LINUX

Emacs

PROGRAMMAZIONE IN PERL

Generalità

Stringhe, liste e vettori associativi

Programmi elementari

Espressioni regolari

Intelligenza artificiale

BIOINFORMATICA

PROGRAMMAZIONE IN PERL

Corso di laurea in informatica a.a. 2001/02 Corso di Logica Numero 1 ◇ 8 Aprile 2002

Programmare in Perl

Il Perl è il linguaggio preferito dagli amministratori di sistema e una felice combinazione di concezioni della linguistica e di abili tecniche di programmazione; è usato nello sviluppo di software per l'Internet e in molte applicazioni scientifiche semplici o avanzate.

Il vantaggio a prima vista più evidente del Perl sul C è che non sono necessarie dichiarazioni per le variabili e che variabili di tipo diverso possono essere liberamente miste, ad esempio come componenti di una lista. Esistono alcune differenze più profonde: nel Perl una funzione può essere valore di una funzione, e il nome di una variabile (compreso il nome di una funzione) può essere usato come stringa. Queste caratteristiche sono molto potenti e fanno del Perl un linguaggio adatto alla programmazione funzionale e all'intelligenza artificiale.

Variabili nel Perl

Il Perl non richiede una dichiarazione per le variabili che distingue invece dall'uso dei simboli *$*, @ e *%* con cui i nomi delle variabili iniziano. Il Perl conosce essenzialmente tre tipi di variabili: *scalari* (riconoscibili dal *$* iniziale), *liste* (o *vettori* di scalari, riconoscibili dal @ iniziale) e *vettori associativi* (*hashes*, che iniziano con %) di scalari. Iniziano invece senza simboli speciali i nomi delle funzioni e dei riferimenti a files (*filehandles*) (variabili improprie). Quindi in Perl *$alfa*, *@alfa* e *%alfa* sono tre variabili diverse, indipendenti tra di loro. Esempio:

```
#! /usr/bin/perl -w

$a=7; @a=(8,$a,"Ciao");
%a=("Galli",27,"Motta",26);
print "$a\n"; print '$a\n';
for (@a) {print "$_"}
print "\n", $a{"Motta"}, "\n";
```

con output

```
7
$a\n8 7 Ciao
26
```

Nella prima riga riconosciamo la direttiva tipica degli script di shell che in questo caso significa che lo script viene eseguito dall'interprete */usr/bin/perl* con l'opzione *-w* (da *warning*, avvertimento) per chiedere di essere avvertiti se il programma contiene parti sospette.

Stringhe sono incluse tra virgolette oppure tra apostrofi; se sono incluse tra virgolette, le variabili scalari e i simboli per i caratteri speciali (ad es. *\n*) che appaiono nella stringa vengono sostituite dal loro valore, non invece se sono racchiuse tra apostrofi.

Il punto e virgola alla fine di un'istruzione può mancare se l'istruzione è seguita da una parentesi graffa chiusa.

A differenza dal C nel Perl ci sono due forme diverse per il *for*. In questo primo caso la variabile speciale *$_* percorre tutti gli elementi della lista *@a*; le parentesi graffe attorno a {*print "$_"*} sono necessarie nel Perl, nonostante che si tratti di una sola istruzione. Si vede anche che il *print*, come altre funzioni del Perl, in situazioni semplici non richiede le parentesi tonde attorno all'argomento. Bisogna però stare attenti anche con *print* perché ad esempio con *print (3-1)*7* si ottiene l'output *2*, perché viene prima eseguita l'espressione *print(3-1)*, seguita da un'inutile moltiplicazione per 7. Quindi qui bisogna scrivere *print((3-1)*7)*.

Parleremo più avanti delle variabili hash; nell'esempio si vede che *$a{"Motta"}* è il valore di *%a* nella voce *"Motta"*. Si nota con un po' di sorpresa forse che *$a{"Motta"}* non inizia con % ma con *$*; la ragione è che il valore della componente è uno scalare dal quale è del tutto indipendente la variabile *$a*. Una sintassi simile vale per i componenti di una lista.

Questa settimana

Input dalla tastiera

Il nostro primo programma è contenuto nel file **alfa**. Dopo il comando *alfa* dalla shell (dobbiamo rendere il file eseguibile con *chmod +x alfa*) ci viene chiesto il nome che possiamo inserire dalla tastiera; il programma ci saluta utilizzando il nome specificato.

```
#! /usr/bin/perl -w
# esempi/alfa
use strict 'subs';

print "Come ti chiami? ";
$nome=<stdin>; chop($nome);
print "Ciao, $nome!\n";
```

Se una riga contiene un # (che però non deve far parte di una stringa), il resto della riga (compreso il #) viene ignorato dall'interprete, con l'eccezione della direttiva *#!* (*shebang*) che viene vista prima dalla shell e le indica quale interprete (Perl, Shell, Python, ...) deve eseguire lo script.

L'istruzione *use strict 'subs';* controlla se il programma non contiene stringhe non contenute tra virgolette o apostrofi (*bar words*); in pratica avverte soprattutto quando si è dimenticato il simbolo *$* all'inizio del nome di una variabile scalare.

<stdin> legge una riga dallo standard input, compreso il carattere di invio finale che viene tolto con *chop*, una funzione che elimina l'ultimo carattere di una stringa.

Liste

Una variabile che denota una lista ha un nome che, come sappiamo, inizia con @. I componenti della lista devono essere scalari. Non esistono quindi liste di liste e simili strutture superiori nel Perl (a differenza ad esempio dal Lisp) che comunque possono, con un po' di fatica, essere simulate utilizzando *puntatori* (che in Perl si chiamano *riferimenti*), che tratteremo quando parleremo della programmazione funzionale.

Perciò, e questo è caratteristico per il Perl, la lista *(0,1,(2,3,4),5)* è semplicemente un modo più complicato di scrivere la lista *(0,1,2,3,4,5)*, e dopo

```
@a=(0,1,2); @b=(3,4,5);
@c=(@a,@b);
```

@c è uguale a *(0,1,2,3,4,5)*, ha quindi 6 elementi e non due. Perciò la seguente funzione può essere utilizzata per stampare le somme delle coppie successive di una lista.

```
sub sdue {my ($x,$y);
  while (($x,$y,@_)=@_) {print $x+$y,"\n"}}
```

Perché termina il ciclo del *while* in questo esempio? A un certo punto la lista *@_* rimasta sarà vuota (se all'inizio consisteva di un numero pari di argomenti), quindi l'espressione all'interno del *while* equivarrà a *($x,$y,@_)=()* (in Perl *()* denota la lista vuota), e quindi anche la parte sinistra in essa è vuota; la lista vuota in Perl però ha il valore booleano *falso*.

Il *k*-esimo elemento di una lista *@a* (cominciando a contare da 0) viene denotato con *$a[k]*. *@a[k]* invece ha un significato diverso ed è uguale alla lista il cui unico elemento è *$a[k]*. Infatti le parentesi quadre possono essere utilizzate per denotare segmenti di una lista: *@a[2..4]* denota la lista i cui elementi sono *$a[2]*, *$a[3]* e *$a[4]*, in *@a[2..4,6]* l'elemento *$a[6]* viene aggiunto alla fine del segmento, in *@a[6,2..4]* invece all'inizio.

(2..5) è la lista *(2,3,4,5)*, mentre *(2..5,0,1..3)* è uguale a *(2,3,4,5,0,1,2,3)*.

La funzione grep del Perl

La funzione *grep* viene usata come filtro per estrarre da una lista quelle componenti per cui un'espressione (nel formato che scegliamo il primo argomento di *grep*) è vera. La variabile speciale *$_* può essere usata in questa espressione e assume ogni volta il valore dell'elemento della lista che viene esaminato. Esempi:

```
sub pari {grep {$_%2==0} @_}
sub negativi {grep {$_<0} @_}
@a=pari(0,1,2,3,4,5,6,7,8);
for (@a) {print "$_ "}
print "\n"; # output 0 2 4 6 8
@a=negativi(0,2,-4,3,-7,-10,9);
for (@a) {print "$_ "}
print "\n"; # output -4 -7 -10
@a=("Ferrara","Firenze","Roma","Foggia");
@a=grep {!/^F/} @a;
for (@a) {print "$_ "}
print "\n"; # output Roma
```

Il cappuccio ^ denota l'inizio della riga, e quindi */^F/* è vera se la riga inizia con *F*; un punto esclamativo anteposto significa negazione.

Alcuni operatori per liste

L'istruzione *push(@a,@b)*aggiunge la lista *@b* alla fine della lista *@a*; lo stesso effetto si ottiene con *@a=(@a,@b)* che è però più lenta e probabilmente in molti casi implica che *@b* viene attaccata a una nuova copia di *@a*. La funzione restituisce come valore la nuova lunghezza di *@a*.

Per attaccare *@b* all'inizio di *@a* si usa invece *unshift(@a,@b)*; anche qui si potrebbe usare *@a=(@b,@a)* che è però anche qui meno efficiente. Anche questa funzione restituisce la nuova lunghezza di *@a*.

shift(@a) risp. *pop(@a)* tolgono il primo risp. l'ultimo elemento dalla lista *@a* e restituiscono questo elemento come valore. All'interno di una funzione si può omettere l'argomento; *shift* e *pop* operano allora sulla lista *@_* degli argomenti.

Una funzione più generale per la modifica di una lista è *splice*; l'istruzione *splice(@a,pos,elim,@b)* elimina, a partire dalla posizione *pos* un numero *elim* di elementi e li sostituisce con la lista *@b*. Esempi:

```
@a=(0,1,2,3,4,5,6,7,8);
splice(@a,0,3,"a","b","c");
print "@a\n"; # output a b c 3 4 5 6 7 8
splice(@a,4,3,"x","y");
print "@a\n"; # output a b c 3 x y 7 8
splice(@a,1,6);
print "@a\n"; # output a 8
```

reverse(@a) restituisce una copia invertita della lista *@a* (che non viene modificata dall'istruzione).

$#a è l'ultimo indice valido della lista *@a* e quindi uguale alla sua lunghezza meno uno.

Contesto scalare e contesto listale

In Perl avviene una specie di conversione automatica di liste in scalari e viceversa; se una variabile viene usata come scalare, si dice anche che viene usata in contesto scalare, e se viene usata come lista, si dice che viene usata in contesto listale. In verità è un argomento un po' intricato, perché si scopre che in contesto scalare le liste definite mediante una variabile si comportano diversamente da liste scritte direttamenta nella forma $(a_0, a_1, \ldots, a_n)$. Infatti in questa forma, in contesto scalare, la virgola ha un significato simile a quello dell'operatore virgola che tratteremo più avanti: se i componenti a_i sono espressioni che contengono istruzioni, queste vengono eseguite; il risultato (in contesto scalare) di tutta la lista è il valore dell'ultima componente.

Il valore scalare di una lista descritta da una variabile è invece la sua lunghezza.

Uno scalare in contesto listale diventa uguale alla lista il cui unico elemento è quello scalare. Esempi:

```
@a=7; # raro
print "$a[0]\n"; # output 7
@a=(8,2,4,7);
$a=@a; print "$a\n"; # output 4
$a=(8,2,4,7);
print "$a\n"; # output 7
@a=(3,4,9,1,5);
while (@a>2) {print shift @a} # output 349
print "\n";
```

Un esempio dell'uso dell'operatore virgola:

```
$a=4;
$a=($a=2*$a,$a--,$a+=3);
print "$a\n"; # output 10
```

Files e operatore <>

stdin, *stdout* e *stderr* sono i *filehandles* (un tipo improprio di variabile che corrisponde alle variabili del tipo *FILE** del C) che denotano standard input, standard output e standard error. Se *File* è un filehandle, *<File>* è il risultato della lettura di una riga dal file corrispondente a *File*. Esso contiene anche il carattere di invio alla fine di ogni riga.

Può accadere che l'ultima riga di un file non termini in un carattere invio, quindi se usiamo *chop* per togliere l'ultimo carattere, possiamo perdere un carattere. Nell'input da tastiera l'invio c'è sempre, quindi possiamo usare *chop*, come abbiamo visto a pagina 1; altrimenti si può usare la funzione *chomp* che toglie l'ultimo carattere da una stringa solo se è il carattere invio.

Per aprire un file si può usare *open* come nel seguente esempio (lettura di un file e stampa sullo schermo):

```
open(File,"lettera");
while (<File>) {print $_}
close(File);
```

oppure

```
$/=undef;
open(File,"lettera");
print <File>;
close(File);
```

Il separatore di fineriga (una stringa) è il valore della variabile speciale *$/* e può essere impostato dal programmatore; di default è uguale a "\n". Se lo rendiamo indefinito con *$/=undef* possiamo leggere tutto il file in un blocco solo come nel secondo esempio.

Per aprire il file *beta* in scrittura si può usare *open(File,">beta")* oppure, nelle più recenti versioni del Perl, *open(File,">","beta")*. La seconda versione può essere applicata anche a files il cui nome inizia con > (una cattiva idea comunque per le evidenti inferenze con il simbolo di redirezione > della shell). Similmente con *open(File,">>beta")* si apre un file per aggiungere un testo.

Il filehandle diventa allora il primo argomento di *print*, il testo da scrivere sul file è il secondo argomento, come nell'esempio che segue e nelle funzioni *files::scrivi* e *files::aggiungi*.

```
open(File,">beta");
print File "Ciao, Franco.\n";
close(File);
```

Abbiamo già osservato che la variabile che abbiamo chiamato *File* negli usi precedenti di *open* è impropria; una conseguenza è che questa variabile non può essere usata come variabile interna (mediante *my*) o locale di una funzione, in altre parole non può essere usata in funzioni annidate. Per questo usiamo i moduli *FileHandle* e *DirHandle* del Perl che permettono di utilizzare variabili scalari per riferirsi a un filehandle, come nel nostro modulo *files* che viene descritto a lato.

Funzioni del Perl

Una funzione del Perl ha il formato seguente

```
sub f {...}
```

dove al posto dei puntini stanno le istruzioni della funzione. Gli argomenti della funzione sono contenuti nella lista @_ a cui si riferisce in questo caso l'operatore *shift* che estrae da una lista il primo elemento. Le variabili interne della funzione vengono dichiarate tramite *my* oppure con *local* (che però ha un significato leggermente diverso da quello che uno si aspetta). La funzione può restituire un risultato mediante un *return*, altrimenti come risultato vale l'ultimo valore calcolato prima di uscire dalla funzione. Alcuni esempi tipici che illustrano soprattutto l'uso degli argomenti:

```
sub raddoppia {my $a=shift; $a+$a}
sub somma2 {my ($a,$b)=@_; $a+$b}
sub somma {my $s=0;
  for (@_) {$s+=$_} $s}
print raddoppia(4)," ";
print somma2(6,9)," ";
print somma(0,1,2,3,4),"\n";
```

con output *8 15 10*.

Alcuni operatori abbreviati che vengono usati in C e Perl:

```
$a+=$b ... $a=$a+$b
$a-=$b ... $a=$a-$b
$a*=$b ... $a=$a*$b
$a/=$b ... $a=$a/$b
$a++   ... $a=$a+1
$a-    ... $a=$a-1
```

Moduli

Le raccolte di funzioni in Perl si chiamano *moduli*; è molto semplice crearle. Assumiamo che vogliamo creare un modulo *matematica*; allora le funzioni di questo modulo vanno scritte in un file **matematica.pm** (quindi il nome del file è uguale al nome del modulo a cui viene aggiunta l'estensione **.pm** che sta per *Perl module*).

Il modulo può contenere anche istruzioni al di fuori delle sue funzioni; per rendere trasparenti i programmi queste istruzioni dovrebbero solo riguardare le variabili proprie del modulo.

Nell'utilizzo il modulo restituisce un valore che è uguale al valore dell'ultima istruzione in esso contenuto; se non ci sono altre istruzioni, essa può anche consistere di un *1;* all'inizio del file (che però non deve essere invalidata da un'altra istruzione che restituisce un valore falso).

Dopo di ciò altri moduli o il programma principale possono usare il modulo matematica con l'inclusione *use matematica;*.

Se alcuni moduli che si vogliono usare si trovano in cartelle α, β, γ che non sono tra quelle nelle quali il Perl cerca di default, si indica ciò con *use lib 'α', 'β', 'γ';*.

Il modulo files

```
1; # files.pm
use DirHandle; use FileHandle;

package files;

sub aggiungi {my ($a,$b)=@_;
  my $file=new FileHandle;
  open($file,">>$a"); print $file $b; close($file)}

sub leggi {local $/=undef; my $a;
  my $file=new FileHandle;
  if (open($file,shift))
  {$a=<$file>; close($file); $a} else {''}}

sub scrivi {my ($a,$b)=@_;
  my $file=new FileHandle;
  open($file,">$a"); print $file $b; close($file)}

sub catalogo {my $dir=new DirHandle;
  opendir($dir,shift);
  my @a=grep {!/^\./} readdir($dir);
  closedir($dir); @a}
```

In *catalogo* il significato di *opendir* e *closedir* è chiaro; *readdir* restituisce il catalogo della cartella associata con il *dirhandle $dir*, da cui, con un *grep* (pagina 2) il cui primo argomento è un espressione regolare, vengono estratti tutti quei nomi che non iniziano con un punto (cioè che non sono files o cartelle nascosti). Esempi d'uso:

```
use files;

print files::leggi("lettera");

for (files::catalogo('.')) {print "$_\n"}

$catalogo=join("\n",files::catalogo('/'));
files::scrivi("root",$catalogo);
```

Abbiamo usato la funzione *join* per unire con caratteri di nuova riga gli elementi della lista ottenuta con *files::catalogo* in un'unica stringa.

Vero e falso

La verità di un'espressione in Perl viene sempra valutata in contesto scalare. Gli unici valori scalari falsi sono la stringa vuota *“”* e il numero 0. La lista vuota in questo contesto assume il valore 0 ed è quindi anch'essa falsa. Lo stesso vale però per *(0)* e ogni lista scritta in forma esplicita il cui ultimo elemento è 0.

Attenzione: In Perl il numero 0 e la stringa *“0”* vengono identificati (si distinguono solo nell'uso), quindi anche la stringa *“0”* è falsa, benché non vuota. Le stringhe *“00”* e *“0.0”* sono invece vere.

Operatori logici del Perl

Per la congiunzione logica (AND) viene usato l'operatore *&&*, per la disgiunzione (OR) l'operatore | | . Come in molti altri linguaggi di programmazione questi operatori non sono simmetrici; infatti, se *A* è falso, in *A&&B* il valore del secondo operando *B* non viene più calcolato, e lo stesso vale per *A*| | *B* se *A* è vero.

In particolare *if (A&&B)* {α} è equivalente a *if (A)* {*if (B)* {α;}} e *if (A*| | *B)* {α;} è equivalente a *if (A)* {α} *else* {*if (B)* {α}}.

Il punto esclamativo viene usato per la negazione logica; anche *not* può essere usato a questo scopo.

and e *or* hanno una priorità minore di *&&* e | | ; tutti e quattro gli operatori restituiscono l'ultimo resultato calcolato, con qualche piccola sorpresa:

```
$a = 1 and 2 and 3; print “$a\n”;
   # output: 1 (sorpresa!)
$a = (1 and 2 and 3); print “$a\n”; # output: 3
$a = 1 && 2 && 3; print “$a\n”; # output: 3
$a = 0 or “” or 4; print “$a\n”;
   # output: 0 (sorpresa)
$a = (0 or “” or 4); print “$a\n”; # output: 0
$a = 0 || “” || 4; print “$a\n”; # output: 4
$a = 1 and 0 and 4; print “$a\n”;
   # output: 1 (sorpresa)
$a = (1 and 0 and 4); print “$a\n”; # output: 0
$a = 1 && 0 && 4; print “$a\n”; # output: 0
$a = 5 && 7 && “”; print “$a=\n”;
   # Nessun output!
```

Operatori di confronto

Il Perl distingue operatori di confronto tra stringhe e tra numeri. Per il confronto tra numeri si usano gli operatori ==, *!=*, <, >, <=, >=, per le stringhe invece *eq*, *ne*, *lt*, *le*, *gt* e *ge*. Si osservi che, mentre le stringhe *“1.3”*, *“1.30”* e *“13/10”* sono tutte distinte, le assegnazioni *$a=1.3*, *$b=1.30* e *$c=13/10* definiscono le tre variabili come numeri che hanno la stessa rappresentazione come stringhe, come si vede dai seguenti esempi:

```
$a=1.3; $b=1.30; $c=13/10;

print “ok 1\n” if $a==$b; # output: ok 1
print “ok 2\n” if $a==$c; # output: ok 2
print “ok 3\n” if $a eq $c; # output: ok 3
print “ne 4” if “1.3” ne “1.30”; # output: ne 4
```

Emacs

Emacs è un programma di scrittura non comune, da alcuni chiamato “il re degli editor”. Apparentemente spartano e complicato, è di una versatilità e configurabilità senza uguali. Oltre alle numerosissime funzioni già presenti, nuove funzioni possono essere aggiunte, programmate in *Elisp*, un dialetto del *Lisp*, forse il più potente linguaggio dell'intelligenza artificiale. Possono essere ridefiniti tutti i tasti per chiamare le funzioni disponibili. Si può fare in modo che un semplice tasto apra un certo file o una certa directory oppure che faccia in modo che venga compilato o eseguito un certo programma oppure stampato il file su cui si sta lavorando (infatti tutti i comandi della shell possono far parte delle funzioni di Emacs oppure essere eseguiti direttamente da una command line).

Oltra alla normale scrittura di testi, Emacs può essere combinato con altri programmi, ad esempio per leggere la posta elettronica. Come editor offre delle funzioni di ricerca molto potenti, funzioni di sostituzione, la possibilità di lavorare allo stesso tempo con moltissimi files, una comoda gestione delle directory.

È ideale per scrivere programmi, sorgenti HTML oppure testi in Latex, o anche solo per preparare i messaggi per la posta elettronica.

Seguono alcuni libri su Emacs.

Cameron/Rosenblatt: Learning GNU Emacs. O'Reilly 1996.

Glickstein: Writing GNU Emacs extensions. O'Reilly 1997.

Lewis/La Liberte/Stallman: GNU Emacs Lisp reference manual. Disponibile su Internet.

McEnaney: Programming techniques in Emacs. Prima 2000.

Comandi fondamentali di Emacs

Nella nostra interfaccia grafica (governata dal configurabilissimo window manager *fvwm2*) Emacs può essere invocato tramite *Alt-e*, dalla consolle con **emacs**. Abituarsi ad usare la tastiera, e fare a meno del menu (di cui serve solo la funzione *Select and Paste* sotto *Edit*).

I tasti funzione: **f1** per vedere l'elenco dei buffer attivi, **f2** e **f3** liberi, **f4** per uscire da Emacs, **f5** per tornare all'inizio del buffer e all'impostazione fondamentale, **f6** per andare alla fine del buffer, **f7** per avere tutta la finestra per il buffer di lavoro, **f8** per salvare il buffer nel file dello stesso nome, **f9** per poter inserire un comando, **f10** per le sostituzioni, **f11** e **f12** liberi.

Nell'impostazione di default invece di **f4** si usa **^xc**, e **^xs** invece di **f8**.

Comandi di movimento: **^a** per andare all'inizio della riga, **^e** per andare alla fine della riga, **^f** per andare avanti di un carattere, **^b** per andare indietro di un carattere, **^p** per andare una riga in su, **^n** per andare una riga in giù, **^v** per andare in giù di circa 3/4 di pagina, **^z** per andare in su di circa 3/4 di pagina, **^cv** per salvare il buffer e passare a un eventuale secondo buffer nella stessa finestra.

Funzioni di ricerca: **^s** ricerca in avanti, **^r** ricerca all'indietro. Molto comode e potenti.

Altri comandi: **^o** per aprire una riga (si usa spesso), **^k** per cancellare il resto della riga, **^y** per incollare, **^tu** per annullare l'effetto di un comando precedente, *TAB* o **^xf** per aprire un file.

I tre comandi più importanti: **^g** per uscire da un comando e, come abbiamo già visto, **^xc** o **f4** per uscire, **^xs** o **f8** per salvare il file.

Comandi di aiuto: **^th a** informazioni su funzioni il cui nome contiene una stringa, **^tw** informazioni su una funzione di un certo nome, **^th k** informazioni sull'effetto di un tasto.

L'inventore di Emacs è Richard Stallman, iniziatore della *GNU* e della *Free Software Foundation*.

Sulla foto lo si vede al centro durante una visita in Cina nel 2000.

Il tasti di Emacs

Riportiamo il contenuto del file **/home/varia/Emacs.el/tasti.el**.

```
; tasti.el

(global-unset-key "\C-c"
(global-unset-key "\C-i"
(global-unset-key "\C-j"
(global-unset-key "\C-t"
(global-unset-key "\C-y"
(global-unset-key "\C-z"

(define-key global-map "\C-c\C-v" 'cambia)
(define-key global-map "\C-h" 'backward-delete-char)
(define-key global-map "\C-i" 'find-file-other-window)

(define-key global-map "\C-t\C-c" 'fileretro)
(define-key global-map "\C-t\C-h" 'help-command)
(define-key global-map "\C-t\C-i" 'insert-file)
(define-key global-map "\C-t\C-k" 'kill-this-buffer)
(define-key global-map "\C-t\C-m" 'maiuscole)
(define-key global-map "\C-t\C-n" 'minuscole)
(define-key global-map "\C-t\C-r" 'cancella-da-qui)
(define-key global-map "\C-t\C-u" 'undo)
(define-key global-map "\C-t\C-v" 'describe-variable)
(define-key global-map "\C-t\C-w" 'describe-function)
(define-key global-map "\C-t\C-x" 'apropos-variable)

(define-key global-map "\C-y" 'yank-salva)
(define-key global-map "\C-z" 'scroll-down)

(define-key global-map [mouse-1] 'clickfile)
(define-key global-map [mouse-3] 'yank)

(define-key global-map [f1] 'elencobuffer)
(define-key global-map [f4] 'save-buffers-kill-emacs)

(define-key global-map [f5] 'fondamentale)
(define-key global-map [f6] 'end-of-buffer)
(define-key global-map [f7] 'delete-other-windows)
(define-key global-map [f8] 'salvabuffer)

(define-key global-map [f9] 'execute-extended-command)
(define-key global-map [f10] 'replace-string)

(define-key global-map [home] 'goto)
(define-key global-map [end] 'es-alfa)
(define-key global-map [prior] 'cambia-incolla-cambia)
(define-key global-map [next] 'prefisso)

(define-key global-map [print] 'stampa-selezione)
(define-key global-map [pause] 'split-window-vertically)

(define-key global-map [insert] 'stampa-buffer)
(define-key global-map [delete] 'make)
```

Istruzioni per alcune delle funzioni più usate: Quando la finestra è suddivisa, con *cambia* (**^cv**) si passa al buffer nell'altra parte; *fileretro* (**^tc**) permette di tornare al buffer precedente; *kill-this-buffer* (**^tk**) elimina il buffer (ma non il file dello stesso nome); *execute-extended-command* (**f9**) permette di chiamare una qualsiasi funzione di Emacs (anche una tra quelle create da noi); *cambia-incolla-cambia* (**Pag↑**) copia il testo selezionato sulla seconda parte della finestra; *ordina* (da chiamare mediante **f9**) ordina il file alfabeticamente; *elencobuffer* (**f1**) elenca i buffer aperti (si usa continuamente); *eval-current-buffer* (anche questo da **f9**) rende validi i cambiamenti in uno script di Emacs (ad esempio **.emacs**) senza che si debba chiudere e riavviare Emacs; *split-window-vertically* (**Pausa**) divide la finestra in due parti; *prefisso* (**Pag↓**) permette di inserire alcuni caratteri davanti al testo selezionato.

Quando avremo configurato la stampante, dovremmo poter utilizzare anche i comandi *stampa-buffer* (**Ins**) per stampare un buffer intero e *stampa-selezione* (**Stamp**) per stampare un testo precedentemente selezionato (con il mouse oppure mediante un **^k**).

Abituarsi a salvare spesso il testo con **f8**.

Per inserire commenti in uno script di Emacs si usa il punto e virgola (**;**) - il punto e virgola e tutto il resto della riga alla sua destra sono considerati commento. I comandi *es-alfa* (**Fine**), *make* (**Canc**) e *goto* (↖) sono spiegati nella pagina successiva.

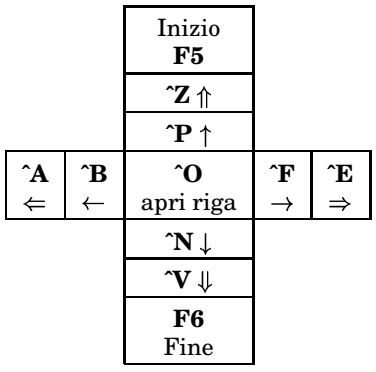
***

L'aiuto di Emacs

Emacs fornisce moltissime funzioni di aiuto, tra cui quelle utilizzate più frequentemente sono *describe-function* (**^tw**), *apropos-variable* (**^tx**), *describe-key* (**^th k**), *apropos-command* (**^th a**) e *describe-variable* (**^tv**).

Una documentazione online si trova nel World Wide Web sotto *http://www.geek-girl.com/emacs/emacs.html*. Per approfondire di più si può consultare il libro di Cameron/Rosenblatt nell'armadietto.

Comandi Emacs



elenco buffer aperti	**F1**	incolla	**^Y**
uscire	**F4** (**^XC**)	cancella carattere	**^D**
salvare	**F8** (**^XS**)	cancella ←	**^H**
comando	**F9**	cancella resto riga	**^K**
sostituzione	**F10**	cancella riga	**^AK**
apri file	**TAB** (**^XF**)	cancella da qui	**^TR**
inserisci file	**^TI**	cerca →	**^S**
togli buffer	**^TK**	cerca ←	**^R**
tutta la finestra	**F7**	termina comando	**^G**
cambia mezzafinestra	**^CV**	undo	**^TU**
apropos espressione	**^TH A**	maiuscole	**^TM**
apropos tasto	**^TH K**	minuscole	**^TN**
apropos variabile	**^TX**	compila	**Canc**
descrivi comando	**^TW**	esegui alfa	**Fine**
descrivi variabile	**^TV**	goto	↖

Per battere un carattere tabulatore bisogna usare **^Q TAB**.
Per scambiare due righe si utilizza **^XT**.

Il file .emacs

Quando Emacs viene avviato esegue prima i comandi in **.emacs**. Emacs viene programmato in *Elisp*, un linguaggio di programmazione molto simile al *Common Lisp*, nonostante alcune differenze più di forma e sintassi che di sostanza. Il file **.emacs** può a sua volta eseguire altri script in Elisp (mediante l'istruzione **load**), e infatti metteremo il grosso della configurazione di Emacs nei files contenuti nella directory **/home/varia/Emacs.el**. Il contenuto iniziale di **.emacs** è il seguente:

```
; .emacs
(setq diremacs "/home/varia/Emacs.el")
(setq load-path (append load-path (list diremacs)))

(let ((files (directory-files diremacs nil ".*\.el$" nil)))
  (while files (load (car files)) (setq files (cdr files))))

(load "fondamentale")
```

Ogni utente può comunque aggiungere al proprio **.emacs** ulteriori comandi.

La programmazione in *Lisp* non fa parte di questo corso, possiamo quindi solo spiegare alcuni degli aspetti più semplici. **setq x a** corrisponde all'incirca a un'assegnazione **x = a** in altri linguaggi, il **let** è una forma abbastanza complessa di assegnazione locale.

I comandi in questo file hanno questo significato: Prima viene introdotta la variabile *diremacs* come abbreviazione per la directory dove si trovano i nostri files per Emacs, e viene indicato a Emacs di cercare i files da caricare in quella directory. Il blocco **let** carica da essa tutti i files il cui nome termina con **.el** (c'è un'espressione regolare dopo il **nil**!), dopodiché per ultimo viene caricato il file **fondamentale**.

Scrivere programmi con Emacs

Emacs è un editor ideale per scrivere programmi in qualsiasi linguaggio di programmazione. La possibilità di combinare la scrittura dei files sorgenti con l'esecuzione del programma o di operazioni aggiuntive permette un lavoro efficiente e creativo.

Dobbiamo ancora spiegare l'uso dei comandi *es-alfa* (**Fine**), *make* (**Canc**) e *goto* (↖).

Premendo il tasto ↖ sulla riga di comando di Emacs appare *goto:* e si può indicare il numero della riga a cui si vuole andare. Questo è comodo nella programmazione, perché i messaggi d'errore spesso contengono la riga in cui il compilatore ritiene probabile che si trovi l'errore.

Con il tasto **Canc** si ottiene la compilazione di un programma in C, se il file su cui si sta lavorando sotto Emacs fa parte di un progetto. I messaggi di compilazione appaiono su una nuova pagina di Emacs. Con il tasto **Fine** viene invece eseguito il programma **alfa**, se la directory contiene un file eseguibile di questo nome.

I nostri programmi in Elisp per queste funzioni si trovano nella directory **/home/varia/Emacs.el** nei files **buffer.el**, **generali.el** e **input-output.el**:

```
(defun es-alfa() (interactive)
  (salvabuffer) (compile "alfa"))

(defun goto() (interactive)
  (goto-line (input-numero "goto: ")))

(defun make() (interactive)
  (salvabuffer) (compile "make"))

(defun input-numero (&optional domanda)
  (string-to-number (input-parola domanda)))

(defun input-parola (&optional domanda)
  (format "%s" (read-from-minibuffer (parola domanda) "")))

(defun parola (x) (if x x ""))

(defun salvabuffer() (interactive)
  (save-buffer)) (save-some-buffers)

(defun clickfile() (interactive)
  (if (string= mode-name "Dired by name") (dired-find-file-other-window))
  (if (string= mode-name "Buffer Menu") (Buffer-menu-other-window)))
```

Chi è interessato alla programmazione in Elisp può consultare il manuale in *http://www.delorie.com/gnu/docs/elisp-manual-20/elisp_toc.html*.

Le funzioni di Elisp

Una funzione (nell'esempio di due variabili) in Common Lisp o Elisp è della forma **(defun f (x y) ...)**, dove i puntini indicano una o più espressioni. Quando l'interprete esce dalla funzione, restituisce come risultato della funziona l'ultima espressione che ha elaborato. In Elisp quasi sempre dopo la lista degli argomenti (nell'esempio **(x y)**) segue **(interactive)** che permette la chiamata della funzione durante l'elaborazione di un testo con Emacs.

Le directory in Emacs

Oltre ai files normali anche le directory vengono visualizzate come buffer (così come risultati di compilazioni e altri messaggi). Per muoversi in una directory si possono usare più o meno gli stessi comandi come per i files; spesso il tasto *Ctrl* però non è necessario (quindi in una directory per passare alla riga successiva si possono usare sia **^n** che **n** da solo). Consultare eventualmente il libro di Cameron/Rosenblatt.

La nostra funzione *clickfile* - click col mouse sulla riga che contiene il nome di un buffer in una directory - chiama questo buffer.

---

ange ftp

Questa comodissima funzione di Emacs permette di leggere e scrivere un file su un altro computer come se il file si trovasse sul proprio PC. Il trasferimento dei dati avviene mediante **ftp**, ma nell'uso di Emacs non cambia niente. Solo nell'apertura di un file bisogna rispettare il metodo seguente. Assumiamo che il computer remoto sia *pc.remo.to* e che il mio nome utente su quel computer sia *rossi*. Allora con **^xf** (oppure *TAB* nella nostra configurazione) eseguo il comando di apertura di un file, di cui mi viene chiesto il nome. A questo punto inserisco

/rossi@pc.remoto:

(non dimenticare la barra iniziale e il doppio punto alla fine), mi viene chiesta la password (per il PC remoto) e mi trovo con Emacs nella mia directory di login sul computer remoto e posso continuare a lavorare allo stesso tempo sull'altro PC oppure su quello da cui sono partito.

map

***map** è una funzione importante del Perl e di tutti i linguaggi funzionali. Con essa da una lista $(a_1, \ldots, a_n)$ si ottiene la lista $f(a_1, \ldots, a_n)$, se f è una funzione a valori scalari anch'essa argomento di map. Il Perl prevede un'estensione molto utile al caso che la funzione f restituisca liste come valori; in tal caso, se ad esempio $f(1) = 9$, $f(2) = (21, 22, 23)$, $f(3) = (31, 32)$, $f(4) = 7$, da $(2, 4, 3, 1, 3)$ si ottiene $(21, 22, 23, 7, 31, 32, 9, 31, 32)$; il map del Perl effettua quindi un push per calcolare il risultato. Nella programmazione insiemistica useremo il map per la creazione della diagonale, ma ha tante altre applicazioni. La sintassi che usiamo è simile a quella di **grep** (pag. 2). Esempi:*

```
sub quadrato {my $a=shift; $a*$a}
@a=map {quadrato($_)} (0..8);
print "@a\n"; # output: 0 1 4 9 16 25 36 49 64


$pi=3.14159265358979323846; $pid180=$pi/180;
sub alfaecosalfa {my $alfa=shift; ($alfa,cos($alfa*$pid180))}
%tabellacoseni = map {alfaecosalfa($_)} (0..360);
# crea una tabella dei coseni
# gli angoli vengono indicati in gradi
print $tabellacoseni{30};
# output: 0.866025403784439
```

Vettori associativi

Vettori associativi (realizzati internamente mediante tabelle di *hash*) sono uno degli elementi linguistici più potenti del Perl. Un vettore associativo può essere considerato come un vettore i cui elementi vengono identificati da indici che possono essere scalari arbitrari invece che solo numeri $0, \ldots, n$.

Un vettore associativo è rappresentato da una lista con un numero pari di elementi che quindi possono essere immaginati come raggruppati in coppie di cui il primo elemento funge da indice (chiave), il secondo da componente corrispondente a quell'indice. L'elemento del vettore associativo *%a* corrispondente all'indice u è dato da $\$a\{u\}$. Esempi:

```
%ab=("Belluno",36, "Padova",212, "Rovigo",51, "Treviso",82,
   "Venezia",292, "Verona",255, "Vicenza",110);
print "$ab{'Padova'}\n"; # output 212
$ab{'Bologna'}=382;
```

Con *keys %a* si ottiene una lista che contiene (in ordine casuale, per il modo in cui vengono memorizzati i valori di una tabella hash) gli indici (o chiavi) del vettore associativo *%a*; *values %* è invece una lista dei componenti di *%a*. Esempi:

```
%stip=("Antoni",4100, "Berti",5200, "Mora",2300, "Rossi",3800);
print "$stip{'Rossi'}\n"; # output 3800
$s=0;
for (keys %stip) {$s+=$stip{$_}}
print "$s\n"; # output 15400
```

La funzione **keys**, che crea una lista degli indici di un vettore associativo, permette di percorrere gli elementi del vettore, ad esempio per eseguire un'operazione per ciascun elemento del vettore. Se il numero degli elementi è però molto grande, diciamo nell'ordine di alcune decine di migliaia, ciò può richiedere molta memoria per la creazione di questa lista.

Per vettori associativi grandi si usa perciò un'altra costruzione, in cui appare la funzione **each**, come nell'esempio che segue:

```
$s=0;
while (($x,$y)=each %stip) {$s+=$y}
print "$s\n";
```

Bisogna qui usare *while*, non *for*!

Questa settimana

Nascondere le variabili con my

Consideriamo le seguenti istruzioni:

```
$a=7;
sub quadrato {$a=shift; $a*$a}
print quadrato(10),"\n";
# output: 100
print "$a\n";
# output: 10
```

Vediamo che il valore della variabile esterna *$a* è stato modificato dalla chiamata della funzione *quadrato* che utilizza anch'essa la variabile *$a*. Probabilmente non avevamo questa intenzione e si è avuto questo effetto solo perché accidentalmente le due variabili avevano lo stesso nome. Per evitare queste collisioni dei nomi delle variabili il Perl usa la specifica **my**:

```
$a=7;
sub quadrato {my $a=shift; $a*$a}
print quadrato(10),"\n";
# output: 100
print "$a\n";
# output: 7
```

In questo modo la variabile all'interno di *quadrato* è diventata una variabile privata o locale di quella funzione; quando la funzione viene chiamata, alla *$a* interna viene assegnato un nuovo indirizzo in memoria, diverso da quello corrispondente alla variabile esterna. Consideriamo un altro esempio:

```
sub quadrato {$a=shift; $a*$a}
sub xpiu1alcubo {$a=shift;
   quadrato($a+1)*($a+1)}
print xpiu1alcubo(2);
# output: 36 invece di 27
```

Viene prima calcolato *quadrato(2+1)*, ponendo la variabile globale *$a* uguale all'argomento, cioè a 3, per cui il risultato finale è $9(3 + 1) = 36$.

I moduli CPAN

Già come linguaggio nella sua versione standard di una estrema ricchezza e versatilità, il Perl dispone di una vastissima raccolta di estensioni (moduli, cfr. pagina 3), il *Comprehensive Perl Archive Network* (*CPAN*), accessibile al sito *www.perl.com/CPAN*. Scegliendo *FetchFiles* su questo sito si viene indirizzati a un mirror (ad esempio a Roma) più vicino; adesso si può scegliere il link *recent modules* per vedere i moduli più recenti (ogni settimana arrivano fino a cento nuovi moduli!) oppure *modules* per la raccolta intera (scegliere poi *modules by category* oppure *modules by name*).

L'installazione di un modulo CPAN sotto Unix è semplice e sempre uguale. Installiamo ad esempio il modulo *Tk* necessario per l'interfaccia grafica tramite il *Perl/Tk*. Il modulo è attualmente disponibile, come file *Tk-800.024.tar.gz*, su CPAN (o su *felix.unife.it*), da cui con *gunzip* e *tar -xf* otteniamo una directory *Tk800.024*, in cui entriamo. Adesso bisogna dare, nell'ordine, i seguenti comandi, di cui l'ultimo (*make install*) come *root*:

```
perl Makefile.PL
make test
make
make install
```

Per usare questo modulo nei programmi dovremo inserire l'istruzione *use Tk;*.

Per ogni altro mudolo CPAN la procedura è esattamente la stessa; può però accadere che si rivela necessaria l'installazione di altri moduli prima di poter installare il modulo desiderato.

printf e sprintf

Queste funzioni si usano in modo molto simile alle analoghe funzioni del C. Il primo parametro di **printf** è sempre una stringa. Questa può contenere delle *istruzioni di formato* (dette anche *specifiche di conversione*) che iniziano con `%` e indicano il posto e il formato per la visualizzazione degli argomenti aggiuntivi. Ad esempio `%3d` tiene il posto per il valore di una variabile che verrà visualizzata come intero di tre cifre, mentre `-12.0f` indica una variabile di tipo **double** di al massimo 12 caratteri totali (compreso il punto decimale quindi), di cui 0 cifre dopo il punto decimale (che perciò non viene mostrato), allineati a sinistra a causa del - (l'allineamento di default avviene a destra). I formati più usati sono:

%d	intero	%c	carattere
%ld	intero lungo	%%	carattere %
%f	double	%s	stringa
%ud	intero senza segno		

$a=sprintf(...) fa in modo che *$a* diventi uguale all'output che si otterrebbe con **printf**.

```
$x=3; $y=7;
$a=sprintf("x = %d, y = %d\n",$x,$y);
print $a;
```

Istruzioni di controllo

Nelle alternative di un *if* si possono usare sia *else* che *elsif* (come abbreviazione di *else* {*if* ...}):

```
sub sgn {my $a=shift; if ($a<0) {-1}
  elsif ($a>0) {1} else {0}}
```

if può anche seguire un'istruzione o un blocco *do*:

α *if A* oppure *do* $\{\alpha; \beta; \gamma\}$ *if A*.

Il *goto* è un'istruzione di salto come nel seguente esempio:

```
$k=0;
ciclo: $k=$k+1; if ($k==7) {goto fine}
  # altre istruzioni
  goto ciclo;
fine: print "Fine\n";
```

Le etichette (in questo caso due) si riconoscono dal doppio punto finale.

In Perl esistono due forme alquanto diverse del *for*, da un lato l'analogo del *for* del C con una sintassi praticamente uguale, dall'altro il *for* che viene utilizzato per percorrere una lista.

Il *for* classico ha la seguente forma:

for (α;*A*;β) {γ}

equivalente a

α;
ciclo: if (A) {γ; β; *goto ciclo*}

oppure anche a

for (α;*A*;) {γ; β}

α, β e γ sono successioni di istruzioni separate da virgole; l'ordine in cui le istruzioni in ogni successione vengono eseguite non è prevedibile. Ciascuno dei tre campi può anche essere vuoto.

while (A) è equivalente a *for (;A;)* e *until (A)* equivalente a *for (;not A;)*.

Da un *for* (o *while* o *until*) si esce con *last* (o con *goto*), mentre *next* fa in modo che si torni ad eseguire il prossimo passaggio del ciclo, dopo esecuzione di β. Quindi

for (;A;β) {γ_1; *if (B) break;* γ_2;}

è equivalente a

```
ciclo: if (not A) {goto fuori}
  γ1; if (B) {goto fuori}
  γ2; β;
  goto ciclo;
fuori:
```

mentre

for (;;β) {γ_1; *if (B)* {*continue*} γ_2}

è equivalente a

ciclo: γ_1; *if (!B)* {γ_2} β; *goto ciclo;*

Il *last* e il *next* possono essere seguiti da un'etichetta, ad esempio *last alfa* significa che si esce dal ciclo *alfa*, mentre con *next alfa* viene eseguito il prossimo passaggio dello stesso ciclo. Esempio:

```
alfa: for $a (3..7) {for $b (0..20)
  {$x=$a*$b; next if $x%2 or $x<20;
  print "$x "; last alfa if $x>60}}
# output: 24 30 36 42 48 54 60 20 24 28 32 ...
```

Esistono anche le costruzioni *do* {...} *while A* e *do* {...} *until A* in cui le istruzioni nel blocco vengono eseguite sempre almeno una volta.

In *for $k (@a)* la variabile *$k* (locale per il *for*) percorre la lista *@a*. Se manca *$k* come in *for (@a)*, viene utilizzata la variabile speciale *@_*.

```
for (0..10) {last if $_>5; print $_}
  # output: 012345

for ($k=0;$k<=10;$k++)
  {next if $k<5; print $k}
  # output: 5678910

sub max {my $max=shift;
  for (@_)
  {$max=$_ if $_>$max} $max}
sub min {my $min=shift;
  for (@_)
  {$min=$_ if $_<$min} $min}
```

Concatenazione di stringhe

Il simbolo di concatenazione per stringhe è il punto: *$a."Roma".$b.$b* è la concatenazione delle stringhe *$a*, "*Roma*", *$b* e *$c*.

In questo caso avremmo anche potuto scrivere *"${a}Roma$b$c"*; il punto si usa ad esempio in *("(".f($a).")".riga(3)*, dove *f* e *riga* sono funzioni che restituiscono stringhe. Anche il (doppio) doppio punto può generare ambiguità, quindi si dovrà scrivere *$a.'::beta'* invece di *"$a::beta"*.

Talvolta si può utilizzare la funzione *sprintf* (fra poco).

I numeri di Fibonacci

La successione dei numeri di Fibonacci è definita dalla ricorrenza $F_0 = F_1 = 1, F_n = F_{n-1} + F_{n-2}$ per $n \geq 2$. Quindi si inizia con due volte 1, poi ogni termine è la somma dei due precedenti: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$. Un programma iterativo in Perl per calcolare l'n-esimo numero di Fibonacci:

```
sub fib1 {my $n=shift; my ($a,$b,$k);
  return 1 if $n<=1;
  for ($a=$b=1,$k=2;$k<=$n;$k++)
  {($a,$b)=($a+$b,$a)} $a}
```

Possiamo visualizzare i numeri di Fibonacci da F_0 a F_{20} e da F_{50} a F_{60} con la seguente funzione:

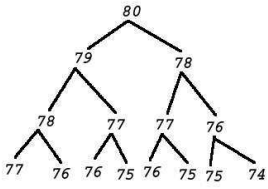
```
sub visfibonacci
  {for (0..20,50..60) {printf("%3d %-12.0f\n",$_,fib1($_))}}
```

La risposta è fulminea.

La definizione stessa dei numeri di Fibonacci è di natura ricorsiva, e quindi sembra naturale usare invece una funzione ricorsiva:

```
sub fib2 {my $n=shift;
  return 1 if $n<=1;
  fib2($n-1)+fib2($n-2)}
```

Se però adesso nella funzione **fibonacci** sostituiamo *fib1* con *fib2*, ci accorgiamo che il programma si blocca dopo la serie dei primi 20 numeri di Fibonacci, cioè che anche i velocissimi Pentium non sembrano in grado di calcolare F_{50}. Infatti qui incontriamo il fenomeno di una ricorsione con *sovrapposizione dei rami*, cioè una ricorsione in cui le operazioni chiamate ricorsivamente vengono eseguite molte volte. Questo fenomeno è frequente nella ricorsione doppia e diventa chiaro se osserviamo l'illustrazione a lato che mostra lo schema secondo il quale avviene ad esempio il calcolo di F_{80}. Si vede che F_{78} viene calcolato due volte, F_{77} tre volte, F_{76} cinque volte, ecc. (si ha l'impressione che riappaia la successione di Fibonacci e infatti è così, quindi un numero esorbitante di ripetizioni impedisce di completare la ricorsione). È noto che F_n è approssimativamente (con un errore minore di 0.5) uguale a $\frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{n+1}$ e quindi si vede che questo algoritmo è di complessità esponenziale.



Il metodo del sistema di primo ordine

In analisi si impara che un'equazione differenziale di secondo ordine può essere ricondotta a un sistema di due equazioni di primo ordine. In modo simile possiamo trasformare la ricorrenza di Fibonacci in una ricorrenza di primo ordine in due dimensioni. Ponendo $x_n := F_n, y_n := F_{n-1}$ otteniamo il sistema

$$x_{n+1} = x_n + y_n$$
$$y_{n+1} = x_n$$

per $n \geq 0$ con le condizioni iniziali $x_0 = 1, y_0 = 0$ (si vede subito che ponendo $F_{-1} = 0$ la relazione $F_{n+1} = F_n + F_{n-1}$ vale per $n \geq 1$). Per applicare questo algoritmo dobbiamo però in ogni passo generare due valori, avremmo quindi bisogno di una funzione che restituisce due valori numerici. Ciò in Perl, dove una funzione può restituire come risultato una lista, è molto facile:

```
sub fib3 {my $n=shift; my ($x,$y);
  return (1,0) if $n==0;
  ($x,$y)=fib3($n-1); ($x+$y,$x)}
```

Per la visualizzazione dobbiamo ancora modificare la funzione **fibonacci** nel modo seguente:

```
sub visfibonacci {my ($x,$y);
  for (0..20,50..60)
  {($x,$y)=fib3($_);
  printf("%3d %-12.0f\n",$_,$x)}}
```

Lo schema di Horner

Sia dato un polinomio $f = a_0x^n + a_1x^{n-1} + \cdots + a_n \in A[x]$, dove A è un qualsiasi anello commutativo. Per $\alpha \in A$ vogliamo calcolare $f(\alpha)$.

Sia ad esempio $f = 3x^4 + 5x^3 + 6x^2 + 8x + 17$. Poniamo

$$\begin{array}{lcll} b_0 & = & 3 & \\ b_1 & = & b_0\alpha + 5 & = 3\alpha + 5 \\ b_2 & = & b_1\alpha + 6 & = 3\alpha^2 + 5\alpha + 6 \\ b_3 & = & b_2\alpha + 8 & = 3\alpha^3 + 5\alpha^2 + 6\alpha + 8 \\ b_4 & = & b_3\alpha + 17 & = 3\alpha^4 + 5\alpha^3 + 6\alpha^2 + 8\alpha + 17 \end{array}$$

e vediamo che $b_4 = f(\alpha)$. Lo stesso si può fare nel caso generale:

$$\begin{array}{lcl} b_0 & = & a_0 \\ b_1 & = & b_0\alpha + a_1 \\ \cdots & & \\ b_k & = & b_{k-1}\alpha + a_k \\ \cdots & & \\ b_n & = & b_{n-1}\alpha + a_n \end{array}$$

con $b_n = f(\alpha)$, come dimostriamo adesso. Consideriamo il polinomio

$$g := b_0x^{n-1} + b_1x^{n-2} + \cdots + b_{n-1}.$$

Allora, usando che $\alpha b_k = b_{k+1} - a_{k+1}$ per $k = 0, \ldots, n-1$, abbiamo

$$\begin{array}{rl} \alpha g = & \alpha b_0x^{n-1} + \alpha b_1x^{n-2} + \cdots + \alpha b_{n-1} = \\ = & (b_1 - a_1)x^{n-1} + (b_2 - a_2)x^{n-2} + \cdots + \\ & +(b_{n-1} - a_{n-1})x + b_n - a_n = \\ = & (b_1x^{n-1} + b_2x^{n-2} + \cdots + b_{n-1}x + b_n) - \\ & -(a_1x^{n-1} + a_2x^{n-2} + \ldots a_{n-1}x + a_n) = \\ = & x(g - b_0x^{n-1}) + b_n - (f - a_0x^n) = \\ = & xg - b_0x^n + b_n - f + a_0x^n = xg + b_n - f \end{array}$$

quindi

$$f = (x - \alpha)g + b_n,$$

e ciò implica

$$f(\alpha) = b_n.$$

$b_0, \ldots, b_{n-1}$ sono perciò i coefficienti del quoziente nella divisione con resto di f per $x - \alpha$, mentre b_n è il resto, uguale a $f(\alpha)$.

Questo algoritmo si chiama *schema di Horner* o *schema di Ruffini* ed è molto più veloce del calcolo separato delle potenze di α (tranne nel caso che il polinomio consista di una sola o di pochissime potenze, in cui si userà invece l'algoritmo del contadino russo). Quando serve solo il valore $f(\alpha)$, in un programma in Perl si può usare la stessa variabile per tutti i b_k:

```
sub horner # y=horner($x,@a)
  {my ($x,@a)=@_; my $b=shift @a;
  for (@a) {$b=$b*$x+$_} $b}
```

Una frequente applicazione dello schemo di Horner è il calcolo del valore corrispondente a una rappresentazione binaria o esadecimale. Infatti

$(1,0,0,1,1,0,1,1,1)_2 = \text{horner}(2,1,0,0,1,1,0,1,1,1)$

$(A,F,7,3,0,5,E)_{16} = \text{horner}(16,10,15,7,3,0,5,14).$

La moltiplicazione russa

Esiste un algoritmo leggendario del contadino russo per la moltiplicazione di due numeri, uno dei quali deve essere un intero positivo. Assumiamo che vogliamo calcolare $86x$, dove x è un numero reale:

$$86 \cdot x \longrightarrow 43 \cdot 2x \xrightarrow{2x} 42 \cdot 2x \longrightarrow 21 \cdot 4x \xrightarrow{4x} 20 \cdot 4x \longrightarrow 10 \cdot 8x \longrightarrow 5 \cdot 16x \xrightarrow{16x} 4 \cdot 16x \longrightarrow 2 \cdot 32x \longrightarrow 1 \cdot 64x \xrightarrow{64x} \longrightarrow 0 \cdot 128x \longrightarrow \bullet$$

Lo schema va interpretato così: p sarà il risultato della moltiplicazione; all'inizio poniamo $p = 0$. $86x = 43 \cdot 2x$, quindi possiamo sostituire x con $2x$ e dimezzare il primo fattore che così diventa dispari. Con il nuovo x abbiamo $43x = x + 42x$. Aggiungiamo x a p e procediamo con $42x = 21 \cdot 2x$ come nel primo passo: sostituiamo quindi x con $2x$ e dimezziamo il primo fattore. E così via: Quando arriviamo a un primo fattore dispari, sommiamo l'ultimo valore di x a p e diminuiamo il primo fattore di uno che così diventa pari. Quando il primo fattore è pari, sostituiamo x con $2x$ e dimezziamo il primo fattore. Ci fermiamo quando il primo fattore è uguale a 0.

Altrettanto semplice e importante è la formulazione puramente matematico-ricorsiva dell'algoritmo - scriviamo f per la funzione definita da $f(n, x) = nx$:

$$f(n,x) = \begin{cases} 0 & \text{se } n = 0 \\ f(\frac{n}{2}, 2x) & \text{se } n \text{ è pari} \\ x + f(n-1, x) & \text{se } n \text{ è dispari} \end{cases}$$

Diamo una versione ricorsiva e una versione iterativa in Perl per questo algoritmo:

```
sub mrussa # $p=mrussa($n,$x)
  {my ($n,$x)=@_;
  return $x+mrussa($n-1,$x) if $n%2;
  return mrussa($n/2,$x+$x) if $n>0; 0}
sub mrussa # $p=mrussa($n,$x)
  {my ($n,$x)=@_; my $p;
  for ($p=0;$n;) {if ($n%2) {$p+=$x; $n--}
  else {$x+=$x; $n/=2}} $p}
```

Come trovare la rappresentazione binaria

Non è difficile convincersi che ogni numero naturale $n > 0$ possiede una rappresentazione binaria, cioè della forma

$$n = a_k 2^k + a_{k-1} 2^{k-1} + \cdots + a_2 2^2 + a_1 2 + a_0 \qquad (*)$$

con coefficienti (o cifre) $a_i \in \{0, 1\}$ e $a_k = 1$ univocamente determinati. Sia $\text{rapp2}(n) = (a_k, \ldots, a_0)$ la lista i cui elementi sono queste cifre. Dalla rappresentazione (*) si deduce la seguente relazione ricorsiva, in cui utilizziamo il meccanismo della fusione di liste del Perl (cfr. pag. 2):

$$\text{rapp2}(n) = \begin{cases} (1) & \text{se } n = 1 \\ (\text{rapp2}(\frac{n}{2}), 0) & \text{se } n \text{ è pari} \\ (\text{rapp2}(\frac{n-1}{2}), 1) & \text{se } n \text{ è dispari} \end{cases}$$

Questa relazione può essere tradotta immediatamente in un programma in Perl:

```
sub rapp2 # @cifre=rapp2($n)
  {my $n=shift; return (1) if $n==1;
  return (rapp2($n/2),0) if $n%2==0;
  (rapp2(($n-1)/2),1)}
```

La potenza russa

Per il calcolo dipotenze con esponenti reali arbitrari si può usare la funzione **pow** del Perl: la terza radice si ottiene ad esempio con *pow(x,1/3)*. Come molte altre funzioni matematiche (ad esempio *sin* e *cos*) anche *pow* richiede l'inclusione *use POSIX;* all'inizio del file.

Per esponenti interi positivi si può usare invece un altro metodo del contadino russo. Assumiamo di voler elevare x alla 937-esima potenza.

$$x^{937} \xrightarrow{x} x^{936} \longrightarrow (x^2)^{468} \longrightarrow (x^4)^{234} \longrightarrow (x^8)^{117} \xrightarrow{x^8} (x^8)^{116} \longrightarrow (x^{16})^{58} \longrightarrow (x^{32})^{29} \xrightarrow{x^{32}} (x^{32})^{28} \longrightarrow (x^{64})^{14} \longrightarrow (x^{128})^7 \xrightarrow{x^{128}} (x^{128})^6 \longrightarrow (x^{256})^3 \xrightarrow{x^{256}} (x^{256})^2 \longrightarrow (x^{512})^1 \xrightarrow{x^{512}} (x^{512})^0 \longrightarrow \bullet$$

Lo schema va interpretato così: $x^{937} = x \cdot x^{936}$. Ci ricordiamo il fattore x. $x^{936} = (x^2)^{468}$, quindi sostituendo x con x^2 dobbiamo solo fare la 468-esima potenza del nuovo x oppure la 234-esima se sostituiamo ancora x con il suo quadrato. Quando arriviamo a un esponente dispari, moltiplichiamo l'ultimo valore di x con il fattore fino a quel punto memorizzato e possiamo quindi diminuire l'esponente di uno, ottenendo di nuovo un esponente pari. E così via. In ogni passaggio dobbiamo solo o formare un quadrato e dimezzare l'esponente oppure moltiplicare il fattore con il valore attuale di x e diminuire l'esponente di uno.

Anche in questo caso invece del ragionamente iterativo si può riformulare il problema in modo matematico-ricorsivo - stavolta $f(x, n) = x^n$:

$$f(x,n) = \begin{cases} 1 & \text{se } n = 0 \\ f(x^2, \frac{n}{2}) & \text{se } n \text{ è pari} \\ x f(x, n-1) & \text{se } n \text{ è dispari} \end{cases}$$

È facile tradurre queste idee in un programma ricorsivo o iterativo in Perl:

```
sub potenza # $p=potenza($x,$n)
  {my ($x,$n)=@_;
  return $x*potenza($x,$n-1) if $n%2;
  return potenza($x*$x,$n/2) if $n>0; 1}
sub potenza # $p=potenza($x,$n)
  {my ($x,$n)=@_; my $p;
  for ($p=1;$n;) {if ($n%2) {$p*=$x; $n--}
  else {$x*=$x; $n/=2}} $p}
```

Lo schema di Horner ricorsivo

Lo schema di Horner per il calcolo del valore $f(\alpha)$ di un polinomio $f = a_0 x^n + a_1 x^{n-1} + \cdots + a_n$ permette una elegante versione ricorsiva. Infatti, se denotiamo quel valore con $\text{val}(\alpha, a_0, a_1, \ldots, a_n)$, allora abbiamo la relazione

$$\text{val}(\alpha, a_0, a_1, \ldots, a_n) = \alpha \cdot \text{val}(\alpha, a_0, a_1, \ldots, a_{n-1}) + a_n$$

come si vede da

$$a_0\alpha^n + a_1\alpha^{n-1} + \cdots + a_{n-1}\alpha + a_n = \\ = \alpha \cdot (a_0\alpha^{n-1} + a_1\alpha^{n-2} + \cdots + a_{n-1}) + a_n$$

con la condizione iniziale $\text{val}(\alpha) = 0$. Ciò può essere tradotto in un programma in Perl:

```
sub val # y=val($x,a0,...,a1)
  {my $x=shift; return 0 if not @_;
  my $a=pop; $a+$x*val($x,@_)}
```

PROGRAMMAZIONE IN PERL

Corso di laurea in informatica a.a. 2001/02 Corso di Logica Numero 3 ◇ 22 Aprile 2002

Espressioni regolari

Un'espressione regolare è una formula che descrive un insieme di parole. Questo importante concetto dell'informatica teorica è entrato in molti linguaggi predecessori del Perl, soprattutto nel mondo Unix, ai quali il Perl si è sostituito aggiungendo la versatilità e la potenza di un linguaggio ad altissimo livello. Il Perl è nato come "Practical Extraction and Report Language" (contiene infatti istruzioni per creare un output formattato adatto per rapporti sullo schermo o dattiloscritti, piuttosto valide, però poco usate quando si utilizza un linguaggio di composizione tipografico come il LaTEX e rimpiazzabili da apposite procedure in Perl che possono essere create per un'applicazione concreta) e le espressioni regolari rimangono uno degli strumenti più frequentemente utilizzati dal programmatore in Perl nel lavoro quotidiano; spesso inserendo o togliendo pochi caratteri in un'istruzione è possibile effettuare una modifica che in C poteva richiedere la completa riscrittura di una parte consistente del programma.

Espressioni regolari nel Perl

Una parola che non contiene caratteri speciali, come espressione regolare corrisponde all'insieme di tutte le parole che la contengono (ad esempio *alfa* è contenuta in *alfabeto* e *stalfano*, ma non in *stalfino*). *^alfa* indica invece che *alfa* si deve trovare all'inizio della riga, *alfa$* che si deve trovare alla fine. È come se ^ e *$* fossero due lettere invisibili che denotano inizio e fine della riga. Il carattere spazio viene trattato come gli altri, quindi con *a lfa* si trova *kappa lfa*, ma non *alfabeto*.

Il punto . denota un carattere qualsiasi, ma un asterisco * non può essere usato da solo, ma indica una ripetizione arbitraria (anche vuota) del carattere che lo precede. Quindi *a** sta per le parole *a*, *aa*, *aaa*, ... e anche per la parola vuota. Per quest'ultima ragione *alfa*ino* trova *alfino*. Per escludere la parola vuota si usa + al posto dell'asterisco. Ad esempio + indica almeno uno e possibilmente più spazi vuoti. Questa espressione regolare viene spesso usata per separare le parti di una riga. Per esempio **, +* trova *alfa , beta*, ma non *alfa ,beta*. Il punto interrogativo *?* dopo un carattere indica che quel carattere può apparire una volta oppure mancare, quindi *alfa?ino* trova *alfino* e *alfaino*, ma non *alfaaino*.

Le parentesi quadre vengono utilizzate per indicare insiemi di caratteri oppure il complemento di un tale insieme. *[aeiou]* denota le vocali minuscole e *[^aeiou]* tutti i caratteri che non siano vocali minuscole. È il cappuccio ^ che indica il complemento. Quindi *r[aeio]ma* trova *rima* e *romano*, mentre *[Rr][aeio]ma* trova anche *Roma*. Si possono anche usare trattini per indicare insiemi di caratteri successivi naturali, ad esempio *[a–zP]* è l'insieme di tutte le lettere minuscole dell'alfabeto comune insieme alla *P* maiuscola, e *[A–Za–z0–9]* sono i cosiddetti caratteri alfanumerici, cioè le lettere dell'alfabeto insieme alle cifre. Per questo insieme si può usare l'abbreviazione $\backslash w$, per il suo complemento $\backslash W$.

La barra verticale | tra due espressioni regolari indica che almeno una delle due deve essere soddisfatta. Si possono usare le parentesi rotonde: *a| b| c* è la stessa cosa come *[abc]*, *r(oma| ume)no* trova *romano* e *rumeno*.

Per indicare i caratteri speciali ., *, ^ ecc. bisogna anteporgli \, ad esempio \. per indicare veramente un punto e non un carattere qualsiasi. Vedremo nelle pagine successive il significato preciso dei caratteri speciali nelle espressioni regolari del Perl.

Nelle varianti più semplici le espressioni regolari possono essere utilizzate anche in alcuni comandi della shell di Unix, in particolari nell'istruzione **grep**.

Questa settimana

I modificatori /*m* ed /*s*

Il punto *(.)* nelle espressioni regolari sta per un carattere qualsiasi diverso dal carattere di nuova riga. Aggiungendo *s* alla fine dell'istruzione di matching, si ottiene che . comprende anche il carattere di nuova riga; si farà così quando con *(.*)* si vuole denotare una successione arbitraria di caratteri che si può estendere anche su più righe.

Come visto a sinistra, ^ e *$* vengono utilizzati per indicare l'inizio e la fine della stringa. Questo significato cambia se all'istruzione di matching aggiungiamo *m*: in questo caso ^ indica anche l'inizio di una riga (cioè l'inizio della stringa oppure una posizione preceduta da un carattere di nuova riga), e similmente *$* indica anche la fine di una riga (cioè la fine della stringa oppure una posizione a cui segue un carattere di nuova riga).

Se, mentre si usa il modificatore /*m*, ci si vuole riferire all'inizio della stringa, si utilizza $\backslash A$ (che senza /*m* ha lo stesso significato di ^); mentre similmente $\backslash z$ indica la fine della stringa anche in presenza di /*m* (ed è invece equivalente a *$* in assenza di /*m*).

Siccome stringhe prelevate da un file spesso contengono un ultimo carattere di nuova riga, esiste un altro simbolo $\backslash Z$ che corrisponde alla posizione precedente a questo ultimo carattere di nuova riga, quando presente, altrimenti alla vera fine della stringa.

Questi simboli perdono il loro significato all'interno di parentesi quadre.

I metacaratteri

I seguenti caratteri hanno un significato speciale nelle espressioni regolari: \, | , *(,)*, *[,]*, {, }, ^, *$*, *, +, *?*, . e, all'interno di *[ ]*, anche -. Per privare questi caratteri del loro significato speciale, è sufficiente preporgli un \.

- \ viene usato per dare a un carattere il suo significato normale.
- | indica scelte alternative, che vengono esaminate da sinistra a destra.
- () Le parentesi rotonde vengono usate in più modi. Possono servire a racchiudere semplicemente un espressione per limitare il raggio d'azione di un'alternativa, per distinguere ad esempio *a(u| v)* da *au| v*, oppure per catturare una parte da usare ancora. Altri usi delle parentesi rotonde vengono descritti separatamente.
- [] Le parentesi quadre racchiudono insiemi di caratteri oppure il loro complemento (se subito dopo la parentesi iniziale *[* si trova un ^ che in questo contesto non ha più il significato di inizio di parola che ha al di fuori delle parentesi quadre).
- {} Le parentesi graffe permettono la quantificazione delle ripetizioni: *a{3}* significa *aaa*, *a{2,5}* comprende *aa*, *aaa*, *aaaa* ed *aaaaa*.
- ^ Questo carattere indica l'inizio di parola (oppure, quando è presente il modificatore */m*, anche l'inizio di una riga, cfr. pag. 11), quando non si trova all'interno delle parentesi quadre, dove, se si trova all'inizio, significa la formazione del complemento.
- $ indica la fine della parola o della riga a seconda che manchi o sia presente l'operatore */m*.
- * L'asterisco è un quantificatore e indica che il simbolo precedente può essere ripetuto un numero arbitrario di volte (o anche mancare). Un *?* altera il comportamento di * come vedremo.
- + Ha lo stesso significato di *, tranne che il simbolo deve apparire almeno una volta. Un *?* altera il comportamento di +.
- ? *a?* significa che *a* può apparire oppure no, con preferenza per il primo caso; *a??* invece con preferenza per il secondo caso. **?* significa che viene scelta la corrispondenza più breve possibile (altrimenti il Perl sceglie la più lunga); un discorso analogo vale per *+?*.
- . sta per un singolo carattere che deve essere diverso dal carattere di nuova riga se non è presente il modificatore */s* (pag. 11).
- - all'interno di parentesi quadre può essere usato per denotare un insieme di caratteri attigui. Per avere un semplice - all'interno delle parentesi graffe si deve usare \-.

Gli operatori *m* ed *s*

m significa *matching* (corrispondere, accordarsi), *s* sostituzione. Il primo operatore viene usato per trovare parti di una parola che corrispondono a un'espressione regolare, il secondo per sostituire parti di una parola. In questo articolo impariamo soltanto la sintassi fondamentale di questi operatori e usiamo quindi negli esempi espressioni regolari molto semplici, riconducibili a quelle trattate a pag. 11. Iniziamo con l'operatore *m* e consideriamo le istruzioni

```
$a="fenilalanina e tirosina sono aminoacidi";

if ($a=~/nina/) {print "nina\n"} # output: nina
if ($a=~/[fct]iro/) {print "trovato\n"} # output: trovato
if ($a=~/([fct]iro)/) {print "$1\n"} # output: tiro
if ($a=~/([fct]i)ro/) {print "$1\n"} # output: ti
if ($a=~/a([a-z])i([a-z])o/) {print "$1-$2\n"} # output: m-n
```

Il significato di *$1* e *$2* è spiegato a pag. 14. Esempi di sostituzione:

```
$a="andare area dormire stare"; $b=$a;

$a=~s/are/ava/; print "$a\n";
# output: andava area dormire stare

$a=$b;

$a=~s/([ai])re( |$)/${1}va$2/g; print "$a\n";
# output: andava area dormiva stava

$a="ababacodelbababbo"; $b=$a;
$a=~s/aba/ibi/g; print "$a\n";
# output: ibibacodelbibibbo

$a=$b;
$a=~s/aba/iba/g; print "$a\n";
# output: ibabacodelbibabbo
# Si vede che l'elaborazione continua dalla posizione già raggiunta.

$a=~s/[aeiou]//g; print "$a\n";
# output: bbcdlbbbb

$a=~s/b+/b/g; print "$a\n";
# output: bcdlb
```

Si noti bene che in $s/\alpha/\beta/$ la prima parte (α) è un'espressione regolare, mentre β è una normale stringa (tranne nel caso in cui si utilizza il modificatore */e*). In entrambe le parti le variabili (ad esempio un *$a*) vengono espanse come se fossero contenute tra virgolette.

I metasimboli

Abbiamo già spiegato il significato di *\A*, *\z* e *\Z*. I simboli *\0*, *\n* e *\t* vengono usati come in C e indicano il carattere ASCII 0, il carattere di nuova riga e il tabulatore. Esiste numerosi altri metasimboli, di cui elenchiamo quelli più comuni, sufficienti in quasi tutte le applicazioni pratiche:

\w	carattere alfanumerico, equivalente a [A-Za-z0-9_]
\W	non carattere alfanumerico
\d	[0-9] – il *d* deriva da *digit* (cifra)
\D	[^0-9]
\s	spazio bianco, normalmente [\t\n\r\f]
\S	non spazio bianco

I modificatori */i* ed */o*

Aggiungendo *i* all'istruzione di matching, nell'espressione regolare non viene distinto tra maiuscole e minuscole.

Aggiungendo *o* (da *optimize*), eventuali variabili contenute nell'espressione regolare vengono espanse una volta sola e quindi rimangono uguali anche quando durante l'esecuzione dell'istruzione di matching formalmente dovrebbero venir modificate. Esempio:

```
$cifra="\\d";
$naturale="[+]?$cifra+";
$intero="[+\\-]?$cifra+";

$a="88 alfa -603 beta 13";
@a=$a=~/$intero/go; for (@a) {print "$_"}
# output: 88 -603 13
```

Il modificatore /g

Aggiungendo un *g* (da *global*) all'istruzione di matching, nel caso di una sostituzione si ottiene che le sostituzioni richieste vengono effettuate (in successione) tutte le volte che è possibile.

$m/\alpha/g$ invece in contesto listale restituisce una lista di tutte le corrispondenze trovate, se α non contiene parentesi di cattura; altrimenti solo le parti catturate. Esempio:

```
$a="E' tornata nell'Alto Volta.";
@lista=$a=~/t[aeiou]/g;
for (@lista) {print "$_ "} # output: to ta to ta
```

Nell'esempio che segue, più tipico per l'uso del Perl, estraiamo da una stringa (che potrebbe essere stata prelevata da un file) i valori di certe variabili creando un vettore associativo:

```
$a="a=6, b=200, at=130";
%valori=$a=~/([a-z]+)\s*=\s*([0-9]+)/g;
for (keys %valori) {print "$_ $valori{$_} "}
# output: a 6 b 200 at 130
```

Il modificatore /e

Questo modificatore potente permette di inserire nelle istruzioni di sostituzione espressioni che contengono codici di Perl; la *e* deriva da *evaluation*. Più precisamente $s/\alpha/\beta/e$ significa che α viene sostituito dal valore calcolato di β. Quest'ultima espressione può contenere anche istruzioni che eseguono operazioni che non riguardano la sola sostituzione; il modificatore /*e* è quindi uno strumento estremamente versatile. Se la *e* viene ripetuta (una o più volte), anche la valutazione avviene un numero corrispondente di volte. Esempi:

```
$a="8 2 3 4 6 7";
$a=~s/(\d+) +(\d+)/$1*$2/eg;
print $a; # output 16 12 42

$a=3; $b="cerchi";
$c='$a $b';
$c=~s/(\$\w+)/$1/eeg; print $c;
# output: 3 cerchi

sub f {my $a=shift; "$a+g($a)"}

sub g {my $a=shift; $a*$a}

$a=$b="5";

$a=~s/([0-9])/f($1)/e; print "$a\n";
# output: 5+g(5)

$b=~s/([0-9])/f($1)/ee; print "$b\n";
# output: 30
```

Si tratta di caratteristiche molto potenti.

Riassunto dei modificatori

- **/m** ^ e $ si riferiscono all'inizio e alla fine di ogni riga.
- **/s** Il punto comprende anche il carattere di nuova riga.
- **/g** Matching ripetuto globale.
- **/i** Non viene fatta distinzione tra maiuscole e minuscole.
- **/o** Variabili contenute nell'istruzione di matching vengono calcolate solo all'inizio.
- **/e** La stringa di sostituzione viene valutata.

$_ sottinteso nelle espressioni regolari

Quando un'istruzione di matching si applica alla variabile sottintesa *$_*, anche =~ può essere tralasciato:

```
@a=("vero","verde","rosso","giallo","cara");
for (@a) {if (/o$/) {print "$_ termina in o.\n"}}

@a=("vero","verde","rozzo","giallo","cara");
for (@a) {s/o/a/g} # Modifica la lista!
for (@a) {print "$_\n"}

for (@a) {if (!/oz/) {print "Non trovato in $_\n"}}
```

Alternative a / ... /

Invece di $/\alpha/$ si può anche usare $m\{\alpha\}$ e invece di $s/\alpha/\beta/$ anche $s\{\alpha\}\ [\beta]$ oppure una di tante altre forme. La seconda forma è talvolta più trasparente; è utile quando α e β sono troppo lunghe per stare insieme sulla stessa riga. Se anche da sole sono lunghe, ci si può aiutare ad esempio introducendo variabili.

---

split e join

Da una stringa *$a* con *@a=split(reg,$a)* si ottiene una lista *@a* che consiste delle parti di *$a* che si ottengono separando la stringa usando l'espressione regolare *reg* come separatore. Esempi:

```
@a=split(/ /,"parolalunga");
for (@a) {print "$_\n"} # output: le singole lettere

@a=split(/ /,"alfa beta gamma");
for (@a) {print "$_\n"} # output: le tre parole

@a=split(/[ ,\-]+/,"alfa, beta - gamma");
for (@a) {print "$_\n"} # output: le tre parole
```

join($a,@b) restituisce una stringa che consiste degli elementi di *@b* uniti da *$a*.

Usiamo *split* per definire una funzione che crea una funzione booleana data in forma normale disgiuntiva; l'argomento di questa funzione è una stringa che contiene, separati da virgole, i valori per cui la funzione booleana assume il valore 1:

```
sub boole {my $a=shift; my @a=split(/\s*,\s*/,$a);
  sub {my $x=shift; for (@a) {return 1 if $x eq $_} 0}}
```

Possiamo sperimentarla con la funzione *f* dell'esercizio 1.24:

```
$f=boole('0001,0010,0011,0101,1100,1011,1101,1111');
for ('0000','1011','0001','0110','1110') {print $f->($_)}
```

La funzione pos

Questa funzione restituisce la posizione in cui l'ultimo passaggio di un'istruzione /... / si è fermato. Esempio:

```
$a="La Divina Commedia";
while ($a=~/([aeiou])/g) {print "$1 ", pos $a, " "}
# output: a 2 i 5 i 7 a 9 o 12 e 15 i 17 a 18
```

Siccome la posizione indicata è quella dopo l'ultima corrispondenza, il primo valore restituito è la posizione dopo la prima *a*, cioè 2. Infatti:

```
$a="0123 e 456";
while ($a=~/(\d+)/g) {print "$1 ", pos $a, " "}
# output: 0123 4 456 10
```

Parentesi tonde

Le parentesi tonde possono essere usate semplicemente per raggruppare gli elementi di una parte di un'espressione regolare. Allo stesso tempo però il contenuto della parte della corrispondenza rilevata viene memorizzato in variabili numerate *$1*, *$2*, *$3*, ... che possono essere utilizzate al di fuori dell'espressione regolare stessa (nelle sostituzioni anche nella stringa sostituente):

```
$a="fine 65 345 era 900";
$a=~/(\d+) +(\d+).*?(\d+)/; print "$1:$2:$3\n";
# output: 65:345:900
```

All'interno dell'espressione regolare invece alle parti rilevate ci si riferisce con *\1*, *\2*, ecc. L'esempio che segue mostra come eliminare caratteri multipli da una stringa:

```
$a="brrrrhhhh... che freddo!";
$a=~s/(.)\1+/$1/g;
print "$a\n"; # output: brh. che fredo!
```

Esistono anche parentesi il cui contenuto non viene memorizzato nelle variabili *$1*, ...:

(?:x)	Semplice parentesi non memorizzata.
a(?=x)	*a* deve essere seguito da *x*. (*)
a(?!x)	*a* non deve essere seguito da *x*. (*)
(?<=x)a	*a* deve essere preceduto da *x*. (*)
(?<!x)	*a* non deve essere preceduto da *x*. (*)
(?i:x)	Attiva il modificatore */i* per il contenuto della parentesi.
(?-i:x)	Disattiva il modificatore */i* per il contenuto della parentesi.
(?s:x)	Attiva il modificatore */s* per il contenuto della parentesi.
(?-s:x)	Disattiva il modificatore */s* per il contenuto della parentesi.

(*) Le parentesi condizionali non occupano posto!

```
$a=$b="a315 b883";
$a=~s/b\d+//; print "$a\n"; # output: a315
$b=~s/(?<=b)\d+//; print "$b\n"; #output: a315b

$a=$b="alfa [9] beta [7]";
sub f {$a=shift; $a*$a}
$a=~s/\[(\d+)\]/f($1)/ge; print "$a\n";
# output: alfa 81 beta 49
$b=~s/(?<=\[)(\d+)(?=\])/f($1)/ge; print "$b\n";
# output: alfa [81] beta [49]

$a=$b="AalEmiAreOtAea";
$a=~s/([eiou]|(?-i:a))//gi; print "$a\n";
# output: bt kfr
```

Ricerca massimale e ricerca minimale

/α/* cerca una corrispondenza di lunghezza massimale con una parola della forma *α**. Per ottenere una corrispondenza minimale si aggiunge un punto interrogativo: */α*?/*. Lo stesso discorso vale per *α+*. Esempi:

```
$a="era [ter] e [bis] uno";
$a=~/\[(.*)\]/; print "$1\n"; # output: ter] e [bis

$a="era [ter] e [bis] uno";
$a=~/\[(.*?)\]/; print "$1\n"; # output: ter
```

Attenzione: Bisogna però tener conto del punto in cui si trova l'elaborazione:

```
$a="babaaaaa";
$a=~/(a+)/; print "$1\n"; # output: a
```

Lettura a triple del DNA

Nel codice genetico l'aminoacido isoleucina è rappresentato dalle triple *ATA*, *ATC* e *ATT*. Una stringa deve però essere letta a triple, quindi il primo *ATA* (a partire dalla seconda lettera) in *TATATCTGCAATTTGATAGATCGA* non verrà tradotto in isoleucina, perché appartiene in parte alla tripla *TAT* e in parte ad *ATC*. Presentiamo prima un modo errato di lettura, poi due versioni corrette, nella seconda delle quali facciamo uso di *grep* (pag. 2).

```
$a=$b="TATATCTGCAATTTGATAGATCGA";
```

Triple: TAT ATC TGC AAT TTG ATA GAT CGA.

```
@a=$a=~/AT[ACT]/g;
for (@a) {print "$_ "} print "\n";
# output errato: ATA ATT ATA ATC
```

Lettura: T**AT A**TC TGC A**AT T**TG **ATA** G**AT C**GA.

```
$b=~s/(...)/($1)/g;
# $b="(TAT)(ATC)(TGC)(AAT)(TTG)(ATA)(GAT)(CGA)"
@a=$b=~/\((AT[ACT])\)/g;
for (@a) {print "$_ "} print "\n";
# output corretto: ATC ATA
```

L'uso delle parentesi tonde nell'istruzione di matching fa in modo che *@a* contenga solo le parti catturate dalle parentesi.

```
@a=grep {/AT[ACT]/} $a=~/.../g;
# $a=~/.../g è la lista delle triple!
for (@a) {print "$_ "} print "\n";
# output corretto: ATC ATA
```

Uso di | nelle espressioni regolari

/Franco (Nero|[a-z]+)/ rileva *Franco Nero* e *Franco piangeva*, ma non *Franco Gotti*.

/a|b|c/ richiede una corrispondenza con *a* oppure con *b* oppure con *c*, nell'ordine indicato. Quindi *a|abc* non applica mai ad *abc*, perché viene subito scoperta la corrispondenza con *a*, dopodiché l'elaborazione continua con *bc*.

```
sub tira {grep {$_}
   split(/(scia|scie|scio|sciu|sce|sci|[a-z])/,shift)}
for (tira("lascio il casco col fascino che nasce sul vascello"))
   {print "+$_\n"}
```

Questo esempio è molto importante. Provarlo subito! Si vede che split restituisce anche i separatori se sono individuati da parentesi tonde.

/a|b/ cerca in ogni posizione *a* oppure *b*; */a/ or /b/* cerca invece prima *a* in tutta la stringa e *b* solo, se la stringa non contiene *a*.

index e rindex

Il Perl prevede alcune funzioni per le stringhe che, quando applicabili, sono più veloci delle espressioni regolari (che spesso richiedono numerosi confronti).

index($a,$b) fornisce la posizione della prima apparizione della stringa *$b* nella stringa *$a* oppure *-1*, se *$b* non è sottostringa (cioè fattore) di *$a*. Con **index($b,$a,$start)** si ottiene la posizione della prima apparizione a partire da *$start*. **rindex** funziona nello stesso modo, ma con una ricerca da destra a sinistra (con la posizione però calcolata sempre a partire da sinistra). Attenzione all'ordine degli argomenti - la stringa più grande viene indicata per prima.

PROGRAMMAZIONE IN PERL

I puntatori del Perl

Il Perl permette di utilizzare puntatori che dal punto di vista funzionale sono molto simili ai puntatori del C, anche se la sintassi è piuttosto diversa. Il termine inglese è reference (riferimento), ma, almeno per il programmatore C/C++, questa è una terminologia un po' infelice perché la somiglianza con i riferimenti del C++ è minore che con i puntatori del C. Nel C e nel Perl i puntatori vengono usati soprattutto per tre scopi: modifica di un argomento; risparmio di memoria (un puntatore nel Perl è uno scalare, nel C un intero lungo), ad esempio nel passaggio dei parametri a una funzione; creazione di strutture complesse (liste di liste, alberi). Puntatori a funzioni permettono la programmazione funzionale.

I puntatori sono estremamente utili; in Perl il loro uso e soprattutto le notazioni che bisogna usare sono sicuramente meno semplici e trasparenti di quanto accada nel C. Le regole più importanti sono queste:

Un puntatore è uno scalare.

Il puntatore a un oggetto x viene denotato con \x (equivalente a &x nel C).

Per ottenere il valore dell'oggetto a cui punta il puntatore, bisogna anteporgli il simbolo caratteristico del tipo di dati ($ per gli scalari, @ per le liste, % per i vettori associativi, & per le funzioni).

Se il puntatore $u punta a una lista, l'elemento con indice i di quella lista lo si ottiene con $$u[i] (in accordo con la notazione precedente) oppure con $u→[i] (un'abbreviazione che è talvolta da preferire). Le notazioni per vettori associativi e funzioni sono simili. Esempi:

```
$a=5; $u=\$a;
print "$$u\n"; # output: 5

@a=(1,2,3); $u=\@a;
print "@$u\n"; # output: 1 2 3
print $u→[1], "\n"; # output 2

%a=("Rossi",30, "Verdi",27); $u=\%a;
while (($x,$y)=each %$u)
   {print "$x: $y, "}
# output: Rossi: 30, Verdi: 27,
print "\n", $u→{"Verdi"},"\n";
# output: 27

sub f {my $a=shift; $a*$a}

$u=\&f; print &$u(6); # output: 36
print "\n",$u→(4); # output: 16
```

Liste di liste e matrici

Il Perl nella sua impostazione lineare a liste non conosce matrici, liste di liste o alberi. Queste strutture complesse devono essere create tramite l'uso di puntatori.

La matrice $\begin{pmatrix} 3 & 5 \\ 1 & 8 \end{pmatrix}$ può essere rappresentata in modi diversi, ad esempio con

```
@a=(3,5); @b=(4,8); @m=(\@a,\@b);
print $m[0]→[0]; # output: 3
```

In questo esempio *$m[0]* è il puntatore alla lista *@a*. Spesso più comodo è l'uso di liste anonime: *[1,2,3]* è il puntatore a una lista anonima (cioè senza nome) i cui elementi sono *1*, *2* e *3*. Diamo in questo modo un'altra rappresentazione della nostra matrice:

```
$u=[[3,5],[4,8]];
print $u→[0]→[0]; # output: 3
```

Esaminiamo la prima riga in dettaglio. *$u* è il puntatore a una lista, i cui due elementi sono i puntatori *[3,5]* e *[4,8]* (quindi scalari), i quali a loro volta puntano a liste con due elementi.

Quando, in un linguaggio qualsiasi (C, Perl, Lisp), si usano strutture i cui elementi sono puntatori, bisogna fare molta attenzione alle operazioni di copiatura. La semplice assegnazione infatti copia in tal caso i puntatori, e quindi una modifica nell'originale modifica anche la copia e viceversa, cioè copia e originale non sono più indipendenti. Copie indipendenti si chiamano *copie profonde*; per crearle bisogna avere delle informazioni sul modo in cui sono organizzati i dati che devono essere copiati. Esempi:

```
@a=([1,2],[3,4]); @b=@a;
$b[0]→[0]=7; print $a[0]→[0];
# output: 7 (copia superficiale)

sub copia {my $a=shift;
   ([$a→[0]→[0],$a→[0]→[1]],
    [$a→[0]→[0],$a→[0]→[1]])}

@a=([1,2],[3,4]); @b=copia(\@a);
$b[0]→[0]=7; print $a[0]→[0];
# output: 1 (copia profonda)
```

Questa settimana

Numeri casuali e rand

rand(n) restituisce un numero casuale reale x con $0 \le x < n$. *rand()* è equivalente a *rand(1)*. Per ottenere un valore casuale intero, ad esempio tra 1 e 6, si può usare *int(rand(6))+1)*. Possiamo quindi definire una funzione *dado*:

```
sub dado {my ($a,$b)=@_;
   if (not defined $b) {$b=$a; $a=1}
   $a+int(rand($b-$a+1))}
```

In questo modo *dado(n)* è equivalente a *dado(1,n)*. Facciamo una prova per *dado(3)* calcolando anche le frequenze:

```
$a{0}=$a{1}=$a{2}=0;
for (0..50)
{$x=dado(3); $a{$x}++; print $x}
print "\n$a{1} $a{2} $a{3}\n";
```

L'impostazione dei valori iniziali viene gestita automaticamente dal Perl e non deve essere fatta dal programmatore.

***

Ordinare una lista con sort

```
@a=(2,5,8,3,5,8,3,9,2,1);

@b=sort {$a <=> $b} @a;
print "@b\n";
# output: 1 2 2 3 3 5 5 8 8 9

@b=sort {$b <=> $a} @a;
print "@b\n";
# output: 9 8 8 5 5 3 3 2 2 1

@a=("alfa","gamma","beta","Betty");
@b=sort {lc($a) cmp lc($b)} @a;
print "@b\n";
# output: alfa beta Betty gamma
```

La funzione *sort* prende come argomento un criterio di ordinamento e una lista e restituisce la lista ordinata come risultato. La lista originale rimane invariata. Nel criterio di ordinamento le variabili *$a* e *$b* hanno un significato speciale, simile a quello di *$_* in altre occasioni.

substr

Sia $\$a = a_0a_1 \dots a_n$ una stringa. **substr($a,i)** fornisce allora la stringa $a_ia_{i+1} \dots a_n$, mentre **substr($a,i,k)** è uguale a $a_ia_{i+1} \dots a_{i+k-1}$ (una stringa con k caratteri) oppure una stringa più breve, se $\$a$ non possiede tutti i caratteri richiesti. Se i è negativo, indica la posizione $n - i + 1$.

```
$a="012345678";
print substr($a,2),"\n"; # output: 2345678
print substr($a,2,4),"\n"; # output: 2345
print substr($a,-3),"\n"; # output: 678
print substr($a,-3,2),"\n"; # output: 67
```

substr($a,i,k,$b) sostituisce $a_ia_{i+1} \dots a_{i+k-1}$ con *$b*:

```
$a="Vivo a Pisa da molti anni.";
substr($a,7,4,"Ferrara"); print $a;
# output: Vivo a Ferrara da molti anni.
```

Strumenti per le stringhe

Definiamo una funzione che elimina gli spazi bianchi (cfr. pag. 12) all'inizio e alla fine di una stringa.

```
sub strip {$_[0]=~s/^\s+//; $_[0]=~s/\s+$//}

$a=" alfa "; strip($a);
print "+$a-\n"; # output: +alfa-
```

La funzione **sepmai** separa una stringa non solo a seconda degli spazi bianchi, ma estraendo anche le parti iniziando con maiuscola:

```
sub sepmai {my $a=shift; split(/(?=[A-Z])|\s+/,$a)}

$a="ABxC13 xD14"; @a=sepmai($a);
$b=""; for (@a) {$b.="$_ "} chop($b); chop($b);
print $b; # output: A, Bx, C13, x, D14
```

---

lc e uc

Queste funzioni convertono una stringa in minuscole oppure maiuscole. *lc* è un'abbreviazione di *lowercase*, *uc* un'abbreviazione di *uppercase*. *ucfirst* e *lcfirst* convertono solo la prima lettera della parola. Esempi:

```
$a="Carlo Magno imperatore d'Europa.";

$b=uc($a); print "$b\n";
# output: CARLO MAGNO IMPERATORE D'EUROPA.

$c=lc($a); print "$c\n";
# output: carlo magno imperatore d'europa.
```

Puntatori a variabili locali

A differenza dal C, nel Perl il valore di una variabile locale (dichiarata con *my*) rimane utilizzabile se esistono ancora puntatori che puntano ad essa. Esempio:

```
sub f {my $a=6; \$a}

$u=f(); print $$u; # output: 6
```

Passaggio di parametri in Perl

La funzione seguente non è in grado di modificare il valore del proprio parametro; ciò non significa però che anche in Perl il passaggio dei parametri avviene per valore, come adesso vedremo.

```
sub aumenta0 {my $x=shift; $x++}
$a=5; aumenta0($a); print $a; # output: 5
```

Ci sono almeno tre modi in Perl per ottenere il risultato desiderato. *aumenta1* funziona come la corrispondente funzione in C, prendendo come argomento l'indirizzo della variabile da modificare; le altre due versioni sono tipici meccanismi del Perl. Si vede che la variabile passata mantiene la propria identità; il passaggio dei parametri avviene per indirizzo.

```
sub aumenta1 {my $x=shift; $$x++}
$a=5; aumenta1(\$a); print "$a\n"; # output: 6

sub aumenta2 {$_[0]++}
$a=5; aumenta2($a); print "$a\n"; # output: 6

sub aumenta3 {my $x=\shift; $$x++}
$a=5; aumenta3($a); print "$a\n"; # output: 6
```

La ragione perché non funziona *aumenta0* è che l'istruzione *$a=shift*, necessaria perché la dichiarazione di una funzione in Perl non contiene nomi per le variabili, crea in quel momento una copia del primo argomento il cui aumento non modifica l'argomento stesso, a cui posso invece riferirmi con *\shift* oppure *$_[0]*. Due altri esempi:

```
sub f {$_[2]=$_[2]+7}
$x=3; f(0,0,$x); print "$x\n"; # output: 10

sub g {for (@_) {$_++}}
@a=(3,4,9); g(@a); print "@a\n"; # output 4 5 10
```

Però

```
sub f {push (@_,7)}
@a=(1,2); f(@a); print "@a\n"; # output: 1 2
```

e invece – notare le parentesi graffe necessarie in @{*$_[0]*}:

```
sub g {push (@{$_[0]},7)}
@a=(1,2); g(\@a); print "@a\n"; # output: 1 2 7
```

Invertiparola e eliminacaratteri

Il modo migliore di invertire una parola in Perl è tramite l'utilizzo della funzione *reverse* (pag. 2):

```
sub invertiparola {my $a=shift;
  my @a=reverse split(//,$a); join("",@a)}

$a="012345"; $b=invertiparola($a);
print $b; # output: 543210
```

Per eliminare un insieme di caratteri da una stringa si può usare (ad esempio) l'operatore di sostituzione.

```
sub eliminacaratteri {my ($a,$b)=@_; $a=~s/[$b]//g; $a}

$a="un famoso tenore"; $b=eliminacaratteri($a,"aeiou");
print "$a - $b\n"; # output: un famoso tenore - n fms tnr
```

Esercizio: Riscrivere queste funzioni in modo tale che modificano i propri argomenti, usando il metodo spiegato nell'articolo precedente (cfr. *strip*).

Se il programma principale si trova nella stessa cartella e se non richiede inserimenti da tastiera dell'utente, sotto Emacs (con la nostra impostazione) per l'esecuzione è sufficiente premere il tasto *Fine*.

ELIZA

Nel 1966 Joseph Weizenbaum presentò un programma che simulava una conversazione. Chiamò il programma ELIZA; spesso anche le imitazioni si chiamano così e con Perl è molto facile realizzarle. Nonostante la semplicità del programma, molti utenti ne rimasero fortemente affascinati o persino assoggettati, talvolta credendo addirittura che si trattasse di un interlocutore umano.

Il programma, nella nostra imitazione, consiste di quattro files di complessivamente 4 K e utilizza i files tematici descritti nel articolo successivo. Il file principale ***alfa*** *contiene soltanto le istruzioni* ***use*** *e chiama le funzioni* ***presentazione*** *del modulo* ***saluti*** *e* ***generico*** *del modulo* ***dialogo****.*

Il file ***saluti.pm*** *contiene la presentazione e il saluto d'addio, oltre a una funzione di data che viene utilizzata per contrassegnare con la data il file su cui verrà registrata la conversazione.*

Il file ***dialogo.pm*** *contiene una funzione di interfaccia (****generico****), funzioni per determinare la risposta e la funzione* ***apprendi*** *che permette a ELIZA una specie di apprendimento attraverso la registrazione delle reazioni dell'utente ai commenti del programma.*

Il quarto file, chiamato ***aus.pm****, contiene alcune funzioni ausiliarie, tra cui quella di caricamento dei temi.*

Le funzioni di tutti e quattro i files verranno descritte in dettaglio sulle pagine seguenti.

Struttura dei files tematici

I files tematici, contenuti nella cartella **Temi**, sono di tre specie.

Il file **casuali** è un semplice elenco di osservazioni casuali, che il programma utilizza in mancanza di stimoli più specifici.

Un gruppo di files (**conversazione**, **lettere**, **linux**, **proverbi**, **racconti** e il file **appresi** creato dal programma stesso) contiene delle coppie stimolo/risposta, separate da righe vuote, ad esempio:

```
eseguibile -rendere
Per avere informazioni sulla locazione e sul tipo di programmi
eseguibili e alias di comandi si usa type.

eseguibile rendere
Per rendere eseguibile un file alfa si usa chmod +x alfa.
```

Il programma confronta la frase dell'utente con la prima riga della coppia e fornisce la risposta indicata, se tutte le parole della prima riga che non sono precedute dal segno – appaiono nella frase, mentre le parole precedute da – non devono comparire.

Il file **trasformazioni**, che riportiamo per intero, contiene invece regole formulate con l'uso di espressioni regolari.

```
(non | )e' (molto | )(importante| triste| brutto| difficile)
Perche' $1è $2$3?

(non | )(?:voglio| vorrei) ([\w ']*)
Perche' $1vorresti $2?

(non | )ho bisogno di ([\w ']*)
Perche' $1hai bisogno di $2?

(non | )credo
Perche' $1lo credi?

(non | )devo ([\w ']*)
Perche' $1devi $2?

manca ([\w ']*)
E' importante che manca $1?

mi sento ([\w ']*)
Perché ti senti $1?
```

Questa settimana

Risposte casuali

L'inizio del file **casuali**:

```
Anima grande!
C'è chi sta peggio.
Capita anche a me.
Certo.
Chi lo sa?
Come dici?
Come ti senti adesso?
Continua.
Cosa è successo?
Cosa pensi di fare?
Cosa stai pensando?
Cosa vuoi fare?
Dimmi di più, forse ti posso aiutare.
Dove abiti?
E quindi?
E tu cosa pensi a questo proposito?
E tu sei contenta/o?
È proprio strano.
È proprio una bella storia.
È un'esperienza importante.
È un'idea tua?
Fermati.
Figurati.
I tuoi amici cosa dicono?
Infatti.
Ma dai!
Ma guarda.
Me lo posso immaginare.
Ne hai parlato con un'amica/o?
Ne ho sentito parlare.
Ne possiamo parlare quando vuoi.
Non c'è dubbio.
Non è mai troppo tardi.
Non è possibile.
Non è vero.
Non lo dimenticherò.
Non ne sarei così certa/o.
```

Una conversazione con ELIZA

Ciao, sono Eliza. Come ti chiami?
Buon giorno, sono l'ing. Cirrantini.
Buon giorno, ing. Cirrantini! Come stai?
Abbastanza bene. E poi oggi è domenica.
Talvolta il destino è testardo.
Ci aspetta una settimana intensa.
Parli bene l'italiano.
Potrebbe essere meglio.
Cosa è successo?
Domani devo lavorare.
Perché devi lavorare?
Perché ho una scadenza giovedì prossimo.
Conosci Linux?
Certamente. Ci lavoro tutto il giorno.
Talvolta il destino è testardo.
È la seconda volta che lo dici. Ma sabato vado al mare.
Ti piace il mare?
Sì, quando non c'è troppa gente. In autunno è bello.
Si sta bene qui in autunno.
È vero. È la stagione migliore.
Ne hai parlato con un'amica?
Sì, forse viene anche lei.
Capita anche a me.
Vorrei anche migliorare il programma per Eliza.
Perché vorresti anche migliorare il programma per eliza?
Dovrei inserire ancora più elementi linguistici.
Fermati.
Come? Ne hai paura? Più cose sai fare, più impressioni la gente.
Devi farti coraggio. Qualcuno ti aiuterà.
Infatti. Appena ho tempo, ci penso.
Ti piace Linux?
È il sistema operativo migliore. Come si fa il catalogo sotto Unix?
ls (ls -l per il catalogo in formato lungo).
Bravo. Credo che impari presto.
Perché lo credi?
Perché vedo che fai progressi.
Conosci Linux?
Sì, te l'ho già detto. Sembra che ti manca la memoria.
È importante che manca la memoria?
Eh sì, se no mi ripeti sempre le stesse cose.
Però tu sei molto brava.
Sono maschio.
Ah, scusa.
Niente.
Quando ti laurei?
Sono già laureato. E tu quando ti laurei?
Sono già laureato. E tu quando ti laurei?
Ma dai. Non devi ripetere quello che dico io.
Cosa vuoi fare?
Vorrei stampare questo dialogo.
Perché vorresti stampare questo dialogo?
Mi serve per la lezione. Comunque non è male.
Ad ognuno la sua croce.
Certo. Ci sono ancora tanti problemi per l'umanità.
Cosa è successo?
Succede sempre qualcosa purtroppo.
Tu?
Come io? Il mondo fa paura.
Come io? Il mondo fa paura.
Non devi ripetere quello che dico. Devo cambiare il programma.
Perché devi cambiare il programma?
Perché se no ripeti quello che dico io. Adesso ti dovrò salutare.
Come io? Il mondo fa paura.
Vedo che impari le cose sbagliate. Ma migliorerai.
Ti capisco bene.
Ottimo. Allora arrivederci.
Ciao, ing. Cirrantini, buona giornata!

Il file Eliza/alfa

```
#! /usr/bin/perl -w
use strict 'subs'; use strict 'refs';

use lib '..';

use aus; use dialogo; use files; use saluti;

saluti::presentazione();
dialogo::generico();
```

L'istruzione *use lib '..'* ci permette di usare il modulo *files* che si trova nella cartella che contiene il nostro programma.

Il file Eliza/saluti.pm

```
1; # Eliza/saluti.pm
package saluti;

*salva=\&aus::salva;

$saluti='(Ciao, | Buon ?giorno, | Buona ?sera, )?';
$presentazione='([Ss]ono (?:il | la | l\')?| [Mm]i chiamo )?';

# Le variabili globali $addio, $file e $nome vengono
# impostate in presentazione.
####################################
sub addio {my $a=shift;
   return ", buona giornata!" if $a=~/^Buon ?giorno/;
   return ", buona notte." if $a=~/^Buona ?sera/; "!"}

sub data {my @a=localtime(); my $anno=1900+$a[5];
   my $mese=1+$a[4];
   my $giorno=$a[3]; $ora=$a[2]; $min=$a[1];
   $mese="0$mese" if $mese<10;
   $giorno="0$giorno" if $giorno<10;
   $ora="0$ora" if $ora<10; $min="0$min" if $min<10;
   "$anno$mese$giorno-$ora$min"}

sub fine {my $eliza="\nCiao, $nome$addio\n";
   salva($eliza); print $eliza}

sub presentazione {my $data=data();
   my $eliza="\nCiao, sono Eliza. Come ti chiami?\n\n";
   $file="Protocollo/eliza-$data";
   files::scrivi($file,$eliza); print $eliza;
   my $utente=<main::stdin>; salva($utente);
   chomp($utente); $utente=~s/[.\s]*$//; my $saluto;
   $utente=~/$saluti$presentazione(.*)/;
   if (not defined $1) {$saluto="Ciao, "} else {$saluto=$1}
   $addio=addio($saluto);
   $nome=$3; $eliza="\n$saluto$nome! Come stai?\n\n";
   salva($eliza); print $eliza}
```

Studiare prima le variabili *$saluti* e *$presentazione*. In *$saluti* si noti che il saluto deve trovarsi all'inizio, per distinguerlo dalle altre parti, anche se è improbabile che qualcuno si presenti dicendo ad esempio *"Io sono il buon Giorno"*. La variabile *$addio* che viene calcolata dalla funzione *addio* verrà utilizzata alla fine nel congedo.

La conversazione viene registrata in un file della cartella *Protocollo*; il nome del file contiene la data calcolata dalla funzione *data* che a sua volta contiene la funzione *localtime* del Perl che restituisce una lista con le componenti della data. Queste componenti sono, nell'ordine: secondi, minuti, ora, giorno del mese (1-31), mese (0-11), anno (dal 1900, quindi 2001 corrisponde a 101), giorno della settimana (0-6, con 0 per la domenica), giorno dell'anno (1-366), e infine un valore booleano (0-1) che è uguale a 1 durante il tempo estivo.

La funzione *salva* che registra le frasi della conversazione e che verrà usata anche dal modulo *dialogo* è definita in *aus.pm*.

Il file Eliza/dialogo.pm

```
# Eliza/dialogo.pm
package dialogo;

*carica=\&aus::carica;
*mf=\&aus::mf;
*rispostacasuale=\&aus::rispostacasuale;
*salva=\&aus::salva;

@temi=("appresi","conversazione","lettere","linux",
   "proverbi","racconti");
$ulteliza=0;
1;
################################################
sub apprendi {my ($eliza,$utente)=@_;
   my @a=grep {length($_)>=4} split(/\W+/,$eliza);
   if (@a>3) {@a=@a[0,1,-1]} $eliza=lc(join(" ",@a));
   files::aggiungi("Temi/appresi","$eliza\n$utente\n\n")}

sub condizione {my ($a,$x)=@_; my @x=split(/ +/,$x); my ($p,$u);
   for (@x) {$p=substr($_,0,1);
   if ($p eq "=") {return 0 if $a ne substr($_,1); next}
   if ($p ne "-") {return 0 if $a!~/$_/; next}
   $u=substr($_,1); return 0 if $a=~/$u/} 1}

sub fine {my $a=shift; $a=~/ciao|arrivederci/}

sub generico {my ($utente,$eliza); while (1)
   {$utente=<main::stdin>; salva($utente); chomp($utente);
   if (fine(lc($utente))) {saluti::fine(); return}
   apprendi($ulteliza,$utente) if $ulteliza;
   $utente=lc($utente);
   $eliza="\n".risposta($utente)."\n\n"; salva($eliza); print $eliza}}

sub risposta {my $a=shift; my ($temi,$x,$y,$u,$v,$w);
   my @x; my $trasfo;
   if ($a=~/sono (un )?maschio/)
   {$aus::femmina=0; return "Ah, scusa."}
   $trasfo=carica("Temi/trasformazioni");
   while (($x,$y)=each %$trasfo)
   {if ($a=~/$x/) {if (defined $1) {$u=$1} else {$u=""}
   if (defined $2) {$v=$2} else {$v=""}
   if (defined $3) {$w=$3} else {$w=""}
   $y=~s/\$1/$u/g; $y=~s/\$2/$v/g; $y=~s/\$3/$w/g;
   $risposta=$y; goto fine}}
   for (@temi) {$temi=carica("Temi/$_");
   while (($x,$y)=each %$temi)
   {if (condizione($a,$x)) {$risposta=$y; goto fine}}}
   $risposta=rispostacasuale();
   fine: while ($risposta eq $ulteliza or $risposta eq $a)
   {$risposta=rispostacasuale()}
   $ulteliza=$risposta; mf($risposta)}
```

La funzione d'ingresso di questo file è la funzione *generico* che riceve la frase dell'utente, controlla se contiene *ciao* o *arrivederci* e termina la conversazione se ciò accade, memorizza nel file *Temi/appresi* la frase dell'utente come risposta del programma da utilizzare in futuro, infine calcola la risposta che viene registrata insieme alla frase dell'utente.

La funzione *risposta* corregge la variabile *$femmina* del modulo *aus* se necessario, carica poi prima il file *trasformazioni* e, se non è applicabile, utilizza gli altri file tematici. Se non individua una risposta adatta, trova una risposta casuale. La terzultima riga serve per impedire che il programma ripeta quello che ha detto l'utente (eliminando così il problema visto nel dialogo a pag. 18) o la propria risposta precedente. La variabile *$ulteliza* contiene sempre l'ultima risposta di ELIZA.

Si esaminino bene le funzioni *apprendi* e *condizione*.

A differenza di quanto accada in molte implementazioni che contengono le regole e le frasi nel corpo del programma, l'uso di files tematici permette di aggiungere a piacere nuovi elementi di conversazione o di cambiare quelli esistenti. Si possono anche aggiungere o togliere files tematici semplicemente modificando la lista *@temi* all'inizio del modulo.

Il file Eliza/aus.pm

```
1; # Eliza/aus.pm
package aus;

use files;

$femmina=1;
@casuali=split(/\n/,files::leggi("Temi/casuali"));
################################################
sub ao {if ($femmina==1) {"a"} else {"o"}}

sub carica {my $file=shift; my $a=files::leggi($file);
   $a=~s/\n+$//;
   my @a=split(/\n\n/,$a); my %a; my ($x,$y); for (@a)
   {($x,$y)=split(/\n/,$_,2); $a{$x}=$y} \%a}

sub mf {my $a=shift; $a=~s/a\/o/ao()/ge; $a}

sub rispostacasuale {my $n=int(rand @casuali); $casuali[$n]}

sub salva {files::aggiungi($saluti::file,shift)}
```

Nell'impostazione iniziale il programma si aspetta un interlocutore femminile; l'utente può modificare questa impostazione con una risposta che contiene *sono maschio* oppure *sono un maschio* (cfr. l'inizio della funzione *risposta* in *dialogo.pm*). Le funzioni *mf* e *ao* insieme adattano le desinenze all'utente, usando in tutti quei casi, in cui la risposta contiene *a/o* la *a* per un utente femminile, altrimenti la *o*.

Nella funzione *rispostacasuali* si vede come si ottiene un numero casuale intero; si ricordi che in contesto scalare *@casuali* è la lunghezza della lista che qui contiene le righe del file tematico *casuali*.

I files tematici organizzati a coppie vengono letti da *carica*: prima le coppie vengono separate in corrispondenza delle righe vuote (caratterizzate da un doppio $\backslash n$, poi viene separata la prima riga di ogni coppia dal resto usando il terzo argomento di *split* che indica il numero delle parti in cui la stringa deve essere separata (cfr. pag. 18, dove non abbiamo menzionato questa comoda possibilità):

```
$a="Erano le quattro del mattino.";

@a=split(/ +/,$a,2); print "$a[0] ... $a[1]\n";
# output: Erano ... le quattro del mattino.

@a=split(/ +/,$a,3); print "$a[0] ... $a[1] ... $a[2]\n";
# output: Erano ... le ... quattro del mattino.
```

Typeglobs

Per utilizzare una funzione *f* di un package *alfa* al di fuori di quest'ultimo la funzione deve essere invocata con *alfa::f*. Così abbiamo fatto in alcune istruzioni nelle due pagine precedenti:

```
saluti::presentazione();
dialogo::generico();
files::scrivi($file,$eliza);
my $utente=<main::stdin>;
saluti::fine();
files::aggiungi($saluti::file,shift)
```

Torneremo su questo meccanismo e sui pacchetti più avanti quando parleremo della programmazione orientata agli oggetti in Perl.

Esistono vari meccanismi in Perl per poter usare nomi di funzioni o variabili senza il prefisso che individua il modulo, uno dei quali è l'utilizzo dei *typeglobs* che, ai nostri scopi, avviene nel modo seguente:

```
*salva=\&aus::salva;
*carica=\&aus::carica;
*mf=\&aus::mf;
*rispostacasuale=\&aus::rispostacasuale;
*salva=\&aus::salva;
```

eval

La funzione eval *permette di eseguire codice in Perl che è stato generato duranto l'esecuzione del programma. Questa caratteristica potente di molti linguaggi interpretati può essere utilizzata addirittura per scrivere programmi che si automodificano. L'argomento di* eval *può essere una stringa oppure un blocco di codice tra parentesi graffe.*

```
$a="3+7"; print "$a\n"; # output: 3+7
print eval($a),"\n"; # output: 10
$u=3; $a="\$u+7"; print "$a\n";
# output: $u+7
print eval($a),"\n"; # output: 10
```

Nel prossimo esempio la parte variabile contiene operatori binari.

```
@a=("+","*","-","/");
$a=12; $b=3;
for (@a) {print eval("$a$_$b")," "}
print "\n"; # output: 15 36 9 4
```

Un altro esempio di eval *applicato a una stringa:*

```
$a='for (0..2) {print "$_"}; print "\n"';
eval $a;
# output: 012
```

L'istruzione eval *applicata a un blocco viene spesso usata per catturare errori senza interrompere il programma. Consideriamo prima*

```
for (2,4,0,5) {print 1/$_," "}
print "Adesso continuo.\n";
```

Nell'esecuzione si avrà un output 0.5 0.25*, dopodiché la divisione per zero interrompe il programma: Non viene più stampato* Adesso continuo.*, mentre appare il messaggio* Compilation exited abnormally ....

Proviamo invece

```
eval {for (2,4,0,5) {print 1/$_," "}};
print "Adesso continuo.\n";
# output: 0.5 0.25 Adesso continuo.
```

Stavolta la divisione per zero causa soltanto l'uscita dal blocco che è argomento di eval*, ma poi l'esecuzione continua e quindi vediamo anche l'output successivo* Adesso continuo..

L'errore commesso viene assegnato alla variabile speciale $@ che può essere stampata come stringa:

```
print $@;
# output: Illegal division by zero
  at ./alfa line 20.
```

Attenzione: Abbiamo in questo modo la possibilità di controllare il tipo di errore commesso, ma l'output del messaggio d'errore è avvenuto tramite il nostro print $@; e non a causa di un'interruzione del programma che infatti, come abbiamo visto, continua.

eval può essere applicata a stringhe qualsiasi, quindi anche a stringhe inserita dalla tastiera da un utente oppure prelevate da un file. Ciò può essere estremamente utile, ma è anche pericoloso. Esempio:

```
while (1) {print "\n"; eval <stdin>}
```

Eseguendo il programma dalla shell, si può avere una sessione come la seguente - provare!

```
:) alfa
print 8;
8
$a=3; $b=7; print $a+$b;
10
exit;
:)
```

Problemi di sicurezza con eval

L'uso di *eval* può essere pericoloso. Se per esempio il programma che contiene la riga

```
while (1) {print "\n"; eval <stdin>}
```

può essere usato da un utente che non conosciamo collegato in rete o da un utente inesperto e questi inserisce dalla tastiera il comando *system("rm -f *")* o, peggio ancora, *system("rm -rf *")*, ci cancellerà tutti i files o addirittura tutte le cartelle. Lo stesso problema si pone se le stringhe a cui *eval* viene applicata sono contenute in files che non conosciamo o di cui non ricordiamo con precisione il contenuto.

Questa settimana

Attributi di files e cartelle

$file sia il nome di un file (in senso generalizzato). Gli attributi di files sono operatori booleani che si riconoscono dal – iniziale; elenchiamo i più importanti con il loro significato:

-e	*$file* ...	esiste
-r	*$file* ...	diritto d'accesso *r*
-w	*$file* ...	diritto d'accesso *w*
-x	*$file* ...	diritto d'accesso *x*
-d	*$file* ...	cartella
-f	*$file* ...	file regolare
-T	*$file* ...	file di testo

(-s $file) è la lunghezza del file. Con

```
@files=("alfa","beta","mu","alfa-6","Eliza");
for (@files) {printf("%-8s ",$_);
do {print "non esiste\n"; next} if not -e $_;
print "eseguibile " if -x $_;
print "cartella " if -d $_;
print "file " if -f $_;
print -s $_,"\n"}
```

otteniamo adesso l'output

```
alfa     eseguibile file 332
beta     file 14
mu       non esiste
alfa-6   eseguibile file 530
Eliza    eseguibile cartella 4096
```

Esercizio: Usare le funzioni per files e cartelle viste a pagina 3 e la funzione *cwd* (pagina 21) per esaminare il contenuto della cartella di lavoro e delle sue sottocartelle con un programma ricorsivo.

Cercare gli errori

Esistono vari strumenti per la ricerca degli errori sotto Perl. Il più semplice è l'inclusione del modulo **diagnostics** con l'istruzione *use diagnostics;* all'inizio del file. In questo modo si otterrà una dettagliata descrizione degli errori, soprattutto quelli di sintassi. Unico inconveniente è la lunghezza dei messaggi.

Il Perl contiene anche un proprio **debugger** che viene attivato con l'opzione *-d*, ad esempio **perl -d alfa**. Una descrizione dei comandi possibili sotto il debugger si ottiene con *h h* (corta) oppure con *|h* (lunga).

Spesso però la ricerca degli errori può essere anche fatta con semplici comandi di interruzione combinati con output di risultati o valori intermedi. Qui è utile anche il comando **exit** che ferma un programma nel punto in cui si trova.

Puntatori a vettori associativi

Per trasformare una funzione in un hash usiamo

```
sub hashdafun {my ($f,$x)=@_; my %a=();
  for (@$x) {$a{$_}=&$f($_)} \%a}
sub f {my $a=shift; $a*$a}
$a=hashdafun(\&f,[1,5,3,8]);
$out="";
for (keys %$a) {$out.="$_ $a→{$_}, "}
chop($out); chop($out); print "$out\n";
# output: 8 64, 1 1, 3 9, 5 25
```

In questo modo da una funzione otteniamo in pratica una tabella, la cui prima colonna contiene gli elementi del dominio di definizione della funzione (rappresentati dagli elementi della lista a cui punta *$x*), mentre la seconda contiene i rispettivi valori della funzione.

Studiare attentamente questo esempio; sembra semplice, ma è molto istruttivo perché contiene una buona selezione dei concetti trattati in questo numero.

Vettori associativi anonimi

A pagina 15 abbiamo introdotto puntatori a liste anonime; puntatori a vettori associativi anonimi possono essere definiti in modo molto simile, utilizzando le partentesi graffe al posto delle quadre. *{"Rossi",27,"Bianchi",28,"Verdi",25}* è il puntatore a una tabella hash con i valori indicati.

Puntatori non possono essere utilizzati come chiavi di un vettore associativo (esiste un modulo *RefHash* che dovrebbe permetterlo, ma non sembra del tutto affidabile); ciò restringe l'uso ricorsivo dei vettori associativi.

Liste normali e anonime e vettori associativi normali e anonimi possono essere combinati a piacere.

Cartelle e diritti d'accesso

Molti comandi della shell hanno equivalenti nel Perl, talvolta sotto nome diverso; nella tabella che segue i corrispondenti comandi della shell sono indicati a destra:

chdir	...	*cd*
unlink	...	*rm*
rmdir	...	*rm -r*
link alfa, beta	...	*ln alfa beta*
symlink alfa, beta	...	*ln -s alfa beta*
mkdir	...	*mkdir*
chmod 0644, alfa	...	*chmod 644 alfa*
chown 501,101,alfa	...	*chown 501.101 alfa*

Come si vede, in *chown* bisogna usare numeri (UID e GID). Per ottenere il catalogo di una cartella si usa *opendir* che è già stata utilizzata nella nostra funzione *catalogo* a pag. 3.

Per conoscere la cartella in cui ci si trova (a questo fine nella shell si usa *pwd*), in Perl bisogna usare la funzione *cwd*, che necessita del modulo *Cwd*. Quindi:

```
use Cwd;
$cartella=cwd(); # current working directory
```

Uso procedurale degli operatori logici

Abbiamo già osservato a pagina 4 che gli operatori logici *and* e *or* nel Perl non sono simmetrici. Più precisamente

$$A_1 \text{ and } A_2 \text{ and } \ldots \text{ and } A_n$$

è uguale ad A_n se tutti gli A_i sono veri, altrimenti il valore dell'espressione è il primo A_i a cui corrisponde il valore di verità *falso*. Similmente

$$A_1 \text{ or } A_2 \text{ or } \ldots \text{ or } A_n$$

è uguale ad A_n se nessuno degli A_i è vero, altrimenti è uguale al primo A_i che è vero.

Inoltre l'operatore *and* lega più strettamente dell'*or*, quindi invece di *(A and B) or (C and D)* possiamo semplicemente scrivere *A and B or C and D*.

Possiamo utilizzare questo comportamento degli operatori logici per eliminare molti *if* (o anche tutti) nei programmi, avvicinandoci a uno stile di programmazione logica spesso più trasparente e più efficiente. Talvolta si vorrebbe che il valore di un'espressione booleana sia 1 se è vera e 0 se è falsa; il primo esempio definisce una funzione *boole* che si comporta in questo modo. Seguono altri esempi per illustrare l'utilizzo procedurale degli operatori logici.

```
sub boole {shift and 1 or 0}
# invece di: {return 1 if shift; 0}
sub positivo {shift>0 and 1 or 0}
# invece di: {return 1 if shift>0; 0}
sub sgn {my $a=shift;
  $a>0 and 1 or $a<0 and -1 or 0}
sub fatt {my $n=shift;
  $n==0 and 1 or $n*fatt($n-1)}
sub fattiterativo {my $n=shift;
  my $k=0; my $p=1;
  loop: $k==$n and $p or
   do {$k++; $p*=$k; goto loop}}
sub potenza {my ($x,$n)=@_;
  $n==0 and 1 or
  $n%2 and $x*potenza($x,$n-1)
  or potenza($x*$x,$n/2)}
```

La versione iterativa dell'ultima funzione è leggermente più complicata, rimane però sempre chiara e logica:

```
sub potenzaiterativa {my ($x,$n)=@_;
  my $p=1; loop: $n==0 and $p or
  do {$n%2==0 and do {$x*=$x; $n/=2}
  or do {$p*=$x; $n--}; goto loop}}
sub c01 {shift==0 and 1 or 0}
sub scambia01 {my $a=shift;
  $a=~s/([01])/c01($1)/ge; $a}
```

Un errore in cui si incorre facilmente è di usare invece, per ottenere una stringa in cui tutti gli 1 sono sostituiti da 0 e viceversa,

```
$a=~s/0/1/g; $a=~s/1/0/g;
```

Perché non funziona?

Si possono elaborare queste idee per giungere a affascinanti tecniche di *programmazione logica* in Perl.

Funzioni anonime

Una funzione anonima viene definita tralasciando dopo il *sub* il nome della funzione. Più precisamente il *sub* può essere considerato come un operatore che restituisce un puntatore a un blocco di codice; quando viene indicato un nome, questo blocco di codice può essere chiamato utilizzando il nome.

Funzioni anonime vengono utilizzate come argomenti o come risultati di funzioni come vedremo adesso. Funzioni anonime, essendo puntatori, quindi scalari, possono anche essere assegnate come valori di variabili:

```
$quadrato = sub {my $a=shift; $a*$a};
print &$quadrato(6).“\n”; # output: 36
print $quadrato→(6); # output: 36
```

Funzioni come argomenti di funzioni

Quando ci si riferisce indirettamente a una funzione (ad esempio quando è argomento di un'altra funzione), bisogna premettere al nome il simbolo *&*. Esempi:

```
sub val {my ($f,$x)=@_; &$f($x)}
sub cubo {my $a=shift; $a*$a*$a}
sub id {shift}
sub quadrato {my $a=shift; $a*$a}
sub uno {1}
$a=“quadrato”;
print val(\&quadrato,3),“\n”; # output: 9
print val(\&$a,3),“\n”; # output: 9
$s=0; for (“uno”,“id”,“quadrato”,“cubo”)
{$s+=val(\&$_,4)} print “1 + 4 + 16 + 64 = $s\n”;
# output: 1 + 4 + 16 + 64 = 85
```

In verità nell'ultimo esempio lo stesso risultato lo si ottiene con

```
$s=0; for (“uno”,“id”,“quadrato”,“cubo”)
{$s+=&{\&$_}(4)} print “1 + 4 + 16 + 64 = $s\n”;
# output: 1 + 4 + 16 + 64 = 85
```

oppure, in modo forse più comprensibile, con

```
$s=0; for (“uno”,“id”,“quadrato”,“cubo”)
{$f=\&$_; $s+=&$f(4)} print “1 + 4 + 16 + 64 = $s\n”;
# output: 1 + 4 16 + 64 = 85
```

Queste versioni sono permesse anche con l'impostazione *use strict 'refs'* che impedisce l'utilizzo di stringhe come riferimenti. Senza questa impostazione (che è comunque consigliabile in programmi seri) si potrebbe anche scrivere:

```
$s=0; for (“uno”,“id”,“quadrato”,“cubo”)
{$s+=&$_(4)} print “1 + 4 + 16 + 64 = $s\n”;
# output: 1 + 4 + 16 + 64 = 85
```

Nella grafica al calcolatore è spesso utile definire figure come funzioni. Una funzione che disegna una tale figura, applicando una traslazione e una rotazione che dopo il disegno vengono revocate, potrebbe seguire il seguente schema:

```
sub disegna {my ($f,$dx,$dy,$alfa)=@_;
  traslazione($dx,$dy), gira($alfa); &$f();
  gira(-$alfa); traslazione(-$dx,-$dy)}
sub figura1 {}
sub figura2 {}
disegna(\&figura1,2,3,60);
```

Programmazione orientata agli oggetti

Benché non esplicitamente visibile alla superficie, in Perl la programmazione orientata agli oggetti è particolarmente facile ed efficiente. In Perl sono separati tre aspetti elementari dalla strutturazione del codice sorgente: raccolta del codice su più files, suddivisione dello spazio dei nomi (packages), assegnazione di un oggetto ad una classe. Questa separazione dei tre componenti permette al programmatore disciplinato un'organizzazione versatile, efficace ed esteticamente gradevole del proprio lavoro. Presentiamo prima brevemente questi componenti che verranno successivamente discussi in dettaglio. Si noti quanto, a differenza di altri linguaggi della programmazione orientata agli oggetti, essi siano separati ed indipendenti tra loro.

(1) **Files.** Nel file principale che chiamiamo *alfa* e che inizierà con la riga *#! /usr/bin/perl -w*, vengono indicate le cartelle aggiuntive dove l'interprete cerca i files sorgente, con *use lib 'a, b, ... ';* e poi i files che intendiamo utilizzare con *use alfa; use beta; ...* In Perl questi files si chiamano moduli (pag. 3) e devono avere il nome *alfa.pm, beta.pm, ...* (*pm* significa *Perl module*). I moduli *non* sono associati ad un particolare spazio di nomi.

(2) **Pacchetti.** Ogni istruzione della forma *package* π; fa in modo che le variabili successivamente dichiarate (fino alla fine del file o fino alla prossima istruzione *package*) prendono il cognome"(o nome di classe) π e all'esterno del package la variabile x deve essere chiamata π*::x* (con il giusto prefisso quindi ad esempio *$π::x* per gli scalari). La suddivisione tematica per nomi di classe è completamente indipendente dai files in cui si trovano i nomi. Quando non è dichiarato esplicitamente un package, le variabili appartengono al package *main;* infatti *$a* è in tal caso lo stesso come *$main::a*. Ciò vale sia per il file principale sia per i moduli; ad esempio il file *beta.pm* sia così strutturato

```
$a=6;
package Mat;
$a=5;
package Fis;
$a=10;
package Mat;
$b=9;
```

Allora in *alfa* con l'istruzione:

```
print “$a $Mat::a $Fis::a $Mat::b\n”;
```

otteniamo l'output

6 5 10 9.

perché il primo *$a* nel modulo non essendo ancora definito un package è di *main*. In verità l'istruzione *$prova::a = 7* definisce una variabile di classe *prova* senza che debba essere dichiarato un pacchetto.

(3) **Oggetti.** Un riferimento (o puntatore) $r con *bless* $r diventa un oggetto del package in cui si trova l'istruzione.

Lo si può anche assegnare ad un altro package π con

```
bless $r,“π”;
```

Il bless si usa soprattutto nella definizione di costruttori

```
sub nuovo {bless @_}
sub nuovo {my $a=shift; bless $a}
sub nuovo {my ($x,$y,$z)=@_; bless {'x',$x, 'y',$y, 'z',$z}}
```

Adesso possiamo usare

```
$v=Vettore::nuovo(3,6,10);
print $v→{z}; # output: 10
```