Introduzione al Perl

Dispense del corso di introduzione alla programmazione in Perl

Marco Liverani

liverani@mat.uniroma1.it – marco@isinet.it
Roma, Settembre 1996

Questo fascicolo è stato prodotto utilizzando il software LaTeX 2.09 in ambiente Linux. I numerosi esempi presenti nel testo sono stati verificati utilizzando l'interprete Perl v. 5.001 in ambiente Linux.

Prima versione Settembre 1996

Indice

Introduzione

Il linguaggio Perl è uno strumento formidabile: al tempo stesso un po' magico ed estremamente elegante, potente e compatto. Superate le prime resistenze di chi tradizionalmente ha avuto a che fare con linguaggi di programmazione come C, Fortran o Pascal, il Perl si lascerà apprezzare per le sue doti straordinarie. Innanzi tutto la compattezza: oltre ad essere uno dei suoi maggiori pregi è all'inizio anche il più grosso ostacolo; gli script dei programmatori esperti risultano essere molto sintetici e poco comprensibili per un principiante. Con il tempo, acquistando familiarità con il linguaggio tutto apparirà estremamente chiaro e l'eleganza del Perl potrà essere apprezzata in ogni sua parte. Anzi potrà anche accadere di trovare fastidiose certe ridondanze e la pignoleria delle definizioni presenti in altri linguaggi di programmazione. Infine la potenza: con poche istruzioni saremo in grado di compiere operazioni che con altri strumenti sarebbero risultate talmente complesse o noiose che spesso si sarebbe cercato di agirare il temuto ostacolo. Col passare del tempo e con l'aumentare della mia esperienza nell'uso del Perl non è affatto diminuito lo stupore per l'eleganza, la sinteticità e l'efficacia con cui il Perl mi premette di effettuare determinate operazioni: è per questo che lo considero uno strumento circondato da un alone di *magia* che non si finisce mai di studiare e che rivela in ogni occasione un nuovo modo in cui il medesimo algoritmo può essere implementato.

Oggi il Perl gode di una certa notorietà in quanto è probabilmente lo strumento più usato per sviluppare script CGI su server HTTP nel mondo Internet ed Intranet.

Il linguaggio Perl si sposa perfettamente con il sistema UNIX; anche se oggi ne sono stati effettuati dei *porting* su numerose altre piattaforme (OS/2, MS-DOS, Windows 95, Windows NT, solo per citarne alcune) l'ambiente più naturale per sviluppare ed eseguire script in Perl rimane UNIX, dove è possibile integrare e far interagire gli script con i numerosi strumenti presenti nel sistema (sto pensando a *sed*, *awk*, *grep*, *mail* e più in generale alle modalità operative e di intercomunicazione tra i processi offerte da questo ambiente) ed ottenere il meglio dal linguaggio.

Chi ha già un po' di esperienza con UNIX e con alcuni suoi strumenti (*vi*, *grep*, ecc.) si troverà un passo avanti rispetto ad altri nell'esplorazione del linguaggio. Tuttavia questo brevissimo fascicolo è stato pensato come uno strumento elementare per un primo approccio al Perl: nessun prerequisito è richiesto per affrontare la lettura di queste pagine; è però utile poter disporre di un interprete Perl per effettuare direttamente le prove ed i test necessari, anche solo sperimentando gli esempi proposti nei vari capitoli.

Questo fascicolo non ha la pretesa di trattare in modo esaustivo l'argomento: si tratta semplicemente di un'introduzione elementare per chi non ha mai utilizzato il Perl e, prima di affrontare letture più approfondite ed impegnative, decide di voler dare uno sguardo ad ampio raggio alle potenzialità offerte dal linguaggio.

Per chi desiderasse sapere tutto sul Perl consiglio sicuramente la lettura dell'ottimo testo di Larry Wall (autore del Perl) e Randal L. Schwartz, *Programming Perl*, edito dalla O'Reilly & Associates (1991). Questo libro ha un approccio didatticamente assai interessante: in un primo capitolo descrive con una serie di esempi pratici le principali caratteristiche del linguaggio, mettendo l'utente smaliziato nelle condizioni di scrivere i primi script di una certa sostanza. Seguono quindi un paio di capitoli in cui si entra nei dettagli dei vari aspetti del Perl. Infine ampio spazio è lasciato alla soluzione di problemi pratici (*practical programming*) anche di una certa complessità.

Una guida sintetica di riferimento è invece offerta dal libricino di Johan Vromans, *Perl 5 – Desktop Reference*, edito dalla stessa O'Reilly & Associates (1996).

Per un approccio più tradizionale e "passo-passo" al linguaggio, è anche da ricordare il testo di Randal L. Schwartz, *Learning Perl*, sempre della O'Reilly & Associates (1993).

Questo fascicolo è strutturato in otto brevi capitoli. Il primo è una introduzione sintetica ed informale alla programmazione ed ai concetti riguardanti le strutture dati. Sono stato a lungo incerto sull'opportunità di inserire un capitolo come questo in una guida introduttiva al Perl. Alla fine ho deciso di lasciare questo primo capitolo nella speranza che possa essere utile a quanti si accostano per la prima volta alla programmazione e quindi necessitano se non altro di una prima infarinatura sulla terminologia usata in questo contesto.

Il secondo capitolo introduce, in modo forse un po' brusco ed informale, le basi del linguaggio Perl. Nel capitolo 3 questi concetti vengono sviluppati introducendo le strutture dati in Perl. Nel capitolo 4 vengono affrontati gli operatori logici e le strutture di controllo del linguaggio. Il capitolo 5 affronta uno degli aspetti più importanti ed interessanti del linguaggio: le *espressioni regolari* ed il *pattern matching*. Nel sesto capitolo vengono esaminate in estrema sintesi alcune delle *variabili speciali* usate dal linguaggio, mentre nel capitolo 7 si affronta il problema dell'interazione di uno script Perl con altri programmi esterni. Il capitolo 8 riporta una descrizione sintetica delle principali istruzioni divise per categorie funzionali.

Come si intuirà fin dalle prime pagine, in questo fascicolo non sono mai entrato nei dettagli di ciò che espongo: descrivere ogni particolare sintattico delle istruzioni del linguaggio sarebbe stato assai noioso ed avrebbe trasformato questa guida introduttiva in un manuale tecnico. L'approccio scelto è stato molto diverso: ho preferito fornire una serie di flash ad ampio spettro sulle possibilità offerte dal linguaggio, lasciando il compito di chiarire (o far intuire) alcuni dettagli sintattici ai numerosi esempi e tabelle presenti in ogni capitolo.

Le convenzioni tipografiche impiegate nel testo sono quelle ormai consuete per questo genere di pubblicazione: con il carattere `typewriter` ho indicato le istruzioni Perl o l'output di un comando; il carattere *slanted* è stato usato per rappresentare l'input immesso dall'utente, mentre nella descrizione delle istruzioni il carattere *italic* è stato usato per descrivere ciò che dovrà essere sostituito con le variabili e le espressioni vere e proprie nel programma dell'utente.

Spero con questo fascicolo di avere effettivamente fornito un utile strumento didattico e non soltanto di aver riprodotto in forma sintetica ciò che meglio di me hanno già scritto altri autori. Ogni eventuale commento, suggerimento, critica, sarà bene accetto: le mie mailbox sono sempre aperte per chiunque!

M. L. (marco@isinet.it, liverani@mat.uniroma1.it)
Roma, Settembre 1996

Capitolo 1

Introduzione alla programmazione

Questo primo capitolo è da intendersi un po' come un capitolo zero: tratteremo in modo estremamente sintetico e con una terminologia necessariamente informale e talvolta priva del necessario rigore gli aspetti di base della programmazione. Con queste pagine non si intende certo fornire uno strumento esaustivo per chi non ha mai scritto un programma per un calcolatore elettronico, tuttavia potranno essere utili ai lettori meno esperti per acquisire una terminologia che riprenderemo spesso nei capitoli seguenti.

1.1 Algoritmi

Un computer è una macchina costituita da un insieme di componenti elettroniche: è facile immaginare quindi, come tutto il funzionamento della macchina dipenda da un flusso continuo di segnali elettrici, sincronizzati da un clock, che codificano, in notazione binaria, le informazioni immagazzinate ed elaborate all'interno della macchina stessa.

Per quanto possa essere complessa e sofisticata l'architettura di un elaboratore elettronico, questo potrebbe sembrare non molto dissimile da un qualsiasi altro elettrodomestico presente nella nostra casa. Tuttavia esiste una differenza sostanziale tra un computer ed una lavatrice: per quanto moderna e dotata di sofisticati meccanismi elettronici, una lavatrice potrà sempre e solo lavare dei panni, mentre il computer può essere *istruito* per svolgere compiti anche molto differenti tra di loro.

Come si istruisce un computer? Mediante un *programma*, ossia tramite una sequenza ben definita di istruzioni che la macchina è in grado di tradurre in funzioni elementari facilmente eseguibili. Schematicamente possiamo dire che le caratteristiche principali di un calcolatore sono:

- *velocità*: è rapidissimo nell'eseguire i compiti che gli vengono assegnati;
- *precisione*: è estremamente preciso nell'eseguire calcoli matematici e logici;
- *affidabilità*: svolge con rigore e puntualità i compiti assegnati dall'utente;
- *duttilità*: è in grado (se programmato correttamente) di svolgere lavori assai diversi.

Come rovescio della medaglia di queste caratteristiche "positive", possiamo individuare alcune caratteristiche "negative" di cui è bene tenere conto:

- è privo di "buon senso" e di intuizione;
- è incapace di verificare se c'è corrispondenza tra quanto sta facendo e gli obbiettivi del programmatore.

Riassumendo, possiamo dire che il calcolatore è un esecutore di ordini impartiti dall'utente-programmatore, assai preciso ed efficiente, ma un po' stupido, si limita cioè ad eseguire ciò che gli viene chiesto, a prescindere dalla correttezza di quanto gli viene ordinato di compiere ai fini della soluzione del problema (in un certo senso si può dire che è un po' cieco).

Per capire meglio quanto è stato detto vediamo un esempio elementare di programma per un calcolatore elettronico. Supponiamo che il problema da risolvere sia quello di effettuare la media aritmetica tra 5 numeri forniti dall'utente. Il calcolatore deve leggere in input i cinque numeri, sommarli, memorizzando la somma da qualche parte nella sua memoria ed infine dividere per cinque la somma totale e stampare il risultato. Una *codifica* di questo *algoritmo* risolutore potrebbe essere la seguente:

1. *assegna* ad n il valore 0
2. *assegna* ad s il valore 0
3. *leggi* un numero in input ed *assegnalo* ad a
4. *incrementa* di 1 il valore di n
5. *somma* a ed s ed *assegna* il risultato ad s
6. *se* n è minore di 5 *vai* al passo 3 *altrimenti* prosegui
7. *dividi* s per 5 ed *assegna* il risultato ad m
8. *visualizza* il valore di m.
9. *fermati*

Eseguendo in sequenza le istruzioni del programma (che sono evidentemente molto elementari), il calcolatore è in grado di risolvere il problema che noi ci eravamo posti. Naturalmente l'algoritmo è espresso in lingua italiana: per essere capito dal computer sarebbe stato necessario tradurlo in *linguaggio macchina* (un linguaggio assai ostico ed innaturale, basato essenzialmente sulla codifica binaria su cui il computer è in grado di operare). Il linguaggio macchina ed il nostro linguaggio naturale sono estremamente distanti fra di loro: quest'ultimo pieno di ambigue sfumature, estremamente coinciso e criptico il primo; per raggiungere un buon compromesso tra le due parti che si devono comprendere (noi e la macchina) sono stati progettati numerosi *linguaggi di programmazione ad alto livello*, ovvero un insieme di parole chiave e di regole sintattiche, non troppo distanti dal nostro linguaggio naturale, ma facilmente traducibili (in modo automatico, mediante un programma chiamato *compilatore*) in istruzioni di linguaggio macchina.

Ogni programma per un calcolatore elettronico, utilizza la logica ed i concetti espressi (in estrema sintesi) nei paragrafi precedenti. È bene saperlo, anche se magari non ci si troverà mai a dover scrivere un programma, perché questo può aiutarci a capire che quasi sempre, quando il calcolatore non esegue ciò che noi vorremmo, a meno di guasti hardware non è lui che sbaglia, ma siamo noi ad aver fornito delle istruzioni incomplete o fuorvianti che non lo portano a raggiungere la soluzione cercata.

1.2 Diagrammi di flusso

È spesso utile aiutarsi nella progettazione di un algoritmo, mediante la stesura di appositi *diagrammi di flusso* che, con una simbologia standard, ci permettono di rappresentare graficamente il flusso seguito dall'elaboratore durante l'esecuzione del nostro programma.

In generale, le operazioni fondamentali che è possibile compiere mediante un programma deterministico sono cinque:

1. leggere un input dall'esterno e memorizzarlo in una cella di memoria;
2. compiere operazioni elementari sui dati contenuti nelle celle di memoria, memorizzando il risultato di tali operazioni in altre celle;
3. scrivere in output il contenuto di una cella di memoria;
4. confrontare il contenuto di due celle di memoria ed effettuare due operazioni differenti in base all'esito di tale confronto;
5. effettuare dei salti che modifichino il flusso sequenziale dell'algoritmo.

Nella rappresentazione di un algoritmo mediante un diagramma di flusso si usa un simbolo diverso per ognuna di queste operazioni:

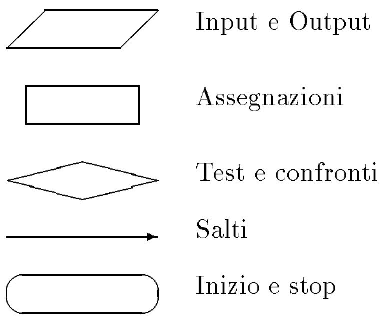

Figura 1.1: Simbologia utilizzata nella stesura di diagrammi di flusso

L'esempio riportato nella pagina precedente potrebbe quindi essere rappresentato con il diagramma di flusso rappresentato in figura 1.2.

1.3 Programmazione strutturata

Per rendere più chiari e leggibili i programmi è opportuno rifarsi alle regole della *programmazione strutturata*. Questo paradigma impone l'uso esclusivo di tre strutture di base per la costruzione di ogni algoritmo. Queste tre strutture possono essere combinate tra loro, giustapponendole o nidificandole una dentro l'altra. Le tre strutture sono rappresentate in figura 1.3.

Ciò che è assolutamente "proibito" dalle regole della programmazione strutturata è l'inserimento di salti ("*go to*") al di fuori delle strutture iterativa e condizionale; questi salti renderebbero particolarmente illeggibile e poco ordinata la struttura del programma, con pesanti conseguenze sulla sua efficienza e sulla possibilità di apportarvi delle correzioni. In caso di programmi scritti con uno stile particolarmente disordinato si parla di "spaghetti programming".

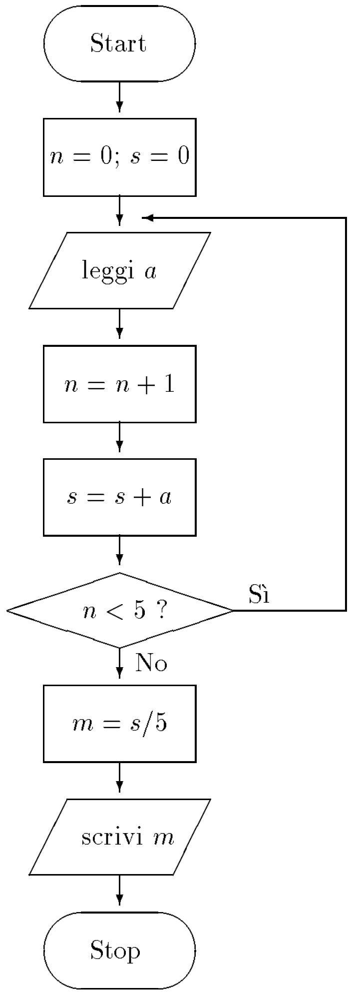

Figura 1.2: Diagramma di flusso dell'esempio delle pagine precedenti

1.4 Compilatori ed interpreti

Per poter essere eseguito da un elaboratore elettronico, ogni algoritmo deve essere codificato mediante le parole chiave di un *linguaggio di programmazione*. Un simile linguaggio è costituito da un insieme di parole e di regole sintattiche che ci consentono di tradurre in termini precisi e non ambigui i singoli passi che costituiscono un algoritmo. In un certo senso si può anche pensare ad un linguaggio di programmazione, come ad un tramite tra la nostra lingua parlata (ricca di forme e sfumature che la rendono ambigua ed inadatta a comunicare con una macchina) ed il linguaggio dell'elaboratore elettronico, il cosiddetto linguaggio macchina (che al contrario della nostra lingua parlata è un linguaggio assai povero di espressioni, ma dotato di una precisione "millimetrica").

Il calcolatore non è in grado di tradurre in operazioni da eseguire le istruzioni codificate mediante un linguaggio di programmazione. L'unico linguaggio con cui il computer è in grado di operare è il linguaggio macchina. Dobbiamo quindi dotarci di appositi programmi (scritti in linguaggio macchina!) che si occupino della traduzione delle istruzioni dei nostri programmi scritti con un linguaggio di programmazione, nel linguaggio della macchina.

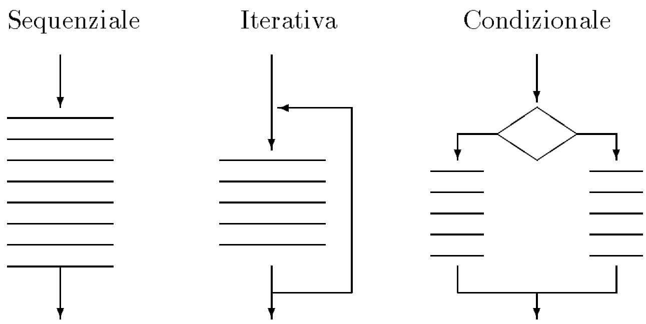

Figura 1.3: Le tre regole della programmazione strutturata

Questo tipo di programmi *traduttori* sono classificabili in due categorie principali, in base al loro modo di operare la traduzione:

compilatori: effettuano la traduzione delle righe del programma una volta per tutte e producono in output un programma in linguaggio macchina equivalente a quello fornito in input dal programmatore; il programma *compilato* potrà poi essere eseguito più volte senza che sia necessaria ogni volta una operazione di traduzione;

interpreti: effettuano la traduzione di una istruzione di programma per volta e la eseguono immediatamente dopo averla tradotta; quindi ogni volta che si esegue un programma l'interprete lo traduce un'istruzione dopo l'altra durante l'esecuzione.

Il vantaggio dell'uso di un interprete rispetto ad un compilatore sta nel fatto che il programma sorgente può essere modificato in ogni momento e poi inviato immediatamente in esecuzione, senza la necessità di dover eseguire ulteriori passaggi. In generale però, un programma compilato è molto più efficiente di un programma interpretato. Il Perl[1], oggetto di queste dispense, è un linguaggio interpretato, ma garantisce una elevata efficienza grazie al fatto che integra delle macro istruzioni già pronte per essere utilizzate. Uno dei più famosi linguaggi di programmazione, il BASIC[2], è spesso un linguaggio interpretato. Viceversa il C, il Fortran ed il Pascal sono dei linguaggi dotati di un compilatore.

1.5 Cenni sulle strutture dati

Abbiamo accennato nelle pagine precedenti al fatto che l'elaboratore elettronico è in grado di memorizzare i dati su cui opera in apposite *celle di memoria*. Queste celle possono essere utilizzate con un linguaggio di programmazione di alto livello, mediante l'uso delle variabili.

1.5.1 Variabili e tipi

Le *variabili* sono delle strutture in cui è possibile memorizzare dei dati. L'uso delle variabili semplifica di molto l'accesso diretto alle *locazioni di memoria*[3]. La memoria

[1] Perl è un acronimo: Practical Extraction and Report Language
[2] BASIC: Beginner Allporpouse Symbolic Instruction Code
[3] La memoria RAM (Random Access Memory) di un calcolatore elettronico è organizzata in celle, chiamate *locazioni di memoria*; ogni cella è identificata da un indirizzo numerico.

RAM di un computer è un insieme molto grande di *bit*, che possono assumere il valore di zero o uno (acceso o spento). Un gruppo di 8 bit costituisce un *byte*; gruppi di due, tre, quattro, otto byte costituiscono una parola, la cui lunghezza varia a seconda dell'architettura hardware della macchina (si parla in questo caso di macchine a 16, 24, 32 o 64 bit).

Per rappresentare in notazione binaria un numero intero compreso tra 0 e 255, sono necessari 8 bit (1 byte). Ogni carattere alfabetico e di punteggiatura può essere codificato mediante un numero compreso tra 0 e 127, quindi per rappresentare un carattere alfanumerico nella memoria di un computer è sufficiente un byte. Per memorizzare una parola di 10 caratteri saranno necessari 10 byte.

Esistono diversi tipi di dato che possono essere memorizzati all'interno di un calcolatore elettronico, in particolare di solito distinguiamo i seguenti:

- numeri interi;
- numeri razionali, cioè i numeri "con la virgola" (*floating point*);
- caratteri alfanumerici;
- stringhe (sequenze di caratteri alfanumerici);
- puntatori.

I puntatori sono un tipo di dato fondamentale in un linguaggio come il C. In pratica un puntatore è una struttura dati che permette di memorizzare l'indirizzo di memoria di un'altra variabile. In Perl non faremo uso di puntatori, visto che non si accede mai direttamente alle locazioni di memoria della macchina.

La maggior comodità nell'uso delle variabili sta nel non doversi preoccupare (se non in casi particolari) della dimensione in bit del dato che intendiamo trattare e nel poter attribuire un identificativo mnemonico (nome della variabile) alle variabili stesse.

Così invece di dover lavorare a diretto contatto con le locazioni di memoria e i loro indirizzi, per memorizzare un numero o una parola ci basterà assegnarla ad una variabile: "`$a=13`", oppure "`$b="casa"`", "`$nome="Marco"`", "`$x=$y+$z-17`" e così via.

I linguaggi di programmazione tradizionali, come il Fortran, il Pascal o il BASIC, richiedono che sia definita a priori il tipo di ogni variabile: se la variabile "`a`" è di tipo numerico intero, non potrà contenere null'altro che numeri interi; se la variabile "`b`" è una variabile stringa, non potrà contenere altro che sequenze di caratteri alfanumerici. Inoltre le operazioni consentite sulle variabili dipenderanno dal tipo di variabili usate (ad esempio sarà possibile effettuare il prodotto tra due numeri, ma non tra due stringhe).

In Perl (ma già in C è presente questo principio) i tipi di dato tendono a confondersi; in ogni caso l'interprete del linguaggio si regola di volta in volta in base al contesto. Ad esempio se imposteremo "`$a=3`", allora vorrà dire che la variabile `$a` è intera, mentre impostando "`$a="gatto"`" comunicheremo automaticamente all'interprete che la variabile `$a` in questo contesto è da considerarsi una stringa.

1.5.2 Array

Un *array* è una struttura dati che ci permette di accedere ad un insieme di variabili identificate da uno stesso nome. Un array ha una lunghezza pari al numero di variabili di cui è costituito. Può essere comodo pensare ad un array come ad una tabella costituita da una o più righe e da più colonne. Se l'array è formato da un'unica riga allora si parlerà di *vettore*, altrimenti parleremo di *matrice*.

Gli elementi dell'array (le variabili che lo costituiscono) sono identificate dallo stesso nome dell'array e da uno o più indici, che indicano la posizione dell'elemento all'intero del vettore o della matrice.

Ad esempio, pensiamo di voler memorizzare i nomi dei giorni della settimana all'interno dell'array di tipo stringa chiamato `giorno`; avremo la seguente struttura:

```
$giorno[0] = "lunedi´"
$giorno[1] = "martedi´"
$giorno[2] = "mercoledi´"
$giorno[3] = "giovedi´"
$giorno[4] = "venerdi´"
$giorno[5] = "sabato"
$giorno[6] = "domenica"
```

In Perl, come vedremo in seguito, gli indici degli elementi degli array cominciano dal numero 0; se l'array ha n elementi, allora l'indice dell'ultimo elemento sarà $n-1$.

1.5.3 Liste e grafi

Una *lista* è una struttura simile a quella di un vettore; a differenza degli array, la lunghezza della lista può variare in ogni momento aggiungendo o rimuovendo elementi; inoltre non è possibile raggiungere direttamente un preciso elemento della lista stessa, perché gli elementi non sono indicizzati come in un vettore: è necessario quindi scorrere la lista dal primo elemento, fino a quando non viene identificato l'elemento desiderato.

Un *grafo* è una generalizzazione di una lista: nelle liste ogni elemento ha un successore (escluso l'ultimo) ed un predecessore (escluso il primo); in un grafo invece la struttura può essere molto più "intricata": un elemento può avere più successori e più predecessori.

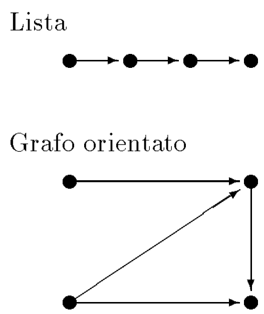

Figura 1.4: Esempio di lista e grafo orientato

1.5.4 File

Quando si memorizza un dato su un supporto magnetico come un hard disk o un nastro, o più in generale su un'unità di memoria di massa, lo si memorizza utilizzando una struttura chiamata *file*. Un file è una sequenza di byte; ai nostri fini un byte equivale ad un carattere. Il sistema non impone al file nessun tipo di struttura e non assegna nessun significato al suo contenuto. I byte assumono significato in funzione del programma che li interpreta . Inoltre, come vedremo, tutto questo è vero non solo per i file su disco, ma anche per le periferiche. I dischi, i messaggi di posta elettronica in partenza ed in arrivo, i caratteri battuti sulla tastiera, l'output sul video del terminale, i dati che passano da un programma all'altro attraverso i *pipe* sono tutti visti dal sistema e dai programmi da esso gestiti come file, ed in quanto tali non sono altro che sequenze di byte.

Per operare su di un file è necessario "aprirlo" associandogli un puntatore (*handler*) per successivi riferimenti. Al termine delle operazioni sul file, questo dovrà essere "chiuso"; questa operazione implica la distruzione del puntatore al file stesso.

Capitolo 2

Introduzione al Perl

In questo capitolo eviteremo di approfondire la trattazione teorica degli argomenti, per cercare di dare una visione sintetica ed orientata ad aspetti più prettamente pratici e sintattici, che ci possano introdurre all'uso del linguaggio.

2.1 Finalità del Perl

Il Perl (Processing Estraction Report Language) è un linguaggio finalizzato principalmente alla trattazione di stringhe e file di testo. Ciò che in Perl è estremamente naturale fare è effettuare ricerche di sequenze di caratteri all'interno di stringhe (*pattern matching*), sostituzioni di sottostringhe (*pattern substitution*), operazioni su file di testo strutturati in campi e record oppure non strutturati.

Per questi motivi il Perl è utilizzato pesantemente nella scrittura di procedure CGI[1] installate su un server web, o per lo sviluppo di procedure di manutenzione delle attività di un server. Per gli stessi motivi il Perl non è invece indicato per sviluppare procedure di puro calcolo scientifico o comunque programmi che richiedano una elevata velocità e precisione nell'effettuare calcoli o elaborazioni numeriche complesse.

2.2 Intestazione di un programma

Abbiamo detto che il Perl è un linguaggio interpretato, quindi ogni *script* in Perl richiede di essere lanciato attraverso l'interprete per poter essere eseguito. Consideriamo il seguente esempio elementare, costituito da un'unica riga di programma:

```
print "Salve a tutti\n";
```

Innanzi tutto dovremo scrivere questo "programma" con un editor su un file di testo, quindi, una volta salvato il file su disco, ad esempio con il nome `salve.pl`, potremo passarlo all'interprete attraverso il comando[2]:

```
$ perl salve.pl
Salve a tutti
$ _
```

[1] CGI: Common Gateway Interface, è l'interfaccia tra le pagine web HTML e le procedure che elaborano i dati passati da tali pagine e forniscono in output dati in formato HTML.

[2] D'ora in avanti supporremo di utilizzare un sistema UNIX, quindi, dove sarà necessario, faremo riferimento ai comandi della shell UNIX ed alle peculiarità di questo sistema; il simbolo "$" sta ad indicare il prompt della shell.

Questo comando comunica al sistema di lanciare l'interprete Perl e di passargli come input il contenuto del file `salve.pl`. L'output del programma viene visualizzato sul terminale dell'utente, quindi lo script termina e viene visualizzato nuovamente il prompt del sistema. La sequenza di controllo "`\n`" inserita alla fine del messaggio da stampare indica all'interprete di inserire un *ritorno a capo* (*newline*) in quella posizione.

Un modo più comodo per richiamare l'interprete Perl ed eseguire un certo script è quello di specificare direttamente nell'intestazione dello script il nome dell'interprete che dovrà eseguirlo. La sintassi da utilizzare è in questo caso quella consueta degli shell-script UNIX. Supponiamo ad esempio che l'interprete Perl sia il file `/usr/local/bin/perl`[3]; il nostro programmino potrà essere riscritto nel modo seguente:

```
#!/usr/local/bin/perl
print "Salve a tutti\n";
```

La prima riga del file inizia con la sequenza "`#!`" (cancelletto e punto esclamativo) seguita dal nome dell'interprete che dovrà eseguire il programma. Affinché la shell UNIX possa eseguire lo script, dovranno essere impostati i diritti di esecuzione su questo file con il comando `chmod`. Ad esempio, supponiamo di voler rendere eseguibile il nostro script per tutti gli utenti del sistema, consentendo anche a chiunque di leggere il contenuto del file, ma riservando il diritto di modificare lo script solo al proprietario del file stesso. Il comando da impartire sarà il seguente:

`$` *chmod 755 salve.pl*

Per eseguire lo script a questo punto sarà sufficiente digitarne il nome:

```
$ salve.pl
Salve a tutti
$ _
```

2.3 Istruzioni di base

Abbiamo già visto l'istruzione `print`, che ci permette di stampare l'output dei nostri script. Questa istruzione è piuttosto potente e permette di formattare l'output in vari modi utilizzando una serie di caratteri di controllo che vedremo in maggiore dettaglio nelle prossime pagine.

Le *variabili scalari* in Perl sono precedute da un simbolo "`$`", in modo da distinguerle da liste ed array. Il Perl non distingue tra variabili numeriche e stringhe: su entrambe è consentito fare le stesse operazioni, lasciando all'interprete il compito di assegnare un valore numerico o stringa al contenuto delle variabili a seconda del contesto in cui queste sono usate.

Il seguente script è una banale evoluzione dello script precedente:

```
#!/usr/local/bin/perl
# semplice script di esempio
$nome = "Marco";
print "Buongiorno $nome\n";
```

[3] Per conoscere l'esatta collocazione dell'interprete Perl nel filesystem della nostra macchina basta dare il comando "`which perl`" oppure "`find / -name perl -print`"; se l'interprete è installato il sistema ne visualizzerà il path.

Il cancelletto è utilizzato in Perl per introdurre delle linee di commento; quindi dopo un simbolo "`#`" potete inserire ogni tipo di commento che possa aiutare voi o qualcun altro a comprendere anche a distanza di tempo il funzionamento dello script. L'interprete Perl eviterà di analizzare e di tradurre ogni parola che segue il simbolo "`#`" fino alla fine della riga.

L'istruzione complementare a quella di stampa è quella di lettura, che consente di assegnare alle variabili dello script dei valori digitati dall'operatore o letti da un file. Il seguente programmino è una ulteriore evoluzione dell'esempio precedente e fa uso dell'istruzione di lettura:

```
#!/usr/local/bin/perl
# un altro esempio elementare
print "Come ti chiami? ";
$nome = <STDIN>;
print "Buongiorno $nome\n";
```

Lanciando in esecuzione quest'ultima versione del programma `salve.pl`, il sistema ci chiede di inserire il nostro nome mediante la tastiera del terminale, e quando avremo battuto RETURN per segnalare la fine del nostro inserimento, visualizzerà il consueto messaggio di buongiorno. Il simbolo `<STDIN>` è infatti l'handler del file di input standard, un file sempre aperto in lettura durante tutta l'esecuzione del programma e che rappresenta generalmente la tastiera del terminale dell'utente. Con l'istruzione `$nome=<STDIN>` si assegna alla variabile `$nome` la linea di input proveniente dal canale di input standard, compreso il carattere di "fine riga".

```
$ salve.pl
Come ti chiami?  Marco
Buongiorno Marco
$ _
```

È possibile redirigere il contenuto di un file attraverso il canale dello standard input utilizzando le funzionalità di redirezione e di *piping* messi a disposizione dalla shell di comandi del sistema operativo. Ad esempio supponiamo di aver creato un file di testo (chiamato `nome`) con il nostro editor, in cui abbiamo inserito soltanto una parola, ad esempio "Marco". Allora potremo redirigere il contenuto di questo file verso il canale di input standard del nostro programma, nel seguente modo:

```
$ salve.pl < nome
Come ti chiami?  Buongiorno Marco
$ _
```

Allo stesso modo in cui redirigiamo l'input standard, possiamo anche redirigere verso un file il canale di output standard, in modo da memorizzare su file ogni messaggio che il nostro programma avrebbe altrimenti visualizzato sullo schermo del terminale. Il seguente comando non produce alcun output:

```
$ salve.pl < nome > output
$ _
```

Infatti ciò che il programma doveva leggere in input gli è stato passato attraverso il file `nome`, mentre l'output è stato scritto nel file `output`.

2.4 Input/output su file

I canali standard di comunicazione di ogni programma sono tre e ad ognuno di questi è associato un handler:

<STDIN>	standard input
<STDOUT>	standard output
<STDERR>	standard error

Tabella 2.1: I file handler standard

Così come ogni input avviene di default attraverso <STDIN>, così anche l'output viene emesso attraverso <STDOUT>; in alcuni casi è opportuno dichiarare esplicitamente il canale di output:

```
print STDOUT "Buongiorno $nome\n";
```

Il canale <STDERR> è simile a <STDIN>, ma viene solitamente utilizzato per visualizzare i messaggi di errore in modo tale che, se anche lo standard output è rediretto attraverso un file, il messaggio di errore viene visualizzato ugualmente sul terminale dell'utente o sulla console del sistema.

Oltre a questi file handler standard che non è necessario aprire o chiudere esplicitamente, è possibile aprire anche un canale di comunicazione verso un normale file di testo sull'unità di memoria di massa.

Prima di poterlo utilizzare, un file deve essere aperto dichiarandone anche l'uso che se ne intende fare: il file infatti può essere aperto in modalità differenti:

read:	open(*handle*, "<*nome*")	si possono effettuare solo operazioni di lettura dal file specificato;
write:	open(*handle*, ">*nome*")	si possono effettuare soltanto operazioni di scrittura sul file; se il file già esiste verrà cancellato e ne verrà creato uno nuovo;
append:	open(*handle*, ">>*nome*")	si possono effettuare solo operazioni di scrittura aggiungendo dati in coda al file; se il file già esiste in questo caso non verrà cancellato, se non esiste verrà creato;

Tabella 2.2: Modalità di apertura di un file

Il seguente script legge tutte le linee di un file di testo e le visualizza sullo schermo del terminale. Viene fatto uso della struttura di controllo `while` che ancora non è stata descritta; per il momento basti sapere che questa struttura consente di ripetere un certo blocco di istruzioni fintanto che una certa condizione non risulta verificata.

```
#!/usr/local/bin/perl
# simpcat.pl: versione semplificata di ´cat´
print STDERR "nome del file: ";
$filename = <STDIN>;
open(FILE, "< $filename");
while (!eof(FILE)) {
```

```
  $riga = <FILE>;
  print "$riga";
}
```

Al solo scopo di cominciare ad introdurre alcuni dei comandi che diventeranno di uso comune nelle prossime pagine, possiamo riscrivere in maniera equivalente lo script precedente come segue:

```
#!/usr/local/bin/perl
# simpcat.pl: versione semplificata di ´cat´
$filename = $ARGV[0];
open(FILE, "< $filename") || die "Errore!\n\n";
while ($riga = <FILE>)) {
  print "$riga";
}
```

Nel primo esempio il programma chiede di inserire il nome del file da visualizzare, mentre nel secondo caso si aspetta che il nome gli venga passato come parametro. Inoltre, nella seconda versione, se il file risulta impossibile da aprire lo script visualizza il messaggio "`Errore!`" e l'esecuzione ha termine.

Già da queste due versioni equivalenti di un esempio così elementare si intravede la grande libertà lasciata dal Perl al programmatore: non c'è mai un solo modo per compiere una certa operazione in Perl.

Capitolo 3

Strutture dati in Perl

3.1 Variabili scalari ed array

In Perl, a differenza di quanto accade con altri linguaggi di programmazione, non è necessario dichiarare le variabili prima di utilizzarle; il simbolo "`$`" che le precede sta ad indicare che si tratta di variabili che possono accettare un unico valore, che potrà essere un numero, un singolo carattere o una stringa. Chiameremo questo tipo di variabili *scalari*, per distinguerle da altri tipi più complessi (variabili *vettoriali*).

L'interprete Perl, utilizzando le variabili nelle istruzioni del nostro programma, stabilirà dal contesto se queste devono essere utilizzate come variabili numeriche o come stringhe.

`$nome = 'Marco';`	stringa
`$n = 3;`	numero intero
`$x = 3.1415;`	numero razionale
`$saluto = "Ciao $nome\n";`	stringa con interpolazione
``$giorno = `date`;``	comando

Tabella 3.1: Tipi di dato

Un *array* è una struttura dati che permette di accedere ad un set di variabili scalari direttamente attraverso un *indice numerico*; potremmo dire, in altri termini, che si tratta di una variabile a più componenti.

I nomi degli array in Perl sono preceduti dal simbolo "`@`" per distinguerli dalle variabili scalari. Le singole componenti del vettore (array) sono delle variabili scalari. Ad esempio l'istruzione

```
@frutta = ('mele', 'pere', 'banane', 'uva');
```

definisce l'array `@frutta` come un vettore a quattro componenti; per accedere alla singola componente si deve usare un indice numerico, come nel seguente esempio:

```
#!/usr/local/bin/perl
# script "frutta.pl"
@frutta = ('mele', 'pere', 'banane', 'uva');
print "un cesto pieno di $frutta[1].\n";
```

Il primo elemento di un vettore in Perl ha indice 0, quindi se un vettore ha n elementi, l'ultimo elemento ha indice $n-1$. Ad esempio lo script precedente fornisce il seguente risultato:

```
$ frutta.pl
un cesto pieno di pere.
$
```

Può essere interessante osservare come sia possibile utilizzare gli array nel trattamento di file di testo. A seconda dei casi, e soprattutto in funzione della dimensione del file da leggere, può essere opportuno leggere il file una riga per volta, assegnando la riga letta ad una variabile scalare, ovvero leggere il file per intero, assegnando ogni riga del file ad un elemento di un array di stringhe.

```
#!/usr/local/bin/perl
# leggi.pl
# legge i dati dallo standard input una riga per volta
# e li stampa
$i = 0;
while ($riga = <STDIN>) {
  $i++;
  print "$i) $riga";
}
```

```
#!/usr/local/bin/perl
# leggi_2.pl
# legge i dati dallo standard input per intero
# e poi li stampa
@riga = <STDIN>;
for ($i=0; $i<=$#riga; $i++) {
  print "$i) $riga[$i]";
}
```

Nel primo esempio (`leggi.pl`) il file viene letto una riga alla volta ed immediatamente stampata; viene quindi fatto un uso limitato della memoria della macchina, ma non è possibile utilizzare in seguito le righe del file lette ai passi precedenti. Viceversa nel secondo esempio (`leggi_2.pl`) il file viene caricato in memoria per intero, assegnando ad ogni componente dell'array `@riga` una linea del file di testo; successivamente le linee vengono stampate (facendo uso della struttura di controllo `for` che introdurremo in seguito). In questo caso il programma può utilizzare in qualsiasi momento ogni linea del file, senza doverla rileggere; tuttavia se il file è di grosse dimensioni, viene fatto un uso pesante della memoria della macchina.

L'espressione "`$#riga`" fornisce l'indice dell'ultima componente dell'array `@riga`.

3.2 Liste ed array associativi

Una *lista* in Perl è semplicemente un insieme ordinato di scalari. Una lista può contenere numeri, stringhe o anche un misto di entrambi questi tipi di dato. In effetti in Perl la differenza tra una lista ed un array è molto sottile, tanto che potremmo dire che se assegnamo un nome ad una lista ciò che otteniamo è un array. Più che altro la distinzione, ben marcata in altri linguaggi di programmazione, in Perl riguarda soprattutto il modo in cui vengono utilizzati certi insiemi di dati.

Una lista viene definita elencandone gli elementi, come, ad esempio:

```
('cani', 'gatti', 'cavalli', 'pesci')
(1, 2, 3, 4, 5, 47, 1.34, 12)
```

Come abbiamo già visto possiamo inizializzare un array con il contenuto di una lista con la seguente istruzione:

```
@animali = ('cani', 'gatti', 'cavalli', 'pesci');
```

Gli elementi di una lista possono essere ordinati alfabeticamente utilizzando l'istruzione `sort`; ad esempio, la seguente istruzione ci permette di ordinare il contenuto dell'array `@animali` appena definito:

```
@animali = sort @animali;
```

Cogliamo l'occasione per elencare tre importanti istruzioni che consentono di effettuare delle operazioni elementari sulle liste. L'istruzione `shift` restituisce il primo elemento della lista e lo elimina dalla lista stessa. L'operatore `pop` restituisce invece l'ultimo elemento della lista e lo elimina dalla lista. Infine l'operatore `push` inserisce un elemento al termine della lista.

Veniamo ora al tipo di dato più importante in Perl, gli *array associativi*; citando Larry Wall (l'autore del Perl) si può dire che se non si pensa in termini di array associativi non si sta pensando in Perl.

Un normale array ci permette di "puntare" ai suoi elementi usando numeri interi come indici. Un array associativo ci permette di fare la stessa cosa utilizzando delle stringhe come indici (o *chiavi*, in questo caso). Visto che spesso gli elementi dell'array sono stringhe, in questo modo possiamo associare tra loro coppie di stringhe: il dato vero e proprio e la chiave (o l'indice) che lo identifica all'interno dell'array.

Un array associativo è una lista di valori come ogni altro array, salvo che per il fatto che invece di indicizzare l'array mediante la posizione di ogni elemento, si indicizza ogni coppia di valori dell'array mediante il primo elemento della coppia.

Ad esempio supponiamo di voler tenere conto del numero di giorni di ogni mese dell'anno (supponiamo anche di non essere in un anno bisestile); il seguente programmino potrebbe aiutarci:

```
#!/usr/local/bin/perl
# giorni.pl: dato il nome di un mese ne stampa il
#            numero di giorni
%giorni = ('gennaio', 31, 'febbraio', 28, 'marzo', 31, 'aprile', 30,
           'maggio', 31, 'giugno', 30, 'luglio', 31, 'agosto', 31,
           'settembre', 30, 'ottobre', 31, 'novembre', 30,
           'dicembre', 31);
print "mese: ";
$mese = <STDIN>;
chop($mese);
print "$mese ha $giorni{$mese} giorni.\n";
```

In maggiore dettaglio possiamo dire che ci si riferisce ad un array associativo con il simbolo "`%`" (e non con la chiocciola "`@`", come per i normali array o le liste); il singolo elemento dell'array associativo (che, come al solito è uno scalare) è individuato dalla chiave associata; così il terzo elemento dell'array associativo definito nell'esempio precedente (31, il numero di giorni del mese di marzo) è individuato dall'espressione `$giorni{'marzo'}`.

La funzione `chop` usata dopo aver letto un input da file, serve ad eliminare il carattere di fine riga dalla variabile che contiene l'input (nel nostro caso la variabile `$mese`).

Osserviamo infine che la funzione `keys()` restituisce una lista con le chiavi dell'array associativo passato come argomento, in modo da consentirci di operare sugli elementi dell'array nell'ordine desiderato. La funzione `values()` effettua l'operazione complementare: restituisce una lista con i valori degli elementi dell'array associativo passato come argomento.

3.3 Apici, stringhe ed interpolazione

Il Perl adotta in pieno la convenzione sull'uso degli apici, già adottata da numerosi altri tool in ambiente UNIX, tanto da rendere questa convenzione uno standard per questo ambiente.

Una coppia di apici singoli "´" (l'apostrofo) viene utilizzata per delimitare una stringa che non dovrà essere oggetto di ulteriore interpretazione da parte del Perl. Così, volendo assegnare alla variabile `$nome` il valore "Pippo" potremo usare l'espressione "`$nome = ´Pippo´`".

I doppi apici "`"`" (le virgolette) sono invece utilizzate per delimitare una stringa il cui valore dovrà essere ulteriormente sviluppato dall'interprete Perl. In gergo tecnico si dice che l'interprete effettuerà l'*interpolazione* del valore della stringa, ossia esploderà il valore di eventuali variabili o stringhe di controllo contenute all'interno della stringa stessa.

Consideriamo il seguente esempio:

```
# !/usr/local/bin/perl
# somma.pl: esempio di interpolazione
$a = 3;
$b = 4;
$c = $a+$b;
print ´con apici: $a+$b=$c\n´;
print "\ncon virgolette: $a+$b=$c\n";
```

Eseguendo il programma `somma.pl` otterremo il seguente output:

```
$ somma.pl
con apici:  $a+$b=$c\n
con virgolette:  3+4=7
$ _
```

Nel primo caso, invece di stampare il contenuto delle variabili e di eseguire la stringa di controllo "`\n`" (ritorno a capo), l'interprete si è limitato a stampare letteralmente ciò che era contenuto tra gli apici. Quando invece sono state utilizzate le virgolette il contenuto della stringa è stato correttamente sviluppato dall'interprete.

Per rappresentare all'interno di una stringa delimitata da virgolette un carattere che in Perl ha un significato speciale, è necessario farlo precedere dal simbolo "\" (backslash) per evitare che l'interprete lo utilizzi attribuendogli il significato che ha nella sintassi del linguaggio. Ad esempio per visualizzare il simbolo "`$`" dovremo usare la sequenza "`\$`", e lo stesso faremo per rappresentare simboli particolari come "`"`", "`%`", "`@`" ed altri ancora; il carattere "\" segue la medesima regola: per rappresentare il backslash sarà sufficiente ripeterne due di seguito ("\\").

L'apice inverso "`" (accento), presente sulle tastiere americane in alto a sinistra (codice ASCII 96), ha un significato completamente diverso rispetto agli apici ed alle virgolette. Serve infatti a racchiudere un comando che verrà eseguito aprendo un processo figlio dell'interprete Perl (tipicamente si tratta quindi di comandi della shell UNIX). L'output prodotto dal comando eseguito mediante l'apice inverso viene restituito all'interprete Perl (processo padre).

Ad esempio, sappiamo che il comando `date` della shell UNIX serve a visualizzare la data e l'ora corrente; volendo assegnare questo valore ad una variabile di un nostro script in Perl possiamo utilizzare la seguente espressione:

```
$data_odierna = `date +%d/%m/%y`;
```

L'output del comando `date` viene catturato, senza che venga visualizzato sullo standard output, e restituito come valore di ritorno della chiamata del processo stesso.

3.4 Operatori

Il Perl è un linguaggio in cui, come abbiamo visto, c'è una certa promisquità tra i tipi di dati: non è necessario stabilire se una determinata variabile conterrà dati di tipo numerico o stringhe; tuttavia è ben diverso eseguire una addizione tra numeri o una concatenazione di stringhe. A livello di operatori il Perl effettua quindi un *cast* delle variabili in base al tipo di operazione richiesta. In un certo senso si può dire che, trovandosi di fronte ad un operatore, l'interprete stabilisce il tipo di dato contenuto nelle variabili che compaiono nell'espressione da valutare, "in base al contesto".

Se verrà utilizzato un operatore aritmetico tra due variabili, il Perl tenterà di stabilire il valore numerico delle variabili stesse; viceversa se l'operatore sarà di tipo stringa allora il Perl tratterà le variabili coinvolte nell'espressione come delle stringhe.

Nelle tabelle 3.2, 3.3 e 3.4 sono riportati i principali operatori.

`$a + $b`	addizione	somma il valore di `$a` e quello di `$b`
`$a - $b`	sottrazione	sottrae a `$a` il valore di `$b`
`$a * $b`	prodotto	moltiplica `$a` e `$b`
`$a / $b`	divisione	divide `$a` per `$b`
`$a % $b`	modulo	restituisce il resto della divisione `$a/$b`
`$a ** $b`	esponente	restituisce `$a` elevato a `$b`
`++$a`, `$a++`	incremento	aumenta di 1 il valore di `$a`
`--$a`, `$a--`	decremento	diminuisce di 1 il valore di `$a`

Tabella 3.2: Operatori aritmetici

`$a . $b`	concatenazione	restituisce una stringa che contiene `$a` seguito da `$b`
`$a x $b`	ripetizione	restituisce una stringa che riporta il valore di `$a` ripetuto `$b` volte
`substr($a,$n,$l)`	sottostringa	restituisce una sottostringa di `$a`, a partire dal carattere `$n` e lunga `$l` caratteri
`index($a,$b)`	indice	restituisce la posizione del primo carattere della sottostringa `$b` all'interno di `$a`

Tabella 3.3: Operatori stringa

`$a = $b`	assegnazione	assegna a `$a` il valore di `$b`
`$a += $b`	aggiungi a	aggiunge al valore di `$a` il valore di `$b` (equivalente a `$a = $a+$b`)
`$a -= $b`	sottrai a	sottrae al valore di `$a` il valore di `$b` (equivalente a `$a = $a-$b`)
`$a .= $b`	concatena a	concatena alla stringa `$a` la stringa `$b` (equivalente a `$a = $a.$b`)

Tabella 3.4: Operatori di assegnazione

Capitolo 4

Strutture di controllo

4.1 Operatori logici

Il Perl tratta ogni istruzione come una funzione che quindi restituisce un valore. Ogni valore può essere interpretato in Perl come vero o falso; in generale 0 e la stringa vuota indicano falso, mentre 1 ed una stringa non vuota (e non 0) indicano vero.

Gli operatori logici vengono generalmente usati all'interno di una struttura di controllo (come l'istruzione `if-else` o l'istruzione `while` che vedremo nelle pagine seguenti) utilizzata per variare il flusso del programma, modificandone la struttura sequenziale. Ci accorgeremo presto che in Perl, sfruttando proprio il fatto che ogni operazione restituisce comunque un valore di vero o falso, si può controllare il flusso del programma anche al di fuori delle tradizionali strutture di controllo.

Vediamo innanzi tutto in quale modo vengono rappresentati in Perl gli operatori logici. La seguente tabella ne riassume la sintassi:

`$a && $b`	AND	l'espressione logica è vera se `$a` e `$b` sono entrambe vere; se `$a` o `$b` è falsa allora l'espressione è falsa.
`$a || $b`	OR	l'espressione logica è vera se almeno una delle due variabili è vera; se `$a` e `$b` sono entrambe false allora l'espressione è falsa.
`!$a`	NOT	l'espressione logica è vera se `$a` è falsa; l'espressione è falsa se `$a` è vera.

Tabella 4.1: Operatori logici

4.2 Operatori di Confronto

Il Perl ci mette a disposizione tre insiemi di operatori di confronto; ognuno di questi insiemi si applica su un tipo di dato differente: numeri, stringhe e nomi di file.

Come per ogni altro operatore il Perl effettuerà un cast (conversione di tipo) delle variabili prima di effettuare il confronto per rendere omogeneo e coerente con l'operatore il tipo dei dati trattati.

Le tabelle 4.2, 4.3 e 4.4 riportano i principali operatori:

`$a == $b`	uguaglianza	vero se `$a` è uguale a `$b`
`$a < $b`	minore di	vero se `$a` è minore di `$b`
`$a <= $b`	minore o uguale	vero se `$a` è minore o uguale a `$b`
`$a > $b`	maggiore	vero se `$a` è maggiore di `$b`
`$a >= $b`	maggiore o uguale	vero se `$a` è maggiore o uguale a `$b`
`$a != $b`	diverso	vero se `$a` è diverso da `$b`

Tabella 4.2: Operatori di confronto tra numeri

`$a eq $b`	uguaglianza	vero se `$a` è uguale a `$b`
`$a lt $b`	minore di	vero se `$a` è minore di `$b` (ordine alfabetico
`$a le $b`	minore o uguale	vero se `$a` è minore o uguale a `$b` (ordine alfabetico)
`$a gt $b`	maggiore	vero se `$a` è maggiore di `$b` (ordine alfabetico
`$a ge $b`	maggiore o uguale	vero se `$a` è maggiore o uguale a `$b` (ordine alfabetico)
`$a ne $b`	diverso	vero se `$a` è diverso da `$b`

Tabella 4.3: Operatori di confronto tra stringhe

4.3 Struttura condizionale if-else

La prima e più elementare struttura di controllo è la `if-else`. Permette di modificare il flusso delle istruzioni del programma a seconda che una certa condizione sia o meno verificata.

La sintassi dell'istruzione è la seguente:

```
if (condizione) {
    primo blocco di istruzioni
} else {
    secondo blocco di istruzioni
}
```

Se la condizione è vera allora viene eseguito il primo blocco di istruzioni racchiuso tra parentesi graffe, altrimenti viene eseguito il secondo blocco di istruzioni. Al termine di ognuno dei due blocchi il flusso del programma riprende con le istruzioni presenti dopo la chiusura dell'ultima parentesi graffa.

Il blocco `else` può essere omesso:

```
if (condizione) {
    blocco di istruzioni
}
```

In questo caso se la condizione è verificata allora viene eseguito il blocco di istruzioni riportato tra parentesi graffe, quindi l'esecuzione del programma continuerà con le istruzioni che seguono la parentesi graffa chiusa; se invece la condizione risulta falsa allora il blocco di istruzioni tra parentesi graffe non viene eseguito ed il programma continua semplicemente saltando il blocco.

`-r $a`	leggibile	vero se il file `$a` è leggibile
`-w $a`	scrivibile	vero se è possibile scrivere sul file `$a`
`-d $a`	directory	vero se `$a` è una directory
`-f $a`	file regolare	vero se `$a` è un file regolare (non un device o altro)
`-T $a`	file di testo	vero se `$a` è un file di testo
`-e $a`	esiste	vero se il file `$a` esiste

Tabella 4.4: Operatori su file

La struttura di controllo `if` ha alcune interessanti variazioni che rendono in certi casi più sintetico il programma. Ad esempio è possibile utilizzare la *notazione post-fissa*:

istruzione `if` *condizione*;

In questo caso l'istruzione (singola) viene eseguita solo se è verificata la condizione (che con la notazione post-fissa può non essere racchiusa tra parentesi tonde).

Un altro modo di utilizzare una struttura di controllo simile alla `if` è, come accennavo precedentemente, quello di sfruttare il fatto che l'esecuzione di ogni istruzione in Perl restituisce un valore che può essere interpretato come vero o falso.

Supponiamo quindi di voler eseguire l'istruzione $istruz_2$ solo se l'istruzione $istruz_1$ va a buon fine (restituisce un valore vero); potremo allora utilizzare l'operatore AND e concatenare le due istruzioni con la seguente espressione:

$istruz_1$ `&&` $istruz_2$;

L'espressione appena vista equivale a scrivere

`if` ($istruz_1$) {
    $istruz_2$;
}

Viceversa se si vuole eseguire l'istruzione $istruz_2$ solo se l'istruzione $istruz_1$ restituisce un valore falso, potremo utilizzare la seguente espressione:

$istruz_1$ `||` $istruz_2$;

che equivale alla struttura canonica:

`if ( !` $istruz_1$) {
    $istruz_2$;
}

Cerchiamo di chiarire meglio questo comportamento assai interessante del Perl, ma piuttosto insolito se confrontato con altri linguaggi di programmazione.

Innanzi tutto negli esempi precedenti abbiamo utilizzato una istruzione al pari di una espressione logica. La differenza tra questi due oggetti è piuttosto marcata: una istruzione serve per compiere una determinata operazione (scrivere qualcosa sul canale di output, eseguire una operazione matematica, eseguire una operazione su una stringa di caratteri, ecc.), mentre una espressione logica è un oggetto che in base al valore logico (vero o falso) delle sue componenti ci fornisce un valore logico (vero o falso). In Perl però le due cose si confondono visto che ogni istruzione restituisce comunque un valore che può essere interpretato in termini logici, ossia in termini di "*vero* o *falso*".

Quindi non è sbagliato utilizzare una istruzione come operatore logico che ci premette di compiere una decisione.

In secondo luogo è importante capire il modo in cui il Perl valuta le espressioni logiche. Come per ogni espressione si parte da sinistra e si procede verso destra nella valutazione delle componenti; le parentesi servono a stabilire priorità diverse da quelle standard. Inoltre, per ottimizzare i tempi di esecuzione, il Perl interrompe la valutazione di una espressione logica non appena è in grado di stabilirne il valore di vero o falso.

Quindi nel caso di una espressione del tipo "A AND B" (in Perl scriveremo "A `&&` B") se dopo aver valutato l'espressione logica A si ottiene un valore falso, allora il Perl non valuterà l'espressione B, visto che questa operazione risulterebbe comunque inutile al fine della valutazione dell'espressione "A AND B" (l'operatore logico and restituisce vero se e solo se entrambi i termini sono veri). L'espressione B verrà quindi valutata solo se l'espressione A risulta vera.

Viceversa nella valutazione dell'espressione logica "A OR B" se il termine A risulta vero allora il Perl non valuterà il termine B, visto che comunque basta che uno dei due termini sia vero per stabilire il valore (vero) dell'intera espressione. Se il termine A risulta falso allora il Perl non essendo ancora in grado di stabilire il valore dell'intera espressione "A OR B" e quindi procederà alla valutazione anche del secondo termine.

4.4 Il ciclo while

In Perl esistono sostanzialmente due strutture di controllo per realizzare cicli iterativi: la struttura `while` e la struttura `for`.

La struttura `while` ci permette di ripetere un certo blocco di istruzioni finché l'espressione logica che controlla il ciclo risulta vera. Quando dovesse risultare falsa il flusso del programma uscirebbe fuori dal ciclo. Se l'espressione è falsa già prima di entrare nel ciclo `while` allora questo non verrà eseguito neanche una volta. La sintassi è la seguente:

```
while (espressione) {
   blocco di istruzioni
}
```

Vediamo tre esempi di come può essere utilizzata la struttura iterativa `while`. Il primo è il più semplice: il ciclo viene ripetuto fino a quando la variabile flag `$n` non assume il valore zero.

```
#!/usr/local/bin/perl
$n=10;
while ($n > 0) {
   $n--;
   print "$n ";
}
```

Il secondo esempio legge e stampa il contenuto di un file:

```
#!/usr/local/bin/perl
open(IN, "< /tmp/dati") || die "Impossibile aprire il file\n\n";
while ($r = <IN>) {
```

```
  print $r;
}
close(IN);
```

Infine il terzo esempio visualizza il contenuto di una lista:

```
#!/usr/local/bin/perl
@frutta = ("mele", "pere", "pesche", "albicocche");
while (@frutta) {
  $frutto = shift @frutta;
  print "$frutto\n";
}
```

4.5 Il ciclo for

La struttura `for` consente di ripetere un numero prefissato di volte un certo blocco di istruzioni, controllando la ripetizione del ciclo mediante un contatore. La sintassi dell'istruzione `for` è la seguente:

`for` (*condizione iniziale*; *condizione finale*; *incremento*) {
    blocco di istruzioni
}

Nell'esempio seguente viene utilizzato un ciclo `for` per stampare tutti gli elementi di un array.

```
#!/usr/local/bin/perl
@frutta = ("mele", "pere", "pesche", "albicocche");
for ($i=0; $i<=$#frutta; $i++) {
  print "$frutta[$i]\n";
}
```

La variabile contatore `$i` assume inizialmente il valore 0, viene incrementata ad ogni ciclo di 1, fino a quando non raggiunge il valore pari al numero di elementi dell'array `@frutta` (ricordiamo che questo numero è espresso da `$#frutta`).

Una struttura di controllo per iterare un blocco di istruzioni, simile alla `for`, è la `foreach`, che consente di inserire in una variabile scalare uno dopo l'altro tutti gli elementi di una lista. Il seguente esempio, del tutto equivalente al precedente, ci sarà di aiuto per comprendere il funzionamento di questa istruzione:

```
#!/usr/local/bin/perl
@frutta = ("mele", "pere", "pesche", "albicocche");
foreach $frutto (@frutta) {
  print "$frutto\n";
}
```

Ad ogni iterazione del ciclo la variabile scalare `$frutto` assume un valore pari a quello dell'elemento "corrente" della lista `@frutta`.

4.6 Altre istruzioni di controllo

Chiudiamo questo capitolo riportando altre due istruzioni che possono risultare utili se utilizzate in modo oculato all'interno di un ciclo.

L'istruzione `next` forza il Perl a saltare alla successiva iterazione del ciclo, senza eseguire le rimanenti istruzioni del blocco.

L'istruzione `last` invece termina l'esecuzione di un ciclo facendo saltare il flusso del programma alla prima istruzione successiva al ciclo stesso.

Vediamo con un esempio un possibile utilizzo di queste due potenti istruzioni. Supponiamo di voler leggere il contenuto di un file e di volerne stampare solo le prime 20 linee saltando però tutte le linee vuote. Le linee stampate dovranno anche essere numerate. Il programma potrebbe essere il seguente:

```
#!/usr/local/bin/perl
$cont = 0;
open (IN, "< /tmp/data") || die "Impossibile aprire il file\n\n";
while ($riga = <IN>) {
  chop($riga);
  $riga || next;
  $cont++;
  print "$cont) $riga\n";
  ($cont == 20) && last;
}
close(IN);
```

Capitolo 5

Pattern matching

Una delle caratteristiche principali del Perl è sicuramente quella di poter operare in modo estremamente flessibile sulle stringhe di caratteri. Il concetto che sta alla base di questa funzionalità è la possibilità di descrivere in modo generico un certo *pattern*, ossia uno schema di costruzione di una stringa, per poter effettuare dei confronti o delle sostituzioni all'interno di altre stringhe.

Ciò che ci permette di fare la maggior parte dei linguaggi di programmazione è di verificare che una certa sottostringa sia contenuta in una stringa più lunga. Il Perl, come vedremo nelle prossime pagine, ci consente di generalizzare questa operazione ricercando all'interno di una stringa non una sottostringa ben precisa, ma una generica sottostringa la cui struttura viene descritta utilizzando una certa sintassi (espressioni regolari). In altri termini possiamo dire che l'operazione di pattern matching ci permette di verificare se una stringa appartiene o meno ad un insieme di stringhe, descritto mediante una espressione regolare.

5.1 Espressioni regolari

In questo contesto parlando di *espressioni regolari* intenderemo una espressione costruita secondo una sintassi ben precisa che ci permette di descrivere uno schema di stringa.

Senza entrare in ulteriori dettagli cerchiamo di capire questo concetto mediante un esempio elementare. Supponiamo di voler "descrivere" tutte le stringhe composte secondo il seguente schema: un numero con un qualsiasi numero di cifre, un carattere di spaziatura o tabulazione ed una parola composta da caratteri qualsiasi. Con una espressione regolare diventa banale descrivere questa "ricetta" di composizione di una stringa:

```
\d+\s.+
```

I primi due caratteri "`\d`" indicano la presenza di un carattere numerico (0, 1, 2, ..., 9); il carattere "`+`" che segue una certa sequenza indica che il carattere rappresentato può essere ripetuto una o più volte. La sequenza "`\s`" indica un qualsiasi carattere di spaziatura o di tabulazione. Infine il punto "`.`" indica un carattere qualsiasi e il simbolo "`+`" finale sta ad indicare che questo carattere può essere ripetuto una o più volte.

Questa espressione regolare descriverà quindi stringhe del tipo "`1234 pippo`", "`1 ab$%&xy`", "`1 2`". Le seguenti stringhe invece non risultano "descritte" dalla precedente espressione regolare: "`a b`", "`pippo`", "`albero casa`", "`1+3=4`".

La seguente tabella descrive sinteticamente i termini che possono comporre una epressione regolare:

<table>
<tr><td>.</td><td>qualsiasi carattere escluso il new line ("\n")</td></tr>
<tr><td>[a-z0-9]</td><td>qualsiasi carattere di questo insieme</td></tr>
<tr><td>[^a-z0-9]</td><td>qualsiasi carattere esclusi quelli di questo insieme</td></tr>
<tr><td>\d</td><td>una cifra qualsiasi; equivalente a "[0-9]"</td></tr>
<tr><td>\D</td><td>un carattere che non sia una cifra; equivalente a "[\^0-9]"</td></tr>
<tr><td>\w</td><td>un carattere alfanumerico; equivalente a "[a-zA-Z0-9]"</td></tr>
<tr><td>\W</td><td>un carattere non alfanumerico; equivalente a "[\^a-zA-Z0-9]"</td></tr>
<tr><td>\s</td><td>un carattere di spaziatura (spazio, tabulazione, new line, ecc.)</td></tr>
<tr><td>\S</td><td>un carattere non di spaziatura</td></tr>
<tr><td>\n</td><td>il carattere new line</td></tr>
<tr><td>\r</td><td>il carattere return (ritorno carrello)</td></tr>
<tr><td>\t</td><td>il carattere di tabulazione</td></tr>
<tr><td>\f</td><td>form feed, carattere di avanzamento di pagina</td></tr>
<tr><td>\b</td><td>backspace, cancellazione di un carattere a sinistra</td></tr>
<tr><td>\0</td><td>null, il carattere nullo</td></tr>
<tr><td>\|</td><td>il carattere "|"</td></tr>
<tr><td>\\</td><td>il carattere "\"</td></tr>
<tr><td>*</td><td>il carattere "*"</td></tr>
<tr><td>x?</td><td>il carattere x ripetuto 0 o 1 volta</td></tr>
<tr><td>x*</td><td>il carattere x ripetuto 0 o più volte</td></tr>
<tr><td>x+</td><td>il carattere x ripetuto una o più volte</td></tr>
<tr><td>pippo</td><td>la stringa "pippo"</td></tr>
<tr><td>aa|bb|cc</td><td>la stringa "aa" oppure la stringa "bb" oppure la stringa "cc"</td></tr>
<tr><td>^</td><td>la stringa inizia con l'espressione regolare seguente</td></tr>
<tr><td>$</td><td>la stringa termina con l'espressione regolare precedente</td></tr>
</table>

Tabella 5.1: Termini per la composizione di una espressione regolare

5.2 Pattern matching

Con il termine *pattern matching* si intende l'operazione di verifica se una certa stringa o una sua sottostringa corrisponde ad un certo pattern, ossia se è costruita secondo un determinato schema.

In Perl l'operatore che consente di effettuare il pattern matching è indicato da "`m/`*espressione regolare*`/`"; in effetti la lettera "`m`" (*match*) che precede la coppia di slash ("/") può essere omessa.

Per verificare se la stringa contenuta nella variabile `$a` (o una sua sottostringa) corrisponde al pattern descritto da una certa espressione regolare si utilizzerà la seguente istruzione, che restituisce naturalmente il valore "vero" se il matching riesce e "falso" altrimenti:

`$a =~ /`*espressione regolare*`/`

Supponiamo quindi di voler leggere dal canale di input standard una stringa immessa dall'utente e di voler verificare se la stringa è strutturata in un certo modo (ad esempio un numero, seguito da uno spazio e quindi da una stringa qualsiasi); il programma potrebbe essere il seguente:

```
#!/usr/local/bin/perl
print "Inserisci una stringa: ";
$a = <STDIN>;
```

```
if ($a =~ /^\d+\s.+/) {
  print "Ok, il formato della stringa e´ corretto\n";
} else {
  print "Il formato non e´ quello desiderato\n";
}
```

Supponiamo ora di voler stampare tutte le righe di un certo file che contengono una stringa inserita dall'utente. Un possibile programma per la soluzione di questo semplice problema è il seguente:

```
#!/usr/local/bin/perl
print "Nome del file: ";
$file = <STDIN>;
chop($file);
-e $file || die "Il file non esiste!\n\n";
-T $file || die "Il file non e´ un file di testo!\n\n";
print "Stringa da cercare: ";
$stringa = <STDIN>;
chop($stringa);
open (IN, "< $file") || die "impossibile aprire $file.\n\n";
while ($r = <IN>) {
  $r =~ /$stringa/ && print $r;
}
close(IN);
```

Nell'effettuare il *matching* di una espressione è possibile tenere memoria di alcune componenti dell'espressione stessa. In un'espressione regolare, ogni termine racchiuso tra parentesi tonde verrà memorizzato e ci si potrà riferire ad esso all'interno della stessa espressione mediante la sequenza "\\n", dove n è il numero progressivo del termine memorizzato.

Ad esempio la seguente espressione regolare "descrive" tutte le stringhe del tipo "x = x":

`/^(.+)=\1$/`

Infatti l'espressione indica che la stringa deve iniziare ("`^`") con una certa stringa non nulla ("`(.+)`") che viene memorizzata in "`\1`", quindi deve essere presente il simbolo di uguaglianza ("`=`"), e deve terminare ("`$`") con una stringa identica alla precedente ("`\1`").

Nelle istruzioni successive ad un pattern matching ci si potrà riferire alle variabili "\\n" mediante le variabili "speciali" `$`n.

5.3 Pattern substitution

L'operazione di *pattern substitution* consente di cercare una certa sottostringa all'interno di una stringa (effettuando il *pattern matching* con una espressione regolare) e di sostituire a questa sottostringa un altro insieme di caratteri. L'operatore che consente di effettuare questa operazione ha la seguente sintassi:

`s/`*pattern di ricerca*`/`*pattern di sostituzione*`/`

La sostituzione del primo pattern con il secondo viene effettuata una sola volta (sulla prima occorrenza del primo pattern) a meno che dopo il terzo slash non si aggiunga il carattere "g" (*globally*) che indica di ripetere la sostituzione su tutte le occorrenze del pattern di ricerca e non solo sulla prima.

Modifichiamo l'esempio precedente per produrre un semplice programmino che cerca una certa stringa all'interno di un file di input e genera un file di output in cui ogni occorrenza della stringa è stata sostituita da una seconda stringa inserita dall'utente.

```
#!/usr/local/bin/perl
print "Nome del file di input: ";
$file = <STDIN>;
chop($file);
-e $file || die "Il file non esiste!\n\n";
-T $file || die "Il file non e´ un file di testo!\n\n";
print "Stringa da cercare: ";
$stringa = <STDIN>;
chop($stringa);
print "Stringa da sostituire: ";
$sost = <STDIN>;
chop($sost);
open (IN, "< $file") || die "Impossibile aprire $file.\n\n";
open (OUT, "> $file.new") || die "Impossibile creare $file.new\n\n";
while ($r = <IN>) {
  $r =~ s/$stringa/$sost/g;
  print OUT $r;
}
close(IN);
close(OUT);
```

Capitolo 6

Variabili speciali

Il Perl ci mette a disposizione un insieme piuttosto ricco di *variabili speciali*, ossia di variabili gestite direttamente dall'interprete che contengono parametri spesso assai utili.

6.1 Argomenti su riga di comando e variabili di ambiente

In molti casi può essere utile passare dei parametri sulla linea di comando al nostro script Perl. I parametri vengono immagazzinati automaticamente nell'array `@ARGV`: come per ogni altro array `$#ARGV` indica l'indice dell'ultimo elemento dell'array.

Supponiamo ad esempio di voler scrivere una versione molto semplificata del comando *grep* di UNIX. Ciò che vogliamo realizzare è uno script che stampi in output le righe dei file di testo della directory corrente che contengono la stringa passata come argomento al programma stesso. Quella che segue è una delle possibili implementazioni di questo programma:

```
#!/usr/local/bin/perl
# grep.pl: versione semplificata di grep
$ARGV[0] || die "Non e´ stato specificata la stringa da cercare\n\n";
@file = `ls -1`;
chop(@file);
foreach $file (@file) {
  -T $file || next;
  open(IN, "< $file") || next;
  while ($r = <IN>) {
    $r =~ /$ARGV[0]/ && print "$file: $r";
  }
  close(IN);
}
```

L'esempio precedente è sicuramente sovradimensionato per illustrare l'uso dell'array `@ARGV`, ma può essere utile per richiamare vari concetti visti in precedenza.

Un altro dato che spesso può risultare assai utile è costituito dalle *variabili di ambiente*, ossia quelle variabili impostate a livello di shell che possono essere utilizzate anche all'interno del programma Perl che le "eredita". Le variabili di ambiente sono memorizzate dall'interprete nell'array associativo `%ENV`; la chiave per indicizzare gli elementi dell'array è il nome della variabile stessa. Ad esempio la variabile `$ENV{term}` contiene il tipo di terminale utilizzato dall'utente; la variabile

`$ENV{path}` contiene il *path* di ricerca delle applicazioni sul sistema dell'utente. Chiaramente impostando un valore nell'array associativo `%ENV` si imposterà una variabile d'ambiente ereditata anche dai processi richiamati dallo script Perl.

6.2 Variabili speciali sul pattern matching

In seguito ad un pattern matching vengono istanziate le seguenti variabili speciali:

`$1`, `$2`,...	Contengono i pattern contenuti nelle parentesi del pattern matching (corrispondono ai pattern a cui ci si riferisce con `\1`, ..., `\9` all'interno del pattern matching).
`$&`	La stringa che ha verificato l'ultimo pattern matching.
`` $` ``	La stringa che precede quella che ha verificato l'ultimo pattern matching.
`$'`	La stringa che segue quella che ha verificato l'ultimo pattern matching.
`$+`	L'ultima parentesi del pattern matching.

Tabella 6.1: Variabili speciali per il pattern matching

6.3 Variabili speciali e file

Le seguenti variabili speciali hanno a che fare con le operazioni di I/O su file:

`$_`	La variabile di default per le operazioni di input e di pattern matching.
`$.`	La linea corrente del file di input.
`$/`	Il separatore di record (di default `$/ = "\n"`); può essere impostata ad una qualsiasi sequenza di separazione dell'input letto da file.

Tabella 6.2: Variabili speciali per l'uso di file

6.4 Variabili speciali e processi

Infine le seguenti variabili speciali possono essere utili nel controllare i processi ed i sottoprocessi attraverso uno script Perl:

`$$`	Il numero di processo dello script in esecuzione.
`$?`	Lo stato di ritorno dell'ultima operazione di esecuzione di un processo esterno
`$0`	Il nome del file contenente lo script in Perl.
`$!`	L'ultimo codice di errore generato dallo script.
`$@`	L'ultimo messaggio di errore prodotto dallo script.
`$<`	Lo user ID reale del processo.
`$>`	Lo user ID effettivo del processo.

Tabella 6.3: Variabili speciali e processi

Capitolo 7

Subroutine

In questo breve capitolo descriveremo tutte quelle tecniche che possono servire a rendere più modulare uno script, in modo da poterne riutilizzare in più contesti alcune sue parti o anche per poter integrare nello script strumenti software già esistenti e che sarebbe dispendioso riscrivere da capo.

7.1 Subroutine

Se una certa parte di codice deve essere richiamata più volte all'interno di uno script o semplicemente per motivi di leggibilità del codice stesso, può essere utile spezzare un lungo script in più *subroutine*, ossia piccoli pezzi di sottoprogramma con una loro propria identità logica e funzionale. In Perl chiameremo quindi *subroutine* ciò che in Pascal viene chiamato *procedura* ed in C *funzione*.

Una subroutine deve essere definita prima di poterla utilizzare; per far questo si usa il seguente costrutto:

```
sub nome_subroutine {
    blocco di istruzioni della subroutine
    return valore_di_ritorno;
}
```

È possibile passare dei parametri alla subroutine, che verranno memorizzati dall'interprete nell'array "`@_`". La subroutine può a sua volta restituire un valore che deve essere specificato come argomento della funzione `return`.

Per richiamare una subroutine definita nello script si deve specificare il nome preceduto dal simbolo "`&`" e seguito dalle parentesi che possono eventualmente contenere i parametri (separati da virgole) passati alla subroutine.

Con la funzione `local` possono essere definite delle *variabili locali* che hanno validità solo all'interno della subroutine.

Nell'esempio seguente viene definita una subroutine "`media`" che restituisce la media aritmetica dei numeri passati come argomento:

```
#!/usr/local/bin/perl
# media.pl
# legge dei valori in input e ne stampa la
# media aritmetica

sub media {
  local($n, $somma, $media);
  foreach $n (@_) {
```

```
    $somma += $n;
  }
  $media = $somma/($#_+1);
  return($media);
}

print "Inserisci i numeri:\n";
@numeri = <STDIN>;
chop(@numeri);
$media = &media(@numeri);
print "media = $media\n";
```

7.2 Ricorsione

In Perl possono anche essere definite delle *funzioni ricosive*, ossia delle funzioni che possono richiamare se stesse per produrre il risultato desiderato.

Ad esempio supponiamo di voler calcolare il fattoriale di un numero intero; il fattoriale di n è quella funzione che restituisce il prodotto dei numeri da 2 ad n. Possiamo definire due subroutine diverse per calcolare il fattoriale di n, una iterativa ed una ricorsiva:

```
#!/usr/local/bin/perl
# fattoriale.pl

sub fattoriale_1 {
  local($i, $n, $f);
  $n = shift(@_);
  $f = $n;
  for ($i=$n -1; $i>1; $i--) {
    $f = $f*$i;
  }
  return($f);
}

sub fattoriale_2 {
  local($n) = shift(@_);
  ($n == 1) && return(1);
  return($n * &fattoriale_2($n-1));
}

$n = <STDIN>;
$f1 = &fattoriale_1($n);
$f2 = &fattoriale_2($n);
print "fattoriale iterativo = $f1\n";
print "fattoriale ricorsivo = $f2\n";
```

7.3 Programmi esterni

Con il Perl esistono vari modi per integrare tra loro più programmi esterni, richiamandoli in modo opportuno da uno script in modo che cooperino per la soluzione del problema posto. Brian W. Kernighan[1] nella prefazione di un suo famoso volume,

[1] B. W. Kernighan, R. Pike, *UNIX*, Zanichelli, Bologna, 1995.

nel sostenere che UNIX è un sistema operativo assai potente ed efficace, asserisce che il punto centrale è costituito dal concetto che la forza di un sistema nasce più dall'interazione tra i programmi che non dai programmi veri e propri. Se presi isolatamente—sostiene Kernighan—molti programmi UNIX compiono dei lavori banali, ma combinati con altri, diventano strumenti utilissimi e generalizzati. Il Perl consente di realizzare questa integrazione in modo assai versatile ed efficiente.

I modi usati più comunemente per interagire con programmi esterni attraverso uno script in Perl sono sostanzialmente tre:

- mediante gli apici inversi;
- mediante la funzione `system` o la funzione `exec`;
- mediante l'apertura di un canale di *pipe*.

Vediamo in estrema sintesi queste tre tecniche.

7.3.1 Chiamata mediante apici inversi

È il modo più semplice e diretto di richiamare un programma esterno. Sostanzialmente in questo modo si lancia un processo figlio che esegue il suo compito e restituisce l'output come valore di ritorno, senza visualizzare nulla sullo `STDOUT`.

Ad esempio per inserire la data corrente nella variabile `$data` si può utilizzare la seguente chiamata:

```
$data= `date +%d/%m/%y`;
```

Chiaramente si possono eseguire operazioni anche molto più complesse di questa, magari concatenando in un *pipe* più chimate. Ad esempio per leggere un file di input (`$file`) ordinandone le righe e selezionando solo quelle che contengono la stringa `$s` si può usare la seguente espressione:

```
@linee = `cat $file | grep $s | sort`
```

7.3.2 Le funzioni `system` ed `exec`

Queste due funzioni aprono un processo figlio e lo eseguono interamente (senza intercettarne l'output come per l'esecuzione di programmi mediante gli apici inversi).

Se l'esecuzione di un programma esterno viene effettuata mediante la funzione `system`, allora lo script aspetterà che il programma esterno sia completamente terminato prima di riprendere l'esecuzione dall'istruzione successiva alla `system`.

Con la chiamata `exec` invece lo script termina l'esecuzione e passa definitivamente il controllo al programma esterno richiamato.

7.3.3 Esecuzione mediante il canale di pipe

Aprendo un programma esterno come se fosse un file in modalità di "*piping*", se ne possono sfruttare le funzionalità da uno script in Perl.

Ad esempio, supponiamo di voler visualizzare il contenuto di un array, impaginando l'output mediante il comando UNIX `more`. Il seguente frammento di script fornisce una possibile implementazione:

```
open (OUT, "| more") || die "Errore\n\n";
foreach $riga (@array) {
  print OUT $riga;
}
close(OUT);
```

Un esempio un po' più sofisticato è il seguente. Supponiamo di voler inviare per posta elettronica ad un certo indirizzo l'output del nostro script; il seguente frammento di codice suggerisce una possibile implementazione (in questo caso sfrutteremo il comando `mail` disponibile su numerosi sistemi UNIX):

```
$indirizzo = 'liverani@mat.uniroma1.it';
open (MAIL, "| /bin/mail $indirizzo") || die "Impossibile inviare
  il messaggio all'indirizzo $indirizzo.\n\n";
print MAIL <<"END";
Subject: Risultato dello script
Lo script ha prodotto il seguente risultato...
END
close(MAIL);
```

7.4 Librerie esterne

Un'altra tecnica per rendere modulare e riutilizzabile il codice delle nostre subroutine è quello di produrre una *libreria* di subroutine di uso comune e salvarle su uno o più file. Successivamente se avremo bisogno di utilizzare una subroutine già definita su un file esterno, potremo includere tale file nel nostro script utilizzando la funzione `require`.

Alla funzione `require` deve essere passato come parametro il nome del file che si vuole includere nello script. Il parametro deve quindi includere il *path assoluto* del file desiderato, a meno che il file non si trovi in una delle directory di default in cui l'interprete Perl va a cercare le librerie di subroutine da includere negli script. Questo *path* di ricerca è contenuto nell'array `@INC`; se si vuole inserire un'altra directory nell'array basterà aggiungerla alla lista con la funzione `push`. Il *path* di ricerca di default dipende dall'installazione dell'interprete Perl di cui si dispone sul proprio sistema (tipicamente su un sistema UNIX esiste una directory `/usr/local/lib/perl`).

7.5 Valutazione di espressioni

Concludendo questo capitolo è opportuno citare anche la funzione `eval`, che consente di valutare espressioni Perl. Questa funzione, che appare subito estremamente potente, può risultare spesso difficilmente utilizzabile, ma in alcuni casi può invece essere determinante per sbrogliare situazioni intricate e di difficile soluzione.

In alcuni contesti può risultare troppo gravoso scrivere a priori delle subroutine per svolgere determinati compiti, oppure una subroutine molto generale può risultare poi di fatto troppo lenta; in questi casi può invece essere utile produrre *runtime* (al momento dell'esecuzione dello script) delle subroutine sulla base del contesto in cui ci si trova e farle quindi eseguire all'interprete mediante la funzione `eval`.

Riportiamo un esempio elementare che può aiutare a chiarire il funzionamento di questa potente istruzione. Riprendiamo quindi lo script per il calcolo della media aritmetica su un insieme di numeri inseriti dall'utente e riscriviamolo sfruttando la funzione `eval`:

```
#!/usr/local/bin/perl
# media_bis.pl
# legge dei valori in input e ne stampa la media aritmetica

sub media {
  local($n, $somma, $media);
```

```
  $media = "(";
  $media .= join(´+´, @_);
  $media .= ")/($#_+1)";
  return(eval($media));
}

print "Inserisci i numeri:\n";
@numeri = <STDIN>;
chop(@numeri);
$media = &media(@numeri);
print "media = \$media/n";
```

Capitolo 8

Funzioni principali

Riportiamo in quest'ultimo capitolo una descrizione sintetica (quasi un elenco) delle funzioni di uso comune. Per una lista completa ed una descrizione esaustiva delle funzioni di libreria si rimanda alla documentazione sul linguaggio Perl presente anche nella breve bibliografia riportata nell'introduzione.

8.1 Funzioni aritmetiche

`abs(`*espr*`)` valore assoluto dell'espressione

`cos(`*espr*`)` coseno trigonometrico dell'espressione

`exp(`*espr*`)` esponenziale (e elevato a *espr*)

`int(`*espr*`)` valore intero

`log(`*espr*`)` logaritmo naturale (in base e) di *espr*

`rand(`*espr*`)` valore casuale (non intero) tra 0 ed *espr*

`sin(`*espr*`)` seno trigonometrico di *espr*

`sqrt(`*espr*`)` radice quadrata di *espr*

8.2 Funzioni di conversione

`chr(`*espr*`)` restituisce il carattere rappresentato dal valore decimale *espr*

`hex(`*espr*`)` valore decimale del numero esadecimale *espr*

`oct(`*espr*`)` valore decimale del numero ottale *espr*

`ord(`*espr*`)` codice ASCII del primo carattere di *espr*

8.3 Funzioni su stringhe

`chop(`*lista*`)` rimuove l'ultimo carattere da ogni elemento della *lista*

`eval(`*espr*`)` valuta l'espressione Perl *espr*

`index(`*str*`, `*substr*`)` posizione di *substr* all'interno della stringa *str*

`length(`*espr*`)` lunghezza della stringa *espr* `lc(`*espr*`)` restituisce *espr* in caratteri minuscoli

`rindex(`*str*, *substr*`)` posizione dell'ultima occorrenza di *substr* nella stringa *str*

`substr(`*espr*, *offset*, *len*`)` estrae una sottostringa di lunghezza *len* dalla stringa *espr* a partire dal carattere di posizione *offset*

`uc(`*espr*`)` restituisce *espr* in caratteri maiuscoli

8.4 Funzioni su array e liste

`delete($`*array*{*chiave*}`)` elimina l'elemento dall'array associativo

`exists($`*array*{*chiave*}`)` verifica se l'elemento dell'array associativo esiste

`grep(`*espr*, *lista*`)` restituisce gli elementi della *lista* per i quali l'espressione *espr* ha valore vero

`join(`*espr*, *lista*`)` concatena gli elementi della *lista* separandoli mediante la stringa *espr*; restituisce una stringa con gli elementi concatenati

`keys(%`*array*`)` restituisce una lista con le chiavi dell'array associativo

`pop(@`*array*`)` restituisce l'ultimo elemento dell'array e lo elimina dall'array stesso

`push(@`*array*, *lista*`)` inserisce gli elementi della *lista* alla fine dell'array

`reverse(`*lista*`)` restituisce la *lista* in ordine inverso

`shift(@`*array*`)` restituisce il primo elemento della lista e lo elimina dall'array

`sort(`*lista*`)` ordina gli elementi della *lista* e restituisce una lista ordinata

`splice(@`*array*, *offset*, *length*, *lista*`)` rimuove gli elementi dell'array a partire da *offset* per *length* elementi e li rimpiazza con gli elementi della *lista*; restituisce gli elementi rimossi

`split(`*pattern*, *espr*`)` restituisce una lista di elementi generati dividendo la stringa *espr* in elementi ogni volta che viene incontrata la sottostringa *pattern*

`unshift(@`*array*, *lista*`)` inserisce gli elementi della *lista* in testa all'array

`values(%`*array*`)` restituisce un array con i valori degli elementi dell'array associativo

8.5 Funzioni su file e directory

`chmod(`*modo*, *lista*`)` cambia i permessi sui file specificati nella *lista*

`chown(`*user*, *group*, *lista*`)` cambia il proprietario ed il gruppo dei file specificati nella *lista*

`mkdir(`*dir*, *modo*`)` crea la directory *dir* con i permessi specificati in *modo*

`truncate(`*file*, *dim*`)` tronca il file alla dimensione *dim*

`rename(`*vecchio*, *nuovo*`)` cambia il nome di un file

`rmdir(`*dir*`)` elimina la directory *dir*

`unlink(`*lista*`)` cancella dal filesystem i file specificati nella *lista*

8.6 Funzioni di Input/Output

`close(`*filehandle*`)` chiude il file

`eof(`*filehandle*`)` restituisce vero se il file è terminato, falso altrimenti

`getc(`*filehandle*`)` restituisce il successivo carattere letto dal file

`open(`*filehandle*`,`*file*`)` apre il file e gli associa il nome logico *filehandle*

`print` *filehandle* *lista* scrive su file

`read(`*filehandle*`, $`*var*`,`*n*`,`*offset*`)` legge *n* byte dal file a partire dalla posizione *offset* e li memorizza in \$*var*

`seek(`*filehandle*`,`*posizione*`)` posiziona il puntatore all'interno del file

`sprintf` *formato*`,`*lista* restituisce *lista* in una stringa formattata in base al formato

`tell(`*filehandle*`)` restituisce la posizione del puntatore all'interno del file

Indice analitico