Programmazione della shell Bash

Marco Liverani

Agosto 2011

Marco Liverani, *Programmazione della shell Bash*, Agosto 2011

Revisione del 6 gennaio 2016.

Indice

Introduzione

La *shell* Bash è probabilmente il più diffuso interprete di comandi in ambiente UNIX: è disponibile su tutte le piattaforme UNIX in commercio e su tutte le versioni *open source* del sistema operativo (GNU/Linux, FreeBSD, NetBSD, OpenBSD, ecc.). Bash è stata sviluppata nel 1987 da Brian Fox, come evoluzione della Bourne shell (`sh`) scritta da Stephen Bourne nel 1978; oggi lo sviluppo e l'implementazione di nuove versioni della shell Bash è curata da Chet Ramey. Di fatto i comandi e le caratteristiche della shell Bash costituiscono un sovra-insieme dei comandi e delle *feature* della Bourne shell, che è tutt'ora presente, spesso come shell di default, sulla maggior parte dei sistemi operativi UNIX. Bash è un prodotto *open source* realizzato nell'ambito del progetto GNU. Il nome "Bash" è l'acronimo di "Bourne Again Shell", proprio per ricordarne la derivazione originaria dalla shell di Stephen Bourne.

Stephen Bourne, Brian Fox e Chet Ramey, rispettivamente autori della Bourne shell, della Bash originaria e della attuale versione di Bash

Il sito web ufficiale del progetto di manutenzione e sviluppo del programma Bash, a cui è possibile fare riferimento per la documentazione e l'acquisizione del programma stesso (in formato sorgente o binario per diversi sistemi operativi), si trova all'indirizzo http://tiswww.case.edu/php/chet/bash/bashtop.html.

Sito web ufficiale di Bash

Nelle pagine di questa breve guida vengono presentati alcuni elementi di base per la programmazione della shell Bash. Le shell di comandi presenti in ambiente UNIX infatti sono progettate in modo tale da poter essere utilizzate in modalità interattiva da parte degli utenti, ovvero come interprete di un linguaggio di programmazione per l'esecuzione di un programma vero e proprio.

In questo caso si parla di *shell script*: in generale si tratta di programmi di piccole dimensioni scritti per automatizzare operazioni ripetitive, utili alla gestione di un server UNIX. In passato si è fatto largo uso degli *shell script* anche per la realizzazione di applicazioni CGI (*common gateway interface*) in ambiente web. Oggi l'utilizzo dei linguaggi di scripting di derivazione UNIX per la realizzazione di applicazioni web è circoscritta prevalentemente all'uso del linguaggio Perl che, fra i linguaggi di scripting, è sicuramente uno dei più evoluti.

Comandi della shell e shell scripting

Nelle pagine seguenti faremo riferimento prevalentemente alla programmazione della Bash su macchine che eseguono un sistema operativo UNIX. Tuttavia è bene precisare che la shell Bash è un programma scritto nel linguaggio di programmazione C, con grande attenzione alla "portabilità" e, pertanto, è stato possibile riprodurne numerose versioni anche per sistemi operativi non UNIX, come Microsoft Windows. In ambiente Windows è disponibile, ad esempio, una versione di Bash nell'ambito del pacchetto denominato Cygwin, che consente di riprodurre un ambiente di esecuzione per numerosissimi programmi nati in ambiente UNIX, attraverso una libreria di compatibilità tra le funzioni di base del sistema operativo UNIX e quelle del sistema operativo Microsoft Windows. Quanto descritto nelle pagine di questa guida può quindi essere sperimentato anche utilizzando una shell Bash in ambiente Windows: in questo caso, per acquisire ed installare quanto necessario per l'esecuzione della shell, si suggerisce di fare riferimento al prodotto Cygwin, disponibile su Internet all'indirizzo http://www.cygwin.com.

Il programma **Cygwin** per l'esecuzione di Bash in ambiente Windows

Nel seguito di questo manuale assumeremo che siano noti al lettore i comandi UNIX eseguibili attraverso la shell: salvo qualche richiamo specifico per i comandi meno elementari, si rimanda per un approfondimento ai numerosi testi citati tra i riferimenti

bibliografici (ad esempio [2] e [5]). Sono invece trattati in dettaglio i comandi interni della shell Bash e le istruzioni per il controllo del flusso dell'algoritmo codificato nello *shell script*.

Nella presentazione di esempi di comandi della shell in modalità interattiva, viene rappresentato in grassetto il testo inserito in input dall'utente, per distinguerlo dall'output prodotto dalla shell. Nei numerosi listati riportati nelle pagine seguenti, le righe sono numerate per poter fare riferimento nel testo alle istruzioni riportate negli script; inoltre sono evidenziate in grassetto le "parole chiave" che costituiscono i comandi del linguaggio Bash e in corsivo i commenti presenti nello script.

Descrivendo un linguaggio "artificiale" come quello della Bash, con un approccio elementare ed introduttivo come quello adottato in questo breve manuale, si cammina su un crinale sdrucciolevole: da un lato si cerca di fare in modo di mantenere breve e conciso il discorso, in modo da non annoiare chi legge, d'altra parte però così si corre il rischio di tralasciare aspetti rilevanti e che possono impedire una comprensione dell'argomento a lettori meno esperti e dunque privi di quegli strumenti che altrimenti consentirebbero di fare a meno di molti dettagli. Spero quindi di aver trovato il giusto equilibrio fra sintesi ed efficacia del testo. Se così non fosse, avvisatemi!

Questa breve guida è per sua natura sicuramente incompleta e non descrive per intero le caratteristiche del potente linguaggio Bash: lo ripeto, è una precisa scelta dovuta al fatto che ho preferito la sintesi alla completezza. Testi di riferimento più ampi e completi sono indicati in bibliografia a pagina 85; per gli utenti delle diverse versioni del sistema operativo UNIX, un primo utile riferimento è anche la pagina di manuale di Bash, accessibile direttamente dalla shell del sistema operativo mediante il comando "`man bash`".

Nel testo saranno sicuramente presenti degli errori e degli aspetti poco chiari; in entrambi i casi sarò grato a quanti vorranno segnalarmeli via e-mail con un messaggio di posta elettronica inviato ad uno dei miei indirizzi: marco@aquilante.net, liverani@mat.uniroma3.it.

M. L.

Capitolo 1

La shell Bash

Le *shell* dei comandi del sistema operativo UNIX sono dei programmi che, al pari di altri, possono essere eseguiti da ciascun utente del sistema o possono essere richiamati da altri programmi. A differenza di altri software applicativi (es.: un editor di testo, un'applicazione di calcolo scientifico, un software per la posta elettronica, ecc.) la shell dei comandi è progettata per essere uno degli strumenti principali per consentire una interazione tra l'utente ed il sistema operativo. Mediante la shell l'utente è in grado di gestire i file e le directory presenti sul *filesystem* della macchina, di elaborarne il contenuto e di eseguire altri programmi presenti sul sistema utilizzando la tastiera del proprio terminale come unità di input e lo schermo alfanumerico del terminale come unità di output.

La shell è l'interfaccia "alfanumerica" (non grafica) tra l'utente e il sistema operativo

L'utilizzo del sistema operativo in modalità "alfanumerica" è tipico dei sistemi operativi destinati ai server, macchine che eseguono programmi in grado di offrire servizi di vario genere ad altri sistemi informatici o agli utenti connessi in rete. Su queste macchine tipicamente gli utenti non operano in modalità interattiva, a meno che il sistema non consenta una connessione attraverso terminali direttamente connessi al server o mediante software di emulazione di terminale eseguiti sulle postazioni client degli utenti (spesso dei normali personal computer). L'alternativa ad una shell di comandi che opera in modalità alfanumerica è un'interfaccia utente grafica (GUI – *Graphical User Interface*), che consente all'utente di interagire con il sistema operativo attraverso delle componenti grafico-funzionali che spesso contribuiscono a semplificare l'uso del sistema (es.: invece di digitare un comando sulla tastiera del terminale, con una GUI è possibile eseguire un comando o un programma selezionando un'icona visualizzata all'interno di una finestra grafica). Salvo alcune eccezioni, le interfacce utente grafiche sono presenti sulle postazioni di lavoro degli utenti finali (*workstation*) e non sui server, dove tutte le risorse del sistema sono impiegate per l'esecuzione dei programmi che offrono servizi, senza lasciare spazio all'implementazione di interfacce grafiche sofisticate che semplificherebbero in parte l'interazione tra l'utente e la macchina, penalizzando però l'efficienza dell'esecuzione dei programmi applicativi.

1.1 Esecuzione della shell

Il processo iterativo principale della shell di comandi

Con un minimo di semplificazione possiamo dire che la shell è un programma che esegue iterativamente sempre la stessa operazione: attende che gli venga fornito in input un comando da eseguire, lo valuta per verificare che il comando sia sintatticamente corretto e lo esegue; quindi torna ad attendere che sia fornito in input il comando successivo. Questo processo iterativo termina quando la shell riceve un segnale che indica che l'input è terminato e che non le saranno inviati altri comandi; a quel punto il program-

ma shell termina, liberando la memoria allocata ed altre risorse della macchina messe a disposizione dal sistema operativo.

Esecuzione della shell in modalità interattiva

La shell può essere lanciata in esecuzione in modo automatico dal sistema operativo quando l'utente esegue il login sul sistema stesso, ovvero può essere eseguita dall'utente mediante un comando impartito su una shell già aperta, oppure mediante l'utilizzo di apposite utility grafiche, nel caso in cui si stia operando su un sistema con interfaccia utente grafica. A esempio su un computer Apple Macintosh con sistema operativo Mac OS X è possibile utilizzare la shell dei comandi eseguendo il programma di utilità "Terminal", presente nella cartella "Utility" della cartella "Application". Su una workstation Linux dotata di un desktop manager grafico come GNOME o KDE, è possibile aprire la shell dei comandi selezionando il programma "Terminal" dal menù "Applications → Accessories". In ambiente Windows, come abbiamo accennato nell'introduzione, se è stato installato il pacchetto Cygwin con i suoi moduli di base (che comprendono anche la shell Bash), è possibile eseguire la shell selezionando l'icona presente sul Desktop o sul menù "Start → Programmi" denominata "Cygwin".

Una volta attivata la shell dei comandi è possibile visualizzare il nome del programma shell che stiamo utilizzando con il seguente comando:

```
$ echo $SHELL
/bin/bash
```

Nel caso in cui la shell di default non sia la Bash è possibile verificare se è presente sul sistema in una delle directory elencate nella variabile d'ambiente `PATH`, utilizzando il comando "`which`" ed eseguirla con il comando "`bash`", come nell'esempio seguente:

```
$ echo $SHELL
/bin/tcsh
$ which bash
/bin/bash
$ bash
bash-2.03$
```

La shell in questo modo opera in modalità interattiva, acquisendo in input ogni singolo comando ed i parametri specificati sulla riga di comando e mandando in esecuzione il comando stesso; l'output viene visualizzato sulla medesima finestra di terminale, come negli esempi precedenti.

Ogni comando impartito alla shell viene terminato premendo il tasto Invio/Enter. È possibile impartire più comandi sulla stessa riga, separandoli l'uno dall'altro con il carattere "`;`" (punto e virgola). È possibile anche spezzare l'inserimento di un comando su due o più righe, terminando ciascuna riga intermedia con il carattere "`\`" (*backslash*). Ad esempio:

Separazione di comandi digitati sulla stessa riga

```
$ pwd ; echo $SHELL ; hostname
/home/marco
/bin/bash
aquilante
$ echo \
> $SHELL
/bin/bash
```

Shell script

Oltre che in modalità interattiva, come abbiamo visto negli esempi precedenti, è possibile eseguire la shell in modo tale che elabori una sequenza di comandi riportati in un file di testo ASCII; il contenuto del file è il programma che d'ora in avanti chiameremo *shell script*.

Ad esempio supponiamo di aver predisposto un file denominato "`script.sh`", memorizzato nella nostra home directory; il contenuto del file può essere il seguente:

```
1 echo -n "Oggi e' il "
2 date +%d/%m/%Y
```

Possiamo eseguire questo script molto elementare specificando il nome del file sulla linea di comando con cui viene invocata la shell:

```
$ bash script.sh
Oggi e' il 10/6/2011
```

La shell può anche ricevere la sequenza dei comandi da eseguire attraverso un *pipe* che rediriga l'output di un altro comando sullo *standard input* di Bash:

```
$ cat script.sh | bash
Oggi e' il 10/6/2011
```

Esecuzione diretta di uno shell script dalla linea di comando

Infine, e forse questo è il metodo più diffuso per eseguire uno shell script, è possibile specificare sulla prima riga del programma, con la notazione "`#!`", il *path* assoluto dell'interprete da utilizzare per l'esecuzione dello script; se al file che contiene lo script vengono assegnati i permessi di esecuzione, allora sarà possibile lanciarlo direttamente, lasciando che sia il sistema operativo ad eseguire la Bash, passandogli in input lo script:

```
$ cat script.sh
#!/bin/bash
echo -n "Oggi e' il "
date +%d/%m/%Y
$ chmod 755 script.sh
$ ls -l script.sh
-rwxr-xr-x  1 marco  users  49 18 Apr 23:58 script.sh
$ ./script.sh
Oggi e' il 10/6/2011
```

Forse è superfluo osservare che, nell'ultimo comando dell'esempio precedente, invocando direttamente l'esecuzione dello script memorizzato nel file "`script.sh`" presente nella directory corrente, si è indicato il path relativo "`./`" prima del nome del file; è stato necessario indicare il *path* della directory in cui si trova lo script da eseguire perché spesso, per ragioni di sicurezza, la directory corrente, rappresentata dal simbolo "`.`", non è presente nella lista delle directory in cui la shell deve cercare i comandi esterni da eseguire (la lista di tali directory, come vedremo meglio in seguito, è memorizzata nella variabile d'ambiente `PATH`).

1.2 Aspetti sintattici del linguaggio Bash

Gli shell script devono essere memorizzati in un file di testo ASCII creati utilizzando un programma "editor" che non introduca caratteri o sequenze aggiuntive per la formattazione del testo. Ad esempio degli editor adatti alla creazione di shell script sono i programmi *vi* o *Emacs* disponibili in ambiente UNIX/Linux, o programmi come *Notepad*, *TextEdit* o *UltraEdit* in ambiente Microsoft Windows.

Intestazione degli script con il path dell'interprete Bash

Come abbiamo visto nell'esempio precedente, è buona norma inserire come prima riga di ogni script Bash, la sequenza "`#!/bin/bash`" in cui viene riportato il *path* assoluto del programma Bash nel filesystem della macchina su cui si intende eseguire lo script. In questo modo è possibile lanciare direttamente lo script sulla linea di comando, senza

dover specificare il nome del file come argomento del comando "`bash`". L'indicazione del programma interprete che deve essere usato dal sistema operativo per tradurre ed eseguire le istruzioni dello script, viene fornita nella prima riga dello script stesso, immediatamente dopo la sequenza di caratteri "`#!`". Nei nostri esempi supporremo che l'eseguibile dell'interprete Bash si trovi nella directory "`/bin`", ma su sistemi differenti potrebbe essere installato in altre directory (ad esempio: "`/usr/bin`", "`/usr/local/bin`", ecc.).

Commenti nel codice sorgente di uno script

In generale il carattere "`#`" consente di introdurre un commento nel sorgente dello script: qualunque carattere presente su una riga dello script dopo il carattere "`#`" viene ignorato dall'interprete dei comandi. Spesso infatti si usa inserire delle frasi di commento nel sorgente dello script per descriverne il funzionamento o per spiegare l'effetto di specifici comandi.

Come nell'inserimento di comandi in modalità interattiva, anche nella codifica di uno script ogni istruzione del programma può essere scritta su una riga a se stante, oppure spezzandola su più righe e terminando ciascuna riga (tranne l'ultima) con il carattere "\". Più istruzioni possono essere riportate sulla stessa riga utilizzando il carattere "`;`" come separatore delle istruzioni.

Le istruzioni del programma possono essere "indentate" per rendere più leggibile il codice sorgente, ma si deve porre attenzione nell'uso degli spazi: l'interprete Bash è più "pignolo" di altri interpreti o compilatori e, in alcuni casi, non ammette l'inserimento di spazi arbitrari tra i termini che compongono le istruzioni; in altri casi invece l'uso dello spazio è indispensabile per consentire la corretta interpretazione di una determinata istruzione.

Non esistono caratteri per la delimitazione di blocchi di istruzioni inserite all'interno di una struttura di controllo (es.: le istruzioni da ripetere all'interno di una struttura di controllo iterativa). Esistono invece opportune parole chiave del linguaggio che consentono di circoscrivere correttamente l'inizio e la fine di un determinato blocco di istruzioni; tali parole chiave variano a seconda dell'istruzione utilizzata per il controllo del flusso del programma.

Nella sintassi del linguaggio Bash alcuni caratteri assumono un significato speciale, ossia, se presenti in una stringa di caratteri o come argomento di un comando, questi caratteri svolgono una funzione ben precisa. In Tabella 1.1 è riportato un elenco con la descrizione di ciascun carattere speciale.

Variabili per memorizzare informazioni numeriche ed alfanumeriche

Ne parleremo più estesamente nelle pagine seguenti, ma per comprendere meglio anche solo i primi esempi, è bene precisare che, come in ogni altro linguaggio di programmazione, anche la Bash possiede il concetto di variabile di memoria. Mediante le variabili possono essere identificate facilmente delle aree della memoria della macchina entro cui memorizzare temporaneamente un'informazione numerica o alfanumerica (un numero o una stringa di caratteri). Per definire una variabile basta assegnarle un valore, come nell'esempio seguente:

```
$ a=Ciao
$ echo a
a
$ echo $a
Ciao
```

Dopo aver assegnato un valore ad una certa variabile, per fare riferimento a tale variabile in altri comandi della shell, bisogna anteporre al nome il simbolo "`$`": ad esempio, per riferirci al valore della variabile `a` si utilizza il simbolo "`$a`". Quindi, è chiaro che nell'assegnazione di un valore ad una variabile mediante l'operatore "`=`", il carattere "`$`" deve essere omesso: scrivere "`$a=3`" è sbagliato, mentre l'espressione corretta è "`a=3`"; analogamente, se si vuole assegnare alla variabile `a` lo stesso valore della variabile `b`,

Carattere	Descrizione
\ (backslash)	Precede un altro carattere per comporre una *sequenza di escape*; come ultimo carattere di una riga indica all'interprete che l'istruzione prosegue alla riga successiva
# (cancelletto)	Precede un commento del codice sorgente: i caratteri che seguono il cancelletto fino alla fine della riga vengono ignorati dall'interprete Bash
$ (dollaro)	Precede il nome di una variabile
; (punto e virgola)	Indica la conclusione di un'istruzione per separarla dalla successiva, se sono riportate sulla stessa riga
' (apice)	Delimita le stringhe di caratteri costanti, senza consentire alla shell di interpretare eventuali variabili contenute nella stringa
" (doppi apici)	Delimita le stringhe di caratteri, consentendo alla shell di interpretare e sostituire nella stringa i valori di eventuali variabili in essa contenute
` (backtick)	Delimita un comando consentendo alla shell di sostituire il comando con l'output da esso prodotto; spesso si usa la forma `$(`*comando*`)` al posto del *backtick* `` `*comando*` ``

Tabella 1.1: Caratteri con significato speciale

allora si dovrà scrivere "`a=$b`" e non "`a=b`". Quest'ultima espressione, infatti, produce l'assegnazione del carattere "`b`" alla variabile `a`.

1.3 Apici, doppi apici e backtick

Comunemente nei linguaggi di programmazione gli apici e i doppi apici (le "virgolette") sono utilizzati per delimitare le stringhe e l'uso dell'uno o dell'altro carattere dipendono dalla sintassi adottata da un particolare linguaggio. Ad esempio in C gli apici sono utilizzati per delimitare singoli caratteri, mentre i doppi apici sono utilizzati per delimitare le stringhe.

Nei linguaggi di scripting l'uso degli apici, delle virgolette e del *backtick* ha un significato differente e la Bash in questo non fa eccezione.

Apici singoli per delimitare le stringhe di caratteri

Gli apici singoli sono utilizzati per delimitare le stringhe di caratteri. L'interprete Bash non entra nel merito del contenuto della stringa e si limita ad utilizzare la sequenza di caratteri delimitata dagli apici. In questo modo possono far parte della stringa anche caratteri che altrimenti assumerebbero un diverso significato. L'unico carattere che non può essere utilizzato all'interno di una stringa delimitata da apici sono gli stessi apici; per definire una stringa che contiene gli apici, è necessario delimitarla con le virgolette.

Doppi apici per delimitare stringhe e consentire l'interpolazione delle variabili e dei comandi

Anche i doppi apici (virgolette) sono utilizzati per delimitare le stringhe; tuttavia, se la stringa è delimitata da questo carattere, l'interprete Bash esegue quella che in gergo è chiamata "interpolazione" della stringa e risolve il valore di eventuali variabili riportate nella stringa stessa. In pratica, se in una stringa delimitata da doppi apici è presente il riferimento ad una variabile (es.: `$a`) allora nella stringa al nome della variabile viene sostituito il suo valore.

Per stampare i caratteri (come i doppi apici o il dollaro) che altrimenti verrebbero interpretati ed assumerebbero un altro significato, bisogna anteporre a ciascuno di essi il carattere *backslash* "\". Per stampare il carattere *backslash* in una stringa delimitata da doppi apici bisogna riportare di seguito due *backslash*.

```
$ nome='Marco'
$ echo 'Ciao $nome.'
Ciao $nome.
$ echo "Ciao $nome."
Ciao Marco.
$ echo "Ciao \"\$nome\"."
Ciao "$nome".
```

Backtick per delimitare stringhe che devono essere interpretate ed eseguite come comandi

Il carattere *backtick* ha il comportamento più particolare, tipico dei linguaggi di scripting e assente invece nei principali linguaggi di programmazione di alto livello. Il *backtick* consente di delimitare una stringa che viene interpretata dalla Bash come un comando da eseguire, restituendo come valore l'output del comando stesso prodotto sul canale *standard output*.

Nell'esempio seguente viene utilizzato il comando `date` che restituisce in output la data corrente. L'output di questo comando è una stringa, che usando il *backtick*, può essere assegnata ad una variabile per poterla poi riutilizzare in seguito.

```
1 #!/bin/bash
2 data=‘date +%d/%m/%Y‘
3 echo "Oggi e' il $data."
4 saluto=‘echo "ciao"‘
5 echo $saluto
```

Eseguendo lo script (memorizzato nel file "`data.sh`") si ottiene il seguente output:

```
$ ./data.sh
Oggi e' il 20/06/2011.
ciao
```

Il *backtick* viene interpretato anche se è presente all'interno di una stringa delimitata da doppi apici. Ad esempio il seguente comando produce un risultato analogo a quello dello script precedente:

```
$ echo "Oggi e' il ‘date +%d/%m/%Y‘."
Oggi e' il 20/06/2011.
```

$(...): sintassi alternativa al backtick

Il *backtick* può essere sostituito dalla notazione sintattica "`$(`*comando*`)`" che ha lo stesso comportamento e produce gli stessi risultati. L'esempio precedente può essere modificato come segue:

```
$ echo "Oggi e' il $(date +%d/%m/%Y)."
Oggi e' il 20/06/2011.
```

Capitolo 2

Comandi interni, esterni e composti

Le istruzioni di uno script Bash sono costituite da assegnazioni di valori a variabili (es.: "`a=3`", "`nome='Marco Liverani'`", ecc.) o da comandi costituiti da specifiche *parole chiave* riservate, seguite eventualmente da parametri costanti o da variabili; in questo caso le variabili sono precedute dal simbolo "`$`", come ad esempio in "`echo $nome`", per distinguere il nome della variabile da una stringa costante, come in "`echo nome`".

I comandi che possono essere utilizzati in uno script Bash sono distinti in due insiemi: i comandi interni, resi disponibili dall'interprete, ed i comandi esterni, resi disponibili dal sistema operativo in cui viene eseguito l'interprete e lo script (ad esempio il sistema operativo UNIX). Ci soffermeremo nel seguito prevalentemente sui primi, i comandi interni, ma è bene sottolineare che gli script Bash esprimono la loro massima flessibilità e potenza anche perché consentono una integrazione estremamente semplice con i comandi del sistema operativo e con altri script realizzati nello stesso linguaggio o con altri linguaggi di scripting.

Ad esempio il comando "`date`" è un programma presente nel set di base delle utility di tutti i sistemi operativi UNIX; al contrario, il comando "`echo`" è un comando interno della Bash. Sia il programma "`date`" (comando esterno) che l'istruzione "`echo`" (comando interno) possono essere utilizzati nello stesso modo come istruzioni di uno shell script, senza che sia necessario invocare il programma esterno con forme sintattiche particolari.

```
1 #!/bin/bash
2 echo -n "Oggi e' il "
3 date +%d/%m/%Y
```

Il comando interno "`type`" accetta come argomento una o più stringhe di caratteri e fornisce in output l'indicazione del tipo di comando o parola chiave a cui corrisponde ciascuna stringa. Riportiamo di seguito un esempio per l'uso del comando "`type`":

type: indica che tipo di comando è quello riportato come argomento

```
$ type echo ls cat [ { xyz
echo is a shell builtin
ls is aliased to `ls -F'
cat is /bin/cat
[ is a shell builtin
{ is a shell keyword
-bash: type: xyz: not found
```

Un'altra importante distinzione che possiamo compiere analizzando alcuni aspetti generali relativi alla programmazione della shell è la seguente: in termini piuttosto sem-

A	B	A and B	A or B
vero	vero	vero	vero
vero	falso	falso	vero
falso	vero	falso	vero
falso	falso	falso	falso

Tabella 2.1: Tavole di verità dei connettori logici *and* e *or*

Comandi semplici, liste di comandi e comandi composti

plificati possiamo dire che i comandi della shell si distinguono in comandi *semplici* e comandi *composti*. Un comando semplice è costituito da una singola istruzione riportata su una riga dello script. Un comando composto è costituito da una *lista* di comandi semplici, eseguiti in modo "unitario" dalla shell. I comandi che formano la lista sono separati l'uno dall'altro da uno dei seguenti operatori: "`;`", "`&`", "`&&`", "`||`".

Il carattere "`;`" consente di separare un comando dal successivo nell'elenco che compone la lista, senza introdurre un comportamento specifico nell'esecuzione dei comandi da parte della shell.

```
1 a=3; b=4; echo "I valori sono $a e $b"
```

Il carattere "`&`" consente invece di eseguire in *background* il comando che lo precede.[1] In pratica viene lanciato un altro processo, come processo "figlio" di quello che sta eseguendo lo script, eseguito indipendentemente dal processo "padre".

Return code restituito dai comandi

L'esecuzione di ciascun comando restituisce all'interprete Bash un codice di ritorno (*return code*) numerico; tale codice viene interpretato come un valore logico di *vero* (return code uguale a zero) e *falso* (return code diverso da zero). Uno script shell può restituire un codice di ritorno mediante il comando interno "`exit`", che interrompe l'esecuzione dello script e accetta come argomento il codice di ritorno da restituire.

Gli operatori "`&&`" e "`||`" rappresentano i connettori logici *and* e *or*, rispettivamente, e possono essere utilizzati per sfruttare il codice di ritorno restituito dai comandi. Tenendo conto del significato dei connettori logici *and* e *or*, richiamato in Tabella 2.1, il comportamento dell'interprete, che esegue i comandi di una lista interpretandoli da sinistra verso destra, a fronte di un'istruzione di questo genere:

$$comando_1 \;\&\&\; comando_2$$

è il seguente: il primo comando viene eseguito e, se restituisce il valore *vero*, allora viene eseguito anche il secondo comando (il resto della lista dei comandi); in caso contrario l'esecuzione della lista dei comandi viene interrotta. In altri termini $comando_2$ viene eseguito se e solo se $comando_1$ restituisce zero come return code. Infatti l'interprete esegue la lista per desumere il codice di ritorno complessivo dell'intero comando composto: se il primo comando della lista restituisce il valore *falso* ed è connesso al resto della lista mediante l'operatore logico *and*, è inutile proseguire la valutazione della lista, dal momento che il valore dell'intero comando composto non potrà che essere *falso*.

L'operatore "`||`" si comporta nel modo opposto; a fronte di un'istruzione composta di questo tipo:

$$comando_1 \;||\; comando_2$$

se $comando_1$ restituisce il valore *vero*, l'elaborazione del comando composto viene interrotta perché è già possibile stabilire il valore restituito dall'intero comando, dal momento che il connettore logico che unisce $comando_1$ al resto della lista è l'operatore *or*

[1] Per un maggiore approfondimento tra la modalità di esecuzione in *background* e in *foreground* di un programma e la descrizione dei comandi che consentono di controllare lo stato di esecuzione di un processo, si vedano ad esempio [2] e [5].

(vedi Tabella 2.1). In caso contrario, se *comando*$_1$ restituisce il valore *falso*, allora l'interprete esegue anche i comandi successivi e ne valuta il return code per poter stabilire il valore dell'intero comando composto. Ossia, *comando*$_2$ viene eseguito se e solo se il return code di *comando*$_1$ è diverso da zero.

Sfruttando le considerazioni appena riportate possiamo costruire un comando composto che implementa una sorta di struttura di controllo condizionale, del tipo "*se ... allora ... altrimenti ...*":

$$(\textit{comando}_1 \;\&\&\; \textit{comando}_2) \;||\; \textit{comando}_3$$

Se *comando*$_1$ restituisce il valore *vero* allora viene eseguito anche *comando*$_2$, altrimenti, se *comando*$_1$ restituisce il valore *falso*, viene eseguito *comando*$_3$.

La lista di istruzioni che forma un comando composto può essere anche delimitata da parentesi tonde, graffe o quadre: ogni tipo di delimitatore della lista di istruzioni ha un significato diverso e ben preciso, descritto nella sezione seguente.

In termini più analitici si può dire che la shell Bash è in grado di elaborare i seguenti tipi di istruzioni fornite in modalità interattiva dall'utente sulla linea di comando o inserite nelle righe di uno *script*:

Tipi di istruzioni che possono essere elaborate dalla Bash

1. **comandi semplici**: sono singoli comandi interni o esterni;
2. **liste di comandi**: sono sequenze di comandi, concatenate con i connettori **;**, &, &&, ||, descritti nelle pagine precedenti;
3. **comandi in *pipeline***: sono sequenze di comandi (interni o esterni) concatenati fra loro dall'operatore di *pipe* rappresentato dal simbolo "|", che consente di trasformare l'output di un comando nell'input per il comando successivo nella sequenza rappresentata dalla *pipeline*;
4. **comandi composti**: sono comandi più complessi, composti da più istruzioni e da strutture di controllo che consentono di inserire i comandi della shell in una struttura logico-algoritmica non solo sequenziale, in grado di produrre risultati più sofisticati;
5. **funzioni**: sono dei sotto-programmi, identificati da un nome; dopo che una funzione è stata definita, può essere richiamata nell'ambito della shell o dello script in cui è valida tale definizione, come se si trattasse di un qualsiasi comando della shell; il "corpo" della funzione è definito come un comando composto.

Nelle pagine seguenti di questo capitolo introduciamo i costrutti più elementari per la definizione di comandi composti, mediante la redirezione dell'input e dell'output, la costruzione di *pipeline* e la valutazione di condizioni. Nei capitoli successivi saranno introdotte le istruzioni con cui possono essere implementate le strutture di controllo algoritmiche condizionali e iterative e la definizione di funzioni.

2.1 Comandi composti eseguiti nella stessa shell

Se la lista di istruzioni con cui viene formato un comando composto è delimitata da parentesi graffe, le istruzioni che compongono la lista vengono eseguite in sequenza, dalla prima all'ultima, all'interno della stessa shell in cui viene eseguito lo script. Le istruzioni sono separate fra loro da un punto e virgola. Anche l'ultima istruzione della lista deve essere seguita da un punto e virgola e tra le parentesi graffe e la prima e l'ultima istruzione deve essere presente uno spazio: le parentesi graffe sono infatti delle parole riservate del linguaggio Bash e non dei simboli di carattere sintattico, dunque devono essere separate da uno spazio dalle parole chiave "circostanti". Consideriamo ad esempio il seguente script, piuttosto elementare.

{...} comandi composti delimitati da parentesi graffe

```
1 #!/bin/bash
2 a=1; b=2
3 echo "Prima del comando composto: A = $a, B = $b"
4 { b=3; echo "Durante il comando composto: A = $a, B = $b"; }
5 echo "Dopo il comando composto: A = $a, B = $b"
```

Con le istruzioni a riga 2 vengono definiti i valori delle variabili a e b, che vengono poi visualizzati sul terminale dell'utente con l'istruzione alla riga 3. La sequenza di istruzioni che costituiscono il comando composto a riga 4 viene eseguita come un singolo comando, nell'ambito della stessa shell in cui viene eseguito lo script; infatti la variabile a definita fuori dal comando composto è ancora visibile nel comando stesso ed inoltre l'istruzione a riga 5 permette di verificare che il valore della variabile b è stato modificato.

L'effetto dell'esecuzione dello script precedente è il seguente; lo *scope* delle due variabili a e b è esteso a tutte le istruzioni dello script:

```
$ ./script.sh
Prima del comando composto: A = 1, B = 2
Durante il comando composto: A = 1, B = 3
Dopo il comando composto: A = 1, B = 3
```

Potrebbe sembrare non molto utile raggruppare più istruzioni in un unico comando composto attraverso l'uso delle parentesi graffe, tuttavia, come vedremo più avanti, l'uso delle parentesi graffe diventa assai utile in diverse circostanze, come ad esempio quando si intende applicare all'intero comando composto un operatore di redirezione dell'input/output.

2.2 Comandi composti eseguiti in una sotto-shell

Se le istruzioni che costituiscono il comando composto sono delimitate da parentesi tonde, allora il comando composto viene eseguito in una sotto-shell della shell in cui gira lo script; questo significa che il comando composto eredita tutte le variabili definite nello script, ma lo *scope* delle assegnazioni e delle definizioni effettuate nell'ambito del comando composto è limitato al comando stesso. Il comando composto viene eseguito in una sotto-shell, ma nello stesso processo della shell con cui viene eseguito lo script.

(...): i comandi composti delimitati dalle parentesi tonde sono eseguiti in una "sotto shell" della shell corrente

Lo script seguente, analogo all'esempio precedente, dovrebbe aiutare a chiarire il funzionamento di questo tipo di comandi composti, evidenziando la differenza con i comandi composti delimitati da parentesi graffe:

```
1 #!/bin/bash
2 a=1; b=2
3 echo "Prima del comando composto: A = $a, B = $b"
4 (b=3; echo "Durante il comando composto: A = $a, B = $b")
5 echo "Dopo il comando composto: A = $a, B = $b"
```

Le modifiche dei valori delle variabili effettuate nella sotto-shell non sono disponibili alla shell "madre"

L'output prodotto eseguendo lo script è riportato di seguito; l'aspetto significativo di questo esempio è costituito dal fatto che il comando composto (riga 4) può utilizzare le variabili definite in precedenza nello script (variabile a) e può modificare il valore delle variabili (variabile b), tuttavia le modifiche effettuate dal comando composto non sono visibili al termine del comando stesso. Nell'esempio che segue, infatti, il valore di b viene modificato dal comando composto, ma al termine del comando la variabile assume nuovamente il valore impostato in precedenza (riga 2 dello script).

Operatore	Descrizione
++, --	Incremento e decremento di un'unità
+, -, *, /, %	Somma, sottrazione, moltiplicazione, divisione intera, resto della divisione intera
**	Elevamento a potenza
<<, >>	Shift a sinistra e a destra di un bit

Tabella 2.2: Principali operatori aritmetici interpretati dalla shell Bash

```
Prima del comando composto: A = 1, B = 2
Durante il comando composto: A = 1, B = 3
Dopo il comando composto: A = 1, B = 2
```

A differenza delle parentesi graffe, le parentesi tonde che delimitano il comando composto non sono delle parole riservate del linguaggio, per cui non è necessario separarle dalle altre istruzioni con degli spazi; per lo stesso motivo, l'ultima istruzione della lista può anche non essere seguita da un punto e virgola.

2.3 Valutazione di espressioni aritmetiche

Nell'ambito degli script Bash è possibile effettuare operazioni aritmetiche con una sintassi simile a quella del linguaggio C. Un aspetto molto significativo di cui è necessario tenere conto è che le operazioni aritmetiche valutate dall'interprete Bash operano solo nell'ambito dei numeri interi; dunque determinate espressioni, che ci si potrebbe aspettare diano luogo ad un risultato non intero, producono un risultato per cui la parte decimale viene troncata. Ad esempio dall'espressione "`3/2`" si ottiene il risultato 1 e non 1.5 come ci si sarebbe potuti aspettare.

Le espressioni aritmetiche vengono valutate dall'interprete Bash solo se sono rappresentate come comandi composti delimitati da una coppia di doppie parentesi tonde: "`((`*espressione*`))`". Ad esempio per assegnare ad una variabile il risultato di un'espressione aritmetica, che può anche coinvolgere altre variabili, possiamo usare le seguenti istruzioni: ((...))

```
1 ((a=($b+37)/$c))
2 a=$(( ($b+37)/$c ))
```

Nelle espressioni aritmetiche delimitate dalle doppie parentesi tonde è possibile omettere il simbolo "`$`" davanti alle variabili a cui si vuole fare riferimento: in ogni caso un termine letterale sarà considerato come il nome di una variabile. Per cui l'espressione precedente può anche essere riscritta come segue:

```
1 a=$(( (b+37)/c ))
```

Il seguente script fornisce un esempio abbastanza chiaro del modo in cui devono essere codificate le istruzioni che contengano espressioni aritmetiche al loro interno:

```
1 #!/bin/bash
2 a=5; b=2
3 c=$a/$b
4 echo "a=$a, b=$b, c=$c"
5 ((c=a/b))
6 echo "a=$a, b=$b, c=$c"
```

Operatore	Descrizione
>, >=	Maggiore, Maggiore o uguale
<, <=	Minore, Minore o uguale
==, !=	Uguale, Diverso

Tabella 2.3: Operatori di confronto tra espressioni aritmetiche utilizzabili nell'ambiente "((...))"

Lo script produce l'output riportato di seguito. In pratica l'espressione aritmetica riportata a riga 3 viene ignorata in quanto tale e considerata semplicemente come una concatenazione di stringhe: alla variabile `c` viene assegnato il valore di `a`, seguito da un carattere "`/`" e dal valore di `b`. Per assegnare alla variabile `c` il risultato dell'espressione aritmetica "`a/b`" bisogna utilizzare l'istruzione riportata a riga 5, dove viene fatto correttamente uso delle doppie parentesi tonde per delimitare il comando contenente una o più espressioni da valutare.

```
$ ./script.sh
a=5, b=2, c=5/2
a=5, b=2, c=2
```

Return code di ((...))

È importante osservare che l'istruzione "`((...))`" restituisce un codice uguale a zero se il valore dell'espressione aritmetica all'interno delle doppie parentesi è diverso da zero, mentre avrà un return code diverso a zero se il valore dell'espressione è uguale a zero.

Come di consueto, le espressioni aritmetiche possono utilizzare le parentesi tonde per modificare l'ordine con cui vengono valutati i singoli termini. Gli operatori aritmetici sono gli stessi che ritroviamo anche in linguaggio C e in molti altri linguaggi di programmazione. Per semplicità in Tabella 2.2 sono riportati i principali.

Operatori di assegnazione in forma compatta: **++**, **−**, **+=**, **-=**, ***=**, **/=**, **%=**

Come in linguaggio C sono disponibili degli operatori di assegnazione in forma compatta: "`++`", "`--`", "`+=`", "`-=`", "`*=`", "`/=`", "`%=`". Questi operatori consentono di assegnare alla variabile posta sulla sinistra dell'operatore il risultato di un'operazione aritmetica elementare tra il valore della variabile stessa ed il valore dell'espressione posta alla destra dell'operatore.

Ad esempio l'espressione "`((a+=1))`" è equivalente alle espressioni "`((a=a+1))`", "`((a++))`" e "`((++a))`". Allo stesso modo l'espressione "`((a/=2))`" è equivalente all'espressione "`((a=a/2))`", che però è meno sintetica ed efficiente.

Osserviamo che l'espressione "`(($a++))`" è sbagliata e produce un errore: l'interprete Bash, infatti, nel valutare l'istruzione per prima cosa sostituisce il riferimento alla variabile `a` con il suo valore e quindi prova ad eseguire l'operatore "++" sul valore numerico; quest'ultima operazione (es.: "`5++`") è impropria, perché l'operatore "++" si applica ad una variabile e non ad un numero, visto che oltre ad incrementare di un'unità il valore, l'operatore assegna il nuovo valore alla variabile riportata a sinistra dell'operatore stesso.

Nell'ambiente delimitato dalle doppie parentesi tonte è possibile anche effettuare confronti fra espressioni aritmetiche utilizzando operatori molto simili a quelli utilizzati anche in altri linguaggi di programmazione, come ad esempio il C; per comodità sono riportati in Tabella 2.3.

Nelle espressioni possono essere utilizzate le congiunzioni logiche "`&&`" (*and*) e "`||`" (*or*) per costruire condizioni di confronto più complesse. Il seguente comando composto dovrebbe aiutare a chiarire alcuni dei concetti fin qui espressi:

```
1 (( $a!=0 )) && ( (( $b/$a>=1 )) && echo "Ok" || echo "Ko") || \
2 echo "Divisione per zero"
```

Operatore	Descrizione
`-a`, `-e`	Vero se il file esiste
`-s`	Vero se il file esiste ed ha dimensione maggiore di zero byte
`-d`, `-f`, `-h`	Vero se il file esiste ed è una directory, o un file regolare, o un link
`-r`, `-w`, `-x`	Vero se lo script ha i privilegi di lettura, o di scrittura, o di esecuzione sul file
`-nt`, `-ot`	Vero se il primo file è più recente del secondo (*newer than*) o se il secondo è più recente del primo (*older than*)

Tabella 2.4: Operatori di confronto per valutare lo "stato" dei file presenti sul filesystem

Il comando è composto da una sola espressione molto lunga, spezzata su due righe per ragioni di spazio: il *backslash* alla fine della prima riga indica che l'istruzione non è completa e prosegue alla riga successiva. Di fatto questo comando composto implementa delle strutture di controllo condizionali "nidificate": se il valore della variabile `a` è diverso da zero, allora esegue la divisione `b/a` e confronta il risultato con il valore 1; se il risultato è maggiore o uguale ad 1 allora visualizza il messaggio "`Ok`", altrimenti visualizza "`Ko`"; se invece il valore di `a` è zero, allora non esegue la divisione e visualizza il messaggio "`Divisione per zero`".

2.4 Valutazione di espressioni condizionali

È spesso utile effettuare dei confronti fra stringhe o sullo stato di un file presente sul filesystem della macchina. Per far questo la Bash mette a disposizione un insieme di operatori di confronto che per essere utilizzati devono essere riportati in un comando composto delimitato dalle doppie parentesi quadre: "`[[...]]`".

Operatori per valutare lo stato dei file

Il primo tipo di operatori di confronto per la costruzione di espressioni condizionali consente di verificare lo stato dei file. In Tabella 2.4 sono riportati i principali operatori di questo genere. Il seguente esempio aiuterà a comprendere meglio la sintassi degli operatori:

```
1 [[ -e /etc/shadow ]] && echo 'Shadow password attivo' || \
2 echo 'Shadow password non attivo'
```

Questo comando composto visualizza il messaggio "`Shadow password attivo`" se esiste il file `/etc/shadow`, altrimenti visualizza il messaggio "`Shadow password non attivo`".

Il seguente comando composto visualizza il contenuto del file il cui nome è memorizzato nella variabile `$file` se l'utente ha il permesso di leggerlo, altrimenti visualizza un messaggio di errore:

```
1 [[ -r $file ]] && cat $file || echo 'Privilegi insufficienti'
```

Infine, il seguente comando composto cancella il file `$file3` se è più vecchio di `$file1` e di `$file2`:

```
1 [[ $file1 -nt $file3 && $file2 -nt $file3 ]] && rm $file3
```

Operatori di confronto fra stringhe di caratteri

Nell'ambito di un comando composto per la valutazione di espressioni condizionali possono essere confrontate fra loro anche stringhe ed espressioni il cui valore sia un numero intero. Per il confronto tra stringhe la Bash mette a disposizione gli operatori riportati in Tabella 2.5. È importante inserire uno spazio tra gli operandi e l'operatore di confronto.

Operatore	Descrizione
==, !=	Vero se le due stringhe sono uguali (identiche anche nel *case* dei caratteri) o diverse (anche solo nel *case* dei caratteri: `"Abc" != "abc"`)
<, >	Vero se la prima stringa precede (o segue) in ordine lessicografico la seconda
-n	Vero se la stringa ha una lunghezza maggiore di zero

Tabella 2.5: Operatori di confronto tra stringhe di caratteri alfanumerici

Operatore	Descrizione
-eq, -ne	Vero se le due stringhe hanno lo stesso valore intero o due valori interi differenti
-lt, -le	Vero se la prima stringa ha un valore minore o minore o uguale al valore della seconda
-gt, -ge	Vero se la prima stringa ha un valore maggiore o maggiore o uguale al valore della seconda

Tabella 2.6: Operatori di confronto tra numeri interi nell'ambito della valutazione di espressioni condizionali delimitate dalla doppia parentesi quadra

Ordine lessicografico nel confronto tra stringhe

Nella valutazione dell'ordine reciproco di due stringhe vale il cosiddetto "ordine lessicografico", realizzato confrontando l'ordine reciproco della prima coppia di caratteri differenti con la medesima posizione nell'ambito della stringa; l'ordine reciproco dei caratteri alfanumerici è quello stabilito dalla tabella dei codici ASCII.[2] Ad esempio la stringa "`abc100`" precede, in ordine lessicografico, la stringa "`abc20`"; analogamente la stringa "`100`" precede la stringa "`11`". Se le due stringhe hanno lunghezze differenti e la stringa s_1 è un "prefisso" della stringa s_2, allora $s_1 < s_2$. Ad esempio "`10`"<"`100`"; analogamente "`10`"<"`100`"<"`9`".

Il seguente esempio aiuterà a comprendere l'uso e la sintassi degli operatori di confronto fra stringhe:

```
1 [[ "$a" == "$b" ]] && echo "Sono uguali" || echo "Sono diversi"
```

Se le due espressioni da confrontare sono dei numeri interi, allora è possibile effettuare un confronto numerico anche utilizzando le doppie parentesi quadre come delimitatori dell'espressione; questo è utile nel caso in cui si debba definire una condizione mista, composta da confronti fra numeri, file e stringhe: in questo caso non è possibile utilizzare una condizione delimitata da doppie parentesi tonde (valutazione di espressioni aritmetiche), ma è necessario l'uso delle doppie parentesi quadre.

Confronto fra espressioni numeriche

Per il confronto fra espressioni numeriche devono essere utilizzati degli operatori appositi, distinti da quelli utilizzati in questo contesto per il confronto fra stringhe: l'interprete deve infatti distinguere tra un tipo di confronto e l'altro, dal momento che per le stringhe vale l'ordine lessicografico, diverso dall'ordinamento naturale dei numeri interi. Ad esempio 100 > 99, mentre invece la stringa "`100`" precede, nell'ordine lessicografico, la stringa "`99`", per cui risulta"`100`"<"`99`". Gli operatori di confronto fra espressioni numeriche sono riportati in Tabella 2.6.

Di seguito presentiamo un esempio con un'espressione condizionale mista, in cui viene effettuato il confronto tra stringhe e tra numeri interi:

```
1 [[ "$a" < "$b" && $a -lt $b ]] && \
```

[2] Per visualizzare la tabella dei codici ASCII in ambiente UNIX è sufficiente digitare il comando "`man ascii`".

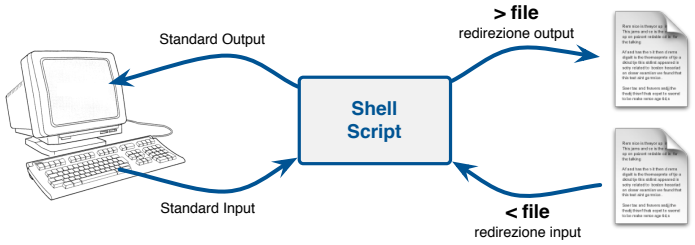

Figura 2.1: Schematizzazione del reindirizzamento dei canali di input e di output attraverso gli operatori di redirezione

```
2 echo "Sono in ordine come stringhe e come numeri" || \
3 echo "Non sono in ordine in almeno uno dei due casi"
```

Nel comando precedente, spezzato per comodità su tre righe di testo, a riga 1 viene valutata un'espressione condizionale "mista" (un confronto fra stringhe di caratteri e un confronto fra numeri): il valore delle variabili `a` e `b` viene confrontato utilizzando un operatore di confronto fra stringhe, per stabilire se `a` precede in ordine lessicografico `b`. Se questa condizione dà esito positivo (il confronto restituisce il valore *vero*) viene valutata anche la seconda condizione del comando composto: le stesse variabili `a` e `b` vengono confrontate verificando se il valore intero della prima è minore del valore intero della seconda. In caso di esito positivo anche di questo secondo confronto viene visualizzato il primo messaggio, in caso contrario (se il primo o, in subordine, il secondo confronto ha restituito il valore *falso*) viene visualizzato il secondo messaggio.

Ad esempio se `$a='123'` e `$b='456'` il comando visualizzerà il messaggio "`Sono in ordine come stringhe e come numeri`"; se invece `$a='abc1'` e `$b='abc2'` il comando visualizzerà il messaggio "`Non sono in ordine in almeno uno dei due casi`", perché i due valori sono in ordine lessicografico corretto, ma valutandone il contenuto come numeri interi, entrambe le variabili hanno un valore pari a zero e dunque non sono una minore dell'altra.

2.5 Redirezione dell'input/output

Una delle caratteristiche che rende maggiormente flessibile l'uso dei comandi della shell Bash e dei comandi esterni resi disponibili dal sistema operativo, è la possibilità di effettuare la cosiddetta *redirezione dell'input/output*. In pratica si tratta della possibilità di redirigere l'output prodotto da un comando e che generalmente sarebbe visualizzato sullo schermo del terminale dell'utente, verso un file o verso il canale di *standard input* di un altro programma; analogamente è possibile fare con l'input, facendo sì che l'input generalmente ricevuto da un programma attraverso la tastiera del terminale dell'utente, venga invece acquisito da un file.

Standard input, standard output e standard error

In generale ogni programma eseguito in modalità batch o interattiva ha a disposizione tre canali di input/output standard: il canale di *standard input*, con cui tipicamente riceve l'input dalla tastiera del terminale su cui opera l'utente, il canale di *standard output*, su cui invia i messaggi da visualizzare sul terminale dell'utente, ed il canale *standard error*, su cui vengono inviati i messaggi di errore prodotti dal programma stesso. Ai tre canali di input/output sono associati gli identificativi 0, 1 e 2, rispettivamente.

Mediante gli operatori di redirezione dell'I/O è possibile modificare il comportamento standard nell'uso di questi tre canali da parte del programma, reindirizzandoli su dei file o concatenando fra di loro più programmi, in modo che l'uno riceva in input ciò che un altro programma ha prodotto come output.

Redirezione dell'input da un file

Per reindirizzare il canale di input in modo che il comando acquisisca l'input da un file, si deve utilizzare l'operatore "<" (o anche "`0`<") per separare il comando dal nome del file:

comando < *file*

sort: ordina una sequenza di dati

Ad esempio, il comando "`sort`" legge in input una sequenza di righe di testo e le presenta in output ordinate in ordine crescente. Le righe di testo vengono lette generalmente da uno o più file il cui nome è specificato sulla riga di comando, come argomento del comando "`sort`". Tuttavia di *default* il comando legge le righe di testo dal canale di input standard, per cui è possibile eseguire il comando "`sort`" e poi digitare una di seguito all'altra le righe da ordinare; al termine dell'inserimento della sequenza è sufficiente battere la sequenza di tasti Control-D per concludere l'input e permettere al comando di visualizzare in output la sequenza ordinata in ordine lessicografico crescente. La sequenza di caratteri Control-D rappresenta il carattere di fine file (EOF: *end of file*) con cui il sistema operativo indica ad un programma che il file che sta leggendo è terminato e che non ci sono altri dati da leggere in input. Una volta ricevuto in input il carattere di fine file, il comando "`sort`" invia in output la sequenza ordinata, come nel seguente esempio:

```
$ sort
topo
cane
gatto
Control-D
cane
gatto
topo
```

Se la sequenza di parole da ordinare è contenuta in un file (il file "`animali`" nell'esempio che segue), allora è possibile redirigere il canale *standard input* in modo tale che l'input sia acquisito dal file:

```
$ sort < animali
cane
gatto
topo
```

Redirezione dell'output su un file

In modo analogo è possibile operare una redirezione dell'output inviandolo, anziché sullo schermo del terminale dell'utente, su un file. Supponiamo di voler ordinare una sequenza di nomi forniti in input dall'utente (che concluderà l'immissione dei dati digitando la sequenza di tasti Control-D) e di volerla memorizzare in un file. L'esempio che segue chiarisce l'uso dell'operatore ">" che consente di redirigere l'output di un comando; il comando viene riportato alla sinistra dell'operatore, mentre sulla destra viene indicato il nome di un file:

comando > *file*

```
$ sort > animali
topo
cane
gatto
Control-D
$ cat animali
```

```
cane
gatto
topo
```

Redirezione dell'input e dell'output di uno stesso comando

Naturalmente i comandi di redirezione dell'input e dell'output possono essere utilizzati simultaneamente, utilizzando una sintassi di questo tipo:

comando < *file input* > *file output*

Ad esempio, supponiamo che il file "`animali`" contenga un elenco di nomi di animali non ordinati. Con il comando che segue possiamo produrre il file "`animali_ordinato`" contenente gli stessi nomi, ma in ordine alfabetico crescente:

```
$ sort < animali > animali_ordinato
```

Il comando dell'esempio precedente non produce alcun messaggio sullo schermo del terminale, né richiede da parte dell'utente che sia digitata alcuna informazione in input, perché sia il canale di input standard che quello di output standard sono stati rediretti su dei file. L'esito del comando è comunque quello desiderato, anche se apparentemente non è accaduto nulla, come si può provare utilizzando il comando "`cat`" per visualizzare il contenuto del file "`animali_ordinato`":

```
$ cat animali_ordinato
cane
gatto
topo
```

Cancellazione e creazione del file su cui viene rediretto l'output

La shell, a fronte di un comando utilizzato con l'operatore di redirezione dell'output ">", crea il file specificato a destra dell'operatore e indirizza tutto ciò che il comando invia sul canale *standard output* su tale file; al termine dell'esecuzione del comando il file viene chiuso. Se il file già esiste, questo viene cancellato automaticamente viene ricreato un file con lo stesso nome.

Naturalmente per poter utilizzare l'operatore di redirezione dell'output l'utente che esegue lo script o il comando deve possedere i permessi di scrittura per la creazione del file sul filesystem.

Redirezione dell'output in modalità "append"

Per redirigere l'output su un file già esistente, senza cancellarlo, ma accodando l'output del comando al termine del contenuto già presente nel file, invece dell'operatore ">", si deve utilizzare l'operatore ">>". Ad esempio, il seguente comando, aggiunge le righe del file "`primo`" al file "`secondo`":

```
$ cat primo >> secondo
```

Come abbiamo accennato nelle pagine precedenti, il sistema operativo mette a disposizione dei programmi e dei comandi eseguiti dagli utenti, due diversi canali di output: lo *standard output* (identificato dal "descrittore" 1) e lo *standard error* (identificato dal "descrittore" 2). Il primo, come abbiamo visto, è il canale di output standard, che coincide con lo schermo del terminale dell'utente e può essere rediretto su un file (o su un altro *device*) mediante gli operatori ">" e ">>". Il secondo è dedicato alla visualizzazione di messaggi di errore o che comunque non rientrano nel normale output del programma. Redirigendo su un file lo *standard output* i messaggi inviati sul canale *standard error* saranno comunque visualizzati sul terminale dell'utente.

A titolo di esempio consideriamo il seguente programma scritto in linguaggio C, che visualizza due messaggi sui due diversi canali di output.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 int main(void) {
4   fprintf(stdout, "Messaggio inviato su standard output.\n");
5   fprintf(stderr, "Messaggio inviato su standard error.\n");
6   return(0);
7 }
```

Per poter eseguire il programma precedente è necessario compilarlo, producendo un programma eseguibile in linguaggio macchina. Per far questo bisogna eseguire il compilatore C, che generalmente, sui sistemi UNIX che ne sono dotati, corrisponde al comando "`cc`" o "`gcc`". Supponiamo quindi che il sorgente del programma in linguaggio C sia memorizzato nel file "`programma.c`"; con il seguente comando si esegue il compilatore C e si ottiene in output il file eseguibile "`programma`".

```
$ cc programma.c -o programma
```

Lanciando in esecuzione il programma eseguibile si ottiene l'output seguente:

```
$ ./programma
Messaggio inviato su standard output.
Messaggio inviato su standard error.
```

Utilizzando l'operatore di redirezione dell'output verso un file si ottiene esclusivamente la redirezione dell'output inviato al canale *standard output*, mentre ciò che è stato inviato su *standard error* sarà comunque visualizzato sul terminale dell'utente:

```
$ ./programma > file
Messaggio inviato su standard error.
$ cat file
Messaggio inviato su standard output.
```

Redirezione del canale standard error

Per redirigere su un file ciò che è stato inviato sul canale *standard error* è necessario utilizzare l'operatore "`2>`", con la stessa sintassi con cui si è utilizzato l'operatore "`>`"; è possibile anche redirigere i due canali di output (*standard output* e *standard error*) su due file differenti con lo stesso comando, utilizzando entrambi gli operatori di redirezione dell'output "`>`" e "`2>`":

```
$ ./programma 2> file
Messaggio inviato su standard output.
$ cat file
Messaggio inviato su standard error.
$ ./programma > file.output 2> file.error
$ cat file.output
Messaggio inviato su standard output.
$ cat file.error
Messaggio inviato su standard error.
```

È possibile anche redirigere il canale *standard error* sullo *standard output* o viceversa: nel primo caso si userà l'operatore di redirezione "`2>&1`", mentre nel secondo caso si userà l'operatore "`1>&2`". Infine, con l'operatore di redirezione "`&>`" si redirigono entrambi i canali *standard output* e *standard error* su un file:

```
$ ./programma &> file
$ cat file
Messaggio inviato su standard error.
Messaggio inviato su standard output.
```

{ ... } > file: redirezione dell'output prodotto da più programmi su uno stesso file

Chiudiamo questa sezione, ricordando che nelle pagine precedenti avevamo introdotto la sintassi "`{` *lista di comandi*`; }`", dicendo che poteva essere utile per redirigere l'output di più programmi eseguiti in successione, su uno stesso file. Per far questo è possibile utilizzare un'espressione di questo tipo:

$$\texttt{\{}\ comando_1\texttt{;}\ comando_2\texttt{;}\ \ldots\texttt{;}\ comando_n\texttt{;}\ \texttt{\}} > file$$

Di seguito riportiamo un esempio elementare di redirezione su uno stesso file dell'output prodotto da più comandi lanciati in sequenza:

```
$ { date +%d/%m/%Y; finger; } > utenti_collegati
$ cat utenti_collegati
26/06/2011
Login    Name                 TTY  Idle  Login  Time   Office  Phone
marco    Marco Liverani      *con  4:31  Dom    18:49
marco    Marco Liverani       s00        Dom    19:24
```

Lo stesso risultato poteva essere ottenuto senza utilizzare le parentesi graffe per costruire il comando composto e usando l'operatore di redirezione "`>>`" (*append*) per redirigere l'output del secondo comando sul file, senza distruggere l'output prodotto dal primo comando. Ad esempio:

```
$ date +%d/%m/%Y > utenti_collegati; finger >> utenti_collegati
$ cat utenti_collegati
26/06/2011
Login    Name                 TTY  Idle  Login  Time   Office  Phone
marco    Marco Liverani      *con  4:31  Dom    18:49
marco    Marco Liverani       s00        Dom    19:24
```

2.6 Comandi in pipeline

Pipe: collegamento dei canali di input e di output di due o più programmi

Oltre alla redirezione dell'input e dell'output, una delle caratteristiche più interessanti ed utili della shell Bash è costituita dalla possibilità di concatenare tra loro due o più programmi inviando sul canale di *standard input* di uno ciò che viene prodotto sul canale di *standard output* di un altro. Questa operazione di concatenazione di programmi attraverso i loro canali di input e di output è nota in gergo con il nome di *pipe* ("tubo", in italiano). L'operatore di *pipe* su Bash è rappresentato dal carattere "`|`": l'output del comando alla sinistra dell'operatore viene passato in input al comando posto sulla destra dell'operatore.

Ad esempio, con la seguente istruzione si invia l'output del comando "`ls -1`" al comando "`wc -l`":

```
$ ls -1 | wc -l
15
```

Nell'esempio viene eseguito il comando "`ls`" con l'opzione "`-1`" per ottenere un elenco dei nomi dei file presenti nella directory corrente, costituito da un solo nome per ogni

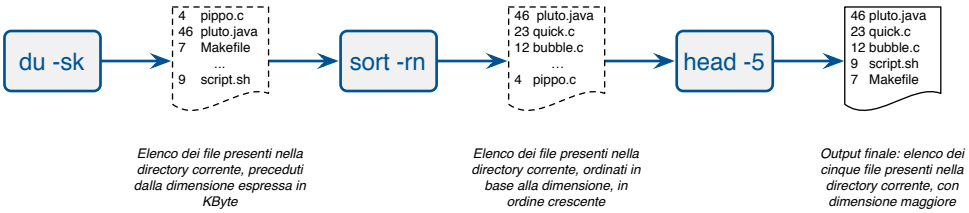

Figura 2.2: Schematizzazione della concatenazione di comandi con l'operatore *pipe*; solo l'ultimo degli output prodotto dai comandi della *pipeline* viene visualizzato sul terminale dell'utente, mentre gli altri output intermedi (nella figura rappresentati in un box tratteggiato) sono passati da un comando all'altro, senza essere mostrati all'utente

riga; l'output di questo comando (una sequenza di righe contenente ciascuna il nome di un file), viene passato in input al comando "`wc`" che, con l'opzione "`-l`", esegue il conteggio delle righe in input e lo stampa in output. In questo modo, con la concatenazione mediante il *pipe* dei due comandi, viene visualizzato il numero di file presenti nella directory corrente.

wc: conta il numero di linee (o anche le parole o i caratteri) in input

Sulla stessa riga di comando possono essere utilizzati più operatori di *pipe* per costruire delle vere e proprie catene di operazioni che, prima di produrre l'output finale, passano i dati attraverso una sequenza di "filtri".

```
$ du -sk * | sort -rn | head -5
10348   mail
10200   src
10044   tex
7344    documenti
5876    tmp
```

du: fornisce informazioni sullo spazio usato sul filesystem

Quest'ultimo esempio è più complesso del precedente, perché utilizza una doppia concatenazione con *pipe*. Il comando "`du`" (*disk usage*) con l'opzione "`-sk`" produce l'elenco dei nomi dei file presenti nella directory corrente preceduti dalla dimensione di ciascuno in KByte. L'output di questo comando viene passato in input al comando "`sort`" che, con le opzioni "`-rn`", dispone in ordine decrescente ("`-r`", *reverse*) le righe ricevute in input considerando l'ordinamento numerico ("`-n`", *numeric*) e non quello lessicografico basato sulla sequenza dei caratteri nella tabella dei codici ASCII con cui si sarebbe ottenuto, ad esempio, che "`10`"<"`2`"; il comando "`sort`" quindi si occupa di "filtrare" l'output del comando "`du`", rielaborarlo cambiando l'ordine delle righe, per passarlo poi in input al comando "`head`". Quest'ultimo comando, con l'opzione "`-5`", visualizza solo le prime cinque righe ricevute in input.

head: seleziona solo le prime righe dalla sequenza ricevuta in input

Complessivamente quindi l'istruzione composta del precedente esempio consente di visualizzare i nomi dei cinque file di dimensione maggiore tra quelli presenti nella directory corrente. Possiamo schematizzare quest'ultimo esempio con il disegno riportato in Figura 2.2.

Anche con l'operatore *pipe* è possibile utilizzare le parentesi graffe per aggregare l'output prodotto da più comandi, prima di inviarlo in input al comando successivo; anche in questo caso la sintassi è la seguente:

{ ... } | ...: invio in pipeline dell'output aggregato prodotto da più comandi

$$\{\ comando_1\ ;\ comando_2\ ;\ \ldots\ ;\ comando_n\ ;\ \}\ \mid\ comando_{n+1}$$

Ad esempio, il seguente comando aggrega l'output prodotto dai due comandi "`du`" eseguiti con parametri diversi: il primo calcola la dimensione di tutti i file con estensione "`.c`" presenti nella sottodirectory "`src`", il secondo calcola la dimensione dei file con

estensione "`.tex`" presenti nella sottodirectory "`tex`"; l'output aggregato, come se fosse prodotto da un solo comando, viene inviato in input (attraverso l'operatore *pipe*) al comando "`sort`"; l'output prodotto da quest'ultimo viene infine inviato al programma "`head`" che, con l'opzione "`-6`", restituirà sullo schermo del terminale solo le prime sei righe dell'output prodotto da "`sort`":

```
$ { du -sk src/*.c ; du -sk tex/*.tex ; } | sort -rn | head -6
68    tex/grafi_clique.tex
20    src/grafica.c
16    tex/IN1.tex
8     tex/soluzione.tex
8     tex/kruskal.tex
8     src/cgilib.c
```

Capitolo 3

Variabili, variabili d'ambiente e variabili speciali

Come ogni linguaggio che si rispetti anche Bash possiede il concetto di *variabile*, ossia di area di memoria della macchina, identificata da un nome ben preciso, definito dalla shell o dall'utente, in cui possono essere memorizzate delle informazioni. Con Bash, come con altri linguaggi di scripting e shell del sistema operativo, è necessario però distinguere fra tre tipi diversi di variabili: le variabili "semplici", le variabili "d'ambiente", condivise fra più programmi eseguiti nell'ambito della medesima shell e variabili "speciali", il cui contenuto ha un significato ben preciso che condiziona il funzionamento della shell o l'esecuzione di determinati programmi.

3.1 Variabili scalari e array

Assegnazione di un valore ad una variabile scalare

Per definire una variabile nell'ambito di una shell Bash o di uno script è sufficiente utilizzare l'operatore di assegnazione "=" con la sintassi "*variabile*=*valore*". Il nome della variabile deve iniziare con un carattere non numerico e non deve contenere spazi o altri caratteri con un significato speciale per Bash (come, ad esempio, i caratteri "=", "*", "$" o "&"). Se il valore da assegnare alla variabile contiene degli spazi, questi devono essere preceduti dal carattere "\", oppure l'intera stringa da assegnare alla variabile deve essere delimitata da singoli o doppi apici. Di seguito riportiamo alcuni esempi.

```
$ a=17
$ nome='Marco Liverani'
$ filename=nome\ file
```

È bene osservare che nell'assegnare un valore ad una variabile non devono esserci spazi tra il nome della variabile e l'operatore di assegnazione "=" e tra questo ed il valore della variabile.

Riferimento alle variabili

Dopo averne definito il valore, è possibile utilizzare una variabile in un'espressione o in un'altra istruzione. Per riferirsi ad una variabile e per distinguere il nome da una qualsiasi altra stringa di caratteri, il nome deve essere preceduto dal simbolo "$", con un'espressione del tipo "$*variabile*"; in alternativa si può anche utilizzare l'espressione "${*variabile*}", sostanzialmente equivalente alla precedente nel caso delle variabili scalari.

Ad esempio, utilizzando il comando "echo" per visualizzare in output un'espressione è possibile visualizzare il valore assegnato precedentemente ad una variabile:

```
$ echo $nome
Marco Liverani
```

```
$ echo ${filename}
nome file
```

Lunghezza di una stringa assegnata ad una variabile

Con l'espressione "`${#variabile}`" si ottiene la lunghezza in caratteri dell'espressione assegnata alla variabile. Ad esempio:

```
$ echo ${#filename}
9
```

Variabili scalari e array

Ricordiamo che una variabile si dice "scalare" se può contenere un solo valore per volta. In alternativa, per memorizzare un insieme di valori, Bash mette a disposizione la struttura dati di *array*, con cui è possibile memorizzare una sequenza di informazioni. Di fatto un array è una collezione di variabili scalari denominate con lo stesso nome e distinte l'una dall'altra mediante un indice numerico intero non negativo; il valore dell'indice va da zero, per identificare il primo elemento della sequenza memorizzata nell'array, fino ad $n-1$, indicando con n il numero di elementi presenti nella sequenza.

Definizione esplicita di un array

Per definire esplicitamente un array si può utilizzare l'espressione "*array*=(*valore*$_0$ *valore*$_1$ *valore*$_{n-1}$)". Se i valori sono costituiti da stringhe contenenti degli spazi, tali valori devono essere delimitati da apici o doppi apici. Ad esempio è possibile definire degli array di valori con i seguenti comandi:

```
$ a=(10 20 30 40)
$ rapaci=(aquila 'falco pellegrino' allocco gheppio)
```

Riferimento agli elementi di un array

Per referenziare i singoli valori di un array si usa l'espressione "`${array[i]}`"; naturalmente i è il numero intero che identifica l'indice dell'elemento dell'array a cui ci si vuole riferire, con $0 \leq i < n$ in un array con n elementi. Facendo riferimento all'esempio precedente, potremo eseguire i seguenti comandi che possono aiutare a capire la sintassi di Bash nell'uso degli array:

```
$ echo $rapaci
aquila
$ echo ${rapaci[0]}
aquila
$ echo ${rapaci[1]}
falco pellegrino
$ a=${rapaci[1]}
$ echo $a
falco pellegrino
$ a=(${rapaci[1]})
$ echo ${a[1]}
pellegrino
```

Numero di elementi di un array

Con l'espressione "`${#array[*]}`" si ottiene il numero di elementi presenti in un array; l'indice del primo elemento dell'array è 0, mentre l'indice dell'ultimo elemento è `${#array[*]}`-1. Ad esempio:

```
$ echo ${#rapaci[*]}
4
```

Infine, per riferirsi a tutti gli elementi di un array si può scrivere "`${array[*]}`"; così, ad esempio, per stampare tutti gli elementi di un array o per copiare tutto il contenuto di un array in una variabile scalare come un'unica stringa, si possono utilizzare le seguenti istruzioni:

```
$ echo ${rapaci[*]}
aquila falco pellegrino allocco gheppio
$ rap=${rapaci[*]}
$ echo $rap
aquila falco pellegrino allocco gheppio
```

3.2 Variabili d'ambiente

Il termine *shell* ("conchiglia", in inglese) rende bene l'idea di un involucro che "avvolge" il suo contenuto e lo scherma dall'ambiente esterno; all'interno dell'involucro "vivono" variabili, funzioni e processi, che sono in parte invisibili all'esterno dell'involucro stesso. Definendo una variabile ed assegnandogli un valore con l'operatore "=", la sua visibilità è limitata alla shell in cui è stata definita; eseguendo uno script che opera in un'istanza della shell differente, la variabile non sarà visibile nella seconda shell e quindi il suo valore non sarà noto allo script eseguito in questo secondo ambiente. Un esempio aiuterà forse a comprendere meglio questo aspetto:

Visibilità delle variabili definite nella shell

```
$ a=1
$ bash
$ echo $a

$ exit
$ echo $a
1
```

Una *variabile d'ambiente* è una variabile definita in modo da essere resa visibile alla shell in cui è stata definita e a tutte le shell eseguite da quest'ultima. Per definire una variabile d'ambiente si deve usare il comando "`export`". Questo comando può essere usato per definire una variabile d'ambiente e contestualmente assegnarle un valore, oppure semplicemente per definire come variabile d'ambiente una variabile, senza assegnarle un valore (che può essere assegnato prima o dopo l'uso del comando "`export`"). Nella stessa istruzione possono essere definite numerose variabili d'ambiente elencandole nella stessa riga di comando, separate l'una dall'altra da uno spazio.

Visibilità delle variabili d'ambiente

export: definizione di variabili d'ambiente

L'uso del comando è il seguente:

`export` *variabile*$_1$ *variabile*$_2$ ...

oppure

`export` *variabile*$_1$=*valore*$_1$ *variabile*$_2$=*valore*$_2$ ...

Siccome il comando "`export`" si aspetta come argomento una sequenza di nomi di variabili, questi non devono essere preceduti dal simbolo "`$`". Di seguito riportiamo un esempio d'uso del comando `export`:

```
$ export a=1
$ b=2
$ echo $a $b
1 2
$ bash
$ echo $a $b
1
$ b=3
```

```
$ echo $a $b
1 3
$ exit
exit
$ echo $a $b
1 2
```

Nell'esempio precedente si nota come il valore della variabile b non sia visibile alla seconda shell Bash, lanciata all'interno della prima (e completamente contenuta nella prima, come in un gioco di scatole cinesi). Inoltre, il valore assegnato a b nella sotto-shell non è visibile alla shell principale. Al contrario, la visibilità di a è estesa ad entrambe le shell, visto che a è una variabile d'ambiente.

Anche uno script viene eseguito in una shell diversa da quella attraverso la quale viene lanciato. Ad esempio consideriamo il seguente script elementare, salvato nel file "stampa.sh":

```
1 #!/bin/bash
2 echo -n "Variabile a="
3 echo $a
```

Nello script di esempio il valore della variabile a non viene definito, quindi ci si aspetta che questo sia stato definito prima di eseguire lo script. Naturalmente la variabile a dovrà essere definita come variabile d'ambiente, altrimenti lo script non potrà avere visibilità sul valore di quella variabile. Il seguente esempio illustra il comportamento della shell:

```
$ a=17
$ ./stampa.sh
Variabile a=
$ echo $a
17
$ export a
$ ./stampa.sh
Variabile a=17
```

env: visualizza le variabili d'ambiente definite nella shell

Per visualizzare tutte le variabili d'ambiente impostate nella sessione corrente, è sufficiente digitare il comando "env"; un risultato analogo è possibile ottenerlo con il comando "export -p". Nell'esempio seguente viene presentato un'estratto delle variabili d'ambiente definite nella mia sessione utente:

```
$ env
TERM=xterm
SHELL=/bin/bash
CVSROOT=/usr/local/CVSrepository
USER=marco
PATH=/bin:/usr/bin:/usr/local/bin:/usr/X11/bin
PWD=/Users/marco
PS1=\$
HOME=/Users/marco
LOGNAME=marco
_=/usr/bin/env
```

Meta-carattere	Descrizione
\h	il nome dell'host su cui viene eseguita la Bash
\u	lo username dell'utente che sta eseguendo la Bash
\w	il path completo della directory corrente
\W	la directory corrente (priva del path completo)
\!	il numero progressivo del comando nella sequenza dei comandi eseguiti nell'ambito della shell
\$	il carattere "#" se lo userid dell'utente è 0, il carattere "$" altrimenti (per evidenziare se la shell è eseguita come utente amministratore del sistema, root, o meno)

Tabella 3.1: Alcuni meta-caratteri per la definizione di un "prompt dinamico"

3.3 Variabili speciali

La shell Bash dispone di alcune variabili "speciali" che consentono di modificare la configurazione della shell stessa o di alcuni programmi che operano nell'ambito della shell, oppure consentono di fornire alcune utili informazioni agli script in esecuzione nella shell. Distinguiamo quindi due tipi di variabili speciali:

- quelle che consentono di personalizzare la configurazione dell'ambiente entro il quale sono eseguiti i comandi della shell; queste variabili possono essere impostate dall'utente e spesso, per alcune di esse, viene preimpostato un valore dalla configurazione di base del sistema operativo ospite o della shell dell'utente;
- quelle che forniscono informazioni utili sullo stato della shell agli script ed ai comandi in esecuzione nell'ambito della shell stessa; il valore di queste variabili non può essere impostato o modificato dall'utente o da comandi e script eseguiti dall'utente, che potranno invece limitarsi ad utilizzare i valori impostati autonomamente dalla shell.

Variabili speciali per la configurazione dell'ambiente per l'esecuzione della Bash

Nella prima famiglia di variabili speciali rientrano, ad esempio, le variabili d'ambiente HOME, PS1 e PATH. La prima indica il *path* della *home directory* dell'utente, ossia della directory che viene resa attiva come directory di lavoro corrente al momento del login sul sistema e ogni volta che viene usato il comando "cd" senza alcun parametro aggiuntivo. Tipicamente, in ambiente UNIX, il valore della variabile HOME è impostato come "/home/username", dove con "username" si intende l'identificativo dell'utente (ad esempio: "/home/marco" per l'utente "marco").

PS1: variabile per la definizione del prompt della shell interattiva

La variabile PS1 contiene una stringa che viene visualizzata dalla shell come *prompt* per le sessioni interattive con l'utente. La stringa può essere una costante (es.: "export PS1='pippo '") oppure può contenere dei "meta-caratteri" che consentono di costruire dinamicamente il prompt dei comandi. I meta-caratteri sono tutti preceduti dal carattere "\" (*backslash*); una elencazione completa dei meta-caratteri supportati per la creazione di un prompt "dinamico" si trova nella sezione "Prompting" della pagina di manuale della Bash[1]. Una sintesi di questi simboli viene proposta in Tabella 3.1.

Il seguente esempio può far comprendere meglio il significato dei meta-caratteri per la definizione del prompt "dinamico":

```
$ export PS1='pippo> '
pippo> export PS1='\u@\h:\w\$ '
marco@aquilante:~/src$ export PS1='\$ '
$
```

[1] Ricordiamo che la pagina di manuale della Bash, sui sistemi UNIX, può essere visualizzata con il comando "man bash".

Variabile	Descrizione
BASH_ARGC	il numero di parametri forniti alla shell sulla riga di comando
BASH_ARGV	un array contenente l'elenco dei parametri forniti alla shell sulla riga di comando (l'elenco è ordinato dall'ultimo al primo, come una pila)
BASH_VERSION	il numero di versione della shell Bash
HOSTNAME	il nome dell'host su cui viene eseguita la shell Bash
MACHTYPE	il tipo di macchina (architettura hardware o CPU) su cui viene eseguita la shell Bash
OSTYPE	il nome e la versione del sistema operativo su cui viene eseguita la shell Bash
PPID	il *process identifier* (PID) del processo padre della shell Bash corrente
PWD	la directory corrente in cui viene eseguita la shell Bash
RANDOM	ogni volta che viene utilizzata questa variabile il suo valore cambia in modo casuale tra 0 e $2^{15} - 1 = 32.767$
UID	lo *user identifier*, il numero progressivo che identifica univocamente l'utente che sta eseguendo la Bash

Tabella 3.2: Alcune variabili speciali gestite direttamente dalla shell

PATH: variabile con l'elenco delle directory in cui l'interprete cerca i programmi eseguibili

La variabile d'ambiente PATH, infine, contiene un elenco di directory, separate dal carattere ":", in cui la shell cerca i comandi impartiti in modalità interattiva dall'utente o presenti in uno script, qualora tali comandi non siano invocati con un *path* assoluto o relativo alla directory corrente. Ad esempio, quando nella shell viene impartito il comando "ls", l'interprete dei comandi, non riconoscendo il comando come interno alla shell, cerca il relativo file eseguibile in una delle directory elencate nella variabile PATH. In genere sui sistemi UNIX questo file eseguibile si trova nella directory "/bin".

L'effetto della modifica del valore di queste variabili è immediato, ma viene perso al termine della sessione di lavoro. Per questo motivo spesso il valore di queste variabili d'ambiente viene definito nel file di configurazione della Bash, tipicamente nel file "~/.bashrc".

Variabili speciali che forniscono informazioni sullo stato della shell

Come abbiamo accennato in precedenza, altre variabili speciali forniscono utili informazioni sullo stato della shell; in questo caso il valore delle variabili viene gestito autonomamente dalla shell stessa e non può essere impostato con un comando impartito dall'utente. Le variabili di questo tipo sono molto numerose; in Tabella 3.2 ne sono riportate solo alcune, con una descrizione sintetica.

RANDOM: variabile per l'accesso alle sequenze di numeri pseudo-casuali

La variabile RANDOM consente di sfruttare il generatore di numeri interi pseudo-casuali della Bash: ogni volta che viene utilizzata la variabile il suo valore viene modificato in modo casuale. La sequenza di numeri pseudo-casuali viene inizializzata autonomamente dalla shell; tuttavia per riprodurre più volte una determinata sequenza, è possibile fornire un "seme" di inizializzazione assegnando un valore alla variabile RANDOM. Il seguente esempio (script "random.sh") visualizza i primi due numeri casuali di una sequenza inizializzata autonomamente dalla Bash e poi i primi due numeri casuali della sequenza ottenuta utilizzando come seme il valore 10:

```
1 #!/bin/bash
2 echo "r1 = $RANDOM"
3 echo "r2 = $RANDOM"
4 RANDOM=10
5 echo "r3 = $RANDOM"
6 echo "r4 = $RANDOM"
```

Eseguendo lo script si ottiene il risultato riportato di seguito; lo script viene eseguito due volte per evidenziare il fatto che la sequenza di numeri casuali prodotta autonomamente dalla Bash (i primi due numeri r_1 e r_2) è costituita da numeri che con elevata probabilità sono sempre diversi, mentre la sequenza prodotta utilizzando un valore costante come "seme" (i numeri r_3 e r_4), è costituita sempre dagli stessi numeri.

```
$ ./random.sh
r1 = 18550
r2 = 29138
r3 = 4543
r4 = 28214
$ ./random.sh
r1 = 19458
r2 = 30487
r3 = 4543
r4 = 28214
```

Lanciando uno script è possibile specificare sulla riga di comando uno o più parametri (argomenti) separati l'uno dall'altro con uno spazio (se un parametro contiene degli spazi, deve essere delimitato con gli apici). Lo script può accedere ai parametri forniti sulla linea di comando a "*run time*", utilizzando la variabile d'ambiente BASH_ARGV, un array dove i parametri sono memorizzati con una struttura a "pila": questo significa che ogni parametro viene memorizzato all'inizio della struttura dati; quindi il primo dei parametri presenti sulla linea di comando si trova memorizzato come ultimo elemento dell'array, mentre l'ultimo dei parametri si trova nella prima posizione dell'array. Il seguente esempio aiuterà a comprendere questo meccanismo. Consideriamo il seguente shell script, memorizzato nel file "parametri.sh":

BASH_ARGV: array con i parametri passati sulla linea di comando alla Bash

```
1 #!/bin/bash
2 echo "Numero di parametri sulla riga di comando: $BASH_ARGC"
3 echo "Primo elemento del vettore: ${BASH_ARGV[0]}"
4 echo "Secondo elemento del vettore: ${BASH_ARGV[1]}"
5 echo "Terzo elemento del vettore: ${BASH_ARGV[2]}"
```

Eseguendo lo script si ottiene il risultato seguente, in cui risulta che il valore della variabile ${BASH_ARGV[0]}, il primo elemento dell'array, è in realtà l'ultima delle stringhe digitate dall'utente sulla riga di comando e, viceversa, il valore di ${BASH_ARGV[2]}, l'ultimo degli elementi del vettore, è pari alla prima delle tre stringhe digitate dall'utente:

```
$ ./parametri.sh pippo pluto paperino
Numero di parametri sulla riga di comando: 3
Primo elemento del vettore: paperino
Secondo elemento del vettore: pluto
Terzo elemento del vettore: pippo
```

Altre variabili gestite autonomamente dalla shell consentono di ottenere informazioni sui parametri passati sulla linea di comando dello shell script o sul numero di processo (PID, *process identifier*) dello script stesso.

$$: variabile che contiene il PID dello script o del comando in esecuzione

La variabile $$ contiene il PID (*process ID*) dello script in esecuzione; questa informazione potrebbe essere utile per definire il nome di un file temporaneo da creare in una directory, in modo tale da ridurre la probabilità di entrare in conflitto con il nome di un altro file creato dallo stesso script o da un altro programma. Ad esempio il seguente comando:

```
$ grep pippo *.c > /tmp/pippo.$$
```

consente di memorizzare l'output prodotto dal comando "`grep`" nel file "`pippo.`*n*", dove *n* è il PID del processo che esegue il comando stesso. In questo modo lanciando due volte lo stesso comando saranno prodotti due file di output differenti (cambia il numero utilizzato come estensione del nome del file).

$@**, **$#, **$1**, **$2**, ...: variabili per l'accesso ai parametri passati sulla riga di comando

La variabile speciale `$@` fornisce una concatenazione di tutti i parametri forniti allo script sulla riga di comando. La variabile `$#` restituisce invece il numero di parametri passati sulla riga di comando. Tali parametri, oltre che mediante l'array `BASH_ARGV`, sono accessibili anche mediante le variabili `$1`, `$2`, ecc. La variabile `$0` invece, fornisce il comando con cui è stato eseguito lo script, privo dei parametri riportati sulla stessa riga di comando. I parametri sono memorizzati nelle variabili `$1`, `$2`, `$3`, ..., esattamente nell'ordine con cui compaiono sulla riga di comando.

shift: estrae dalla pila dei parametri il primo elemento, spostando di una posizione tutti gli altri

Il comando "`shift`" consente di spostare di una posizione a sinistra tutti i parametri passati allo script sulla riga di comando, eliminando in questo modo il primo: dopo l'esecuzione del comando "`shift`" la variabile `$1` conterrà il secondo dei parametri, la variabile `$2` conterrà il terzo e così via. Con l'esecuzione di "`shift`" vengono modificati anche i valori delle variabili `$#` e `$@`. Il seguente script (file "`parametri_bis.sh`") illustra il funzionamento del comando "`shift`" ed il significato delle variabili speciali:

```
1  #!/bin/bash
2  echo "Numero di parametri: $#"
3  echo "Parametri: $@"
4  echo "Comando: $0"
5  echo "Primo: '$1', Secondo: '$2', Terzo: '$3'"
6  shift
7  echo "Numero di parametri: $#"
8  echo "Parametri: $@"
9  echo "Comando: $0"
10 echo "Primo: '$1', Secondo: '$2', Terzo: '$3'"
```

Di seguito riportiamo l'output prodotto dallo script precedente:

```
$ ./parametri_bis.sh uno due tre
Numero di parametri: 3
Parametri: uno due tre
Comando: ./parametri_bis.sh
Primo: 'uno', Secondo: 'due', Terzo: 'tre'
Numero di parametri: 2
Parametri: due tre
Comando: ./parametri_bis.sh
Primo: 'due', Secondo: 'tre', Terzo: ''
```

Dopo l'esecuzione del comando "`shift`" i parametri si riducono da tre a due e i valori vengono spostati dalla variabile `$2` alla `$1` e dalla `$3` alla `$2`; il valore della variabile `$3`, in seguito allo *shift* dei parametri, risulta nullo. Vengono modificati di conseguenza anche i valori delle variabili `$@` e `$#`.

Come vedremo in seguito, il comando "`shift`" e le variabili `$1`, `$2`, `$3`, ..., vengono utilizzati anche per l'elaborazione dei parametri passati ad una funzione definita in uno script.

Capitolo 4

Strutture di controllo

Fino ad ora abbiamo visto caratteristiche fondamentali della Bash che consentono di definire variabili ed eseguire sequenze di comandi. Per costruire strutture algoritmiche più complesse è necessario utilizzare delle istruzioni che permettano di implementare strutture di controllo condizionali o iterative, come avviene in tutti i linguaggi di programmazione strutturata.

4.1 Strutture di controllo condizionali

Una struttura di controllo condizionale consente di valutare un'espressione logica e, sulla base del suo valore (*vero* o *falso*), eseguire determinate istruzioni piuttosto che altre. La forma più elementare di questa struttura di controllo è la seguente:

se *condizione* **allora**
  istruzioni
fine-condizione

Struttura condizionale "**if... then... fi**"

Nel linguaggio Bash questa struttura viene implementata dalle istruzioni "`if`" e "`fi`" secondo la seguente sintassi:

```
1 if lista di comandi; then
2   lista di comandi;
3 fi
```

L'istruzione "`fi`" chiude la struttura di controllo condizionale, aperta dall'istruzione "`if`": in altri termini ad ogni istruzione "`if`" deve corrispondere, nelle righe successive, un'istruzione "`fi`". La lista dei comandi che segue l'istruzione "`then`" viene eseguita se la lista dei comandi che segue l'istruzione "`if`" restituisce il valore 0 (*vero*).

Valutazione del return code di un comando

Ogni comando eseguito dalla shell restituisce un valore numerico che la shell interpreta come "vero" se tale valore è zero, o "falso" se il comando restituisce un qualsiasi codice diverso da zero. Pertanto ciascun comando, anche composto, può essere utilizzato come condizione in un'istruzione "`if`". Nell'esempio seguente si utilizza il return code del comando "`grep`" come condizione per verificare se l'utente "`root`" è definito sul sistema.

```
1 #!/bin/bash
2 if grep -q root /etc/passwd; then
3   echo "L'utente root e' definito!"
4 fi
```

Utilizzando invece la sintassi "`[[...]]`", illustrata nella sezione 2.4 a pagina 13, è possibile eseguire confronti fra variabili e sullo stato dei file. Le espressioni che esprimono

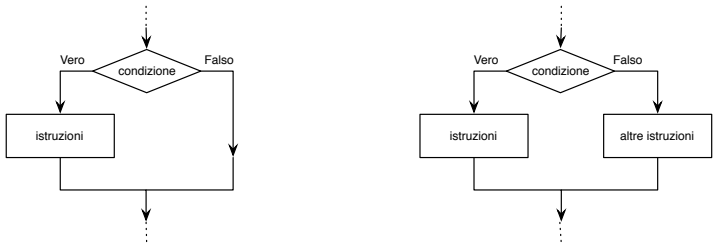

Figura 4.1: Diagrammi di flusso delle strutture di controllo condizionali: (a) "if ... then ... fi" e (b) "if ... then ... else ... fi".

una condizione logica che deve essere valutata interpretando specifici operatori, devono essere delimitate da coppie di doppie parentesi quadrate "`[[...]]`" per poter essere elaborate dalla Bash. Ad esempio il seguente script visualizza il numero di utenti definiti sul sistema se il file "`/etc/passwd`" esiste.

Valutazione di condizioni logiche

```
1 #!/bin/bash
2 if [[ -e /etc/passwd ]]; then
3   wc -l /etc/passwd
4 fi
```

Una versione più completa della struttura di controllo condizionale consente di gestire due liste di istruzioni alternative: la prima viene eseguita nel caso in cui il valore della condizione sia "vero", la seconda viene eseguita se invece il valore della condizione è "falso":

Struttura condizionale "**if... then... else... fi**"

se *condizione* **allora**
  istruzioni
altrimenti
  istruzioni
fine-condizione

Questa struttura algoritmica è implementata dalle istruzioni "`if` ... `then` ... `else` ... `fi`":

```
1 if lista di comandi; then
2   lista di comandi
3 else
4   lista di comandi
5 fi
```

Lo script presentato nell'esempio precedente può essere quindi modificato aggiungendo la clausola "`else`" come segue:

```
1 #!/bin/bash
2 if [[ -e /etc/passwd ]]; then
3   wc -l /etc/passwd
4 else
5   echo 'Il file /etc/passwd non esiste'
6 fi
```

In Figura 4.1 sono rappresentati i diagrammi di flusso corrispondenti alla struttura di controllo "`if` ... `then` ... `fi`" e alla struttura "`if` ... `then` ... `else` ... `fi`".

È possibile codificare una struttura condizionale più articolata, con la verifica di condizioni "a cascata", secondo il seguente schema:

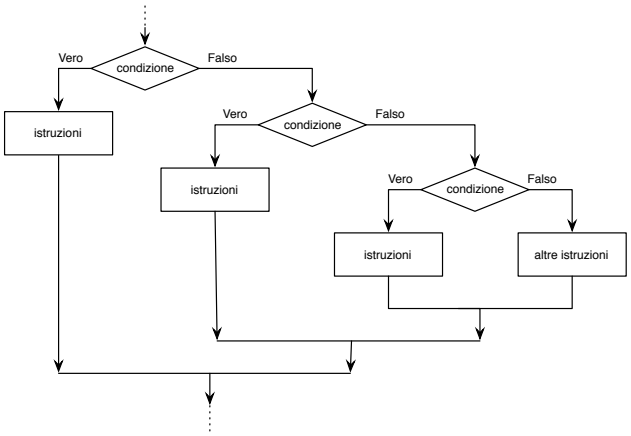

Figura 4.2: Diagramma di controllo delle istruzioni per la definizione di una struttura di controllo condizionale a cascata: "if ... then ... elif ... then ... else ... fi"

se $condizione_1$ **allora**
  $istruzioni_1$
altrimenti se $condizione_2$ **allora**
  ...
altrimenti se $condizione_n$ **allora**
  $istruzioni_n$
altrimenti
  $istruzioni_{n+1}$
fine-condizione

Se risulta verificata la condizione i-esima, allora viene eseguito il blocco i-esimo di istruzioni. Se nessuna delle condizioni risulta verificata, viene eseguito il blocco "else" della struttura, l'$n+1$-esimo blocco di istruzioni. Con il linguaggio della Bash questa struttura viene implementata introducendo una o più istruzioni "`elif`" in una struttura condizionale, ottenendo un costrutto del tipo "`if` ... `then` ... `elif` ... `then` ... `else` ... `fi`"; la struttura condizionale ottenuta concatenando più istruzioni di valutazione delle condizioni assume la seguente forma:

Struttura condizionale "**if... then... elif... then... else... fi**"

```
1 if lista di comandi; then
2   lista di comandi;
3 elif lista di comandi; then
4   lista di comandi;
5   ...
6 else
7   lista di comandi;
8 fi
```

Supponiamo ad esempio di dover predisporre uno script che esegue operazioni diverse in base al parametro passato sulla riga di comando. Un esempio di uno script di questo genere è il seguente (memorizzato nel file "`chi.sh`"):

```
1 #!/bin/bash
2 opzione=$1
3 echo "Il personaggio indicato e' $opzione"
4 if [[ $opzione == pippo ]]; then
5   echo "Pippo e' il miglior amico di Topolino"
6 elif [[ $opzione == pluto ]]; then
```

```
7   echo "Pluto e' il cane di Topolino"
8 elif [[ $opzione == minnie ]]; then
9   echo "Minnie e' la fidanzata di Topolino"
10 else
11   echo "Questo personaggio non lo conosco!"
12 fi
```

Il parametro passato sulla riga di comando viene memorizzato nella variabile `opzione` (riga 2). Quindi vengono valutate in cascata alcune condizioni: ciascuna di queste (righe 4, 6 e 8) non è altro che il confronto del valore della variabile `opzione` con una stringa costante. Se nessuna delle tre condizioni è verificata, viene eseguito il comando che segue l'istruzione "`else`". Un esempio di esecuzione dello script è riportato di seguito:

```
$ ./chi.sh pluto
Il personaggio indicato e' pluto
Pluto e' il cane di Topolino
$ ./chi.sh paperino
Il personaggio indicato e' paperino
Questo personaggio non lo conosco!
```

Struttura di controllo "**case ... esac**"

In questi casi, ossia quando le condizioni da valutare in cascata sono costituite da semplici confronti tra il valore di una variabile ed un insieme di stringhe, l'istruzione "`case`" offre una struttura sintattica più semplice e chiara, appositamente disegnata per questo tipo di operazioni. Il blocco condizionale in questo caso si apre con l'istruzione "`case`" e si chiude con la parola chiave "`esac`"; la struttura delle istruzioni è la seguente:

```
1 case variabile in
2   ( pattern | pattern | ... | pattern )
3     istruzioni;;
4   ( pattern | pattern | ... | pattern )
5     istruzioni;;
6   ...
7 esac
```

Se il valore della variabile riportata dopo l'istruzione "`case`" corrisponde con uno dei pattern riportati tra parentesi tonde, viene eseguito il blocco di istruzioni corrispondenti. Ad esempio, lo script precedente, potrebbe essere riscritto come segue, senza alterarne minimamente il funzionamento:

```
1  #!/bin/bash
2  opzione=$1
3  echo "Il personaggio indicato e' $opzione"
4  case $opzione in
5    ( pippo )
6      echo "Pippo e' il miglior amico di Topolino";;
7    ( pluto )
8      echo "Pluto e' il cane di Topolino";;
9    ( minnie )
10     echo "Minnie e' la fidanzata di Topolino";;
11   ( * )
12     echo "Questo personaggio non lo conosco!"
13 esac
```

È utile osservare che il caso di *default*, quello che identifica il blocco di istruzioni che viene eseguito nel caso in cui nessuna delle condizioni sia risultata valida, è implemen-

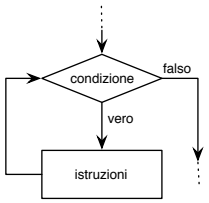

Figura 4.3: Diagramma di flusso della struttura di controllo iterativa implementata con l'istruzione "`while`"

tato riportando tra parentesi un pattern che coincide con qualsiasi stringa, costituito dall'espressione "`*`" (asterisco).

4.2 Strutture di controllo iterative

Per implementare algoritmi che svolgano un numero elevato di operazioni con uno script di poche righe di codice, spesso si ricorre alle strutture di controllo *iterative*. Tali istruzioni permettono di ripetere un certo blocco di comandi più volte, fino a quando non risulta verificata una determinata espressione logica codificata nell'istruzione di controllo della struttura iterativa.

Bash mette a disposizione tre istruzioni principali per la realizzazione di iterazioni (cicli): le istruzioni "`while`", "`until`" e "`for`". Dal punto di vista algoritmico, tutte e tre implementano una struttura in cui, all'inizio del blocco di istruzioni da ripetere, viene valutata una condizione logica (spesso il *return code* di un'istruzione) e, a valle di tale valutazione, viene eseguita un'iterazione del ciclo, ovvero viene terminata definitivamente l'esecuzione del ciclo stesso.

4.2.1 L'istruzione "while"

L'istruzione "`while`" esegue un blocco di istruzioni delimitato dalle parole chiave "`do`" e "`done`" se l'istruzione di controllo che segue l'istruzione "`while`" restituisce il valore "vero" (*exit status* 0). Dal punto di vista algoritmico una struttura iterativa rispetta il seguente schema generale, rappresentato anche dal diagramma di flusso riportato in Figura 4.3:

L'istruzione **while**

fintanto che *l'istruzione di controllo restituisce il valore* 0 *(vero)* **ripeti**
istruzioni
fine-ciclo

La sintassi dell'istruzione "`while`" è la seguente:

```
1 while istruzione di controllo
2 do
3   istruzioni
4 done
```

Viene valutato il return code dell'istruzione di controllo che segue il "`while`" e, se il valore restituito è *vero* (0) vengono eseguite le istruzioni delimitate dalle parole-chiave "`do`" e "`done`", quindi viene eseguita nuovamente l'istruzione di controllo e ne viene valutato il return code. Il ciclo termina quando la valutazione della condizione restituisce il valore *falso* (un valore numerico diverso da 0): in tal caso l'esecuzione dello script prosegue con la prima istruzione successiva alla parola chiave "`done`".

Spesso l'istruzione di controllo che regola l'iterazione del ciclo viene realizzata con un'espressione del tipo "`[` *condizione* `]`", che permette di valutare un'espressione lo-

test / [...]: valutazione di un'espressione logica

gica arbitraria utilizzando gli operatori di confronto visti nel Capitolo 2. Il simbolo "`[`" è un comando interno della Bash equivalente al comando "`test`", che permette di calcolare il valore "booleano" dell'espressione che segue, che termina con il carattere "`]`" (carattere che deve essere usato solo se si usa il comando "`[`" e non se si usa "`test`").

Dal punto di vista sintattico, precisiamo che deve essere presente uno spazio tra le parentesi quadrate e la condizione. Inoltre è possibile riportare la parola chiave "`do`" sulla stessa riga dell'istruzione "`while`", ma in tal caso è necessario inserire un punto e virgola tra la parentesi quadra chiusa e l'istruzione "`do`":

```
1 while [ condizione ]; do
2   istruzioni
3 done
```

Il seguente esempio (`multipli.sh`) stampa i primi k multipli dell'intero n: n e k sono due numeri riportati sulla linea di comando:

```
1  #!/bin/bash
2  # Visualizza i primi k multipli di n
3  n=$1
4  k=$2
5  echo "Ecco i primi $k multipli di $n:"
6  i=1
7  while [ $i -le $k ]; do
8    ((x=n*i))
9    echo -n "$x "
10   ((i++))
11 done
12 echo "Fatto!"
```

Con le istruzioni a riga 3 e 4 vengono assegnati il primo ed il secondo parametro riportati sulla linea di comando (rappresentati dalle variabili `$1` e `$2`) rispettivamente alle variabili `n` e `k`. Quindi, a riga 6, viene inizializzato il valore della variabile `i` con cui si terrà il conto del numero di multipli di `n` che sono stati calcolati. La condizione che regola l'esecuzione del ciclo mediante l'istruzione "`while`" (riga 7) confronta il valore della variabile `i` con quello della variabile `k`, richiedendo che il primo sia minore o uguale al secondo ("`-le`" *less or equal*, minore o uguale). Se la condizione risulta vera (se `i` è minore o uguale a `k`) allora vengono eseguite le istruzioni delimitate da "`do`" e "`done`". A riga 8 viene calcolato il valore dell'i-esimo multiplo di `n`, a riga 9 viene stampato e a riga 10 viene incrementato di uno il valore della variabile `i`. Quindi si torna a riga 7 per eseguire nuovamente il test della condizione e le istruzioni di calcolo del multiplo vengono ripetuto fino a quando il valore di `i` non supera quello di `k`. Solo allora verrà interrotta l'esecuzione del ciclo e lo script terminerà dopo aver eseguito l'istruzione a riga 12. Un esempio di output prodotto dallo script è riportato di seguito:

```
$ ./multipli.sh 3 5
Ecco i primi 5 multipli di 3:
3 6 9 12 15 Fatto!
```

break: interruzione incondizionata di un ciclo

È possibile interrompere "bruscamente" l'esecuzione di un ciclo anche utilizzando l'istruzione "`break`". Questo comando indica alla shell di interrompere l'esecuzione del ciclo entro cui si trova il comando stesso, proseguendo l'esecuzione dello script con la prima istruzione successiva al blocco "`do-done`". Il ciclo viene interrotto a prescindere dal valore di *vero* o *falso* della condizione riportata con l'istruzione "`while`". Di fatto si tratta di un "salto incondizionato" alla prima istruzione fuori dal ciclo e per questo motivo contravviene alle regole fondamentali della programmazione strutturata. Pertanto se

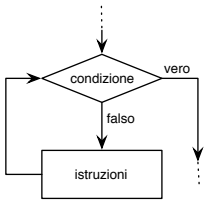

Figura 4.4: Diagramma di flusso della struttura di controllo iterativa implementata con l'istruzione "`until`"

ne sconsiglia l'uso, perché rende più confusi e meno manutenibili gli script shell. È pur vero, d'altra parte, che uno shell script è costituito in genere da poche righe di codice e quindi spesso la compattezza dello script è tale che la sua manutenibilità è comunque garantita.

4.2.2 L'istruzione "until"

L'istruzione "`until`" consente di implementare strutture algoritmiche iterative analoghe a quelle che è possibile realizzare con l'istruzione "`while`": la differenza tra le due istruzioni consiste nel fatto che la condizione riportata dopo la parola chiave "`until`", nel caso in cui risulti falsa produce l'esecuzione delle istruzioni delimitate dalle parole chiave "`do`" e "`done`", mentre, quando risulta vera, il ciclo ha termine e l'esecuzione dello script prosegue con la prima istruzione successiva alla parola chiave "`done`". Il comportamento dell'istruzione "`until`" può essere schematizzato con il diagramma di flusso riportato in Figura 4.4: è evidente, quindi, che la valutazione della condizione che controlla l'esecuzione del ciclo, ha un effetto opposto a quello che si ha con l'istruzione "`while`" (vedi Figura 4.3 a pagina 35).

L'istruzione **until**

La sintassi dell'istruzione "`until`" è del tutto analoga a quella dell'istruzione "`while`":

```
1 until istruzioni di controllo ; do
2   istruzioni
3 done
```

Anche in questo caso le parole chiave "`do`" e "`done`" delimitano le istruzioni che costituiscono il corpo del ciclo e che vengono eseguite se il return code dell'istruzione di controllo è diverso da zero (valore logico *falso*); dopo aver eseguito le istruzioni del blocco del ciclo, viene eseguita nuovamente l'istruzione di controllo: in base al valore del return code restituito vengono eseguite le istruzioni del blocco fino a quando la verifica della condizione non avrà valore *vero*.

La parola chiave "`do`" può essere riportata sulla riga successiva a quella dell'istruzione "`until`", oppure sulla stessa riga; in quest'ultimo caso deve essere usato un punto e virgola per separarla dall'istruzione che la precede.

Lo stesso script riportato nelle pagine precedenti, che esegue il calcolo dei primi k multipli di un intero n, può quindi essere riscritto utilizzando l'istruzione "`until`" invece dell'istruzione "`while`", come riportato di seguito. Naturalmente l'output prodotto dallo script è identico a quello prodotto dallo script precedente.

```
1 #!/bin/bash
2 # Visualizza i primi k multipli di n
3 n=$1
4 k=$2
5 echo "Ecco i primi $k multipli di $n:"
6 i=1
7 until [ $i -gt $k ]; do
```

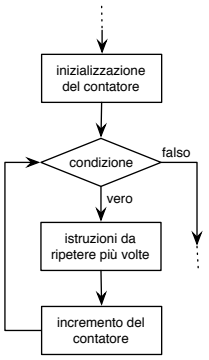

Figura 4.5: Diagramma di flusso della struttura di controllo iterativa basata su una variabile contatore, che è possibile implementare con l'istruzione "`for`"

```
8    ((x=n*i))
9    echo -n "$x "
10   ((i++))
11 done
12 echo "Fatto!"
```

La differenza tra i due script è concentrata nella riga 7 dove in un caso viene usata l'istruzione "`while`" e nell'altro viene usata l'istruzione "`until`"; per ottenere lo stesso risultato è stata quindi modificata la condizione riportata fra parentesi quadrate: nello script che utilizza l'istruzione "`until`" il ciclo prosegue fintanto che la condizione `i > k` risulta falsa; quando il valore di `i` raggiunge il valore di `k`, la condizione diventa vera, il ciclo termina e lo script prosegue con l'esecuzione dell'istruzione a riga 12.

Spesso, come negli esempi precedenti, l'iterazione del ciclo è controllata mediante una variabile che "tiene il conto" del numero di iterazioni effettuate: la variabile in questione, assume un valore iniziale e, prima dell'esecuzione di ogni iterazione, viene controllato che il valore della variabile stessa non abbia superato una soglia prefissata; al termine dell'esecuzione del blocco di istruzioni che costituiscono il corpo del ciclo, prima di ripetere il controllo sul valore della variabile "contatore", viene eseguito l'incremento del contatore stesso.

Queste operazioni possono essere implementate utilizzando le istruzioni "`while`" e "`until`", come effettivamente avviene nei due esempi precedenti. In entrambi gli script, infatti, il ciclo è controllato dalla variabile contatore `i`: a riga 6 viene inizializzato il valore del contatore (`i=1`), a riga 7 viene confrontato il valore di `i` con quello della soglia `k` e, al termine del blocco di istruzioni da ripetere, a riga 10 viene incrementata la variabile `i`. Il diagramma di flusso riportato in Figura 4.5 schematizza la struttura di un ciclo di questo genere.

4.2.3 L'istruzione "for"

L'istruzione **for**

L'istruzione "`for`" consente in modo molto diretto di implementare una struttura di controllo iterativa di questo tipo, basata su un contatore. Una delle forme con cui può essere utilizzata l'istruzione "`for`" è derivata direttamente dal linguaggio C e prevede tre parametri riportati fra doppie parentesi tonde e separati da un punto e virgola:

```
1 for (( espr1 ; espr2 ; espr3 ))
2 do
3   istruzioni
4 done
```

Parametri dell'istruzione **for**

La prima espressione (`espr1`) generalmente è l'istruzione con cui viene inizializzata una variabile contatore; la seconda espressione (`espr2`) è la condizione con cui si verifica se il contatore ha raggiunto o superato la soglia: se la condizione restituisce il valore *vero* viene eseguito il blocco di istruzioni delimitate dalle parole chiave "`do`" e "`done`", altrimenti il ciclo termina e lo script prosegue con l'istruzione immediatamente successiva al "`done`"; la terza espressione (`espr3`), infine, è l'istruzione con cui viene incrementato (o decrementato) il valore del contatore.

Di seguito riportiamo l'esempio del calcolo dei primi k multipli di n, implementato con l'istruzione "`for`".

```
1  #!/bin/bash
2  # Visualizza i primi k multipli di n
3  n=$1
4  k=$2
5  echo "Ecco i primi $k multipli di $n:"
6  for (( i=1 ; i <= k; i++ ))
7  do
8    ((x=n*i))
9    echo -n "$x "
10 done
11 echo "Fatto!"
```

Il valore del contatore può essere modificato a piacere ad ogni iterazione del ciclo, con una qualsiasi istruzione di assegnazione di un valore alla variabile contatore utilizzata al posto di `espr3`. Nell'esempio di seguito viene eseguito il "conto alla rovescia", che visualizza i numeri $10, 9, 8, \ldots, 3, 2, 1$:

```
1  #!/bin/bash
2  # Conto alla rovescia
3  for (( i=10 ; i > 0; i-- ))
4  do
5    echo -n "$i "
6    sleep 1
7  done
8  echo "Bum!"
```

sleep: introduce una pausa di n secondi

Il contatore viene inizializzato con il valore 10, quindi viene eseguito il blocco di istruzioni delimitate da "`do-done`" fintanto che il valore della variabile `i` è maggiore di zero; alla fine dell'esecuzione di ogni blocco di istruzioni viene decrementato di uno il valore del contatore (`i--`). Il comando esterno "`sleep`" (riga 6) introduce una pausa pari al numero di secondi indicati come parametro (un secondo nell'esempio precedente).

Forma generale dell'istruzione **for**

In generale l'istruzione "`for`" consente di ripetere un blocco istruzioni al variare di una variabile di controllo (il *contatore* negli esempi precedenti) in una lista di valori o di un intervallo numerico riportati come argomento dell'istruzione "`for`". La lista di valori può essere espressa in forma "statica", riportando i valori uno di seguito all'altro, separati da spazi, oppure come output di un altro comando o, infine, come insieme di numeri interi descritto mediante gli estremi dell'insieme stesso. In tutti e tre i casi l'istruzione "`for`" viene integrata dalla parola chiave "`in`" che separa la variabile dall'elenco dei valori che questa deve assumere:

```
1  for variabile in lista-di-valori
2  do
3    istruzioni
4  done
```

Nell'esempio seguente viene fornito un insieme di valori "statici" per la variabile, riportati direttamente nel codice sorgente dello script:

```
#!/bin/bash
# Stampa un elenco di animali
for animale in cane gatto 'orso bianco' topo
do
  echo $animale
done
```

Ad ogni iterazione del ciclo la variabile a assume uno dei valori riportati nell'elenco che segue la parola chiave "`in`". Quando non ci sono altri valori da assegnare alla variabile, il ciclo termina.

Lista dei valori per l'istruzione **for** generata da un comando

I valori della lista possono anche essere generati da un comando, come nell'esempio seguente in cui si suppone che il file "`animali.txt`" contenga i nomi di un insieme di animali, uno per ogni riga del file:

```
#!/bin/bash
# Stampa un elenco di animali, seconda versione
echo "Un elenco di animali, in ordine alfabetico:"
for animale in $( sort animali.txt )
do
  echo $animale
done
```

IFS: la variabile speciale in cui è memorizzato il separatore di lista

La variabile speciale `IFS` contiene il carattere utilizzato dalla Bash come separatore degli elementi di una lista. Di *default* il separatore degli elementi di una lista è un qualsiasi simbolo di spaziatura (lo spazio, una tabulazione o un carattere di *new line*), ma, se occorre, è possibile ridefinirlo impostando un valore arbitrario per la variabile `IFS`.

{a..b}

Con Bash versione 3.0 e le successive, nel caso in cui la lista di valori da assegnare alla variabile che controlla l'esecuzione del ciclo sia un insieme di numeri interi, in ordine crescente, allora l'elenco può essere espresso utilizzando le parentesi graffe per delimitare gli estremi dell'intervallo e specificando il valore minimo e il valore massimo, separati da una coppia di punti: {*min*`..`*max*}. In questo modo la variabile assumerà tutti i valori dell'insieme $\{min, min+1, min+2, \ldots, max\}$. Ad esempio l'espressione "`{4..7}`" genera la sequenza di numeri interi $4, 5, 6, 7$.

{a..b..c}

A partire dalla versione 4.0 di Bash, si può anche esprimere l'incremento della variabile, nel caso in cui non si desideri usare tutti i valori interi dell'intervallo; in tal caso si può esprimere il valore dell'incremento di seguito al valore *max*, separandolo con una coppia di punti: {*min*`..`*max*`..`*inc*}. In questo caso la variabile assumerà la seguente successione di valori: $min, min+inc, min+2inc, min+3inc, \ldots$ fino a quando non sarà superato il valore *max*. L'espressione "`{1..15..3}`", ad esempio, genera la sequenza di valori $1, 4, 7, 10, 13$.

Il seguente script (che funziona solo con Bash versione 4.0 o successive) visualizza i primi 10 multipli (la *tabellina*) del numero intero riportato come argomento sulla linea di comando:

```
#!/bin/bash
# Tabellina del ...
n=$1
((m=n*10))
echo "Tabellina del $n:"
for x in {$n..$m..$n}; do
  echo -n "$x "
done
echo "Fine!"
```

Sui sistemi dotati di versioni precedenti della Bash può essere presente il comando esterno "seq" che consente di generare sequenze di numeri interi. Il comando "seq" accetta uno, due o tre parametri: con un solo parametro n visualizza in output la sequenza di numeri $1,2,\ldots,n$; se i parametri sono invece due, ad esempio n ed m, viene generata la sequenza $n, n+1, n+2, \ldots, m$. Infine, se vengono forniti tre parametri n, k ed m, viene prodotta la sequenza $n, n+k, n+2k, n+3k, \ldots$ fino a superare il valore di soglia m.

seq: generatore di sequenze di numeri interi

Utilizzando il comando "seq" possiamo riscrivere come segue lo script precedente che stampa la "tabellina" del numero fornito come argomento sulla linea di comando:

```
1  #!/bin/bash
2  # Tabellina del ... con il comando seq
3  n=$1
4  ((m=n*10))
5  echo "Tabellina del $n:"
6  for x in $( seq $n $n $m )
7  do
8    echo -n "$x "
9  done
10 echo "Fine!"
```

L'output prodotto da quest'ultimo script è identico a quello generato dallo script precedente, ma, se possibile, si consiglia di evitare il ricorso al comando esterno "seq", che rende meno efficiente lo script.

```
$ ./tabellina.sh 7
Tabellina del 7:
7 14 21 28 35 42 49 56 63 70 Fine!
```

4.2.4 L'istruzione "select"

L'ultima istruzione per l'implementazione di strutture di controllo iterative è di carattere meno generale di quelle viste nelle pagine precedenti. L'istruzione "select" consente di visualizzare un menù di scelte numerate (da 1 a n) che l'utente può selezionare interattivamente digitando il numero corrispondente all'opzione desiderata. Dopo aver eseguito le istruzioni contenute nel blocco delimitato dalle parole chiave "do-done", viene nuovamente presentato il prompt e la shell attende che l'utente compia una nuova scelta. Se l'utente batte il tasto Invio/Enter senza digitare un numero corrispondente ad un'opzione, viene visualizzato nuovamente l'intero menù di scelte. Il ciclo viene ripetuto continuamente, fino a quando non viene interrotto mediante l'istruzione "break".

select: implementa menù di scelta interattivi

In pratica l'istruzione "select" implementa una procedura che può essere schematizzata nei seguenti passi:

1: visualizza su *standard output* le opzioni del menù, numerandole da 1 a n;
2: visualizza un *prompt* e attende che l'utente digiti un numero;
3: se l'utente ha selezionato una voce del menù allora esegue le istruzioni delimitate da "do-done";
4: altrimenti visualizza nuovamente il menù;
5: ritorna al passo 2.

L'istruzione "select" fa uso di alcune variabili speciali per la gestione dell'input/output: la variabile PS3 contiene la stringa utilizzata come *prompt* per acquisire la scelta dell'utente; la variabile REPLY contiene il valore numerico dell'opzione di menù selezionata dall'utente.

Le variabili **PS3** e **REPLY**

Il seguente esempio aiuta a chiarire il funzionamento dell'istruzione "select".

```
1  #!/bin/bash
2  PS3='--> '
3  echo "Menu' principale"
4  select s in Primo Secondo Quit; do
5    echo "Voce di menu' n. $REPLY"
6    case $s in
7      ( Primo )
8        echo "Hai selezionato la prima opzione!";;
9      ( Secondo )
10       echo "Hai selezionato la seconda opzione!";;
11     ( Quit )
12       echo "Hai selezionato la terza opzione!"
13       break;;
14     ( * )
15       echo "Questa opzione non e' prevista.";;
16   esac
17   echo "Hai scelto '$s'!";
18 done
19 echo "Termine dello script"
```

Lo script visualizza un menù con tre opzioni descritte dalle voci della lista ("`Primo`", "`Secondo`" e "`Quit`") riportata di seguito all'istruzione "`select`"; ciascuna voce corrisponde ad un numero intero positivo 1, 2, 3, ... in base all'ordine con cui compare l'elemento nella lista. Ad ogni iterazione del ciclo la Bash visualizza il prompt definito impostando la variabile `PS3` a riga 2. Il menù di opzioni viene visualizzato prima del prompt la prima volta che viene eseguito il ciclo ed ogni volta che l'utente effettuerà una scelta nulla, battendo semplicemente il tasto Invio/Enter senza digitare alcuna opzione.

Nell'esempio riportato nello script precedente, selezionando una delle prime due voci del menù viene semplicemente visualizzato un messaggio; selezionando la terza opzione (Quit) viene visualizzato lo stesso messaggio, ma poi viene anche interrotto il ciclo di gestione del menù, con l'istruzione "`break`" presente a riga 13. Selezionando un'opzione non gestita dall'istruzione "`case`", viene eseguito il blocco di istruzioni di default a riga 15. Un esempio di output prodotto dall'esecuzione dello script precedente è riportato di seguito:

```
$ ./menu.sh
Menu' principale
1) Primo
2) Secondo
3) Quit
--> 1
Voce di menu' n. 1
Hai selezionato la prima opzione!
Hai scelto 'Primo'!
--> 5
Voce di menu' n. 5
Questa opzione non e' prevista.
Hai scelto ''!
-->
1) Primo
2) Secondo
3) Quit
--> 3
```

```
Voce di menu' n. 3
Hai selezionato la terza opzione!
Termine dello script
```

Dall'esempio si può vedere come il valore assunto dalla variabile `s`, utilizzata a riga 4 dello script nell'istruzione "`select`", corrisponda all'elemento della lista di opzioni scelto dall'utente digitando il numero della voce di menù corrispondente, che infatti viene visualizzato con l'istruzione a riga 17 che utilizza la variabile `s`; viceversa la variabile `REPLY` assume come valore proprio il numero dell'opzione selezionata dall'utente, visualizzato con l'istruzione a riga 5 che utilizza proprio la variabile `REPLY`.

Capitolo 5

Funzioni

Spesso, progettando uno script complesso, è utile suddividerlo in sotto-programmi che svolgono un compito più circoscritto e che possono anche essere riutilizzati in altri script. Spesso anzi è utile raccogliere in uno o più file dei sotto-programmi di utilità comune, in modo tale da comporre una libreria con cui si possono sviluppare nuovi script più rapidamente, senza dover ogni volta riscrivere anche le procedure già utilizzate in altri casi.

La Bash ci permette di implementare questa prassi comune a molti altri linguaggi di programmazione attraverso la definizione di *funzioni*. Come avviene in linguaggio C, una funzione non è altro che un "sotto-programma", identificato da un nome. Ad una funzione, come vedremo tra breve, è possibile passare una lista di parametri e la funzione può restituire un valore numerico intero.

5.1 Definzione di una funzione

function: definizione di una funzione

Una funzione viene definita utilizzando l'istruzione "`function`" seguita dal nome identificativo della funzione stessa. Il nome della funzione deve essere differente dalle parole chiave del linguaggio Bash e da ogni altro nome di funzione e di variabile definito nell'ambito dello stesso script. Il "corpo" della funzione, le istruzioni del sottoprogramma che ne definiscono il comportamento, sono delimitate da una coppia di parentesi graffe:

```
1 function saluta {
2   echo "Ciao!"
3 }
```

La funzione può essere dichiarata anche omettendo l'istruzione "`function`", ma in tal caso il nome della funzione deve essere seguito da una coppia di parentesi tonde:

```
1 saluta() {
2   echo "Ciao!"
3 }
```

Le istruzioni presenti nel corpo di una funzione vengono eseguite solo quando la funzione viene richiamata. La funzione viene richiamata dallo script Bash in cui è stata definita, semplicemente utilizzandone il nome, come se si trattasse di un qualsiasi altro comando interno della shell:

```
1 #!/bin/bash
2 function saluta {
3   echo -n "Ciao! "
4 }
```

```
5 for ((i=0; i<3; i++)); do
6   saluta
7 done
```

Nelle righe 2, 3 e 4 viene definita la funzione "saluta" che visualizza la stringa "Ciao!" su standard output. Nelle righe 5, 6 e 7 viene definito un ciclo che esegue tre iterazioni utilizzando l'istruzione "for"; ad ogni iterazione del ciclo, con l'istruzione a riga 6 viene richiamata la funzione "saluta". L'output di questo banale esempio ("saluti.sh") è il seguente:

```
$ ./saluti.sh
Ciao! Ciao! Ciao!
```

Lo script deve essere scritto in modo tale che la funzione venga definita prima di essere richiamata da un'istruzione dello stesso script: la definizione della funzione deve sempre precedere la prima delle istruzioni in cui la funzione viene invocata.

Naturalmente in uno stesso script possono essere definite più funzioni, purché siano identificate da nomi differenti. Una funzione può essere anche invocata da un'altra funzione; in questo caso l'ordine con cui sono definite le funzioni non ha nessuna importanza (è possibile definire la funzione "chiamante" prima della funzione "chiamata").

```
1  #!/bin/bash
2  function prima {
3    echo "Prima funzione."
4    seconda
5    echo "Ancora la prima funzione."
6  }
7  function seconda {
8    echo "Seconda funzione."
9  }
10 echo "Inizio."
11 prima
```

Nell'esempio precedente lo script ("funzioni.sh") invoca la funzione "prima" (riga 11), che a sua volta richiama la funzione "seconda"; l'output prodotto è il seguente:

```
$ ./funzioni.sh
Inizio.
Prima funzione.
Seconda funzione.
Ancora la prima funzione.
```

local: definizione di variabili locali ad una funzione

Tutte le funzioni definite nell'ambito di uno script condividono fra loro le variabili, tranne le variabili che sono state definite nell'ambito di una funzione utilizzando l'istruzione "local". Questa istruzione restringe lo *scope*, ossia la visibilità di una variabile, alla sola funzione in cui la variabile viene definita; le altre funzioni potranno definire variabili differenti con lo stesso nome, ma non potranno accedere al contenuto di una variabile locale di un'altra funzione.

```
1  #!/bin/bash
2  function pippo {
3    local a
4    a=2
5    echo "Pippo: a = $a"
6  }
```

```
7
8  function pluto {
9    echo "Pluto: a = $a"
10   a=3
11   echo "Pluto: a = $a"
12 }
13
14 a=1
15 echo "a = $a"
16 pippo
17 echo "a = $a"
18 pluto
19 echo "a = $a"
```

L'esempio precedente mette in evidenza alcuni aspetti relativi allo *scope* delle variabili di uno script Bash. Lo script presenta due funzioni ("`pippo`" e "`pluto`"): nella prima viene definita una variabile locale chiamata `a` (riga 4) e gli viene assegnato il valore 2 (riga 5). Anche nel "corpo principale" dello script viene definita una variabile chiamata `a` assegnandogli il valore 1 (riga 15). Le due variabili sono differenti pur essendo identificate dallo stesso nome. Quest'ultima, quella definita nel corpo principale dello script, è visibile da tutte le funzioni dello script, tranne che nella funzione "`pippo`", in cui è stata definita una variabile locale con lo stesso nome (riga 4); questa variabile è visibile solo nella funzione "`pippo`".

La modifica del valore della variabile `a` ha impatto sulla variabile locale, con l'istruzione a riga 5, mentre ha effetto sulla variabile "globale" con le istruzioni a riga 11 e 15. Di seguito riportiamo l'output dell'esecuzione dello script:

```
$ ./varLocali.sh
a = 1
Pippo: a = 2
a = 1
Pluto: a = 1
Pluto: a = 3
a = 3
```

5.2 Passaggio dei parametri ad una funzione

È possibile invocare una funzione riportando uno o più parametri sulla stessa riga, come argomento della funzione stessa. La funzione riceve i parametri nelle variabili `$1`, `$2`, ecc. secondo l'ordine con cui i parametri stessi sono elencati dopo il nome della funzione. Come per i parametri passati sulla linea di comando allo shell script, anche nel caso dei parametri passati ad una funzione si può utilizzare l'istruzione "`shift`" per eliminare dalla coda dei parametri il primo elemento.

$1, $2, ... variabili che contengono i parametri passati come argomento della funzione

```
1 #!/bin/bash
2 function saluta {
3   while [ $1 ]; do
4     echo "Ciao $1"
5     shift
6   done
7 }
8
9 saluta 'Marina' 'Chiara' 'Elena'
```

L'output della funzione precedente è il seguente:

```
$ ./ciao.sh
Ciao Marina
Ciao Chiara
Ciao Elena
```

return: restituisce un valore numerico intero

La funzione può anche restituire un valore numerico intero utilizzando l'istruzione "return". Il valore restituito da una funzione mediante l'istruzione return viene memorizzato nella variabile "$?". Ad esempio nel seguente script si fa uso della funzione "somma" per calcolare la somma di tutti gli elementi passati come argomento della funzione.

```
1  #!/bin/bash
2  function somma {
3    local s=0
4    while [ $1 ]; do
5      ((s=s+$1))
6      shift
7    done
8    return $s
9  }
10
11 somma ${BASH_ARGV[*]}
12 echo "Somma = $?"
```

I numeri di cui si intende calcolare la somma vengono forniti allo script sulla riga di comando; in questo modo, utilizzando il vettore BASH_ARGV, a riga 11 viene invocata la funzione "somma" passandogli come argomento tutto il vettore. La funzione esegue un ciclo (righe 4–7) e accumula nella variabile locale s la somma degli elementi passati come argomento[1]; quindi restituisce il valore finale della sommatoria utilizzando l'istruzione return (riga 8). Il valore restituito dalla funzione viene stampato utilizzando la variabile "$?" (riga 12).

```
$ ./sommatoria 10 20 30
Somma = 60
```

Uso del comando **echo** per la restituzione di un valore di una funzione

Esiste un modo meno canonico, ma ugualmente efficace, per fare in modo che le funzioni restituiscano qualsiasi tipo di dato, non solo numerico intero. Il metodo è valido se la funzione non produce nessun output su *standard output*. In questo caso, la funzione può utilizzare il comando "echo" per produrre su standard output il valore da restituire. La chiamata della funzione avverrà utilizzando la seguente sintassi:

variabile=$(*funzione argomento*)

In questo modo la stringa prodotta in output mediante il comando "echo" dalla funzione, sarà assegnato alla variabile. Ad esempio lo script precedente che calcola la somma dei numeri forniti sulla *command line*, può essere riscritto come segue:

```
1  #!/bin/bash
2  function somma {
3    local s=0
```

[1] A riga 5, sebbene l'espressione di somma sia delimitata dalle doppie parentesi tonde, per fare riferimento alla variabile speciale $1 è necessario utilizzare comunque il simbolo "$" perché la variabile ha un nome che altrimenti sarebbe impossibile distinguere dal valore numerico 1.

```
4   while [ $1 ]; do
5     ((s=s+$1))
6     shift
7   done
8   echo $s
9 }
10
11 sum=$(somma ${BASH_ARGV[*]})
12 echo "Somma = $sum"
```

Con l'istruzione di riga 11 alla variabile `sum` viene assegnato esplicitamente il valore restituito (mediante il comando "`echo`", non "`return`") dalla funzione `somma`, ottenendo una sintassi simile a quella di molti altri linguaggi di programmazione come, ad esempio, il C, il Pascal o altri ancora.

5.3 Funzioni ricorsive

Il linguaggio Bash supporta le "funzioni ricorsive", ossia funzioni definite in modo tale che nel corpo della funzione venga richiamata la funzione stessa.

Un esempio classico è dato dalla funzione *fattoriale*. In matematica, per denotare il "fattoriale di n" si usa il simbolo "$n!$" e con tale notazione si indica il prodotto dei primi n numeri naturali: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$. Ad esempio $4! = 4 \cdot 3 \cdot 2 \cdot 1$; osservando con attenzione questo esempio elementare ci accorgiamo che $4! = 4 \cdot 3!$ (ovviamente $3! = 3 \cdot 2 \cdot 1$).

Infatti è possibile definire la stessa funzione anche in un altro modo, utilizzando una tecnica "induttiva" basata sulla seguente espressione:

$$n! = \begin{cases} n \cdot (n-1)! & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

Il caso $n = 1$ è facile e costituisce la cosiddetta "base del procedimento ricorsivo": se $n = 1$ allora $n! = 1$. Se invece $n > 1$ allora possiamo calcolare $n!$ come prodotto tra n ed il valore di $(n-1)!$: $n! = n \cdot (n-1)!$ Sfruttando la "ricorsione" applicando più volte la definizione di $n!$ su numeri sempre più piccoli, alla fine arriveremo a calcolare il fattoriale di n. Ad esempio $4! = 4 \cdot 3! = 4 \cdot (3 \cdot 2!) = 4 \cdot (3 \cdot (2 \cdot 1)) = 4 \cdot (3 \cdot 2) = 4 \cdot 6 = 24$.

Utilizzando il linguaggio Bash possiamo scrivere il seguente script che utilizza la funzione "`fattoriale`" definita in modo ricorsivo, per calcolare il fattoriale del numero intero positivo passato sulla linea di comando:

```
1  #!/bin/bash
2  # Calcola il fattoriale del numero passato sulla linea di comando
3  # utilizzando un algoritmo ricorsivo
4
5  function fattoriale {
6    if [ $1 -gt 1 ]; then
7      ((a=$1-1))
8      fattoriale $a
9      ((f=$1*$?))
10   else
11     f=1
12   fi
13   return $f
14 }
15
```

```
16 fattoriale $1
17 echo "$1! = $?"
```

A riga 8 la funzione "`fattoriale`" richiama se stessa passando come parametro un argomento diminuito di un'unità: questo è il cuore del procedimento ricorsivo.

Eseguendo lo script si ottiene il risultato riportato di seguito; è bene osservare che le capacità di calcolo numerico della Bash sono limitate, per cui anche con numeri piccoli ($n > 5$) il calcolo del fattoriale "sballa" fornendo risultati non esatti.

```
$ ./fattoriale.sh 5
5! = 120
```

5.4 Librerie di funzioni

Può essere molto utile raccogliere in uno o più file un insieme di funzioni di uso comune, in modo tale da comporre una "libreria" riutilizzabile in più script. Con Bash una libreria non è altro che un file in cui sono riportate delle funzioni, anche del tutto indipendenti fra di loro; nel file non devono però essere presenti istruzioni esterne al corpo delle funzioni.

source: importa nello script le istruzioni contenute in un file

Per poter usare la libreria, importando le funzioni che vi sono state definite in uno shell script, si deve utilizzare il comando "`source`", che consente di importare in uno script le istruzioni (le definizioni delle funzioni, nel nostro caso) contenute in un altro file.

Ad esempio supponiamo di raccogliere nel file "`libreria.sh`" le funzioni "`multipli`", "`somma`" e "`prodotto`" definite di seguito:

```
1  # Libreria di funzioni per Bash
2  function multipli {
3    local n=$1
4    local k=$2
5    echo "I primi $k multipli di $n:"
6    local i=1
7    while [ $i -le $k ]; do
8      ((x=n*i))
9      echo -n "$x "
10     ((i++))
11   done
12   echo
13 }
14
15 function somma {
16   local s=0
17   while [ $1 ]; do
18     ((s=s+$1))
19     shift
20   done
21   return $s
22 }
23
24 function prodotto {
25   local p=1
26   while [ $1 ]; do
27     ((p=p*$1))
28     shift
```

```
29   done
30   return $p
31 }
```

Nello script "sommaProdotto.sh" costruiamo una procedura che utilizza alcune delle funzioni definire nella libreria. Naturalmente prima di poter usare tali funzioni è necessario "caricare" la libreria utilizzando il comando "source" (riga 5).

```
1 #!/bin/bash
2 # sommaProdotto.sh
3 # Calcola la somma e il prodotto dei numeri passati come argomento
4 # sulla command line
5 source "./libreria.sh"
6 somma ${BASH_ARGV[*]}
7 echo "Somma = $?"
8 prodotto ${BASH_ARGV[*]}
9 echo "Prodotto = $?"
```

Una sintassi alternativa al comando "source" è costituita dal carattere "." (punto). La riga 5 dello script precedente potrebbe quindi essere riscritta come segue:

```
. "./libreria.sh"
```

Il comando "source" (e il comando ".") non si limita a caricare le istruzioni contenute nel file: se sono presenti delle istruzioni esterne alle funzioni, le esegue.

Capitolo 6

Esempi

In quest'ultimo capitolo sono riportati alcuni script che implementano degli esempi più significativi di quelli riportati nelle pagine precedenti per descrivere singole istruzioni del linguaggio. Questi script non sono puramente esemplificativi, ma affrontano e risolvono problemi di una qualche utilità e consentono di vedere "in azione" il linguaggio di scripting su casi di più ampio respiro.

6.1 Rotazione di file di log

Il primo esempio consiste in un tipico task di amministrazione di un sistema operativo UNIX/Linux, ossia la cosiddetta "rotazione" dei file di log di un'applicazione. Per semplificare le operazioni di archiviazione e di gestione dei file contenenti le informazioni di log sul funzionamento e sugli errori di un determinato servizio, spesso si fa in modo di conservare su file distinti i log relativi a giorni differenti: in questo modo ogni file conterrà solo le informazioni relative ad un periodo di tempo di 24 ore e dunque avrà una dimensione contenuta, che è possibile archiviare o gestire con maggiore facilità rispetto ad un unico file di log molto ingombrante. Siccome spesso i file di log sono utili per determinare qualche malfunzionamento o comportamento anomalo da parte delle applicazioni, è possibile che non sia opportuno conservare file di log molto vecchi, ma che sia sufficiente mantenere archiviati (magari in formato compresso) solo quelli relativi agli ultimi giorni.

L'operazione che consente di mantenere un archivio aggiornato solo con gli ultimi file di log, eliminando i precedenti, si chiama in gergo "rotazione dei file di log". Ciascun file viene infatti identificato da un numero progressivo (da 0 ad n, dove $n+1$ è il numero di file di log che si intende conservare) nel nome del file stesso e, ad ogni intervallo di rotazione (ad esempio, pianificando l'esecuzione automatica di questa operazione ogni 24 ore, utilizzando utility come "`crond`"), ciascun file viene rinominato utilizzando il progressivo incrementato di 1; il file con il progressivo n viene eliminato, sovrapponendolo con il file che aveva il progressivo $n-1$. Il file di log corrente, infine, viene rinominato utilizzando il progressivo 0 e si crea un nuovo file di log completamente vuoto, privo di progressivo.

Ad esempio, se l'applicazione "`myapp`" produce un file di log in "`/tmp/mylog`", allora applicare l'operazione di *log rotation* a quel file, significa spostare ogni volta il log "`mylog`" nel file "`mylog.0`" e creare un nuovo file vuoto denominato "`mylog`". Per mantenere in archivio i file relativi alle ultime n turnazioni, prima di rinominare il log corrente si provvederà a rinominare/spostare "`mylog.`$n-1$" in "`mylog.`n", "`mylog.`$n-2$" in "`mylog.`$n-1$" e così via, fino a spostare "`mylog.0`" in "`mylog.1`".

Lo script riportato di seguito esegue proprio questa operazione, avendo cura, in aggiunta, di comprimere i file di log archiviati (non quello corrente) in modo da risparmiare spazio sul filesystem.

```
#!/bin/bash
# logrotate.sh
# Rinomina tutti i log file presenti nella directory $LOGDIR
# il cui nome corrisponde con $LOGNAME; viene eliminato il file
# piu' vecchio (ultimo della coda), creando un nuovo file vuoto
# all'inizio della coda.
#
# Variabili di configurazione dello script
#
LOGDIR='/tmp'
LOGNAME='mylog'
LOGRETENTION=7
GZIP='/usr/bin/gzip'
#
# Verifiche sulla corretta accessibilita' della directory dei log
#
if [[ ! -d "$LOGDIR" ]] ; then
  echo "$0: ERRORE: la directory $LOGDIR non esiste."
  exit 1
fi
if [[ ! -w "$LOGDIR" ]] ; then
  echo "$0: ERRORE: non e' possibile modificare il contenuto della \
    directory $LOGDIR."
  exit 1
fi
#
# Eliminazione del log piu' vecchio e spostamento dei log file
# piu' recenti
#
for (( i=LOGRETENTION ; i>0; i-- )); do
  ((j=i-1))
  if [[ -e $LOGDIR/$LOGNAME.$j.gz ]] ; then
    if [[ -w $LOGDIR/$LOGNAME.$j.gz && \
      (-w $LOGDIR/$LOGNAME.$i.gz || ! -e $LOGDIR/$LOGNAME.$i.gz) ]]
    then
      mv $LOGDIR/$LOGNAME.$j.gz $LOGDIR/$LOGNAME.$i.gz
    else
      echo "$0: ERRORE: non e' possibile ruotare \
        $LOGDIR/$LOGNAME.$j.gz in $LOGDIR/$LOGNAME.$i.gz"
      exit 1
    fi
  fi
done
#
# Spostamento e compressione del log piu' recente e creazione di un
# nuovo log file vuoto
#
mv $LOGDIR/$LOGNAME $LOGDIR/$LOGNAME.0
touch $LOGDIR/$LOGNAME
$GZIP $LOGDIR/$LOGNAME.0
```

Le istruzioni iniziali definiscono alcune variabili che serviranno in seguito a configurare il funzionamento dello script (righe 10–13): la variabile `LOGDIR` contiene il path della directory in cui sono contenuti i file di log (nell'esempio "`/tmp`", spesso nella pratica invece potrebbe essere "`/var/log`"), la variabile `LOGNAME` contiene il nome del file di log ("`mylog`" nel nostro esempio), la variabile `LOGRETENTION` contiene invece il massimo valore numerico progressivo da attribuire ai file di log archiviati: nell'esempio il valore è 7 e quindi saranno conservati otto file di archivio, da "`mylog.0`" fino a "`mylog.7`". La variabile "`GZIP`" contiene infine il path assoluto del comando esterno "`gzip`", una utility di GNU che consente di comprimere efficientemente dei file sul filesystem della macchina.

gzip: una utility per comprimere i file

Le variabili di configurazione sono denominate con caratteri maiuscoli proprio per evidenziarle rispetto ad altre variabili che saranno utilizzate nell'ambito dello script. Inoltre la definizione dei valori di queste variabili è raccolta nelle prime righe dello script, in modo da semplificare la modifica della configurazione da parte di coloro che vorranno utilizzarlo sul proprio sistema. Entrambi questi accorgimenti costituiscono delle "buone pratiche" che è bene rispettare nella programmazione degli shell script.

Con le condizioni alle righe 17 e 21 si verifica, rispettivamente, se esiste la directory il cui path è specificato in `LOGDIR` e se l'utente che sta eseguendo lo script ha il permesso di scrivere in tale directory. Nel caso in cui una delle due condizioni risulti falsa, viene visualizzato un messaggio di errore sul terminale dell'utente e viene interrotta l'esecuzione dello script con l'istruzione "`exit`", restituendo un return code diverso da zero per indicare la conclusione con una condizione di errore. In entrambe le istruzioni "`if`" la condizione è delimitata dalle doppie parentesi quadre "`[[...]]`", necessarie per poter utilizzare gli operatori "`-d`" e "`-w`" che eseguono verifiche sullo stato del filesystem.

Il ciclo "`for`" alle righe 30–43 fa variare il valore della variabile `i` da `LOGRETENTION` (7 nell'esempio) a 1, diminuendo di un'unità ad ogni iterazione il valore di `i`; alla variabile `j` viene assegnato il valore `i`−1. Se esiste il file "`$LOGDIR/$LOGNAME.$j.gz`" (es.: "`/tmp/mylog.5.gz`") e se è possibile modificare con i permessi assegnati all'utente che esegue lo script il file "`$LOGDIR/$LOGNAME.$i.gz`" o se questo file non esiste (riga 34), viene spostato il file "`$LOGNAME.$j.gz`" sul file "`$LOGNAME.$i.gz`"; altrimenti lo script termina con un messaggio di errore.

Dopo aver "ruotato" tutti i file di log archiviati, con le istruzioni riportate alle righe 48–50 lo script archivia il file di log corrente con il progressivo 0 e lo comprime con il programma "`gzip`". Viene anche ricreato un file di log corrente vuoto, utilizzando il comando esterno "`touch`". Il programma "`touch`" crea un file vuoto, se il nome specificato sulla linea di comando identifica un file che non esiste, altrimenti, se il file esiste, vengono aggiornate la data e l'ora di ultima modifica; in particolare nello script precedente, il comando "`touch`" crea certamente un nuovo file vuoto, dal momento che a riga 48 con l'istruzione "`mv`" un file con lo stesso nome era stato rinominato con un nome differente.

touch: crea un file vuoto o modifica la data e l'ora di ultima modifica di un file esistente

6.2 Rubrica degli indirizzi

In questa sezione vogliamo dimostrare l'utilizzo del linguaggio Bash per la realizzazione di una semplice *utility* per la gestione di un archivio di nomi, indirizzi e numeri di telefono. Vogliamo realizzare un programma che operi con una doppia modalità, batch ed interattiva, come spesso avviene per diversi programmi in ambiente UNIX/Linux: il programma deve accettare delle opzioni sulla linea di comando che gli indicano l'operazione da compiere in modalità batch; se non è presente alcuna opzione il programma opererà invece in modalità interattiva, presentando un menù di scelte sul terminale dell'utente.

Vogliamo implementare uno script che supporti le seguenti funzionalità principali:

1. l'inserimento di un record in archivio;

2. la selezione di uno o più record tra quelli presenti in archivio, sulla base di un parametro inserito dall'utente, che permetta di selezionare solo alcuni dei record;

3. la cancellazione di uno o più record tra quelli presenti in archivio, sulla base di un parametro inserito dall'utente che permetta di selezionare i record da eliminare.

Procederemo alla progettazione e alla realizzazione dello script in modo "incrementale", implementando prima le singole funzioni come degli shell script autonomi, e successivamente assemblando tutte le funzionalità in un unico script più grande. In questo modo intendiamo anche suggerire un approccio pragmatico alla realizzazione di script modulari, in modo da arrivare progressivamente all'implementazione di script più complessi.

L'archivio degli indirizzi, come spesso accade per numerosi programmi ed utility in ambiente UNIX, è costituito da un file di testo ASCII, in cui ciascun record è memorizzato su una riga del file ed i campi del record sono distinti fra di loro attraverso un carattere separatore (nel nostro esempio è il carattere *pipe* "|"). Naturalmente in questo modo la struttura dell'archivio è rigidamente posizionale, ossia i campi devono rispettare sempre lo stesso ordine all'interno del record; nel nostro esempio la sequenza dei campi che costituiscono ciascun record è la seguente: nome, cognome, telefono ed e-mail. Un esempio di archivio è il seguente:

```
Marco|Liverani|06 123 456|m.liverani@aquilante.net
Enrico|Pili|329 876 432|epili@pippo.it
Natascia|Piroso|02 214 365|pinat@gmail.com
Marco|Pedicini| |m_pedicini@uniroma3.it
Carlo|Giuffrida|0462 987 654|carlo.giuffrida@pippo.it
```

Da notare che nel quarto record manca il numero telefonico, per cui nella posizione di quel campo c'è una stringa costituita da uno spazio.

6.2.1 Inserimento dati

La prima funzionalità che intendiamo implementare è quella con cui i campi di un nuovo record vengono aggiunti al file di archivio. Lo script "`recordInsert.sh`" che implementa questa funzionalità è riportato di seguito:

```
1  #!/bin/bash
2  # recordInsert.sh
3  # Riceve come argomento sulla linea di comando il nome, cognome,
4  # telefono e e-mail e li inserisce in un nuovo record della rubrica
5  RUBRICA=~/.rubrica
6  nome=$1
7  cognome=$2
8  telefono=$3
9  email=$4
10 echo "$nome|$cognome|$telefono|$email" >> $RUBRICA
11 echo "Inserito record n. $(wc -l $RUBRICA | cut -c 1-8)"
```

Lo script accetta quattro parametri sulla linea di comando, il cui significato, anche in questo caso, è posizionale: i quattro parametri saranno considerati rispettivamente come il il nome da inserire nella rubrica, il cognome, il telefono ed infine l'indirizzo di posta elettronica. Anche se lo script è molto breve, per migliorarne la leggibilità e la manutenibilità, a riga 5 è stata definita la variabile `RUBRICA` che contiene il nome del file di archivio (nell'esempio il file "`.rubrica`" nella home directory dell'utente, rappresentata dal simbolo "~").

Dopo aver memorizzato i quattro parametri nelle variabili `nome`, `cognome`, `telefono` ed `email`, l'istruzione fondamentale, che permette di aggiornare la rubrica degli indirizzi è quella riportata a riga 10, in cui l'output del comando "`echo`" viene rediretto, in modalità *append*, sul file `$RUBRICA`. Con l'operatore di redirezione ">>" l'output di "`echo`" viene aggiunto su una nuova riga alla fine del file; se il file non esiste, questo viene creato, aggiungendo come prima ed unica riga del file quella prodotta dal comando "`echo`".

L'istruzione a riga 11 consente di visualizzare un messaggio sul terminale dell'utente che indica il numero di record presenti nella rubrica dopo l'inserimento appena effettuato. Per effettuare questo calcolo viene utilizzato il comando "`wc`" (*word count*), che con l'opzione "`-l`" esegue il conteggio delle righe presenti nel file indicato sulla linea di comando.

L'output del comando "`wc`" è il seguente, che non si presta bene ad essere inserito nell'output prodotto dallo script:

```
$ wc -l ~/.rubrica
       5 /home/marco/.rubrica
```

Viene infatti riportato, oltre al numero di righe presenti nel file, anche il nome del file stesso ("`~/.rubrica`" nell'esempio).

Per "pulire" l'output prodotto da "`wc`", eliminando i caratteri non desiderati, si può utilizzare il comando "`cut`", con cui è possibile selezionare solo alcuni dei caratteri prodotti su standard output da "`wc`". In particolare nell'esempio precedente, con il comando "`cut -c 1-8`" si indica al comando di selezionare solo i caratteri dal primo all'ottavo, scartando tutto il resto. Collegando in *pipeline* i due comandi si ottiene il seguente risultato:

cut: seleziona (taglia) solo alcuni caratteri ricevuti da standard input

```
$ wc -l ~/.rubrica | cut -c 1-8
       5
```

Lo script "`recordInsert.sh`" produce quindi il seguente risultato:

```
$ ./recordInsert.sh Mario "Di Giovanni" "06 135 246" mdg@gmail.com
Inserito record n.        6
```

Da notare che sono stati utilizzati i doppi apici per delimitare quei valori che contenevano uno spazio: in questo modo la shell considera ciò che è delimitato dalle virgolette come un unico parametro, anche se nella sequenza di caratteri sono presenti degli spazi.

6.2.2 Selezione dati

La seconda funzionalità da implementare è quella per la selezione dei dati dalla rubrica, con cui è possibile effettuare delle ricerche nell'archivio e visualizzare in output i risultati ottenuti. Questa funzionalità si basa sull'uso del comando esterno "`grep`" che, come abbiamo visto in precedenza, consente di selezionare solo alcune delle righe presenti un un file, in base alla corrispondenza con una determinata stringa o *pattern*. Di seguito riportiamo lo script "`patternSelect.sh`" che implementa la funzionalità di ricerca e visualizzazione dei dati presenti nella rubrica.

```
1 #!/bin/bash
2 # patternSelect.sh
3 # Riceve una stringa e visualizza in output i record che contengono
4 # la stringa (selezione "case insensitive")
5 RUBRICA=~/.rubrica
6 LABEL=(' Nome' ' Cognome' 'Telefono' ' E-mail')
7 filtro=$1
```

```
8  echo "Record selezionati: $(grep -i $filtro $RUBRICA | wc -l)"
9  echo " "
10 IFS=$'\n'
11 for record in $(grep -i $filtro $RUBRICA | sort -t \| -k 2); do
12   IFS='|'
13   i=0
14   for campo in $record; do
15     echo "${LABEL[$i]}: $campo"
16     ((i++))
17   done
18   echo " "
19 done
```

Con le istruzioni alle righe 5 e 6 vengono definite due variabili di configurazione dello script. La prima (`RUBRICA`) contiene il path ed il nome del file/archivio della rubrica degli indirizzi; la seconda variabile (`LABEL`) è un array che contiene la definizione delle "etichette" con i nomi dei campi dell'archivio, utilizzate per rendere più chiara la visualizzazione dei dati selezionati dalla rubrica. Alla variabile `filtro` viene assegnato a riga 7 il valore del primo parametro passato allo script sulla linea di comando.

La funzione di ricerca e selezione dei record nell'archivio si basa sul comando esterno "`grep`" ed in particolare sulla seguente istruzione, presente sia a riga 8 che a riga 11:

```
grep -i $filtro $RUBRICA
```

Con questo comando vengono selezionate dal file il cui nome è memorizzato nella variabile `RUBRICA` (il file "`~/.rubrica`" nel nostro esempio) tutte e sole le righe che contengono una sequenza di caratteri uguale a quella memorizzata nella variabile `filtro`, a meno del *case* dei caratteri, visto che è stata usata l'opzione "`-i`" che rende *case insensitive* il comando "`grep`". In questo modo vengono selezionati dall'archivio i record corrispondenti al criterio di ricerca impostato dall'utente.

L'istruzione "`grep`" viene utilizzata due volte: la prima a riga 8 per contare il numero di record selezionati, indirizzando l'output verso il comando "`wc`" con un *pipe* ed utilizzando l'opzione "`-l`" per contare le righe. La seconda volta il comando "`grep`" viene usato per produrre una lista di stringhe (ciascun record selezionato dall'archivio è un elemento della lista) utilizzata nell'istruzione "`for`" a riga 11.

Il ciclo gestito dall'istruzione "`for`" a riga 11, esegue un'iterazione per ogni riga restituita dal comando "`grep`". I record selezionati dall'archivio sono riportati ciascuno su una riga, per cui sono separati l'uno dall'altro mediante un carattere di *carriage return*, identificato dalla sequenza "`\n`". I record dell'archivio, inoltre, possono contenere al loro interno anche dei caratteri di spaziatura. Per questo motivo, prima di eseguire il ciclo che fa variare il valore della variabile `record`, assegnandogli uno dopo l'altro gli elementi della lista restituita dal comando "`grep`", viene ridefinita la variabile speciale `IFS`. Il valore assegnato alla variabile a riga 10 identifica il carattere di "ritorno a capo": in questo modo si sostituisce in `IFS` il carattere di default per la separazione degli elementi di una lista.

\n: la sequenza che corrisponde al carattere *carriage return* o *new line*

IFS: la variabile speciale in cui è memorizzato il separatore di lista

A riga 11 l'output di "`grep`" viene passato, attraverso un *pipe*, al comando "`sort`" che ordina i record estratti dall'archivio in modo da consegnare all'istruzione "`for`" una lista ordinata alfabeticamente. I record dell'archivio contengono il cognome come secondo campo, ed il carattere con cui vengono separati i campi di un record è il simbolo "`|`" (*pipe*). Il comando "`sort`" viene quindi invocato con le opzioni "`-t \| -k 2`" che indicano che l'ordinamento deve essere eseguito sulla base del valore del secondo campo ("`-k 2`") e che il carattere che separa fra loro i campi di ogni riga di input è il *pipe* ("`-t \|`"). Da notare che, siccome il carattere "`|`" ha un significato ben preciso per

la Bash, affinché non venga interpretato come l'operatore *pipe*, viene preceduto da un carattere *backslash* "\".

Per ogni elemento della lista restituita da "grep" lo script suddivide la stringa in singoli campi e stampa in output ciascun campo del record. Per far questo viene modificato ancora una volta il valore della variabile speciale IFS, impostandolo con il carattere *pipe* (riga 12), utilizzato come separatore dei campi contenuti nei record della rubrica. Con il ciclo "for" di riga 14, vengono così selezionati, uno dopo l'altro, i campi del record e vengono visualizzati in output, preceduti dall'etichetta che identifica ciascun campo e che, per semplicità, è memorizzata nella componente i-esima dell'array LABEL.

Considerando la rubrica degli indirizzi riportata a pagina 56, di seguito viene presentato un esempio di output prodotto dallo script:

```
$ ./patternSelect.sh Liverani
    Nome: Marco
 Cognome: Liverani
Telefono: 06 123 456
  E-mail: m.liverani@aquilante.net
```

Naturalmente non c'è mai un solo modo per risolvere un problema con uno script. A titolo di esempio riportiamo di seguito uno script differente, che produce lo stesso risultato del precedente. In questo caso la soluzione proposta è incentrata sull'uso del comando esterno "awk" per la separazione e la visualizzazione dei singoli campi dei record selezionati con "grep".

```
1  #!/bin/bash
2  # patternSelectBis.sh
3  # Riceve una stringa e visualizza in output i record che contengono
4  # la stringa (selezione "case insensitive") utilizzando awk
5  RUBRICA=~/.rubrica
6  filtro=$1
7  echo "Record selezionati: $(grep -i $filtro $RUBRICA | wc -l)"
8  echo " "
9  grep -i $filtro $RUBRICA | sort -t \| -k 2 | awk '{ split($1,a,"|");\
10   printf " Nome: %s\n Cognome: %s\nTelefono: %s\n \
11   E-mail: %s\n\n", a[1], a[2], a[3], a[4] }'
```

awk: un interprete di un linguaggio per l'elaborazione di stringhe

Come scrive Alfred Aho, uno dei tre autori di AWK[1], AWK è un linguaggio per elaborare file di testo. Un file viene trattato da AWK come una sequenza di record, e ogni riga del file è un record. Ogni riga viene suddivisa in una sequenza di campi: possiamo pensare alla prima parola nella riga del file come al primo campo, alla seconda parola come al secondo campo, e così via. Un programma in AWK è una sequenza di istruzioni costituite da un pattern e da un'azione. AWK legge il file in input una riga alla volta e analizza la riga per verificare se corrisponde con uno dei pattern del programma; per ciascun pattern individuato, viene eseguita l'azione corrispondente.

Nel nostro caso il comando "awk" viene utilizzato in modo molto elementare, definendo un'azione molto semplice da applicare a tutte le stringhe che corrispondono ad un pattern nullo: la selezione dei record ed il loro successivo ordinamento alfabetico, sono infatti operazioni affidate ai comandi "grep" e "sort", il cui output viene alla fine fornito ad "awk" mediante l'operatore *pipe*.

Lo script "awk" viene riportato sulla riga di comando (spezzata su tre righe, nel nostro esempio, per ragioni di spazio) delimitato da apici. Al suo interno i comandi che costituiscono l'azione da eseguire sono delimitati da parentesi graffe. Tali comandi eseguono la suddivisione ("split") in campi della riga letta in input e la visualizzazione

[1] Gli altri due sono Peter Weinberger e Brian Kernighan, da cui l'acronimo AWK con cui è stato denominato il programma, utilizzando le iniziali dei cognomi dei tre autori.

in output ("printf") dei quattro campi memorizzati in altrettante celle dell'array a. Il comando "split" del linguaggio AWK accetta infatti tre argomenti: la stringa da suddividere in campi, il nome di un array in cui memorizzare i singoli campi e il carattere utilizzato come delimitatore dei campi (nel nostro esempio il simbolo "|").

Per concludere possiamo riportare una terza versione dello script per la ricerca e la visualizzazione dei dati presenti nella rubrica, in cui la selezione dei record avviene direttamente con il comando "awk", senza l'uso di "grep".

```
1  #!/bin/bash
2  # patternSelectTer.sh
3  # Riceve una stringa e visualizza in output i record che contengono
4  # la stringa utilizzando awk
5  RUBRICA=~/.rubrica
6  filtro=$1
7  echo "Record selezionati: $(grep -i $filtro $RUBRICA | wc -l)"
8  echo " "
9  sort -t \| -k 2 $RUBRICA | awk '/'$filtro'/ {split($1,a,"|"); \
10   printf " Nome: %s\n Cognome: %s\nTelefono: %s\n \
11   E-mail: %s\n\n", a[1], a[2], a[3], a[4]}'
```

In questo caso il comando "sort" esegue l'ordinamento alfabetico dell'intera rubrica in base al cognome, quindi l'output viene passato attraverso un *pipe* ad "awk" che applica l'azione di suddivisione e visualizzazione dei campi solo ai record che corrispondono con il pattern fornito dall'utente, memorizzato come al solito nella variabile filtro.

Il comando "awk" in quest'ultimo esempio viene utilizzato fornendogli uno script in una forma più generale, del tipo

awk '/*pattern*/ { *azione* }'

con cui viene eseguito il confronto tra il pattern ed ogni riga ricevuta in input e, nel caso in cui l'esito del confronto sia positivo, viene eseguita l'azione codificata con le istruzioni delimitate dalle parentesi graffe.

6.2.3 Eliminazione dati

L'ultima funzionalità che deve essere implementata riguarda la cancellazione di record dall'archivio. Proponiamo di seguito una soluzione molto semplice: acquisita in input sulla linea di comando una stringa che costituisce il criterio di selezione dei record da eliminare, si utilizza il comando "grep" con l'opzione "-v" per selezionare tutte le righe del file-rubrica che *non* contengono la stringa specificata dall'utente; il file con l'archivio degli indirizzi viene riscritto completamente solo con le righe selezionate da "grep" e, in questo modo, vengono di fatto eliminati i record (le righe del file) che contengono la stringa fornita dall'utente.

```
1  #!/bin/bash
2  # patternDelete.sh
3  # Riceve una stringa ed elimina dall'archivio tutti i record
4  # che la contengono
5  RUBRICA=~/.rubrica
6  filtro=$1
7  if [ $filtro ]; then
8    n=$(grep $filtro $RUBRICA|wc -l)
9    grep -v $filtro $RUBRICA > $RUBRICA.new
10   mv $RUBRICA $RUBRICA.old
11   mv $RUBRICA.new $RUBRICA
```

```
12   echo "Record eliminati: $n"
13 else
14   echo "$0: ERRORE: non e' stato specificato nessun filtro"
15   exit 1
16 fi
```

Per evitare di eliminare tutti i record dall'archivio per una semplice disattenzione da parte dell'utente, le istruzioni per la cancellazione delle righe dal file sono incluse in una struttura di controllo condizionale implementata dall'istruzione "`if-else`"; in questo modo si verifica, con la condizione a riga 7, che il pattern per la selezione dei record da eliminare non sia nullo.

Se il valore della variabile `filtro` non è nullo, viene creato un nuovo file, il cui nome ha estensione "`.new`", con i record che non contengono la stringa memorizzata nella variabile `filtro` (riga 9). Quindi viene modificato il nome del vecchio file di archivio aggiungendo l'estensione "`.old`" (riga 10) e viene infine rinominato il nuovo archivio con un nome privo dell'estensione "`.new`" (riga 11).

Se invece la variabile `filtro` ha valore nullo (nessuna stringa è stata passata allo script sulla linea di comando), viene visualizzato un messaggio di errore (riga 14) e lo script termina con un *return code* diverso da zero (riga 15).

6.2.4 Assembliamo le funzioni in uno script

A questo punto è possibile costruire uno script unico, che contenga tutte le funzionalità descritte nelle pagine precedenti.

Lo script viene progettato per operare sia in modalità "batch" che interattiva: se viene fornita sulla linea di comando un'opzione corrispondente ad una delle funzionalità implementate, lo script acquisisce i parametri sulla stessa linea di comando ed esegue la funzione corrispondente. Se invece viene lanciato senza alcun parametro sulla *command line*, lo script viene eseguito in modalità interattiva, con la possibilità di attivare le diverse funzionalità selezionando l'opzione corrispondente da un menù principale.

Il codice sorgente degli script visti nelle pagine precedenti, viene inglobato nel corpo di tre diverse funzioni: "`recordInsert`" per l'inserimento di un nuovo record in archivio, "`patternDelete`" per la cancellazione di uno o più record dall'archivio e "`patternSelect`" per la ricerca in archivio e la visualizzazione dei record selezionati.

Di seguito riportiamo il codice sorgente dell'intero script: come risulterà immediatamente evidente, il corpo delle tre funzioni è identico agli script riportati nelle pagine precedenti, a meno della definizione delle variabili `RUBRICA` e `LABEL`, che avviene alle righe 8 e 9, una volta per tutte per l'intero script.

```
1  #!/bin/bash
2  # rubrica.sh
3  # Rubrica telefonica interattiva.
4  #
5  # Variabili di configurazione globali
6  #
7  RUBRICA=~/.rubrica
8  LABEL=(' Nome' ' Cognome' 'Telefono' ' E-mail')
9  #
10 # Funzione: recordInsert
11 # Riceve come argomento nome, cognome, telefono e e-mail
12 # e li inserisce in un nuovo record della rubrica
13 #
14 function recordInsert {
15   nome=$1
```

```
16    cognome=$2
17    telefono=$3
18    email=$4
19    echo "$nome|$cognome|$telefono|$email" >> $RUBRICA
20    echo "Inserito record n. $(wc -l $RUBRICA | cut -c 1-8)"
21  }
22  #
23  # Funzione: patternDelete
24  # Riceve una stringa ed elimina dalla rubrica tutti i record
25  # che contengono esattamente quella stringa
26  #
27  function patternDelete {
28    filtro=$1
29    if [ $filtro ]; then
30      n=$(grep $filtro $RUBRICA|wc -l)
31      grep -v $filtro $RUBRICA > $RUBRICA.new
32      mv $RUBRICA $RUBRICA.old
33      mv $RUBRICA.new $RUBRICA
34      echo "Record eliminati: $n"
35    else
36      echo "$0: ERRORE: non e' stato specificato nessun filtro"
37      exit 1
38    fi
39  }
40  #
41  # Funzione: patternSelect
42  # Riceve una stringa e visualizza in output i record che contengono
43  # la stringa (selezione "case insensitive")
44  #
45  function patternSelect {
46    filtro=$1
47    echo "Record selezionati: $(grep -i $filtro $RUBRICA | wc -l)"
48    echo " "
49    IFS=$'\n'
50    for record in $(grep -i $filtro $RUBRICA | sort -t \| -k 2); do
51      IFS='|'
52      i=0
53      for campo in $record; do
54        echo "${LABEL[$i]}: $campo"
55        ((i++))
56      done
57      echo " "
58    done
59  }
60  #
61  # Procedura principale
62  #
63  opt=$1
64  # Se viene passata un'opzione sulla riga di comando
65  # opera in modalita' batch
66  if [ $opt ]; then
67    case $opt in
68      -h)
69        echo "Rubrica degli indirizzi"
```

```
70      echo " "
71      echo "usage: $0 [-h|-i|-d|-f] [params]"
72      echo " "
73      echo " -h : help"
74      echo " -i : insert ('$0 -i nome cognome telefono e-mail')"
75      echo " -d : delete ('$0 -d pattern')"
76      echo " -f : find ('$0 -f pattern')"
77      echo " ";;
78    -i)
79      recordInsert $2 $3 $4 $5;;
80    -d)
81      patternDelete $2;;
82    -f)
83      patternSelect $2;;
84    *)
85      echo "ERRORE: opzione non prevista";;
86  esac
87 # Altrimenti visualizza un menu' interattivo
88 else
89  PS3='--> '
90  clear
91  echo "RUBRICA INDIRIZZI"
92  select s in 'Inserimento record' 'Eliminazione record' \
93    'Ricerca record' 'Quit'; do
94    case $REPLY in
95      (1)
96        echo " "
97        echo "Inserimento nuovo record"
98        for (( i=0; i<4; i++)); do
99          echo -n "${LABEL[$i]}: "
100         read a[$i]
101       done
102       recordInsert "${a[0]}" "${a[1]}" "${a[2]}" "${a[3]}";;
103     (2)
104       echo " "
105       echo "Eliminazione record"
106       echo -n "Inserisci una stringa: "
107       read s
108       patternDelete $s;;
109     (3)
110       echo " "
111       echo "Ricerca record"
112       echo -n "Inserisci una stringa: "
113       read s
114       patternSelect $s;;
115     (4)
116       clear
117       exit 0;;
118     (*)
119       echo "Opzione errata";;
120   esac
121 done
122 fi
```

Il corpo principale dello script, da riga 60 in poi, è costituito da due parti distinte, gestite dalla struttura condizionale implementata dall'istruzione "`if-else`" alle righe 66 e 88. Alla variabile `opt` viene assegnato il primo dei parametri passati allo script sulla linea di comando (istruzione a riga 63). La condizione permette di controllare se il valore della variabile `opt` è nullo oppure no. In quest'ultimo caso, se la variabile ha un valore non nullo, significa che l'utente ha specificato delle opzioni sulla *command line* invocando lo script, quindi il programma opera in modalità batch in base all'opzione specificata dall'utente (righe 67–86). In caso contrario, se l'utente non ha specificato alcun parametro sulla riga di comando, lo script opera in modalità interattiva presentando un menù di opzioni all'utente (righe 88–122).

Le opzioni disponibili per il funzionamento in modalità batch sono quattro:

`-h` (help): Vengono visualizzate le istruzioni per l'uso del programma;

`-i` (insert): Inserimento di un nuovo record in archivio;

`-d` (delete): Cancellazione di uno o più record dall'archivio;

`-f` (find): Ricerca e visualizza i record presenti in archivio.

La selezione dell'opzione viene gestita mediante l'istruzione "`case`" (riga 67): in corrispondenza alle tre opzioni per l'inserimento, la cancellazione e la ricerca di record, vengono richiamate le rispettive funzioni passando come argomento i parametri forniti dall'utente sulla linea di comando.

La modalità interattiva viene implementata invece mediante l'istruzione "`select`" (righe 92–93) che presenta e gestisce un menù con quattro scelte: inserimento di un nuovo record, eliminazione di record, ricerca di record e chiusura del programma.

Anche in questo caso la scelta effettuata dall'utente viene gestita attraverso l'istruzione "`case`": per ciascuna opzione vengono acquisiti in input mediante l'istruzione "`read`" tutti i dati necessari per eseguire la funzione selezionata dall'utente, che vengono poi passati come argomento alla funzione corrispondente.

read: acquisisce in input dati da standard input

Di seguito, per concludere, riportiamo un esempio di sessione di lavoro con lo script.

```
$ ./rubrica.sh -h
Rubrica degli indirizzi

usage: ./rubrica.sh [-h|-i|-d|-f] [params]

       -h : help
       -i : insert ('./rubrica.sh -i nome cognome telefono e-mail')
       -d : delete ('./rubrica.sh -d pattern')
       -f : find ('./rubrica.sh -f pattern')

$ ./rubrica.sh

RUBRICA INDIRIZZI
1) Inserimento record     3) Ricerca record
2) Eliminazione record    4) Quit
--> 1

Inserimento nuovo record
    Nome: Mario
 Cognome: Rossi
Telefono: 02 345 678
  E-mail: mario.rossi@pluto.com
Inserito record n.        6
```

```
--> 3

Ricerca record
Inserisci una stringa: rossi
Record selezionati:        1

    Nome: Mario
 Cognome: Rossi
Telefono: 02 345 678
  E-mail: mario.rossi@pluto.com

--> 2

Eliminazione record
Inserisci una stringa: rossi
Record eliminati:        1
-->
1) Inserimento record     3) Ricerca record
2) Eliminazione record    4) Quit
--> 4
```

6.3 Script CGI

Navigando sul web spesso vengono visualizzati dei contenuti "statici", ossia che non cambiano nel tempo, come la pagina web contenente un libro o un manuale on-line; in altri casi, invece, il contenuto visualizzato utilizzando il web browser per navigare sulla rete Internet, non è "statico", ma "dinamico", ossia viene generato al momento della richiesta da un programma. Uno *script CGI* non è altro che un programmino che viene lanciato da un server HTTP quando, mediante un web browser, viene invocata una determinata URL gestita da tale server.

6.3.1 Alcuni cenni sulle tecnologie web

Quando eseguiamo una ricerca con Google, inseriamo delle parole in un campo di una form visualizzata mediante il web browser e quindi selezioniamo con il mouse il bottone che avvia la procedura di ricerca negli archivi di Google. Il risultato della nostra ricerca viene visualizzato su una pagina del browser: in questo caso si tratta di un contenuto "dinamico", prodotto in base ai criteri di ricerca che abbiamo inserito nella form. Selezionando il bottone "Cerca con Google" abbiamo inviato al server HTTP di Google le parole che costituiscono i nostri criteri di ricerca; il server Google esegue un programma che, ricevute tali parole, effettua la ricerca su un archivio e presenta in output il risultato; il server HTTP di Google invia il risultato al nostro web browser. In poche parole, e con qualche semplificazione, abbiamo descritto il processo con cui operano la maggior parte degli applicativi web based; un modo per realizzare un programma del genere è quello di scrivere uno *script CGI*.

HTTP: HyperText Transfer Protocol

HTTP, *HyperText Transfer Protocol* è il protocollo di comunicazione applicativo mediante cui vengono inviate le richieste dai web browser ai web server e vengono restituiti i contenuti multimediali di risposta dai web server ai web browser.

HTML: HyperText Mark-up Language

Le pagine web sono codificate con un linguaggio di marcatura del testo chiamato HTML, *HyperText Mark-up Language*. HTML non è un linguaggio di programmazione, ma un linguaggio con cui si possono inserire delle parole chiave all'interno di un testo, in modo tale da definire la struttura logica del documento e da caratterizzarne alcune

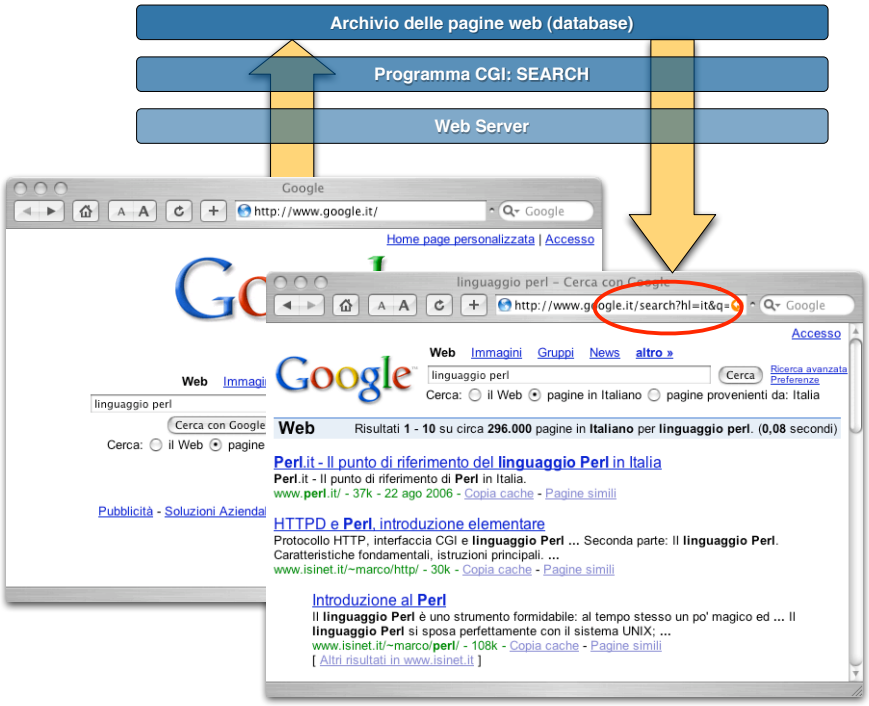

Figura 6.1: Il processo di chiamata del programma CGI e restituzione dell'output

parole o intere frasi. Ad esempio con HTML è possibile marcare alcune frasi come titoli di capitoli o di sezioni, è possibile evidenziare delle parole importanti, è possibile definire degli elenchi puntati o numerati e delle tabelle; è anche possibile definire dei riferimenti incrociati all'interno di uno stesso file o tra file differenti, in modo da trasformare un testo rigidamente sequenziale in un *ipertesto*. Con HTML è possibile anche inserire i riferimenti di un'immagine in formato digitale all'interno di un file di testo.

URI: Uniform Resource Identifier

URL: Uniform Resource Locator

I riferimenti ipertestuali ad altri documenti o contenuti multimediali (immagini, filmati, ecc.), vengono espressi in un formato noto come URI, *Uniform Resource Identifier*, con cui ogni singolo file o documento presente in rete può essere identificato univocamente. Nel formato URI sono espresse le URL, *Uniform Resource Locator*, che esprimono gli indirizzi delle risorse presenti sul web. Una tipica URL è una stringa nel seguente formato:

```
http://www.aquilante.net/bash/index.shtml
```

che identifica univocamente un file denominato "`index.shtml`", presente nella directory "`/bash`" sul server "`www.aquilante.net`", accessibile mediante il protocollo "`http`".

CGI: Common Gateway Interface

L'interfaccia CGI, *Common Gateway Interface*, è una specifica tecnica che permette di interfacciare un server HTTP con un programma che viene eseguito sullo stesso server. La specifica tecnica definisce il modo in cui i dati vengono passati dal server HTTP al programma CGI e il modo con cui l'output prodotto dal programma viene ricevuto dal server HTTP per poi essere spedito al browser dell'utente per la visualizzazione.

L'interfaccia CGI è uno strumento assai semplice, ma non particolarmente efficiente: si presta bene a realizzare rapidamente applicazioni web based semplici o con una bassa mole di traffico, mentre non è adatta per realizzare applicazioni complesse in grado di reggere il carico di centinaia o migliaia di connessioni contemporanee da parte di numerosi utenti. Per la sua semplicità è possibile comunque realizzare dei programmi anche con linguaggi di programmazione molto elementari, come Bash.

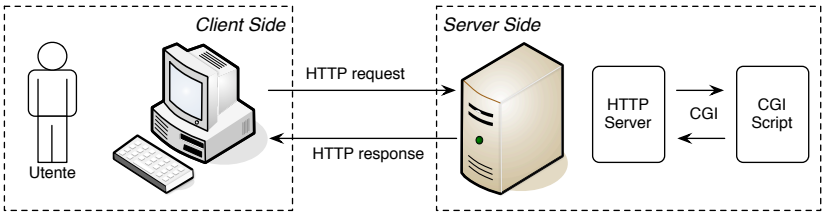

Figura 6.2: Schematizzazione dello scambio di dati dal web browser al web server attraverso il protocollo HTTP e dal web server allo script attraverso l'interfaccia CGI

Nel seguito faremo l'ipotesi di disporre di un server HTTP, ad esempio Apache HTTP Server, configurato in modo tale da poter eseguire script CGI. Negli esempi che seguono il server HTTP sarà eseguito sulla stessa macchina su cui gira il web browser, per cui l'accesso agli script sarà effettuato con URL del tipo "`http://localhost/...`".

6.3.2 L'interfaccia CGI

L'interfaccia CGI, come abbiamo detto, si occupa gestire il passaggio dei parametri dal server HTTP all'applicazione CGI e, al termine dell'esecuzione di quest'ultima, di catturare l'output prodotto su *standard output* e di passarlo al server HTTP. L'utente infatti non interagisce con lo script attraverso la *command line* del proprio terminale, ma mediante un web browser che comunica con il web server usando il protocollo HTTP. Per cui l'utente non è nelle condizioni di visualizzare ciò che lo script produce su *standard output*, a meno che questo non venga catturato ed inviato al web browser come risposta ad una richiesta HTTP: l'interfaccia CGI si occupa anche di questo.

L'utente può fornire un input ad uno script CGI, ad esempio compilando i campi di una web form. I dati inseriti in input dall'utente vengono raccolti dal web browser e trasformati in una stringa in formato *URL-encoded*. Tale formato prevede, tra l'altro, che i parametri siano concatenati fra loro delimitandoli con il carattere "&" e che ciascun dato sia espresso nella forma "*nome*=*valore*". La stringa deve essere priva di spazi, per cui ciascun carattere di spaziatura viene trasformato nel simbolo "+"; altre trasformazioni vengono effettuate per codificare i caratteri con un codice ASCII maggiore di 127 (tra questi le lettere accentate ed altri simboli).

Codifica URL-encoded dei parametri passati in input dall'utente

Ad esempio se l'utente fornisce in input i dati "nome", "cognome", "email", con i rispettivi valori "Marco", "Liverani", "m.liverani@aquilante.net", questi saranno raccolti dal browser e trasformati nella seguente stringa in formato URL-encoded:

```
nome=Marco&cognome=Liverani&email=m.liverani@aquilante.net
```

I dati, codificati in formato URL-encoded, vengono passati al web server attraverso uno dei due meccanismi previsti dal protocollo HTTP: il metodo "GET" o il metodo "POST". Nel primo caso la stringa con i parametri viene concatenata alla URL, utilizzando il carattere "`?`" come separatore; ad esempio:

Metodi HTTP GET e POST per il passaggio dei parametri

```
http://localhost/~marco/script.cgi?nome=Marco&cognome=Liverani&...
```

Nel caso del metodo POST, invece, la stringa in formato URL-encoded viene accodata alla richiesta HTTP senza essere mostrata nella URL.

Il server web, attraverso l'interfaccia CGI, rende disponibile la stringa dei parametri allo script in due modi diversi, a seconda del metodo HTTP usato per il passaggio dei parametri. Il metodo utilizzato viene indicato nella variabile d'ambiente `REQUEST_METHOD`, che assumerà il valore "`GET`" o "`POST`" a seconda dei casi. Se è stato usato il metodo GET,

Figura 6.3: L'output prodotto dallo script CGI "`data.cgi`"

la stringa viene memorizzata nella variabile d'ambiente `QUERY_STRING`, resa disponibile allo script CGI. Nel caso in cui, invece, sia stato usato il metodo POST, la stringa con i parametri viene passata allo script CGI direttamente attraverso il canale *standard input*.

Nel restituire l'output al server HTTP, lo script CGI dovrà semplicemente produrre una pagina in formato HTML su *standard output*, preceduta da una riga contenente la seguente stringa, seguita da una riga completamente vuota:

```
Content-Type: text/html
```

L'output prodotto in questo modo verrà catturato attraverso l'interfaccia CGI dal server HTTP e da questi inviato al web browser.

6.3.3 Un esempio elementare

Un primo esempio banale è quello di uno script che visualizza in output la data e l'ora corrente. Uno script di questo tipo non ha bisogno di acquisire in input alcun parametro e per questo è particolarmente facile implementarlo. Riportiamo di seguito il codice dello script.

```
1  #!/bin/bash
2  # data.cgi
3  # Lo script produce in output la data e l'ora corrente.
4  data=$(date +%d/%m/%Y)
5  ora=$(date +%H:%M)
6  echo "Content-Type: text/html"
7  echo
8  echo "<html>"
9  echo "<head><title>Data e ora</title></head>"
10 echo "<body><p>Oggi e' il $data e sono le $ora</p></body>"
11 echo "</html>"
```

Una volta copiato lo script in una directory compatibile con la configurazione del server HTTP (es.: la directory "`~/public_html`" su una macchina UNIX con server web Apache), abilitando l'esecuzione dello script (es.: "`chmod 755 data.cgi`"), è possibile verificarne il corretto funzionamento utilizzando un web browser, come nell'esempio in Figura 6.3.

Il passo successivo è quello di implementare le funzioni necessarie ad acquisire dei parametri dall'interfaccia CGI. Innanzi tutto predisponiamo una pagina HTML con una form in cui possano essere inseriti dei dati da passare ad uno script CGI.

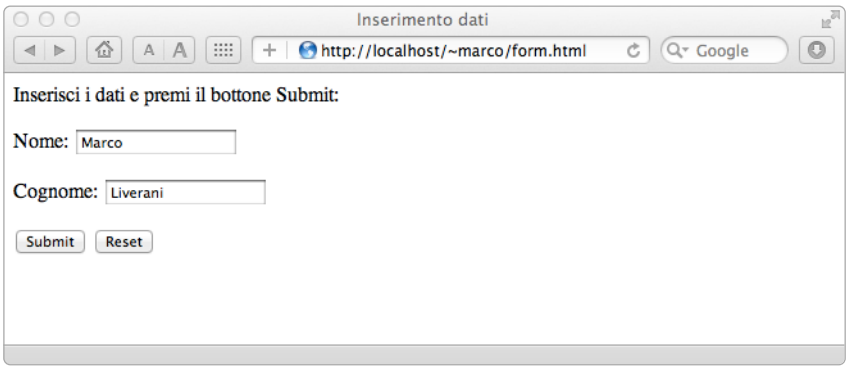

Figura 6.4: La semplice form HTML per l'invio di due parametri ad uno script CGI

La pagina del nostro esempio (Figura 6.4) presenta due campi di testo: uno per l'inserimento di un nome ed uno per l'inserimento di un cognome; la form è corredata dai due tipici bottoni "Submit" e "Reset": uno per il completamento della compilazione della form e l'invio dei dati allo script CGI e l'altro per il ripristino dello stato iniziale dei campi. Il codice sorgente in HTML è riportato di seguito.

```
1  <html>
2    <head>
3      <title>Inserimento dati</title>
4    </head>
5    <body>
6      <p>Inserisci i dati e premi il bottone Submit:</p>
7      <form action="param.cgi" method="GET">
8        <p>Nome: <input name="nome"></p>
9        <p>Cognome: <input name="cognome"></p>
10       <p><input type="submit"> <input type="reset"></p>
11     </form>
12   </body>
13 </html>
```

Da notare che a riga 7, il tag "`form`" presenta due attributi molto importanti: il primo è l'attributo "`action`" con cui deve essere specificata la URL dello script CGI che deve essere richiamato quando l'utente avrà completato la compilazione della form ed avrà premuto il bottone "Submit".

L'attributo "`method`", invece, consente di specificare il metodo HTTP con cui devono essere passati i dati inseriti nella form allo script CGI: il valore dell'attributo "`method`" deve essere quindi "`GET`" (come nell'esempio) o "`POST`".

I campi della form sono due, identificati dai nomi "`nome`" e "`cognome`", specificati con l'attributo "`name`" del tag "`input`". Quindi i dati inseriti dall'utente nella form saranno spediti al server HTTP con una stringa URL-encoded con il seguente formato:

`nome`=*nome inserito dall'utente*&`cognome`=*cognome inserito dall'utente*

A questo punto bisogna codificare una funzione Bash che consenta di acquisire i parametri passati dal server HTTP allo script attraverso l'interfaccia CGI. Come abbiamo detto la stringa con i parametri, in formato URL-encoded, viene resa disponibile allo script nella variabile d'ambiente `QUERY_STRING` oppure mediante il canale di *standard input* a seconda del metodo scelto per il passaggio dei parametri, rispettivamente "GET" o "POST".

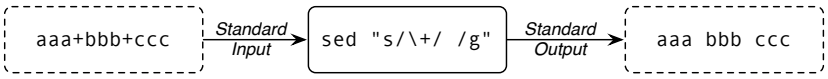

Figura 6.5: Un esempio del funzionamento del comando "sed"

Di seguito riportiamo il codice sorgente della funzione Bash "getCgiString" che acquisisce la stringa URL-encoded e la restituisce in output su *standard output*.

```
1  function getCgiString {
2    local stringa
3    if [ "$REQUEST_METHOD" == "GET" ]; then
4      stringa=$QUERY_STRING
5    else
6      read stringa
7    fi
8    stringa=$(echo $stringa | sed "s/\+/ /g")
9    echo "$stringa"
10 }
```

A riga 2 viene verificato il valore della variabile d'ambiente REQUEST_METHOD impostata dal server HTTP: se il valore è "GET", alla variabile stringa viene assegnato il valore della variabile d'ambiente QUERY_STRING, altrimenti a riga 5 viene letta da *standard input* una stringa di caratteri, che viene memorizzata nella variabile stringa.

Con l'istruzione presente a riga 8, viene parzialmente decodificato il formato URL-encoded della stringa memorizzata nella variabile stringa, trasformando tutti i caratteri "+" in caratteri di spaziatura. Per far questo il valore di stringa viene passato in input mediante l'operatore *pipe* dal comando "echo" al comando esterno "sed".

sed: stream editor, consente di manipolare stringhe con espressioni regolari

Il comando "sed" accetta sulla linea di comando una stringa che rappresenta un'espressione regolare con cui deve essere elaborata la sequenza di caratteri ricevuta in input. Un'espressione regolare è una stringa che, attraverso dei meta-caratteri, consente di definire un pattern che identifica un'insieme di stringhe. Per maggiori informazioni sulla sintassi delle espressioni regolari gestite da "sed" si suggerisce di fare riferimento alla pagina di manuale ("man sed").

Nell'istruzione riportata a riga 8 al comando "sed" viene passata come argomento l'espressione "s/\+/ /g", che indica che tutte le occorrenze di una certa stringa devono essere sostituite con un'altra stringa. Il carattere "s" all'inizio dell'espressione indica che si richiede a "sed" di effettuare una sostituzione; la stringa da cercare è descritta dall'espressione regolare delimitata dalla prima coppia di caratteri "/"; tale stringa è costituita dal carattere "+" identificato dalla sequenza "\+", dal momento che il carattere "+" ha un significato speciale nel contesto delle espressioni regolari. La stringa da sostituire ad ogni occorrenza del pattern è delimitata dalla seconda coppia di caratteri "/" (lo spazio). Il carattere "g" al termine dell'espressione indica che la sostituzione deve essere eseguita globalmente, per ogni occorrenza della prima stringa e non solo per la prima occorrenza.

Ad esempio, se viene passata in input al comando "sed "s/\+/ /g"" la stringa "aaa+bbb+ccc", si ottiene in output la stringa "aaa bbb ccc", come rappresentato in Figura 6.5. Naturalmente la decodifica del formato URL-encoded è del tutto incompleta e dunque, al di là dell'esempio riportato in queste pagine, la funzione "getCgiString" andrebbe implementata ulteriormente, per renderla più robusta ed efficace, permettendo una decodifica completa di tutti i meta-caratteri utilizzati nella codifica URL-encoding.

Una prima versione dello script "param.cgi" che riceve ed utilizza i parametri forniti dall'utente mediante la form HTML presentata a pag. 69, può essere implemen-

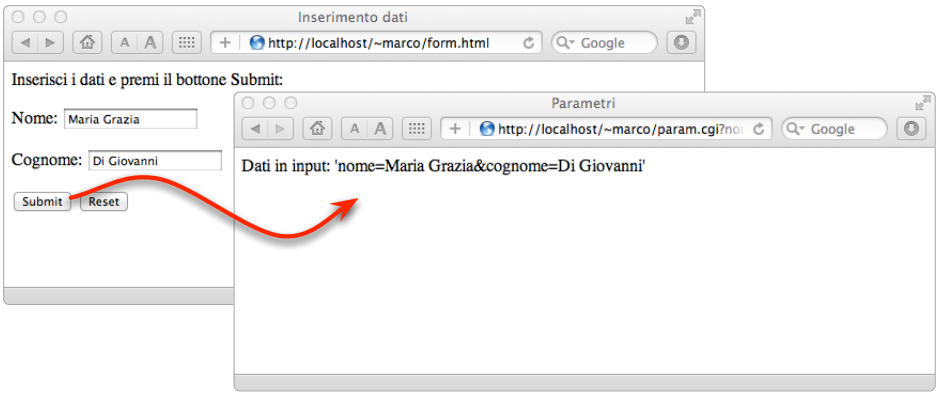

Figura 6.6: Dalla pagina "`form.html`" viene richiamato lo script "`param.cgi`" passando i dati inseriti dall'utente nei campi della form; lo script CGI visualizza i dati ricevuti attraverso l'interfaccia CGI come un'unica stringa di testo

tata facendo uso della funzione "`getCgiString`". Di seguito ne riportiamo il codice sorgente.

```
1  #!/bin/bash
2
3  function getCgiString {
4    local stringa
5    if [ "$REQUEST_METHOD" == "GET" ]; then
6      stringa=$QUERY_STRING
7    else
8      read stringa
9    fi
10   stringa=$(echo $stringa | sed "s/\+/ /g")
11   echo "$stringa"
12 }
13
14 dati=$(getCgiString)
15 echo "Content-Type: text/html"
16 echo
17 echo "<html>"
18 echo "<head><title>Parametri</title></head>"
19 echo "<body>"
20 echo "<p>Dati in input: '$dati'</p>"
21 echo "</body>"
22 echo "</html>"
```

A riga 14 viene invocata la funzione "`getCgiString`" memorizzando la stringa URL-encoded con i parametri prodotta in output, nella variabile `dati`. Poi viene generata dinamicamente una pagina HTML in cui viene presentata la stringa di parametri ricevuta mediante l'interfaccia CGI (riga 20). In Figura 6.6 viene presentato l'output prodotto da questo script.

Per poter utilizzare i dati forniti in input allo script mediante l'interfaccia CGI è necessario predisporre una funzione in grado di estrarre i valori dei singoli parametri dalla stringa URL-encoded.

Di seguito riportiamo il codice sorgente della funzione "`getCgiParam`" che, ricevuto come argomento il nome del parametro che si intende estrarre e la stringa URL-

encoded, restituisce il valore del parametro su *standard output*. Ad esempio, se la stringa URL-encoded è "`nome=Marco&cognome=Liverani`", la seguente istruzione:

```
getCgiParam cognome "nome=Marco&cognome=Liverani"
```

produce in output il valore "`Liverani`".

```
1  function getCgiParam {
2    local nome stringa a oldIFS cgiValue
3    nome=$1
4    stringa=$2
5    oldIFS=$IFS
6    IFS="&"
7    for parametro in $stringa; do
8      IFS="="
9      a=($parametro)
10     if [ "${a[0]}" == "$nome" ]; then
11       cgiValue=${a[1]}
12       break
13     fi
14     IFS="&"
15   done
16   IFS=$oldIFS
17   echo "$cgiValue"
18 }
```

La funzione, dopo aver ridefinito la variabile speciale `IFS` in cui è memorizzato il carattere utilizzato da Bash per separare gli elementi di una lista, impostando come valore il carattere "`&`" (riga 6), esegue una iterazione su ogni elemento della stringa URL-encoded: in questo caso gli elementi della lista sono le stringhe "*nome=valore*" separate l'una dall'altra dal carattere "`&`", memorizzate nella variabile `parametro` ad ogni iterazione del ciclo "`for`".

A riga 8 viene quindi ridefinita la variabile `IFS` impostando come valore il carattere "=". In questo modo il valore della variabile `parametro` viene considerata come una lista composta da due elementi: il primo è il nome del parametro passato attraverso l'interfaccia CGI, il secondo è il valore di tale parametro. Con l'istruzione "`a=($parametro)`" di riga 9, si costruisce un array "`a`" con due elementi: il nome del parametro (`a[0]`) e il suo valore (`a[1]`).

Se il valore di `a[0]` coincide con il nome del parametro che si intende estrarre dalla stringa (riga 10), il valore di `a[1]` viene memorizzato nella variabile `cgiValue` (riga 11). Al termine della funzione viene restituito in output il valore di `cgiValue` (riga 17).

Utilizzando anche la funzione "`getCgiParam`" possiamo migliorare lo script precedente che, ricevuti dei dati da una form, li presenta in output. La form è la stessa già vista a pag. 69, che consente di acquisire in input i campi `nome` e `cognome`. Lo script è riportato di seguito.

```
1  #!/bin/bash
2  # param.cgi
3  # Visualizza i parametri "nome" e "cognome" passati attraverso
4  # l'interfaccia CGI
5  #
6  # Funzione: getCgiParam
7  # Estrae dalla stringa dei parametri quello il cui nome corrisponde
8  # con l'argomento della funzione; restituisce in output il valore
9  # del parametro
10 #
```

```
11 function getCgiParam {
12   local nome stringa a oldIFS cgiValue
13   nome=$1
14   stringa=$2
15   oldIFS=$IFS
16   IFS="&"
17   for parametro in $stringa; do
18     IFS="="
19     a=($parametro)
20     if [ "${a[0]}" == "$nome" ]; then
21       cgiValue=${a[1]}
22       break
23     fi
24     IFS="&"
25   done
26   IFS=$oldIFS
27   echo "$cgiValue"
28 }
29 #
30 # Funzione: getCgiString
31 # Acquisisce i parametri passati allo script attraverso l'interfaccia
32 # CGI e restituisce in output la stringa in formato URL-encoded
33 #
34 function getCgiString {
35   local stringa
36   if [ "$REQUEST_METHOD" == "GET" ]; then
37     stringa=$QUERY_STRING
38   else
39     read stringa
40   fi
41   stringa=$(echo $stringa | sed "s/\+/ /g")
42   echo "$stringa"
43 }
44 #
45 # Corpo principale dello script
46 #
47 stringa=$(getCgiString)
48 nome=$(getCgiParam nome $stringa)
49 cognome=$(getCgiParam cognome $stringa)
50 echo "Content-Type: text/html"
51 echo
52 echo "<html>"
53 echo "<head><title>Parametri</title></head>"
54 echo "<body>"
55 echo "<p>Dati in input: '$stringa'</p>"
56 echo "<p>Nome: '$nome'</p>"
57 echo "<p>Cognome: '$cognome'</p>"
58 echo "</body>"
59 echo "</html>"
```

Un esempio di output prodotto dallo script "`param.cgi`" è riportato in Figura 6.7. Di fatto, dopo aver affidato alle funzioni "`getCgiParam`" e "`getCgiString`" l'acquisizione dei dati passati in input attraverso l'interfaccia CGI, il corpo dello script si riduce ad invocare per prima la funzione "`getCgiString`", memorizzandone l'output nella

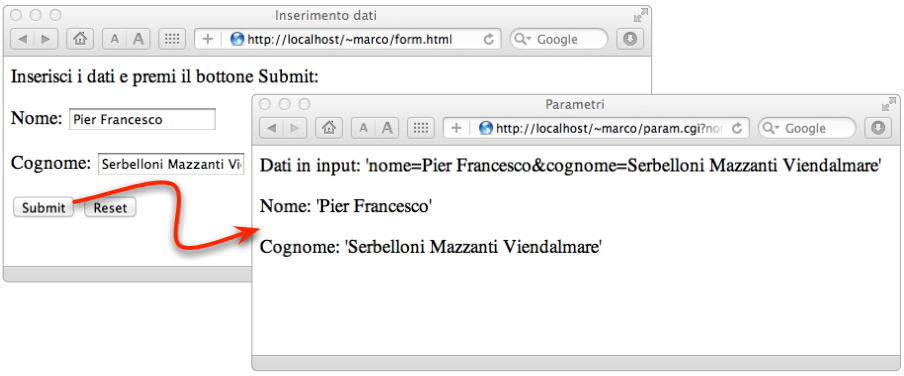

Figura 6.7: Esempio di output prodotto dallo script che acquisisce in input i dati inseriti nella form

variabile `stringa` (riga 47) e successivamente richiamando per due volte la funzione "`getCgiParam`" per memorizzare nelle variabili `nome` e `cognome` i due dati presenti nella stringa ricevuta in input. Alle righe 56 e 57 i due dati vengono presentati in output nella pagina HTML composta dinamicamente.

6.4 Rubrica degli indirizzi web based

In quest'ultimo esempio riscriveremo il programma per la gestione interattiva di una rubrica di indirizzi, presentato nella sezione 6.2, in modo tale da trasformarlo in un'applicazione web based.

Di fatto lo script deve operare in modo simile ad una modalità batch in cui, attraverso uno dei parametri, viene specificata la funzionalità che si intende eseguire. L'output naturalmente sarà prodotto in formato HTML in modo da poter essere visualizzato correttamente su un web browser; le operazioni di input dei dati saranno implementate con delle form HTML.

La struttura dell'applicazione web based è schematizzata in Figura 6.8. L'applicazione è basata su quattro file HTML ed uno script CGI. I file HTML sono dedicati uno al menù principale dell'applicazione e tre alle form di inserimento dei dati necessari ad effettuare le operazioni di inserimento di un record in archivio, di ricerca e visualizzazione dei record in archivio e di cancellazione di record dall'archivio. Lo script CGI è uno soltanto: riceve in input attraverso l'interfaccia CGI diversi parametri; tra questi il parametro "`func`" indica la funzionalità che si intende attivare e può quindi assumere i seguenti valori: "`insert`" per l'inserimento di un record in archivio, "`select`" per la ricerca e la visualizzazione e "`delete`" per la cancellazione di record.

Di seguito riportiamo il sorgente del primo file HTML, quello con il menù principale dell'applicazione. Il menù viene realizzato mediante una lista i cui elementi contengono i collegamenti ipertestuali alle pagine HTML con le form per l'inserimento dei dati da passare allo script CGI.

```
1 <html>
2   <head>
3     <title>Rubrica Indirizzi - Men&ugrave; principale</title>
4   </head>
5   <body>
6     <h1>Rubrica Indirizzi</h1>
7     <p>Seleziona una delle funzioni disponibili:</p>
8     <ul>
```

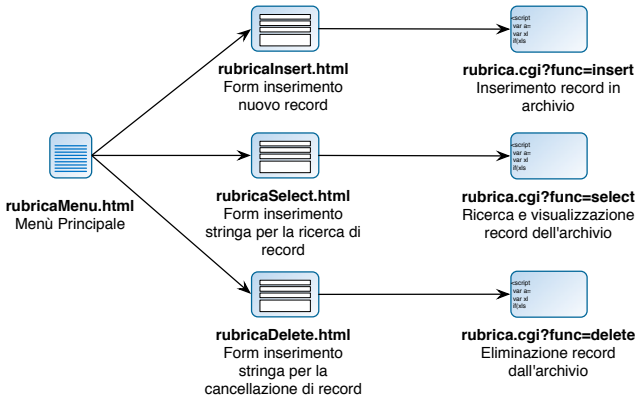

Figura 6.8: Schematizzazione delle relazioni tra le pagine HTML e le chiamate allo script CGI che implementa le diverse operazioni previste dall'applicazione web based: inserimento, selezione/visualizzazione e cancellazione dei record dall'archivio

```
      <li><a href="rubricaInsert.html">Inserimento record</a>:
      Inserimento di un nuovo record nell'archivio.</li>
      <li><a href="rubricaSelect.html">Ricerca record</a>:
      Visualizzazione record selezionati in base ad una stringa</li>
      <li><a href="rubricaDelete.html">Cancellazione record</a>:
      Cancellazione record selezionati in base ad una stringa</li>
    </ul>
  </body>
</html>
```

I file HTML con le form per l'inserimento dei dati sono piuttosto semplici. È importante osservare, però, che contengono tutte un campo di input denominato "`func`", caratterizzato dall'attributo "`hidden`" che indica al browser che quel campo non deve essere visualizzato nella form. Il valore del campo "`func`" è diverso in ciascuna form, perché lo script CGI deve eseguire operazioni differenti a seconda della pagina da cui viene richiamato. Di seguito riportiamo il sorgente delle tre pagine, rispettivamente "`rubricaInsert.html`", "`rubricaSelect.html`" e "`rubricaDelete.html`".

```
<html>
  <head>
    <title>Rubrica Indirizzi - Inserimento record</title>
  </head>
  <body>
    <h1>Rubrica Indirizzi</h1>
    <p>Inserisci i dati nei campi e seleziona il bottone Submit
    per salvarli in archivio:</p>
    <form action="rubrica.cgi" method="POST">
      <input type="hidden" name="func" value="insert">
      <p>Nome: <input name="nome">
         Cognome: <input name="cognome"></p>
      <p>Telefono: <input name="telefono">
         E-Mail: <input name="email"></p>
      <p><input type="submit"> <input type="reset"></p>
    </form>
  </body>
</html>
```

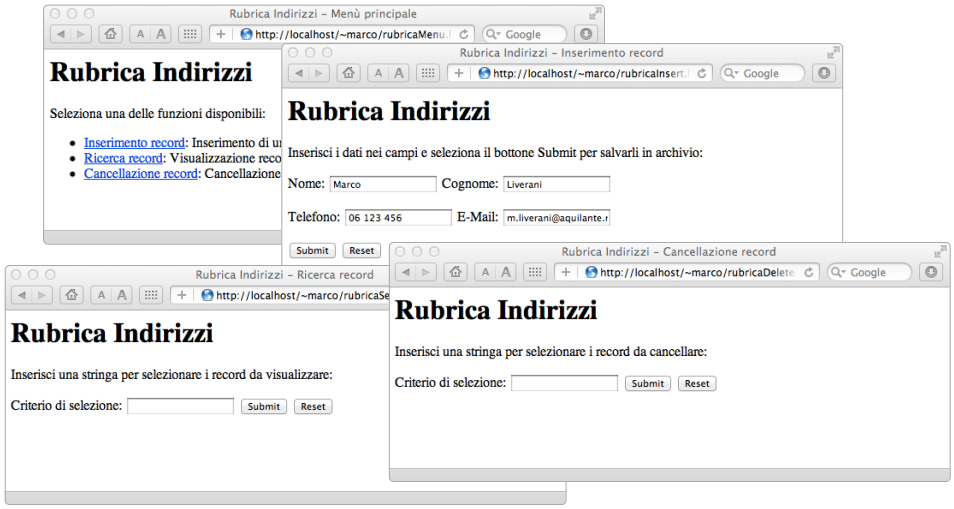

Figura 6.9: L'output prodotto dai file HTML dell'applicazione

```
<html>
  <head>
    <title>Rubrica Indirizzi - Ricerca record</title>
  </head>
  <body>
    <h1>Rubrica Indirizzi</h1>
    <p>Inserisci una stringa per selezionare i record da
       visualizzare:</p>
    <form action="rubrica.cgi" method="POST">
      <input type="hidden" name="func" value="select">
      <p>Criterio di selezione: <input name="stringa">
        <input type="submit"> <input type="reset"></p>
    </form>
  </body>
</html>
```

```
<html>
  <head>
    <title>Rubrica Indirizzi - Cancellazione record</title>
  </head>
  <body>
    <h1>Rubrica Indirizzi</h1>
    <p>Inserisci una stringa per selezionare i record da
       cancellare:</p>
    <form action="rubrica.cgi" method="POST">
      <input type="hidden" name="func" value="delete">
      <p>Criterio di selezione: <input name="stringa">
        <input type="submit"> <input type="reset"></p>
    </form>
  </body>
</html>
```

I quattro file HTML producono l'output riportato in Figura 6.9 se richiamate con un web browser.

Lo script CGI "rubrica.cgi" richiamato dalle tre form HTML, viene costruito modificando lo script "rubrica.sh" riportato a pag. 61. Rispetto a quest'ultimo, lo script CGI non deve gestire il menù interattivo, ma deve solo invocare le diverse funzioni a seconda del valore del parametro "func" passato da ciascuna form HTML attraverso l'interfaccia CGI. Inoltre, naturalmente, viene modificato il modo di presentare l'output che, per l'applicazione web based, deve essere codificato in HTML per poter essere visualizzato correttamente mediante un web browser.

Per l'acquisizione dei dati in input attraverso l'interfaccia CGI, vengono utilizzate le funzioni "getCgiString" e "getCgiParam" già descritte nelle pagine precedenti. Di seguito riportiamo il sorgente completo dello script.

```
#!/bin/bash
#
# rubrica.cgi
# Rubrica telefonica web based.
#
RUBRICA="./rubrica.data"
#
# Funzione: getCgiString
# Acquisisce i parametri passati allo script attraverso l'interfaccia
# CGI e restituisce in output la stringa in formato URL-encoded
#
function getCgiString {
  local stringa
  if [ "$REQUEST_METHOD" == "GET" ]; then
    stringa=$QUERY_STRING
  else
    read stringa
  fi
  stringa=$(echo $stringa | sed "s/%40/@/g")
  stringa=$(echo $stringa | sed "s/\+/ /g")
  echo "$stringa"
}
#
# Funzione: getCgiParam
# Estrae dalla stringa dei parametri quello il cui nome corrisponde
# con l'argomento della funzione; restituisce in output il valore
# del parametro
#
function getCgiParam {
  local nome stringa a oldIFS cgiValue
  nome=$1
  stringa=$2
  oldIFS=$IFS
  IFS="&"
  for parametro in $stringa; do
    IFS="="
    a=($parametro)
    if [ "${a[0]}" == "$nome" ]; then
      cgiValue=${a[1]}
      break
    fi
```

```
42      IFS="&"
43    done
44    IFS=$oldIFS
45    echo "$cgiValue"
46  }
47  #
48  # Funzione: recordInsert
49  # Riceve come argomento nome, cognome, telefono e e-mail
50  # e li inserisce in un nuovo record della rubrica
51  #
52  function recordInsert {
53    local nome=$1
54    local cognome=$2
55    local telefono=$3
56    local email=$4
57    echo "$nome|$cognome|$telefono|$email" >> $RUBRICA
58    echo "<p>Inserito record n. $(wc -l $RUBRICA | cut -c 1-8)</p>"
59  }
60  #
61  # Funzione: patternDelete
62  # Riceve una stringa ed elimina dalla rubrica tutti i record
63  # che contengono esattamente quella stringa
64  #
65  function patternDelete {
66    local f=$1
67    if [ $f ]; then
68      local n=$(grep "$f" $RUBRICA|wc -l)
69      grep -v "$f" $RUBRICA > $RUBRICA.new
70      mv $RUBRICA $RUBRICA.old
71      mv $RUBRICA.new $RUBRICA
72      echo "<p>Filtro: '$f' - Record eliminati: $n</p>"
73    else
74      echo "<p>ERRORE: non e' stato specificato nessun filtro</p>"
75      exit 1
76    fi
77  }
78  #
79  # Funzione: patternSelect
80  # Riceve una stringa e visualizza in output i record che contengono
81  # la stringa (selezione "case insensitive")
82  #
83  function patternSelect {
84    local record campo
85    local f="$1"
86    echo "<p>Filtro: '$f' - "
87    echo "Record selezionati: $(grep -i "$f" $RUBRICA | wc -l)</p>"
88    echo "<table border=\"1\"><tr><th>Nome</th><th>Cognome</th>"
89    echo "<th>Telefono</th><th>E-mail</th></tr>"
90    IFS=$'\n'
91    for record in $(grep -i "$f" $RUBRICA | sort -t \| -k 2)
92    do
93      echo -n "<tr>"
94      IFS='|'
95      for campo in $record
```

```
96      do
97        echo -n "<td>$campo</td>"
98      done
99      echo "</tr>"
100   done
101   echo "</table>"
102   unset IFS
103 }
104 #
105 # Procedura principale
106 #
107 stringa=$(getCgiString)
108 opt=$(getCgiParam func "$stringa")
109 echo "Content-Type: text/html"
110 echo
111 echo "<html><head><title>Rubrica Indirizzi - $opt</title></head>"
112 echo "<body><h1>Rubrica Indirizzi - $opt</h1>"
113 # Viene valutato il valore del parametro "func" che definisce
114 #  l'operazione che deve essere eseguita dallo script
115 case $opt in
116   insert)
117     nome=$(getCgiParam nome "$stringa")
118     cognome=$(getCgiParam cognome "$stringa")
119     telefono=$(getCgiParam telefono "$stringa")
120     email=$(getCgiParam email "$stringa")
121     recordInsert "$nome" "$cognome" "$telefono" "$email";;
122   delete)
123     filtro=$(getCgiParam stringa "$stringa")
124     patternDelete "$filtro";;
125   select)
126     filtro=$(getCgiParam stringa "$stringa")
127     patternSelect "$filtro";;
128 esac
129 echo "<p><a href=\"rubricaMenu.html\">Torna al men&ugrave;</a></p>"
130 echo "</body></html>"
```

Lo script CGI presenta alcune piccole varianti rispetto allo shell script interattivo. Innanzi tutto abbiamo modificato il path del file di archivio: ora si trova nella stessa directory dello script CGI (ad esempio in "`/home/marco/public_html`") perché lo script non viene più eseguito con la nostra utenza personale, ma da un'utenza di sistema con cui opera il servizio HTTP server. Per questo lo script non è più in grado di risolvere il path "`~/`" come la nostra home directory. Inoltre l'utente che esegue il servizio HTTP server non è autorizzato a scrivere nella nostra home directory; per cui è opportuno spostare il file-archivio nella directory che ospita gli altri file della web application. Questa directory deve essere abilitata in scrittura in modo tale che anche l'utente che esegue il server HTTP possa modificare il file-archivio mediante l'esecuzione dello script "`rubrica.cgi`". Il file-archivio è stato rinominato "`rubrica.data`" (riga 6).

Una modifica è stata apportata anche alla funzione "`getCgiString`", introducendo l'istruzione a riga 19 che, utilizzando il comando "`sed`" esegue un'ulteriore decodifica della stringa con i dati in formato URL-encoded, traducendo la stringa "`%40`" nel carattere "`@`" che è presente negli indirizzi di posta elettronica gestiti dalla rubrica.

Il corpo principale dello script è costituito dalle istruzioni alle righe 107–130: dopo aver acquisito la stringa dei dati in input (riga 107), si estrae il valore del parametro "`func`" che viene memorizzato nella variabile `opt` (riga 108). Con un'istruzione "`case`"

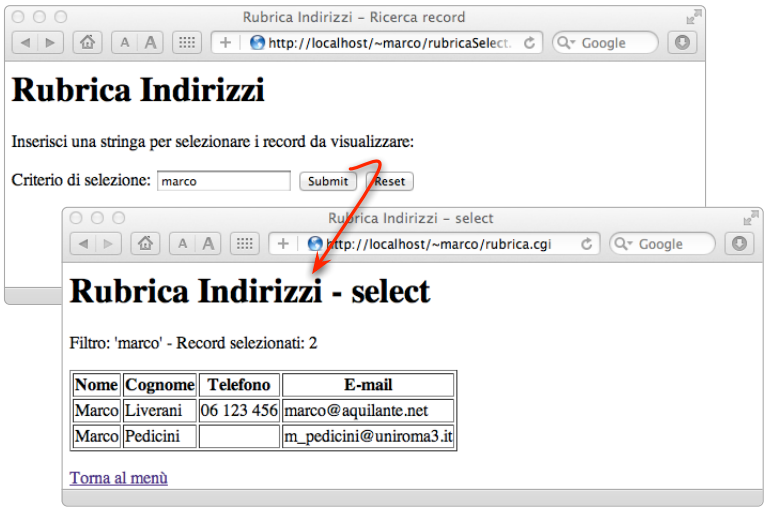

Figura 6.10: Due schermate dell'applicazione rubrica web based

viene quindi valutato il valore contenuto in `opt` e di conseguenza vengono estratti i dati necessari dalla stringa URL-encoded e vengono invocate le funzioni che implementano le tre operazioni di inserimento, cancellazione e ricerca di record nell'archivio.

Concludiamo riportando un esempio di funzionamento del programma in Figura 6.10: vengono visualizzati tutti i record presenti in archivio contenenti la stringa "`marco`".

Appendice A

Sintesi dei comandi principali

In questa appendice viene riportata una descrizione sintetica dei principali comandi del linguaggio Bash (comandi interni) e di alcuni dei principali programmi ed utility presenti tipicamente nella configurazione di base dei sistemi UNIX/Linux (comandi esterni). Per una descrizione più ampia e dettagliata dei comandi si suggerisce di fare riferimento comunque alle pagine di manuale UNIX[1] o ai testi citati a pagina 85 tra i riferimenti bibliografici.

Comandi interni

`alias` Consente di definire un sinonimo di un comando o di un'istruzione complessa o di ridefinire un comando (es.: "`alias ls='ls -F'`").

`bg` Consente di eseguire in background un comando.

`break` Interrompe l'esecuzione del blocco di istruzioni all'interno di una struttura di controllo condizionale o iterativa.

`case` / `esac` È una struttura di controllo condizionale che consente di valutare il contenuto di una variabile ed eseguire istruzioni diverse per ciascun valore.

`cd` Cambia la directory corrente del filesystem in cui viene eseguito lo shell script; se non viene specificato nessun argomento sulla linea di comando, definisce come directory corrente la home directory dell'utente, altrimenti si sposta nella directory specificata come argomento.

`continue` Prosegue l'esecuzione di un ciclo dall'iterazione successiva (o dalla n-esima iterazione, se n viene specificato come argomento).

`do` / `done` Delimitano il blocco di istruzioni che deve essere eseguito nell'ambito di una struttura di controllo iterativa (un ciclo).

`echo` Visualizza in output una stringa di testo costituita da caratteri alfanumerici e variabili; con l'opzione "`-n`" non va a capo dopo aver visualizzato l'output.

`eval` Esegue il comando ottenuto concatenando tutti i parametri passati come argomento.

[1] Ricordiamo che le pagine del manuale UNIX sono accessibili utilizzando il comando "`man`" seguito dal comando di cui si desidera visualizzare la descrizione (es.: "`man bash`").

`exec` Esegue il comando specificato come argomento sostituendo la shell in esecuzione: non viene creato un nuovo processo, ma il processo corrente viene sostituito da quello specificato come argomento del comando exec.

`exit` Termina l'esecuzione dello script e restituisce un *return code* numerico.

`export` Definisce una variabile d'ambiente, ovvero esporta una variabile o una funzione definita nell'ambito dello shell script, nell'ambiente entro cui è stato lanciato lo script ed i comandi eseguiti successivamente nell'ambito della stessa shell; contestualmente all'esportazione di una variabile può anche essere impostato il valore della variabile stessa.

`fg` Riporta in *foreground* il processo il cui *job id* è passato al comando come argomento.

`for` Esegue più volte il blocco di istruzioni delimitato dalle parole chiave "`do`" e "`done`"; il ciclo viene ripetuto per ciascun valore assunto da una variabile di controllo in una lista di valori o in un intervallo numerico.

`function` Definisce una funzione identificata da un nome univoco nell'ambito dello script; il corpo della funzione è delimitato dalle parentesi graffe.

`if` / `then` / `else` / `fi` Definisce una struttura di controllo condizionale: se la condizione riportata dopo l'istruzione "`if`" risulta vera, vengono eseguite le istruzioni che seguono la parola chiave "`then`", altrimenti, se specificate, vengono eseguite le istruzioni riportate dopo la parola chiave "`else`"; la struttura di controllo è terminata dalla parola chiave "`fi`".

`jobs` Visualizza l'elenco dei "job id" dei processi attivi nell'ambito della shell corrente entro cui viene eseguito lo script.

`kill` Invia il segnale specificato come primo argomento del comando ai processi i cui "job id" o "process id" sono passati come argomenti successivi.

`local` Definisce le variabili specificate come argomento; nell'ambito di una funzione definita in uno script limita lo *scope* alla funzione stessa delle variabili specificate argomento.

`printf` Stampa in output (o assegna ad una variabile) una stringa il cui formato viene specificato come primo argomento, che può essere costituita anche dal valore delle variabili riportate di seguito.

`read` Legge da standard input una sequenza di caratteri contenente delle "parole" (o dei valori) separati da spazi; la prima parola viene assegnata alla prima variabile riportata come argomento del comando, la seconda alla seconda variabile e così via.

`return` Termina l'esecuzione di una funzione e restituisce un valore numerico, specificato come primo argomento, come codice di ritorno della funzione stessa.

`select` Realizza un ciclo infinito per la gestione di un menù di scelte interattivo visualizzato sul terminale dell'utente (standard output).

`shift` Rinomina i parametri passati come argomento allo script o alla funzione con i nomi `$1`, `$2`, ... scartando i primi *n* parametri (se *n* è fornito come argomento) oppure soltanto il primo, se nessun argomento viene fornito al comando.

`source` / `.` Consente di leggere in input e di eseguire lo shell script contenuto nel file il cui nome viene passato come argomento al comando.

`test` / `[` Valuta l'espressione passata come argomento e restituisce il valore *vero* o *falso*.

`times` Stampa il tempo utente ed il tempo macchina impiegato dallo script fino all'esecuzione del comando stesso.

`type` Accetta una stringa come argomento e visualizza in output un messaggio che indica se la stringa corrisponde ad un comando interno (*built-in*) di Bash, un comando esterno, un alias, una parola chiave di Bash, ecc.

`unalias` Annulla l'impostazione di un alias definito in precedenza.

`unset` Elimina le variabili o le funzioni precedentemente definite, specificate come argomento del comando.

`until` Esegue il blocco di istruzioni delimitato dalle parole chiave "`do`" e "`done`" se la condizione logica riportata di seguito risulta falsa.

`wait` Attende che termini l'esecuzione del processo con "process id" o "job id" specificato come argomento del comando.

`while` Esegue il blocco di istruzioni delimitato dalle parole chiave "`do`" e "`done`" se la condizione logica riportata di seguito risulta vera.

Comandi esterni

Di seguito è riportata una descrizione sintetica di alcuni dei principali comandi esterni presenti nella maggior parte dei sistemi operativi UNIX.

`awk` Un interprete di un linguaggio per l'elaborazione di stringhe.

`cc` / `gcc` Compilatore per tradurre dei programmi in formato sorgente scritti in linguaggio C, in programmi eseguibili scritti in linguaggio macchina. Su alcuni sistemi è presente il compilatore GNU GCC (attivabile con comando "`gcc`") il cui funzionamento è grosso modo analogo a quello del compilatore C standard.

`chmod` Modifica i permessi di accesso ad un file, una directory o, ricorsivamente, a tutti i file e le sottodirectory contenute nella directory specificata sulla riga di comando (es.: "`chmod 755 script.sh`").

`clear` Cancella il contenuto dello schermo del terminale dell'utente.

`cp` Copia un file (sorgente) su un altro (target); se il secondo file (il target) non esiste, viene creato, in caso contrario viene sostituito con il contenuto del file sorgente.

`cut` Taglia alcuni "campi" dello standard input o del file specificato sulla linea di comando e li stampa in output.

`date` visualizza in output la data corrente; l'output può essere adattato riportando sulla riga di comando il simbolo "+" seguito dalle opzioni di visualizzazione della data (es.: "`date +%d/%m/%Y`").

`du` Disk Usage, visualizza la dimensione dei file e delle directory contenute nella directory corrente o nella directory specificata sulla linea di comando; con l'opzione "`-k`" viene utilizzata l'unità di misura KByte.

`env` Visualizza o imposta il valore delle variabili d'ambiente con cui viene configurata la shell.

`grep` Cerca le righe che corrispondono ad un determinato pattern nello standard input o nei file specificati sulla riga di comando.

`gzip` Una utility per comprimere file; il file originale viene eliminato lasciando sul filesystem solo la versione compressa, con estensione "`.gz`"; per riportare il file nella sua forma originale, non compressa, si può usare il comando "`gunzip`", equivalente al comando "`gzip -d`".

`head` Visualizza le prime righe della sequenza ricevuta in input o del file specificato sulla riga di comando; con l'opzione "`-`*n*" vengono visualizzate le prime *n* righe.

`ls` Visualizza l'elenco dei file presenti nella directory corrente o nella directory specificata sulla riga di comando.

`mv` Sposta o rinomina un file o una directory.

`rm` Elimina dal filesystem il file specificato.

`rmdir` Elimina dal filesystem la directory specificata.

`sed` Stream editor, consente di manipolare stringhe utilizzando espressioni regolari.

`seq` Visualizza una sequenza di numeri interi.

`sleep` Introduce una pausa di *n* secondi; *n* è un numero intero maggiore di zero specificato sulla riga di comando.

`sort` Restituisce in output ordinate le righe fornite in input o presenti nel file specificato sulla riga di comando.

`tail` Visualizza le ultime righe della sequenza ricevuta in input o del file specificato sulla riga di comando; con l'opzione "`-`*n*" vengono visualizzate le ultime *n* righe, mentre con l'opzione "`-f`" viene mantenuto aperto il canale di input visualizzando continuamente le ultime righe ricevute in input (utile ad esempio per controllare un file di log attivo).

`touch` Crea un file vuoto con il nome specificato sulla riga di comando se non esiste, altrimenti aggiorna la data e l'ora di ultima modifica del file con l'ora corrente.

`uniq` Visualizza su standard output l'input acquisito da un file specificato sulla riga di comando o ricevuto da standard input, eliminando le righe ripetute.

`wc` Esegue il conteggio dei caratteri, delle parole e delle righe fornite in input o presenti nel file specificato sulla riga di comando; con l'opzione "`-l`" conta le linee.

`xargs` Esegue il comando riportato sulla command line utilizzando le stringhe ricevute su standard input come parametri del comando.

Bibliografia

[1] G. Anderson, P. Anderson, *The Unix C shell field guide*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986

[2] S. R. Bourne, *Unix System V*, Addison-Wesley, Milano, 1990

[3] B. Fox, *Bash Features*, Free Software Foundation, Maggio 1994 (disponibile su Internet all'indirizzo http://www.cs.utah.edu/dept/old/texinfo/bash/ features_toc.html)

[4] M. Liverani, *Introduzione al Perl*, Settembre 1996 (disponibile su Internet all'indirizzo http://www.aquilante.net/perl/)

[5] M. Liverani, *UNIX: introduzione elementare*, seconda edizione Settembre 2005 (disponibile su Internet all'indirizzo http://www.aquilante.net/unix/)

[6] C. Ramey, *Bash – The GNU shell* (disponibile su Internet all'indirizzo http://tiswww.case.edu/php/chet/bash/article.pdf)

Indice analitico