Scritto delle lezioni del Corso di Programmazione

prof. Josef Eschgfäller

Mantovani Filippo
Dipartimento di Matematica - Università di Ferrara
http:\\members.xoom.it\filimanto\sistemi

Primo trimestre 2003

Indice

1 Introduzione **3**
- 1.1 Alcune definizioni . . . 5
- 1.2 Un esempio: i numeri di Fibonacci . . . 6
- 1.3 Il metodo del sistema di primo ordine . . . 7
- 1.4 Diagrammi di flusso . . . 8
- 1.5 I pseudolinguaggi . . . 9

2 Il linguaggio macchina e assembler **10**
- 2.1 Numeri esadecimali . . . 10
  - 2.1.1 Esempi: . . . 10
- 2.2 Un esempio nel linguaggio del processore 6502 . . . 11
- 2.3 I linguaggi assembler . . . 12

3 Il basic **14**

4 Fortran **16**

5 Pascal **18**
- 5.1 Un programma in Pascal per il calcolo del prodotto scalare . . . 18

6 Delphi **20**

7 C **21**
- 7.1 Un esempio . . . 22
- 7.2 Help: . . . 23

8 C++ **25**
- 8.1 Storia, bibliografia e osservazioni . . . 26

9 Java **27**
- 9.1 Un esempio . . . 28
- 9.2 Osservazioni . . . 28

10 Python **30**

11 Perl **32**
11.1 Help 33

12 Forth e PostScript **34**
12.1 Macroistruzioni 35
12.2 Diagrammi di flusso per lo stack 36
12.3 If e Ifelse 36

13 Lisp **38**
13.1 Il λ-calcolo 38
13.2 Osservazione ed esempi 39

14 Prolog **40**

15 PHP **41**

16 Gli altri linguaggi **44**
16.1 S ed R 44

17 Programmare in Perl **46**
17.1 Variabili nel Perl 46
17.1.1 Il nostro primo programma 47
17.2 Liste 48
17.2.1 Alcune operatori per le liste 49
17.2.2 Contesto scalare e contesto listale 49
17.2.3 La funzione grep del Perl 50
17.2.4 Map 51
17.2.5 Ordinare una lista con sort 51
17.3 Files 52
17.3.1 L'operatore $<>$ 52
17.3.2 Il pacchetto Files 53
17.3.3 Attributi di files e cartelle 54
17.3.4 I diritti d'accesso 54
17.3.5 La cartella di lavoro 55
17.4 Moduli 55
17.4.1 I moduli CPAN 56
17.5 Funzioni 56
17.5.1 Nascondere le variabili con my 57
17.5.2 Funzioni anonime 58
17.5.3 Funzioni come argomenti di funzioni 58
17.5.4 Funzioni come valori di funzioni 59
17.5.5 Programmazione funzionale in Perl 59

Capitolo 1

Introduzione

Un linguaggio di programmazione è un linguaggio che permette la realizzazione di algoritmi su un calcolatore. Il calcolatore esegue istruzioni numeriche (diverse a seconda del processore) che insieme formano il linguaggio macchina del calcolatore. Tipicamente queste istruzioni comprendono operazioni in memoria, istruzioni di flusso o di controllo (test, salti, subrutines, terminazioni), definizione di costanti, accesso a funzioni del sistema operativo, funzioni di input e output.

I numeri che compongono il linguaggio macchina possono essere inseriti direttamente in memoria. Un programma scritto in un altro linguaggio di programmazione deve essere invece convertito in linguaggio macchina; ciò può avvenire prima dell'esecuzione del programma tramite un *compilatore* che trasforma il programma scritto nel *linguaggio sorgente* in codice macchina oppure tramite un *interprete* che effettua una tale trasformazione durante l'esecuzione del programma e solo in parte per quei pezzi del programma che devono essere in quel momento eseguiti. Siccome il codice preparato da un compilatore è già pronto, mentre le operazioni dell'interprete devono essere ripetute durante ogni esecuzione, è chiaro che i linguaggi compilati (Assembler, C, ...) sono più veloci dei linguaggi interpretati (Perl, Lisp, Apl, Basic).

La velocità dell'hardware moderno rende meno importanti queste differenze. Spesso, come nel Perl, il linguaggio utilizza moduli (scritti ad esempio in C) già compilati e quindi la traduzione riguarda solo una parte del programma, oppure è possibile come in Java (e di nascosto anche in Perl) una compilazione in due tempi che traduce il programma sorgente prima in codice indipendente dalla macchina (linguaggio per una macchina virtuale o bytecode) che su un calcolatore specifico viene poi eseguito da un interprete.

Nonostante che algoritmi e programmi dovrebbero essere separati il più possibile la sintassi e soprattutto i tipi di dati previsti o definibili di uno specifico linguaggio inducano a paradigmi di programmazione diversi molto spesso è possibile, una volta appresi i meccanismi, imitare le costruzioni di un

linguaggio anche in altri; per questa ragione è molto utile conoscere linguaggi diversi per avere un bagaglio di tecniche applicabili in molti casi indipendentemente dal linguaggio utilizzato. Daremo adesso una prima panoramica dei più diffusi linguaggi di programmazione.

Per illustrare il loro uso utilizzeremo i seguenti tre problemi:

1. Calcolo del prodotto scalare di due vettori;
2. Calcolo dei numeri di Fibonacci dove faremo una sorprendente scoperta;
3. Retta passante per due punti distinti P e Q del piano.

Una retta R nel piano reale $\mathbb{R}^2$ possiede una rappresentazione parametrica $R = \{p + tv | t \in \mathbb{R}\}$ con $p, v \in \mathbb{R}^2$ e $v \neq 0$.

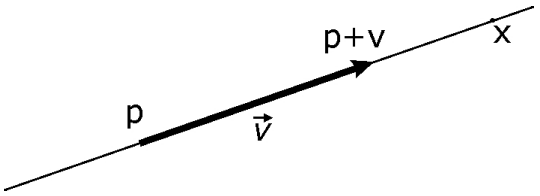

Figura 1.1: La rappresentazione parametrica di una retta in $\mathbb{R}^2$

Il vettore $a = (a_1, a_2) := (-v_2, v_1)$ che si ottiene ruotando v di 90° è anch'esso diverso da zero. a è un vettore magico e molto importante.

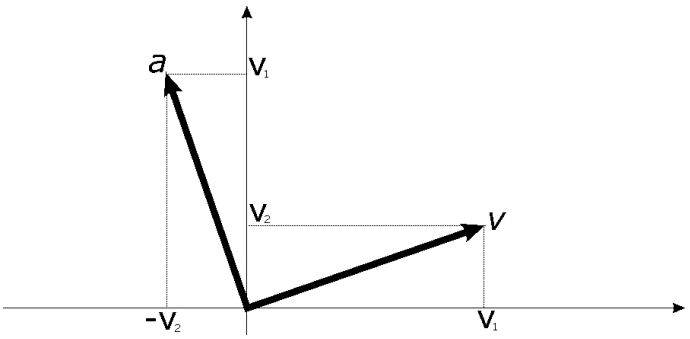

Figura 1.2: Il vettore "magico" a

Un punto $x = (x_1, x_2)$ appartiene alla retta se e solo se il vettore $x - p$ è parallelo a v cioè se e solo se $x - p$ è ortogonale ad a. I punti della retta quindi sono esattamente i punti che soddisfano l'equazione

$$a_1(x_1 - p_1) + a_2(x_2 - p_2) = 0$$

che può essere scritta anche nella forma

$$a_1x_1 + a_2x_2 = a_1p_1 + a_2p_2$$

Se p e q sono due punti distinti di $\mathbb{R}^2$ una rappresentazione parametrica e l'equazione della retta passante per questi due punti si trovano ponendo $v := q - p$. Possiamo quindi risolvere il compito 3 dell'introduzione 1.1 con il seguente algoritmo:

1. input di $p = (p_1, p_2)$ e $q = (q_1, q_2)$
2. $v = (v_1, v_2) = q - p$
3. $a = (a_1, a_2) = (-v_2, v_1)$
4. $c = a_1p_1 + a_2p_2$
5. output dell'equazione nella forma $a_1x_1 + a_2x_2 = c$ dove a_1, a_2, c sono i valori calcolati nei punti 3 e 4.

1.1 Alcune definizioni

Con i semplici programmi che vedremo in questo paragrafo si può naturalmente solo esemplificare la sintassi dei linguaggi presentati e non diventano visibili i loro punti forti o deboli (strutture di dati, modularità, velocità di compilazione o esecuzione, possibilità di una programmazione funzionale).

Una *procedura* in un programma è una parte del programma che può essere chiamata esplicitamente per eseguire determinate operazioni. In un *linguaggio macchina* una procedura o subroutines è un pezzo di codice di cui è noto l'indirizzo iniziale e da cui si torna indietro quando nell'esecuzione del codice vengono incontrate istruzioni di uscita. Una procedura può dipendere da parametri (detti anche argomenti); una procedura che restituisce (in qualche senso intuitivo) un risultato si chiama funzione e corrisponde nell'utilizzo alle funzioni o applicazioni della matematica. Una differenza importante è, nei linguaggi non puramente funzionali, che procedure e funzioni spesso eseguono delle operazioni i cui effetti non sono sempre rilevabili dal risultato; in tal caso la dimostrazione della correttezza di un algoritmo è più complicata di una normale dimostrazione matematica. Per questo oggi si cerca di sostituire il più possibile gli algoritmi procedurali tradizionali con tecniche che utilizzano solo funzioni e operazioni matematiche con funzioni (composizione di funzioni, formazione di coppie).

Una procedura o funzione in un programma si chiama ricorsiva se chiama se stessa. Il concetto di ricorsività è molto importante in matematica, informatica, logica e forse in molti comuni ragionamenti umani; se ad esempio

$f(n) = n!$ denota il fattoriale di un numero intero la relazione

$$f(n) = \begin{cases} 1 & \text{se } n = 0 \\ nf(n-1) & \text{se } n \neq 0 \end{cases}$$

mostra che in un programma potremmo usare una funzione ricorsiva per calcolare $n!$.

1.2 Un esempio: i numeri di Fibonacci

La successione dei numeri di Fibonacci è definita dalla ricorrenza di ordine due (servono due parametri iniziali):

$$F_0 = F_1 = 0$$

$$F_n = F_{n-1} + F_{n-2}$$

quindi si inizia con due volte 1, poi ogni termine è la somma dei due precedenti. Si ottiene quindi: $1, 1, 2, 3, 5, 8, 13, 21, ...$ Un programma iterativo in Perl per calcolare l'ennesimo numero di Fibonacci.

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

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

Se però adesso nella funzione `fibvisualizza` sostituiamo `fibit` con `fibric` ci accorgiamo che il programma si blocca dopo la serie dei primi venti numeri di Fibonacci, cioè che anche il pc più veloce non sembra 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 proviamo ad imitare il calcolo di F_{50} usando l'algoritmo in `fibric`.

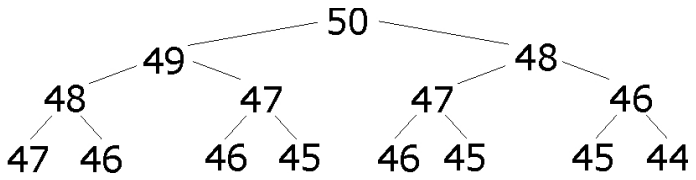

Figura 1.3: Grafico di una ricorsione doppia con complessità esponenziale

Si vede che F_{48} viene calcolato due volte, F_{47} tre volte, F_{46} cinque volte, F_{45} otto volte ecc. Si ha l'impressione che riappaia la successione di Fibonacci e infatti è così (si può dimostrare). Quindi un numero esorbitante di ripetizioni impedisce di completare la ricorsione perchè è noto che

$$\left| F_n - \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{n+1} \right| < \frac{1}{2} \qquad \forall n \geq 0$$

Da cui segue che questo algoritmo è di complessità esponenziale.

1.3 Il metodo del sistema di primo ordine

In analisi di impara che un equazione differenziale di secondo ordine può essere ricondotta a un sistema di due equazione di primo ordine. In modo simile possiamo trasformare la ricorrenza di Fibonacci in una ricorrenza del primo ordine ponendo

$$\begin{aligned} x_n &:= F_n \qquad \text{per } n \geq 0 \\ y_n &:= \begin{cases} F_{n-1} & \text{per } n \geq 1 \\ 0 & \text{per } n = 0 \end{cases} \end{aligned}$$

otteniamo il sistema

$$\begin{aligned} x_{n+1} &= x_n + y_n \quad \text{per } n \geq 0 \\ y_{n+1} &= x_{n+1} \\ x_0 &= 1 \\ y_0 &= 0 \end{aligned}$$

Per applicare questo algoritmo in ogni passo dobbiamo generare due valori; avremo quindi bisogno di una funzione che restituisca due valori numerici. Ciò in Perl dove la funzione dove la funzione può restituire come risultato una lista, è molto facile (in altri linguaggi è più complesso).

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

Per la visualizzazione dobbiamo ancora modificare la funzione `fibvisualizza` nel modo seguente:

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

Stavolta la risposta è di nuovo velocissima; il programma è ricorsivo, ma di primo ordine e non ci sono sovrapposizioni. Probabilmente `fibsist` è circa tre volte più lenta di `fibit` (ma essendo così veloce non ce ne accorgiamo) perchè funzioni ricorsive sono un po' più lente di quelle iterative e perchè, essendo passati a due dimensioni, facciamo un doppio calcolo.

1.4 Diagrammi di flusso

Algoritmi semplici possono essere espressi mediante diagrammi di flusso. Illustriamo questo metodo con un algoritmo per i numeri di Fibonacci.

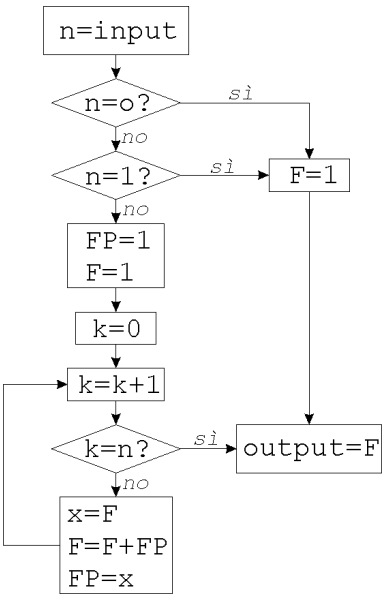

Figura 1.4: Esempio di diagramma di flusso per il calcolo dei numeri di Fibonacci

La tecnica dei diagrammi di flusso oggi viene usata pochissimo; è lenta e poco efficiente e non adatta alla rappresentazione di strutture di dati complesse. Inoltre non è facile controllare la correttezza dei programmi mediante diagrammi di flusso. Può essere utile (nei primi giorni) al programmatore principiante, per apprendere in modo visivo, alcuni meccanismi della programmazione procedurale.

1.5 I pseudolinguaggi

Talvolta, soprattutto nei libri sugli algoritmi, si usano *pseudolinguaggi*, cioè linguaggi che non sono veri linguaggi di programmazione e nemmeno formalmente definiti in tutti i dettagli. In un tale pseudolinguaggio il nostro diagramma di flusso potrebbe essere così tradotto:

```
n=input
if n=0 then goto uno
if n=1 then goto uno
FP=1
F=1
k=0
ciclo: K=k+1
if k=n then goto fine
x=F
F=F+FP
FP=x
goto ciclo
uno: F=1
fine: output=F
```

La rappresentazione di un algoritmo mediante un pseudolinguaggio è spesso adeguata quando si vuole studiare la complessità dell'algoritmo perchè in genere le istruzioni del pseudolinguaggio corrispondono in modo trasparente alle operazioni effettivamente eseguite dal processore. D'altra parte però, questo tipo di rappresentazione induce facilmente il programmatore a una concezione procedurale tradizionale degli algoritmi. Si noti che nel diagramma di flusso e nel pseudolinguaggio abbiamo usato il segno di uguaglianza sia per le assegnazioni (ad esempio k=k+1 ovviamente non può esprimere un uguaglianza, ma significa che il valore di k viene modificato e posto uguale a uno in più di quanto era prima) sia nei test uguaglianza. Per distinguere i due utilizzi, molti linguaggi (tra cui il C, il Perl e il Java) usano un doppio segno di uguaglianza (==) nei test di uguaglianza.

Capitolo 2

Il linguaggio macchina e assembler

2.1 Numeri esadecimali

Nei linguaggi macchina e assembler molto spesso si usano i numeri esadecimali o più correttamente la rappresentazione esadecimale dei numeri naturali, cioè la loro rappresentazione in base 16. Per $2789 = 10 * 16^2 + 14 * 16^1 + 5 * 16^0$ potremmo ad esempio scrivere $2789 = (10, 14, 5)_{16}$ In questo senso 10, 14 e 5 sono le cifre della rappresentazione esadecimale di 2789. Per poter usare lettere singole per le cifre si indicano le cifre da 10 a 15, mancanti nel sistema decimale, nel modo seguente: $10 := A$, $11 := B$, $12 := C$, $13 := D$, $14 := E$ e $15 := F$. In questo modo, adesso, possiamo scrivere $2789 = (AE5)_{16}$. In genere si possono usare anche indifferentemente le corrispondenti lettere minuscole. Si noti che $(F)_{16} = 15$ assume nel sistema esadecimale lo stesso ruolo del 9 nel sistema decimale. Quindi $(FF)_{16} = 255$ Un numero naturale n con $0 \leq n \leq 255$ si chiama byte. Un bit è invece uguale a 0 o a 1.

2.1.1 Esempi:

$0=(0)_{16}$	$2^7 = 128=(80)_{16}$
$14=(E)_{16}$	$203=(CB)_{16}$
$15=(F)_{16}$	$244=(F4)_{16}$
$16=(10)_{16}$	$255=(FF)_{16}$
$28=(1C)_{16}$	$2^8 = 256=(100)_{16}$
$2^5 = 32=(20)_{16}$	$2^{10} = 1024=(400)_{16}$
$2^6 = 64=(40)_{16}$	$2^{12} = 4096=(1000)_{16}$
$65=(41)_{16}$	$65535=(FFFF)_{16}$
$97=(61)_{16}$	$2^{16} = 65536=(10000)_{16}$
$127=(7F)_{16}$	$2^{32} = 4294967296=(100000000)_{16}$

Si vede da questa tabella che i byte sono esattamente quei numeri per i quali sono sufficienti al massimo due cifre esadecimali. Spesso la memo-

ria è organizzata in byte; il byte è allora il più piccolo 'pezzo' di memoria accessibile.

Nell'immissione di una successione di numeri esadecimali come unica stringa, spesso si pone uno zero all'inizio di quei numeri (da 0 a 9) che richiedono una cifra sola, ad esempio la stringa `0532A2014E586A750EAA` può essere usata per rappresentare la successione `(5,32,A2,1,4E,58,6A,75,E,AA)` di numeri esadecimali.

2.2 Un esempio nel linguaggio del processore 6502

Proviamo a scrivere nel linguaggio macchina del processore 6502 [1] l'algoritmo della moltiplicazione con 10 di un byte n. L'aritmetica nel 6502 avviene modulo 256 quindi affinchè il risultato sia corretto deve valere $0 \leq n \leq 25$. Siccome $10 * n = 2 * (2 * (2n)) + 2n$ possiamo usare il seguente algoritmo (detto *del contadino russo*):
p=2n
a=2p ... =4n
a=2a ... =8n
a=a+p ... =10n
In linguaggio macchina eseguiamo le seguenti operazioni: per n usiamo l'indirizzo esadecimale F0=240, per la variabile p l'indirizzo F1. Nel 6502 tutte le operazioni aritmetiche avvengono nell'accumulatore; quindi procederemo così:
a=n
p=2a
a=2p
a=2a
a=a+p

Le istruzioni del 6502 consistono di al massimo 3 bytes di cui il primo indica l'operazione e i rimanenti gli argomenti. Per prima cosa dobbiamo trasferire il contenuto di F0 nell'accumulatore mediante l'istruzione `A5 F0` (comando in linguaggio macchina del processore 6502 che trasferisce il contenuto di F0 nell'accumulatore). Disabilitiamo con `D8` la modalità decimale, poi con `0A` moltiplichiamo il contenuto a dell'accumulatore per 2 (in verità questa istruzione sposta i bit nella rappresentazione binaria di a di una posizione a sinistra e pone l'ultimo bit uguale a 0). Memorizziamo il nuovo valore in F1 con il comando `85 F1` (che incida il trasferimento del contenuto

[1] Il 6502 è un processore utilizzato da alcuni dei più famosi pc di tutti i tempi (Apple2, Commodore, Atari, ...) all'inizio degli anni ottanta. Il suo linguaggio macchina era elegante ed efficiente e molto istruttivo perchè era molto facile lavorare direttamente in memoria. Quei computer piuttosto lenti se programmati in basic si rivelavano vivacissimi se programmati in linguaggio macchina (o in assembler). Il processore ha ancora oggi i suoi appassionati e una pagina web `www.6502.org` offre esempi di codice, descrizione delle istruzioni e link ad altre pagine web.

dell'accumulatore nella casella con indirizzo F1), poi effettuiamo due raddoppi con il comando `0A 0A` (raddoppia l'accumulatore e raddoppia l'accumulatore) mettiamo il riporto uguale a 0 con il comando `18` e addizioniamo all'accumulatore il contenuto di F1 con il comando `65 F1`; infine trasferiamo il valore dell'accumulatore nella locazione di memoria F0 con `85 F0`. Il programma completo in linguaggio macchina a questo punto è[2]:

```
A5 F0   %pone a=n%
D8      %disabilita decimale%
0A      %pone a=2a%
85 F1   %pone p=a%
0A      %raddoppia a%
0A      %raddoppia a%
18      %pone riporto=0%
65 F1   %pone a=a+p%
85 F0   %pone n=a%
60      %termina il programma%
```

Che può essere inserito nella forma:
`A5 F0 D8 0A 85 F1 0A 0A 18 65 F1 85 F0 60`.
A questo punto, se A5 sta nella posizione 800 della memoria, si può eseguire il programma semplicemente scrivendo run 800.

2.3 I linguaggi assembler

Come abbiamo visto è possibile inserire direttamente un programma in linguaggio macchina in memoria. La difficoltà non consiste tanto nel ricordare i codici (almeno per il 6502 che ha relativamente poche istruzioni) perchè è facile ricordarseli dopo pochi giorni di pratica, ma piuttosto nella manutenzione del programma (modifiche, documentazione).

Se ad esempio si volesse inserire un nuovo pezzo di programma in un certo punto, bisogna non solo spostare una parte del programma (ciò è possibile mediante appositi comandi dal terminale), ma è necessario anche aggiustare tutti gli indirizzi dei salti e delle variabili. Per questa ragione sono stati introdotti i *linguaggi assembler*[3]. Così si chiamano i linguaggi le cui istruzioni fondamentali corrispondono esattamente alle istruzioni in linguaggio macchina, ma sono espresse in una forma non numerica più facile da ricordare e inoltre permettono l'utilizzo di nomi simbolici per gli indirizzi.

Gli assembler più recenti (ad esempio per i processori Intel) sono molto complessi e forniscono moltissime funzioni ausiliarie. Il programma in lin-

[2] ciò che sta scritto tra i simboli di % è un commento che è trascurabile ai fini operativi del programma.

[3] Il verbo *to assemble* in inglese significa appunto mettere insieme, assemblare

guaggio macchina precedentemente introdotto viene così tradotto in assembler nel seguente modo:

```
LDA $F0     %load accumulator%
CLD         %clear decimal%
ASL         %arithmetic shift left%
STA $F1     %store accumulator%
ASL ASL
CLC         %clear carry%
ADC $F1     %add with carry%
STA $F0
```

Utilizzando nomi simbolici per gli indirizzi potremmo scrivere:

```
LDA
N
CLD
ASL
STA
P
ASL
ASL
CLC
ADC
P
STA
N
```

Capitolo 3

Il basic

Il Basic è stato uno dei primi e più popolari linguaggi di programmazione per pc. Concettualmente esprime una programmazione orientata ai diagrammi di flusso ed ha un po' gli stessi svantaggi. Anche dal punto di vista della sintassi è molto limitato: i nomi delle variabili possono avere solo due lettere, non ci sono variabili locali e non ci sono veri sottoprogrammi o funzioni. Ci sono molti dialetti e quindi il linguaggio è anche poco portabile. Alcuni dialetti superano in parte le limitazioni del basic classico, ma sono molto complicati e più difficili da imparare dei linguaggi professionali come C, C++, Perl e Java. Risolviamo ora i nostri tre compiti iniziali in basic classico.

Prodotto scalare di due vettori 3-D:

```
10 dim a(3)
20 dim b(3)
30 for i=0 to 3
40 a(i)=i+1
50 next i
60 for i=0 to 3
70 b(i)=(i+1)*(i+1)
80 next i
100 p=0
110 for i=0 to 3
120 p=p+a(i)*b(i)
130 next i
140 print p
```

oppure

```
...
30 for i=0 to 3
40 a(i)=i+1
50 b(i)=(i+1)*(i+1)
```

```
...
80 next i
...
```

Numeri di Fibonacci:

```
10 print "n = "
12 input n
20 if n=0 then 200
30 if n=1 then 200
40 fp=1
50 f=1
60 k=0
70 k=k+1
80 if k=n then 210
90 x=f
100 f=f+fp
110 fp=x
120 goto 70
200 f=1
210 print f
```

Retta passante per p e q:

```
10 print "p1= "
12 input p1
20 print "p2= "
22 input p2
30 print "q1= "
32 input q1
40 print "p2= "
42 input q2
50 v1=q1-p1 : v2=q2-p2
60 a1=-v2 : a2=v1
70 c=a1*p1+a2*p2
80 print a1; "x+ "; a2;
90 print "y= "; c
```

Il punto e virgola nei comandi di stampa non indica la fine di un comando ma un output senza passaggio alla nuova riga.

Capitolo 4

Fortran

Il Fortran è insieme al Lisp uno dei più vecchi linguaggi di programmazione ancora in uso. Anch'esso induce a una programmazione strettamente procedurale e ha molte limitazioni; è largamente diffuso però in ambiente ingegneristico e numerico nonostante i molti difetti soprattutto perchè esistono vaste (e preziose) librerie di software. Se il programma sorgente viene scritto in un file alfa.f sotto Linux il comando
`f77 -ffree-form alfa.f -o alfa`
crea un file eseguibile alfa.

Prodotto scalare

```
c la lettera c a inizio riga indica un commento
real a(4),b(4)
integer i,p
do 10 i=1,4
c Basic 0...3, Fortran 1...4
a(i)=i
10 continue
do 20 i=1,4
b(i)=i*i
20 continue
p=0
do 30 i=1,4
p=p+a(i)*b(i)
30 continue
write(*,*) p
end
```

Retta passante per p e q

```
real p1,p2,q1,q2,v1,v2,a1,a2,c
write(*,20) 'p1,p2,q1,q2'
read(*,10) p1,p2,q1,q2
v1=q1-p1
v2=q2-p2
a1=-v2
a2=v1
c=a1*p1+a2*p2
write(*,30) a1,'x+ ',a2,'y=',c
10 format(f9.2)
20 format(a)
30 format(f9.2,a4,f9.2,a4,f9.2)
end
```

Capitolo 5

Pascal

Il pascal è stato sviluppato negli anni 70 da Niklaus Wirth. E' un linguaggio procedurale in cui però molta importanza è attribuita ai tipi di dati, anche complessi, e alla programmazione strutturale. Wirth, professore di informatica a Zurigo, è anche autore di un famoso libro sugli algoritmi e strutture di dati. Il linguaggio Pascal è un po' simile, ma meno potente e meno popolare del C e quindi meno popolare tra i programmatori; è invece spesso insegnato nei corsi universitari.

Un buon compilatore di Pascal libero che funziona anche con Linux si trova ad esempio all'indirizzo:
gd.tuwien.ac.at/languages/pascal/fpc/www/disc/linux/rpm.
Il file fpc-1.0.10-0.i386.rpm può essere installato con il comando rpm -Uvh fpc-1.0.10-0.i386.rpm (ed eliminato con rpm -e fpc; usare invece man fpc per le istruzioni sull'uso del compilatore).

Per un programma semplice contenuto nel file alpha.pas il comando fpc alpha crea l'eseguibile alpha e un file oggetto alpha.o (che può essere rimosso). Dallo stesso sito si può prelevare anche la documentazione completa che viene installata in /usr/doc.

5.1 Un programma in Pascal per il calcolo del prodotto scalare

Vediamo qui di seguito un programma in Pascal per il calcolo del prodotto scalare:

```
programs scalare,
type vettore4=array[1..4] of real;
var i:integer; a,b:vettore4; p:real;

function scal (x,y:vettore4):real;
var p:real; i:integer;
```

```
begin
p:=0;
for 1:=1 to 4 do p:=p+x[i]*y[i];
scal :=p;
end;

begin
for i:=1 to 4 do a[i]:=i;
for i:=1 to 4 do b[i]:=i*i;
p:=scal(a,b);
writeln(p:6:2); **L'equivalente di format(f6.2) in Fortran**
end.
```

Si vede la meticolosa dichiarazione delle variabili e la forte strutturazione imposta dal programma già in un esempio così semplice. Si noti anche che a differenza di altri linguaggi l'istruzione di assegnazione usa il simbolo := invece del seplice = imitando la notazione che in matematica si usa per indicare l'uguaglianza per definizione. Come in C la i-esima componente di un vettore a viene indicata con $a[i]$.

Capitolo 6

Delphi

E' un ambiente di sviluppo abbastanza diffuso per la programmazione orientata agli oggetti in Pascal sotto Windows.

Capitolo 7

C

Un programma in C o C++ in genere viene scritto in più files che conviene raccogliere nella stessa directory. Avrà anche bisogno di files che fanno parte dell'ambiente di programmazione o di una raccolta di funzioni sviluppate in precedenza. Tutto insieme si chiama *progetto*. I files del progetto devono essere compilati e collegati (*linked*) per ottenere un file eseguibile (detto spesso *applicazione*).

Il programma in C, C++ costituisce il codice sorgente (*source code*) di cui la parte principale è contenuta in files che portano l'estensione .c, mentre un'altra parte che comprende soprattutto le dichiarazioni generali, è contenuta in files che portano l'estensione .h (*header* ossia intestazione).

Il C può essere considerato un linguaggio macchina universale le cui operazioni hanno effetti diretti in memoria anche se la locazione completa degli indirizzi non è nota al programmatore. Il compilatore deve quindi sapere quanto spazio deve riservare alle variabili (e solo a questo serve la dichiarazione del tipo delle variabili) e quanti e di quale tipo sono gli argomenti delle funzioni.

I comandi di compilazione sotto Unix possono essere battuti dalla *shell*, ma ciò deve avvenire ogni volta che il codice sorgente è stato modificato e quindi consuma molto tempo e sarebbe difficile evitare errori. In compilatori commerciali, soprattutto su altri sistemi, queste operazioni vengono spesso eseguite attraverso un'interfaccia grafica; sotto Unix normalmente questi comandi vengono raccolti in un cosiddetto *makefile* che viene successivamente eseguito mediante il comando make (sostanzialmente il make è un programmino ausiliario che permette di ottimizzare il lavoro).

Tra le dichiarazioni del C e la specifica del tipo di una variabile in Basic o Perl esiste una sostanziale differenza. In C le dichiarazioni hanno essenzialmente il solo scopo di indicare la quantità di memoria che una variabile occupa, ma non vincolano il programmatore a usare quella parte della memoria in un modo determinato.

Il codice

```
double x; [1]
sprintf(&x,'Alberto');
puts(&x); [2]
```

Il codice provocherà su molti compilatori un avvertimento (*warning*) perchè l'uso dello spazio occupato dalla variabile reale x per contenere una stringa è insolito, ma non genera un errore. Se con

```
double x;
sprintf((char*)&x),Alberto); [3]
puts((char*)&x); [4]
```

si informa il compilatore delle proprie intenzioni non protesterà più. In altri linguaggi, invece, i tipi servono per definire le operazioni da definire su quelle variabili mentre la gestione della memoria è compito dell'interprete.

7.1 Un esempio

Presentiamo adesso un programma completo in C che contiene tre funzioni che corrispondono ai problemi posti nel capitolo 1.

```
01 // = commento
02 // alfa.c
03 # include <stdio.h>
04 # include <stdlib.h>

05 void fibonacci(), retta(), scalare();
06 int main();

07 ///////

08 int main()
09 {scalare(); fibonacci(); retta();
10 exit(0);}

11 void fibonacci()
12 {char input[40]; int n,k; double a,b,c;
13 printf("Inserisci n: ");
14 fgets(input,38,stdin);
15 n=atoi(input);
16 if (n<=1) {a=1; goto fine;}
17 for (a=b=1,k=2; k<=n; k++)
```

[1] Il comando `double` crea variabili a 12 byte, `integer` invece le crea a 4 byte

[2] Il comando `puts` legge fino al primo carattere `/0` che rappresenta la fine ossia il nuovo paragrafo

[3] la dichiarazione `&x` associa alla variabile il suo indirizzo

[4] il comando `char` va sopra alla cosa scritta in memoria subito dopo a x

```
18 {c=a; a+=b; b=c;}
19 fine: printf("%.0f\n",a);}

20 void retta()
21 {double p1,p2,q1,q2,v1,v2,a1,a2,c;
22 p1=input("p1"); p2=input("p2");
23 q1=input("q1"); q2=input("q2");
24 v1=q1-p1; v2=q2-p2;
25 a1=-v2; a2=v1;
26 c=a1*p1+a2*p2;
27 printf("%.2fx+%.2fy=%.2f\n",a1,a2,c);}

28 void scalare()
29 {double a[4],b[4],p; int i;
30 for(i=0; i<4; i++)
31 {a[i]=i+1; b[i]=(i+1)*(i+1);}
32 for(p=0,i=0;i<4;i++) p+=a[i]*b[i];
33 printf("%9.2f\n",p);}

34 double input (char *A)
35 {char input[40];
36 printf("Inserisci %s: ",A);
37 fgets(input,38,stdin);
38 return atof(input);}
```

7.2 Help:

A riga 03 e 04 ci sono istruzione da mandare al compilatore (copia tutto il contenuto del file in questa posizione). A riga 06 si dichiara la funzione `main()`: questo comando non fa nulla, dichiara e basta. Abbiamo così prima dichiarato la funzione `main()` in riga 06 e poi l'abbiamo *usata* in riga 08. In riga 09 `scalare()` chiama la funzione scalare senza nessun parametro (e così anche `fibonacci()` e `retta()`). A riga 14 `fgets(input,38,stdin)` prende un input e lo mette nella variabile input fino al 38esimo carattere; stdin (=STanDardINput) sta ad indicare che l'input arriva da tastiera. A riga 15 `n=atoi(input)` trasforma la stringa di testo in numero integer con il comando atoi (= Ascii TO Integer). A riga 19 lo zero indica che l'output deve avere zero cifre decimali. A riga 34 `double input (char *A)` è un puntatore cioè una variabile a indirizzo variabile unita al tipo. Oltre ai puntatori ci sono le stringhe che sono vettori di caratteri e i vettori che sono variabili a indirizzo fisso unite al tipo. `char *a` punta ad `a`, mentre `char *a+1` punta ad `a+1`; `int *a+1` punta 4 byte + avanti di `a` perchè `a` occupa 4 spazi in memoria. `((char*)a)+1` punta alla casella di memoria successiva al primo

valore dopo a (vedi figura 7.1). Per i puntatori usiamo per abitudine le lettere maiuscole. A riga 38 `return atof(input)` atof (= Ascii TO Floating).

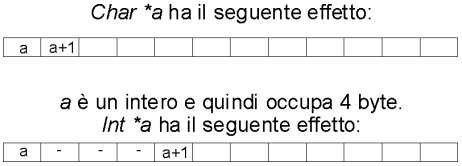

Figura 7.1: Puntatori a caratteri e ad interi

Il commento nella prima riga indica il nome del file; ciò è utile nella stampa. In questo caso semplice possiamo compilare direttamente dalla shell con i comandi gcc -c alfa.c (gcc = Gnu C Compiler) gcc -o alfa alfa.o. Il primo comando crea il file oggetto alfa.o mentre il secondo il programma eseguibile alfa.

Il manuale di riferimento più completo per il C è: S. Harbison, G. Steele *C. A reference manual* Prentice-Hall, 1995.

Capitolo 8

C++

Nato e cresciuto insieme a Unix e perfezionato dai migliori programmatori del mondo il C è probabilmente il linguaggio più su misura per il programmatore. Come Unix invita a una programmazione che utilizza molte piccole funzioni che il programmatore esperto riesce a scrivere rapidamente e che possono essere raccolte in un modo estremamente efficiente. Il programma del paragrafo 7.1 apparentemente è il più complicato di quelli visti finora per i nostri tre problemi; tra l'altro abbiamo scritto un'apposita funzione (`input`) per l'immissione di un numero dalla tastiera. Uno dei vantaggi del C è che queste funzioni ausiliarie possano essere scritte e gestite con grande facilità. Il C pur rimanendo fedele al lavoro diretto in memoria permette la creazione di dati molto complessi.

Molti programmatori in C riescono quindi a realizzare nei loro programmi i concetti della programmazione orientata agli oggetti. Il C++ sviluppato più tardi del C mette più chiaramente in evidenza questi concetti tra cui il più importante è quello di classe. Una classe nel C++ possiede componenti che possono essere sia dati che funzioni (o metodi). Se alfa è una classe ogni elemento di tipo alfa si chiama oggetto della classe.

Così

```
class vettore {public: double x,y,z;
double lun()
{return sqrt(x*x+y*y+z*z);}};
```

definisce una classe i cui oggetti rappresentano vettori tridimensionali e contengono, oltre alle tre componenti, anche la funzione che calcola la lunghezza del vettore. Le componente vengono usate come nel seguente esempio:

```
void prova()
{vettore v;
v.x=v.y=2; v.z=1;
printf("%.3f\n",v.lun());}
```

L'indicazione `public:` (da non dimenticare) fa in modo che le componenti che seguono siano visibili anche al di fuori della classe. Con `private:` invece si

ottiene il contrario. `public:` e `private:` possono apparire nella stessa classe; l'impostazione di default è `private:`. Una tipica classe per una libreria grafica potrebbe essere

```
class rettangolo {public: double x,y,dx,dy;
void disegna(), sposta(), sposta(double,double);};
```

dove stavolta all'interno della classe appaiono solo le dichiarazioni dei metodi che vengono invece definiti all'esterno nella forma

```
void rettangolo::disegna()
{...}
```

8.1 Storia, bibliografia e osservazioni

Negli ultimi anni il C++ si è arricchito di moltissime nuove funzioni comprese soprattutto nella libreria standard (standard library con la standard template library, STL). Le funzioni generiche che da un lato allargano di molto le funzionalità del C++ rendono però il linguaggio anche estremamente e difficile da imparare. Alcuni testi sono:

N. Josuttis. *The C++ standard library.* Addison-Wesley, 1999.
H. Schildt. *C++, The complete reference.* MacGraw-Hill, 2003.
B. Stroustrup[1]. *C++.* Addison-Wesley Italia, 2000.
S. Meyers. *Affective C++.* Addison-Wesley.

E' difficile dire se preferire il C o il C++; i due linguaggi sono altamente compatibili (nel senso che molto spesso i programmi scritti in C possono essere direttamente utilizzati per il C++), il C++ è in un certo senso più confortevole, il C più adatto alla programmazione di sistema sotto Unix.

[1] B. Stroustrup è anche l'inventore del C++

Capitolo 9

Java

Java è un linguaggio di programmazione orientato agli oggetti che eredita alcune sue caratteristiche dal C++.

Uno degli aspetti più importanti della programmazione orientata agli oggetti è la comunicazione: in un certo senso si informano tra di loro sulla propria posizione e sul proprio stato, sulle attività e funzioni. Questa comunicazione nel programma si riflette poi in una comunicazione tra le persone coinvolte nel progetto. Per questo la programmazione orientata agli oggetti è particolarmente adatta al lavoro di gruppo. Essa è estroversa mentre la programmazione ricorsiva e creativa è, almeno in parte più introversa.

Bisogna dire che talvolta (ovviamente anche per ragioni commerciali) l'importanza di nuovi paradigmi di programmazione, in particolare della programmazione orientata agli oggetti, e di strutture innovative nei linguaggi viene spesso esagerata. Un buon programmatore in C riesce facilmente a creare quasi tutte queste strutture e il tempo perso a imparare il linguaggio di moda potrebbe essere usato per migliorare la propria biblioteca di funzioni.

A differenza del C++ nel Java le funzioni possono essere definite soltanto come componenti di una classe e come d'uso nella programmazione orientata agli oggetti, vengono allora dette metodi della classe. La classe `string` possiede ad esempio un metodo `length` che restituisce la lunghezza della stringa per cui viene chiamato.

```
String nome = Maria Stuarda;
n = nome.length();
```

Le funzioni matematiche sono per la maggior parte metodi della classe Math e vengono chiamate ad esempio così:

```
y=Math.sin(x);
y=Math.sqrt(x);
y=Math.log(x);
y=Math.exp(x);
```

9.1 Un esempio

Segue un programma completo per i numeri di Fibonacci; come si vede le operazioni di input e output, a causa dei molti controlli e della rigidità dei tipi di dati, piuttosto difficoltose.

```
import java.io.IOException;
class algoritmi
{public static void main (String parameter[])}
{fibonacci();}

public static void fibonacci()
{int n=0,k,n,a,b,c; byte input[]=new byte[40];
string stringa;
System.out.print('Inserisci n: ');
try {m=System.in.read(input,0,38);
stringa=new String(input,0,m-1);
n=Integer.parseInt(stringa);}
catch(IOException e){}

if(n<=1) a=1; else
{for(a=b=1,k=2,k==n;k<=n;k++)
{c=a,a+=b;b=c;}}
System.out.println(a);}}
```

Questo programma va scritto in un file algoritmi.java (il nome del file deve corrispondere al nome della classe in esso definita) da cui si ottiene con javac algoritmi.java il file algoritmi.class che contiene il bytecode (cfr capitolo 1) che può essere eseguito con java algoritmi.

9.2 Osservazioni

Molti dei paradigmi di comunicazione contenuti in java sono sotto Unix divise tra il sistema operativo e il linguaggio C. Il programmatore in C con Unix può liberamente utilizzare i meccanismi avanzati del sistema operativo e quindi non necessita che essi siano contenuti nel linguaggio.

Java offre invece, come molti altri linguaggi di programmazione, una gestione automatica della memoria. Ciò, almeno in certe circostanze, può essere indubbiamente un punto in favore di java.

I concetti di programmazione orientata agli oggetti in java sono sicuramente più perfezionati rispetto al C, C++, sono però anche più rigidi e lasciano meno libertà al programmatore. La facilità con cui nel C si possono gestire grandi raccolte di programmi con numerosissimi files è unica e

permette al programmatore ben organizzato una programmazione modulare snella ed efficiente.

Unix possiede una potentissima libreria grafica anch'essa direttamente utilizzabile in C. Le notevoli capacità grafiche offerte anche da Java hanno però il vantaggio di essere indipendenti dalla piattaforma e di permettere quindi lo sviluppo di programmi ad intefaccia grafica utilizzabili sotto più sistemi operativi.

Capitolo 10

Python

Molto leggibile negli esempi più semplici, usa l'indentazione per la strutturazione. Come il Perl permette l'assegnazione contemporanea anche degli scambi: `(a,b)=(b,a)` invece di `c=a, a=b, b=c`. Nonostante l'apparente semplicità, in genere, non suscita l'entusiasmo degli studenti ed è probabilmente uno di quei linguaggi (ma VisualBasic è peggio) dove chi inizia non impara a programmare seriamente perchè troppi meccanismi rimangono nascosti.

```
#! /usr/bin/pyton
# alfa

import string

def fibonacci():
  n=string.atoi(raw_input("Inserisci n: ");
  if n<=1 : f=1
  else:
    a=b=1
    for k in range (2,n+1): a,b=a+b,a
  print a

def retta():
  p1=string.atof(raw_input("Inserisci P1: ")
  p2=string.atof(raw_input("Inserisci P2: ")
  q2=string.atof(raw_input("Inserisci q2: ")
  q1=string.atof(raw_input("Inserisci q1: ")
  v1=q1-p1; v2=q2-p2; a1=-v2; a2=v1
  c=a1*p1+a2*p2
  print "%6.2fx + %6.2fy = %6.2f" %(a1,a2,c)

def scalare():
```

```
  a=(1,2,3,4); b=(1,4,9,16)
  p=0
  p in range(0,4): p=p+a[i]*b[i]
  print p

fibonacci() retta() scalare()
```

Capitolo 11

Perl

Al Perl sarà dedicata la maggior parte del corso per cui ci limitiamo qui a risolvere i nostri tre problemi ancora in un unico file direttamente eseguibile dalla shell di Unix (non affronteremo Perl/Tk che è un'estensione per la grafica in Perl).

```
01 #! /usr/bin/perl
02 # alfa(nome del file)

03 fibonacci();retta();scalare();

04 sub fibonacci{my $n=input("Inserisci n: ")};
05 my($a,$b,$k); if($n<=1){$a=1;goto fine}
06 for ($a=$b=1,$k=2; $k<=$n; $k++)
07 {($a,$b)=($a+$b,$a)}
08 fine: print"$a\n"}

09 sub input{print shift; my $a=<stdin>;
10 chop $a; $a}

11 sub retta{my $punti=input("Inserisci p1, p2, q1, q2: ");
12 my($p1,$p2,$q1,$q2)=split(/\s+/,$punti);
13 my($a1,$a2,$v1,$v2,$c);
14 $v1=$q1-$p1; $v2=$q2-$p2;
15 printf("%2fx + %2fy = %2f\n", $a1,$a2,$c)}

16 sub scalare{my @a=(1,2,3,4); my @b=(1,7,9,16);
17 my $p; for $k(0..3) {$p+=$a[$k]*$b[$k]}
18 print "$p\n"}
```

11.1 Help

A riga 03 si nota che a differenza del Phyton in Perl le funzioni possono essere anche chiamate prima di essere definite. A riga 04 `my` significa che `n` è una variabile locale non visibile da fuori. A riga 08 `\n` manda a capo. A riga 10 con il comando `chop $a` taglio l'ultimo carattere dell'input, ossia il carattere di nuovo paragrafo (a capo). A riga 12 il comando `split` spezza l'input allo spazio.

Capitolo 12

Forth e PostScript

Il Forth venne inventato all'inizio degli anni 60 da Charles Moore per piccoli compiti industriali, ad esempio il pilotaggio di un osservatorio astronomico. E' allo stesso tempo un linguaggio semplicissimo ed estremamente estendibile (dipende solo dalla pazienza del programmatore quanto voglia accrescere la biblioteca delle sue funzioni, o meglio macro-istruzioni). Viene usato nel controllo automatico, nella programmazione di sistemi, nell'intelligenza artificiale. Un piccolo e ben funzionante interprete è pfe.

Uno stretto parente e discendente del Forth è il PostScript, un sofisticato linguaggio per stampanti che mediante un interprete (il diffuso Ghostscript) può essere utilizzato anche come linguaggio di programmazione per altri scopi. Forth e PostScript presentano alcune caratteristiche che li distinguono dagli altri linguaggi di programmazione.

1. Utilizzano la notazione polacca inversa (RPN, Reverse Polish Notation) come alcune calcolatrici tascabili (della HP ad esempio). Ciò significa che gli argomenti precedono gli operatori: invece di `a+b` si scrive ad esempio si scrive `a b +` (in PostScript si scrive `a b add`) e quindi `(a+3)*5+1` diventa `a 3 add 5 mul 1 add`. Ciò comporta una notevole velocità di esecuzione perchè i valori vengono semplicemente prelevati da uno stack e quindi, benchè interpretati, Forth e PostScript sono linguaggi veloci con codice sorgente molto breve.

2. Entrambi i linguaggi permettono e favoriscono un uso estensivo di macro-istruzioni (abbreviazioni) che nel PostScript possono essere addirittura organizzate su più dizionari (fornendo così una via alla programmazione orientata agli oggetti in questi linguaggi apparentemente quasi primitivi). Tranne pochi simboli speciali, quasi tutte le lettere possono far parte dei nomi degli identificatori, quindi se, ad esempio, anche in PostScript volessimo usare `+` e `*` al posto di `add` e `mul` basterebbe definire `/+{add} def /*{add} def`.

3. In pratica non esiste distinzione tra procedura e dati. Tutti gli oggetti sono definiti mediante abbreviazioni e la programmazione acquisisce un carattere fortemente logico-semantico. Sul sito di Adobe, www.adobe.com, si trovano manuali e guide alla programmazione in PostScript.

Lo stack è una pila[1] è una delle più elementari e importanti struttura di dati. Uno stack è una successione di dati in cui tutte le inserzioni, cancellazioni ed accessi avvengono ad una sola estremità. L'interpreti e i compilatori di tutti i linguaggi di programmazione utilizzano uno o più stack per organizzare le chiamate annidate di funzioni; in questi casi lo stack contiene soprattutto gli indirizzi di ritorno, i valori di parametri che dopo un ritorno devono essere ripristinati, le variabili locali di una funzione.

Descriviamo brevemente l'esecuzione di un programma in PS (o Forth). Consideriamo la sequenza:
`40 3 add 5 mul`.
Assumiamo che l'ultimo elemento dello stack degli operandi sia `x`. L'interprete incontra prima il numero 40 e lo mette sullo stack degli operandi, poi legge 3 e pone anche questo numero sullo stack (degli operandi quando non specificato altrimenti). In questo momento il contenuto dello stack è `... x 40 3`.

Successivamente l'interprete incontra l'operatore `add` che richiede due argomenti che l'interprete preleva dallo stack; adesso viene calcolata la somma $40 + 3 = 43$ e posta sullo stack i cui ultimi elementi sono così `... x 43`.

L'interprete va avanti e trova 5 e lo pone sullo stack che contiene così `... x 43 5`. Poi trova di nuovo un operatore (`mul`), preleva i due argomenti necessari dallo stack su cui ripone il loro prodotto (43*5=215). Il contenuto dello stack degli operandi adesso è `... x 215`.

12.1 Macroistruzioni

In Postscript invece di funzioni si definiscono macroistruzioni che vengono definite secondo la sintassi:
`/abbreviazione significato def`.
Se il significato è un operatore eseguibile, bisogna racchiudere le operazioni tra parentesi graffe come abbiamo fatto nel paragrafo precedente per `add` e `mul` per impedire che vengano eseguite già la prima volta che l'interprete le incontra cioè nel momento in cui legge l'abbreviazione. Quando il significato invece non è eseguibile, non bisogna mettere parentesi graffe, come ad esempio

```
/e 2.71828182 def
/pi 3.14159265 def
```

[1] dall'inglese stack

12.2 Diagrammi di flusso per lo stack

Vogliamo scrivere una macroistruzione per la funzione f definita da $f(x) = 3x^2 + 5x + 1$. Per tale compito in PostScript (e in Forth) si possono usare diagrammi di flusso per lo stack simili ai diagrammi per altri linguaggi visti in precedenza dove però adesso indichiamo ogni volta gli elementi più a destra dello stack accanto ad ogni istruzione. Utiliziamo due nuove istruzioni: `dup` che duplica l'ultimo elemento dello stack (vedremo subito perchè) `exch` che scambia gli ultimi elementi più a destra dello stack. Assegnamo inoltre di avere definito gli operatori `+` e `*` invece di `add` e `mul` come in precedenza. La macroistruzione che corrisponde a questo diagramma di flusso è: `/f {dup dup * 3 * exch 5 * + 1 +} def`

x	ultimo elemento dello stack
dup	x, x
dup	x, x, x
$*$	x, x^2
3	x, x^2, 3
$*$	x, $3x^2$
exch	$3x^2$, x
5	$3x^2$, x, 5
$*$	$3x^2$, $5x$
$+$	$3x^2 + 5x$
1	$3x^2 + 5x$, 1
$+$	$3x^2 + 5x + 1$

12.3 If e Ifelse

Illustriamo l'uso di If e Ifelse con due esempi; `a m resto` calcola il resto di a mod m anche per $m < 0$ utilizzando la funzione `mod` del PostScript che dà il resto corretto invece solo per $m > 0$, mentre `n fatt` è il fattoriale di \verbn".

```
/resto {2 dict begin /m exch def /a exch def
   m 0 lt {/a a neg def} if a m mod end} def

/fatt {1 dict begin /n exch def
   n 0 eq {1} {n 1 - fatt n *} ifelse end} def
```

Per trasformare un numero in una stringa si può usare
`/stringa-numerica {20 string cvs} def`
Esempi da provare:
`2 6 moveto 117 40 resto stringa-numerica show`
`3 6 moveto 8 fatt stringa-numerica show`

Qui abbiamo usato lo stack dei dizionari; il PostScript permette infatti una gestione manuale di variabili locali mediante dizionari (*dictionary*) che possono essere annidati perchè organizzati tramite un apposito stack con

```
4 dict begin /x 1 def /y 2 def /z 3 def /w (Rossi) def
 ...
end
```

Viene così creato un dizionario di almeno quattro voci (il cui numero viene comunque automaticamente aumentato se vengono definite più voci). Fra `begin` e `end` tutte le abbreviazioni si riferiscono a questo dizionario se in esso si trova una tale voce (altrimenti il significato viene cercato nel prossimo dizionario nello stack dei dizionari); con `end` tali abbreviazioni perdono la loro validità.

Come già osservato il PS è soprattutto un linguaggio per stampanti che permette di creare grafici raffinati di alta qualità.

```
0.05 setlinewidth   %per impostare lo spessore delle linee
28.3 28.3 scale   %scala 1cm=1 (dipende dalla scala)
0 0 move to 5 4 lineto stroke
/rosso {1 0 0 setrgbcolor} def   %definiamo il colore rosso
/nero {0 0 0 setrgbcolor} def    %definiamo il colore nero
rosso 7 5 move to 4 5 3 0 360 arc stroke
% (4,5) è il centro, 3 è il raggio, 0 360 vuol dire
% di disegnare da 0 a 360 gradi.

% per disegnare cerchi con parametri variabili:
/cerchio {3 dict begin /x exch def /y exch def /x exch def
gsave newpath x r add y moveto x y r 0 360 arc stroke grestore end} def

% gsave e grestore per salvare temporaneamente lo stato della grafica
% con fill al posto di stroke mi fa il cerchio pieno!
nero 2 0 2 cerchio
showpage
```

Capitolo 13

Lisp

Il lisp, creato alla fine degli anni 50 da John McCarthy, è ancora oggi uno dei più potenti linguaggi di programmazione. Funzioni e liste sono gli elementi di questo linguaggio e funzioni possono essere sia argomenti che valori di altre funzioni. Per questo in Lisp (e il Perl) si possono scrivere applicazioni non realizzabili in C. Il Lisp è stato dagli inizi il linguaggio preferito dell'intelligenza artificiale. E' un linguaggio difficile che non viene in nessun modo incontro al programmatore e per questo è poco diffuso; sembra però che i programmatori in Lisp guadagnino il doppio dei programmatori in C (newsgroup di Lisp: comp.lang.lisp). Tutto ciò che si può fare in Lisp lo si può fare anche in Perl. Entrambi contengono il λ-calcolo e ciò costituisce la differenza fondamentale con altri linguaggi. Ci sono parecchi dialetti del Lisp: il più importante è il CommonLisp considerato la norma del Lisp, mentre lo Scheme è una versione minore didattica; in Elisp, un dialetto abbastanza simile al Commonlisp è programmato e programmabile l'editor Emacs, uno dei più formidabili strumenti informatici. Un miniprogramma in Lisp:

```
#! /usr/local/bin/clisp
(defun cubo(x) (* x x x))
(format t "~a~%" (cubo3))
```

Si noti che gli operatori precedono gli operandi. Siccome, a differenza del PostScript, gli operatori (ad esempio `*`) non hanno un numero di argomenti fisso, bisogna usare le parentesi. Format è un'istruzione di output abbastanza simile al `printf` del C.

13.1 Il λ-calcolo

Siccome bisogna distinguere tra la funzione f e i suoi valori $f(x)$ introduciamo per la funzione che manda x in $f(x)$ il simbolo $O_x f(x)$. Ad esempio $O_x \sin(x^2+1)$ è la funzione che manda x in $sin(x^2+1)$. E' chiaro che ad

esempio $O_x x^2 = O_y y^2$ (per la stessa ragione per cui $\sum_{i=0}^{n} a_i = \sum_{j=0}^{n} a_j$) mentre $O_x xy \neq O_y yy$ (così come $\sum_{j=0}^{n} a_{ij} \neq \sum_{j=0}^{n} a_{jj}$) e come non ha senso l'espressione $\sum_{i=0}^{n} \sum_{i=0}^{n} a_{ii}$ così non ha senso $O_x O_x x$.

Siccome in logica si scrive $\lambda x.f(x)$ invece di $O_x f(x)$, la teoria molto complicata di questo operatore si chiama λ-calcolo. Su esso si basano il Lisp e molti concetti dell'intelligenza artificale.

$$\lambda - calcolo = \text{teoria delle regole di autoriferimento}$$

13.2 Osservazione ed esempi

Il Lisp è un linguaggio funzionale; funzioni possono quindi essere definite all'interno di altre funzioni e restituite come risultato.

Consideriamo la funzione $add = O_n O_x x + n$ che in Lisp è programmata da: `(defun add(n) (function (lambda(x) (+ x n))))`.
Possiamo utilizzare questa funzione in

```
(format t "~a~%",(funcall(add 20) 8 ))
```

Possiamo però fare lo stesso con una sintassi anche più semplice in Perl

```
sub add {my $n=shift; sub {my $x=shift; $x+$n}}
print add(20)-->(8);
```

Il Lisp è estremamente complesso; ha più di mille funzioni fondamentali e la spesa di tempo necessario per impararlo è formidabile.

Capitolo 14

Prolog

Durante gli anni 70 e i primi anni 80 il Prolog divenne popolare in Europa e in Giappone per applicazioni nell'ambito dell'intelligenza artificiale. È un linguaggio dichiarativo, negli intenti della programmazione logica il programmatore deve solo descrivere il problema e indicare le regole di base della soluzione; ci pensa il sistema Prolog a trovare la soluzione. Purtroppo la cosa non è così facile e non è semplice programmare in Prolog. Un esempio in GNU - Prolog:

```
    % commento alpha.pl
:- initialization(q).
padre(giovanni,maria).
padre(federico,alfonso).
padre(alfonso,elena).
madre(maria,elena).

genitore(A,B) :- padre(A,B);madre(A,B).
    % ; = 'o' , = 'e'
nonno(A,B) :- padre(A,X),genitore(X,B).

trovanonni :- setof(X,nonno(X,elena),A), write(A), nl.
    % L'insieme degli x tali che x è nonno di Elena
q :- trovanonni.

    % output: [federico,giovanni]
```

Capitolo 15

PHP

L'indirizzo principale per PHP è www.php.net. Lì si trovano manuali, tante altre informazioni e, quando necessario, l'interprete nelle sue ultime versioni. Sotto Linux PHP è già installato e la configurazione si trova in /etc/php.ini. Un ottimo libro è: Converse, Park. *Guida a PHP.* McGraw-Hill.

PHP è un linguaggio orientato soprattutto alla creazione di pagine web interattive; si tratta però di un vero linguaggio di programmazione che teoricamente può essere usato anche per altri compiti oltre a quelli per cui è ottimizzato. Uno dei lati più forti di PHP è la possibilità di utilizzarlo per gestire basi di dati (tra cui Oracle, dBase, PostgreSQL, MySQL). PHP può essere impiegato con molti dei protocolli internet più diffusi, possiede molte funzioni per il trattamento di espressioni regolari e di documenti in XML. Contiene varie funzioni per il commercio elettronico. L'unico rivale del PHP è il Perl che però è molto più complesso, più difficile da imparare e non specializzato per il web.

Quando si usa PHP su un server www, le sorgenti in PHP vengono inserite nei files html, ma sono eseguite solo sul server. Le sorgenti non vengono mai trasmesse al browser. Ciò ha il vantaggio che si possono creare applicazioni il cui codice sorgente non è visibile agli utenti ch accedono al server via www, un aspetto molto importante per la sicurezza del server. Il server infatti quando esegue un codice PHP contenuto in un file html crea un nuovo file html che viene trasmesso al browser al posto dell'originale. JavaScript, ad esempio, viene eseguito dal browser e quindi gli utenti possono vedere il codice sorgente. Questo non crea solo problemi di sicurezza, ma rende l'esecuzione dipendente dal browser, cioè se un certo browser non riconosce un comando di JavaScript si avrà un errore. Un errore in PHP, invece, viene rilevato direttamente sul server; il browser riceve un codice html corretto.

Il codice PHP è pienamente portabile, le stesse sorgenti possono essere usate indifferentemente sotto Linux, Windows o Machintosh. La ditta Zend Technologies offre strumenti commerciali per PHP (www.zend.com).

Per utilizzare PHP con Apache bisogna inserire nel file di configurazione

di Apache (/etc/http/conf/httpd.conf) le seguenti due righe:
LoadModule php4_module /usr/lib/httpd/modules/libphp4.so AddType application/x-httpd-php .php
Dopo di ciò bisogna riavviare Apache con:
service httpd stop service httpd start
Il tutto deve essere eseguito ovviamente da /root (utente privilegiato).

A questo punto nei files html possiamo inserire tra <?php e ?> blocchi di codice sorgente PHP, normalmente nel corpo della pagina (*body*) che può anche contenere più di un blocco PHP, in qual caso le definizioni (ad esempio di funzioni) di un blocco rimangono valide anche nei blocchi successivi. Affinchè i blocchi PHP vengano eseguiti dall'interprete PHP, il file stesso deve portare l'estensione .php (o un'altra estensione prevista in httpd.conf).

Esempio completo con file .html:

```
<html><body>
<?php
function fatt($n) {if ($n<=1) return 1;
return $n*fatt($n-1);}
?>
Questa parte esterna al codice php viene trasmessa
al browser cos&igrave; com'&egrave;.
<?php
$sec = date('s'); if (($sec!='00') and ($sec[0]=='0'))
{$sec = str_replace('0','',$sec);}
if ($sec=='1') {$desinenza='o';} else {$desinenza='i';}
$a=sprintf("\n Sono le %s. %s e %s second %s. \n\n",
date ('G'), date('i'), $sec, $desinenza);
print(nl2br($a));

for($n=1; $n<=9; $n++)
{$a=sprintf("%d!=%d\n",$n,fatt($n));
print(nl2br($a));}
?>
</body></html>
```

L'esempio appena scritto è visibile alla pagina:
hhtp://felix.unife.it/php/prog.php.

Si vede che la data cambia ogni volta che si accede alla pagina: si parla per questo di pagine web dinamiche.

Sotto Unix PHP può essere eseguito al di fuori di pagine web come un normale linguaggio di programmazione e quindi senza l'intervento di un programma server www come Apache (nonostante i libri affermino il contrario).

Creiamo un file alfa che rendiamo eseguibile con il comando: chmod +x alfa

```
#! /usr/bin/php -q
<?php
function fatt($n) {if ($n<=1) return 1;
return $n*fatt($n-1);}
for($n=1; $n<=9; $n++)
{printf("%2d! = %7d\n",$n,fatt($n));}
?>
```

Sia # che // possono essere usati per commentare.

Come linguaggio di programmazione il PHP rimane ad un livello in qualche modo primordiale. Anche l'amministrazione di biblioteche di funzioni è difficoltosa. Le numerose funzioni sono quasi tutte pensate per l'utilizzo con un server web. Anche queste funzioni possono essere imitate spesso facilmente in Perl. Due vantaggi del PHP, anche nei confronti del Perl sono comunque:

1. il codice PHP può essere inserito direttamente nei files html invece di risiedere in programmi CGI (Command - Gateway - Interface) esterni. Ciò facilita di molto la gestione delle pagine web e rende l'esecuzione più veloce.

2. Le buone funzioni per l'interazione con banche dati. Tali funzioni esistono anche in Perl, ma forse la combinazione delle basi di dati con il www nel PHP è più semplice.

Capitolo 16

Gli altri linguaggi

Algol e **PL/1** erano famosi linguaggi procedurali degli anni 60. Anche il **Cobol** è un linguaggio antico ancora oggi utilizzato in ambienti commerciali, ad esempio nelle banche. **RPG** è un linguaggio per la creazione di tabulati commerciali ancora usato. **APL** è un linguaggio vettoriale interpretato che richiede tasti speciali. **Ada** doveva diventare un gigantesco linguaggio universale soprattutto per l'utilizzo negli ambienti militari. **Modula-2** era una continuazione del **Pascal**.

Snobol, **Simula**, **Smalltalk** e **Eiffel** sono linguaggi piuttosto accademici per l'elaborazione dei testi e la programmazione orientata agli oggetti. **Tcl/Tk** è una combinazione di due linguaggi usata per creare interfacce grafiche in linguaggi come Perl e Python. Il modulo **Perl/Tk** del Perl è molto completo e permette una comoda programmazione grafica orientata agli oggetti.

Maple, **Mathematica** e **Matlab** sono usati da matematici e ingegneri nella matematica computazionale. Il **Ruby** è un linguaggio ancora nel nascere che in pratica dovrebbe diventare, nelle intenzioni del suo creatore, un Perl più semplice e più leggibile. **HyperTalk** era un linguaggio quasi naturale per Machintosh e PC purtroppo quasi scomparso per ragioni commerciali. Alla programmazione funzionale sono dedicati vari linguaggi ancora poco diffusi come **Haskall** e **ML**.

16.1 S ed R

S è un linguaggio di programmazione sviluppato appositamente per la statistica, S-Plus è il rispettivo ambiente di programmazione comprendente anche un'interfaccia grafica interattiva mentre R è una versione libera di S che attualmente è curata soprattutto da statistici viennesi (www.r-project.org) con un notiziario elettronico *R news* e (cran.r-project.org) una raccolta di moduli per R nello stile del CPAN di Perl. S fu influenzato da un libro del famoso matematico e statistico John Tukey sull'analisi esplorativa dei dati

apparso nel 1977 e all'inizio conteneva soprattutto funzioni per la rappresentazione grafica di dati multidimensionali. La grafica è ancora oggi uno dei punti di forza del sistema; ad essa si sono però aggiunte nell'arco di una intensa evoluzione moltissime funzioni statistiche che costituiscono il vero patrimonio del linguaggio.

S e il gemello R sono linguaggi funzionali; ciò significa, come abbiamo già più volte osservato, che funzioni possono essere sia argomenti che risultati di altre funzioni. Ciò permette in teoria la realizzazione di quasi tutte le costruzioni matematiche astratte (naturalmente ristrette al finito) e quindi una struttura arbitrariamente corretta dei programmi.

Il sistema è concepito sia per Unix e Linux che per Windows; ciò costituisce, soprattutto nel trattamento dei files una certa debolezza perchè non vengono sfruttate in modo ottimale le funzioni superiori di Unix.

Nonostante la sua intrinseca modernità R non è ancora molto noto come linguaggio di programmazione e nemmeno documentato in modo soddisfacente. Nonostante esista un certo numero di libri tutti però indirizzati più allo statistico (soprattutto in campo medico e/o economico) che al programmatore. Esiste comunque un'ottima documentazione interattiva.

L'inventore John Chambers per S-Plus e anche il gruppo R di Vienna preferiscono ancora oggi l'utilizzo interattivo che ostacola però una programmazione sistematica e trasparente, un peccato per un linguaggio di così alto livello. R può essere considerato come un Lisp per la statistica.

Capitolo 17

Programmare in Perl

17.1 Variabili nel Perl

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

Il vantaggio a prima vista più evidente del Perl sul C è che il programmatore non deve occuparsi della gestione della memoria, che non sono necessarie dichiarazioni per le variabili e che variabili di tipo diverso possono essere liberamente viste 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.

Il Perl non richiede una dichiarazione per le variabili e 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);
- vettori associativi (*hashes*, che iniziano con il simbolo `%`) di scalari.

Iniziano invece senza simboli speciali i nomi delle funzioni e dei riferimenti a files (*filehandles*). In Perl quindi `$alpha`, `\@ alpha`, `\% alpha` 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"

# Output:
# 7
# $a\n 8 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 esempio \n) che appaiono nella stringa vengono sostituite dal loro valore, non invece se sono racchiusi tra apostrofi.

Un punto e virgola alla fine dell'istruzione può mancare se l'istruzione è seguita da una parentesi graffa chiusa.

A differenza del C in 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(3-1)*7` si ottiene l'output 2 perchè viene prima eseguita l'espressione `print(3-1)` seguita da un'inutile moltiplicazione per 7. In casi come questo bisogna quindi 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 la variabile `%a` è del tutto indipendente. Una sintassi simile vale per i componenti di una lista.

17.1.1 Il nostro primo programma

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, ecc.) deve eseguire lo script.

L'istruzione `use strict 'subs'` controlla se il programma contiene stringhe; 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 finale di invio, che viene tolto con `chop`, una funzione che elimina l'ultimo carattere di una stringa.

17.2 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 per esempio del Lisp) che comunque possono, con un po' di fatica essere simulate utilizzando puntatori (che in Perl si chiamano riferimenti) che tratteremo più avanti. 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=(0, 1, 2, 3, 4, 5)` è uguale quindi a sei elementi e non a due. La seguente funzione, quindi, 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"}}
```

Perchè termina il ciclo del while in questo esempio? Ad 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 una coppia di parentesi () denota una 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 [ ] possono essere usate

per denotare segmenti di una lista:
`@a[2..4]` denota la lista i cui componenti sono `$a[2]`, `$a[3]`, `$a[4]`. Posso combinare i puntini con la virgola:
`@a[2..4,6]` l'elemento `$a[6]` viene aggiunto alla fine del segmento.

`(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).

17.2.1 Alcune operatori per le 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 più lenta e probabilmente in molti casi indica che `@b` viene attaccata ad una nuova copia di `@a`. La funzione restituisce come valore la nuova lunghezza di `@a`. Per attaccare `@b` all'inizio di `@a` di 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)` e `pop(@a)` tolgono rispettivamente il primo e 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 coppia 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.

17.2.2 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 perchè si scopre che in contesto scalare le liste definite mediante una variabile si comportano diversamente da liste scritte direttamente nella forma $(a_0, a_1, ..., a_n)$ infatti in questa forma, in contesto scalare, la virgola ha un significato simile e quello dell'operatore `,` 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 della 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 3 4 9
```

Un esempio dell'uso dell'operatore `,`:

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

17.2.3 La funzione grep del Perl

La funzione `grep` viene usata come filtro per estrarre dalla 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..8);
```

```
for (@a){print "$_ "}
print "\n";     # output 2 4 6 8
@a=negativi(0,2,-4,3,-7,-10,9);
for (@a){print "$_ "}
print "\n";     # output -4 -7 -10

@a=("Ferrara","Firenze","Roma","Foggia");
@a=grep{!/^F/}@a;
for (@a){print "$_ "}
print "\n";     # output Roma
```

Il `^` denota l'inizio della riga e quindi `/^F/` è vera se la riga inizia con F; un `!` anteposto significa negazione.

17.2.4 Map

`Map` è una funzione importante del Perl e di tutti i linguaggi funzionali. Con essa da una lista $(a_1, ..., a_n)$ si ottiene una lista $(f(a_1), ..., f(a_n))$ anch'essa argomento di `map`. Il Perl prevede una estensione molto utile al caso in cui 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. La sintassi che usiamo è simile a quella di `grep`.

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=3.141592653589; $pid180=$pi/180;
sub alfacosalfa{my $alfa=shift; ($alfa, cos($alfa*$pid180))}
%tabellacoseni = map{afacosalfa($_)} (0..360)
print $tabellacoseni{30};
#output: 0.8666025
```

17.2.5 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;
 #lc trasforma l'iniziale in minuscolo
print "@b\n";
 # output alfa beta Betty gamma
```

La funzione `sort` prende come argomenti 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.

17.3 Files

17.3.1 L'operatore $< >$

`stdin`, `stdout` e `stderr` sono i file handles (un tipo improprio di variabili che corrisponde alle variabili del tipo FILE* del C) che denotano standard input, standard output e standard error. Se `<File>` è un file handle è il risultato di una 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 con un carattere di *invio* quindi se usiamo `chop` per togliere l'ultimo carattere possiamo perdere un carattere. Nell'input da tastiera l'invio finale c'è sempre, quindi possiamo usare `chop` come abbiamo visto nel paragrafo 17.1.1. Altrimenti si può usare la funzione `chomp` che toglie l'ultimo carattere da una stringa solo se il carattere è 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;
    # separatore di riga per i files;
    # non considera gli invio e legge tutto su 1 riga
open(File,"lettera");
print<File>;
close(File);
```

Il separatore di fine riga (la stringa) è il valore della variabile speciale `$/` e può essere impostato dal programmatore; di default è uguale a `\n` ossia al

carattere di nuova riga. 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 a files il cui nome inizia con `>` (una cattiva idea comunque per le evidenti interferenze con il simbolo di redirezione `>` della shell).

Similmente con `open(File,">>beta")` si apre un file per raggiungere un testo. Il file handle 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 varibile 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 varibili scalari per riferirsi a un File handle, come nel nostro modulo Files che verrà descritto adesso.

17.3.2 Il pacchetto Files

```
1;#files.pm
    # moduli sono file che terminano in .pm
    # che posso usare con use files
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 sotto forma di lista il catalogo della cartella associata con il dirhandle `$dir` da cui con un `grep` (vedi paragrafo 17.2.3) il cui primo argomento è una 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.

17.3.3 Attributi di files e cartelle

`$File` sia il nome di un file (in senso generalizzato). Gli attributi di file 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=lettura
`-w $file`	...diritto d'accesso w=scrittura
`-x $file`	...diritto d'accesso x=esecuzione
`-d $file`	...cartella
`-f $file`	...file regolare
`-T $file`	...file di testo
`-s $file`	...size=lunghezza file (*non è booleano*)

17.3.4 I diritti d'accesso

Molti comandi della shell hanno equivalenti nel Perl, talvolta con 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 (user ID) e GID (Group ID) per ottenere il catalogo di una cartella si usa opendir che è già stata utilizzata nella nostra funzione catalogo nel paragrafo 17.3.2

17.3.5 La cartella di lavoro

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.

```
use Cwd;
$cartella=cwd();
```

17.4 Moduli

Le raccolte di funzioni in Perl si chiamano moduli ed 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 PerlModule). 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 contenuta. 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'istruzione `use matematica;`. Se alcuni moduli che si vogliono usare si trovano in cartelle alfa, beta, gamma che non sono tra quelle tra le quali il Perl cerca di default, si indica ciò con

```
use lib 'alfa','beta','gamma';
use lib "home/rosa/lib/perl";
use matematica;
```

17.4.1 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): il Comprehensive Perl Active Network (CPAN) accessibile al sito www.perl.com /CPAN. 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 Tk-800.024.tar.gz otteniamo una directory Tk-800.024 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 rise Tk;. Per ogni altro modulo CPAN la procedura è esattamente la stessa. Può però accadere che si riveli necessaria l'installazione di altri moduli prima di poter installare il modulo desiderato. Esiste anche la possibilità di un'installazione automatica durante il collegamento. Dalla shell come /root si fa:
perl -MCPAN -e shell
install gino
Il vantaggio di questo metodo è che si ottiene sempre la versione più nuova (confronta man CPAN man perlmodinstall manuali).

17.5 Funzioni

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 dalla lista il primo elemento. Le variabili interne della funzione vengono dichiarate tramite `my`, mentre `local` ha un significato diverso da quello che uno si aspetta (vedi 17.3.2). La funzione può restituire un risultato mediante `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 doppio{my @a=shift; $a+$a}
sub somma2{my ($a,$b)=@_; $a+$b}
sub somma{my $s=0; for(@_){$s+=$_}$s}
print doppio(4), " ";
print somma2(6,9), " ";
print somma(0,1,2,3,4), " ";
```

```
# output 8 15 10
```

Alcuni operatori abbreviati che vengono usati in C e in 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
```

17.5.1 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 perchè accidentalmente 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 locale o privata 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 (da non imitare):

```
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 di 2piu1 ponendo la variabile `$a` uguale all'argomento cioè a 3 per cui il risultato è 9*(3+1)=36

17.5.2 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 ad 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 adesso vedremo. Funzioni anonime, essendo puntatori, quindi scalari, possono essere assegnate come valori di variabili.

```
$quadrato = sub {my $a=shift; $a*$b}
print &$quadrato(6), "\n"   # output: 36
print $quadrato->(6);       # output: 36
```

17.5.3 Funzioni come argomenti di funzioni

Quando ci si riferisce indirettamente ad una funzione (ad esempio quando è argomento di altre funzioni) bisogna premettere il simbolo di `&` (che una volta doveva essere messo sempre). 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 realtà nell'ultimo esempio lo stesso risultato 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

```
$s=0;
for("uno","id","quadrato","cubo")
{$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 disegno{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);
```

17.5.4 Funzioni come valori di funzioni

La seguente funzione può essere utilizzata per ottenere una funzione da un vettore associativo:

```
sub fundahash{my $a=shift; sub{my $x=shift; $a -> {$x}}}
%a=(1,1,2,4,3,9,4,16,5,25,6,36);
print fundahash(\%a) -> (3);   # output 9
```

Funzioni come valori di funzioni sono il concetto fondamentale della programmazione funzionale di cui diamo adesso alcuni esempi.

17.5.5 Programmazione funzionale in Perl

```
sub comp {my ($f,$g)=@_; sub {&$f(&$g(@_))}}
sub valut {my @a=@_; sub {my $f=shift; &$f(@a)}}
sub id {shift}
sub cubo {my $a=shift; $a*$a*$a}
sub diff2 {my($a,$b)=@_; $a-$b}
sub piu1 {my $a=shift; $a+1}
sub quad {my $a=shift; $a*$a}

$f=comp(\&quad;\&piu1);
print &$f(2);"\n";        # output 9
#oppure
print comp(\&quad,\&piu1)->(2),"\n";

$f=comp(\&quad,\&diff2);
print &$f (6,1), "\n";    # output 25
#oppure
print comp(\&quad,\&diff2)->(6,1),"\n";

$f=valut(3);
```

```
 print &$f(\&cubo), "\n"; # output 27
#oppure
 print valut(3)->(\&cubo), "\n";
```