POLITECNICO DI BARI
CORSO DI LAUREA IN INGEGNERIA ELETTRONICA
CORSO DI SISTEMI OPERATIVI
ANNO ACCADEMICO 1998 / 99

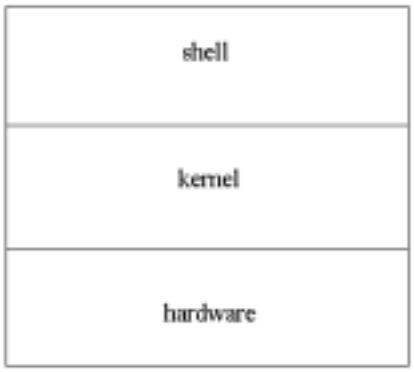
La shell BASH

Stefano Coppi, Marco Fago, Giuseppe Gallo

Indice

Introduzione

La *shell* è lo strato più esterno di un sistema operativo. Essa fa da interfaccia tra l'utente e il kernel.



Gli sviluppatori di Unix decisero di separare la shell dal resto del sistema operativo, infatti la maggior parte dei sistemi operativi di quell ' epoca aveva la shell inserita nel s.o. e di conseguenza non personalizzabile da parte dell'utente.
Alla prima shell di Unix venne dato il nome di shell Bourne, dal cognome del suo autore. La scelta di separare la shell dal s.o. si rivelò vincente perché con il passare del tempo apparvero altre shell che miglioravano quella originale :
gli sviluppatori dell' università di Berkeley decisero di rendere la shell più simile ad un linguaggio di programmazione ed inserire alcune funzioni del linguaggio C : nacque la shell C , abbreviata in csh.
Successivamente il progetto GNU decise di avere bisogno di una shell che fosse priva di restrizioni di royalty, visto che la shell Bourne era proprietaria della AT&T. Così nacque una nuova shell che doveva essere compatibile con la shell originaria Bourne ma ne estendeva le possibilità introducendo la programmazione; questa nuova shell venne denominata BASH ossia Bourne Again Shell.
Nel seguito parleremo proprio della shell Bash .
Dopo aver effetuato il login, appare il prompt di BASH :

```
[nome_utente@nome_host]$
```

A questo punto BASH è pronta a ricevere i comandi.
Ci sono alcuni tasti molto comodi durante la digitazione dei comandi :

- il tasto FRECCIA GIU richiama a video il precedente comando; premendolo più volte si può ripercorrere all'indietro la lista dei comandi inseriti (history list).
- il tasto FRECCIA SU avanza di un comando nella history list.
- mediante il tasto TAB la shell è in grado di completare il nome del file che si sta scrivendo : ad esempio se esiste un file di nome `pippo` si può digitare `pip` e premere TAB : la shell completerà automaticamente il nome del file.

Adesso esamineremo i principali comandi, divisi per categoria.

Documentazione in linea

man

Sintassi:

```
man [<numero-di-sezione>][-f][-h][-k][nome]
```

Opzione	Descrizione
<numero-di-sezioni>	Se prima del nome del comando o dell'argomento appare un numero, si intende che si vuole ottenere la pagina di manuale da una sezione determinata.
-f	Cerca le stringhe all'interno di un set di file di database contenenti una breve descrizione dei comandi di sistema. Il risultato della ricerca viene emesso attraverso lo *standard output*. Sono visualizzate solo le corrispondenze con parole intere.
-h	Visualizza una breve guida di se stesso.
-k	Cerca le stringhe all'interno di un set di file di database contenenti una breve descrizione dei comandi di sistema. Il risultato della ricerca viene emesso attraverso lo *standard output*. Sono visualizzate tutte le corrispondenze.

Formatta ed emette attraverso lo standard output la pagina di manuale indicata dal nome.
Lo scorrimento del testo che compone le pagine di manuale indicate negli argomenti, viene fatto attraverso un programma esterno, richiamato automaticamente da man. Solitamente si tratta di *more* o di *less*. Di conseguenza, i comandi per lo scorrimento del testo dipendono dal tipo di programma utilizzato. Se si tratta di uno di questi due appena citati, sono sempre validi almeno quelli riportati.

Esempi:
volendo visualizzare la pagina di manuale relativa al comando *ls:*

```
$ man ls
```

volendo visualizzare la pagina di manuale della sezione 8 relativa al programma *lilo:*

```
$ man 8 lilo
```

Gestione delle directory

pwd

Sintassi:

```
pwd
```

PWD sta per Print Working Directory : visualizza il path completo della directory corrente.

cd

Sintassi:

```
cd [directory]
```

Cambia la directory corrente, che diventa quella specificata. Se non viene specificata la directory dopo cd (le parentesi quadre indicano che si tratta di un argomento opzionale), la directory corrente diventa la home directory dell'utente.
Cd – serve per tornare alla directory precedente.
Quando si crea una directory, Unix aggiunge automaticamente due nomi di file alla directory : . e .. che indicano rispettivamente la directory corrente e la directory padre di quella corrente. Il comando cd .. consente di ritornare alla directory padre.

Esempi:

`cd /usr/bin`	la directory /usr/bin diventa la directory corrente
`cd -`	torna alla directory precedente
`cd ..`	torna alla directory padre

ls

Sintassi:

```
ls [-a] [-F] [-l] [-r] [-t] [-u] [percorsi]
```

Visualizza nomi e informazioni sui file contenuti nella directory corrente o in quella specificata come opzione.

Opzione	**Funzione**
-a	Include nell'elenco tutti i files e le directory, compresi quelli nascosti , il cui nome inizia con un punto.
-F	Aggiunge ai nomi dei files un carattere che ne indica il tipo: * per i file eseguibili, / per le directory, @ per i link simbolici.
-l	Visualizza un elenco con informazioni dettagliate tra cui il proprietario, i permessi, la data di modifica, la dimensione etc.
-r	Inverte l'ordinamento(per default viene assunto l'ordine alfabetico).
-t	Ordina i file per data di ultima modifica
-u	Ordina i file per data di ultimo accesso

Si possono elencare tutti i file che hanno il nome che inizia con un particolare carattere : per esempio per visualizzare tutti i file di usr/bin i cui nomi iniziano con r scrivere:

```
ls /usr/bin/r*
```

esempio:

```
ls –l
```

```
permessi di accesso   proprietario  gruppo     dimensione    data            nome file
-rw-r--r--   1 root     root          1158 Apr 26 20:53 CollezioneArticoli.cc
-rw-r--r--   1 root     root           770 Apr 26 19:59 CollezioneArticoli.h
-rw-r--r--   1 root     root          1424 Apr 26 20:54 CollezioneArticoli.o
-rw-r--r--   1 root     root           632 Apr 26 19:54 GestioneEmeroteca.bak
-rw-r--r--   1 root     root           622 Apr 26 20:11 GestioneEmeroteca.cc
-rw-r--r--   1 root     root           442 Apr 26 19:40 GestioneEmeroteca.h
-rw-r--r--   1 root     root          1384 Apr 26 20:11 GestioneEmeroteca.o
```

mkdir

Sintassi :

```
mkdir [-m modo] directories
```

Crea una nuova directory , consentendo di definire i permessi di accesso alle directory in fase di creazione.

Opzioni:

`-m modo` imposta i permessi delle directory con modo. Se si omette questa opzione i permessi di default sono di tipo 755.

In Unix ogni file o directory ha i seguenti permessi di accesso :

permesso di accesso	**Significato per una directory**	**Per un file**
R - lettura	Permette di accedere ai files in essa contenuti.	Permette di leggere il file.
W – scrittura	Pemettere di creare, cancellare , modificare i files in essa contenuti.	Permette di modificare il contenuto del file.
X – esecuzione	Permette di attraversare la directory.	Permette di eseguire il file, se è un file eseguibile.

I permessi sono associati ,rispettivamente , al proprietario del file (user) , al gruppo (group) ed agli altri utenti (others).

Ad ogni file o directory è associato un campo di 9 bit contenente i permessi :

proprietario			**gruppo**			**altri**		
Read	*Write*	*Execute*	*Read*	*Write*	*Execute*	*Read*	*Write*	*Execute*
1	1	1	0	0	1	0	0	1

Nell' esempio il proprietario ha il premesso di lettura, scrittura ed esecuzione mentre i membri del gruppo e gli altri hanno solo il permesso di esecuzione.

Un insieme di 3 bit si può rappresentare mediante una cifra ottale; quindi per rappresentare la maschera di 9 bit dei permessi sono necessarie 3 cifre ottali. Per l'esempio in figura i permessi sono codificati dal numero ottale 711 quindi dovremmo usare l'opzione –m 711.

Esiste un formato simbolico per specificare i permessi :
per settare ciascun bit si usa una stringa di 3 simboli , ognuno dei quali è opzionale :
1. *tipo di proprietà* : **u** per l'utente, **g** per il gruppo e **a** per tutti gli altri.
2. *Operatore* : **+** per aggiungerre un permesso , - per toglierlo
3. *Tipo di permesso* : **r** per lettura, **w** per scrittura , **x** per esecuzione

Se nell'esempio precedente volessimo consentire l'accesso in lettura per il gruppo : g+r

Esempio:
Supponiamo di voler creare nella propria home directory due directory di nome "Cibo_buono" e "Cibo_cattivo". Si vuole concedere a tutti il permesso di lettura ed esecuzione (ricerca) nelle subdirectory, ma riservare il permesso di scrittura a sé stesso e al proprio gruppo di lavoro, il comando sarà :

```
mkdir –m 775 ~/Cibo_buono ~/Cibo_cattivo
```

rmdir

Sintassi:

```
rmdir directories
```

Rimuove le directories specificate. Le directories devono essere vuote.

Gestione dei file

cp

Sintassi:

```
cp [-f] [-i] [-p] [-R] [-v] file_orig file_dest
cp [-f] [-i] [-p] [-R] [-v] file_orig directory_dest
```

Opzione	Funzione
-f	Forza la sovrascrittura dei files destinazione esistenti
-i	Richiede interattivamente se sovrascrivere un file esistente
-p	Mantiene le caratteristiche originali di un file , inclusi il proprietario, i permessi e la data di ultima modifica
-R	Copia ricorsivamente le directory oltre ai file normali
-v	Scrive i nomi dei files man mano che vengono copiati

Nella prima forma serve per creare una copia di un file.
Nella seconda forma serve per copiare uno o più files in un'altra directory.

Esempi:
Si supponga di voler creare il duplicato di un proprio file di nome hello.c :

```
cp hello.c hello.bak
```

il file duplicato si chiamerà hello.bak.

Supponiamo di voler copiare tutti i files sorgenti c dalla directory source alla propria home directory:

```
cp –v  /source/*.c ~/
```

mv

Sintassi:

```
mv [-f] [-i] [-v] file_orig file_dest
mv [-f] [-i] [-v] file_orig directory_dest
```

Opzione	Funzione
-f	Forza la sovrascrittura di un file esistente
-i	Spostamento interattivo : chiede la conferma all'utente di ogni file da spostare.
-v	Scrive i nomi dei files da spostare

Nella prima forma serve per cambiare il nome di un file.
Nella seconda forma serve per spostare uno o più files in un'altra directory.

Esempi:
Supponiamo di voler cambiare il nome di un file da pippo.c a pluto.c :

```
mv pippo.c pluto.c
```

Supponiamo di voler spostare tutti i files contenuti nella directory /pippo in /paperino :

```
mv /pippo/* /paperino
```

rm

Sintassi:

```
rm [-f] [-i] [-r] files
```

Opzione	Funzione
-f	Forza la cancellazione
-i	Per ogni file, richiede interattivamente una conferma prima di cancellarlo.
-r	Rimuove ricorsivamente il contenuto delle subdirectories contenute nella directory corrente.

Cancella i files specificati.

Esempio:
Per cancellare dalla directory corrente tutti i file il cui nome finisce in .bak :

```
rm –i *.bak
```

Per ogni file verrà richiesta la conferma.

ln

Sintassi:

```
ln nomefile nuovonome
ln nomefile directory
```

Crea un link (collegamento) ad un file.

Esempio:
Fred e Lisa stanno lavorando insieme ad un progetto ed ognuno di essi ha spesso bisogno di accedere al file dell'altro. Se Fred ha come directory di lavoro /usr/fred , egli può fare riferimento al file x nella directory di Lisa come /usr/lisa/x. Siccome il pathname assoluto è troppo lungo, per evitare di digitarlo per intero ogni volta, Fred può creare nella sua directory un nuovo elemento x che punta al file /usr/lisa/x; per fare ciò basta digitare:

```
cd /usr/fred
ln /usr/lisa/x x
```

da questo momento in poi Fred può usare semplicemente `x` per fare riferimento a /usr/lisa/x.
Se vediamo il listing della directory /usr/fred possiamo notare come accanto al nome del file x ci sia una freccia che indica il file a cui punta ovvero /usr/lisa/x :

```
-rw-rw-r--   1 fred  team_a   1298 May 12 20:19 x -> /usr/lisa/x
```

file

Sintassi:

```
file files
```

Determina il tipo di uno o più files.

Esempio:
Supponiamo di avere una directory con un certo numero di files, dei quali si vuole sapere il contenuto.Scrivendo :

```
file *
```

si ottiene un risultato di questo tipo:

```
Articolo:   directory
Emeroteca:  directory
Gldemo:     directory
Mlms:       directory
Opengl:     directory
Sorg:       directory
Xrootenv.0: ASCII text
core:       ELF 32-bit LSB core file (signal 4096), Intel 80386, version 1
emerot:     directory
nsmail:     directory
ouput:      ASCII text
output:     ASCII text (with escape sequences)
root:       directory
```

```
temp:       directory
```

chmod

Sintassi:

```
chmod [-R] modo files
```

Opzione	Funzione
-R	Cambia ricorsivamente i permessi delle directory e del loro contenuto
Modo	Permessi di accesso in formato simbolico o ottale.
Files	Lista dei files o directory a cui cambiare i permessi.

Cambia i permessi di accesso ai files.

Esempio:
Per il file pippo.c vogliamo impostare i seguenti permessi : lettura,scrittura per il proprietario e sola lettura per il gruppo e gli altri; in ottale la codifica è 644 quindi :

```
chmod 644 pippo.c
```

chown

Sintassi:

```
chown [-R] [utente][.][gruppo] files
```

Opzione	Funzione
-R	Cambia ricorsivamente il proprietario delle directory e del loro contenuto.
Utente	Nuovo ID utente per i files.
Gruppo	Nuovo ID gruppo per i files.

Ogni file è associato ad un proprietario e ad un gruppo. Con questo comando è possibile cambiare l'utente , il gruppo (usando .group) o entrambi.

Esempio:
Vogliamo assegnare tutti i files nella directory /local all'utente di nome pippo ,del gruppo disney:

```
chown –R pippo.disney  /local
```

chgrp

Sintassi:

```
chgrp [-R] gruppo files
```

Opzione	Funzione
-R	Cambia ricorsivamente il proprietario delle directory e del loro contenuto.
Gruppo	Nuovo ID gruppo per i files.

Consente di cambiare il gruppo di appartenenza di un file.

cat

Sintassi:

```
cat [<file>...]
```

Concatena dei file e ne emette il contenuto attraverso lo *standard output*. Il comando emette di seguito i file indicati come argomento attraverso lo *standard output* (sullo schermo), in pratica qualcosa di simile al comando `TYPE` del Dos. Se non viene fornito il nome di alcun file, viene utilizzato lo *standard input.*

Esempi:

```
$ cat pippo pluto
```

Mostra il contenuto di *pippo* e *pluto.*

```
$ cat pippo pluto > paperino
```

Genera il file *paperino* come risultato del concatenamento in sequenza di *pippo* e *pluto.*

more e less

Sintassi:

```
more [<opzione>][<file>]...

less [<opzioni>][<file>]...
```

Questi due programmi sono utilizzati per lo scorrimento a video di un file di testo; le funzionalità essenziali di questi due comandi sono simili per cui verranno descritti insieme.
Nell'utilizzo normale non vengono fornite opzioni e se non viene indicato alcun file negli argomenti, viene fatto lo scorrimento di quanto ottenuto dallo standard input.

Una volta avviato uno di questi due programmi, lo scorrimento del testo dei file da visualizzare avviene per mezzo di comandi impartiti attraverso la pressione di tasti. Il meccanismo è simile a quello del programma `EDIT` del Dos: alcuni comandi richiedono semplicemente la pressione di uno o più tasti in sequenza; altri richiedono un argomento e in questo caso, la digitazione appare nell'ultima riga dello schermo o della finestra a disposizione.

Comando	**Descrizione**
h	Richiama una breve guida dei comandi disponibili.
H	Come *h*.
Spazio	Scorre il testo in avanti di una schermata.
Invio	Scorre il testo in avanti di una riga alla volta.
B	Quando possibile, scorre il testo all'indietro di una schermata.
/<*modello*>	Esegue una ricerca in avanti, in base all'espressione regolare indicata.
n	Ripete l'ultimo comando di ricerca.
Ctrl+l	Ripete la visualizzazione della schermata attuale.
q	Termina l'esecuzione del programma.
Q	Come *q*.

La differenza fondamentale tra questi due programmi sta nella possibilità da parte di less di scorrere il testo all'indietro anche quando questo proviene dallo standard input, mentre per more non è possibile.

less permette inoltre di utilizzare i tasti freccia e i tasti pagina per lo scorrimento del testo, e di effettuare ricerche all'indietro.

Comando	**Descrizione**
y	Scorre il testo all'indietro di una riga alla volta.
?<*modello*>	Esegue una ricerca all'indietro, in base all'espressione regolare indicata.
N	Ripete l'ultimo comando di ricerca nella direzione inversa.

Esempi:

```
$ ls -l | more

$ ls -l | less
```

Scorre sullo schermo l'elenco del contenuto della directory corrente che probabilmente è troppo lungo per essere visualizzato senza l'aiuto di uno tra questi due programmi.

```
$ more README

$ less README
```

Scorre sullo schermo il contenuto del file README.

sort

Sintassi:

```
sort [-c][-m][-b][-d][-f][-i][-n][-r][-o <file>][-u]
```

Opzione	**Descrizione**
-c	Controlla che i file indicati siano già ordinati; se non losono, viene emessa una segnalazione di errore e il programma termina restituendo valore 1.
-m	Fonde insieme i file indicati che devono essere già stati ordinati in precedenza. Nel caso non lo siano, si può sempre usare la modalità di ordinamento normale. L'utilizzo di questa opzione fa risparmiare tempo quando la situazione lo consente.
-b	Ignora gli spazi bianchi iniziali.
-d	Durante l'ordinamento ignora tutti i caratteri che non siano lettere, numeri e spazi.
-f	Non distingue tra maiuscole e minuscole.
-i	Ignora i caratteri speciali al di fuori del set ASCII puro (da 0x30 a 0x7E compresi).
-n	Esegue una comparazione, o un ordinamento, numerico, tenendo conto anche del segno meno e del punto decimale.
-r	Inverte l'ordine della comparazione o dell'ordinamento.
-o <file>	Invece di utilizzare lo standard output per emettere il risultato dell'ordinamento o della fusione, utilizza il file indicato come argomento di questa opzione.

Permette di riordinare o fondere insieme (*merge*) il contenuto dei file.

Ricerche

Esistono due tipi di ricerche possibili: quelle all'interno dei file per identificare delle stringhe che corrispondono a un modello e quelle fatte all'interno di un *filesystem* alla ricerca di file o directory in base alle loro caratteristiche (nome, date e altri attributi).

Per questo si utilizzano due programmi fondamentali: **grep** per le ricerche di stringhe e **find** per la ricerca dei file.

grep

Sintassi:

```
grep [-G][-E][-F][-i][-n][-c][-h][-l][-L][-v] <modello> [<file>...]

grep [-G][-E][-F][-i][-n][-c][-h][-l][-L][-v] [–e <modello>] [<file>...]
```

Opzione	**Descrizione**
-G	Utilizza un’espressione regolare elementare (comportamento predefinito).
-E	Utilizza un'espressione regolare estesa.
-F	Utilizza un modello fatto di stringhe fisse.
-e <modello>	Specifica il modello di ricerca.
-i	Ignora la differenza tra maiuscole e minuscole.
-n	Aggiunge il numero di riga.
-c	Emette solo il totale delle righe corrispondenti per ogni file.
-h	Elimina l'intestazione normale per le ricerche su più file.
-l	Emette solo i nomi dei file per i quali la ricerca ha avuto successo.
-L	Emette solo i nomi dei file per i quali la ricerca non ha avuto successo.
-v	Inverte il senso della ricerca: valgono le righe che non corrispondono.

Esegue una ricerca all'interno dei file indicati come argomento oppure all'interno dello standard input. Il modello di ricerca può essere semplicemente il primo degli argomenti che seguono le opzioni, oppure può essere indicato precisamente come argomento dell'opzione -e.

Espressioni regolari

Un'espressione regolare è un modello che descrive un insieme di stringhe. Le espressioni regolari sono costruite, in maniera analoga alle espressioni matematiche, combinando espressioni più brevi.

Per tradizione esistono due tipi di espressioni regolari: elementari ed estese.

Espressioni regolari estese

Il punto di partenza sono le espressioni regolari con le quali si ottiene una corrispondenza con un singolo carattere. La maggior parte dei caratteri, includendo tutte le lettere e le cifre numeriche, sono espressioni regolari che corrispondono a loro stessi. Ogni metacarattere con significati speciali può essere utilizzato per il suo valore normale facendolo precedere dalla barra obliqua inversa (\).

Una fila di caratteri racchiusa tra parentesi quadre corrisponde a un carattere qualunque tra quelli indicati; se all'inizio di questa fila c'è l'accento circonflesso, si ottiene una corrispondenza con un carattere qualunque diverso da quelli della fila. Per esempio, l'espressione regolare **[0123456789]** corrisponde a una qualunque cifra numerica, mentre **[^0123456789]** corrisponde a un carattere qualunque purché non sia una cifra numerica.

All'interno delle parentesi quadre, invece che indicare un insieme di caratteri, è possibile indicarne un intervallo mettendo il carattere iniziale e finale separati da un trattino (-). I caratteri che vengono rappresentati in questo modo dipendono dalla codifica ASCII che ne determina la sequenza. Per

esempio, l'espressione regolare **[9-A]** rappresenta un carattere qualsiasi tra: **9**, **:**, **;**, <, =, >, **?**, **@** e **A**, perché così è la sequenza ASCII.

All'interno delle parentesi quadre si possono indicare delle classi di caratteri attraverso il loro nome: **[:alnum:]**, **[:alpha:]**, **[:cntrl:]**, **[:digit:]**, **[:graph:]**, **[:lower:]**, **[:print:]**, **[:punct:]**, **[:space:]**, **[:upper:]** e **[:xdigit:]**. Per essere usati, questi nomi di classi possono solo apparire all'interno di un'espressione tra parentesi quadre, di conseguenza, per esprimere la corrispondenza con un qualunque carattere alfanumerico si può utilizzare l'espressione regolare **[[:alnum:]]**. Nello stesso modo, per esprimere la corrispondenza con un qualunque carattere non alfanumerico si può utilizzare l'espressione regolare **[^[:alnum:]]**.

Il punto (**.**) corrisponde a un qualsiasi carattere. Il simbolo **\w** è un sinonimo di **[[:alnum:]]** e **\W** è un sinonimo di **[^[:alnum:]]**.

L'accento circonflesso (^) e il dollaro (**$**) sono metacaratteri che corrispondono rispettivamente alla stringa nulla all'inizio e alla fine di una riga. I simboli \< e \> corrispondono rispettivamente alla stringa vuota all'inizio e alla fine di una parola. Un'espressione regolare che genera una corrispondenza con un singolo carattere può essere seguita da uno o più operatori di ripetizione. Questi sono rappresentati attraverso i simboli **?**, *****, **+** e dai "contenitori" rappresentati da particolari espressioni tra parentesi graffe.

Codifica	**Corrispondenza**
*x**	Nessuna o più volte x. Equivalente a *x{0,}*.
x?	Nessuna o al massimo una volta *x*. Equivalente a *x{0,1}*.
x+	Una o più volte *x*. Equivalente a *x{1,}*.
x{n}	Esattamente *n* volte *x*.
x{n,}	Almeno *n* volte x.
x{n,m}	Da *n* a *m* volte *x*.

Due espressioni regolari possono essere concatenate (in sequenza) generando un'espressione regolare corrispondente alla sequenza di due sottostringhe che rispettivamente corrispondono alle due sottoespressioni.

Due espressioni regolari possono essere unite attraverso l'operatore |; l'espressione regolare risultante corrisponde a una qualunque stringa per la quale sia valida la corrispondenza di una delle due sottoespressioni.

La ripetizione (attraverso gli operatori di ripetizione) ha la precedenza sul concatenamento che a sua volta ha la precedenza sull'alternanza (quella che si ha utilizzando l'operatore |). Una sottoespressione può essere racchiusa tra parentesi tonde per modificare queste regole di precedenza.

La notazione **\n**, dove n è una singola cifra numerica diversa da zero, rappresenta un riferimento all'indietro a una corrispondenza già avvenuta tra quelle di una sottoespressione precedente racchiusa tra parentesi tonde. La cifra numerica indica l'n-esima sottoespressione tra parentesi a partire da sinistra.

Esempi:

```
$ grep –F –e ciao –i –n *
```

Cerca all'interno di tutti i file contenuti nella directory corrente la corrispondenza della parola ciao senza considerare la differenza tra le lettere maiuscole e quelle minuscole. Visualizza il numero e il contenuto delle righe che contengono la parola cercata.

```
$ grep -E -e "scal[oa]" elenco
```

Cerca all'interno del file elenco le righe contenenti la parola scalo o scala.

```
$ grep -E -e \`.*\' elenco
```

Questo è un caso di ricerca particolare in cui si vogliono cercare le righe in cui appare qualcosa racchiuso tra apici singoli, nel modo `...'. Si immagina però di utilizzare la shell bash con la quale è necessario proteggere gli apici da un altro tipo di interpretazione. In questo caso bash fornisce a grep solo la stringa `.*'.

```
$ grep -E -e "\`.*\'" elenco
```

Questo esempio deriva dal precedente. Anche in questo caso si suppone di utilizzare la shell bash, ma questa volta bash fornisce a grep la stringa `.*\' che fortunatamente viene ugualmente interpretata da grep nel modo corretto.

find

Sintassi:

```
find [<percorso>...][<espressione>]
```

Esegue una ricerca all'interno dei percorsi indicati per i file che soddisfano l'espressione di ricerca. Il primo argomento che inizia con -, (,), , o ! (trattino, parentesi tonda, virgola, punto esclamativo) viene considerato come l'inizio dell'espressione, mentre gli argomenti precedenti sono interpretati come parte dell'insieme dei percorsi di ricerca.

Se non vengono specificati percorsi di ricerca, si intende la directory corrente; se non viene specificata alcuna espressione o semplicemente se non viene specificato nulla in contrario, viene emesso l'elenco dei nomi trovati.

Espressioni di find

Il concetto di espressione nella documentazione di find è piuttosto ampio e bisogna fare un po' di attenzione. Si può scomporre idealmente in

[<opzione>...] [<condizioni>]

e a sua volta le condizioni possono essere di due tipi: test e azioni. Ma, mentre le opzioni devono apparire prima, test e azioni possono essere mescolati tra loro.

Le opzioni rappresentano un modo di configurare il funzionamento del programma, così come di solito accade nei programmi di utilità. Le condizioni sono espressioni che generano un risultato logico e come tali vanno trattate: per concatenare insieme più condizioni occorre utilizzare gli operatori booleani.
Di seguito si riportano le opzioni e le condizioni usate più importanti.

Opzione	**Descrizione**
-depth	Elabora prima il contenuto delle directory. In pratica si ottiene una scansione che parte dal livello più profondo fino al più esterno.
-xdev | -mount	Non esegue la ricerca all'interno delle directory contenute all'interno di filesystem differenti da quello di partenza. Tra i due è preferibile usare *-xdev*.

Esempi:

```
$ find . –xdev –print
```

Elenca tutti i file e le directory a partire dalla posizione corrente restando nell'ambito del *filesystem* di partenza.

Condizione	**Descrizione**
-user <*nome-dell'utente*>	Si avvera quando il file o la directory appartiene all'utente indicato.
-nouser	Si avvera per i file e le directory di proprietà di utenti non esistenti.
-group <*nome-del-gruppo*>	Si avvera quando il file o la directory appartiene al gruppo indicato.
-nogrop	Si avvera per i file e le directory di proprietà di gruppi non esistenti.
-perm <*permessi*>	Si avvera quando i permessi del file o della directory corrispondono esattamente a quelli indicati con questo test. I permessi si possono indicare in modo numerico (ottale) o simbolico.
-perm -<*permessi*>	Si avvera quando i permessi del file o della directory comprendono almeno quelli indicati con questo test.
-perm +<*permessi*>	Si avvera quando alcuni dei permessi indicati nel modello di questo test corrispondono a quelli del file o della directory.
-name <*modello*>	Si avvera quando viene incontrato un nome di file o directory corrispondente al modello

	indicato, all'interno del quale si possono utilizzare i caratteri jolly. La comparazione avviene utilizzando solo il nome del file (o della directory) escludendo il percorso precedente. I caratteri jolly (*, ?, [,]) non possono corrispondere al punto iniziale (.) che appare nei cosiddetti file nascosti.
-iname <*modello*>	Come *–name* ma ignora la differenza fra maiuscole e minuscole.
-lname <*modello*>	Si avvera quando si tratta di un collegamento simbolico e il suo contenuto corrisponde al modello che può essere espresso utilizzando anche i caratteri jolly. Un collegamento simbolico può contenere anche l'indicazione del percorso necessario a raggiungere un file o una directory reale. Il modello espresso attraverso i caratteri jolly non tiene conto in modo particolare dei simboli punto (.) e barra obliqua (/) che possono essere contenuti all'interno del collegamento.
-ilname <*modello*>	Come *–lname* ma ignora la differenza fra maiuscole e minuscole.
-path <*modello*>	Si avvera quando il modello, esprimibile utilizzando caratteri jolly, corrisponde a un percorso. Per esempio, un modello del tipo ./i*no può corrispondere al file *./idrogeno/ossigeno.*
-ipath <*modello*>	Come *–ipath* ma ignora la differenza fra maiuscole e minuscole.
-mmin *n* -amin *n* -cmin *n*	Si avvera quando la data di modifica del file o della directory corrisponde a n minuti fa. Si avvera quando la data di accesso del file o della directory corrisponde a n minuti fa. Si avvera quando la data di creazione del file o della directory corrisponde a n minuti fa.
-mtime *n* -atime *n* -ctime *n*	Si avvera quando la data di modifica del file o della directory corrisponde a n giorni fa. Si avvera quando la data di accesso del file o della directory corrisponde a n giorni fa. Si avvera quando la data di creazione del file o della directory corrisponde a n giorni fa. Più precisamente, il valore *n* fa riferimento a multipli di 24 ore.
-newer <*file*> -anewer <*file*> -cnewer <*file*>	Si avvera quando la data di modifica (accesso o creazione) del file o della directory è più recente di quella del file indicato.
-empty	Si avvera quando il file o la directory sono vuoti.

Oltre a queste opzioni è possibile specificare alcune azioni da compiere per ogni file o directory che si ottiene per la scansione.

Azione	**Descrizione**
-exec <*comando*>*;*	Esegue il comando indicato, nella directory di partenza, restituendo il valore Vero se il comando restituisce il valore zero. Tutti gli argomenti che seguono vengono considerati come parte del comando fino a quando viene incontrato il simbolo punto e virgola (**;**). All'interno del comando, la stringa {} viene interpretata come sinonimo del file attualmente in corso di elaborazione.
-ok <*comando*>	Si comporta come *–exec* ma, prima di eseguire il comando, chiede conferma all'utente.
-print	Emette il nome completo del file (e delle directory) che avvera le condizioni specificate.

Esempi:

```
$ find / -name "lib*" -print
```

Esegue una ricerca su tutto il *filesystem* globale, a partire dalla directory radice, per i file e le directory il cui nome inizia per lib. Dal momento che si vuole evitare che la *shell* trasformi lib* in qualcosa di diverso, si utilizzano le doppie virgolette.

```
# find / -xdev -nouser -print
```

Esegue una ricerca nel filesystem root a partire dalla directory radice, escludendo gli altri *filesystem*, per i file e le directory appartenenti a utenti non registrati (privi di account).

```
$ find /usr -xdev -atime +90 -print
```

Esegue una ricerca a partire dalla directory */usr/*, escludendo altri *filesystem* diversi da quello di partenza, per i file la cui data di accesso è più vecchia di 2160 ore (24*90=2160).

```
$ find / -xdev -type f -name core -print
```

Esegue una ricerca a partire dalla directory radice, all'interno del solo *filesystem root*, per i file core (solo i file normali).

```
$ find / -xdev -size +5000k -print
```

Esegue una ricerca a partire dalla directory radice, all'interno del solo *filesystem root*, per i file la cui dimensione supera i 5000 Kbyte.

```
$ find ~/dati -atime +90 -exec mv \{\} ~/archivio \;
```

Esegue una ricerca a partire dalla directory *~/dati/* per i file più vecchi di 90 giorni e sposta quei file all'interno della directory *~/archivio/*. Il tipo di *shell* a disposizione ha costretto a usare spesso il carattere di escape (\) per poter usare le parentesi graffe e il punto e virgola secondo il significato che gli attribuisce find e non la shell stessa.

Exit status o valore restituito dai comandi

Un comando che termina la sua esecuzione restituisce un valore, così come fanno le funzioni nei linguaggi di programmazione. Un comando, che quindi può essere un comando interno, una funzione di shell o un programma, può restituire solo un valore numerico. Di solito, si considera uno stato di uscita pari a zero come indice di una conclusione regolare del comando, cioè senza errori di alcun genere.

Dal momento che può essere restituito solo un valore numerico, quando il risultato di un'esecuzione di un comando viene utilizzato in un'espressione logica (booleana), si considera lo zero come equivalente a Vero, mentre un qualunque altro valore viene considerato equivalente a Falso.

In casi particolari è bash ad assegnare i valori di uscita di un comando:

- quando un programma viene interrotto a causa di un segnale (di interruzione), il suo valore di uscita è pari a 128 più il valore di quel segnale;

- quando un programma non viene trovato e quindi non può essere avviato, il suo valore di uscita è 127;

-quando un presunto programma viene trovato, ma non risulta eseguibile, il suo valore di uscita è 126.

Per conto suo, bash restituisce il valore di uscita dell'ultimo comando eseguito, se non riscontra un errore di sintassi, nel qual caso genera un valore diverso da zero (Falso).

Pipeline

La pipeline è una sequenza di uno o più comandi separati dal simbolo pipe, ovvero la barra verticale (|). Il formato normale per una pipeline è il seguente:

```
[!] <comando1> [ | <comando2>...]
```

Lo *standard output* del primo comando è diretto allo *standard input* del secondo comando. Questa connessione è effettuata prima di qualsiasi redirezione specificata dal comando. Come si vede dalla sintassi, per poter parlare di pipeline basta anche un solo comando. Normalmente, il valore restituito dalla pipeline corrisponde a quello dell'ultimo comando che viene eseguito all'interno di questa. Se all'inizio della pipeline viene posto un punto esclamativo (!), il valore restituito corrisponde alla negazione logica del risultato normale. La shell attende che tutti i comandi della pipeline siano terminati prima di restituire un valore. Ogni comando in una pipeline è eseguito come un processo separato (cioè, in una *subshell*).

Lista di comandi

La lista di comandi è una sequenza di una o più pipeline separate da **;**, **&**, **&&** o ||, e terminata da **;**,**&** o dal codice di interruzione di riga. Parti della lista sono raggruppabili attraverso parentesi (tonde o graffe) per controllarne la sequenza di esecuzione. Il valore di uscita della lista corrisponde a quello dell'ultimo comando della stessa lista che ha potuto essere eseguito.

Separatore di comandi “;”

I comandi separati da un punto e virgola (**;**) sono eseguiti sequenzialmente. Il simbolo punto e virgola può essere utilizzato per separare una serie di comandi posti sulla stessa riga, o per terminare una lista di comandi quando c'è la necessità di farlo (per distinguerlo dall'inizio di qualcos'altro). Idealmente, il punto e virgola sostituisce il codice di interruzione di riga.

Esempi

```
# ./config ; make ; make install
```

Avvia in sequenza una serie di comandi per la compilazione e installazione di un programma ipotetico.

```
$ echo "uno" ; echo "due"

$ echo "uno" ; echo "due" ;
```

I due comandi sono equivalenti: nel secondo la lista viene conclusa con un punto e virgola, ma ciò non produce alcuna differenza di comportamento.

Di seguito si vedono due spezzoni di script equivalenti: nel secondo si sostituisce il punto e virgola con un codice di interruzione di riga, dato che il contesto lo consente.

```
$ ls ; echo "Ciao a tutti"

$  ls
$  echo "Ciao a tutti"
```

Operatore di controllo “&&”

L'operatore di controllo && si comporta come l'operatore booleano AND: se il valore di uscita di ciò che sta alla sinistra è zero (Vero), viene eseguito anche quanto sta alla destra. Dal punto di vista pratico, viene eseguito il secondo comando solo se il primo ha terminato il suo compito con successo.

Esempi

```
$ mkdir ./prova && echo "Creata la directory prova"
```

Viene eseguito il comando mkdir ./prova. Se ha successo viene eseguito il comando successivo che visualizza un messaggio di conferma.

Operatore di controllo “||”

L'operatore di controllo || si comporta come l'operatore booleano OR: se il valore di uscita di ciò sta alla sinistra è zero (Vero), il comando alla destra non viene eseguito. Dal punto di vista pratico, viene eseguito il secondo comando solo se il primo non ha potuto essere eseguito, oppure se ha terminato il suo compito riportando un qualche tipo di insuccesso.

Esempi

```
$ mkdir ./prova || mkdir ./prova1
```

Si tenta di creare la directory *prova/*, se il comando fallisce si tenta di creare *prova1/* al suo posto.

Alias

Attraverso i comandi interni **alias** e **unalias** è possibile definire ed eliminare degli alias, ovvero dei sostituti ai comandi. Prima di eseguire un comando di qualunque tipo, bash cerca la prima parola di questo comando (quello che lo identifica) all'interno dell'elenco degli alias e, se la trova lì, la sostituisce con il suo alias. La sostituzione non

avviene se il comando o la prima parola di questo è protetta attraverso il **quoting**, ovvero se è tra virgolette. Il nome dell'alias non può contenere il simbolo =. La trasformazione in base alla presenza di un alias continua anche per la prima parola del testo di rimpiazzo della prima sostituzione. Quindi, un alias può fare riferimento a un altro alias e così di seguito. Questo ciclo si ferma quando non ci sono più corrispondenze con nuovi alias in modo da evitare una ricorsione infinita.
Se l'ultimo carattere del testo di rimpiazzo dell'alias è uno spazio o una tabulazione, allora anche la parola successiva viene controllata per una possibile sostituzione attraverso gli alias. Non c'è modo di utilizzare argomenti attraverso gli alias. Gli alias non vengono espansi quando la shell non funziona in modalità interattiva; di conseguenza, non sono disponibili durante l'esecuzione di uno script.

In generale, l'utilizzo di alias è superato dall'uso delle funzioni.

Esempi

```
# alias rm="rm -i"
```

Crea un alias al comando (programma) *rm* in modo che venga eseguito automaticamente con l'opzione -i che implica la richiesta di conferma per ogni file che si intende cancellare.

```
# alias cp="cp -i"
```

Crea un alias al comando (programma) *cp* in modo che venga eseguito automaticamente con l'opzione -i che implica la richiesta di conferma per ogni file che si intende eventualmente sovrascrivere.

Redirezione

Prima che un comando sia eseguito, si può ridirigere il suo input e il suo output utilizzando una speciale notazione interpretata dalla shell. La redirezione viene eseguita, nell'ordine in cui appare, a partire da sinistra verso destra.

Se si utilizza il simbolo < da solo, la redirezione si riferisce allo *standard input* (corrispondente al descrittore di file 0). Se si utilizza il simbolo > da solo, la redirezione si riferisce allo *standard output* (corrispondente al descrittore di file 1). La parola che segue l'operatore di redirezione è sottoposta a tutta la serie di espansioni e sostituzioni possibili. Se questa parola si espande in più parole, bash emette un errore.
Si distinguono tre tipi di descrittori di file per l'input e l'output:

0 = *standard input*;

1 = *standard output*;

2 = *standard error*.

Redirezione dell'input

```
[n]< <parola>
```

La redirezione dell'input fa sì che il file il cui nome risulta dall'espansione della parola alla destra del simbolo < venga letto e inviato al descrittore di file *n*, oppure, se non indicato, allo standard input pari al descrittore di file 0.

Esempi

```
$ sort < ./elenco
```

Emette il contenuto del file elenco (che si trova nella directory corrente) riordinando le righe.**sort** riceve il file da ordinare dallo *standard input*.

```
$ sort 0< ./elenco
```

Esegue la stessa cosa dell'esempio precedente, con la differenza che viene esplicitamente indicato il descrittore dello *standard input*.

Redirezione normale dell'output

```
[n]> <parola>
```

La redirezione dell'output fa sì che il file il cui nome risulta dall'espansione della parola alla destra del simbolo > venga aperto in scrittura per ricevere quanto proveniente dal descrittore di file *n*, oppure, se non indicato, dallo *standard output* pari al descrittore di file 1.

Esempi

```
$ ls > ./dir.txt
```

Crea il file *dir.txt* nella directory corrente e gli inserisce l'elenco dei file della directory corrente.

```
$ ls 1> ./dir.txt
```

Esegue la stessa operazione dell'esempio precedente con la differenza che il descrittore che identifica lo *standard output* viene indicato esplicitamente.

```
$ ls 1>| ./dir.txt
```

Esegue la stessa operazione del primo esempio, ma si indica in maniera inequivocabile che il file *dir.txt* deve essere creato.

```
$ ls Pippo * 2> ./errori.txt
```

Crea il file errori.txt nella directory corrente e gli inserisce i messaggi di errore generati da *ls* quando si accorge che il file Pippo non esiste.

Processi e shell

La *shell* è l'intermediario tra l'utente e il sistema, e di conseguenza il mezzo normale attraverso cui si può avviare e controllare un processo. Un comando impartito attraverso una *shell* può generare più di un processo, per esempio quando viene avviato un programma o uno *script* che avvia a sua volta diversi programmi, oppure quando si realizzano delle *pipeline*. Per questo motivo, quando si vuole fare riferimento all'attività derivata da un comando dato attraverso una *shell*, si parla di *job* e non di singoli processi.

Controllo dei job di shell

Attraverso alcune *shell* è possibile gestire i *job* che in questo caso rappresentano raggruppamenti di processi generati da un unico comando.
La *shell* **`bash`**, e in generale le *shell* POSIX, oltre a **`ksh`** e **`csh`**, gestiscono i *job*. Nelle sezioni seguenti si fa riferimento al comportamento di **`bash`** (in qualità di *shell* POSIX), ma la maggior parte di quanto spiegato in queste sezioni vale anche per **`ksh`** e **`csh`**.

Non si deve confondere un *job* di *shell* con un processo. Un processo è un singolo eseguibile messo in funzione: se questo a sua volta avvia un altro eseguibile, viene generato un nuovo processo a esso associato. Un *job* di *shell* rappresenta tutti i processi che vengono generati da un comando impartito tramite la *shell* stessa. Basta immaginare cosa succede quando si utilizza una canalizzazione di programmi (*pipe*), dove l'output di un programma è l'input del successivo.

Foreground e background

L'attività di un *job* può avvenire in *foreground* (in primo piano) o in *background* (in sottofondo). Nel primo caso, il *job* impegna la *shell* e quindi anche il terminale, mentre nel secondo la *shell* è libera da impegni e così anche il terminale.
Di conseguenza, non ha senso pretendere da un programma che richiede l'interazione continua con l'utente che possa funzionare in *background*.
Se un programma richiede dati dallo *standard input* o ha la necessità di emettere dati attraverso lo *standard output* o lo *standard error*, per poterlo avviare come *job* in *background*, bisogna almeno provvedere a ridirigere l'input e l'output.

Avvio di un job in background

Un programma viene avviato esplicitamente come *job* in *background* quando alla fine della riga di comando viene aggiunto il simbolo **`&`**. Per esempio:
`# make zImage > ~/make.msg &`
avvia in *background* il comando **`make zImage`**, per generare un *kernel*, dirigendo lo *standard output* verso un file per un possibile controllo successivo dell'esito della compilazione.
Dopo l'avvio di un programma come *job* in *background*, la *shell* restituisce una riga contenente il numero del *job* e il numero del processo terminale generato da questo *job* (PID). Per esempio:
```
[1] 173
```
rappresenta il *job* numero 1 che termina con il processo 173.
Se viene avviato un *job* in *background* e questo a un certo punto ha la necessità di emettere dati attraverso lo *standard output* o lo *standard error* e questi non sono stati ridiretti, si ottiene una segnalazione simile alla seguente:
```
[1]+ Stopped (tty output) pippo
```
Nell'esempio, il *job* avviato con il comando **`pippo`** si è bloccato in attesa di poter emettere dell'output. Nello stesso modo, se viene avviato un *job* in *background* che a un certo punto ha la necessità di ricevere dati dallo *standard input* e questo non è stato ridiretto, si ottiene una segnalazione simile alla seguente:
```
[1]+ Stopped (tty input) pippo
```

Sospensione di un job in foreground

Se è stato avviato un *job* in *foreground* e si desidera sospenderne l'esecuzione, si può inviare attraverso la tastiera il carattere **`susp`**, che di solito si ottiene con la combinazione [*Ctrl+z*]. Il *job* viene sospeso e posto in *background*. Quando un *job* viene sospeso, la *shell* genera una riga come nell'esempio seguente:

```
[1]+ Stopped pippo
```

dove il *job* **`pippo`** è stato sospeso.

jobs

`jobs [<opzioni>] [<job>]`

Il comando di *shell* **`jobs`**, permette di conoscere l'elenco dei *job* esistenti e il loro stato. Per poter utilizzare il comando **`jobs`** occorre che non ci siano altri *job* in esecuzione in *foreground*, di conseguenza, quello che si ottiene è solo l'elenco dei *job* in *background*.

Alcune opzioni

`-l`

Permette di conoscere anche i numeri PID dei processi di ogni *job*.

`-p`

Emette solo i numeri PID del processo leader (quello iniziale) di ogni *job*.

Esempi

$ **jobs**

Si ottiene l'elenco normale dei *job* in *background*. Nel caso dell'esempio seguente, il primo *job* è in esecuzione, il secondo è sospeso in attesa di poter emettere l'output, l'ultimo è sospeso in attesa di poter ricevere l'input.

```
[1]   Running                 yes >/dev/null &
   [2]-  Stopped (tty output)    mc
   [3]+  Stopped (tty input)     unix2dos
```

$ **jobs -p**

Si ottiene soltanto l'elenco dei numeri PID dei processi leader di ogni *job*.

```
232
   233
   235
```

Per comprendere l'utilizzo dell'opzione **`-l`**, occorre avviare in sottofondo qualche comando un po' articolato.

$ **yes | cat | sort > /dev/null &**[*Invio*]

```
[1] 594
```

$ **yes | cat > /dev/null &**[*Invio*]

```
[2] 596
```

$ **jobs -l**[*Invio*]

```
[1]-   592 Running                 yes

          593                      | cat

          594                      | sort >/dev/null &

   [2]+   595 Running                 yes

          596                      | cat >/dev/null &
```

Come si può osservare, l'opzione **`-l`** permette di avere informazioni più dettagliate su tutti i processi che dipendono dai vari *job* presenti.

Riferimenti ai job

L'elenco di *job* ottenuto attraverso il comando **`jobs`**, mostra in particolare il simbolo **`+`** a fianco del numero del *job* attuale, ed eventualmente il simbolo **`-`** a fianco di quello che diventerebbe il *job* attuale se il primo termina o viene comunque eliminato.
Il *job* attuale è quello a cui si fa riferimento in modo predefinito tutte le volte che un comando richiede l'indicazione di un *job* e questo non viene fornito.
Di norma si indica un *job* con il suo numero preceduto dal simbolo **`%`**, ma si possono anche utilizzare altri metodi elencati nella tabella 20.1.

Simbolo	Descrizione
%*n*	Il *job* con il numero indicato dalla lettera *n*.
%<*stringa*>	Il *job* il cui comando inizia con la stringa indicata.
%?<*stringa*>	Il *job* il cui comando contiene la stringa indicata.
%%	Il *job* attuale.
%+	Il *job* attuale.
%-	Il *job* precedente a quello attuale.

Tabella 20.1: Elenco dei parametri utilizzabili come riferimento ai *job* di *shell*.

fg

`fg [<job>]`
Il comando **`fg`** porta in *foreground* un *job* che prima era in *background*. Se non viene specificato il *job* su cui agire, si intende quello attuale.

bg

`bg [<job>]`
Il comando **`bg`** permette di fare riprendere (in *background*) l'esecuzione di un *job* sospeso. Ciò è possibile solo se il *job* in questione non è in attesa di un input o di poter emettere l'output. Se non si specifica il *job*, si intende quello attuale.
Quando si utilizza la combinazione [*Ctrl+z*] per sospendere l'esecuzione di un *job*, questo viene messo in *background* e diviene il *job* attuale. Di conseguenza, è normale utilizzare il comando **`bg`** subito dopo, senza argomenti, in modo da fare riprendere il *job* appena sospeso.

kill

`kill [-s <segnale> | -<segnale>] [<job>]`
Il comando **`kill`** funziona quasi nello stesso modo del programma omonimo. Di solito, non ci si rende conto che si utilizza il comando e non il programma. Il comando **`kill`** in particolare, rispetto al programma omonimo, permette di inviare un segnale ai process di un *job*, indicando direttamente il *job*.
Quando si vuole eliminare tutto un *job*, a volte non è sufficiente un segnale **`SIGTERM`**. Se necessario si può utilizzare il segnale **`SIGKILL`** (con prudenza però).

Esempi

```
$ kill -KILL %1
```

Elimina i processi abbinati al *job* numero 1, inviando il segnale **`SIGKILL`**.

```
$ kill -9 %1
```

Elimina i processi abbinati al *job* numero 1, inviando il segnale **`SIGKILL`**, espresso in forma numerica.

Gestione del file-system

mount

Sintassi:

```
mount nodo
```

Opzione	Funzione
Nodo	Nome della directory dove deve essere montato il nuovo file system.

In Unix un utente può agganciare ad un nodo della directory radice un altro file system ,ad esempio un altro hard-disk, un cd-rom o il floppy.
Il super user deve inserire una voce nel file "/etc/fstab" per consentire agli utenti di montare un file-system.

Vediamo come è strutturato il file "/etc/fstab":

```
dispositivo             punto di montaggio      tipo    opzioni       dump
fsck
/dev/hda5               /                       ext2    defaults      1    1
/dev/hda6               swap                    swap    defaults      0    0
/dev/fd0                /floppy                 ext2    noauto        0    0
/dev/cdrom              /cdrom                  iso9660 noauto,ro     0    0
none                    /proc                   proc    defaults      0    0
```

Parametro	Significato
Dispositivo	Nome del file speciale dove si trova il file system da montare.
Punto di montaggio	Directory dove il file system deve essere montato.
Tipo	Tipo di file system.
Opzioni	Rw – lettura/scrittura; ro – solo lettura; noauto – non montare automaticamente al boot;
dump	Frequenza con la quale potrebbe essere fatto il backup del filesystem dal programma di utilità dump.
Fsck	Numero decimale che specifica l'ordine in base al quale il programma di utilità fsck controlla il filesystem.

umount

Sintassi:

```
umount nodo
```

Consente di smontare un file system.

df

Sintassi:

```
df [-k] filesystems
```

Opzione	funzione
-k	Visualizza le dimensioni in kilobytes anziché in blocchi di 512 bytes come per default.

Mostra lo spazio libero su un file system. Per default fornisce le informazioni su tutti i file system montati, ma è possibile indicare esplcitamente il file system.

Esempio:
Supponiamo di voler vedere lo spazio libero sulla partizione /dev/hda5

```
Df –k /dev/hda5

Filesystem         1024-blocks  Used Available Capacity Mounted on
/dev/hda5             497667  207918   264047     44%   /
```

du

Sintassi:

```
du [-a] [-b] [-k] [-S] directory
```

Opzione	**Funzione**
-a	Visulizza i dati per tutti i files, non solo per le directory
-b	Visualizza le dimensioni in bytes. (il default è blocchi di 512 bytes)
-k	Visualizza le dimensioni in kilobytes.
-S	Calcola la dimesnione delle directory senza includere le subdirectory.

Visualizza lo spazio utilizzato in una directory.

Esempio:
per vedere la dimensione di tutte le subdirectory della directory pippo scrivere:

```
du  /pippo
```

si ottiene:

```
53    ./Emeroteca
34    ./Articolo
23    ./Opengl
20    ./Gldemo
427   ./Sorg
114   ./Mlms
3     ./.wpe
28    ./emerot
6     ./temp
```

Programmazione della shell.

Parametri e variabili

Le shell di Unix consentono all'utente l'utilizzo di parametri e variabili.

Per definire una **variabile** è sufficiente assegnare ad essa un valore (è ammessa anche la stringa vuota), utilizzando una dichiarazione del tipo `<nome di variabile>=[<valore>]` (non specificando il valore, si inizializza il contenuto della variabile stessa con la stringa vuota); in modo analogo a molti linguaggi di programmazione il nome della variabile deve contenere soltanto lettere, cifre numeriche (ma non come primo carattere) ed il segno di sottolineatura. La lettura del contenuto di una variabile si ottiene invece facendo precedere il nome della stessa dal simbolo `$`. Per eliminare una variabile si può far ricorso al comando interno `unset` (seguito dal nome della variabile).

L'assegnazione o la creazione di una variabile ha validità solo nell'ambito della shell, ed i programmi avviati dalla shell stessa non risentono di queste variazioni; per ovviare a tale problema è necessario "esportare" le variabili da rendere visibili anche all'esterno della shell, utilizzando il comando interno `export <nome variabile>`.

Alcune variabili sono definite (ed inizializzate) direttamente dalla shell; di seguito ne vengono riportate alcune, specificando anche la shell di origine (tutte le variabili indicate sono comunque presenti in bash):

Variabile	Descrizione	Valore predefinito	Origine
`OPTIND`	Contiene l'indice del prossimo argomento elaborato dal comando *getopts* (si tratta di un comando interno utilizzato dagli *script* di shell per analizzare i parametri posizionali della linea di comando).		Bourne shell (sh).
`OPTARG`	Contiene il valore dell'ultimo argomento elaborato da *getops*.		Bourne shell (sh).
`IFS`	Internal Field Separator: contiene i caratteri da utilizzare come separatori all'interno della linea di comando.	<SP> <HT> <LF> cioè <*Spazio*> <*Tab*> <*newline*>.	Bourne shell (sh).
`PATH`	Contiene i percorsi di ricerca dei comandi (si tratta di un elenco di directory separate dal carattere `:`) (la ricerca nella directory corrente viene effettuata solo se nel path si include anche la directory `.`).		Bourne shell (sh).
`HOME`	Contiene la directoy home dell'utente.		Bourne shell (sh).
`CDPATH`	Contiene il percorso di ricerca per il comando *cd*.		Bourne shell (sh).
`MAILPATH`	Contiene la directory dei file di posta.		Bourne shell (sh).
`PS1`	Contiene il prompt primario.		Bourne shell (sh).
`PS2`	Contiene il prompt secondario.		Bourne shell (sh).
`IGNOREEOF`	Contiene il numero di <EOF> necessari per terminare.	1	C shell (csh)
`RANDOM`	Contiene un numero intero pseudocasuale.		Korn shell (ksh).
`REPLY`	Contiene la destinazione predefinita per il comando interno *read*.		Korn shell (ksh).
`SECONDS`	Contiene il numero di secondi trascorsi dall'avvio della shell (assegnando un nuovo valore a tale variabile, si provoca il riavvio del conteggio a partire dal valore stesso).		Korn shell (ksh).
`PWD`	Contiene la directory corrente.		Korn shell (ksh).
`OLDPWD`	Contiene la directory precedentemente visitata.		Korn shell (ksh).
`LINENO`	Contiene il numero della riga dello script o della funzione.		Korn shell (ksh).
`PS3`	Contiene il prompt del comando *select* (consente all'utente di effettuare una scelta		Korn shell (ksh).

	inserendo un valore attraverso la tastiera).		
`TMOUT`	Contiene il massimo tempo di attesa (in secondi).		Korn shell (ksh).
`HISTCMD`	Contiene l'indice del registro storico dei comandi.		Bourne Again Shell (bash).
`UID`	Contiene l'user ID dell'utente.		Bourne Again Shell (bash).
`EUID`	Contiene l'user ID efficace dell'utente.		Bourne Again Shell (bash).
`GROUPS`	Si tratta di un *array* contenente i numeri di ID dei gruppi di cui l'utente è membro.		Bourne Again Shell (bash).
`HOSTTYPE`	Contiene il nome del tipo di computer.		Bourne Again Shell (bash).
`OSTYPE`	Contiene il nome del sistema operativo.		Bourne Again Shell (bash).
`BASH_VERSION`	Contiene il numero di versione di bash.		Bourne Again Shell (bash).
`PPID`	Contiene il process ID del processo genitore della shell attuale.		Bourne Again Shell (bash).
`SHLVL`	Contiene il livello di annidamento di bash.		Bourne Again Shell (bash).
`HISTSIZE`	Contiene il numero di comandi da conservare nel registro storico dei comandi.	500	Bourne Again Shell (bash).
`HISTFILE`	Contiene il file storico dei comandi inseriti.	~/.bash_history	Bourne Again Shell (bash).
`BASH_ENV`	Contiene il file di configurazione per l'esecuzione degli script.		Bourne Again Shell (bash).

Un **parametro** è un particolare tipo di variabile di sola lettura; di conseguenza è possibile leggerne il contenuto facendo precedere il nome del parametro dal simbolo `$` (come per le variabili), ma non impostarne il valore attraverso un comando di assegnazione. Nell'ambito dei parametri, è possibile effettuare una distinzione fra quelli posizionali e quelli speciali.

Un **parametro posizionale** rappresenta uno degli argomenti forniti ad un comando (o funzione): `$1` è il primo argomento, `$2` il secondo, `$3` il terzo, ecc. (quando si utilizza un parametro composto da più di una cifra numerica, è necessario racchiuderlo fra parentesi graffe: `${10}` , `${11}` , ecc.).

Un **parametro speciale** viene indicato con uno `0` o con un altro carattere speciale; di seguito vengono indicati alcuni dei parametri speciali disponibili:

Parametro	Significato
`$0`	Restituisce il nome della shell o dello script.
`$*`	Rappresenta l'insieme di tutti i parametri posizionali a partire dal primo. Quando viene utilizzato all'interno di apici doppi, rappresenta un'unica parola composta dal contenuto dei parametri posizionali (separati dal primo carattere presente nella variabile `IFS`, oppure, se questa contiene una stringa vuota o non è definita, da uno spazio).
`$@`	Rappresenta l'insieme di tutti i parametri posizionali a partire dal primo. Quando viene utilizzato all'interno di apici doppi, rappresenta una serie di parole, ognuna composta dal contenuto del rispettivo parametro posizionale.
`$#`	Restituisce il numero di parametri posizionali.
`$?`	Restituisce lo stato dell'ultima pipeline eseguita in foreground (cioè in primo piano).
`$-`	Restituisce i flag di opzione.
`$$`	Restituisce il process ID della shell (se viene utilizzato all'interno di una subshell, restituisce il process ID della shell principale).
`$!`	Restituisce il valore dell'errore.
`$_`	Restituisce l'ultimo argomento del comando precedente.

Array

Oltre alle normali variabili scalari, con la *shell* bash è possibile fare uso di **array dinamici** ("dinamici" perché non è necessario stabilirne la dimensione, dato che questa viene modificata di volta in volta, in base all'ultimo elemento definito dell'*array*).
Per far riferimento ad uno specifico elemento di tale struttura dati, si fa seguire il nome della stessa dall'indice dell'elemento richiesto, racchiuso fra parentesi quadre (il primo elemento ha indice 0); se invece ci si vuol riferire al contenuto dell'elemento bisogna utilizzare la forma `${<array>[<indice>]}` (in questo caso le parentesi

quadre fanno parte dell'istruzione), omettendo l'indice ed indicando soltanto il nome dell'*array* (preceduto da $), ci si riferisce solo alla prima cella dello stesso.

Per creare un *array* è possibile assegnare singolarmente un valore ad i suoi elementi, oppure utilizzare il comando interno `declare` o `local` (per array locali), usando l'opzione `-a` . E' anche consentito assegnare in un sol colpo tutti i valori degli elementi di un *array*: si può utilizzare la sintassi

`<array>=(<valore 1> <valore 2> ... <valore n>)`

che riempie progressivamente le celle del vettore con i valori indicati, oppure è possibile indicare esplicitamente l'indice di un elemento al quale associare un certo valore, adottando la notazione `[<indice>]=<valore>` nell'ambito della precedente assegnazione (in questo caso le parentesi quadre fanno parte dell'istruzione).

In alcuni casi può essere utile espandere gli elementi di un *array* tutti contemporaneamente, utilizzando il simbolo * o @ al posto dell'indice (nel primo caso si ottiene un'unica parola, nel secondo si ottiene una parola per ogni elemento).
Tali simboli intervengono anche nel caso in cui si voglia conoscere la dimensione di un *array*: le espressioni `${<array>[*]}` e `${<array>[@]}` forniscono infatti il numero di elementi che costituiscono il vettore (le parentesi quadre fanno parte dell'istruzione).

Come nel caso delle variabili scalari, il comando `unset` permette di eliminare un *array*. Se però si fa riferimento a un particolare elemento di questo, si elimina solo l'elemento stesso, senza rimuovere l'intero *array*.

Espansione e sostituzione

Prima di poter eseguire gli ordini contenuti nella riga di comando, la shell deve effettuarne la suddivisione in parole ed eseguire la traduzione di parametri, variabili ed altre entità analoghe (in questo contesto per *parola* si intende una sequenza di caratteri che rappresenta qualcosa di diverso da un operatore, cioè una stringa che viene presa così come è e che rappresenta una cosa sola); tale meccanismo viene indicato con il termine **espansione**.

Per influenzare il procedimento di espansione, evitando che la shell interpreti alcuni caratteri speciali come tali, si utilizza la tecnica del **quoting** (che significa "racchiudere tra virgolette"):

1. Il carattere **** serve per evitare che il carattere seguente venga interpretato come carattere speciale (se però il carattere \ è l'ultimo di una riga, indica una continuazione sulla riga successiva).
2. Racchiudendo un serie di caratteri fra apici singoli (**'**), si evita che questi vengano interpretati come caratteri speciali.
3. Racchiudendo una serie di caratteri fra apici doppi (**"**), si evita che questi vengano interpretati come caratteri speciali, escludendo però i simboli $ e ` (che mantengono il loro significato speciale) e \ (che si comporta come carattere speciale solo se seguito da $, ` , **"** e \ ).

Il procedimento di espansione avviene in più passi consecutivi (eseguiti nell'ordine di seguito indicato):

- **Espansione delle parentesi graffe:**
  Si tratta di una caratteristica di bash derivante dalla C shell, che consiste nell'espansione di una scrittura del tipo `<prefisso>{<elenco>}<suffisso>` in una serie di parole composte dal prefisso, da uno degli elementi dell'elenco (uno per parola) e dal suffisso.
  Esempi:
  a) La scrittura `a{b,c,d}e` genera le seguenti tre parole: `abe ace ade` .
  b) Il comando `mkdir /usr/local/src/bash/{old,new,dist,bugs}` crea le quattro directory `old`, `new`, `dist` e `bugs` a partire da `/usr/local/src/bash/` .
  c) Il comando `chown root /usr/{paperino/{qui,quo,qua},topolino/{t??.*,minnie}}` cambia le proprietà di una serie di file:
  ```
  /usr/paperino/qui
  /usr/paperino/quo
  /usr/paperino/qua
  /usr/topolino/t??.*  (verrà ulteriormente espanso)
  /usr/topolino/minnie
  ```

- **Espansione della tilde (~) :**
  Si tratta di una caratteristica di bash derivante dalla C shell: se il simbolo tilde (~) è da solo oppure seguito dallo slash (/), ad esso verrà sostituito il nome della directory home dell'utente corrente; se invece il carattere

tilde è seguito dal nome di login di un utente, si sostituisce alla sequenza "tilde + nome utente " il nome della directory home dell'utente specificato.
Il simbolo tilde può essere espanso anche in modi diversi, se seguito dal carattere `+` o `-` : la coppia `~+` viene infatti espansa nel contenuto della variabile `PWD` (directory corrente), la coppia `~-` viene invece espansa nel contenuto della variabile `OLDPWD` (directory precedentemente visitata).
Esempi:

a) Il comando `cd ~` effettua uno spostamento nella directory home dell'utente corrente.
b) Il comando `cd ~tizio` effettua uno spostamento nella directory home dell'utente `tizio`.

- **Espansione di parametri e variabili:**
Consiste nella sostituzione di una scrittura del tipo `$<parametro>` o `$<variabile>` oppure `${<parametro>}` o `${<variabile>}` con il valore contenuto nel parametro (o nella variabile); la sintassi fra parentesi graffe è da preferire nel caso di parametri posizionali composti da più di una cifra decimale oppure all'interno di stringhe.
Oltre a questi modi standard di sostituzione di parametri e variabili, bash ne prevede altri (quasi tutti derivanti dalla Korn shell):
  a) Segnalazione di errore: `${<parametro o variabile>:?<messaggio>}`
  Definisce un messaggio di errore da utilizzare nel caso in cui la variabile (o parametro) non sia stata definita o sia pari alla stringa vuota.
  b) Valore predefinito: `${<parametro o variabile>:-<valore>}`
  Definisce un valore di default da utilizzare nel caso in cui la variabile (o parametro) non sia definita o sia pari alla stringa vuota.
  c) Rimpiazzo: `${<parametro o variabile>:+<valore>}`
  Definisce un valore da utilizzare al posto del contenuto della variabile (o parametro), nel caso in cui questo sia definito e diverso dalla stringa nulla.
  d) Lunghezza del contenuto: `${#<parametro o variabile>}`
  Si tratta di una caratteristica propria di bash: sostituisce il numero di caratteri del contenuto della variabile (o del parametro); se però il parametro è `*` o `@`, viene utilizzato il numero di parametri posizionali presenti.
  e) Valore predefinito con assegnamento: `${<variabile>:=<valore>}`
  Nel caso in cui la variabile non sia definita o risulti pari alla stringa vuota, assegna il valore indicato alla variabile stessa (ed utilizza tale valore in fase di espansione).

- **Sostituzione dei comandi:**
La sostituzione dei comandi consente di utilizzare quanto emesso attraverso lo *standard output* da un comando. La forma standard per ottenere ciò è `` $`<comando>` `` (bisogna ricordarsi però che il simbolo \ mantiene il suo significato letterale, a meno che non sia seguito dai simboli `$` , `` ` `` o `\`); la sostituzione dei comandi può anche essere annidata (ma è necessario far precedere i simboli `` ` `` interni dal carattere `\`).
La shell bash prevede anche una forma derivante dalla Korn shell, che elimina i problemi legati al simbolo \ ed al simbolo `` ` `` : `$(`*`comando`*`)` .
Esempio: il comando `ELENCO=$(ls)` crea la variabile `ELENCO`, assegnandole l'elenco dei file della directory corrente.

- **Espansione di espressioni aritmetiche:**
La shell bash è in grado di valutare delle espressioni aritmetiche; in questa fase l'intera espressione viene sostituita con il risultato della stessa. Ci sono due possibili forme utilizzabili: `$[<espressione>]` o `$((<espressione>))` ; l'espressione indicata viene trattata come se fosse racchiusa fra doppi apici.
Nel caso in cui l'espressione aritmetica non sia valida, la shell segnala l'errore e non esegue la sostituzione.
Tutti gli elementi contenuti nell'espressione sono sottoposti all'espansione di parametri e variabili, alla sostituzione di comandi e all'eliminazione dei simboli superflui per il *quoting*.
Esempi:
  a) Il comando `echo "$((123+123))"` emette il numero 146 (=123+23).
  b) Il comando `VALORE=$[123+23]` assegna alla variabile `VALORE` la somma di 123 e 23.
  c) Il comando `echo "$[123*VALORE]` emette il prodotto di 123 per il valore contenuto nella variabile `VALORE`.

- **Suddivisione in parole:**
I risultati delle espansioni di parametri e variabili, della sostituzione dei comandi e delle espressioni aritmetiche, se non sono avvenuti all'interno di apici doppi vengono ulteriormente suddivisi in parole; a questo proposito, la

shell considera ogni carattere contenuto all'interno della variabile `IFS` come un possibile delimitatore fra parole.

- **Espansione di pathname:**
  Le parole ottenute dalla precedente suddivisione vengono scandite, alla ricerca dei caratteri `*` , `?` o `[` (vengono indicati con il termine *caratteri jolly* o *metacaratteri*; le parole che li contengono vengono infatti considerate come un modello e ad esse viene sostituito un elenco (ordinato alfabeticamente) di tutti i *pathname* corrispondenti al modello (se non si ottiene nessuna corrispondenza, la parola resta immutata).
  I *caratteri jolly* vengono interpretati nel modo seguente:

*****	Corrisponde a qualsiasi stringa, compresa la stringa vuota.
?	Corrisponde ad un qualsiasi carattere (solo uno).
[…**]**	Corrisponde ad uno qualsiasi dei caratteri racchiusi tra le parentesi quadre.
[*a***-***z***]**	Corrisponde ad uno qualsiasi dei caratteri compresi nell'intervallo da *a* a *z*.
[!…**]**	Corrisponde ad un qualsiasi carattere, escluso quelli indicati all'interno delle parentesi quadre.
[!*a***-***z***]**	Corrisponde ad un qualsiasi carattere, escluso quelli compresi nell'intervallo da *a* a *z*.

  In particolare, per includere il carattere `-` o `[` o `]` all'interno di un raggruppamento fra parentesi quadre, occorre indicarli come primi o ultimi caratteri.

- **Eliminazione dei simboli di quoting rimanenti:**
  Al termine dei vari processi di espansione, tutti i simboli di *quoting* interpretati come tali (cioè non protetti da altri simboli di *quoting*) vengono rimossi.

Bash: programmazione.

Programmare una shell significa realizzare dei file ***script***, cioè normalissimi file di testo contenenti una serie di istruzioni che possano essere eseguite attraverso un interprete. Per eseguire lo *script* è dunque necessario avviare tale interprete, indicandogli quale file deve interpretare; in alternativa, è possibile inserire (solitamente nella prima riga dello script, nella forma: `#! <nome del programma interprete>`) l'informazione circa il nome del programma che deve interpretare lo *script* stesso (in questo modo, se si attribuisce al file il permesso di esecuzione, una richiesta di esecuzione dello stesso provocherà l'avvio automatico dell'interprete, che inizierà ad elaborare lo *script*).
Nel caso della shell bash, l'interprete di cui si è appena parlato è la shell stessa: per eseguire uno script è quindi necessario utilizzare una richiesta del tipo `bash <nome dello script>` oppure scrivere nella prima riga del file di testo i caratteri `#!/bin/bash` .

Commenti

Il simbolo **#** (in modo analogo al `//` del C++) indica l'inizio di un commento che prosegue fino alla fine della riga, l'interprete dello *script* ignora quindi tutta la porzione di riga che segue tale carattere. Non è un caso che l'indicazione del nome dell'interprete sia preceduta da tale simbolo: in questo modo tale informazione (utile alla shell per avviare il programma adatto all'interpretazione) non interferisce con il resto delle istruzioni, in quanto viene semplicemente ignorata.

Strutture

Per la formulazione di comandi complessi si possono utilizzare le tipiche strutture di controllo ed iterazione dei linguaggi di programmazione più comuni; è possibile farne uso sia nella normale riga di comando, sia (come soprattutto avviene) all'interno di *script* di shell.

Prima di proseguire con la descrizione delle strutture utilizzabili, è opportuno sottolineare una differenza rispetto al C: il simbolo di `;` viene utilizzato come carattere di separazione, ma in questo contesto può essere sostituito con uno o più codici di interruzione di riga.

- **`for`**
  Il comando `for` esegue una scansione di elementi, ed in corrispondenza di questi esegue una lista di comandi.
  La sintassi è:

```
for <variabile> [in <parola>...]
do
      <lista di comandi>
done
```

  Quando bash incontra una struttura di questo tipo, espande l'elenco di parole che segue `in`, generando una lista di elementi; la variabile indicata dopo `for` viene posta, di volta in volta, uguale al valore di ciascun elemento presente in questa lista, ed ogni volta vengono eseguiti i comandi indicati fra `do` e `done`. Omettendo la porzione di comando `in <parola>` , la shell esegue la lista di comandi richiesta una volta per ogni parametro posizionale esistente (omettere dunque `in <parola>` equivale quindi a specificare `in $@`).
  Il comando `for` restituisce il valore restituito dall'ultimo comando eseguito (fra quelli presenti all'interno della lista), oppure zero se non ne è stato eseguito nessuno.

  Esempio:

```
#!/bin/bash
for i in $*
do
      echo $i
done
```

  Questo *script* emette in sequenza gli argomenti che gli sono stati forniti.

- **`select`**
  Il comando `select` consente all'utente di effettuare una scelta inserendo un valore attraverso la tastiera.
  La sintassi è:

```
select <variabile> [in <parola>...]
do
      <lista di comandi>
done
```

Quando bash incontra una struttura di questo tipo, espande l'elenco di parole che segue `in`, generando una lista di elementi; l'insieme di tali elementi viene quindi emesso attraverso lo *standard error*, ognuno preceduto da un numero (omettere `in <parola>` equivale a scrivere `in $@`, in modo analogo a quanto accade per il comando `for`). Dopo l'emissione dell'elenco, viene mostrato il *prompt* contenuto nella variabile `PS3` e viene letta una riga dallo *standard input* (tale riga viene salvata nella variabile REPLY); se questa contiene un numero corrispondente ad una delle opzioni mostrate, il contenuto della variabile indicata dopo `select` viene posto uguale al numero stesso; se la riga è vuota vengono riemessi l'elenco ed il *prompt*; se viene letto il carattere di `<EOF>` (Ctrl + d), il comando termina; se si inserisce qualsiasi altro valore, la variabile viene invece riempita con la stringa vuota.
La lista dei comandi che segue `do` viene eseguita dopo ciascuna selezione, fino a che non viene incontrato un comando `break` o `return`; il valore restituito da select è pari a quello dell'ultimo comando eseguito (fra quelli presenti all'interno della lista), oppure zero se non ne è stato eseguito nessuno.

Esempio:

```
#!/bin/bash
select i in $*
do
      echo "hai selezionato $i premendo $REPLY"
      echo ""
      echo "premi Ctrl+c per terminare"
done
```

Questo *script* fa apparire un menu composto dagli argomenti fornitigli; ad ogni selezione mostra quello selezionato.

- **`case`**

Il comando `case` permette di eseguire una scelta nell'esecuzione di varie liste di comandi.
La sintassi è:

```
case <parola> in
      [<modello> [| <modello>]...) <lista di comandi>;; ]
      ...
esac
```

Quando bash incontra una struttura di questo tipo, espande la parola che segue `case` e la confronta con ciascuno dei modelli, utilizzando le stesse regole dell'espansione di *pathname* (è possibile utilizzare il carattere `|` per separare i modelli, quando questi rappresentano possibilità diverse di un'unica scelta); quando viene trovata una corrispondenza, viene eseguita la lista di comandi ad essa relativa; dopo il primo confronto riuscito, non ne viene effettuato nessuno dei successivi.
Il valore restituito da case è zero se nessun modello combacia, altrimenti è pari a quello restituito dall'ultimo comando eseguito (fra quelli presenti all'interno della lista associata alla condizione vera).

Esempio:

```
#!/bin/bash
case $1 in
      -a | -A | --alpha)      echo "alpha";;
      -b)                     echo "bravo";;
      -c)                     echo "charlie";;
esac
```

Questo *script* fa apparire un messaggio diverso in base all'argomento fornitogli (per ottenere la visualizzazione del messaggio "alpha" possono essere utilizzate tre diverse opzioni).

- **`if`**

Il comando `if` permette di eseguire liste di comandi differenti, in funzione di una o più condizioni (tali condizioni possono essere espresse, in generale, come liste di comandi).
La sintassi è:

```
if <lista-condizione>; then
      <lista di comandi>;
[ elif <lista-condizione>; then
      <lista di comandi>;]...
```

```
[ else <lista di comandi>;]
fi
```

Quando bash incontra una struttura di questo tipo, esegue la lista di comandi che segue `if` (e che costituisce una condizione); se il valore restituito è zero (che significa VERO in questo contesto), viene eseguita la lista di comandi posta dopo `then` ed il comando termina, altrimenti viene eseguita ogni `elif` in sequenza, fino a che non ne viene trovata una per la quale la condizione si verifica; se nessuna condizione risulta verificata, viene eseguita la lista di comandi che segue `else` (ammesso che esista).
Il valore restituito da `if` è pari a quello fornito dall'ultimo comando eseguito, o zero se non ne è stato eseguito nessuno.

Esempio:

```
#!/bin/bash
if [ $# = 0 ]
then
      echo "devi fornire almeno un argomento"
else
      echo $*
fi
```

Questo *script* fa apparire un messaggio di avvertimento se non gli è stato fornito nessun argomento, altrimenti visualizza tutti quelli ricevuti.
Una condizione racchiusa fra parentesi quadre, come quella mostrata in questo esempio, è una forma abbreviata del comando interno `test` (valuta l'espressione indicata e restituisce VERO o FALSO in base al risultato dell'espressione esaminata).

Esempio:

```
#!/bin/bash
if ! mkdir deposito
then
      echo "Non è stato possibile creare la directory \"deposito\""
else
      echo "È stata creata la directory \"deposito\""
fi
```

Questo *script* tenta di creare una directory, se l'operazione fallisce viene emessa una segnalazione di errore, in caso contrario viene visualizzato un messaggio indicante l'avvenuta creazione della directory.

- **`while`**

Il comando `while` permette di eseguire un gruppo di comandi in modo ripetitivo, fino a quando una certa condizione rimane vera (come per il comando `if`, anche in questo caso la condizione può essere una lista di comandi).
La sintassi è:

```
while <lista-condizione>
do
      <lista di comandi>
done
```

Quando bash incontra una struttura di questo tipo, esegue ripetitivamente la lista di comandi che specifica la condizione, e nel caso in cui questa risulti vera (cioè se restituisce 0), esegue i comandi indicati dopo `do`, dopo di che ricomincia il ciclo (tale procedimento si interrompe quando la condizione non risulta più verificata).
Il valore restituito da `while` è quello dell'ultimo comando fra quelli indicati dopo `do`, oppure zero se la condizione non è mai verificata.

Esempio:

```
#!/bin/bash
RISPOSTA="continua"
while [ $RISPOSTA != "fine" ]
do
      echo "usa la parola fine per terminare"
      read RISPOSTA
```

```
done
```

Questo *script* continua a leggere una riga dallo *standard input* dopo aver visualizzato un messaggio, fino a quando l'utente non inserisce la parola "fine". Il comando `read` si occupa proprio della lettura di tale riga, assegnando la prima parola alla prima variabile indicata come argomento (in questo caso "RISPOSTA"), la seconda parola alla seconda variabile, ecc. (i caratteri da considerare separatori sono quelli contenuti nella variabile `IFS`).

- **`until`**
  Il comando `until` permette di eseguire un gruppo di comandi in modo ripetitivo, fino a quando una certa condizione rimane falsa (la condizione è, in generale, il risultato fornito da una lista di comandi).
  La sintassi è:
  ```
  until <lista-condizione>
  do
        <lista di comandi>
  done
  ```

  Il comando `until` si comporta in modo opposto rispetto al comando `while`.

Funzioni

Le funzioni consentono di assegnare un nome ad un gruppo di comandi. In modo da poterlo richiamare come se si trattasse di un normale comando interno.
Per definire una funzione si utilizza una sintassi del tipo:

```
[ function ] <nome> ()
{
      <lista di comandi>
}
```

Ogni volta che viene utilizzato il nome della funzione come un comando, viene eseguita (all'interno della shell corrente, non viene cioè attivato nessun altro processo) la lista di comandi che la costituisce; il valore restituito dalla funzione è quello dell'ultimo comando eseguito all'interno della stessa.

Nell'ambito della funzione è possibile utilizzare parametri e variabili: i parametri posizionali forniti durante la chiamata alla funzione (bisogna però sottolineare che il parametro `$0` mantiene il valore precedente, ad esempio, se la funzione è richiamata dall'interno di uno *script*, contiene il nome dello *script* stesso), le variabili esterne e le variabili locali (definibili all'interno della funzione utilizzando il comando interno `local`, rispettando la sintassi: `local [<variabile locale>[=<valore>]...]`).
Per terminare anticipatamente l'esecuzione di una funzione, è possibile adoperare il comando interno `return`, rispettando la seguente sintassi: `return [n]` ; dove *n* è il valore restituito dalla funzione stessa (se omesso, la funzione restituirà il risultato dell'ultimo comando eseguito al suo interno).

In modo analogo a quanto avviene per le variabili, le funzioni definite all'interno di una shell non sono visibili alle subshell; per ovviare a questo problema è necessario "esportare" le funzioni da rendere visibili, facendo uso del comando interno `export`.

Esempio:

```
#!/bin/bash
messaggio ()
{
      echo "ciao,"
      echo "bella giornata vero?"
}

messaggio
```

Questo *script* dichiara la funzione "messaggio" e la richiama (come se si trattasse di un qualsiasi comando).

Esempio:

```
#!/bin/bash
function verifica ()
{
```

```
        if [ -e "/var/log/packages/$1" ]
        then
              return 0
        else
              return 1
        fi
}

if verifica pippo
then
        echo "il pacchetto pippo esiste"
else
        echo "il pacchetto pippo non esiste"
fi
```

Questo *script* dichiara la funzione "verifica", che sfrutta il comando interno `return` per restituire un valore vero (=0) o falso (!=0), in base all'esistenza o inesistenza di un file; il risultato di questa funzione viene utilizzato per stampare un messaggio che informi l'utente circa la reale presenza del file.

Espressioni aritmetiche

In alcune circostanze la shell consente di risolvere delle espressioni aritmetiche e logiche, il cui calcolo viene effettuato su interi di tipo long (quando si utilizzano come operandi parametri o variabili, il loro contenuto viene convertito in un numero intero).

La forma generale per esprimere un numero è `[<base>#]n` , dove `<base>` rappresenta la base numerica utilizzata, compresa fra 2 e 64 (se non viene indicata esplicitamente, si intende pari a 10). Le cifre utilizzabili per esprimere un numero sono, nell'ordine, le cifre numeriche, le lettere minuscole, le lettere maiuscole, il simbolo _ ed il simbolo `@` (se però la base di numerazione è inferiore a 36, non viene fatta distinzione fra lette maiuscole e minuscole). Nel caso di costanti ottali è possibile evitare di indicare esplicitamente la base se si fa iniziare il numero stesso con uno `0` (zero); nel caso di costanti esadecimali, invece, il numero deve iniziare per `0x` o `0X`.

Gli operatori utilizzabili (valutati in ordine di precedenza e tenendo conto delle parentesi), sono:

- **operatori aritmetici:**

espressione	Descrizione
`+<op>`	Non ha alcun effetto.
`-<op>`	Inverte il segno dell'operando.
`<op1>+<op2>`	Somma i due operandi.
`<op1>-<op2>`	Sottrae il secondo operando dal primo.
`<op1>*<op2>`	Moltiplica i due operandi.
`<op1>/<op2>`	Divide il primo operando per il secondo.
`<op1>%<op2>`	Restituisce il resto della divisione fra primo e secondo operando.
`<var>=<valore>`	Assegna il valore indicato alla variabile.
`<op1>+=<op2>`	Somma primo e secondo operando e pone il risultato nel primo operando.
`<op1>-=<op2>`	Sottrae il secondo operando dal primo e pone il risultato nel primo operando.
`<op1>*=<op2>`	Moltiplica primo e secondo operando e pone il risultato nel primo.
`<op1>/=<op2>`	Divide il primo operando per il secondo e pone il risultato nel primo.
`<op1>%=<op2>`	Pone nel primo operando il resto della divisione fra primo e secondo operando.

- **operatori di confronto**

Espressione	Descrizione
`<op1>=<op2>`	Restituisce vero se i due operandi sono uguali.
`<op1>!=<op2>`	Restituisce vero se i due operandi sono diversi.
`<op1> < <op2>`	Restituisce vero se il primo operando è minore del secondo.
`<op1> > <op2>`	Restituisce vero se il primo operando è maggiore del secondo.
`<op1> <= <op2>`	Restituisce vero se il primo operando è minore o uguale rispetto al secondo.
`<op1> >= <op2>`	Restituisce vero se il primo operando è maggiore o uguale rispetto al secondo.

- **operatori logici**

Espressione	Descrizione
`!<op>`	Inverte il risultato logico dell'operando.
`<op1> && <op2>`	Restituisce vero solo se gli operandi sono entrambi veri (se il primo operando è falso, non valuta il secondo e restituisce il risultato del primo, cioè falso).
`<op1> || <op2>`	Restituisce falso solo se gli operandi sono entrambi falsi (se il primo operando è vero, non valuta il secondo e restituisce il risultato del primo, cioè vero).

- **operatori binari**

Espressione	Descrizione
`<op1> & <op2>`	Esegue l'AND bit a bit fra i due operandi.
`<op1> | <op2>`	Esegue l'OR bit a bit fra i due operandi.
`<op1> ^ <op2>`	Esegue l'XOR bit a bit fra i due operandi.
`<op1> << <op2>`	Esegue uno shift a sinistra del primo operando del numero di posizioni indicate dal secondo.
`<op1> >> <op2>`	Esegue uno shift a destra del primo operando del numero di posizioni indicate dal secondo.
`~<op>`	Inverte tutti i bit dell'operando.
`<op1> &= <op2>`	Esegue l'AND bit a bit fra i due operandi e pone il risultato nel primo.
`<op1> |= <op2>`	Esegue l'OR bit a bit fra i due operandi e pone il risultato nel primo.
`<op1> ^= <op2>`	Esegue l'XOR bit a bit fra i due operandi e pone il risultato nel primo.
`<op1> <<= <op2>`	Esegue uno shift a sinistra del primo operando del numero di posizioni indicate dal secondo, il risultato viene posto nel primo operando.
`<op1> >>= <op2>`	Esegue uno shift a destra del primo operando del numero di posizioni indicate dal secondo, il risultato viene posto nel primo operando.