Appendice

C

Cenni sul linguaggio di programmazione Perl

Ogni linguaggio di programmazione presenta caratteristiche peculiari che lo rendono più o meno adatto a diversi scopi. Perl è un linguaggio di programmazione di pubblico dominio, il cui sviluppo fu iniziato da Larry Wall nel 1988, ma a cui successivamente hanno contribuito (e stanno contribuendo) migliaia di persone. Questo linguaggio di programmazione è interpretato, anziché essere compilato (vedere Appendice B), e consente di processare file e manipolare testi con estrema facilità, pertanto è particolarmente adatto per la realizzazione di programmi in cui sia necessaria un'intensa interazione con il sistema operativo; per esempio il lancio automatico di applicazioni, la gestione di file e il controllo di processi. I programmi interpretati sono spesso chiamati **script**.

Perl è un linguaggio caratterizzato da una sintassi molto flessibile: lo stesso algoritmo può essere implementato in molti modi diversi, producendo lo stesso risultato. Dal momento che si tratta di un linguaggio interpretato, Perl non richiede di essere compilato prima dell'esecuzione, rendendo molto semplice la realizzazione e il **debugging** dei programmi. Per queste caratteristiche Perl è stato adottato da molti programmatori e sistemisti come linguaggio di **scripting**; per esempio, i programmi **cgi** che accompagnano molte pagine web interattive sono generalmente scritti proprio in Perl. Come tutti i linguaggi interpretati, Perl non brilla per velocità, tuttavia può essere utilizzato anche per la realizzazione di programmi complessi, specialmente se il tempo di esecuzione non rappresenta un fattore particolarmente limitante.

A differenza di altri linguaggi come C e Java, Perl non richiede la dichiarazione esplicita delle variabili prima del loro utilizzo. Le normali variabili di Perl sono semplicemente rappresentate da un nome preceduto dal simbolo del dollaro. Per esempio, il comando `$base=50;` attribuisce alla variabile `$base` il valore numerico di 50, per cui, fintanto che il valore di `$base` non sarà cambiato, ogni volta che si farà riferimento a `$base` sarà restituito il valore numerico di 50. Allo stesso modo il comando `$base="ACGTACGT";` attribuisce alla variabile il contenuto della stringa di otto caratteri. L'automatismo con cui si possono trattare le variabili di Perl rende il linguaggio molto immediato, anche se, nella realizzazione di un buon programma, sarebbe comunque opportuno pianificare anticipatamente le variabili da utilizzare e il tipo di dati che esse sono destinate a rappresentare.

Come per la maggior parte dei linguaggi di programmazione anche Perl comporta che un programma consista tipicamente in una serie di comandi. Alcune sequenze di comandi possono essere racchiuse tra parentesi graffe, costituendo in tal modo un *blocco*. L'esecuzione dei diversi comandi e dei blocchi che compongono un programma è controllata da *operatori di controllo del flusso*, cioè da particolari istruzioni che determinano il procedere del programma in base alla realizzazione di alcune condizioni.

C.1 Disponibilità e installazione di Perl

Per eseguire un programma scritto in Perl è necessario che un computer disponga dell'interprete Perl, cioè del programma in grado di leggere il codice scritto in Perl e di eseguirlo. L'interprete Perl è disponibile per numerosi sistemi operativi tra cui Unix, MS/DOS, Windows, Macintosh, OS/2, Amiga ed è possibile scaricarlo liberamente dalla rete internet. Oltre agli interpreti di Perl, sono disponibili in rete diversi manuali, di ogni livello di complessità. ►La Tabella C.2 riporta una serie di siti web utili.

I programmi scritti in Perl sono generalmente trasportabili da un sistema operativo all'altro, spesso senza che si renda necessaria alcuna modifica. Ovviamente fanno eccezione le chiamate dirette al sistema operativo che dovranno essere adattate di volta in volta.

Anche se Perl può essere utilizzato con diversi sistemi operativi, esso è stato sviluppato tenendo in particolare considerazione l'ambiente Unix. Il sistema operativo Unix, quando esegue uno script, legge nella prima riga la modalità con cui lo script deve essere interpretato. Più precisamente, la prima riga dello script deve indicare il percorso in cui è localizzato il programma con cui interpretare lo script, preceduto dal simbolo `#!` ; per esempio `#!/usr/bin/perl` . Prima di lanciare uno script in ambiente Unix è necessario renderlo eseguibile, per esempio con il comando `chmod +x nomefile.pl` , dove `nomefile.pl` rappresenta il nome del file che contiene lo script (Appendice B).

Nei prossimi paragrafi vengono illustrati a titolo d'esempio tre programmi scritti in Perl. Si raccomanda a chiunque volesse iniziare a programmare con questo linguaggio di installare l'interprete Perl sul proprio computer (se non è già installato) e di provare a eseguire e a migliorare i tre programmi forniti come esempio, variando alcune istruzioni e aggiungendone di nuove.

C.2 Somma.pl: un semplice programma per sommare numeri

Somma.pl è uno script di poche righe di programma, ma racchiude diversi importanti concetti. La sua funzione è di sommare una serie di numeri passati con la linea di comando. Per esempio:

```
somma.pl 22 3 3 0 4 9 -14 7 8 5 <invio>
```

restituisce come output il seguente messaggio:

```
Sommati 10 valori
Il risultato è 47
```

Il listato del programma somma.pl è riportato nel ►Box C.1. Si consideri innanzi tutto la prima riga `#!/usr/bin/perl` , di cui si è già parlato nell'introduzione di questa Appendice, che indica al sistema operativo che si tratta di uno script in Perl. A eccezione della prima riga, tutte le righe che iniziano con il simbolo `#` sono considerate commenti e come tali sono ignorati nell'esecuzione del programma. Anche quando il simbolo `#` appare all'interno di una riga, tutto quanto segue viene considerato come commento. Un'altra regola generale (vedere il listato del programma) riguarda l'uso del punto e virgola a indicare la fine di ogni comando.

Il programma inizia con l'assegnazione del valore zero alla variabile `$tot` . Questa variabile sarà utilizzata come totalizzatore in cui, uno dopo l'altro saranno sommati i valori. Quindi con questa operazione, attuata all'inizio dell'esecuzione del programma, si stabilisce che il valore iniziale del totalizzatore è zero.

Box D1.1 **LISTATO DEL PROGRAMMA SOMMA.PL.**

```
#!/usr/bin/perl
# Programma somma.pl

$tot = 0;          # il totale è inizialmente zero
$n = $#ARGV + 1;   # $n contiene il numero di addendi

# Operazione ciclica "for", ripetuta $n volte
for($x=0; $x<$n; $x++) {$tot = $tot + $ARGV[$x];}

# stampa i risultati
print "Sommati $n valori\nIl risultato e' $tot\n";
```

C.2.1 Argomenti passati con la linea di comando

La riga successiva è più complessa e merita una accurata descrizione:

```
$n =$#ARGV +1;
```

Anche in questo caso si tratta dell'assegnazione di un valore (definito dall'espressione `$#ARGV + 1`) a una variabile (`$n`), in cui si vuole memorizzare il numero di addendi, che nel caso dell'esempio illustrato corrisponde a 10 (il numero di argomenti passati con la linea di comando). A questo proposito bisogna capire come un programma scritto in Perl fa propri gli argomenti passati con la linea di comando. Il primo argomento, quello che segue il nome del programma (nel caso dell'esempio illustrato sopra corrisponde al numero 22), è automaticamente assegnato alla variabile `$ARGV[0]` , il secondo argomento alla variabile `$ARGV[1]` , e così via fino all'ultimo argomento. Nel caso dell'esempio sono passati complessivamente 10 argomenti, l'ultimo dei quali viene automaticamente assegnato alla variabile `$ARGV[9]` . Il numero di argomenti passati al programma può essere ricavato dalla variabile `$#ARGV` in cui è automaticamente memorizzato l'ultimo indice utilizzato. Quindi, nel caso dell'esempio ora esposto, alla variabile `$ARGV[9]` viene attribuito il valore di 5 (l'ultimo numero inserito con la linea di comando), mentre alla variabile `$#ARGV` è assegnato il valore di 9. Siccome la numerazione inizia dal numero zero, il numero di argomenti passati corrisponde a `$#ARGV +1`.

C.2.2 Operazioni cicliche: l'istruzione «for»

La riga successiva del programma somma.pl è la più difficile da interpretare. Si tratta di un'operazione ciclica controllata dall'istruzione `for` .

```
for($x=0; $x<$n; $x++) {$tot = $tot + $ARGV[$x];}
```

Per (`for`) tutti i valori di `$x` a partire da 0 e fino che `$x<$n` viene eseguito il blocco racchiuso in parentesi graffe, cioè alla variabile `$tot` viene assegnato il valore di `$tot + $ARGV[$x]` . Nel caso dell'esempio illustrato, `$n` è uguale a 10, quindi `$x` assume i 10 valori compresi tra 0 e 9; con il risultato che, uno dopo l'altro, i valori di `$ARGV[0], $ARGV[1], $ARGV[2], ... $ARGV[9]` sono aggiunti alla valiabile `$tot` .

L'istruzione `for` è molto usata in Perl e in modo simile in altri linguaggi di programmazione. La sintassi di questo comando può essere così schematizzata: `for(operazioneIniziale; condizionePerContinuare; operazioneCiclica) { … blocco … }`.

Con questo comando si ordina al programma di ripetere il ciclo di istruzioni, corrispondente al blocco chiuso tra le parentesi graffe, per un certo numero di volte. All'inizio del primo ciclo viene svolta l'`operazioneIniziale` che nel caso dell'esempio assegna il valore zero alla variabile `$x`, che funge da contatore. La `condizionePerContinuare` determina fino a quando deve essere ripetuto il ciclo; nell'esempio la condizione è `$x<$n` . All'inizio di ogni ciclo il programma verifica se tale condizione sia soddisfatta e in caso contrario esce dal ciclo, procedendo con la prima istruzione dopo la

chiusura della parentesi graffa. L'`operazioneCiclica` è eseguita alla fine di ogni singolo ciclo. L'operazione illustrata nell'esempio è `$x++` che incrementa la variabile `$x` di un'unità. Infatti, il doppio segno di addizione o il doppio segno di sottrazione posti dopo il nome di una variabile ne incrementano o decrementano il valore di un'unità.

Per eseguire un'operazione aritmetica su una variabile generalmente non si usa la modalità illustrata sopra, ma una forma contratta, adottata non solo da Perl, ma anche da altri linguaggi di programmazione come C e Java. Per esempio, `$a=$a+5;` si può scrivere come `$a+=5;` allo stesso modo `$a-=5;` sottrae 5 al valore di `$a`, mentre `$a*=5;` e `$a/=5;` rispettivamente moltiplicano e dividono il valore di `$a` per 5. Quindi, nel programma somma.pl si può sostituire il blocco racchiuso tra parentesi graffe nel seguente modo: `{$tot += $ARGV[$x];}`.

Per concludere il programma somma.pl stampa il risultato. Notare il fatto che all'intermo delle doppie virgolette che seguono l'istruzione **print** si possono inserire variabili, di cui viene stampato il contenuto. Il simbolo `\n` (*newline*) consente di andare a capo.

C.3 Solodueparole.pl: un programma per cercare due parole vicine in un file di testo

Lo scopo del programma *solodueparole.pl* è di cercare in un file di testo (per esempio una sequenza di DNA) due parole (per esempio due siti di restrizione) in modo tale che, ogni volta che una parola sia trovata a una distanza dall'altra entro una soglia definita, il programma indichi la posizione assoluta e la distanza in caratteri tra le due parole. L'input del programma è rappresentato:

1. dal nome del file da analizzare;
2. dalla prima parola da cercare;
3. dalla seconda parola da cercare;
4. dalla distanza massima da considerare tra le due parole.

Per esempio: `solodueparole.pl solodueparole.pl string parola 13`

Nel caso dell'esempio, il computer eseguirà il programma «solodueparole.pl» che a sua volta aprirà e leggerà il file «solodueparole.pl» (lo stesso file!) cercando le due parole *string* e *parola* a una distanza non superiore a 13 caratteri. L'output del programma è il seguente (confrontare il testo del Box C.2), in cui i numeri tra parentesi quadre rappresentano la distanza tra le due parole:

```
parola [10] string a pos. 829
string [10] parola a pos. 1210
parola [12] string a pos. 1355
string [5] parola a pos. 1366
string [5] parola a pos. 1789
```

Per la realizzazione del programma è stato applicato il seguente algoritmo in cui Pos1 e Pos2 memorizzano l'ultima posizione letta di Parola1 e Parola2:

- leggere dal file di input un carattere alla volta e per ogni carattere;
  - determinare la parola che finisce alla posizione corrente;
  - se la parola letta corrisponde alla Parola1 definita dall'utente;
    - definire Pos1 = posizioneCorrente;
    - se Pos2 non è stata ancora definita ignora la prossima riga;
      - se la distanza tra Pos1 e Pos2 è entro la soglia, stampa…;
  - se la parola letta corrisponde alla Parola2 definita dall'utente;
    - definire Pos2 = posizioneCorrente;
    - se Pos1 non è stata ancora definita ignora la prossima riga;
      - se la distanza tra Pos2 e Pos1 è entro la soglia, stampa…;
- continuare fino alla fine del testo.

Box C.2

LISTATO DEL PROGRAMMA SOLODUEPAROLE.PL, PER CERCARE IN UN FILE DI TESTO DUE PAROLE POSTE ENTRO UN LIMITE DI DISTANZA DEFINITO DALL'UTENTE. LA DESCRIZIONE DEL PROGRAMMA E L'OUTPUT DEI RISULTATI SONO FORNITI NEL TESTO.

```
    #!/usr/bin/perl
##############################################################################
# solodueparole.pl - Trova in un file due parole entro una distanza soglia
##############################################################################

die "Uso: solodueparole.pl nomefile parola1 parola2 distanza.
Il programma legge il file e cerca Parola1 e Parola2 entro la distanza
definita tra le due parole che devono avere la stessa lunghezza.\n"
unless $ARGV[3] ne "";

# legge gli argomenti passati dalla linea di comando
$nomefile = $ARGV[0];
$parola1 = $ARGV[1];
$parola2 = $ARGV[2];
$dist = $ARGV[3];

# imposta le condizioni iniziali
$lungparola = length($parola1);
die "Diversa lunghezza parole\nTerminato\n" unless $lungparola == length($parola2);
for ($x=0, $string = "" ; $x<$lungparola; $x++) {$string = $string . " ";} # spazi

# Apre il file di input
die "File $nomefile non trovato\n" unless open( INFILE, "< $nomefile " );

# Legge un carattere alla volta dal file di input, fino all'end-of-file (eof)
for ($posiz=0; !eof(INFILE); $posiz++) {
    $string = $string . getc(INFILE);    # aggiunge il carattere alla fine di $string
    $string = substr($string, 1, $lungparola); # toglie il primo carattere da $string
    next if $posiz<$lungparola; # va al carattere successivo finché $posiz<$lunparola

    if ($string eq $parola1){            # la stringa corrente corrisponde a parola1
        $pos1=$posiz;                    # ne memorizza la posizione
        if(defined($pos2)) {             # verifica se $pos2 è stata gia' trovata
           $sep=$pos1-$pos2-$lungparola; # calcola la separazione dall'ultima parola2
           print "$parola2 [$sep] $parola1 a pos. $posiz\n" if ($sep < $dist);
        }
    }
    elsif ($string eq $parola2){         # applica la stessa logica a parola2
        $pos2=$posiz;                    # ...
        if ( defined ( $pos1 ) ) {       # ...
           $sep=$pos2-$pos1-$lungparola; # ... e stampa se entro i limiti di distanza
           print "$parola1 [$sep] $parola2 a pos. $posiz\n" if ($sep < $dist);
        }
    }
}
# Chiude il file di input
close(INFILE);
```

L'algoritmo è molto semplice e può essere migliorato in molti aspetti. Allo stesso modo, l'implementazione dell'algoritmo nel programma descritto più avanti non deve essere considerata come la soluzione migliore, né la più elegante, ma piuttosto come un semplice esempio per illustrare l'uso del linguaggio Perl. Il programma solodueparole.pl è riportato nel ▶Box C.2.

C.3.1 Controllo del flusso: die, if, unless

Il programma solodueparole.pl inizia con la seguente istruzione:

```
die " … messaggio … " unless $ARGV[3] ne "";
```

La funzione `die` (morire) è utilizzata per uscire dal programma lanciando come ultimo messaggio il testo racchiuso tra le virgolette (tre intere righe di testo nell'esempio). Molto spesso, come nel caso illustrato, questa funzione è associata a una condizione; nel caso specifico `unless $ARGV[3] ne "";`. L'istruzione completa deve quindi

essere letta «Esci dal programma se non si verifica la condizione che (*unless*) la variabile `$ARGV[3]` non è uguale (*ne* = *not equal*) a una stringa vuota (`""`)». Lo stesso effetto poteva essere ottenuto con l'istruzione `die «… messaggio …» if $ARGV[3] eq "".`

Le istruzioni `if` e `unless` sono particolarmente importanti perché consentono al programma di fare delle scelte, cioè di seguire dei percorsi diversi in base a **condizioni** codificate.

Un'altra osservazione interessante riguarda i simboli adottati per verificare l'eguaglianza (`eq`) o disuguaglianza (`ne`) tra stringhe di caratteri. Nel caso specifico si confronta il contenuto della variabile `$ARGV[3]` con una stringa vuota. Il programma solodueparole.pl prevede infatti che con la linea di comando siano passati 4 argomenti, tramite le variabili `$ARGV[]` . Quindi l'istruzione:

```
die "… messaggio …" unless $ARGV[3] ne "";
```

ha il significato di verificare che il quarto argomento (che ha indice 3) sia stato definito; se il quarto argomento è presente, ovviamente, sono presenti anche i primi tre. In caso contrario il programma muore inviando come ultimo messaggio le indicazioni sulla sintassi da usare.

In Perl i comandi `if` e `unless` si possono usare in due modi alternativi; il primo come illustrato sopra, è tipicamente usato per eseguire in modo condizionale un singolo comando, per esempio:

```
print "00100" if($citta eq "Roma");
```

Il secondo modo, più simile a quanto usato in altri linguaggi di programmazione, prevede di definire prima la condizione da verificare e poi il blocco di comandi racchiuso tra parentesi graffa:

```
if($citta eq "Roma") {
    ...;
    ...;
    }
```

C.3.2 L'uguale non è sempre uguale!

Il programma solodueparole.pl continua con le seguenti istruzioni:

```
$lungparola = length($parola1);
die "... messaggio ..." unless $lungparola == length($parola2);
```

La funzione `length()` consente di determinare la lunghezza di una stringa di caratteri, nel caso specifico di `$parola1` . Il valore è quindi assegnato alla variabile `$lungparola` .

Dato che è stato imposto che le due parole abbiano la stessa lunghezza, il programma effettua una verifica e muore qualora le lunghezze siano diverse. È importante notare che i simboli usati per verificare l'eguaglianza e la disuguaglianza di valori numerici sono diversi da quelli usati per le stringhe di caratteri (Box C.2): per l'eguaglianza invece di `eq` si usa `==` mentre per la disuguaglianza invece di `ne` si usa `!=`.

Un aspetto che spesso confonde le idee è che la parola *uguale* sia tradotta nel linguaggio informatico in diversi modi.

Per assegnare il valore a una variabile si usa il simbolo `=` (per esempio `$a=5;`); mentre si usano i simboli `eq` e `==` rispettivamente per confrontare due stringhe di caratteri e due valori numerici. In realtà, anche se nel linguaggio corrente viene usata in tutti tre i casi la parola *uguale*, le tre operazioni sono diverse, specialmente le operazioni di assegnazione e di confronto; quindi esse sono rappresentate da simboli diversi. Se per esempio si scrivesse `print "ciao" if($a=5);` il programma scriverebbe «ciao» indipendentemente dal valore di `$a;` inoltre il programma assegnerebbe alla variabile `$a` il valore di 5.

C.3.3 Creazione di una stringa di spazi

L'uso dell'istruzione `for` è già stato spiegato nel Paragrafo C.2.2. Nella seguente riga del programma solodueparole.pl questa istruzione è usata per creare una stringa di `$lungparola` spazi.

```
for ($x=0, $string = "" ; $x<$lungparola; $x++) {$string = $string . " ";}
```

All'inizio del primo ciclo viene svolta l'*operazioneIniziale* che nel caso dell'esempio è duplice: la prima operazione assegna il valore zero alla variabile `$x`, che funge da contatore; la seconda operazione, separata da una virgola, assegna alla variabile `$string` una stringa di testo vuota. La *condizionePerContinuare* è che `$x<$lungparola` . All'inizio di ogni ciclo il programma verifica se tale condizione sia vera e in caso contrario esce, procedendo con la prima istruzione dopo la chiusura della parentesi graffa. L'*operazioneCiclica* è `$x++` che incrementa la variabile `$x` di un'unità alla fine di ogni ciclo.

Con il ciclo `for` illustrato, il blocco di istruzioni racchiuso tra le parentesi graffe è quindi ripetuto un numero di volte pari al contenuto della variabile `$lungparola` . A ogni ciclo viene eseguita la seguente istruzione `{$string = $string . " ";}` che assegna alla variabile `$string` lo stesso valore di `$string` con l'aggiunta di uno spazio. Il punto posto dopo `$string` è un operatore che concatena due stringhe tra loro, nel caso specifico `$string` e lo spazio. Il risultato finale è che alla variabile `$string` è assegnata una stringa di spazi, di lunghezza pari a `$lungparola` . Come già visto per le operazioni aritmetiche, anche in questo caso si può usare la forma contratta, ponendo l'operatore *punto* prima dell'uguale: `{$string .= " ";}`.

C.3.4 Apertura di un file

Nella linea successiva del programma viene aperto il file di input e viene controllato che l'operazione sia portata a termine con successo.

```
die "File $nomefile non trovato\n" unless open( INFILE, "< $nomefile " );
```

La seconda parte della riga (dopo `unless`) considera la variabile `$nomefile` in cui all'inizio del programma era stato memorizzato il primo argomento passato con la linea di comando (cioè il nome del file da leggere). Quindi il comando `open` apre il file con quel nome, in lettura (input). Il simbolo `'<'` , posto prima del nome del file, indica l'apertura in lettura; alternativamente il simbolo `'>'` indicherebbe che si vuole creare un nuovo file in cui scrivere l'output del programma. Se non viene data alcuna indicazione sulla modalità di apertura, allora il file è aperto in lettura; il simbolo `'<'` non è dunque necessario, ma è stato incluso ugualmente per maggiore chiarezza.

In un programma possono essere aperti simultaneamente molti file; è quindi necessario attribuire a ogni file un *filehandle* con cui maneggiarlo. Nell'esempio illustrato, al filehandle è stato attribuito il nome `INFILE` . Quindi, ogni volta che il programma trova il termine `INFILE` lo associa al file corrispondente, per esempio per leggerne un carattere come illustrato successivamente con l'istruzione `getc(INFILE)` . Il filehandle contiene anche le informazioni relative al puntatore interno, che all'atto dell'apertura viene posto all'inizio del file per spostarsi avanti mano a mano che il file è letto.

Ma cosa succederebbe se l'istruzione `open` non riuscisse trovare il file? Per esempio per un errore di battitura. In tal caso verrebbe restituito un messaggio di `falso` che determinerebbe l'esecuzione dell'istruzione `die` , con la conseguente interruzione del programma. Si raccomanda di controllare sempre che le operazioni di apertura dei file siano completate con successo, utilizzando una riga di comando come: `die "… messaggio …" unless open(INFILE, " $nomefile ")`, che si può interpretare come «muori e invia l'ultimo messaggio se non si verifica l'apertura del file».

C.3.5 Ciclo principale del programma solodueparole.pl

Finalmente, nella riga successiva, il programma entra nel ciclo principale che legge un carattere alla volta dal file di input e cerca le due parole. Anche in questo caso è utilizzato il comando `for`. La variabile `$posiz` funge da contatore del numero di caratteri letti, mentre la «condizione per continuare» è determinata dall'espressione `!eof(INFILE)` che assume un valore `falso` quando si raggiunge l'`end-of-file (eof)`. Bisogna notare che alla fine del file `eof` restituisce `vero` e non `falso`. Per questa ragione è modificato dall'operatore `!` (punto esclamativo) che significa `NOT`. Quindi la condizione per continuare è che non sia stata raggiunta la fine del file.

Le prime due righe del blocco aggiornano `$string` che memorizza la parola corrente letta dal file di input.

```
$string = $string . getc(INFILE);
$string = substr($string, 1, $lungparola);
next if $posiz<$lungparola;
```

Nella prima riga è usata la funzione `getc(INFILE)` per leggere il prossimo carattere dal file di input. Il nuovo carattere è aggiunto alla fine della parola, con l'operatore di concatenazione di stringhe «.» già descritto precedentemente. La seconda riga usa la funzione `substr()` per estrarne una sottostringa: il primo argomento è la stringa di partenza, nel caso dell'esempio `$string`. Il secondo argomento definisce l'inizio della sottostringa da estrarre, considerando zero la posizione del primo carattere. Quindi il valore di uno indica che la sottostringa parte dal secondo carattere di `$string`. Infine, il terzo argomento definisce la lunghezza della sottostringa, che nell'esempio è pari a `$lungparola`. In pratica, la prima riga aggiunge un carattere alla fine di `$string`. Mentre la seconda riga toglie il primo carattere.

L'istruzione `next`, all'inizio della riga successiva, forza il programma a continuare con il prossimo ciclo, ignorando il resto del blocco di istruzioni. Il significato di questa riga è ovvio: se non è ancora stato letto un numero di caratteri pari alla lunghezza delle parole cercate allora non ha senso cercare le parole, anzi potrebbe portare a errori. Quindi si passa direttamente alla lettura del prossimo carattere. L'istruzione `next` non deve essere confusa con l'istruzione `last` che invece forza il programma a uscire definitivamente dal ciclo e a continuare con l'istruzione successiva al blocco racchiuso tra le parentesi graffe. Nei due blocchi che seguono viene verificata la presenza rispettivamente di `$parola1` e di `$parola2`.

```
if ($string eq $parola1){
   $pos1=$posiz;
   if(defined($pos2)) {
      $sep=$pos1-$pos2-$lungparola;
      print "$parola2 [$sep] $parola1 a pos. $posiz\n" if ($sep < $dist);
   }
}
```

Se viene trovata `$parola1` allora la variabile `$pos1` è aggiornata e si procede a stampare il risultato se le condizioni di distanza tra le due parole sono soddisfatte (`print "$parola2 ..." if ($sep < $dist)`). Ma deve essere verificata anche un'altra condizione: che `$parola2` sia stata già trovata almeno una volta. Altrimenti la variabile `$pos2` non sarebbe stata ancora definita e non avrebbe senso calcolarne la distanza da `$pos1`, anzi si rischierebbe un errore. Tale verifica è effettuata dall'istruzione `if(defined($pos2))` che restituisce un risultato positivo (`true`) solo se `$pos2` è già stata definita, cioè se anche `$parola2` è stata trovata.

L'algoritmo qui descritto offre la possibilità di analizzare anche file di grandi dimensioni. Si può facilmente arguire che il tempo di esecuzione è una funzione direttamente proporzionale alla lunghezza del file di input, mentre le risorse di memoria richieste dal programma sono irrisorie e non sono dipendenti dalla gran-

dezza del file di input dal momento che il computer memorizza solo la parola corrente.

Alcune utili modifiche potrebbero migliorare l'algoritmo e il programma. Per esempio si potrebbe includere la possibilità di cercare parole di lunghezza diversa. Un'altra opzione, utile nell'analisi di file di biosequenze, potrebbe essere quella di ignorare tutti gli spazi vuoti, gli «a capo» e altri caratteri come numeri che spesso sono inclusi nei file di sequenza, ma non devono essere considerati nell'analisi. Si potrebbe inoltre aggiungere un controllo che consenta di ignorare la prima riga del file nel caso inizi con il simbolo `'>'` che per convenzione contiene i commenti nei file in formato FASTA.

C.4 Un programma per lanciare Blast e analizzare automaticamente i risultati

Il linguaggio Perl è particolarmente adatto alla realizzazione di script in grado di interagire con il sistema operativo per lanciare programmi applicativi o per manipolare file. Nell'esempio di *SmistaBlast.pl* sono stati utilizzati diversi comandi di interazione con il sistema operativo, per esempio per verificare se una directory sia presente, oppure per creare o eliminare un file. La parte più interessante di questo programma consiste nel lanciare Blast e nel leggere automaticamente i risultati per decidere come smistare le sequenze in directory diverse. Per sperimentare l'esempio qui illustrato è necessario installare localmente il programma Blast; inoltre bisogna effettuare il *download* dell'insieme delle sequenze proteiche del lievito *Saccharomyces cerevisiae* e di *Drosophila melanogaster* che saranno usate per effettuare le ricerche di similarità.

C.4.1 Installazione del programma Blast

Il programma Blast (Paragrafo 3.2.4) è stato sviluppato dai ricercatori del NCBI ed è disponibile presso il loro sito web sia come programma direttamente eseguibile in rete, sia come pacchetto software da scaricare e installare sul proprio computer (▶Tabella C.2). Sebbene il linguaggio Perl abbia la possibilità di formulare comandi per accedere e consultare pagine web remote, è più semplice e veloce installare Blast sul proprio computer ed eseguire le ricerche di similarità a livello locale, come illustrato in questo Paragrafo.

Al sito del NCBI sono disponibili pacchetti Blast facilmente installabili su diversi sistemi operativi. L'esempio qui illustrato si riferisce al sistema operativo **Unix/Linux** e assume che la directory di lavoro sia `bioinfo` . Prima di iniziare l'installazione si crei quindi una directory con tale nome sul proprio computer, usando il comando `mkdir bioinfo` ; per entrare nella directory usare il comando `cd bioinfo`, per tornare indietro di una directory usare il comando `cd ..`; infine, per verificare il nome della directory corrente usare il comando `pwd` (*print working directory*) (Appendice B per ulteriori informazioni sui comandi del sistema operativo Unix).

Per mantenere ordine, all'interno della directory «bioinfo» si creino due subdirectory: `bioinfo/blast/` , dove sarà installato il pacchetto Blast, e `bioinfo/db/` , dove saranno scaricati e formattati i database di lievito e di *Drosophila*. È importante ricordare che il comando «mkdir» crea la nuova directory all'interno della directory di lavoro corrente; ci si assicuri quindi di essere in «bioinfo» prima di creare le due nuove directory.

Dopo avere predisposto il sistema come descritto sopra, scaricare all'interno della directory `bioinfo/blast/` il pacchetto *blast.linux.tar.Z* reperibile presso il sito del NCBI. Se si usasse un sistema operativo diverso da Linux (per esempio Windows/DOS) ci si assicuri di identificare il pacchetto software corrispondente. L'estensione **.Z** indica che si tratta di un file compresso, mentre l'estenzione **.tar** indi-

ca che si tratta di un file di archivio, che a sua volta contiene diversi file; quindi si dovranno lanciare i seguenti comandi:

```
ls -l
uncompress blast.linux.tar.Z
ls -l
tar -xvf blast.linux.tar
ls -l
```

Prima e dopo ogni commando si consiglia di digitare `ls -l`, come indicato sopra, per listare il contenuto della directory. Dopo `uncompress` il file perde l'estensione **.Z** e aumenta di dimensione. Invece, dopo il comando **tar** vengono generati molti file e directory, che erano contenuti nell'archivio, come per esempio la directory *data* contenente diversi file, tra cui le matrici di similarità PAM e BLOSUM (Capitolo 3).

Per verificare la funzionalità del programma Blast digitare `./blastall` che restituisce una lunga lista di istruzioni di cui si parlerà nei paragrafi seguenti.

C.4.2 Reperimento dei dati di sequenza e formattazione con formatdb

Per continuare con l'esempio illustrato, le banche di sequenze proteiche di lievito e di *Drosophila* dovranno essere installate all'interno della directory *bioinfo/db*. L'intero corredo di sequenze proteiche del lievito *Saccharomyces cerevisiae* è reperibile presso diversi siti web (Tabella C.2). Per esempio, presso il sito del *MIPS* è possibile scaricare l'insieme di tutte le sequenze proteiche codificate nel genoma di lievito. Allo stesso modo le sequenze di *Drosophila* sono reperibili per esempio presso il sito del NCBI. Se si scaricano file compressi (per esempio con estensione **.Z** oppure con estensione **.gz**) si dovrà ovviamente procedere alla loro decompressione prima di poterli utilizzare.

I file di sequenze dovrebbero contenere semplice testo ASCII, quindi dovrebbero essere leggibili con il comando *more* di Unix, oppure con il programma *Notepad* di Windows. In un file devono essere contenute le sequenze proteiche di lievito e nell'altro quelle di *Drosophila*. Assicurarsi che i due file siano in formato FASTA multiplo come illustrato in ▶Figura C.1.

Le banche di sequenze, prima di poter essere utilizzate dal programma Blast, devono essere formattate con il programma **formatdb**, incluso nel pacchetto software di Blast. Nell'esempio qui illustrato, i due file contenenti le banche di sequenze si chiamano *yeast-prot* e *droso-prot* e sono contenuti all'interno della directory *bioinfo/db/*. Per formattare la banca di sequenze proteiche di lievito ci si porti all'interno della directory «*db*» e si esegua il seguente comando:

```
../blast/formatdb -i yeast-prot -o T
```

Dall'interno della directory «*db*», il percorso per trovare il programma *formatdb* comporta che si debba risalire una directory per poi entrare in *blast/*. Quindi *formatdb* è preceduto dal percorso `../blast/` . L'argomento «-i» è obbligatorio e definisce il nome del file contenente le sequenze, mentre l'argomento «-o» indica al programma di creare un indice dei nomi delle sequenze; in tal modo è possibile estrarre una sequenza dalla banca dati con estrema facilità, come sarà illustrato più avanti. L'argomento «-o» è *False* di default, pertanto se si vuole attivare questa opzione è necessario impostarla espressamente a T (*True*) come indicato sopra.

Il programma *formatdb* può essere usato anche per la formattazione di sequenze di acidi nucleici, ma in tal caso deve essere specificato anche l'argomento «-p F», che indica che si tratta di sequenze di DNA:

```
formatdb -p F -i yeast-DNA
```

Dopo l'esecuzione di *formatdb* eseguire il comando «ls –l» per verificare la presenza dei file formattati che saranno usati da Blast. Tra gli altri dovrebbero essere presenti i file *yeast-prot.phr*, *yeast-prot.pin* e *yeast-prot.psq*. Comunque, il programma Blast

```
......
>gi|7290031|gb|AAF45498.1| (AE003417) ac gene product [Drosophila melanogaster]
MALGSENHSVFNDDEESSSAFNGPSVIRRNARERNRVKQVNNGFSQLRQHIPAAVIADLSNGRRGIGPGANKKLSKVSTL
KMAVEYIRRLQKVLHENDQQKQKQLHLQQQHLHFQQQQQHQHLYAWHQELQLQSPTGSTSSCNSISSYCKPATSTIPGAT
PPNNFHTKLEASFEDYRNNSCSSGTEDEDILDYISLWQDDL
>gi|7290032|gb|AAF45499.1| (AE003417) sc gene product [Drosophila melanogaster]
MKNNNNTTKSTTMSSSVLSTNETFPTTINSATKIFRYQHIMPAPSPLIPGGNQNQPAGTMPIKTRKYTPRGMALTRCSES
VSSLSPGSSPAPYNVDQSQSVQRRNARERNRVKQVNNSFARLRQHIPQSIITDLTKGGGRGPHKKISKVDTLRIAVEYIR
RLQDLVDDLNGGSNIGANNAVTQLQLCLDESSSHSSSSSTCSSSGHNTYYQNTISVSPLQQQQQLQRQQFNHQPLTALSL
NTNLVGTSVPGGDAGCVSTSKNQQTCHSPTSSFNSSMSFDSGTYEGVPQQISTHLDRLDHLDNELHTHSQLQLKFEPYEH
FQLDEEDCTPDDEEILDYISLWQEQ
>gi|7290033|gb|AAF45500.1| (AE003417) 1sc gene product [Drosophila melanogaster]
MTSICSSKFQQQHYQLTNSNIFLLQHQHHHQTQQHQLIAPKIPLGTSQLQNMQQSQQSNVGPMLSSQKKKFNYNNMPYGE
QLPSVARRNARERNRVKQVNNGFVNLRQHLPQTVVNSLSNGGRGSSKKLSKVDTLRIAVEYIRGLQDMLDDGTASSTRHI
YNSADESSNDGSSYNDYNDSLDSSQQFLTGATQSAQSHSYHSASPTPSYSGSEISGGGYIKQELQEQDLKFDSFDSFSDE
QPDDEELLDYISSWQEQ
>gi|7290034|gb|AAF45501.1| (AE003417) pcl gene product [Drosophila melanogaster]
MPRFQIKSLVFLAGLIVLVGEANSTLRRIPIQKSPNFKRSHKNIVAERDFVQQKYNRQYTANGYPMEHLSNYDNFQYYGN
ISIGTPGQDFLVQFDTGSSNLWVPGSSCISTACQDHQVFYKNKSSTYVANGTAFSITYGTGSVSGYLSVDCVGFAGLTIQ
SQTFGEVTTEQGTNFVDAYFDGILGMGFPSLAVDGVTPTFQNMMQQGLVQSPVFSFFLRDNGSVTFYGGELILGGSDPSL
YSGSLTYVNVVQAAYWKFQTDYIKVGSTSISTFAQAIADTGTSLIIAPQAQYDQISQLFNANSEGLFECSSTSYPDLIIN
INGVDFKIPAKYYIIEEEDFCSLAIQSINQDFWIMGDVFLGRIYTEFDVGNQRLGFAPVNSAVGLRKAVLWRIFTLLLAC
GMWKLKN
>gi|7290035|gb|AAF45502.1| (AE003417) ase gene product [Drosophila melanogaster]
MAALSFSPSPPPKENPKENPNPGIKTTLKPFGKITVHNVLSESGANALQQHIANQNTIIRKIRDFGMLGAVQSAAASTTN
TTPISSQRKRPLGESQKQNRHNQQNQQLSKTSVPAKKCKTNKKLAVERPPKAGTISHPHKSQSDQSFGTPGRKGLPLPQA
VARRNARERNRVKQVNNGFALLREKIPEEVSEAFEAQGAGRGASKKLSKVETLRMAVEYIRSLEKLLGFDFPPLNSQGNS
.......
```

Figura C.1.
Esempio di una porzione del file in formato FASTA multiplo, contenente cinque sequenze proteiche di *Drosophila*. Questo formato prevede che ogni sequenza inizi con una riga di commenti, preceduta dal simbolo '>'. In tal modo è possibile stabilire l'inizio e la fine di ogni sequenza, oltre che la rispettiva identità.

riconoscerà questi file nel loro insieme come *yeast-prot*. Ripetere poi la stessa operazione con le sequenze di *Drosophila*.

Per poter utilizzare Blast c'è bisogno anche di un file con la sequenza query, cioè la sequenza che si vuole cercare. Per mantenere ordine nei file, si crei una directory «*bioinfo/queryseq/*» in cui salvare le sequenze query. Queste potranno per esempio essere estratte dai file *droso-prot* e *yeast-prot* sfruttando il vantaggio di avere precedentemente usato l'opzione di *formadb* «-o». A tale scopo ci si porti all'interno della directory *bioinfo/queryseq/* e si digiti per esempio:

```
../blast/fastacmd -d droso-prot -s AAF45498
../blast/fastacmd -d droso-prot -s AAF45499
```

Il programma *fastacmd* ha la funzione di estrarre singole sequenze da un database formattato con l'opzione «-o» di *formatdb* e di farle apparire sul monitor. Ciò potrebbe non accadere qualora i nomi delle sequenze fossero cambiati; per accertarsene aprire il file *droso-prot* (a tale scopo si può utilizzare il comando `more droso-prot`) e cercare le sequenze riprodotte nella ▶Figura C.1 (digitare `/AAF4549` all'interno dell'ambiente «*more*» per effettuare la ricerca). In tal modo si possono visualizzare i nomi delle sequenze effettivamente presenti. Scegliere quindi una sigla corretta da utilizzare con il programma *fastacmd*. Per salvare l'output su un file usare il comando di ridirezionamento «`>`» come segue:

```
../blast/fastacmd -d droso-prot -s AAF45499 > AAF45499.aa
```

In tal modo l'output del programma *fastacmd* invece che apparire sullo schermo sarà salvato nel file «*AAF45499.aa*». Usare il comando «`ls`» per accertarsene. Poi usare il comando «`more AAF45499.aa`» per vedere il contenuto del file.

■ **Suggerimento** Molto spesso è necessario ripetere un comando, apportando piccole variazioni. Il sistema operativo Unix (o meglio, il programma chiamato shell che gestisce l'input dalla tastiera) consente di ripescare i comandi precedenti, utilizzando

i tasti «freccia su» e «freccia giù», mentre con la freccia destra e sinistra ci si può portare all'interno dei comandi per modificarli (vedere Appendice B.2.8).

C.4.3 Uso di Blast con comandi di linea

Il programma Blast è spesso usato in modo remoto, tramite interfacce web che ne rendono facile e intuitivo l'uso. La funzione di queste interfacce è di raccogliere e inviare al programma i dati e i parametri necessari a effettuare la ricerca di similarità. Se invece si dispone localmente di Blast, allora è più semplice utilizzare il comando di linea per passare direttamente i parametri al programma, specialmente se la ricerca di similarità è lanciata in modo automatico come nel caso di *SmistaBlast.pl*. In questo paragrafo è illustrata la sintassi da utilizzare per usare Blast con comando di linea.

Come già accennato nel Paragrafo C.4.1, il comando *blastall* stampa la lista degli argomenti che possono essere passati al programma. Per seguire l'esempio qui illustrato, ci si porti all'interno della directory «*bioinfo*» (Paragrafo C.4.1) e si digiti «*blast/blastall*». Il computer dovrebbe restituire un output simile a quello illustrato nella ►Tabella C.1.

Per lanciare il programma Blast dall'interno della directory «*bioinfo*» si digiti il seguente comando che presuppone che la sequenza query si chiami «*AAF45499.aa*» (Paragrafo C.3.2) e sia posta all'interno della directory «*bioinfo/db/*»:

```
blast/blastall -p blastp -d db/droso-prot -i db/AAF45499.aa -e 0.1 -F F -b 0 -v 1
```

In questo testo sono spiegate solo le opzioni utilizzate nel precedente comando, che saranno poi riprese dal programma *SmistaBlast.pl* illustrato nel prossimo paragrafo. Per approfondire il significato delle altre opzioni si consulti il manuale «in linea» di Blast. Prima di tutto si noti l'argomento «-p» che indica il sottoprogramma da utilizzare, nel caso specifico «blastp» (Paragrafo 3.2.4 per ulteriori dettagli su Blast). L'argomento «-d» specifica invece il database e l'argomento «-i» la sequenza query usata come input. Gli altri argomenti sono spesso superflui in una normale ricerca di similarità, ma come si vedrà più avanti sono opportuni nel programma *SmistaBlast.pl*.

L'argomento «-e» definisce il numero atteso (*Expected*) di «*High-scoring Segment Pairs*», quindi un valore di 0,1 (invece del valore di default di 10) comporterà che gli allineamenti poco significativi non saranno considerati dal programma (Paragrafo 3.2.4). Per meglio apprezzare l'effetto di questa impostazione, si provi a lanciare il programma modificando il parametro «-e» con valori di 10, oppure con valori bassi come 0,000001. È meglio fare queste prove sulla banca dati di *Drosophila* che contiene un numero maggiore di sequenze e quindi ha più probabilità di presentare un'ampia gamma di similarità casuali.

L'argomento «-F» definisce se la sequenza query debba essere mascherata nelle regioni a bassa complessità. A volte le sequenze di acidi nucleici e di proteine contengono segmenti a bassa complessità (si pensi al *poly(A)* negli mRNA) che, pur allineandosi con diverse sequenze della banca dati, non contribuiscono in modo positivo all'identificazione di vere omologie. Anzi, possono portare a risultati ingannevoli specialmente se il livello di similarità tra le due sequenze è molto basso. Porzioni di sequenza a bassa complessità sono presenti anche in proteine; per esempio, la stringa «QQQQQ», che in una sequenza casuale di 20 aminoacidi ha solo una probabilità su 3,2 milioni, è presente in entrambe le prime due sequenze della Figura C.1. Se l'opzione di filtro non fosse deliberatamente specificata, Blast la attiverebbe automaticamente (default) con la conseguenza che le eventuali regioni a bassa complessità sarebbero mascherate con delle «X». Nel caso specifico del programma *SmistaBlast.pl*, dal momento che si vogliono considerare solo sequenze ad alto grado di similarità, è più opportuno impostare questo parametro a «F» (*False*), altrimenti le regioni a bassa complessità sarebbero mascherate e di conseguenza escluse dall'allineamento. Per meglio apprezzare la differenza si provi a lanciare la stessa ricerca

Tabella C.1.
Argomenti che possono essere passati al programma Blast. Le abbreviazioni Int, Str, T/F e Real indicano rispettivamente Interi, Stringhe, Vero/Falso e numeri reali.

	Argomento	**Tipo di variabile**	**Valore di default**
-p	program name	Str	Obbligatorio
-d	database	Str	nr
-i	query file	Str	stdin
-e	expectation value (E)	Real	10.0
-m	options: 0 = pairwise; 1 = query-anchored showing identities; 2 = query-anchored no identities; 3 = flat query-anchored, show identities; 4 = flat query-anchored, no identities; 5 = query-anchored no identities and blunt ends; 6 = flat query-anchored, no identities and blunt ends; 7 = XML Blast output; 8 = tabular; 9 tabular with comment lines	Int	0
-o	BLAST report output file	Str	stdout
-F	filter query sequence (DUST with blastn, SEG with others)	Str	T
-G	cost to open a gap (zero invokes default behavior)	Int	0
-E	cost to extend a gap (zero invokes default behavior)	Int	0
-X	X dropoff value for gapped alignment (in bits) (zero invokes default behavior)	Int	0
-I	show GI's in deflines	T/F	F
-q	penalty for a nucleotide mismatch (blastn only)	Int	-3
-r	reward for a nucleotide match (blastn only)	Int	1
-v	number of database sequences to show one-line descriptions for (V)	Int	500
-b	number of database sequence to show alignments for (B)	Int	250
-f	threshold for extending hits, default if zero	Int	0
-g	perfom gapped alignment (not available with tblastx)	T/F	T
-Q	query genetic code to use	Int	1
-D	DB genetic code (for tblast[nx] only)	Int	1
-a	number of processors to use	Int	1
-O	SeqAlign file [File Out]	Str	optional
-J	believe the query defline	T/F	F
-M	matrix	Str	BLOSUM62
-W	word size, default if zero	Int	0
-z	effective length of the database (use zero for the real size)	Str	0
-K	number of best hits from a region to keep (off by default, if used a value of 100 is recommended)	Int	0
-P	0 for multiple hits 1-pass, 1 for single hit 1-pass, 2 for 2-pass	Int	0
-Y	effective length of the search space (use zero for the real size)	Real	0
-S	query strands to search against database (for blast[nx], and tblastx). 3 is both, 1 is top, 2 is bottom	Int	3
-T	produce HTML output	T/F	F
-l	restrict search of database to list of GI's	Str	optional
-U	use lower case filtering of FASTA sequence	T/F	F
-y	dropoff (X) for blast extensions in bits (0,0 invokes default behavior)	Real	0.0
-Z	X dropoff value for final gapped alignment (in bits)	Int	0
-R	PSI-TBLASTN checkpoint file [File In]	Str	optional
-n	megaBlast search	T/F	F
-L	location on query sequence	Str	optional
-A	multiple hits window size (zero for single hit algorithm)	Int	40

precedente, ma con il parametro «-F T», che essendo la condizione di default può essere omessa.

Infine gli argomenti «-b» e «-v» indicano rispettivamente il numero di allineamenti e il numero di risultati da visualizzare. Si consiglia ancora una volta di provare a cambiare i parametri dell'esempio precedente per apprezzare la variazione dell'output.

C.4.4 SmistaBlast.pl

Il programma *SmistaBlast.pl* presuppone che ci sia una directory contenente uno o più file di sequenze proteiche da analizzare. Un ulteriore presupposto è che i nomi dei file terminino con l'estensione «.aa» e che ogni file contenga una singola sequenza proteica. Lo scopo del programma è di analizzare ogni singola sequenza con Blast per cercare similarità tra le sequenze proteiche di lievito e di *Drosophila*. In base alle similarità trovate, le sequenze dovranno essere automaticamente smistate in quattro directory diverse, «*lievito*», «*droso*», «*lievidroso*» e «*newseq*», a seconda che siano state trovate rispettivamente solo nel lievito, solo in *Drosophila*, in entrambi o in nessuno dei due.

Il listato completo del programma *SmistaBlast.pl* è riportato nel ▶Box C.3. Molte funzioni di Perl sono già state illustrate nei paragrafi precedenti, quindi il programma *SmistaBlast.pl* non sarà commentato in ogni dettaglio, ma solo per quanto riguarda alcuni aspetti che non sono stati ancora coperti nei paragrafi precedenti. Altri punti del programma possono essere approfonditi consultando il manuale di Perl accessibile in rete (vedere Tabella C.2).

C.4.5 Le subroutine

Il programma *SmistaBlast.pl* fa uso di **subroutine** per effettuare delle operazioni complesse che devono essere ripetute in punti diversi del programma. Le subroutine sono poste alla fine del programma principale e comprendono la dichiarazione di subroutine, seguita da una serie di istruzioni poste tra parentesi graffe:

```
sub nomeSubroutine { ... istruzioni ... }
```

Per fare entrare il programma in una subroutine basta scrivere l'istruzione «`&nomeSubroutine()`». In tal modo il programma esegue le istruzioni della subroutine e, una volta terminate, torna all'istruzione successiva alla chiamata della subroutine.

Le variabili definite nel programma principale oppure all'interno di una subroutine sono normalmente visibili anche dall'interno di altre subroutine; nello stesso modo una variabile definita all'interno di una subroutine è visibile quando si torna al programma principale. Per questa ragione le variabili di Perl sono definite **variabili pubbliche**. In altri linguaggi di programmazione come C e Java le variabili sono invece normalmente **private**, quindi il loro scopo (e la loro visibilità) è limitato all'ambito della subroutine di cui fanno parte. In Perl, per rendere una variabile privata (cioè relativa esclusivamente alla subroutine in cui è stata definita), si deve usare l'istruzione «`my`», come illustrato poco più avanti.

Si consideri ora l'esempio della subroutine `printscore()`, che è chiamata in due punti diversi di *SmistaBlast.pl*, dopo avere lanciato le due ricerche di Blast. La sua funzione è di stampare il risultato di Blast che è già stato elaborato dalla subroutine `bestblastscore()`, ma se il valore è uguale a 20, allora deve stampare «`>1`». Si noti in primo luogo la modalità con cui il valore da stampare è passato alla subroutine, per esempio:

```
&printscore( $bestyeast );
```

Il valore della variabile `$bestyeast` è in tal modo passato all'interno della subroutine, nella variabile temporanea «`$_[0]`», analogamente a quanto descritto per le variabili

`$ARGV` (Paragrafo C.3.1). Se dovessero passare più valori (separati da virgole all'interno della parentesi rotonda) allora l'indice di `$_` servirebbe a distinguerli, con l'indice zero sempre corrispondente al primo valore. L'istruzione:

```
my $val = $_[0];
```

considera quindi il valore della variabile passata alla subroutine e lo copia in `$val`. L'istruzione «`my`» non è essenziale, ma limita la visibilità di `$val` alla sola subroutine; in tal modo non ci si deve preoccupare se un'altra variabile con lo stesso nome sia usata in un'altra parte del programma.

Sulla base di quanto detto sopra, ci si può chiedere perché sia necessario effettuare questa complessa operazione di trasferimento di valori quando, come si è visto, le variabili di Perl sono globali. Infatti, la variabile `$bestyeast` è accessibile anche dall'interno della subroutine (si provi per esempio ad aggiungere all'interno della subroutine l'istruzione `print $bestyeast;`). La risposta è abbastanza ovvia; si consideri un'altra volta l'esempio di *printscore()*, che, come si è visto, può essere eseguito dopo l'analisi di Blast sulle sequenze di lievito oppure su quelle di *Drosophila*. Una volta dentro la subroutine, il programma non può sapere se deve considerare `$bestyeast` oppure `$bestdroso`, invece, se il valore da considerare è passato alla variabile `$val` il programma non si deve preoccupare di nient'altro se non di procedere con il valore che gli è stato passato.

C.4.6 Gli array

Molto spesso è utile disporre di variabili indicizzate, per esempio `$giorno[1]` potrebbe contenere «lunedì», `$giorno[2]` «martedì», e così via. Un esempio di questo tipo è già stato visto nella trattazione delle variabili `$ARGV[]`.

Questo tipo di variabile complessa viene definito **array** (matrice) e in Perl è rappresentato nel suo insieme con il simbolo **@**. Nel listato del Box C.3 è definito un array nel seguente modo:

```
@opzDir=("newseq", "lievito", "droso", "lievidroso");
```

Il risultato è che `$opzDir[0]` contiene la stringa «*newseq*», `$opzDir[1]` contiene «*lievito*», ecc. Per sapere quanti elementi sono contenuti in un array si può usare `$#opzDir` che restituisce l'ultimo indice occupato, nel caso dell'esempio il valore 3.

Nel programma *SmistaBlast.pl* alcune proprietà degli array sono esemplificate con l'uso della funzione **split**, alla fine della subroutine `bestblastscore()`, in cui si trovano le seguenti righe:

```
@resline = split(/ /, $line);
$value = $resline[$#resline];
@resline = split(/e/, $value);
if ($#resline == 0 ) { ...
```

Le quattro righe riportate sopra hanno la funzione di analizzare i risultati di Blast, che generalmente si presentano con un formato simile al seguente:

```
gi|7293165|gb|AAF48549.1|(AE003500) Or13a gene product 537 e-153
```

Nel caso specifico si tratterebbe di estrarre il numero 153. Nella prima riga di programma la funzione split legge la stringa di caratteri (contenuta in `$line`) e a ogni spazio la spezza in tante parole che sono messe una dopo l'altra nell'array `@resline`. Nella seconda riga, l'ultimo elemento (con indice `$#resline`) dell'array (che nel caso dell'esempio sarebbe *e-153*) viene assegnato alla variabile `$value`. Nella terza riga il contenuto di `$value` viene ulteriormente suddiviso a ogni «e». Infine, nella quarta riga, il programma valuta il nuovo valore di `$#resline` per stabilire se sia stata trovata la lettera «e»; infatti in molti casi Blast restituisce questo valore in forma non esponenziale (per esempio 0,054 nel caso di una similarità molto bassa, oppure 0 se la similarità è altissima).

Box C.3 **LISTATO DEL PROGRAMMA SMISTABLAST.P**

```
#!/usr/bin/perl
##############################################################################
# Script:  SmistaBlast.pl                                                    #
# Questo programma legge i nomi dei file contenuti nella directory specificata #
# nella linea di comando e considera quelli con un nome contenente ".aa", che  #
# assume contengano sequenze di proteine. I file sono poi automaticamente      #
# smistati in quattro directory diverse in funzione delle similarità trovate.  #
##############################################################################

# definisce i percorsi dei database di blast e altre variabili
$yeastdb="db/yeast-prot";
$drosodb="db/droso-prot";
@opzDir=("newseq", "lievito", "droso", "lievidroso");
$blast="/usr/local/blast/blastall -p blastp ";

# apre la directory indicata in linea di comando
die "can't opendir $ARGV[0]\n$!" unless opendir(DIR, $ARGV[0]);

# se non esistono le directory di destinazione le crea
for ($conta=0; $conta<4; $conta++){
   die "Programma terminato" unless &verifica_dir("$opzDir[$conta]");
}

# definisce valori sotto i quali le sequenze sono considerate "diverse"
$soglia_yeast = -20 ;
$soglia_droso = -15 ;

# considera, uno alla volta, tutti i file il cui nome finisce con ".aa"
while($file= readdir(DIR)) { # In questo modo legge un nome di file per volta
   chomp $file;  # Questo comando toglie l'a capo dalla fine della riga
   next unless ($file =~ /\.aa$/);  #  pattern matching
   print "\n******************** File $file **********************\n";

   #  lievito
   print "Blast $file su lievito... ";
   system "$blast -d $yeastdb -i $file -e 0.1 -F F -b 0 -v 1 -o blastres.tmp";
   $bestyeast = &bestblastscore( "blastres.tmp" );
   &printscore( $bestyeast );

   #  drosophila
   print "Blast $file su Drosoph... ";
   system "$blast -d $drosodb -i $file -e 0.1 -F F -b 0 -v 1 -o blastres.tmp";
   $bestdroso = &bestblastscore( "blastres.tmp" );
   &printscore( $bestdroso );
   unlink <*.tmp>;  # elimina i file con estensione .tmp

   # smista il file corrente
   $bina = 0;
   $bina |= 1 if ($bestyeast < $soglia_yeast) || ($bestyeast==0);
   $bina |= 2 if ($bestdroso < $soglia_droso) || ($bestdroso==0);
   die "Imposs. trasferire $file" unless rename($file, "$opzDir[$bina]/$file");
   print "File $file trasferito nella directory $opzDir[$bina]\n";
}
closedir DIR;
exit(0);


##############################################################################
#                        V E R I F I C A _ D I R                             #
#  Questa subroutine verifica se una directory esiste, altrimenti la crea    #
#  Ritorna 1 se OK, altrimenti ritorna 0                                     #
##############################################################################
sub verifica_dir {
   my $nomedir = $_[0];           # -d $nomedir ritorna TRUE se la dir esiste...
   unless( -d $nomedir) {         # quindi entra qui se la directory non esiste
      unless(mkdir($nomedir)) {   # entra anche qui se non riesce a crearla
          print "Impossibile creare la directory $nomedir\n";
          return(0);
```

```
      }
      print "La directory '$nomedir' era assente, quindi e' stata creata\n"
   }
   return(1);
}
[ continua ]


################################################################################
#               S T A M P A   R I G A   D I   R I S U L T A T O                #
#  Se il numero passato alla sub e' 20 allora stampa "max e-value > 1",        #
#  altrimenti stampa max e-value = numero passato.                             #
################################################################################
sub printscore {
   my $val = $_[0];
   print "  max e-value ";
   if ($val == 20) { print "> 1\n" ;}
   else { print "= $val\n" ;}
}


################################################################################
#                        B E S T   B L A S T   S C O R E                       #
#            Questa subroutine ritorna il migliore e-value di BLAST.           #
#              Oppure 1 se il valore non e' un numero esponenziale             #
#                 Oppure 20 se non trova nessuna similarita'                   #
################################################################################
sub bestblastscore {
   my $blastFileName = $_[0];
   open( BlastFile, $blastFileName ) || die "File $blastFileName ?!?!\n";

   # Step 1: trova la prima linea che inizia con "Database:"
   $flag=0;
   while( $line = <BlastFile> ) {
      if(substr($line, 0, 9) eq "Database:") {$flag=1; last;}
   }
   $flag || die "Impossibile interpretare i risultati di BLAST (Errore 1)\n";

   # Step 2: prosegue fino alla riga che inizia con ' **' oppure 'Seq'
   $flag=0;
   while( $line = <BlastFile> ) {
      if(substr($line, 0, 3) eq " **") {
         $flag=1;  #  ***** No hits found ******
         last;
      }
      elsif (substr($line, 0, 19) eq "Sequences producing") {
         $flag=2;
         last;
      }
   }
   $flag || die "Impossibile interpretare i risultati di BLAST (Errore 2)\n";
   return(20) if ($flag == 1);

   # c'e' un risultato (puo' essere trovato due righe piu' sotto)
   $line = <BlastFile>; # salta una riga ...
   $line = <BlastFile>; # ... e ora ha preso la riga con l'e-value
   close( BlastFile );
   chomp( $line ); # questa istruzione elimina l'a-capo alla fine della riga

   @resline = split(/ /, $line);  # separa la riga ad ogni spazio
   $value=$resline[$#resline];    # porzione della riga dopo l'ultimo spazio
   @resline = split(/e/, $value); # separa alla 'e' (se c'e')
   if ($#resline == 0 ) {         # Non c'e' nessuna 'e' ...
      if($value>0) {return (1);}  # ... se non e' zero ritorna 1
      else {return (0);}          # altrimento ritorna zero
   }                              # Se invece c'era una 'e' ...
   return ($resline[1]);          # ... ritorna il numero dopo la 'e'
}
```

C.4.7 Pattern matching

Il linguaggio Perl offre delle ottime possibilità di analisi e di elaborazione di stringhe. Tra queste, le funzioni di *pattern matching* sono veramente potenti (vedere anche il Capitolo 5 e il Box 5.1). Questo testo non può trattare in dettaglio la vasta serie di opzioni offerte da Perl per il pattern matching, pertanto si rimanda al manuale di Perl disponibile in rete, per ulteriori informazioni su questo argomento.

Generalmente (ma non necessariamente) il pattern da cercare è racchiuso tra due barre: (`/..pattern../`). Se per esempio si volesse verificare la presenza di una cifra numerica seguita da un carattere maiuscolo, seguito a sua volta da una «X» si dovrebbe usare `/[0-9][A-Z]X/` . Nel programma *SmistaBlast.pl* è usata la seguente funzione di pattern matching:

```
next unless ($file =~ /\.aa$/);
```

che istruisce il programma a passare al prossimo caso (`next`) a meno che si verifichi la condizione (`unless`) che la variabile `$file` contenga una stringa che termina con i caratteri «.aa». In questo comando sono presenti due complicazioni; la prima è dovuta al fatto che il pattern cercato deve essere posto alla fine della stringa. Il simbolo `$` posto alla fine del pattern indica proprio che il pattern deve essere alla fine della stringa.

La seconda complicazione è dovuta al fatto che si vuole cercare il carattere «.». Infatti il punto ha il significato particolare di rappresentare qualsiasi carattere. Per informare il programma che si vuole proprio il punto bisogna anteporre al punto la barra rovesciata «\.».

Oltre alle istruzioni di pattern matching sopra illustrate, Perl dispone della possibilità di associare il pattern matching a funzioni di trasformazione di testo. Per esempio la funzione:

```
tr/a-zA-Z/ /cs;
```

cambia qualsiasi carattere non alfanumerico in spazio, mentre la funzione:

```
$cognome =~ tr/a-z/A-Z/;
```

cambia tutti i caratteri minuscoli della variabile `$cognome` in caratteri maiuscoli.

C.4.8 Altre particolarità di SmistaBlast.pl

Il programma *SmistaBlast.pl* presenta diverse altre particolarità che meritano attenzione, per esempio l'istruzione `system` , che consente di eseguire comandi direttamente a livello di sistema operativo.

Nell'esempio illustrato si tratta di lanciare il programma Blast. Se il comando «`system` » è di facile comprensione, il seguente comando usato nella subroutine `best-blastscore()` è molto più ostico:

```
$flag || die " ... ";
```

La doppia barra verticale rappresenta l'operazione logica `OR`. Spesso questa operazione è usata per verificare una doppia condizione, per esempio:

```
$tono = "chiaro" if(($colore eq "bianco") || ($colore eq "celeste"));
```

Con l'operatore logico `OR` basta che una delle condizioni definite sia soddisfatta affinché l'esito finale sia *true*. Nello stesso modo, l'operazione logica `AND` (abbreviata con il simbolo `&&`) richiede che entrambe le condizioni siano soddisfatte per avere un esito finale *true*. È interessante notare che se la prima condizione è soddisfatta, l'operazione logica OR non ha bisogno di verificare la seconda condizione. Quindi la seconda condizione viene verificata solo se la prima è *false*.

Detto questo, l'istruzione `$flag || die "…"` significa: «*o $flag è vera (cioè non è*

zero) o muori!». In altre parole il programma muore solo se `$flag` ha un valore diverso da zero.

L'operatore `OR` logico non deve essere confuso con l'operatore `OR` «bitwise» che è rappresentato da una sola barra verticale. Allo stesso modo l'operatore `AND` bitwise è rappresentato da un solo «&». Le operazioni bitwise riguardano l'aritmetica binaria, dove il numero binario 1111 equivale al numero decimale 15, mentre il numero binario 1010 equivale al numero decimale 10 e così via. È facile calcolare il valore decimale di un numero binario: basta attribuire 1 alla prima cifra (quella più a destra), 2 alla seconda, 4 alla terza, 8 alla quarta, ecc. L'ennesima cifra avrà quindi un valore pari a 2^{n-1}.

Gli operatori `OR` e `AND` bitwise agiscono su due numeri binari, bit per bit. In altre parole prima vengono considerati i due bit che occupano la prima posizione nei due operandi; se l'operazione è `OR` allora basta che uno dei due bit sia positivo perché il corrispondente primo bit del risultato sia positivo. Invece, se l'operazione è `AND` è necessario che entrambi i bit siano positivi affinché anche il risultato sia positivo. L'operazione è ripetuta su tutti i bit dei due operandi per avere il risultato finale. Per esempio, tutte le seguenti espressioni sono corrette:

```
1001 | 0100 = 1101;      0001 | 0010 = 0011;      1111 | 1010 = 1111;
1001 & 0100 = 0000;      0001 & 0010 = 0000;      1111 & 1010 = 1010;
```

L'aritmetica binaria è stata utilizzata da *SmistaBlast.pl* per definire la directory in cui destinare il file dopo l'analisi con Blast. Ognuna delle quattro directory di destinazione può essere definita dalla corrispondente posizione nell'array `@opzDir` . La directory *newseq* ha indice 0, *lievito* ha indice 1, *droso* ha indice 2 e *lievidroso* ha indice 3. Quindi, sulla base dei risultati di Blast, bisogna definire un numero da 0 a 3 che rappresenti la directory di destinazione; questa è calcolata nella variabile `$bina` come segue:

```
$bina = 0;
   $bina |= 1 if ($bestyeast < $soglia_yeast) || ($bestyeast==0);
   $bina |= 2 if ($bestdroso < $soglia_droso) || ($bestdroso==0);
```

Si consideri che le espressioni che seguono l'`if` nella seconda e nella terza riga sono true se è stata superata la soglia di similarità rispettivamente in lievito e in *Drosophila*. Si tornerà più avanti su questo punto. Si inizia con il definire la variabile `$bina=0;` poi, se è stata trovata una similarità con le sequenze di lievito, si esegue l'operazione `$bina |= 1;` che equivale a `$bina = $bina | 1;` rende positivo il primo bit della variabile `$bina` .

Allo stesso modo `$bina |= 2;` nella riga successiva rende positivo il secondo bit di `$bina` se è stata trovata similarità con *Drosophila*. Il risultato finale di queste due semplici operazioni è che se non viene trovata alcuna similarità `$bina` resta al suo valore iniziale di 0; se viene trovata similarità solo con il lievito diventa 1; se viene trovata similarità solo con *Drosophila* diventa 2; infine, se viene trovata similarità sia con l'uno che con l'altro, allora i due bit assumono entrambi valore positivo e il valore di `$bina` diventa 3.

Nella seconda parte delle due righe di programma illustrate sopra viene verificato se il livello di similarità sia superiore alla soglia. Il numero che si ricava dai risultati di Blast è un esponente negativo oppure uno zero che indica la quasi identità di due sequenze. Il valore esponenziale e lo zero devono essere considerati separatamente e sono valutati con un operatore logico `OR`.

In altre parole, perché si possa considerare simile una sequenza analizzata da Blast deve avere uno score pari a zero, oppure il numero estratto dalla subroutine `best-blastscore()` (cioè l'esponente negativo) deve avere un valore superiore alla soglia prefissata.

C.5 Principali risorse disponibili in rete

Tabella C.2

Descrizione	URL
Perl	
home page di Perl	http://www.perl.com
sviluppo di Perl	http://www.deja.com/
manuali e altro materiale	http://www.activestate.com/
Altri siti utili	
download di sequenze (NCBI)	ftp://ftp.ncbi.nih.gov/pub/koonin/
download di sequenze (MIPS)	http://mips.gsf.de/
blast	ftp://ftp.ncbi.nih.gov/blast/executables/