vertical skyline/limit

Sorvolerò sul suggerirvi di ridirigere l'output in un file, anche se trovo che sia sempre un buon esercizio[16].

Può invece essere interessante provare a "disegnare" un rudimentale diagramma che ci aiuti a visualizzare i valori della variabile `$zipf` in maniera meno numerica e più visiva.

Inizierò con il fornirvi qualche suggerimento su come potrebbe essere realizzato un listato che mostra il prodotto di rango e frequenza per ogni parola, auspicando che ad ogni suggerimento voi proviate ad immaginare come potrebbe essere realizzato il programma. Soltanto alla fine vi mostrerò una possibile realizzazione.

1. qualcosa di ovvio: il nuovo listato è una parziale modifica del precedente (al punto che vale la pena di duplicare il file con il primo listato, rinominarlo ed agire su quello);

2. quel che va modificato del vecchio listato è la parte dell'output, visto che solo l'output presenta qualche novità;

3. il grafico correlerà due distinte variabili: ogni parola e ogni valore della variabile `$zipf` relativo a quella parola; dunque il grafico avrà due dimensioni;

4. tuttavia, non essendo provvisti di un foglio di gigantesche dimensioni, dovremmo immaginare che almeno una delle due dimensioni del grafico sia limitata: e il numero di parole e i valori della variabile `$zipf` potrebbero essere maggiori del numero di righe e colonne presenti in un foglio di carta; per comodità limiteremo la dimensione orizzontale (asse delle ascisse), immaginando che se anche la variabile rappresentata in verticale (asse delle ordinate) possa anche continuare in un secondo foglio (sarebbe più complicato stampare in orizzontale);

5. delle due variabili prese in considerazione, quella che rappresenta il numero di parole non può proprio essere compressa in alcun modo, mentre i valori della variabile `$zipf` possono, ad esempio, essere tutti divisi per lo stesso numero (tre, mille, diciassette) mantenendo una certa proporzione e risultando più "piccoli";

6. (questo è l'ultimo suggerimento) per esempio, si potrebbe fare in modo che ogni valore di `$zipf`, opportunamente diviso per cento, corrisponda al numero di spazi prima di

[16] Se lavorate su X, in realtà, vi basta lanciare il programma così:
`$> ./nomeprogramma.pl nomeFileDaAnalizzare > nomeFileDestinazione` [invio]

un asterisco: in tal modo avremmo asterischi più lontani o più vicini per parole che hanno un valore di `$zipf` maggiore o minore.

Siete pronti a vedere la vostra linea di asterischi che disegna gli effetti della legge di Zipf? Ecco come fare:

```
my(%parole,@forme,$forma,$tot,$dist,$zipf);
my $punteggiatura = '[.,;:\-?!"=\)\(\|_<>~+]';
my $c = 1;

while (<>)
    {
    tr/A-Z/a-z/;
    s/(\w\')/$1 /g;
    s/$punteggiatura//g;
    @forme = split;
    foreach $forma (@forme)
        {
        $parole{$forma}++;
        $numero++;
        }
    }
# ordina per occorrenze
foreach $forma
       (sort {$parole{$b}<=>$parole{$a}} keys(%parole))
    {
    $c++;
    $zipf = ($parole{$forma}*$c)/100;
    print ' ' x $zipf;
    print "* ($forma)\n";
    }

$tot = keys(%parole);
print "Nel documento ci sono $numero parole
($tot forme non ripetute)\n";
```

Non era necessario che sapeste scrivere questo programma, ma se siete stati in grado di anticipare l'ultimo suggerimento, o se leggendolo avete avuto l'impressione che qualcuno stesse pronunciando una parola che avevate sulla punta della lingua (insomma, ci siamo capiti), siete già praticamente dei programmatori: potete passare ad un manuale per informatici.

Il segreto di questo programma sta nella penultima linea all'interno del ciclo `foreach`: quella linea stampa uno spazio per ogni valore di `$zipf`.

Il lessico impiegato in quella linea non è intuitivo[17], soprattutto il carattere x come operatore aritmetico della moltiplicazione per le stringhe. Non ne abbiamo parlato prima, ma questa è una delle più importanti lezioni della programmazione: non si sa tutto prima di iniziare, si scopre man mano quel che serve.

Nella linea prima, il numero di spazi è diviso per cento; mentre nella linea successiva vengono scritti l'asterisco e, dopo uno spazio, la parola di riferimento. Mettere lì la parola ci permette di non doverci preoccupare di quante lettere è costituita ogni parola e di conseguenza non ci costringe a modificare ad ogni iterazione il valore di $zipf.

In questo caso è davvero molto utile ridirezionare l'output in un altro file e poi giocare con la barra di scorrimento per leggere con calma il diagramma.

Serriamo i ranghi

Si è detto che le liste di frequenza (e perciò anche i valori di `$zipf` che possono essere calcolati a partire dalle liste) possono avere ranghi bloccati e ranghi non bloccati: nel primo caso il rango cambia quando cambia anche il numero di occorrenze, nel secondo il rango aumenta ad ogni nuova parola.

Noi fino ad ora abbiamo usato liste con ranghi non bloccati[18].

Torniamo indietro: come dovrei modificare il penultimo listato visto per ottenere dei valori di `$zipf` con ranghi bloccati?

Pensateci un po', magari distraetevi: a volte il nostro encefalo lavora anche quando noi ci distraiamo[19] (l'invito a guardare un po' di Simpson è sempre valido, ma per questo tipo di distrazione anche Futurama va bene).

È intuitivo: provate a osservare il codice, se poi proprio non vi illumina ne parliamo.

```
my(%parole,@forme,$forma,$tot,$dist,$zipf,$vval);
my $punteggiatura = '[.,;:\-?!"=\)\(\|_<>~+]';
my $c = 1;

while (<>)
    {
    tr/A-Z/a-z/;
    s/(\w\')/$1 /g;
    s/$punteggiatura//g;
    @forme = split;
```

[17] Non è intuitivo in relazione a quello che abbiamo visto per il perl: in italiano – o nella logica che soggiace alla lingua italiana – è perfettamente comprensibile.

[18] Se questa informazione vi giunge nuova e sorprendente siete invitati a chiudere il manuale, farvi una birra guardando una puntata dei Simpson, e soltanto dopo ricominciare dalla spiegazione di cosa è la legge di Zipf.

[19] Il mio lavora solo quando mi distraggo: vederlo lavorare mi fa venire i sensi di colpa, quindi cerco di farlo smettere ogni volta che lo scopro.

```
    foreach $forma (@forme)
        {
        $parole{$forma}++;
        $numero++;
           }
    }
# ordina per occorrenze
foreach $forma
       (sort {$parole{$b}<=>$parole{$a}} keys(%parole))
    {
    $c++ if $parole{$forma}!=$vval;
    $zipf = $parole{$forma}*$c;
    print "$c) $zipf \t\t $forma \t\t $parole{$forma}\n";
    $vval = $parole{$forma};
    }

$tot = keys(%parole);
print "Nel documento ci sono $numero parole
($tot forme non ripetute)\n";
```

Sta tutto nell'ultimo `foreach`: nella prima linea l'incremento della variabile `$c`, che corrisponde al rango, è sottoposta ad una condizione. Traduciamo la linea: *incrementa di uno il valore di $c se il numero di occorrenze della parola $forma è diverso da $vval.*

Prima di chiederci cosa significa aggiungiamo un tassello: l'ultima linea di questo ciclo iterativo riporta il valore di `$vval`: è uguale alla frequenza della parola in esame.

Ora possiamo chiederci cosa significa il periodo ipotetico. Per capirlo, possiamo immaginare cosa accade ad ogni iterazione. Immaginiamo che le prime tre parole per frequenza del testo siano *a, di, non* occorrenti rispettivamente 775, 523 e 523 volte.

Alla prima iterazione (*a*, 775 occorrenze) il rango `$c` sarebbe uguale a zero, ma viene portato ad uno perché la frequenza della parola in esame, 775, è diversa dall'attuale valore di `$vval`, cioè zero. Dunque viene calcolata `$zipf`, viene prodotto un output e il valore di `$vval` diventa 775.

Alla seconda iterazione (*di*, 523 occorrenze), il rango `$c` diventa 2, perché `$vval` (uguale a 775) è diverso da 523, che è il numero di occorrenze della parola in esame. Di nuovo, viene calcolata `$zipf`, è prodotto l'output e `$vval` diventa uguale a 523.

Alla terza iterazione (*non*, 523 occorrenze), il rango `$c` non viene incrementato, perché `$vval` è uguale al numero di occorrenze della parola in esame.

Nella variabile `$vval`[20] risiede la "memoria" del mio programma, al quale interessa ricordare un solo dato per volta: il numero di occorrenze della parola appena vista. Questa

[20] Che non a caso sta per: *vecchio valore.*

informazione viene confrontata con il numero di occorrenze della parola in esame e, se c'è differenza, il rango aumenta, altrimenti no.

Non avevo detto che era intuitivo?

6.4 Estrattore di concordanze

6.4.1 Risultato desiderato

Le concordanze di un'opera o di un autore sono costituite dall'elenco di tutte le parole impiegate in quell'opera o da quell'autore e dei passi in cui esse compaiono. Per estensione è pure lecito riferirsi alle concordanze di una singola parola, intendendo la raccolta di tutti i contesti nei quali tale parola compare.

Vogliamo scrivere, per una persona che in cambio saprà farci raggiungere le più alte vette del piacere, un programma in grado di estrarre le concordanze di una parola data in un testo dato.

Come output ci basterà avere l'intera frase nella quale la parola data compare[21].

6.4.2 Codice

```
my $conta = 0;
print "pattern: ";
my $pattern = <STDIN>;
print "dir: ";
my $dir = <STDIN>;
chomp($dir);
my($testo,@frasi,$frase);

opendir(DIR, "$dir") or die "non so aprire $dir: $!";
while (defined($file = readdir(DIR)))
    {
    next if $file =~ /^\.\.?/;
    open(TEMPO,"< $dir/$file") or print "ho fallito: $!";
    while(<TEMPO>)
        {
        $testo .= $_;
        }
    @frasi = split /(?:[.?!][^.?!\w])/sox, $testo;
    foreach $frase(@frasi)
        {
        if($frase=~/$pattern/gi)
```

[21] All'inizio del paragrafo "orizzonti" troverete alcune informazioni su come usare il listato qui presentato.

```
            {
            $conta++;
            print ".......$conta, in $file..........\n";
               print "$frase.\n";
            }
        }
    close(TEMPO);
    $testo = "";
    }
print "trovate $conta occorrenze\n\n";
```

6.4.3 Commenti

Questo codice è abbastanza complesso, e richiede qualche spiegazione.

Possiamo immaginarlo costituito da due parti: la fase di acquisizione delle informazioni e lo svolgimento del programma. La seconda parte del programma va ulteriormente divisa in tre parti, che possiamo identificare con tre strutture di controllo che implicano un ulteriore grado di indentazione: il primo `while`, il secondo `while` e il ciclo di `foreach`.
Ognuna di queste tre parti (le ripresenterò procedendo, ma controllatele ADESSO nel codice) inizia in realtà nella linea prima rispetto a quella nella quale compare la struttura di controllo: il primo `while` inizia con la funzione `opendir()`, il secondo con la funzione `open()`; il foreach con la creazione della variabile array `@frasi`.

Non proseguite prima di aver individuato le linee di codice che ho menzionate.

Visti? Bene, ora possiamo proseguire.
La prima parte, quella di acquisizione delle informazioni, non dovrebbe presentare motivi di stupore o preoccupazione: si tratta del solito codice che prende dello STDIN e lo ciompa in variabili da usare.

Vediamo il primo blocco della seconda parte:

```
opendir(DIR, "$dir") or die "non so aprire $dir: $!";
while (defined($file = readdir(DIR)))
```

```
{
```

Abbiamo già visto la funzione `opendir()` e anche quel che segue suonerà familiare: se non riesce ad aprire la cartella, il listato avverte del problema: la funzione `die()` stampa direttamente nello STDOUT un segnale d'allarme[22].

UN DETTAGLIO NON SECONDARIO. In generale, quando si deve aprire un file, fornire al programma una alternativa è cosa buona e giusta: esistono numerosi casi, soprattutto quelli nei quali devono essere aperti molti documenti – automaticamente e uno dietro l'altro – nei quali se non si pone una coordinata avversativa il programma funziona lo stesso, ma vengono stampati numerosi messaggi di errore nello STDOUT

La seconda linea è più articolata; iniziamo a leggere la riga da destra[23].

A destra troviamo il verbo *readdir* che regge come suo argomento un dirhandle, dovrebbe esservi chiaro. Come dovrebbe esservi chiaro il funzionamento di questo verbo, che itera la lettura restituendo ad ogni ripetizione un elemento letto dalla cartella. Ad ogni ripetizione (spostiamoci verso sinistra) la variabile `$file` ha quindi un nuovo valore. Questo valore viene sottoposto ad una verifica: ancora più a sinistra la funzione `defined()` verifica se il valore di `$file` corrisponde effettivamente ad un file[24] e non, ad esempio, ad una cartella o ad un collegamento o ad un alias.

Se la funzione *defined()* restituisce un responso favorevole, si parte per la guerra, e il ciclo `while` inizia a processare, uno per volta, gli elementi nel suo elenco che sono effettivamente file utili.

Vediamo ora il secondo blocco.

```
next if $file =~ /^\.\.?/;
open(TEMPO,"< $dir/$file") or print "ho fallito: $!";
while(<TEMPO>)
    {
    $testo .= $_;
    }
```

Per prima cosa, subito dopo l'apertura delle parentesi graffe, il programma verifica che il nome di file non inizi con un punto o un doppio punto (riuscite a decrittare l'espressione

[22] Il valore della variabile `$!` lo trovate nella sezione sulle variabili predefinite a pag. 53.

[23] Iniziare a leggere da destra quando non si capisce è una pratica che mi fu suggerita anni fa da un docente di filologia, il quale spiegava che si tratta di un'ottima strategia per rintracciare l'etimologia di una parola. Portava l'esempio della parola *pezzente*, che spesso si immagina derivata da *pezza*. Invece – come la flessione della parola suggerisce, in fondo a destra – si tratta di un participio presente, che quindi chiede come radice non un sostantivo, come *pezza*, ma un verbo, come il latino *petere* [chiedere]. Da cui si desume che *pezzente* non significa [portatore di pezze] ma [chi chiede], [questuante].

[24] `defined()` è una funzione piuttosto ricca, con la quale generalmente si usano gli operatori di confronto file che non abbiamo visto. In questo caso viene presentata come standard una sintassi che standard non è, visto che compare un solo argomento.
Ma se volete aprire le vostre porte della percezione, potete iniziare approfondendo la sintassi di questa funzione e gli operatori di confronto file.

regolare?). Di solito con quei caratteri iniziano file di sistema o di preferenze, che a noi e alla persona per la quale scriviamo il codice certamente non interessano.
Qualsiasi perversione ha un limite.

Se il nome del file non inizia con un punto (in tal caso semplicemente il listato passa al nome successivo), allora il documento viene aperto in sola lettura. Oppure viene stampato ancora nello STDOUT un avvertimento di occorso errore.

Come potete vedere, in questo caso `print()` e `die()` hanno un effetto in comune: entrambi stampano un avvertimento, ma c'è una differenza che non è visibile in fase di stesura del programma: `die()` interrompe l'esecuzione del programma, lo chiude *manu militari*.
Ha dunque senso impiegare due diverse funzioni? Certo: se il programma non riesce ad aprire l'intera cartella non ha senso che rimanga aperto, quindi è naturale usare `die()`; se invece non riesce ad aprire solo uno o più documenti è bene che chi usa il programma ne sia informato, ma non c'è motivo perché l'esecuzione dell'intero programma ne sia compromessa, quindi si usa `print()`.

Quel che accade tra le parentesi graffe, mentre il documento viene letto, è il progressivo incremento di una variabile di testo, che si chiama appunto `$testo`. Alla prima ripetizione del ciclo (primo frammento del primo file), la linea in questione, `$_` viene concatenata a `$testo` (che prima del concatenamento è uguale a zero). Ad ogni successiva iterazione, `$_` viene sempre aggiunta a `$testo` fino a quando il documento non è stato letto tutto.

Quando il documento è finito, il suo intero contenuto è memorizzato nella variabile `$testo`, che può così essere sottoposta alle manipolazioni del caso.

Ora attenzione, perché bisogna fare un salto al di là del terzo blocco, fuori dal ciclo `foreach`, dove rimane un'appendice di due righe di codice che appartengono logicamente al secondo blocco:

```
close(TEMPO);
$testo = "";
```

Cosa avviene all'interno di queste due linee di codice? Dopo le manipolazioni descritte nel ciclo `foreach`, viene chiuso il documento aperto e viene cancellato l'intero contenuto di `$testo`. Altrimenti alla fine della lettura del documento successivo avremmo come valore di questa variabile il contenuto del documento appena letto e anche il contenuto di tutti gli altri documenti letti: il contenuto del quinto documento sarebbe uguale alla somma dei contenuti del primo, del secondo, del terzo, del quarto e del quinto documento. Il che è ovviamente senza senso.

Veniamo infine il terzo blocco, il più interessante.

Avevamo detto che l'output doveva essere costituito dalle frasi nelle quali compariva la forma cercata; quindi dobbiamo dividere in frasi. Frugate nell'apparente banalità della mia ultima asserzione e troverete il primo problema da risolvere: né l'interprete perl né tantomeno il resto del vostro computer sanno cosa diamine sia una frase.

Provate a immaginare come spiegarglielo, e se proprio non vi viene in mente leggete il codice qui di seguito.

```
@frasi = split /(?:[.?!][^.?!\w])/sox, $testo;
foreach $frase(@frasi)
    {
    if($frase=~/$pattern/gi)
        {
        $conta++;
        print ".........$conta, in $file............\n";
           print "$frase.\n";
        }
    }
```

Visto? proprio la prima linea di codice istanzia l'array @frasi e lo popola[25]: come fa? Con *split()* e una espressione regolare.

Leggetela con attenzione: */(?:[.?!][.?!\w])/*. Io ci vedo tre paia di parentesi: il primo, di tonde, che a loro volta contiene gli altri due più un punto interrogativo e due punti seguiti da due di parentesi quadre.
Le parentesi tonde con il digramma *?:*, usate insieme alla *x* come modificatore dell'espressione regolare, rendono semplicemente l'esecuzione di quel frammento di codice più veloce[26]. A noi interessano soprattutto le altre due parentesi.

Sono di facile lettura; guardate le prime: potrebbero essere parafrasate come: *un punto fermo o un punto interrogativo o un punto esclamativo*[27].

Le altre due parentesi quadre possono essere lette come: *diverso da un punto fermo o un punto interrogativo o un punto esclamativo o una lettera*.

Quindi come criterio di segmentazione alla funzione `split()` è stato detto di segmentare ogni volta che incontra un segno di interpunzione di quelli che usualmente segnano la fine di una frase purché non fosse seguito da un altro segno di interpunzione identico o da una lettera. Questo è l'importante. Adesso sprecherò un paio di capoversi per spiegare perché questa espressione regolare è scritta così e non diversamente.

La forma più semplice che si poteva pensare per segmentare in frasi era quella di spezzare ogni volta che si incontravano [.?!]. Su questo siete d'accordo? Bene. Ma cosa

[25] Il verbo *popolare* si usa frequentemente, in ambito informatico, per indicare l'azione di inserire dei dati in una base di dati. Intendo dire, semplicemente, che l'array viene anche dotato di elementi.

[26] Letteralmente indicano che il contenuto delle parentesi non deve essere memorizzato per successive invocazioni tramite `$1`, `$2` eccetera. Il programma sa di non doverle memorizzare, quindi agisce più rapidamente.
Robaccia da nerd.
Si usa ancora il termine *nerd*?

[27] Ricordate che all'interno delle parentesi quadre il punto fermo e il punto interrogativo non si riferiscono a ciò a cui si riferiscono fuori dalle parentesi?

sarebbe accaduto con digrammi o trigrammi del tipo: *?! !? !!! ... !?!* ??? È semplice: `split()` avrebbe visto diverse frasi a lunghezza zero[28].

Per evitare la spiacevolezza di frasi a lunghezza zero si è posta come condizione che i summenzionati segni di interpunzione non fossero replicati, almeno in quanto classe[29]. Inoltre, per evitare di individuare negli indirizzi di e-mail o nelle URL[30], che frasi non sono, i segni di interpunzione non dovevano essere seguiti da un qualsiasi carattere alfanumerico (\w). In fondo, se non sono indirizzi di posta elettronica o di sito web, immediatamente dopo il punto ci dovrebbe essere uno spazio.

Ora che avete capito il perché di quella espressione regolare, proseguiamo. Isolate le singole frasi, il programma le legge una per volta e, se il `$pattern` richiesto dall'utente compare in una frase, la stampa. Prima, però, stampa un indice numerico progressivo (la variabile `$conta`) l'indicazione del file nel quale la frase compare.

Ultima annotazione: nell'istruzione `print ‘‘$frase.\n’’;` compare un punto. Quel punto non è in perl, ma in italiano; infatti `split()` non solo segmenta, ma elimina anche quel che ha trovato, ciò che gli permette di segmentare. Per evitare lo spiacevole effetto di concordanze senza nemmeno uno straccio di punto fermo alla fine[31], abbiamo scritto un listato che aggiunge indiscriminatamente un punto fermo, pure al prezzo di perdere il segno interpuntivo originario. Tutto qui.

Abbiamo visto come il programma apre la cartella selezionata, come apre ogni file e come fa a isolare le singole frasi, a cercarle ed eventualmente a scriverle nello STDOUT (a voi la libertà di modificare il listato in modo da scrivere in un nuovo file[32]).

6.4.4 Orizzonti

Uso del programma

Prima di prospettarvi possibili linee d'azione, ecco alcune note d'utilizzo importanti per il funzionamento di questo programma.

Esiste una minima varietà nel modo in cui l'interprete perl sbircia nelle cartelle dei vari

[28] Questo avrebbe comportato significative conseguenze sull'esecuzione del programma? No, ma da un punto di vista meramente linguistico sarebbe stata una schifezza. Il fatto di scrivere codice non ci autorizza a ignorare la lingua italiana: siamo persone civili, checcavolo.

[29] Che cioè ad un qualsiasi segno non seguisse un altro qualsiasi segno di quelli elencati, non solo che ogni segno non fosse duplicato o triplicato.

[30] Gli indirizzi dei siti web.

[31] Che gli appassionati di Woobinda (Aldo Nove) potrebbero apprezzare.

[32] L'idea di scrivere il risultato di un programma in un nuovo file, potrebbe dimostrarsi di grande utilità. Valutate il fatto che programmatori disinvolti con l'elaborazione di testi – quali voi potreste diventare con un po' di esercizio – sono poi soliti, ad esempio, rielaborare con altri listati i risultati di precedenti elaborazioni. Immaginate di voler estrarre una lista di frequenza solo delle frasi che contengono una certa forma; con corpora molto grandi potrebbe trattarsi di un modo spiccio per verificare se la forma da noi cercata si presta a particolari costruzioni sintattiche o co-occorre con specifici insiemi di altre parole.

sistemi operativi, quindi è possibile che all'inizio non riesca ad aprire la cartella per questo motivo o perché voi avete dato un indirizzo di cartella sbagliato.

Se la cartella che contiene il corpus si trova nella stessa cartella che contiene il programma, dovrebbe essere sufficiente digitare, quando il programma chiede di specificare la cartella,
`$> nomecartella/` [invio]
Ma non è bene escludere a priori, se quel primo tentativo non va a buon fine, la possibilità con l'asterisco:
`$> nomecartella/*` [invio]

Se la cartella del corpus non si trova nella stessa cartella nella quale si trova il programma, dovrete definire l'intero percorso con la sintassi del doppio punto. Facciamo tre esempi:

1. il programma concorDancer.pl che abbiamo appena scritto si trova nella cartella fuffa che si trova nella cartella Desktop. Nella cartella fuffa si trova anche la cartella LettCommerc06 nella quale volete cercare concordanze di una particolare forma.

   Alla richiesta del programma di specificare la cartella risponderete:
   `$> LettCommerc06/` [invio]
   che significa: [apri LettCommerc06].

   Questo tipo di indirizzo è **relativo**, perché "muove" a partire dal punto nel quale si trova il programma. Si può anche usare un indirizzo/percorso **assoluto**, che cioè parte da un punto convenzionale di origine. In tal caso il percorso su X sarà qualcosa del tipo:
   `$> /home/utente/Desktop/fuffa/LettCommerc06/` [invio]
   e su Windows sarà qualcosa del tipo:
   `$> C:/WIN/Desktop/fuffa/LettCommerc06/` [invio]

2. il programma concorDancer.pl che abbiamo appena scritto si trova nella cartella fuffa che si trova nella cartella Desktop. Sempre sul Desktop si trova la cartella Corpora che contiene la cartella LettCommerc06 nella quale volete cercare concordanze di una particolare forma.

   Alla richiesta del programma di specificare la cartella risponderete:
   `$> ../Corpora/LettCommerc06/` [invio]
   che significa: [esci da questa cartella salendo di un livello][33], [apri Corpora/LettCommerc06].

   Nel caso di percorsi assoluti dovremmo scrivere su X:
   `$> /home/utente/Desktop/Corpora/LettCommerc06/` [invio]
   e su Windows:

[33] Si "sale", perché il Desktop contiene la cartella fuffa.

`$> C:/WIN/Desktop/Corpora/LettCommerc06/` [invio]

3. concorDancer.pl si trova nella cartella Esperimenti, nella cartella fuffa che si trova in Desktop. Sempre sul desktop si trova la cartella Corpora che contiene la cartella LettCommerc06 nella quale volete cercare concordanze di una particolare forma.

   Alla richiesta del programma di specificare la cartella risponderete:
   `$> ../../Corpora/LettCommerc06/` [invio]
   che significa: [esci da questa cartella salendo di due livelli], [apri Corpora/LettCommerc06].

   Gli indirizzi assoluti non cambiano rispetto a quelli già visti per l'esempio precedente.

Ricordatevi anche che per leggere dei file dovete averne i permessi: se il programma non riesce ad aprire i documenti, provate a controllare che è il loro proprietario e che accesso è permesso agli altri.

Infine, valutate il fatto che potete chiedere al programma di cercare una espressione regolare!

Due esercizi

Potete provare a riscrivere il programma usando l'operatore diamante, in modo da passare come argomento del programma (quando lo lanciate da shell) la cartella nella quale deve cercare.

Vi renderete conto di quanto l'uso di questo operatore semplifichi e velocizzi il vostro listato.

Il fatto che alla fine di ogni frase non compaia che un punto fermo potrebbe far storcere il naso alla persona per la quale abbiamo scritto questo listato[34].

Provate a fantasticare su come potrebbe essere fatto un programma che ricordi il tipo di interpunzione e la ponga laddove serve. Poi provate a realizzare la vostra fantasia.

6.5 Correttore semi-automatico

6.5.1 Risultato desiderato

Abbiamo già visto un correttore automatico[35]; adesso passiamo a qualcosa di più interattivo.

Immaginate di voler eliminare da una raccolta lunghissima di leggi, di e-mail o di concordanze tutti i blocchi di testo che contengono una data parola o una espressione regolare.

[34] Ricordate sempre che il nostro scopo non era scrivere il programma, ma raggiungere le vette del piacere, come detto in fase di presentazione dell'estrattore di concordanze.

[35] Pag. 87 e segg.

Nel listato che state per leggere incontrerete inoltre due nuove funzioni, molto particolari, potenti ed utili che rispondono ad una domanda che vi ho posta parlando di funzioni: e se io potessi definire delle mie funzioni, sarebbe una cosa buona oppure solo una perdita di tempo?

6.5.2 Codice

```
sub informazione
    {
    $che = shift();
    print "dimmi $che:\n";
    $pixel = <STDIN>;
    chomp $pixel;
    return $pixel;
    }

my $daaprire = informazione("il file da aprire ");
my $dabeccare = informazione("la parola da beccare ");
$daaprire2 = informazione("il nome del nuovo file ");
my ($testo, $tot, $canc, $poss) = "";

open (FILE1, "<$daaprire") or die("problema con
                                  $daaprire: $!\n");
while(<FILE1>)
    {
    $tot++;
    $cosa = "N";
    if(/$dabeccare/)
        {
        $poss++;
        $cosa = informazione("se cancellare (Y/N) $_");
        }
    if($cosa eq "Y" || $cosa eq "")
        {
        $canc++;
        next;
        }
    $testo.=$_;
    }
close(FILE1);
```

```
open (FILE2, ">$daaprire2") or die("problema con
                                    $daaprire2: $!\n");
print FILE2 $testo;
close(FILE2);

print "aperto $daaprire, corretto in $daaprire2\n
    corrette occorrenze della parola: $dabeccare\n
    totale linee: $tot\n
    linee viste: $poss\n
    linee cancellate: $canc\n";
```

6.5.3 Commenti

Cinque blocchi, in questo listato, divisi da una riga bianca: li potete vedere ad occhio nudo, senza il filtro della vostra competenza in fatto di linguaggio perl. Si tratta della definizione di una nuova funzione; dell'acquisizione di dati; dell'apertura del file; della produzione di file corretti (che nel seguito saranno trattati insieme); della generazione di un output di controllo.

Ritratto di signora

Nel primo blocco campeggia la funzione `sub{}`. Avete letto bene: due graffe invece di due tonde, perché questa, signore e signori, è una funzione con l'abito da cerimonia, dedicatele per cortesia qualche attimo di contemplazione, perché se continuate a programmare potrebbe diventare una delle vostre migliori amiche. `sub{}` è la funzione che descrive altre funzioni.

Come funziona? Ecco la sua sintassi:

```
sub nomeFunzioneDecisoDaVoi
    {
    codice che deve eseguire
    }
```

Per ora accontentatevi di sapere che si tratta della descrizione di una funzione, nel sottoparagrafo successivo vedremo come si usa.

Poiché questo programma, in quanto semi-automatico, richiede un certo grado di interazione con l'utente (viene chiesto non solo cosa trovare e dove, ma anche, per ogni frammento di testo trovato, se deve essere cancellato oppure no), lo scopo della nuova funzione è quello di ridurre nel listato le righe di codice dedicate all'interazione. In particolare, questa funzione chiede un'informazione e la restituisce: fa tutto il lavoro di STDIN e `chomp()` che avete già visto altrove. Osservate:

```
1  sub informazione
2     {
3     $che = shift();
4     print "dimmi $che:\n";
5     $pixel = <STDIN>;
6     chomp $pixel;
7     return $pixel;
8     }
```

Nella prima linea viene definita (o battezzata) la funzione, che si chiamerà `informazione()`.

Nella terza linea compare una nuova funzione: `shift()`, che ha il compito di "prendere" un argomento della funzione e di immagazzinarlo in una variabile (in questo caso la variabile `$che`).

Come leggiamo nella linea 4, la variabile `$che` corrisponde a quello che il programma chiede all'utente; potrebbe trattarsi del nome di un file o di un pattern o di un valore numerico, a noi non interessa: `informazione()` ha come primo argomento l'oggetto di una richiesta.

Nella riga 5 viene creata una variabile che ha come valore quel che l'utente risponde; nella riga 6 tale valore è ciompato nel modo consueto; nella riga 7 tale valore viene restituito con la funzione `return()`[36].

Sono operazioni che abbiamo già visto fare, ma questa volta sono inserite in un contesto nuovo, diverso: vediamo le cose dall'"interno".

acquisizione informazioni

Ma come si usa la nuova funzione `informazione()`? Eccolo mostrato nella prima linea di codice che segue la definizione della funzione, la prima della fase di acquisizione delle informazioni:

```
my $daaprire = informazione("il file da aprire ");
```

Viene creata una variabile il cui valore corrisponde al valore restituito dalla funzione `informazione()`. Sappiamo quindi che il valore della variabile `$daaprire` sarà uguale al testo digitato dall'utente nello STDIN in risposta alla richiesta del programma *dimmi il file da aprire:*

Avete capito quel che succede? (Se la risposta è affermativa, saltate pure il resto del capoverso).
Quando devo attribuire un valore alla variabile `$daaprire` invoco la funzione `informazione()`

[36] Questa funzione, comune praticamente a qualsiasi linguaggio di programmazione o scripting è il torbido motivo per il quale spesso persone con un solido retroterra informatico impiegano il verbo *ritornare* come un verbo trivalente dotato di complemento oggetto: *Il programma mi ha ritornato questo valore e non capisco perché!*

dotandola dell'argomento *il file da aprire*. La funzione legge l'argomento e lo trascrive, con `print()`, facendolo precedere dalla richiesta *dimmi*, poi memorizza la risposta dell'utente in una variabile, la ciompa e la restituisce con `return()`. Così restituito, il testo digitato dall'utente finisce dritto dritto nella variabile.
È un piccolo programma.

E vi è chiara l'utilità di scrivere una funzione come questa? (Di nuovo, in caso di risposta affermativa saltate pure il resto del capoverso).
Avrei potuto chiamare la variabile `$x`, tanto sapevo che il valore dell'ipotetica `$x` sarebbe stato "il file da aprire", ma non è questo il vero vantaggio della *subroutine*[37].
Il vero vantaggio si vede leggendo le successive righe di codice:

```
my $daaprire = informazione("il file da aprire ");
my $dabeccare = informazione("la parola da beccare ");
$daaprire2 = informazione("il nome del nuovo file ");
```

Avrei dovuto scrivere tre diversi `print()` con tre diverse variabili che raccoglievano lo STDIN e tre `chomp()` su ognuna di quelle variabili. Non si tratta tanto di quel che scrivo tutto sommato una volta, ma pensate alla **leggibilità** del codice: è infinitamente più chiaro.

Come quando scrivete un testo e fin dall'inizio spiegate quale significato attribuite ad un certo termine, quale accezione, quale sfumatura, quale versione di una teoria, quali dati. Fatto una volta, non c'è più bisogno di ripeterlo. È per questo che in molti libri esistono dei glossari terminologici.

Correzioni

La parte delle correzioni è divisa in due parti: l'interazione con l'utente e la scrittura su file.

La parte di interazione è apparentemente lunga; in realtà ci sono diversi contatori e due periodi ipotetici, ma lo scheletro del programma è piuttosto semplice.

Probabilmente non avrete neppure bisogno dei miei commenti, se leggerete il codice sapendo che `$tot` è il totale dei blocchi di testo; `$cosa` è cosa deve fare il programma (tenere o cancellare); `$poss` è il totale dei blocchi di testo riconosciuti dal programma (nei quali compare il pattern `$dabeccare`) e `$canc` è il totale dei blocchi cancellati.

```
open (FILE1, "<$daaprire") or die("problema con
                                  $daaprire: $!\n");
while(<FILE1>)
    {
    $tot++;
    $cosa = "N";
```

[37] Questo il significato del trigramma *sub*.

```
    if(/$dabeccare/)
        {
        $poss++;
        $cosa = informazione("se devo cancellare (Y/N) $_");
        }
    if($cosa eq "Y" || $cosa eq "")
        {
        $canc++;
        next;
        }
    $testo.=$_;
    }
close(FILE1);
```

Si tratta sempre di cose che abbiamo già visto, anche per differenti usi: viene aperto un file in lettura, ad ogni nuovo blocco di testo letto si aggiorna la variabile `$tot` e si imposta la variabile `$cosa` uguale ad *N*, cioè [no], [non cancellare, tieni il blocco di testo in questione]. Poi si verifica che il blocco di testo contenga il pattern cercato, se è così si aggiorna la variabile `$poss` e si chiede cosa fare (con la funzione `informazione()`). Subito dopo, se la variabile `$cosa` è uguale a *Y* [yes, sì, butta il blocco nel cesso] oppure se la variabile `$cosa` è uguale a niente[38], si aggiorna `$canc` e si salta il resto del codice (con `next()`).
Se invece `$cosa` è uguale ad *N* o a qualsiasi altro carattere diverso da *Y* e da [invio], allora il blocco attuale viene memorizzato nella variabile `$testo`.
Come al solito.

A questo punto diventa importante il blocco di scrittura su file, il quale compie un'operazione davvero molto semplice, guardate:

```
open (FILE2, ">$daaprire2") or die("problema con
                                    $daaprire2: $!\n");
print FILE2 $testo;
close(FILE2);
```

Apre, scrive, chiude. Di una linearità disarmante.

Output

A questo punto il programma ha finito di fare quel che si voleva che facesse, ma noi, che amiamo certi dettagli, abbiamo ancora un piccolo rapporto finale sull'andamento dei lavori:

```
print "aperto $daaprire, corretto in $daaprire2\n
```

[38] Come accade se l'utente preme soltanto [invio], visto che quel carattere è poi cancellato da `chomp()`. I blocchi di testo sono quindi cancellati sia con la pressione del tasto [Y] che con la pressione del tasto [invio].

```
    corrette occorrenze della parola: $dabeccare\n
    totale linee: $tot\n
    linee viste: $poss\n
    linee cancellate: $canc\n";
```

Direi che non c'è nulla da aggiungere.

6.5.4 Orizzonti

Quando ho scoperto la funzione `sub{}` ho smesso di fare quel che stavo facendo ed ho iniziato a scrivere un altro programmino, per gioco, in cui due macchinine facevano a gara.

Era un programma molto rozzo e brutto, che non ho più con me e non ho tempo né voglia di riscrivere; vi basti sapere che le automobiline non si vedevano affatto: vedevo solo a che punto del percorso di cento chilometri erano le auto R e B, che acceleravano o deceleravano in base alla velocità del turno precedente, alla posizione di vantaggio o svantaggio e alla distanza dalle curve (c'erano tre curve: sul venticinquesimo chilometro, sul sessantesimo e sul novantesimo; la gara durava tre giri). Avrei potuto scrivere lo stesso listato senza `sub{}`, ma l'avere questa nuova possibilità mi aveva aperto le porte dell'immaginazione.

Potrei suggerirvi di provare a modificare il correttore semi-automatico in modo da farlo lavorare sulle frasi (come l'estrattore di concordanze), ma se vi viene in mente qualcosa di meglio non vedo perché non dovreste provare a fare quel qualcos'altro. Il manuale avrà raggiunto il suo scopo.

6.6 Congedo

A posteriori devo ammettere che esistono diversi manuali di informatica brillanti: pur senza raggiungere le vette del libretto rosso di Photoshop, molti testi contengono battute, citazioni e anche aneddoti divertenti, come se gli autori di questo genere di manuali sapessero che quello che stanno raccontando è potenzialmente la roba più noiosa del mondo.

A me l'idea di scrivere un manuale leggero nei toni è venuta quando per la prima volta ho tentato di spiegare la programmazione perl a persone (perché è questo che sono gli studenti) che non avevano mai masticato nulla di informatica. Quando vedi certi cipigli, certe fronti aggrottate, certi occhi da cagnolino sul bordo dell'autostrada o acceleri e scappi oppure ti fermi e provi ad affrontare il problema diversamente.

Visto che l'insegnamento è per me anche un lavoro, la fuga non era una opzione che potessi seriamente contemplare. Ho quindi scritto questo libro insieme ai miei studenti, sperimentando con loro metafore, artifici narrativi e quella infingarda retorica (fatta perlopiù di occhioni sul punto di piangere) che, sola, riesco a gestire.
Se questo libro vi è stato gradito, sappiate che avete un grosso debito di riconoscenza verso quegli studenti. E che potrebbero da un momento all'altro telefonarvi per chiedervi del denaro.

Ora che siete avvertiti, voglio spendere le ultime parole sul perl e su questo manuale[39].

Perl è un linguaggio molto bello, facile ed utile, soprattutto se volete o dovete lavorare sul testo. Rappresenta, sia come oggetto di studio che come strumento, un ottimo punto di partenza per ulteriori approfondimenti sia sul funzionamento delle macchine sia su questioni di natura più propriamente linguistica (non avete idea di quante cose si imparino su un testo dovendolo trattare informaticamente).

Inoltre, conoscendolo potreste fare bella figura in certe occasioni.
Più o meno nello stesso numero di occasioni in cui si potrebbe fare bella figura sapendo usare uno storditore elettrico per bovini, ma questo è un dettaglio.

Se avete intenzione di approfondire la conoscenza di questo linguaggio, oltre a guide di riferimento sia su carta che in rete, posso consigliarvi il ricorso al vero valore aggiunto di Perl: la comunità di sviluppatori (santi, poeti e navigatori).
La comunità degli utenti perl è semplicemente una cosa bellissima, e quasi sempre qualcuno che ne sa più di voi vi spiegherà in due righe perché quello che avete fatto non funziona e come correggerlo. Non vi dico l'emozione, quando qualche guru del perl scende fra i comuni utenti a dispensare saggezza e conoscenza.

In conclusione, questo manuale è solo un primo passo, un po' come la grammatica elementare di accadico di Saporetti ([Sap87]): qualcosa che spiega a grandi linee il sistema verbale e quello nominale senza perdersi nei dettagli che diventano importanti dopo un po'.

Perché, attenzione, all'inizio solo il quadro generale è davvero importante: non la sin-

[39] Prima delle appendici, è chiaro. Speravate di aver finito, eh?

tassi, non le regolette e le scorciatoie, ma la capacità di immaginare in un modo diverso, la tensione verso una nuova pratica di soluzione dei problemi, cui alla bisogna si può ricorrere anche in altri ambiti.

(È stato bello finché è durato, ma amici come prima. Vi richiamo io, al limite).

Appendice A

Alternative di vita (informatica)

A.1 LaTeX, passare a miglior vita

A.1.1 Perché?

Questo manuale è stato scritto tutto con LaTeX.

LaTeX è un insieme di strumenti – basati su un programma che si chiama TeX, scritto tra il 1977 e il 1982 da Donald Knuth – per impaginare testi.

La cosa interessante di questo strumento è che ciò su cui si lavora è puro testo, senza procedure[1]. Mi spiego: quando su un moderno *editor* di testi dovete scrivere qualcosa in grassetto, potete procedere in due modi:

1. scrivete la parola, poi la selezionate, infine premete il bottone del grassetto e la deselezionate;
2. prima di iniziare a scrivere la parola premete i tasti [ctrl][b] oppure [mela][b], scrivete la parola e poi ripremete le coppie di tasti di dovere.

Con LaTeX invece si scrive:

```
\textbf{parola}
```

E volete vedere come ho appena scritto l'elenco numerato che voi avete appena letto? Così:

```
\begin{enumerate}
\item scrivete la parola, poi la selezionate, infine premete
 il bottone del grassetto e la deselezionate;
\item prima di iniziare a scrivere la parola premete i tasti
 [ctrl][b] oppure [mela][b], scrivete la parola e poi
 ripremete le coppie di tasti di dovere.
\end{enumerate}
```

[1] Cioè sequenze di azioni, come cliccare su un tasto disegnato nell'interfaccia.

Tutto è solo testo, non ci sono procedure. I vantaggi? Vale la pena che alcuni di essi li scopriate se avete voglia di scaricarvelo e imparare il minimo di sintassi e lessico[2]; io vi dico solo che:

- i miei documenti sono di puro testo, in termini di memoria leggerissimi anche quando sono estremamente lunghi e articolati;
- come cercate gli elenchi numerati? Io uso normalmente la funzione `trova` del mio editor di testi;
- ho una cartella, nel desktop, che si chiama "bibliografie", nella quale sono archiviati tutti i testi che man mano cito nei miei articoli, interventi a conferenze, poster e libri. Quando scrivo i testi cito solo il loro codice, e un pezzo di LaTeX che si chiama BIBTeX crea per me la bibliografia selezionando solo i testi che ho citato;
- avete presente tutti i rimandi ad altre pagine? Be', di certo non sono andato io a cercare i riferimenti corretti e ad aggiornarli ogni volta che era il caso;
- il mio programma non si blocca, mai. E il mio output nativo è in PDF, questo significa che quello che vedo sullo schermo del portatile è esattamente quello che vedrò sulla carta;
- è un programma dell'ottantadue, non ci sono problemi di compatibilità con versioni più vecchie o più nuove, gira su qualsiasi macchina, anche quelle a vapore;
- solo testo, credo che infilare un virus qua dentro – e non farsi scoprire – sia più difficile;
- è gratis ed open source: non lo pagate, non pagate i manuali ed è tutto legale;
- so che sembra incredibile, ma se pensate che lavorare sulla tastiera è inevitabile, smettere di usare il mouse e concentrarsi solo sulle parole rende il lavoro molto più veloce;
- se avete bisogno di "trattare" i vostri testi con degli script perl, i programmi non perdono tempo con le intestazioni nascoste e le informazioni occulte dei programmi ad interfaccia grafica: volete trasformare tutti i grassetti i corsivo e tutti i corsivi in grassetto? In sei linee di codice potete farlo;
- e qui la finisco, giuro. Lavorare sul solo testo (nel quale, ad esempio, non ci sono gli stili, ma ci sono i codici per l'inizio dei capitoli, dei paragrafi e sottoparagrafi) costringe ad un utilissimo lavoro di progettazione iniziale.

[2]Ebbene sì: anche LaTeX richiede la competenza in un linguaggio. Che però non è un linguaggio di programmazione; è più facile: è un linguaggio di formattazione come l'HTML.

Insomma, come per Linux: una curva di apprendimento più ripida all'inizio, ma il godimento in corso d'opera è notevole. E anche alla fine, quando vedete il risultato delle vostre fatiche (spesso gli altri guarderanno con invidia i vostri documenti impaginati da dio).

A.1.2 Come?

In generale, dovrebbe essere sufficiente scaricare LaTeX o TeX dalla rete: basta chiedere a Google "latex for windows" o "latex for mac" (su Linux dovrebbe essere già installato) e sgattare un po' tra i risultati.

In generale sia per win che per mac esistono ottimi programmi:
miktex per win (http://www.miktex.org/)[3]
TeXshop per mac (http://www.uoregon.edu/~koch/texshop/).

Entrambi i siti che ho menzionato forniscono inoltre alcuni interessanti collegamenti a pagine in rete e manuali per l'uso di LaTeX, che consiglio vivamente a chiunque decida di intraprendere il cammino dei Fortunati Redattori di Testi del Santo LaTeX.

Personalmente ho iniziato con un manuale molto semplice ed efficace (anche se incompleto) che si intitola "Impara LaTeX! e mettilo da parte". Dovreste poterlo scaricare da:
www.mat.uniroma1.it/centro-calcolo/ manuali/impara_latex.pdf

Infine, vale sempre l'invito a dare un'occhiata al sito del Gruppo Utilizzatori Italiani di TeX:
http://www.guit.sssup.it/

A.1.3 Quando?

Che domande, subito!

A.2 Altri casi in cui può essere utile Perl

Abbiamo visto che perl può essere usato per il lavoro sui testi[4]. Ma non tutti gli informatici sono dei linguisti, quindi potreste chiedervi perché ad un certo punto degli informatici abbiano sviluppato una cosa come il perl.

Il motivo, molto semplice, è che nei sistemi operativi X-like **tutto è un file** (scritto apposta in grassetto, perché si tratta di uno dei fondamenti del sistema operativo UNIX, padre di tutti i sistemi operativi X-like).
Tutto è un file significa che per la macchina non sono file solo i file e le cartelle, ma anche le periferiche (il mouse, la videocamera), gli altri computer, i programmi, i flussi di dati. Ma se tutto è file, per quanto inumanamente scritto, può essere letto, dunque è bene avere uno

[3]Tutti i collegamenti menzionati qui e altrove nel manuale sono aggiornati al 31 marzo 2006. Ma se anche qualcosa non dovesse funzionare un minimo di intraprendenza vi caverà sicuramente d'impiccio.

[4]Se questo dettaglio vi è sfuggito, forse questa non è la fine del romanzo che stavate leggendo.

strumento – il perl appunto – che aiuti gli utenti umani a leggere, correggere, analizzare anche i file meno umanamente leggibili.

Il motivo per cui vi racconto questi dettagli è che hanno delle conseguenze; infatti se tutto è file, anche i programmi, perl può essere usato per scrivere testi che interagiscono con altri testi (i programmi). Questa affermazione, che in teoria è sempre vera, in pratica trova degli ostacoli, ma è più utile pensare a quello che si può fare piuttosto che sprecare tempo su quel che invece è semplicemente impossibile o troppo difficile.

Una delle possibilità che più mi ha colpito all'inizio del mio percorso di apprendimento del perl riguarda quel che il perl può fare per la grafica.

A.2.1 GIMP! e ImageMagik

GIMP! è come Photoshop, solo un po' meno accessoriato e infinitamente più economico (gratuito, per la precisione). Come altri prodotti nati in ambiente Linux[5], però, si è fortemente ispirato ad analoghi prodotti commerciali; quindi dispone di palette, diversi tipi di pennelli, maschere e strumenti vari per selezionare e modificare le selezioni, ha una cronologia e un menù dei livelli... insomma, (quasi) tutto quel che ha Photoshop.

Se siete sufficientemente arditi potrete scrivere per GIMP delle funzioni, in perl, che si chiamano Perl-Fu, e che permettono di adoperare degli effetti visivi e dei filtri personalizzati.

ImageMagick è una suite di programmi da linea di comando (che cioè vengono lanciati direttamente dalla shell) che modificano i file di grafica senza aprirli. Potete ritagliarli, incorniciarli, trasformare la loro estensione, alterare i loro colori, aggiungere scritte senza neanche vederli e fare molto altro. È utile? Non sempre. È veloce? Da morire.

Il perl, che non a caso è stato definito il coltellino svizzero dei programmatori, può essere usato per creare delle semplici interfacce testuali tra l'utente ed ImageMagik. È utile? Tutte le volte che lo è ImageMagick.

[5] L'esempio più immediato è probabilmente OpenOffice.

Appendice B

Sguardi al futuro

In realtà il presente manuale è appena una sbirciatina sul perl, tuttavia desidero spendere qualche parola ancora su alcuni aspetti che finora non sono ancora stati toccati.

B.1 Webscripting in CGI

CGI significa Common Gate Interface ed indica quell'insieme di tecnologie che sono abitualmente usate per fornire contenuti dinamici alle pagine web. Quando consultate un catalogo o un quotidiano in rete, le pagine che leggete sono state composte al momento della richiesta, automaticamente, consultando un database[1]. Lo stesso accade quando inviate un e-mail non dal vostro programma di posta elettronica ma da qualche sito. I CGI fanno tutte queste cose, e anche molto di più, e il Perl è uno dei linguaggi usati per scrivere CGI.

Il Perl è ormai usato meno per la produzione di pagine dinamiche in rete: soppiantato da linguaggi specializzati come PHP, meno potenti ma più facili da usare, parrebbe avere pochi vantaggi per la scrittura di pagine web.

Eppure dispone di caratteristiche che lo rendono ancora piuttosto utile per la produzione di pagine per Internet; ad esempio sono abbastanza recenti per PHP, ASP, JSP e gli altri linguaggi di *web-scripting* le scorciatoie che il perl fornisce – da quando esiste il modulo GCI.pm – per la produzione di form o moduli da compilare.

Se doveste iniziare da zero, probabilmente non ve lo consiglierei, ma se conoscete già un poco di perl, iniziare ad usarlo anche per i CGI potrebbe fornirvi ottime occasioni per imparare cose nuove.

[1] Nella maggior parte dei casi, ovviamente.

B.2 Interfacce grafiche

Ovviamente, il perl può essere impiegato per scrivere applicazioni autonome, veri e propri programmi con interfaccia grafica che si aprono una loro finestra e da lì agiscono. In tutti i casi, quando si vuole creare un'interfaccia grafica, è necessario scaricare da CPAN (http://www.cpan.org) i moduli necessari, che sono delle estensioni al linguaggio. Una volta che i moduli sono installati, il gioco è fatto. Esistono diverse possibilità; io ne menzioni un paio, perché a volte basta avere un'indicazione di massima per capire che cosa si sta guardando.

Il modulo storico per interfacce grafiche per il perl si chiama Tk [toolkit]; è molto documentato (questo è importante, quando si inizia a fare qualcosa di nuovo) anche se soprattutto in inglese, e piuttosto ricco di possibilità: con Tk riuscirete a fare più o meno tutto quel che vi viene in mente.

Il motivo per cui è bene almeno menzionare una alternativa è che Tk è francamente bruttino, da un punto di vista estetico, e fuori dai sistemi operativi Linux o unix Solaris è troppo diverso dal sistema operativo stesso. Tk, questo è il punto, porta con sé un corredo di immagini che usa per disegnare sullo schermo le finestre, i bottoni e i menù che costituiscono la vostra interfaccia grafica; quindi la probabilità che il corredo di Tk sia coordinato con il vostro sistema operativo è decisamente bassa.

wxPerl funziona secondo un'altra logica, perché "prende in prestito" gli elementi grafici del sistema operativo, quindi, ovunque lo usiate, produrrà un'interfaccia perfettamente coordinata con il resto di quel che vedete sullo schermo.

Gli aspetti negativi di wxPerl risiedono tutti nella sua giovane età: è poco documentato, ha ancora qualche baco ed è un po' più difficile di Tk (ma, per alcuni versi, più potente).

Tuttavia, se cercherete in rete ragguagli sull'argomento, certamente troverete moltissime informazioni su queste e su altre possibilità (come OpenGL, ad esempio) concesse ai programmatori di perl.

B.3 Moduli e Perl Object Oriented

In questo manuale abbiamo visto applicato un paradigma di programmazione: la programmazione imperativa o procedurale[2].

La programmazione imperativa pensa il programma come una linea (non necessariamente una retta semplice) da un punto A ad un punto Z e, pur immaginando possibili corsi e ricorsi, procede decisa verso la sua meta.
Se ripensate a tutti i listati che abbiamo visto fino ad ora, riconoscerete in essi una logica comune, come una sequenza si istruzioni tutto sommato ordinata.

[2]Per un approfondimento potete vedere ad esempio [Man03], pagg. 46 e segg.

Esistono altri paradigmi di programmazione, che implicano idee diverse riguardo a cosa è e come funziona un programma.

Nella programmazione orientata agli oggetti il programma viene scomposto in moduli, segmenti, oggetti. Questi oggetti sono realizzazioni di classi, cioè di oggetti teorici (dotati di proprie variabili e di proprie funzioni) ed astratti. Quando il programma viene avviato, le classi sono istanziate negli oggetti ed eseguite.

In qualche modo è la logica della *subroutine* `informazione()`, ma è anche molto altro, perché un conto è pensare ad una sequenza nella quale alcuni frammenti sono ripetuti (e allora ha senso farne un modello e richiamarlo ogni volta) ed un conto è (ri)pensare l'intera sequenza come qualcosa di diverso da una sequenza.

In parte, poi, le differenze tra i paradigmi di programmazione non possono essere visibili a questo livello di descrizione, perché sono eminentemente tecniche. Ad esempio, nella programmazione orientata agli oggetti da una classe può essere fatta derivare un'altra classe, con caratteristiche simili (ereditate) e con novità che la distinguono come altra classe. E d'altronde ciò che costruisce i vari oggetti (crea delle istanze delle classi) è a propria volta una classe istanziata direttamente dall'utente del programma.

Trovo che il cambiamento di prospettiva si riveli ostico a chi, con solide basi di programmazione procedurale, passi alla programmazione orientata agli oggetti (POO), ma è interessante almeno quanto lo è capire davvero la diffidenza cartesiana o la rivoluzione del criticismo kantiano. Si tratta di una bella sfida intellettuale, intendo dire.

Perl non è nato come linguaggio POO[3], ma esiste un modo di scrivere in perl programmi POO, nel quale ogni oggetto è un `package` che può essere usato con la funzione `use()`.

La sintassi della POO del perl sta alla base dei moduli, che sono collezioni di *subroutines*, o funzioni, che non fanno parte della distribuzione base di perl ma che nel tempo sono state scritte da sviluppatori che ne avevano bisogno per risolvere specifici problemi.

Ho già menzionato CPAN, il grande archivio *on line* di tutti i moduli disponibili: fateci un giro[4], se iniziate a programmare per i fatti vostri. Ci troverete di tutto, dalle cose più inutili (come un modulo per scrivere un dialetto del perl basata sulla lingua degli orchi di J.R.R.Tolkien, nel quale il punto esclamativo sostituisce il punto e virgola e ogni proposizione sembra una bestemmia) a quelle assolutamente essenziali per determinati scopi (come il modulo che riconosce la flessione verbale dell'italiano).

B.4 Perl6

Alla fine, parliamo del futuro di perl e non del vostro.

[3]Per questo è piuttosto criticata la pratica della POO con perl da parte dei puristi.
Se vi interessa intraprendere la via della POO, potete scegliere tra linguaggi diversi dal Perl (come Objective C o Java), ma io vi consiglio Ruby oppure Python. Di Python esiste anche un buon manuale in italiano: [BT01].

[4]http://www.cpan.org

Perl è attualmente alla versione 5.8.* e la comunità sta febbrilmente lavorando alla versione 6, che costituirà una piccola rivoluzione.

Esistono due gruppi di lavoro sulla nuova versione di perl: gli svilupparori del motore e Phalanx.

Gli sviluppatori del motore stanno riscrivendo il perl (l'interprete) in modo che sia un po' più orientato agli oggetti, più performante e che abbia una Virtual Machine come Java, cioè un interprete più diffuso e comune che faccia in modo di non costringere gli sviluppatori a scrivere codice diverso per i diversi sistemi operativi[5].

Phalanx è un esercito di programmatori che gratuitamente e per passione sta invece riscrivendo tutti i moduli in modo che siano compatibili (e soprattutto più performanti[6]) con il nuovo motore.

Queste informazioni vi servono davvero? Forse no, ma se volete togliervi dalle scatole qualcuno che non sopportate potete domadargli: "hai sentito la novità su Phalanx?"

Quando capirà che state parlando di maniaci del perl che lavorano gratis se ne andrà da solo.

[5]Mentre questo accade, ora, per certi programmi molto grossi e impegnativi.

[6]A quanto si può leggere in rete, Perl 6 sarà perfettamente retro-compatibile, ciò significa che non dovremo riscrivere i nostri programmi per passare alla nuova versione.

115. Un'*immagine* ci teneva prigionieri. E non potevamo venirne fuori, perché giaceva nel nostro linguaggio, e questo sembrava ripetercela inesorabilmente.

[([Wit53]) 1999:67]

Bibliografia

[Bon73] Andrea Bonomi, editor. *La struttura logica del linguaggio*. Bompiani, Milano, 1973.

[BT01] David Brueck and Stephen Tanner. *Python 2.1 Bible*. Hungry Minds, Inc., London, 2001. Trad. it. di D. Parola / G. Taiana / S.Marconi (2002), Apogeo, Milano.

[Gig97] Giuseppe Gigliozzi. *Il testo e il computer. Manuale di informatica per gli studi letterari*. Bruno Mondadori, Milano, 1997.

[Ham03] Michael Hammond. *Perl for Language Researchers*. Blackwell, Oxford, 2003.

[Lev94] Steven Levy. *Hackers. Heroes of the computer revolution*. Copyright dell'autore, 1994. Trad. it. di E. Guarnieri e L. Piercecchi (1996), Milano, ShaKe.

[Man03] Vincenzo Manca. *Metodi informazionali*. Bollati Boringhieri, Torino, 2003.

[Ras00] Jeff Raskin. *The human Interface*. Addison Wesley, Upper Saddle River, NJ, 2000.

[Sap87] Claudio Saporetti. *Elementare accadico*. Sellerio, Palermo, 1987.

[SC05] Francesco Sabatini and Vittorio Coletti. *il Sabatini Coletti. Dizionario essenziale della Lingua Italiana*. Rizzoli Larousse, Milano, 2005.

[Sin99] Simon Singh. *The Code Book. The Secret of History of Codes and Code-Breaking*. Copyright dell'autore, 1999. Trad. it. di S. Galli (2001), Rizzoli, Milano.

[SSP99] Ellen Siever, Stephen Spainhour, and Nathan Patwardhan. *Perl in a Nutshell*. O'Reilly, Sebastopol, 1999. Trad. it. di G. L. Nasoni (2000), Milano, Apogeo.

[Stu01] Michael Stutz. *The Linux Cookbook*. Copyright dell'autore, 2001. Trad. it. di SEI Servizi s.r.l. (2003), Mondadori, Milano.

[Tes59] Lucien Tesnière, editor. *Éléments de syntaxe structurale*. Editions Klincksieck, Paris, 1959. Trad. it. di G. Proverbio / A. Trocini Cerrina (2001), Rosemberg e Sellier, Torino.

[War04] Brian Ward. *How linux Works - What Every Super-User Should Know*. Copyright dell'autore, 2004. Trad. it. a cura di Publish Art - Pavia (2005), Mondadori, Milano.

[WCO00] Larry Wall, Tom Christiansen, and John Orwant. *Programming Perl, 3rd ed.* O'Reilly, Sebastopol, 2000.

[Wit53] Ludwig Wittgenstein, editor. *Philosophische Untersuchungen*. Blackwell, Oxford, 1953. Trad. it. di R. Piovesan / M. Trinchero (1999), Einaudi, Torino.