Shell Unix

17.1 Introduzione alla shell Unix 543
17.1.1 Shell POSIX 544
17.1.2 Invito della shell 544
17.1.3 Storico dei comandi 544
17.1.4 Comandi interni 545
17.1.5 Alias 545
17.1.6 Ambiente 545
17.1.7 Condotti 545
17.1.8 Script 545
17.1.9 Sostituzione o espansione 546
17.1.10 Suddivisione in parole 546
17.2 Utilizzo generale 547
17.2.1 Avvio e conclusione 547
17.2.2 Interpretazione dei comandi: parametri, variabili, espansione e sostituzione 547
17.2.3 Comandi 555
17.2.4 Ridirezione 558
17.2.5 Controllo dei *job* 561
17.2.6 Esecuzione dei comandi 562
17.2.7 Configurazione di ambiente 562
17.2.8 Particolarità importanti della shell Bash 563
17.3 Programmazione 567
17.3.1 Caratteristiche di uno script 567
17.3.2 Strutture 568
17.3.3 Espressioni aritmetiche 571
17.3.4 Comandi interni 572
17.4 Accesso ai file 584
17.4.1 Utilizzo dei descrittori 584
17.4.2 Lettura e scrittura 585
17.4.3 Contenuto senza file 586
17.5 Traduzione dei messaggi 587
17.5.1 Esempio iniziale 587
17.5.2 utilizzo più sofisticato 589
17.6 Libreria Readline 590
17.6.1 Comandi 590
17.6.2 Completamento automatico 591
17.6.3 Configurazione 591
17.6.4 Utilizzo di «cle» 592
17.7 Tabelle riepilogative 593
17.7.1 Particolarità della shell Bash 593
17.7.2 Parametri comuni 594
17.7.3 Variabili di ambiente comuni 594
17.7.4 Espansione e sostituzione 595
17.7.5 Comandi e *job* 596
17.7.6 Ridirezione 596
17.7.7 Strutture di controllo 597
17.7.8 Comando «echo» 599
17.7.9 Comando «set» 599
17.7.10 Comando «test» 600
17.7.11 Comando «ulimit» 602
17.7.12 Altri comandi interni 602
17.8 Riferimenti 609

`.inputrc` 565 591 `.profile` 547 `case` 568 `exit` 547 `for` 568 `getopts` 580 `gettext.sh` 587 `if` 569 `inputrc` 565 591 `profile` 547 `set` 582 `until` 570 `while` 570 `$ENV` 547 `$OPTARG` 580 `$OPTIND` 580

17.1 Introduzione alla shell Unix

La shell è il programma più importante in un sistema operativo, dopo il kernel. È in pratica il mezzo con cui si comunica con il sistema e attraverso il quale si avviano e si controlla l'esecuzione degli altri programmi.

La shell ha questo nome (conchiglia) perché di fatto è la superficie con cui l'utente entra in contatto quando vuole interagire con il sistema: la shell che racchiude il kernel.

Una shell è qualsiasi programma in grado di consentire all'utente di interagire con il sistema. Può trattarsi di qualcosa di molto semplice come una riga attraverso cui è possibile digitare dei comandi, oppure un menù di comandi già pronti, o un sistema grafico a icone, o qualunque altra cosa possa svolgere questo compito. Nei sistemi Unix si usano ancora shell a riga di comando, ma queste, anche se povere esteticamente, sono comunque molto potenti e difficilmente sostituibili.

La shell tipica di un sistema Unix è l'interprete di un linguaggio di programmazione orientato all'avvio e al controllo di altri programmi. Questo interprete è in grado di eseguire quanto richiesto da un utente attraverso una riga di comando in modo interattivo, oppure di eseguire un file script, scritto nel linguaggio della shell.

In origine esisteva una sola shell nei sistemi Unix; ovvero la «shell Unix». Attualmente le cose non sono più così e si fa riferimento alla shell storica con il nome del suo autore, Steve R. Bourne, pertanto si parla piuttosto di shell Bourne. La shell Bourne originale ha subito molti rimaneggiamenti e ne esistono diverse varianti e riscritture complete. In generale, a causa di queste diversificazioni conviene fare riferimento allo standard POSIX.

17.1.1 Shell POSIX

Le shell derivate da quella di Bourne che dichiarano di essere aderenti allo standard POSIX sono molte, purtroppo con tante piccole differenze tra di loro. Meritano attenzione, in particolare, la shell Bash,[1] ovvero *Bourne again shell*, predisposta per la massima compatibilità POSIX, ma ricca di estensioni proprie, e la shell Ash,[2] ovvero *Almquist shell*, usata soprattutto nei sistemi BSD.

17.1.2 Invito della shell

Quando una shell attende ed esegue i comandi impartiti dall'utente, si trova in una modalità di funzionamento interattivo. La disponibilità da parte della shell di ricevere comandi viene evidenziata dall'apparizione sullo schermo del terminale di un messaggio di invito o *prompt*. Questo, per lo più, è composto da simboli e informazioni utili all'utente per tenere d'occhio il contesto in cui sta operando.

In questo senso, l'invito è un elemento importante della shell, tenendo conto soprattutto della possibilità di configurarlo in base alle proprie esigenze. Il concetto di «invito» riguarda tutti i programmi che richiedono un'interazione con l'utente attraverso una riga di comando.

17.1.3 Storico dei comandi

Lo storico dei comandi è un registro degli ultimi comandi inseriti dall'utente[3] Quando la shell lo gestisce, l'utente è in grado di ripescare facilmente un comando utilizzato poco prima, senza doverlo riscrivere completamente, con la possibilità di modificarlo o di completarlo.

17.1.4 Comandi interni

Le shell POSIX e la maggior parte delle altre, mettono a disposizione dei comandi interni (o comandi incorporati) che vengono richiamati nello stesso modo con cui si avvia un programma normale. Solitamente, se esiste un programma con lo stesso nome di un comando interno, è il comando ad avere la precedenza, ma logicamente, i programmi standard che hanno lo stesso nome di comandi interni delle shell principali, svolgono di norma un compito simile, anche se non necessariamente identico.

Spesso, questo fatto è causa di equivoci fastidiosi: alle volte non si è in grado di capire il motivo per il quale un certo programma non funziona esattamente come ci si aspetterebbe.

17.1.5 Alias

Le shell POSIX e altre permettono la definizione di nuovi comandi in forma di ***alias*** di comandi già esistenti. L'utilità di questo sta nella possibilità di permettere l'uso di nomi differenti per uno stesso risultato, oppure per definire l'utilizzo sistematico di opzioni determinate.

Per comprendere il senso di questo si può considerare un esempio: si potrebbe creare l'alias '**`dir`**' che in realtà esegue il comando '**`ls -l`**'.

17.1.6 Ambiente

Ogni programma in funzione in un sistema Unix ha un proprio ***ambiente*** definito in base a delle ***variabili di ambiente***. Le variabili di ambiente sono un mezzo elementare e pratico di configurazione del sistema: i programmi, a seconda dei loro compiti e del loro contesto, cercano di leggere alcune variabili di loro interesse e in base al contenuto di queste adeguano il proprio comportamento.

L'ambiente consegnato a ogni programma che viene messo in esecuzione, è controllato dalla shell che è in grado di assegnare ambienti diversi a programmi diversi.

La shell può quindi creare, modificare e leggere queste variabili, cosa particolarmente utile per la realizzazione di file script.

17.1.7 Condotti

La shell mette in esecuzione i comandi ed è in grado di ridirigere il flusso di dati standard: standard input, standard output e standard error.

Questa caratteristica è importantissima per la realizzazione di comandi complessi attraverso l'elaborazione successiva da parte di una serie di programmi.

Dal punto di vista della shell, ogni comando, anche se composto dalla richiesta di esecuzione di un solo programma, è un condotto.

17.1.8 Script

Con il termine script si identifica un programma scritto ed eseguito nella sua forma sorgente senza l'intervento di alcuna compilazione. Normalmente, le shell sono in grado di eseguire dei file script, scritti secondo il loro linguaggio.

Per convenzione, gli script di shell e anche di altri linguaggi interpretati, iniziano con una riga che specifica il programma in grado di interpretarli.

```
#!/bin/sh
…
…
```

Questa riga, per esempio, è l'inizio di uno script che deve essere interpretato dal programma '`/bin/sh`', ovvero da una shell compatibile con quella di Bourne e possibilmente anche con la shell POSIX.

17.1.9 Sostituzione o espansione

Una caratteristica molto importante delle shell tradizionali è la possibilità di effettuare delle sostituzioni, o espansioni, nel comando impartito interattivamente o contenuto in un programma script.

17.1.9.1 Caratteri jolly o metacaratteri

I caratteri jolly, o metacaratteri, sono quei simboli utilizzati per fare riferimento facilmente a gruppi di file o di directory. Nei sistemi Unix sono le shell a occuparsi della traduzione dei caratteri jolly. In questo modo, una riga di comando che ne contiene, viene trasformata dalla shell che fornisce così, al programma da avviare, l'elenco completo di file e directory che si ottengono dall'espansione di questi caratteri speciali.

Dal momento che tale attività è competenza delle shell, dipende dalla shell utilizzata il tipo di caratteri jolly a disposizione e anche il loro significato.

È importante ricordare che alcuni testi fanno riferimento a questo concetto con il termine *globbing*; inoltre, a volte si utilizza la definizione di *shell regular expression*, ovvero *shell regexp*.

17.1.9.2 Variabili e parametri

Come accennato, le shell permettono di creare o modificare il contenuto di variabili di ambiente. Queste variabili possono essere utilizzate per la costruzione di comandi, ottenendo così la sostituzione con il valore che contengono, prima dell'esecuzione di questi.

Nello stesso modo i parametri, ovvero un tipo particolare di variabili a sola lettura, possono essere usati nelle righe di comando. Di solito si tratta degli argomenti passati a uno script.

17.1.9.3 Sostituzione di comandi

Le shell POSIX e altre, consentono di comporre un comando utilizzando lo standard output di un altro. In pratica, questi tipi di shell mettono in esecuzione prima i comandi da utilizzare per la sostituzione e quindi, con il risultato che ne ottengono, eseguono il comando ottenuto.

17.1.9.4 Protezione dalla sostituzione e dall'espansione

Dal momento che ogni shell può attribuire a dei simboli particolari un significato speciale, quando si ha la necessità di utilizzare tali simboli per il loro significato letterale (normale), occorre fare in modo che la sostituzione e l'espansione non abbiano luogo.

Generalmente si dispone di due tecniche possibili: l'uso di delimitatori all'interno dei quali la sostituzione e l'espansione non deve avere luogo (oppure può avvenire solo in parte) e l'uso di un carattere di escape. Il carattere di escape viene usato davanti al simbolo che non deve essere interpretato, mentre i delimitatori aprono e chiudono una zona protetta della riga di comando.

Dal momento che si devono usare dei simboli per delimitare o per rappresentare il carattere di escape, quando questi simboli devono essere usati nella riga di comando, occorre proteggere anch'essi. Sembra un circolo vizioso, ma alla fine tutto diventa molto semplice.

Il vero problema è che quando ci si abitua a una shell particolare, **ci si abitua** anche **a utilizzare delle tecniche consuete, perdendo di vista la sintassi vera dei comandi.**

17.1.10 Suddivisione in parole

Il compito di una shell tradizionale, quando viene usata in modo interattivo, è quello di interpretare le istruzioni date dall'utente e di avviare di conseguenza i comandi richiesti.

A questi comandi vengono passati normalmente degli argomenti e la separazione tra questi (argomenti) è fondamentale per il significato che assume l'istruzione data dall'utente. Infatti, non è compito dei comandi scomporre l'insieme degli argomenti, ma è compito della shell passarli debitamente separati. In questo modo, i comandi si possono limitare all'analisi di ogni singolo argomento.

Gli oggetti suddivisi che la shell riesce a individuare e quindi a passare ai comandi, sono le ***parole***. È molto importante la conoscenza del modo in cui una shell suddivide una riga di comando in parole.

17.2 Utilizzo generale

La shell POSIX è in pratica la shell Bourne standardizzata. Non esiste una sola shell POSIX, ma tante interpretazioni diverse, più o meno derivate da quella di Bourne.

Il primo elemento comune di queste shell è il programma eseguibile che le rappresenta: '`/bin/sh`'. In pratica, si tratta normalmente di un collegamento simbolico alla shell effettiva che ricopre quel ruolo. In particolare, ci sono shell come Bash che si adeguano agli standard quando sono avviate con quel nome.

17.2.1 Avvio e conclusione

L'eseguibile della shell POSIX è '**sh**', collocato nella directory '`/bin/`':

`sh` [*opzioni*] [*file_script*] [*argomenti*]

Si distinguono fondamentalmente due tipi di modalità di funzionamento: interattiva e non interattiva. Quando l'eseguibile '**sh**' viene avviato con l'indicazione del nome di un file, questo tenta di eseguirlo come uno script (in tal caso non conta che il file abbia i permessi di esecuzione e nemmeno che contenga la dichiarazione iniziale '**#!/bin/sh**'). Gli eventuali argomenti che possono seguire il nome del file, vengono passati allo script in forma di parametri (come viene descritto più avanti).

La shell è interattiva quando interagisce con l'utente e di conseguenza mostra un invito a inserire dei comandi. L'eseguibile '**sh**' può essere avviato eventualmente in modo esplicitamente interattivo utilizzando l'opzione '**-i**'.

Quando la shell funziona in modo interattivo, la variabile di ambiente ***PS1*** determina l'aspetto dell'invito, mentre il parametro **-** contiene anche la lettera '**i**' (i concetti relativi a variabili e parametri vengono chiariti in seguito).

```
$ echo $- [Invio]

himBH
```

Una shell interattiva può a sua volta essere una «shell di *login*» o meno. La distinzione serve alla shell per determinare quali file di configurazione utilizzare. Una shell di *login* è quella in cui il parametro zero, contiene un trattino ('**-**') come primo carattere (di solito contiene esattamente il valore '**-sh**'). In pratica è, o dovrebbe essere, quello che si ha di fronte quando è stata completata la procedura di accesso.

```
$ echo $0 [Invio]

-sh
```

Secondo lo standard POSIX, la shell di *login* esegue il contenuto del file indicato nella variabile di ambiente ***ENV***; tuttavia, di solito queste shell si comportano come la shell Bourne, per cui eseguono il contenuto dei file '`/etc/profile`' e '`~/.profile`' in sequenza. La shell POSIX interattiva esegue inizialmente il contenuto del file indicato nella variabile di ambiente ***ENV***.

Una shell non interattiva conclude il suo funzionamento al termine dello script che interpreta. Una shell interattiva termina di funzionare quando le si impartisce il comando '**exit**'.

17.2.2 Interpretazione dei comandi: parametri, variabili, espansione e sostituzione

Una volta avviata la shell in modo interattivo, questa mostra l'invito a inserire dei comandi, che prima di essere eseguiti sono soggetti

a un'interpretazione da parte della shell stessa. Nello stesso modo viene interpretato un comando contenuto all'interno di uno script che la shell esegue.

Il meccanismo di interpretazione della shell è molto complesso, perché prevede molte situazioni differenti, in cui ciò che appare deve essere sostituito da qualcosa di diverso. Si può pensare inizialmente a questo meccanismo come a qualcosa che assomiglia alle variabili di un linguaggio di programmazione comune; tuttavia la realtà di una shell è molto più varia e difficile, tanto che comprenderne bene il funzionamento richiede anche più impegno rispetto a un linguaggio di programmazione comune.

17.2.2.1 Protezione

Il *quoting* è un'azione con la quale si toglie il significato speciale che può avere qualcosa per la shell. Si distinguono tre possibilità: il carattere di escape (rappresentato dalla barra obliqua inversa), gli apici semplici e gli apici doppi (o virgolette). In generale, il concetto può essere trasferito in quello della protezione da un'interpretazione errata di ciò che si intende veramente.

È importante notare che il concetto di «protezione» è utilizzato in molte situazioni estranee all'uso della shell e ogni contesto può avere una logica differente.

La barra obliqua inversa ('\') rappresenta il carattere di escape. Serve per preservare il significato letterale del carattere successivo, cioè evitare che venga interpretato diversamente da quello che è veramente (salvo quando il contesto associa a una sequenza '\x' determinata un significato speciale).

Un caso particolare si ha quando il simbolo '\' è esattamente l'ultimo carattere della riga, o meglio, quando questo è seguito immediatamente dal codice di interruzione di riga: rappresenta una continuazione nella riga successiva.

Il simbolo '\', utilizzato per interrompere un'istruzione e riprenderla nella riga successiva, può essere utilizzato sia con una shell interattiva, sia all'interno di uno script. In ogni caso, bisogna fare bene attenzione a non lasciare spazi dopo questo simbolo, altrimenti non si comporterebbe più come segno di continuazione, ma come protezione di un carattere spazio.

L'esempio seguente mostra l'uso del comando '**echo**' per visualizzare un asterisco, ma dal momento che questo verrebbe rimpiazzato dall'elenco dei file e delle directory presenti nella directory corrente, viene protetto con la barra obliqua inversa:

```
$ echo \* [Invio]

*
```

L'esempio successivo rappresenta uno script, in cui il comando '**echo**' viene usato per visualizzare una stringa che nello script viene divisa su due righe, per comodità:

```
#!/bin/sh
echo "Saluti e baci \
bla bla bla."
```

Racchiudendo una sequenza di caratteri tra una coppia di apici semplici ('') si mantiene il valore letterale di questi caratteri. Evidentemente, un apice singolo non può essere contenuto in una stringa del genere.

Si tenga presente che l'apice inclinato nel modo opposto ('') viene usato con un altro significato che non rientra in quello della protezione delle stringhe delimitate.

L'esempio seguente mostra l'uso del comando '**echo**' per visualizzare una frase, contenente simboli che in condizioni normali verrebbero rimpiazzati da altre cose:


```
$ echo 'Attenzione: * e \ restano "inalterati".' [Invio]

Attenzione: * e \ restano "inalterati".
```

Racchiudendo una sequenza di caratteri tra una coppia di apici doppi si mantiene il valore letterale di questi caratteri, a eccezione di '**$**', '' e '\'. I simboli '**$**' e '' (dollaro e apice inverso) mantengono il loro significato speciale all'interno di una stringa racchiusa tra apici doppi, mentre la barra obliqua inversa ('\') si comporta come carattere di escape (di protezione) solo quando è seguita da '**$**', '', '**"**' e '\'; inoltre, quando si trova al termine della riga serve come indicatore di continuazione nella riga successiva.

Si tratta di una particolarità molto importante, attraverso la quale è possibile definire delle stringhe in cui si possono inserire: variabili, parametri e comandi da sostituire.

L'esempio seguente mostra l'uso del comando '**echo**' per mostrare una frase in cui si fa riferimento al parametro posizionale zero (che viene descritto in seguito). Questo parametro viene prima indicato proteggendo il dollaro, in modo da impedire che venga interpretato come tale, quindi viene inserito in modo da ottenerne il contenuto:

```
$ echo "Il parametro \$0 contiene: \"$0\"" [Invio]

Il parametro $0 contiene: "-sh"
```

17.2.2.2 Parametri e variabili

Nella documentazione comune si utilizza il termine «parametro» per identificare diversi tipi di entità: parametri posizionali; parametri speciali; variabili di shell. In questo documento, per evitare confusioni, si riserva il termine parametro solo ai primi due tipi di entità.

L'elemento comune tra i parametri e le variabili è il modo con cui questi oggetti devono essere identificati quando si vuole leggere il loro contenuto: occorre il simbolo '**$**' davanti al nome (o al simbolo) dell'entità in questione, mentre per assegnare un valore all'entità (sempre che ciò sia possibile), questo prefisso non deve essere indicato. Per la precisione, per leggere il contenuto di un parametro o di una variabile si usa normalmente una delle due forme seguenti:

```
$nome
```

```
${nome}
```

In pratica si usano le parentesi graffe per circoscrivere il nome o il simbolo associato alla variabile o al parametro, quando c'è la necessità di evitare ambiguità di qualche tipo.

I parametri sono delle variabili speciali che possono essere solo lette e rappresentano alcuni elementi particolari dell'attività della shell. Un parametro è definito, cioè esiste, quando contiene un valore, compresa la stringa nulla.

Parametro	Descrizione
n	Un parametro posizionale è definito da una o più cifre numeriche a eccezione dello zero che ha invece un significato speciale. I parametri posizionali rappresentano gli argomenti forniti al comando: '**$1**' si espande nel primo, '**$2**' si espande nel secondo e così di seguito. Quando si utilizza un parametro composto da più di una cifra numerica, è indispensabile racchiuderlo tra parentesi graffe; per esempio: '**${10}**', '**${11}**',...
0	Restituisce il nome della shell o dello script. Se la shell viene avviata con un file di comandi, '**$0**' si espande nel nome di quel file. Se la shell viene avviata con l'opzione '**-c**', '**$0**' si espande nel primo argomento dopo la stringa dei comandi (sempre che ce ne sia uno).

Parametro	Descrizione
*	L'asterisco 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, spaziati dal primo carattere contenuto nella variabile speciale ***IFS***. Se questa variabile non è definita, viene utilizzato uno spazio singolo. Per esempio, se ***IFS*** contenesse la sequenza '**xyz**', '**"$*"**' sarebbe equivalente a '**"$1x$2x…"**'. La variabile di shell ***IFS*** contiene di solito la sequenza: *<SP><HT><LF>* (corrispondente a uno spazio normale, un carattere di tabulazione e al codice di interruzione di riga nella maggior parte dei sistemi Unix). Di conseguenza, viene utilizzato normalmente il carattere spazio (*<SP>*) per staccare i vari parametri posizionali. Per cui, di solito, '**"$*"**' equivale a '**"$1 $2…"**'.
@	Rappresenta l'insieme di tutti i parametri posizionali a partire dal primo. Quando viene utilizzato all'interno di apici doppi, rappresenta delle parole, ognuna composta dal contenuto del parametro posizionale rispettivo. Di conseguenza, '**"$@"**' equivale a '**"$1" "$2" … "$*n*"**'. Questo comportamento rappresenta un'eccezione rispetto agli altri parametri che invece si limitano a generare una sola parola.
#	Rappresenta il numero di parametri posizionali esistenti.
?	Rappresenta il valore restituito dall'ultimo condotto eseguito in primo piano (*foreground*). In pratica, restituisce il valore dell'ultimo comando eseguito.
-	Il trattino rappresenta la serie di lettere corrispondenti alle modalità configurabili attraverso il comando interno '**set**' o con opzioni particolari della riga di comando.
$	Restituisce il numero PID della shell. Se viene utilizzato all'interno di una subshell, cioè tra parentesi tonde, restituisce il numero PID della shell principale e non quello della subshell.
!	Restituisce il numero PID del processo avviato più di recente e messo sullo sfondo.

Una variabile è definita quando contiene un valore, compresa la stringa nulla. L'assegnamento di un valore si ottiene con una dichiarazione del tipo seguente:

***nome_di_variabile*=[*valore*]**

Il nome di una variabile può contenere lettere, cifre numeriche e il trattino basso, ma il primo carattere non può essere un numero.

Se non viene fornito il valore da assegnare, si intende la stringa nulla. Come già accennato, la lettura del contenuto di una variabile si ottiene facendone precedere il nome dal simbolo '**$**'.

Tabella 17.9. Elenco delle variabili più importanti di una shell POSIX.

Variabile	Contenuto
PWD	La directory corrente. Il contenuto della variabile viene modificato dal comando '**cd**'.
OLDPWD	La directory corrente visitata precedentemente. Il contenuto della variabile viene modificato dal comando '**cd**'.
PPID	Il numero PID del processo genitore della shell attuale.
IFS	*Internal field separator*. Il contenuto predefinito della variabile dovrebbe essere: *<SP><HT><LF>*.
PATH	I percorsi di ricerca per i comandi, separati dal carattere '**:**'.
HOME	La directory personale dell'utente.
CDPATH	Il percorso di ricerca per il comando '**cd**' (di solito la variabile contiene la stringa nulla).
MAIL	Il percorso del file che rappresenta la cartella di posta in entrata dell'utente.

Variabile	Contenuto
MAILCHECK	La frequenza, in secondi, con cui si deve verificare la presenza di messaggi nuovi nella cartella corrispondente alla variabile ***MAIL***. Se ***MAILCHECK*** è vuota o contiene il valore zero, il controllo avviene ogni volta che deve essere emesso un nuovo invito.
MAILPATH	Questa variabile, se definita, prevale su ***MAIL*** e definisce un elenco di percorsi per altrettante cartelle di posta elettronica alternative. L'elenco è separato con il carattere '**:**'.
OPTIND	Contiene l'indice del prossimo argomento da elaborare dal comando '**getopts**'.
OPTARG	Il valore dell'ultimo argomento elaborato da '**getopts**'.
PS1	L'invito primario. Di solito, il valore predefinito di questa variabile fa sì che sia rappresentato un dollaro o un cancelletto a seconda che si tratti di un utente comune o dell'utente '**root**'.
PS2	L'invito secondario, che appare quando si deve completare un comando. Il valore predefinito è normalmente '**>** '.
ENV	Il nome di un file di configurazione per una shell POSIX.

Quando si creano o si assegnano delle variabili, queste hanno una validità limitata all'ambito della shell stessa, per cui, i comandi interni sono al corrente di queste variazioni mentre i programmi che vengono avviati non ne risentono. Perché anche i programmi ricevano le variazioni fatte sulle variabili, queste devono essere ***esportate***. L'esportazione delle variabili si ottiene con il comando interno '**export**'. L'esempio seguente mostra la creazione della variabile ***PIPPO***, a cui viene assegnato un valore, quindi si vede anche la sua esportazione per gli altri programmi:

```
$ PIPPO="ciao" [Invio]

$ export PIPPO [Invio]
```

17.2.2.3 Espansione

Con questo termine si intende la traduzione di parametri, variabili e altre entità analoghe, nel loro risultato finale. L'espansione, intesa in questo modo, viene eseguita sulla riga di comando, dopo che questa è stata scomposta in parole. Esistono almeno sei tipi di espansione eseguiti nell'ordine seguente:

1. tilde;
2. parametri e variabili;
3. comandi;
4. aritmetica (da sinistra a destra);
5. suddivisione delle parole;
6. percorso o *pathname*.

Solo la suddivisione in parole e l'espansione di percorso, possono cambiare il numero delle parole di un'espressione. Gli altri tipi di espansione trasformano una parola in un'altra parola con l'unica eccezione del parametro **@** che invece si espande in più parole.

Alla conclusione dei vari processi di espansione e sostituzione, tutti i simboli usati per la protezione ('\', '**'**' e '**"**') che a loro volta non siano stati protetti attraverso l'uso della barra obliqua inversa o di virgolette di qualche tipo, vengono rimossi.

Il termine ***parola*** ha un significato particolare nella terminologia utilizzata per la shell: si tratta di una sequenza di caratteri che rappresenta qualcosa di diverso da un operatore. Per descriverlo in modo differente, si può definire come una stringa che viene presa così com'è e rappresenta una cosa sola. Per esempio, un argomento fornito a un programma è una parola.

L'operazione di ***suddivisione in parole*** riguarda il meccanismo con cui una stringa viene analizzata e suddivida in parole in base a un criterio determinato. Questo problema viene ripreso più avanti in una sezione apposita.

17.2.2.4 Espansione della tilde

Se una parola inizia con il simbolo tilde (‘**~**’) si cerca di interpretare quello che segue, fino alla prima barra obliqua (‘**/**’), come un nominativo-utente, facendo in modo di sostituire questa prima parte con il nome della directory personale dell’utente stesso. In alternativa, se dopo il carattere ‘**~**’ c’è subito la barra, o nessun altro carattere, si intende il contenuto della variabile ***HOME***, ovvero la directory personale dell’utente attuale. Segue la descrizione di due esempi:

- `$ cd ~` [*Invio*]

  corrisponde a uno spostamento nella directory personale dell’utente;

- `$ cd ~tizio` [*Invio*]

  corrisponde a uno spostamento nella directory personale dell’utente ‘**tizio**’ (ammesso che i permessi lo consentano).

17.2.2.5 Espansione di parametri e variabili

Come già accennato in precedenza, il modo normale con cui si fa riferimento a un parametro o a una variabile è quello di anteporvi il simbolo dollaro (‘**$**’), ma questo metodo può creare problemi all’interno delle stringhe, oppure quando si tratta di un parametro posizionale composto da più di una cifra decimale. La sintassi normale è quindi la seguente:

```
$parametro | ${parametro}
```

```
$variabile | ${variabile}
```

In uno di questi modi si ottiene quindi la sostituzione del parametro o della variabile con il suo contenuto. Si osservino gli esempi seguenti. Il primo di questi visualizza in sequenza l’elenco degli argomenti ricevuti, fino all’undicesimo:

```
#!/bin/sh
echo " 1 arg. = $1"
echo " 2 arg. = $2"
echo " 3 arg. = $3"
…
echo "10 arg. = ${10}"
echo "11 arg. = ${11}"
```

L’esempio seguente, invece, compone il nome ‘**Daniele**’ unendo il contenuto di una variabile con una terminazione costante:

```
#!/bin/sh
UNO="Dani"
echo "${UNO}ele"
```

Oltre a questi modi «normali», è possibile espandere un parametro o una variabile indicando valori predefiniti; inoltre è possibile eseguire qualche operazione sulle stringhe, ma questi modelli di espansione non vengono descritti.

17.2.2.6 Sostituzione dei comandi

La sostituzione dei comandi consente di utilizzare quanto emesso attraverso lo standard output da un comando. Ci sono due forme possibili:

```
$(comando)
```

```
‘comando‘
```

Nel secondo caso dove si utilizzano gli apici inversi, la barra obliqua inversa (‘\’) che fosse contenuta eventualmente nella stringa, mantiene il suo significato letterale a eccezione di quando è seguita dai simboli ‘**$**’, ‘**‘**’ o ‘\’.

Bisogna fare attenzione a non confondere gli apici usati per la sostituzione dei comandi con quelli usati per la protezione delle stringhe.

La sostituzione dei comandi può essere annidata. Per farlo, se si utilizza il vecchio metodo degli apici inversi, occorre fare precedere a quelli più interni il simbolo di escape, ovvero la barra obliqua inversa.

Se la sostituzione è inserita in una stringa delimitata tra apici doppi, la suddivisione in parole e l’espansione di percorso non sono eseguite nel risultato.

Segue la descrizione di alcuni esempi:

- `$ ELENCO=$(ls)` [*Invio*]

  Crea e assegna alla variabile ***ELENCO*** l’elenco dei file della directory corrente.

- `$ ELENCO=‘ls‘` [*Invio*]

  Esattamente come nell’esempio precedente.

- `$ ELENCO=$(ls "a*")` [*Invio*]

  Crea e assegna alla variabile ***ELENCO*** l’elenco dell’unico file ‘`a*`’, ammesso che esista.

- `$ ELENCO=‘ls "a*"‘` [*Invio*]

  Esattamente come nell’esempio precedente.

- `$ rm $(find / -name "*.tmp")` [*Invio*]

  Elimina da tutto il file system i file che hanno l’estensione ‘`.tmp`’. Per farlo utilizza Find che genera un elenco di tutti i nomi che soddisfano la condizione di ricerca.

- `$ rm ‘find / -name "*.tmp"‘` [*Invio*]

  Esattamente come nell’esempio precedente.

17.2.2.7 Espansione di espressioni aritmetiche

Le espressioni aritmetiche consentono la valutazione delle espressioni stesse e l’espansione utilizzando il risultato:

```
$((espressione))
```

L’espressione viene trattata come se fosse racchiusa tra apici doppi, ma un apice doppio all’interno delle parentesi non viene interpretato in modo speciale. Tutti gli elementi all’interno dell’espressione sono sottoposti all’espansione di parametri, variabili, sostituzione di comandi ed eliminazione di simboli superflui per la protezione. La sostituzione aritmetica può essere annidata. Se l’espressione aritmetica non è valida, si ottiene una segnalazione di errore senza alcuna sostituzione.

Segue la descrizione di alcuni esempi:

- `$ echo "$((123+23))"` [*Invio*]

  Emette il numero 146 corrispondente alla somma di 123 e 23.

- `$ VALORE=$((123+23))` [*Invio*]

  Assegna alla variabile ***VALORE*** la somma di 123 e 23.

- `$ echo "$((123*$VALORE))"` [*Invio*]

  Emette il prodotto di 123 per il valore contenuto nella variabile ***VALORE***.

17.2.2.8 Suddivisione di parole

La shell esegue la suddivisione in parole dei risultati delle espansioni di parametri e variabili, della sostituzione di comandi e delle espansioni aritmetiche, purché non siano avvenuti all’interno di stringhe protette attraverso la delimitazione con apici doppi.

La shell considera ogni carattere contenuto all'interno di ***IFS*** come un possibile delimitatore utile a determinare i punti in cui effettuare la separazione in parole.

Perché le cose funzionino così come si è abituati, è necessario che ***IFS*** contenga i valori predefiniti: *<Spazio><Tab><new-line>* (ovvero *<SP><HT><LF>*). La variabile ***IFS*** è quindi importantissima: non può mancare e non può essere vuota.

Segue la descrizione di alcuni esempi.

- `$ cd /` [*Invio*]

  `$ Pippo="b* d*"` [*Invio*]

  `$ echo $Pippo` [*Invio*]

  In questo caso, avviene la suddivisione in parole del risultato dell'espansione della variabile ***Pippo***. In pratica, è come se si facesse: '`echo b* d*`'. Il risultato potrebbe essere quello seguente:

  ```
  bin boot dev
  ```

- `$ echo "$Pippo"` [*Invio*]

  In questo caso non avviene la suddivisione in parole di quanto contenuto tra la coppia di apici doppi e di conseguenza non può avvenire la successiva espansione di percorso:

  ```
  b* d*
  ```

- `$ echo '$Pippo'` [*Invio*]

  Se si utilizzano gli apici semplici, non avviene alcuna sostituzione della variabile ***Pippo***:

  ```
  $Pippo
  ```

- ```
  #!/bin/sh
  mio_programma "$*"
  ```

  Questo script avvia il programma '`mio_programma`' fornendo come **unico argomento** l'elenco di tutti gli argomenti ottenuti a sua volta.

- ```
  #!/bin/sh
  mio_programma $*
  ```

  Questo script avvia il programma '`mio_programma`' fornendo gli stessi argomenti ottenuti a sua volta.

- ```
  #!/bin/sh
  mio_programma $@
  ```

  Questo script avvia il programma '`mio_programma`' fornendo gli stessi argomenti ottenuti a sua volta (senza dipendere dalla variabile ***IFS***).

- ```
  #!/bin/sh
  mio_programma "$@"
  ```

  Esattamente come nell'esempio precedente, perché il parametro **@** tra apici doppi si espande in parole distinte.

17.2.2.9 Espansione di percorso

Dopo la suddivisione in parole, la shell scandisce ogni parola per la presenza dei simboli '`*`', '`?`' e '`[`'. Se incontra uno di questi caratteri, la parola che li contiene viene trattata come modello e sostituita con un elenco ordinato alfabeticamente di percorsi corrispondenti al modello. Se non si ottiene alcuna corrispondenza, il comportamento predefinito comune è tale per cui la parola resta immutata, consentendo quindi l'utilizzo dei caratteri jolly per il *globbing* (i metacaratteri) per identificare un percorso.

In generale, sarebbe meglio essere precisi quando si vuole indicare espressamente un nome che contiene effettivamente un asterisco o un punto interrogativo: si deve usare la barra obliqua inversa che funge da carattere di escape.

Per convenzione, si considerano nascosti i file e le directory che iniziano con un punto. Per questo, normalmente, i caratteri jolly non permettono di includere i nomi che iniziano con tale punto. Se necessario, questo punto deve essere indicato espressamente.

La barra obliqua di separazione dei percorsi non viene mai generata automaticamente dall'espansione di percorso (il *globbing*).

Tabella 17.19. Modelli utilizzabili per ottenere un'espansione di percorso.

Modello	Descrizione
`*`	Corrisponde a qualsiasi stringa, compresa la stringa nulla.
`?`	Corrisponde a un carattere qualsiasi (uno solo).
`[…]`	Corrisponde a uno qualsiasi dei caratteri racchiusi tra parentesi quadre.
`[!…]`	Corrisponde a tutti i caratteri esclusi quelli indicati.
`[a-z]`	Corrisponde a uno qualsiasi dei caratteri compresi nell'intervallo da ***a*** a ***z***.
`[!a-z]`	Corrisponde a tutti i caratteri esclusi quelli appartenenti all'intervallo indicato.

17.2.3 Comandi

Con il termine «comando» si intendono diversi tipi di entità che hanno in comune il modo con cui vengono utilizzate: attraverso un nome seguito eventualmente da alcuni argomenti. Può trattarsi dei casi seguenti.

- ***Comandi interni***

  Detti anche comandi di shell, sono delle funzioni predefinite all'interno della shell.

- ***Funzioni***

  Dette anche funzioni di shell, sono funzioni scritte all'interno di uno script di shell.

- ***Alias***

  Sono dei nomi associati ad altri comandi, di solito con l'aggiunta di qualche argomento. In maniera semplificata, possono essere visti come un modo diverso per identificare comandi già esistenti.

- ***Programmi***

  Detti anche comandi esterni perché non sono contenuti nella shell che li avvia.

17.2.3.1 Valore restituito dai comandi: «exit status»

Un comando che termina la sua esecuzione restituisce un valore, così come fanno le funzioni nei linguaggi di programmazione. Un comando, il quale può essere sia un comando interno, sia una funzione di shell, sia un programma, può restituire solo un valore numerico. Di solito, si considera un valore 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*.[4]

Per conto suo, la shell 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*).

17.2.3.2 Condotto

Il condotto (*pipeline*) è una sequenza di uno o più comandi separati da una barra verticale ('`|`'). Il formato normale per un condotto è il seguente:

[!] ***comando1*** [| ***comando2*** …]

Lo standard output del primo comando è incanalato nello standard input del secondo comando. Questa connessione è effettuata prima di qualsiasi ridirezione specificata dal comando. Come si vede dalla sintassi, per poter parlare di condotto basta anche un solo comando.

Normalmente, il valore restituito dal condotto corrisponde a quello dell'ultimo comando che viene eseguito all'interno di questo.

Se all'inizio del condotto viene posto un punto esclamativo ('`!`'), il valore restituito corrisponde alla negazione logica del risultato normale.[5]

Si osservi che il punto esclamativo **deve** essere separato dal comando che inizia il condotto, altrimenti potrebbe essere interpretato come parte del nome del comando, oppure, come avviene con la shell Bash, potrebbe servire per richiamare un comando dallo storico degli ultimi comandi inseriti.

La shell attende che tutti i comandi del condotto siano terminati prima di restituire un valore.

Ogni comando in un condotto è eseguito come un processo separato.

17.2.3.3 Lista di comandi

La lista di comandi è una sequenza di uno o più condotti separati da '`;`', '`&`', '`&&`' o '`||`', 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 è stato possibile eseguire.

I comandi separati da un punto e virgola ('`;`') sono eseguiti sequenzialmente. Il simbolo punto e virgola può essere utilizzato per separare dei 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.

L'esempio seguente avvia in sequenza dei comandi per la compilazione e installazione di un programma ipotetico:

```
# ./configure ; make ; make install [Invio]
```

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.

Nell'esempio seguente viene eseguito il comando '`mkdir ./prova`'. Se ha successo viene eseguito il comando successivo che visualizza un messaggio di conferma:

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

L'operatore di controllo '`||`' si comporta come l'operatore booleano OR: se il valore di uscita di ciò che 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.

Nell'esempio seguente si tenta di creare la directory '`prova/`', se il comando fallisce si tenta di creare '`prova1/`' al suo posto:

```
$ mkdir ./prova || mkdir ./prova1 [Invio]
```

17.2.3.4 Avvio sullo sfondo con «&»

I comandi seguiti dal simbolo '`&`' vengono messi in esecuzione sullo sfondo. La descrizione del meccanismo con cui i programmi possono essere messi e gestiti sullo sfondo viene fatta nella sezione 17.2.5. Dal momento che non si attende la loro conclusione per passare all'esecuzione di quelli successivi, il valore restituito è sempre zero. Segue la descrizione di alcuni esempi.

- `$ yes > /dev/null & echo "yes sta funzionando" [Invio]`

  Il programma '`yes`' viene messo in esecuzione sullo sfondo e di seguito viene visualizzato un messaggio. Al termine dell'esecuzione della lista, '`yes`' continua a funzionare.

- `$ echo "yes sta per essere avviato" ; yes > /dev/null & [Invio]`

  In questo caso viene prima emesso il messaggio e quindi viene avviato '`yes`' sullo sfondo.

- `# gpm -t ms & [Invio]`

  Avvia sullo sfondo il programma '`gpm`' di gestione del mouse.

Le liste, o parti di esse, possono essere racchiuse utilizzando delle **parentesi tonde**. Questo tipo di lista viene eseguita in una **copia della shell** (a volte si usa il termine subshell). Gli assegnamenti di variabili e l'esecuzione di comandi interni che influenzano l'ambiente della copia della shell che si occupa di eseguire la lista racchiusa tra parentesi, non lasciano effetti dopo che il comando composto è completato. Il valore restituito è quello dell'ultimo comando eseguito all'interno delle parentesi.

L'esempio seguente crea la directory '`prova/`' o '`prova1/`'. Se ci riesce, visualizza il messaggio.

```
$ (mkdir ./prova || mkdir ./prova1) ↩
↪&& echo "Creata la directory" [Invio]
```

Si osservi che il contenuto delle parentesi tonde può essere a contatto delle parentesi stesse, così come si vede nell'esempio.

Le liste possono essere raggruppate utilizzando delle **parentesi graffe**. Queste vengono eseguite nell'ambiente di **shell corrente**. Si tratta quindi di un semplice raggruppamento di liste su più righe. Il valore restituito è quello dell'ultimo comando eseguito all'interno delle parentesi.

L'uso delle parentesi graffe è indicato particolarmente nella preparazione di script di shell. Gli esempi seguenti sono equivalenti.

```
#!/bin/sh
{  mkdir ./prova ; cd ./prova ; ls ; }
```

```
#!/bin/sh
{  mkdir ./prova
   cd ./prova
   ls
}
```

Si osservi che quanto contenuto tra parentesi graffe, così come si vede negli esempi, **non può** essere aderente alle parentesi stesse; inoltre è indispensabile che dopo l'ultimo comando si dia il punto e virgola, oppure che la parentesi di chiusura appaia dopo un codice di interruzione di riga.

17.2.3.5 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, la shell cerca la prima parola di questo comando (quello che lo identifica) all'interno dell'elenco degli alias; se la trova lì, la sostituisce con il suo alias. La sostituzione non avviene se il comando o la prima parola di questo è delimitata 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.

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, se queste sono disponibili con la shell che si ha a disposizione.

L'uso di alias può essere utile se questi vengono definiti automaticamente per ogni avvio della shell, per esempio inserendoli all'interno di '`/etc/profile`'.

Segue la descrizione di alcuni esempi.

- `# alias rm="rm -i"` [*Invio*]

  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"` [*Invio*]

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

- `# alias mv="mv -i"` [*Invio*]

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

- `# alias ln="ln -i"` [*Invio*]

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

- `# alias spegni="shutdown -h -t 5 now"` [*Invio*]

  Crea l'alias '**`spegni`**' per abbreviare il comando di spegnimento normale.

17.2.4 Ridirezione

Prima che un comando sia eseguito, si possono ridirigere i suoi flussi di dati in ingresso e in uscita, utilizzando una notazione speciale che viene interpretata dalla shell. La ridirezione viene eseguita, nell'ordine in cui appare, a partire da sinistra verso destra.

Se si utilizza il simbolo '**`<`**' da solo, la ridirezione si riferisce allo standard input (corrispondente al descrittore di file zero. Se si utilizza il simbolo '**`>`**' da solo, la ridirezione si riferisce allo standard output (corrispondente al descrittore di file numero uno). La parola che segue l'operatore di ridirezione è sottoposta a tutta la serie di espansioni e sostituzioni possibili. Se questa parola si espande in più parole dovrebbe essere segnalato un errore.

Si distinguono normalmente tre tipi standard di descrittori di file per l'input e l'output:

- 0 = standard input;
- 1 = standard output;
- 2 = standard error.

Tabella 17.22. Sintassi per la ridirezione.

Sintassi	Descrizione
[*n*]< *file*	La ridirezione 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 zero.
[*n*]> *file*	La ridirezione 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 numero uno. Di solito, se il file da aprire in scrittura esiste già, viene sovrascritto.
[*n*]>> *file*	La ridirezione dell'output fatta in questo modo fa sì che se il file da aprire in scrittura esiste già, questo non sia sovrascritto, ma gli siano semplicemente aggiunti i dati.
<<[-] *parola_di_delimitazione* *testo* ... *parola_di_delimitazione*	Si tratta di un tipo di ridirezione particolare e poco usato. Istruisce la shell di leggere le righe successive fino a quando viene incontrata la parola indicata (senza spazi iniziali); successivamente invia quanto accumulato in questo modo allo standard input del comando indicato. In pratica, la parola indica la fine della fase di lettura. Non è possibile fare giungere l'input da una fonte diversa. Se la parola viene racchiusa tra virgolette, quelle usate per la protezione delle stringhe, si intende che il testo contenuto non deve essere espanso. Altrimenti, il testo viene espanso come di consueto. Se si usa il trattino ('**`<<-`**'), significa che le tabulazioni iniziali nel testo vengono eliminate.
[*n*]<&*m*	In questo modo si unisce il descrittore ***m*** al descrittore ***n*** di ingresso oppure, in mancanza dell'indicazione di ***n***, si unisce allo standard input.
[*n*]<&-	Chiude il descrittore ***n*** oppure, in mancanza dell'indicazione di ***n***, chiude lo standard input.
[*n*]>&*m*	In questo modo si unisce il descrittore ***n*** al descrittore ***m*** di uscita oppure, in mancanza dell'indicazione di ***n***, si unisce lo standard output.
[*n*]>&-	Chiude il descrittore ***n*** oppure, in mancanza dell'indicazione di ***n***, chiude lo standard output.
[*n*]<> *file*	In questo modo si apre il file indicato in lettura e scrittura, collegando i due flussi al descrittore ***n***. Se questo descrittore non è specificato si intende l'utilizzo di entrambi standard input e standard output.

La tabella successiva riduce i modelli alle situazioni più comuni.

Sintassi	Descrizione
< *file*	Invia il contenuto del file allo standard input.
> *file*	Crea o sovrascrive il file con quanto ottenuto dallo standard output.
2> *file*	Crea o sovrascrive il file con quanto ottenuto dallo standard error.
>> *file*	Crea o estende il file con quanto ottenuto dallo standard output.
2>> *file*	Crea o estende il file con quanto ottenuto dallo standard error.
2>&1	Invia lo standard error nello standard output.

Segue la descrizione di alcuni esempi.

- `$ sort < ./elenco` [*Invio*]

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

- `$ sort 0< ./elenco` [*Invio*]

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

- `$ ls > ./dir.txt` [*Invio*]

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

- `$ ls 1> ./dir.txt` [*Invio*]

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

- `$ ls XtgEWSjhy * 2> ./errori.txt` [*Invio*]

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

- `$ ls >> ./dir.txt` [*Invio*]

  Aggiunge al file '`dir.txt`' l'elenco dei file della directory corrente.

- `$ ls 1>> ./dir.txt` [*Invio*]

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

- `$ ls XtgEWSjhy * 2>> ./errori.txt` [*Invio*]

  Aggiunge al file '`errori.txt`' i messaggi di errore generati da '**`ls`**' quando si accorge che il file '`XtgEWSjhy`' non esiste.

- `$ ls XtgEWSjhy * > ./tutto.txt 2>&1` [*Invio*]

  Crea il file '`tutto.txt`' nella directory corrente e vi inserisce i messaggi di errore generati da '**`ls`**' quando si accorge che il file '`XtgEWSjhy`' non esiste, insieme all'elenco dei file esistenti.

17.2.4.1 Ridirezione e script

Lo standard input di uno script è diretto al primo comando a essere eseguito che sia in grado di riceverlo. Lo standard output e lo standard error di uno script provengono dai comandi che emettono qualcosa attraverso quei canali.

Mentre il fatto che l'output derivi dai comandi contenuti nello script dovrebbe essere intuitivo, il modo con cui è possibile ricevere l'input potrebbe non esserlo altrettanto. Il problema di creare uno script che sia in grado di ricevere dati dallo standard input si pone in particolare quando si deve realizzare il classico filtro di input per un file '`/etc/printcap`'. Nell'esempio seguente, il filtro di input riceve dati dallo standard input attraverso '**`cat`**'; quindi, con un condotto si arriva a un testo stampabile che viene inviato alla stampante predefinita (esistono molte interpretazioni differenti del programma '**`unix2dos`**'; in questo caso si considera che si tratti di un filtro che elabora ciò che gli viene passato attraverso lo standard input, restituendo il risultato dallo standard output).

```
#!/bin/sh
# /var/spool/text/input-filter
cat | /usr/bin/unix2dos | lpr
```

Un'altra cosa interessante in uno script è l'uso delle parentesi graffe per raggruppare un insieme di istruzioni che devono generare un flusso di dati comune da inviare a un solo comando:

```
#!/bin/sh
{ ls -l / ; ls /bin ; } | sort
```

In questo caso, vengono eseguiti i due comandi '**`ls`**' e quanto emesso da questi attraverso lo standard output viene inviato complessivamente a '**`sort`**'.

17.2.5 Controllo dei *job*

La shell standard prevede la gestione dei *job*, ovvero dei «gruppi di elaborazione», che in questo caso rappresentano raggruppamenti di processi generati da un solo comando.

Il controllo dei gruppi di elaborazione si riferisce alla possibilità di sospendere e ripristinare selettivamente l'esecuzione dei processi. La shell associa un gruppo di elaborazione a ogni condotto e mantiene una tabella di quelli in esecuzione, la quale può essere letta attraverso il comando interno '**`jobs`**'. Quando la shell avvia un processo sullo sfondo (ovvero in modo asincrono), emette una riga simile alla seguente, con cui indica, rispettivamente, il numero del gruppo di elaborazione (tra parentesi quadre) e il numero dell'ultimo processo (il PID) del condotto relativo:

```
[1] 12432
```

Si distinguono due tipi di gruppi di elaborazione:

- in primo piano o in *foreground*;
- sullo sfondo, o asincroni, o in *background*.

Un gruppo di elaborazione è in primo piano quando è collegato alla tastiera e al video del terminale che si sta utilizzando, mentre si trova a funzionare sullo sfondo quando è indipendente e asincrono rispetto all'attività del terminale.

Un gruppo di elaborazione in esecuzione in primo piano può essere sospeso immediatamente attraverso l'invio del carattere di sospensione, il quale si ottiene di solito con [*Ctrl z*], in modo da avere di nuovo a disposizione l'invito della shell. In alternativa si può sospendere un gruppo di elaborazione in esecuzione in primo piano, con ritardo, attraverso l'invio del carattere di sospensione con ritardo, che di solito si ottiene con [*Ctrl y*], in modo da avere di nuovo a disposizione l'invito della shell, ma solo quando il processo in questione tenta di leggere l'input dal terminale. È possibile gestire i gruppi di elaborazione sospesi attraverso i comandi '**`bg`**' e '**`fg`**'. Il comando '**`bg`**' consente di fare riprendere sullo sfondo l'esecuzione del gruppo di elaborazione sospeso, mentre '**`fg`**' consente di farne riprendere l'esecuzione in primo piano. Il comando '**`kill`**' consente di eliminare definitivamente il gruppo di elaborazione.

Per fare riferimento ai gruppi di elaborazione sospesi si utilizza il carattere '**`%`**'.

Riferimento ai gruppi di elaborazione	Descrizione
%***n***	Il simbolo '**`%`**' seguito da un numero fa riferimento al gruppo di elaborazione con quel numero.
%***prefisso***	Il simbolo '**`%`**' seguito da una stringa fa riferimento a un gruppo di elaborazione con un nome che inizia con quel prefisso. Se esiste più di un gruppo di elaborazione sospeso con lo stesso prefisso si ottiene una segnalazione di errore.
%?***stringa***	Il simbolo '**`%`**' seguito da '**`?`**' e da una stringa fa riferimento a un gruppo di elaborazione con una riga di comando contenente quella stringa. Se esiste più di un gruppo di elaborazione del genere si ottiene una segnalazione di errore.
%% %+	Le notazioni '**`%%`**' o '**`%+`**' fanno riferimento al gruppo di elaborazione corrente dal punto di vista della shell, il quale corrisponde all'ultimo a essere stato sospeso quando questo si trovava a funzionare in primo piano.
%-	La notazione '**`%-`**' fa riferimento al penultimo gruppo di elaborazione sospeso. Utilizzando i comandi '**`bg`**' e '**`fg`**', in mancanza di un riferimento esplicito al gruppo di elaborazione, viene preso in considerazione quello «corrente» dal punto di vista della shell.

Segue la descrizione di alcuni esempi.

- `$ fg %1` [*Invio*]

Porta in primo piano il gruppo di elaborazione numero uno.

- `$ %1` [*Invio*]

  Porta in primo piano il gruppo di elaborazione numero uno.

- `$ bg %1` [*Invio*]

  Mette sullo sfondo il gruppo di elaborazione numero uno.

- `$ %1 &` [*Invio*]

  Mette sullo sfondo il gruppo di elaborazione numero uno.

- `$ bg` [*Invio*]

  Mette sullo sfondo il gruppo di elaborazione corrente.

- `$ fg` [*Invio*]

  Porta in primo piano il gruppo di elaborazione corrente.

17.2.6 Esecuzione dei comandi

Dopo che un comando è stato suddiviso in parole, se il risultato è quello di un comando singolo, con eventuali argomenti, vengono eseguite le azioni seguenti.

- Se il nome del comando contiene una o più barre (‘`/`’), questo viene inteso essere un percorso del file system e di conseguenza il comando è inteso riferirsi precisamente a un file eseguibile, per cui la shell tenta di avviarlo.
- Se il nome del comando non contiene alcuna barra (‘`/`’):
  - se esiste una funzione di shell con quel nome, questa viene eseguita (purché sia disponibile la gestione delle funzioni);
  - se esiste un comando interno con quel nome, questo viene eseguito;
  - viene cercato all’interno del percorso di ricerca degli eseguibili contenuto nella variabile ***PATH***.

  Se la ricerca fallisce si ottiene una segnalazione di errore e la restituzione di un valore di uscita diverso da zero.

Quando la shell ha determinato che si tratta di un eseguibile esterno ed è riuscita a trovarlo, vengono svolte le azioni seguenti.

- La shell tenta di avviarlo.
- La shell avvia il programma configurando gli argomenti nel modo consueto: il primo, cioè zero, contiene il nome del programma, quelli successivi, contengono gli argomenti forniti eventualmente nella riga di comando.
- Se non si tratta di un programma e nemmeno di una directory (in tal caso verrebbe comunque emessa una segnalazione di errore), viene inteso essere uno script di shell. In tal caso viene generata una copia della shell (subshell) per la sua esecuzione, la quale si reinizializza in modo da presentare allo script una situazione simile a quella di una nuova shell.
- Se il programma è un file di testo che inizia con ‘`#!`’, si intende che si tratti di uno script che deve essere interpretato attraverso il programma indicato nella parte restante della prima riga. La shell esegue quindi quel programma dando come argomenti il nome dello script e altri eventuali argomenti ricevuti nella riga di comando originale.

17.2.7 Configurazione di ambiente

Quando viene avviato un programma gli viene fornito un vettore di stringhe che rappresenta la configurazione dell’ambiente. Si tratta di una lista di coppie di nomi e valori loro assegnati, espressi nella forma seguente:

```
nome=valore
```


La shell permette di manipolare la configurazione dell’ambiente in molti modi. Quando la shell viene avviata, esamina la sua configurazione di ambiente e crea una variabile per ogni nome trovato. Queste variabili vengono rese automaticamente disponibili, nello stato in cui sono in quel momento, ai processi generati dalla shell. Questi processi ereditano così l’ambiente. Possono essere aggiunte altre variabili alla configurazione di ambiente attraverso l’uso del comando interno ‘`export`’, mentre è possibile eliminare delle variabili attraverso il comando interno ‘`unset`’.

Le variabili create all’interno della shell che non vengono esportate nell’ambiente, attraverso il comando ‘`export`’, o che non vengono create attraverso il comando ‘`declare`’ (con l’opzione ‘`-x`’), non sono disponibili nell’ambiente dei processi discendenti (ovvero quelli generati durante il funzionamento della shell stessa).

Se si vuole fornire una configurazione di ambiente speciale all’esecuzione di un programma, basta anteporre alla riga di comando l’assegnamento di nuovi valori alle variabili di ambiente che si intendono modificare. L’esempio seguente avvia il programma ‘`mio_programma`’ sullo sfondo con un percorso di ricerca diverso, senza però influenzare lo stato generale della configurazione di ambiente della shell.

```
$ PATH=/bin:/sbin mio_programma &[Invio]
```

17.2.8 Particolarità importanti della shell Bash

Bash è una shell POSIX con delle estensioni proprie, piuttosto sofisticate. Nei sistemi GNU, la shell Bash è normalmente quella predefinita ed è bene conoscere alcune particolarità di questa shell, perché non sempre viene configurata per un’aderenza stretta alle specifiche POSIX.

17.2.8.1 File di configurazione

La shell Bash messa in funzione a seguito di un accesso (*login*), se non è stata specificata l’opzione ‘`--noprofile`’:

- tenta di leggere ed eseguire il contenuto di ‘`/etc/profile`’;
- tenta di leggere ed eseguire il contenuto di ‘`~/.bash_profile`’, se non ci riesce, tenta con ‘`~/.bash_login`’ e se anche questo file non è accessibile o non esiste, tenta ancora con il file ‘`~/.profile`’.

Al termine della sessione di lavoro:

- se esiste, legge ed esegue il contenuto di ‘`~/.bash_logout`’.

Quando la shell Bash funziona in modo interattivo, senza essere una shell di *login*, se non è stata specificata una delle opzioni ‘`--norc`’ o ‘`--rcfile`’, sempre che esista, legge ed esegue il contenuto di ‘`~/.bashrc`’.

Spesso si include l’esecuzione del contenuto del file ‘`~/.bashrc`’ anche nel caso di shell di *login*, attraverso un accorgimento molto semplice: all’interno del file ‘`~/.bash_profile`’ si includono le righe seguenti.

```
if [ -f ~/.bashrc ]
then
    . ~/.bashrc
fi
```

Il significato è semplice: viene controllata l’esistenza del file ‘`~/.bashrc`’ e se viene trovato viene caricato ed eseguito.

Quando la shell Bash viene utilizzata in modo non interattivo, ovvero per eseguire uno script, controlla il contenuto della variabile di ambiente ***BASH_ENV***; se questa variabile non è vuota esegue il file nominato al suo interno.

In pratica, attraverso la variabile ***BASH_ENV*** si indica un file di configurazione che si vuole sia eseguito dalla shell prima dello script. In situazioni normali questa variabile è vuota, oppure non esistente del tutto.

Se l'eseguibile della shell Bash viene avviato con il nome '**sh**' (per esempio attraverso un collegamento simbolico), per quanto riguarda l'utilizzo dei file di configurazione si comporta come la shell Bourne, mentre, per il resto, il suo funzionamento è conforme alla shell POSIX.

Nel caso di shell di *login*, tenta di eseguire solo '`/etc/profile`' e '`~/.profile`', rispettivamente. L'opzione '**`--noprofile`**' può essere utilizzata per disabilitare la lettura di questi file di avvio.

Se l'eseguibile '**bash**' viene avviato in modalità POSIX, attraverso l'opzione '**`--posix`**', allora la shell segue lo standard POSIX per i file di avvio. In tal caso, per una shell di *login* o interattiva viene utilizzato il nome del file contenuto nella variabile ***ENV***.

Tabella 17.29. A seconda del modo con cui l'eseguibile della shell Bash viene avviato si utilizzano diversi tipi di file di configurazione.

Comando	Tipo	All'avvio	Alla conclusione
`bash`	login	'`/etc/profile`', più '`~/.bash_profile`', oppure '`~/.bash_login`', oppure '`~/.profile`'	'`~/.bash_logout`'
`bash`	interattiva	'`~/.bashrc`'	
`bash`	non interattiva	il file indicato nella variabile di ambiente ***BASH_ENV***	
`sh`	login	'`/etc/profile`', più '`~/.profile`'	
`sh`	interattiva	il file indicato nella variabile di ambiente ***ENV***	
`sh`	non interattiva	--	
`bash` ↩ ↪`--posix`	login	il file indicato nella variabile di ambiente ***ENV***	
`bash` ↩ ↪`--posix`	interattiva	il file indicato nella variabile di ambiente ***ENV***	
`bash` ↩ ↪`--posix`	non interattiva	--	

17.2.8.2 Opzioni

La shell Bash interpreta due tipi di opzioni: a carattere singolo e multicarattere. Le opzioni multicarattere devono precedere necessariamente quelle a carattere singolo.

Opzione	Descrizione
`--norc`	Riguarda la modalità interattiva: non esegue il file di configurazione '`~/.bashrc`'. Quando si avvia l'eseguibile della shell Bash utilizzando il nome '**sh**' per mantenere la compatibilità con la shell Bourne, questo file di configurazione non deve essere letto e questa opzione è sottintesa.
`--noprofile`	Riguarda la modalità interattiva di *login*: non esegue i file di inizializzazione '`/etc/profile`', '`~/.bash_profile`', '`~/.bash_login`' o '`~/.bashrc`'.
`--rcfile` ***file***	Riguarda la modalità interattiva: non esegue il file di inizializzazione personalizzato '`~/.bashrc`', ma quello indicato come argomento.
`--version`	Visualizza il numero di versione.
`--login`	Fa in modo che funzioni in qualità di shell di *login*.
`--noediting`	Quando è avviata in modalità interattiva, non usa la libreria GNU Readline per gestire la riga di comando.
`--posix`	Fa in modo di adeguarsi il più possibile alle specifiche POSIX.

Opzione	Descrizione
`-c` ***stringa***	Vengono eseguiti i comandi contenuti nella stringa. Eventuali argomenti successivi vengono passati ai parametri posizionali a partire dal parametro zero.
`-i`	Forza l'esecuzione in modalità interattiva.
`-s`	La shell legge i comandi dallo standard input.

17.2.8.3 Uso sommario della tastiera

La shell Bash fornisce un sistema di gestione della tastiera molto complesso, attraverso un gran numero di funzioni. Teoricamente è possibile ridefinire ogni tasto speciale e ogni combinazione di tasti a seconda delle proprie preferenze. In pratica, non è consigliabile un approccio del genere, dal momento che tutto questo serve solo per gestire la riga di comando.

La tabella 17.31 mostra un elenco delle funzionalità dei tasti e delle combinazioni più importanti.

Tabella 17.31. Elenco delle funzionalità dei tasti e delle combinazioni più importanti.

Comando	Descrizione
Caratteri normali	Inseriscono semplicemente i caratteri corrispondenti.
[*Ctrl b*]	Sposta il cursore all'indietro di una posizione.
[*Ctrl f*]	Sposta il cursore in avanti di una posizione.
[*Backspace*]	Cancella il carattere alla sinistra del cursore.
[*Ctrl d*]	Cancella il carattere corrispondente alla posizione del cursore.
[*Ctrl a*]	Sposta il cursore all'inizio della riga.
[*Ctrl e*]	Sposta il cursore alla fine della riga.
[*Alt f*]	Sposta il cursore in avanti di una parola.
[*Alt b*]	Sposta il cursore all'indietro di una parola.
[*Ctrl l*]	Ripulisce lo schermo.

Generalmente funzionano anche i tasti freccia per spostare il cursore. In particolare, i tasti [*freccia-su*] e [*freccia-giù*] permettono di richiamare le righe di comando inserite precedentemente. Quando si preme un tasto o una combinazione non riconosciuta, si ottiene una segnalazione di errore.

Eventualmente si può intervenire nella configurazione della libreria Readline, attraverso il file '`/etc/inputrc`' oppure anche '`~/.inputrc`'. L'esempio seguente si riferisce alla configurazione necessaria per l'uso ottimale di una console virtuale su un elaboratore con architettura x86.

```
# Abilita l'inserimento di caratteri a 8 bit.
set meta-flag           on

# Disabilita la conversione dei caratteri con l'ottavo bit
# attivo in sequenze di escape.
set convert-meta        off

# Abilita la visualizzazione di caratteri a 8 bit.
set output-meta         on

# Modifica l'abbinamento con i tasti rispetto a determinati
# comportamenti.
"\e[1~": beginning-of-line      # [home]            era C-a
"\e[4~": end-of-line            # [fine]            era C-e
"\e[3~": delete-char            # [canc]            era C-d
"\e[5~": backward-word          # [pagina su]       era M-b
"\e[6~": forward-word           # [pagina giù]      era M-f
```

17.2.8.4 Invito o «prompt»

Quando la shell funziona in modo interattivo, può mostrare due tipi di invito:

- quello primario definito nella variabile ***PS1*** quando è pronta a ricevere un comando;

- quello secondario definito nella variabile ***PS2*** quando necessita di maggiori dati per completare un comando.

Il contenuto di queste variabili è una stringa che può essere composta da alcuni simboli speciali contrassegnati dal carattere di escape (‘`\`’); i principali sono descritti nella tabella 17.33.

Tabella 17.33. Elenco di alcuni codici speciali per definire l’invito con la shell Bash.

Codice	Descrizione
`\t`	Orario attuale nel formato ***hh***:***mm***:***ss*** (ore, minuti, secondi).
`\d`	Data attuale.
`\n`	Interruzione di riga.
`\s`	Nome della shell.
`\w`	Percorso assoluto della directory corrente.
`\W`	Nome finale del percorso della directory corrente (*basename*).
`\u`	Utente.
`\h`	Nome dell’elaboratore.
`\#`	Numero del comando attuale.
`\!`	Numero del comando nello storico.
`\$`	‘`#`’ se UID = 0; ‘`$`’ se UID > 0.
`\`***nnn***	Carattere corrispondente al numero ottale indicato.
`\\`	Una barra obliqua inversa singola (‘`\`’).
`\[`	Inizio di una sequenza di controllo.
`\]`	Fine di una sequenza di controllo.

In particolare merita attenzione ‘**`\$`**’, il cui significato potrebbe non essere chiaro dalla descrizione fatta nella tabella. Rappresenta un simbolo che cambia in funzione del livello di importanza dell’utente: se si tratta di un UID pari a zero (se cioè si tratta dell’utente ‘**`root`**’) corrisponde al simbolo ‘`#`’, negli altri casi corrisponde al simbolo ‘**`$`**’.

La stringa dell’invito, dopo la decodifica dei codici di escape appena visti, viene eventualmente espansa attraverso i processi di sostituzione dei parametri e delle variabili, della sostituzione dei comandi, dell’espressione aritmetica e della suddivisione delle parole.

L’esempio seguente fa in modo di ottenere un invito che visualizza il nome dell’utente, il nome dell’elaboratore, la directory corrente e il simbolo ‘**`$`**’ o ‘`#`’ a seconda del tipo di utente:

```
$ PS1='\u@\h:\w\$ ' [Invio]

tizio@dinkel:~$
```

Disponendo di una shell Bash, è possibile costruire anche un invito dinamico, con l’ausilio di funzioni. Naturalmente, resta però la necessità di garantire una certa compatibilità anche con delle shell POSIX standard. L’esempio seguente rappresenta una porzione di codice che potrebbe essere inserita nel file ‘`/etc/profile`’:


```
dynamic_prompt () {
    if [ $? = 0 ]
    then
        echo ":)"
    else
        echo ":("
    fi
}
export -f dynamic_prompt
…
PS1="\u@\h:\w\\$ "
…
if [ "$BASH" != "" ]
then
    # This is BASH.
    PS1="\$(dynamic_prompt) $PS1"
fi
export PS1
```

Come si può vedere, la variabile di ambiente ***PS1*** viene dichiarata inizialmente in modo compatibile con le shell standard, quindi, se si riesce a verificare che si tratta di una shell Bash, il contenuto della variabile viene modificato in modo da includere la funzione ‘**`dynamic_prompt`**’, la quale serve a mostrare un «sorriso» se l’ultimo comando è terminato con successo, ovvero restituendo il valore *Vero*.

17.3 Programmazione

La programmazione con una shell POSIX implica la realizzazione di file script. Alcune istruzioni sono particolarmente utili nella realizzazione di questi programmi, anche se non sono necessariamente utilizzabili solo in questa circostanza.

17.3.1 Caratteristiche di uno script

Nei sistemi Unix esiste una convenzione attraverso la quale si automatizza l’esecuzione dei file script. Prima di tutto, uno script è un normalissimo file di testo contenente delle istruzioni che possono essere eseguite attraverso un interprete. Per eseguire uno script occorre quindi avviare il programma interprete e informarlo di quale script questo deve eseguire. Per esempio, il comando seguente avvia l’eseguibile ‘**`sh`**’ come interprete dello script ‘**`pippo`**’, ovvero il file ‘`pippo`’ collocato nella directory corrente:

```
$ sh pippo [Invio]
```

Per evitare questa trafila, si può dichiarare all’inizio del file script il programma che deve occuparsi di interpretarlo. Per questo si usa la sintassi seguente:

```
#!percorso_del_programma_interprete
```

Quindi, si attribuisce a questo file il permesso di esecuzione:

```
$ chmod +x pippo [Invio]
```

Quando si tenta di avviare questo file come se si trattasse di un programma, il sistema avvia in realtà l’interprete.

Perché tutto possa funzionare, è necessario che il programma indicato nella prima riga dello script sia raggiungibile così come è stato indicato, cioè sia provvisto del percorso necessario. Per esempio, nel caso di uno script per la shell ‘**`sh`**’ (‘`/bin/sh`’), la prima riga deve essere composta così:

```
#!/bin/sh
…
…
```

Il motivo per il quale si utilizza il simbolo ‘`#`’ iniziale, è quello di permettere ancora l’utilizzo dello script nel modo normale, come argomento del programma interprete: rappresentando un commento non interferisce con il resto delle istruzioni.

Come appena accennato, il simbolo ‘`#`’ introduce un commento che termina alla fine della riga, cioè qualcosa che non ha alcun valore per

l'interprete; inoltre, le righe vuote e quelle bianche vengono ignorate nello stesso modo.

17.3.2 Strutture

Per la formulazione di comandi complessi si possono usare le strutture di controllo e di iterazione tipiche dei linguaggi di programmazione più comuni. Queste strutture sono particolarmente indicate per la preparazione di script di shell, ma possono essere usate anche nella riga di comando di una shell interattiva.

È importante ricordare che il punto e virgola singolo ('**;**') viene utilizzato per indicare una separazione e può essere rimpiazzato da uno o più codici di interruzione di riga.

17.3.2.1 for

Il comando '**for**' esegue una scansione di elementi e in corrispondenza di questi esegue una lista di comandi.

```
for variabile [in valore…]
do
    lista_di_comandi
done
```

L'elenco di parole che segue la sigla '**in**' viene espanso, generando una lista di elementi; la variabile indicata dopo '**for**' viene posta, di volta in volta, al valore di ciascun elemento di questa lista; infine, la lista di comandi che segue '**do**' viene eseguita ogni volta (una volta per ogni valore disponibile). Se la sigla '**in**' (e i suoi argomenti) viene omessa, il comando '**for**' esegue la lista di comandi ('**do**') una volta per ogni parametro posizionale esistente. In pratica è come se venisse usato: '**in $@**'.

Il valore restituito da '**for**' è quello dell'ultimo comando eseguito all'interno della lista '**do**', oppure zero se nessun comando è stato eseguito.

L'esempio seguente mostra uno script che, una volta eseguito, emette in sequenza gli argomenti che gli sono stati forniti:

```
#!/bin/sh
for i in $@
do
    echo $i
done
```

L'esempio seguente mostra uno script un po' più complicato che si occupa di archiviare, singolarmente, i file e le directory che si mettono come argomenti:

```
#!/bin/sh
ELENCO_DA_ARCHIVIARE=$@
for DA_ARCHIVIARE in $ELENCO_DA_ARCHIVIARE
do
    tar czvf ${DA_ARCHIVIARE}.tgz $DA_ARCHIVIARE
done
```

17.3.2.2 case

Il comando '**case**' permette di eseguire una scelta nell'esecuzione di varie liste di comandi. La scelta viene fatta confrontando una parola (di solito una variabile) con dei modelli. Se viene trovata una corrispondenza con uno dei modelli, la lista di comandi relativa viene eseguita.

```
case parola in
    [modello [ | modello]… ) lista_di_comandi ;; ]
    …
    [*) lista_di_comandi ;; ]
esac
```

La parola che segue '**case**' viene espansa e quindi confrontata con ognuno dei modelli, usando le stesse regole dell'espansione di percorso (i nomi dei file). La barra verticale ('**|**') viene usata per separare i modelli quando questi rappresentano possibilità diverse di un'unica scelta.

Quando viene trovata una corrispondenza, viene eseguita la lista di comandi corrispondente. Dopo il primo confronto riuscito, non ne vengono controllati altri dei successivi. L'ultimo modello può essere '***)**', corrispondente a qualunque valore, con cui si specifica un'alternativa finale in mancanza di un'altra corrispondenza.

L'esempio seguente mostra uno script che fa apparire un messaggio diverso a seconda dell'argomento fornitogli:

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

Come si può notare, per selezionare '**alpha**' si possono utilizzare tre opzioni diverse.

17.3.2.3 if

Il comando '**if**' permette di eseguire liste di comandi differenti, in funzione di una o più condizioni, espresse anch'esse in forma di lista di comandi.

```
if lista_condizione
then
    lista_di_comandi
[elif lista_condizione
then
    lista_di_comandi]
…
[else
    lista_di_comandi]
fi
```

Inizialmente viene eseguita la lista che segue '**if**', in qualità di condizione. Se il valore restituito da questa lista è zero (cioè *Vero*, ai fini della shell), allora viene eseguita la lista seguente '**then**' e il comando termina. Altrimenti viene eseguita ogni '**elif**' in sequenza, fino a che ne viene trovata una la cui condizione si verifica. Se nessuna condizione si verifica, viene eseguita la lista che segue '**else**', sempre che esista.

L'esempio seguente mostra uno script che fa apparire un messaggio di avvertimento se non è stato utilizzato alcun argomento, altrimenti si limita a visualizzarli:

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

L'esempio seguente mostra uno script attraverso il quale si tenta di creare una directory e se l'operazione fallisce viene emessa una segnalazione di errore:

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

È importante comprendere subito che le parentesi quadre sono un sinonimo del comando '**test**' e come tali **devono essere distaccate** da ciò che appare prima e dopo. Il comando '**test**' viene descritto

nella sezione 17.3.4 e la tabella 17.49 riporta le espressioni che con questo comando possono essere valutate.

17.3.2.4 while

Il comando '**while**' permette di eseguire un gruppo di comandi in modo ripetitivo mentre una certa condizione continua a dare il risultato *Vero*.

```
while lista_condizione
do
    lista_di_comandi
done
```

Il comando '**while**' esegue ripetitivamente la lista che segue '**do**' finché la lista che rappresenta la condizione continua a restituire il valore zero (*Vero*).

Lo script dell'esempio seguente contiene un ciclo perpetuo, in cui viene richiesto di inserire qualcosa, ma solo se si inserisce la stringa '**fine**' si conclude l'iterazione:

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

17.3.2.5 until

Il comando '**until**' permette di eseguire un gruppo di comandi in modo ripetitivo mentre una certa condizione continua a dare il risultato *Falso*.

```
until lista_condizione
do
    lista_di_comandi
done
```

Il comando '**until**' è analogo a '**while**', cambia solo l'interpretazione della lista che rappresenta la condizione nel senso che il risultato di questa viene invertito (negazione logica). In generale, per avere maggiori garanzie di compatibilità conviene utilizzare solo il comando '**while**', invertendo opportunamente la condizione.

17.3.2.6 Funzioni

Attraverso le funzioni è possibile dare un nome a un gruppo di comandi, in modo da poterlo richiamare come si fa per un comando interno normale. Sotto questo aspetto, le funzioni vengono impiegate normalmente all'interno di file script.

```
[function] nome () {
    lista_di_comandi
}
```

Le funzioni vengono eseguite nel contesto della shell corrente e quindi non vengono attivati altri processi per la loro interpretazione (ciò al contrario di quanto capita quando viene avviata l'interpretazione di un nuovo script).

La lista di comandi viene eseguita ogni volta che il nome della funzione è utilizzato come comando. Il valore restituito dalla funzione è quello dell'ultimo comando a essere eseguito all'interno di questa.

Pertanto, la funzione della shell può restituire solo un valore di uscita (*exit status*).

Quando viene eseguita una funzione, i parametri posizionali contengono gli argomenti di questa funzione e anche '**$#**' si espande in un

valore corrispondente alla situazione. Tuttavia, il parametro posizionale zero continua a restituire il valore precedente, di solito il nome dello script.

All'interno della funzione possono essere dichiarate delle variabili locali usando la parola chiave '**local**', prima della dichiarazione:

```
local nome[=valore_iniziale]
```

È possibile utilizzare il comando interno '**return**' per concludere anticipatamente l'esecuzione della funzione. Al termine dell'esecuzione della funzione, i parametri posizionali riacquistano il loro contenuto precedente e l'esecuzione dello script riprende dal comando seguente alla chiamata della funzione.

Se la shell lo consente, le funzioni possono essere esportate e rese disponibili a una sua copia (subshell) utilizzando il comando interno '**export**', così come si fa per le variabili di ambiente.

L'esempio seguente mostra uno script che prima dichiara una funzione denominata '**messaggio**' e subito dopo la esegue semplicemente nominandola come un comando qualsiasi:

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

messaggio
```

Nell'esempio seguente, una funzione si occupa di emettere il riepilogo della sintassi per l'uso di un ipotetico script:

```
function sintassi () {
    echo "al {--latex | --html | --txt | --check}"
    echo ""
    echo "--latex   esegue la conversione in latex;"
    echo "--html    esegue la conversione in html;"
    echo "--txt     esegue la conversione in testo normale;"
    echo "--check   esegue il controllo sintattico SGML;"
}
```

Nell'esempio seguente, si utilizza il comando '**return**' per fare in modo che l'esecuzione della funzione termini in un punto determinato restituendo un valore stabilito. Lo scopo dello script è quello di verificare che esista il file '`pippo`' nella directory '`/var/log/packages/`':

```
#!/bin/sh
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
```

17.3.3 Espressioni aritmetiche

La shell consente di risolvere delle espressioni aritmetiche in certe circostanze. In generale, nella maggior parte delle shell per le quali si dichiara la compatibilità POSIX, si ottiene l'espansione di un'espressione aritmetica con la forma seguente:

```
$((espressione))
```

Il calcolo avviene su interi senza controllo dello straripamento (*overflow*), anche se normalmente la divisione per zero viene intercettata e segnalata come errore. Oltre alle espressioni puramente aritmetiche si possono risolvere espressioni logiche e binarie, anche se l'utiliz-

zo di queste ultime non è indicato. La tabella 17.46 riporta l'elenco degli operatori aritmetici disponibili.

Tabella 17.46. Operatori aritmetici.

Operatore e operandi	Descrizione
`+op`	Non ha alcun effetto.
`-op`	Inverte il segno dell'operando.
`op1 + op2`	Somma i due operandi.
`op1 - op2`	Sottrae dal primo il secondo operando.
`op1 * op2`	Moltiplica i due operandi.
`op1 / op2`	Divide il primo operando per il secondo.
`op1 % op2`	Modulo: il resto della divisione tra il primo e il secondo operando.
`var = valore`	Assegna alla variabile il valore alla destra.
`op1 += op2`	`op1 = op1 + op2`
`op1 -= op2`	`op1 = op1 − op2`
`op1 *= op2`	`op1 = op1 * op2`
`op1 /= op2`	`op1 = op1 / op2`
`op1 %= op2`	`op1 = op1 % op2`

Le variabili di shell possono essere utilizzate come operandi; l'espansione di parametri e variabili avviene prima della risoluzione delle espressioni. Quando una variabile o un parametro vengono utilizzati all'interno di un'espressione, vengono convertiti in interi. Una variabile di shell non ha bisogno di essere convertita.

Gli operatori sono valutati in ordine di precedenza. Le sottoespressioni tra parentesi sono risolte prima.

17.3.4 Comandi interni

I comandi interni sono quelli eseguiti direttamente dalla shell, come se si trattasse di funzioni. La tabella 17.47 descrive brevemente alcuni comandi a disposizione di una shell standard tipica, anche leggermente oltre ciò che richiede strettamente lo standard POSIX. In particolare, i comandi '**`getopts`**' e '**`set`**' sono ripresi in sezioni separate.

È bene ricordare che dal punto di vista della shell, il valore numerico zero corrisponde a *Vero*, mentre qualunque valore diverso da zero corrisponde a *Falso*.

Tabella 17.47. Descrizione sintetica di alcuni comandi interni di una shell POSIX.

Comando	Descrizione
`: [argomenti]`	Comando nullo. Ciò che inizia con il simbolo '`:`' non viene eseguito. Si ottiene solo l'espansione degli argomenti e l'esecuzione della ridirezione. Il valore restituito alla fine è sempre zero.

Comando	Descrizione
`. file_script`	Vengono letti ed eseguito il contenuto del file indicato, il quale è sufficiente sia leggibile. Se il nome del file non fa riferimento a un percorso, questo viene cercato all'interno dei vari percorsi elencati dalla variabile ***PATH*** (ci sono shell che cercano il file anche nella directory corrente). Il valore restituito dallo script è: quello dell'ultimo comando eseguito al suo interno; zero (*Vero*) se non vengono eseguiti comandi; *Falso* (un valore diverso da zero) se il file non è stato trovato.
`alias [nome[=valore]]...`	Il comando '**`alias`**' permette di definire un alias, oppure di leggere il contenuto di un alias particolare, o di elencare tutti gli alias esistenti. Se viene utilizzato senza argomenti, emette attraverso lo standard output la lista degli alias nella forma '***nome=valore***'. Se viene indicato solo il nome di un alias, ne viene emesso il nome e il contenuto. Se si utilizza la sintassi completa si crea un alias nuovo. La coppia '***nome=valore***' deve essere scritta senza lasciare spazi prima e dopo del segno di uguaglianza ('**`=`**'). Il comando '**`alias`**' restituisce il valore *Falso* quando è stato indicato un alias inesistente senza valore da assegnare, negli altri casi restituisce *Vero*.
`bg [specificazione_del_job]`	Mette sullo sfondo il gruppo di elaborazione (*job*) indicato, come se fosse stato avviato aggiungendo il simbolo e-commerciale ('**`&`**') alla fine. Se non viene specificato il gruppo di elaborazione, viene messo sullo sfondo quello corrente, dal punto di vista della shell. Se l'operazione riesce, il valore restituito è zero.
`break [n]`	Interrompe un ciclo '**`for`**', '**`while`**' o '**`until`**'. Se viene specificato il valore numerico ***n***, l'interruzione riguarda ***n*** livelli. Il valore ***n*** deve essere maggiore o uguale a uno. Se ***n*** è maggiore dei cicli annidati in funzione, vengono semplicemente interrotti tutti. Il valore restituito è zero purché ci sia un ciclo da interrompere.
`cd [directory]`	Cambia la directory corrente. Se non viene specificata la destinazione, si intende la directory contenuta nella variabile ***HOME*** (che di solito corrisponde alla directory personale dell'utente). Il funzionamento di questo comando può essere alterato dal contenuto della variabile ***CDPATH*** che può indicare dei percorsi di ricerca per la directory su cui ci si vuole spostare. Di norma, la variabile ***CDPATH*** è opportunamente vuota, in modo da fare riferimento semplicemente alla directory corrente.

Comando	Descrizione
`command` *comando* [*argomento*...]	Esegue un «comando» con degli argomenti eventuali. Il comando che si avvia può essere solo un comando interno oppure un programma, mentre sono escluse espressamente le funzioni.
`continue` [*n*]	Riprende, a partire dall'iterazione successiva, un ciclo '**for**', '**while**' o '**until**'. Se viene specificato il valore numerico ***n***, il salto riguarda ***n*** livelli. Il valore ***n*** deve essere maggiore o uguale a uno. Se ***n*** è maggiore dei cicli annidati in funzione, si fa riferimento al ciclo più esterno. Il valore restituito è zero, a meno che non ci sia alcun ciclo da riprendere.
`echo` [`-n`] [*argomento*...]	Emette gli argomenti separati da uno spazio. Restituisce sempre il valore zero. '**echo**' riconosce alcune sequenze di escape che possono essere utili per comporre il testo da visualizzare. Queste sono elencate nella tabella 17.48. L'opzione '**-n**' (non prevista dallo standard, ma generalmente disponibile) consente di impedire che alla fine del testo visualizzato sia inserito il codice di interruzione di riga finale, in modo che il testo emesso successivamente prosegua di seguito. Si osservi che la shell Bash non riconosce le sequenze di escape se non si aggiunge espressamente l'opzione '**-e**', specifica di Bash, oppure si abilita l'opzione '**xpg_echo**' con il comando: '**shopt -s xpg_echo**'.
`eval` [*argomento*...]	Esegue gli argomenti come parte di un comando unico. Restituisce il valore restituito a sua volta dal comando rappresentato dagli argomenti. Se non vengono indicati argomenti, o se questi sono vuoti, restituisce *Vero*.
`exec` [*comando* [*argomenti*]]	Se viene specificato un comando (precisamente deve essere un programma), questo viene eseguito rimpiazzando la shell, in modo da non generare un nuovo processo ulteriore. Se sono stati indicati degli argomenti, questi vengono passati regolarmente al comando. Il fatto di rimpiazzare la shell implica che, al termine dell'esecuzione del programma, non c'è più la shell. Se si utilizza questo comando da una finestra di terminale, questa potrebbe chiudersi semplicemente, oppure, se si tratta di una shell di *login* potrebbe essere riavviata la procedura di accesso.
`exit` [*n*]	Termina l'esecuzione della shell restituendo il valore ***n***. Se viene omessa l'indicazione esplicita del valore da restituire, viene utilizzato quello dell'ultimo comando eseguito.

Comando	Descrizione
`export` *nome*...	Le variabili elencate vengono segnate per l'esportazione, nel senso che vengono trasferite all'ambiente dei programmi eseguiti successivamente all'interno della shell stessa.
`fg` [*job*]	Pone il gruppo di elaborazione (*job*) indicato in primo piano, ovvero in *foreground*. Se non viene specificato il gruppo di elaborazione, si intende quello attuale, ovvero, l'ultimo a essere stato messo sullo sfondo (*background*).
`getopts` *stringa_di_opzioni* ↩ ↪*nome_di_variabile* [*argomenti*]	Il comando interno '**getopts**' serve per facilitare la realizzazione di script in cui si devono analizzare le opzioni della riga di comando. Ogni volta che viene chiamato, '**getopts**' analizza l'argomento successivo nella riga di comando, restituendo le informazioni relative attraverso delle variabili di ambiente. Per la precisione, '**getopts**' analizza gli argomenti finali della sua stessa riga di comando (quelli che sono stati indicati nello schema sintattico come un elemento facoltativo) e in mancanza di questi utilizza il contenuto del parametro **@**. L'utilizzo di '**getopts**' può risultare complesso, pertanto viene descritto meglio in una sezione apposita.
`hash` [`-r`] [*comando*...]	Per ciascun comando indicato, viene determinato e memorizzato il percorso assoluto. Se non viene dato alcun argomento, si ottiene l'elenco dei comandi memorizzati. Se si usa l'opzione '**-r**' si cancellano i percorsi memorizzati.
`jobs` [*job*...]	Elenca i gruppi di elaborazione attivi. Se viene indicato esplicitamente un gruppo di elaborazione, l'elenco risultante è ristretto alle sole informazioni sullo stesso.
`kill` [`-s` *segnale*] *pid*|*job*... `kill -l` [*numero_del_segnale*]	Invia il segnale indicato al processo corrispondente al numero del PID o del gruppo di elaborazione (*job*). Il segnale viene definito attraverso un nome, come per esempio '**KILL**' (senza il prefisso '**SIG**'), o un numero di segnale. Spesso viene tollerata l'assenza dell'indicazione del segnale, ma in tal caso si intende '**TERM**'. Un argomento '**-l**' elenca i nomi dei segnali.
`local` [*nome*[=*valore*]]	All'interno di una funzione, dichiara una variabile con un campo di azione limitato alla funzione stessa. Il comando non è previsto dallo standard POSIX ma è diffuso tra le shell che sono comunque aderenti allo standard.
`pwd` [`-P`]	Emette il percorso assoluto della directory corrente. Se viene usata l'opzione '**-P**', i percorsi che utilizzano collegamenti simbolici vengono tradotti in percorsi reali. Restituisce zero se non si verifica alcun errore mentre si legge il percorso della directory corrente.

Comando	Descrizione
`read [-p prompt] variabile...`	Viene letta una riga dallo standard input, assegnando la prima parola di questa riga alla prima variabile indicata come argomento, assegnando la seconda parola alla seconda variabile e così via. All'ultima variabile indicata nella riga di comando viene assegnata la parte restante della riga dello standard input che non sia stata attribuita diversamente. Per determinare la separazione in parole della riga dello standard input si utilizzano i caratteri contenuti nella variabile ***IFS***. L'opzione '**-p**' permette di definire un invito particolare. Questo viene visualizzato solo se l'input proviene da un terminale.
`readonly [variabile[=valore]...]` `readonly -p`	Le variabili indicate vengono marcate per la sola lettura e i valori di queste non possono essere cambiati dagli assegnamenti successivi. Se viene indicata l'opzione '**-p**', si ottiene una lista di tutti i nomi a sola lettura.
`return [n]`	Termina l'esecuzione di una funzione restituendo il valore ***n***. Se viene omessa l'indicazione di questo valore, la funzione che termina restituisce il valore restituito a sua volta dall'ultimo comando eseguito al suo interno. Se il comando '**return**' viene utilizzato al di fuori di una funzione, ma sempre all'interno di uno script, termina l'esecuzione dello script stesso.

`set [{-|+}x]...`
`set {-|+}o [modalità]`
`set valore_param_1 ↩`
`↪[valore_param_2...]`
`set -- [valore_param_1 ↩`
`↪[valore_param_2...]]`
	Questo comando, se usato senza argomenti, emette l'impostazione generale della shell, nel senso che vengono visualizzate tutte le variabili di ambiente e le funzioni. Se si indicano degli argomenti si intendono alterare alcune modalità (opzioni) legate al funzionamento della shell.
	Dal momento che si tratta di un comando molto complesso, il suo utilizzo viene descritto in una sezione apposita.

`shift [n]`	I parametri posizionali da ***n***+1 in avanti sono spostati a partire dal primo in poi (il parametro zero non viene coinvolto). Se ***n*** è 0, nessun parametro viene cambiato. Se ***n*** non è indicato, il suo valore predefinito è uno. Il valore di ***n*** deve essere un numero non negativo minore o uguale al parametro **#** (cioè al numero di parametri posizionali esistenti). Se ***n*** è più grande del parametro **#**, i parametri posizionali non vengono modificati. Restituisce *Falso* se ***n*** è più grande del parametro **#** o minore di zero; altrimenti restituisce *Vero*.

Comando	Descrizione
`test espressione_condizionale` `[ espressione_condizionale ]`	Risolve (valuta) l'espressione indicata (la seconda forma utilizza semplicemente un'espressione racchiusa tra parentesi quadre). Il valore restituito può essere *Vero* (corrispondente a zero) o *Falso* (corrispondente a uno) ed è pari al risultato della valutazione dell'espressione. Le espressioni possono essere unarie o binarie. Le espressioni unarie sono usate spesso per esaminare lo stato di un file. Vi sono operatori su stringa e anche operatori di comparazione numerica. Ogni operatore e operando deve essere un argomento separato. Se si usa la forma tra parentesi quadre, è indispensabile che queste siano spaziate dall'espressione da valutare. Nella tabella 17.49 vengono elencate le espressioni elementari che possono essere utilizzate in questo modo. Non si tratta necessariamente di un comando interno.
`times`	Emette i tempi di utilizzo accumulati.
`trap [argomento segnale...]`	Il comando espresso nell'argomento deve essere letto ed eseguito quando la shell riceve il segnale, o i segnali indicati. Se non viene fornito l'argomento, o viene indicato un trattino ('**-**') al suo posto, tutti i segnali specificati sono riportati al loro valore originale (i valori che avevano al momento dell'ingresso nella shell). Se l'argomento fornito corrisponde a una stringa nulla, questo segnale viene ignorato dalla shell e dai comandi che questo avvia. Se il segnale è '**EXIT**', pari a zero, il comando contenuto nell'argomento viene eseguito all'uscita della shell. Se viene utilizzato senza argomenti, '**trap**' emette la lista di comandi associati con ciascun numero di segnale. I segnali intercettati sono riportati al loro valore originale in un processo discendente quando questo viene creato.
`true`	Si tratta di un comando nullo che restituisce zero, pari a *Vero*.
`type nome...`	Determina le caratteristiche di uno o più comandi indicati come argomento. Restituisce *Vero* se uno qualsiasi degli argomenti viene trovato, *Falso* se non ne viene trovato alcuno.

Comando	Descrizione
`ulimit` [*opzioni*] [*limite*]	Fornisce il controllo sulle risorse disponibili per la shell e per i processi avviati da questa, sui sistemi che permettono un tale controllo. Il valore del limite può essere un numero nell'unità specificata per la risorsa, o il valore '**`unlimited`**'. Se l'indicazione dell'entità del limite viene omessa, si ottiene l'informazione del valore corrente. Quando viene specificata più di una risorsa, il nome del limite e l'unità vengono emessi prima del valore. Il controllo pratico dei limiti impostati in questo modo dipende dal sistema operativo, il quale potrebbe anche ignorarne alcuni, per carenze realizzative nelle funzioni che dovrebbero attuare questi compiti. Se il limite viene espresso, questo diventa il nuovo valore per la risorsa specificata. Se non viene indicata alcuna opzione, si assume normalmente '**`-f`**' (l'unica a essere prevista dallo standard POSIX). I valori, a seconda dei casi, sono espressi in multipli di 1024 byte, o in «blocchi» da 512 byte, tranne per '**`-t`**' che è riferito a secondi e '**`-n`**' che rappresenta una quantità precisa. Il valore restituito è zero se non vengono commessi errori. La tabella 17.50 riepiloga le opzioni e i limiti più comuni che possono essere impostati con '**`ulimit`**'.
`umask` [*modalità*]	La maschera dei permessi per la creazione dei file dell'utente viene modificata in modo da farla coincidere con la modalità indicata. Generalmente può essere inserita la modalità soltanto in forma di numero ottale. Se la modalità viene omessa si ottiene il valore corrente della maschera.
`unalias` *nome_di_alias*… `unalias -a`	Rimuove l'alias indicato dalla lista degli alias definiti. Se viene fornita l'opzione '**`-a`**', sono rimosse tutte le definizioni di alias.
`unset` [`-v`] *nome_variabile*… `unset -f` *nome_funzione*…	Vengono rimosse le variabili o le funzioni indicate. Se viene utilizzata l'opzione '**`-f`**', si fa riferimento espressamente a funzioni; se si indica l'opzione '**`-v`**' ci si riferisce espressamente a variabili. Se non si indicano opzioni e ci può essere ambiguità tra i nomi, vengono rimosse le variabili.
`wait` [*n*]	Attende la conclusione del processo specificato e restituisce il suo valore di uscita. Il numero ***n*** può essere un PID o un gruppo di elaborazione (*job*); se viene indicato un gruppo di elaborazione, si attende la conclusione di tutti i processi nel condotto relativo. Se ***n*** non viene indicato, si aspetta la conclusione di tutti i processi discendenti ancora attivi, restituendo il valore zero. Se ***n*** specifica un processo o un gruppo di elaborazione che non esiste, viene restituito il valore *Falso*, altrimenti il valore restituito è lo stesso dell'ultimo processo o gruppo di elaborazione di cui si è attesa la conclusione.

Tabella 17.48. Alcune sequenze di escape che possono essere riconosciute dal comando '**`echo`**'.

Codice	Descrizione
`\\`	Inserisce la barra obliqua inversa ('\').
`\a`	Inserisce il codice <*BEL*> (avvisatore acustico).
`\b`	Inserisce il codice <*BS*> (*backspace*).
`\c`	Alla fine di una stringa previene l'inserimento di una nuova riga.
`\f`	Inserisce il codice <*FF*> (*formfeed*).
`\n`	Inserisce il codice <*LF*> (*linefeed*).
`\r`	Inserisce il codice <*CR*> (*carriage return*).
`\t`	Inserisce una tabulazione normale (<*HT*>).
`\v`	Inserisce una tabulazione verticale (<*VT*>).
`\0`***n***	Inserisce il carattere corrispondente al codice ottale ***n***.

Tabella 17.49. Espressioni per il comando '**`test`**'.

Espressione	Descrizione
`-e` ***file***	*Vero* se il file esiste ed è di qualunque tipo.
`-b` ***file***	*Vero* se il file esiste ed è un dispositivo a blocchi.
`-c` ***file***	*Vero* se il file esiste ed è un dispositivo a caratteri.
`-d` ***file***	*Vero* se il file esiste ed è una directory.
`-f` ***file***	*Vero* se il file esiste ed è un file normale.
`-h` ***file***	*Vero* se il file esiste ed è un collegamento simbolico.
`-p` ***file***	*Vero* se il file esiste ed è un file FIFO (*pipe* con nome).
`-S` ***file***	*Vero* se il file esiste ed è un socket (socket di dominio Unix).
`-t` ***descrittore***	*Vero* se lo standard output è aperto su un terminale.
`-g` ***file***	*Vero* se il file esiste ed è impostato il suo bit SGID.
`-u` ***file***	*Vero* se il file esiste ed è impostato il suo bit SUID.
`-k` ***file***	*Vero* se il file ha il bit Sticky attivo.
`-r` ***file***	*Vero* se il file esiste ed è leggibile.
`-w` ***file***	*Vero* se il file esiste ed è scrivibile.
`-x` ***file***	*Vero* se il file esiste e dispone del permesso di esecuzione, oppure se la directory esiste e c'è il permesso di attraversamento.
`-O` ***file***	*Vero* se il file esiste e appartiene all'UID efficace dell'utente attuale.
`-G` ***file***	*Vero* se il file esiste e appartiene al GID efficace dell'utente attuale.
`-s` ***file***	*Vero* se il file esiste e ha una dimensione maggiore di zero.
file1 `-nt` ***file2***	*Vero* se il primo file ha la data di modifica più recente.
file1 `-ot` ***file2***	*Vero* se il primo file ha la data di modifica più vecchia.
file1 `-et` ***file2***	*Vero* se i due nomi corrispondono allo stesso inode.
stringa	*Vero* se la stringa è diversa dalla stringa nulla.
`-z` ***stringa***	*Vero* se la lunghezza della stringa è zero.
`-n` ***stringa***	*Vero* se la lunghezza della stringa è diversa da zero.
stringa1 `=` ***stringa2***	*Vero* se le stringhe sono uguali.

Espressione	Descrizione
stringa1 != *stringa2*	*Vero* se le stringhe sono diverse.
stringa1 < *stringa2*	*Vero* se la prima stringa è lessicograficamente precedente.
stringa1 > *stringa2*	*Vero* se la prima stringa è lessicograficamente successiva.
op1 -eq *op2*	*Vero* se gli operandi hanno valori numerici uguali.
op1 -ne *op2*	*Vero* se gli operandi hanno valori numerici differenti.
op1 -lt *op2*	*Vero* se il primo operando ha un valore numerico inferiore al secondo.
op1 -le *op2*	*Vero* se il primo operando ha un valore numerico inferiore o uguale al secondo.
op1 -gt *op2*	*Vero* se il primo operando ha un valore numerico maggiore del secondo.
op1 -ge *op2*	*Vero* se il primo operando ha un valore numerico maggiore o uguale al secondo.
! *espressione*	Inverte il risultato logico dell'espressione.
espressione -a *espressione*	*Vero* se entrambe le espressioni danno un risultato *Vero*.
espressione -o *espressione*	*Vero* se almeno un'espressione dà un risultato *Vero*.
(*espressione*)	*Vero* se il risultato dell'espressione è *Vero*.

Tabella 17.50. Opzioni per l'uso del comando '**ulimit**'.

Opzione	Descrizione
-H	Viene impostato il limite fisico (*hard*) per la data risorsa. Un limite fisico non può essere aumentato una volta che è stato impostato. Se non viene specificata questa opzione, si intende l'opzione '**-S**' in modo predefinito.
-S	Viene impostato il limite logico (*soft*) per la data risorsa. Un limite logico può essere aumentato fino al valore del limite fisico. Questa opzione è predefinita se non viene specificata l'opzione '**-H**'.
-a	Sono riportati tutti i limiti correnti.
-c	La grandezza massima dei file 'core' creati, espressa in blocchi che dovrebbero essere di 512 byte.
-d	La grandezza massima del segmento dati di un processo, in multipli di 1024 byte.
-f	La grandezza massima dei file creati dalla shell, espressa in blocchi che dovrebbero essere di 512 byte.
-m	La grandezza massima della memoria occupata, in multipli di 1024 byte.
-s	La grandezza massima della pila del processo (*stack*), in multipli di 1024 byte.
-t	Il massimo quantitativo di tempo di CPU in secondi.
-n	Il numero massimo di descrittori di file aperti (la maggior parte dei sistemi non permette che questo valore sia impostato, consentendo solo la sua lettura).

17.3.4.1 Comando «getopts»

Il comando interno '**getopts**' è qualcosa di diverso dalle solite cose. Serve per facilitare la realizzazione di script in cui si devono analizzare le opzioni della riga di comando.

getopts *stringa_di_opzioni* *nome_di_variabile* [*argomenti*]

Ogni volta che viene chiamato, '**getopts**' analizza l'argomento successivo nella riga di comando, restituendo le informazioni relative attraverso delle variabili di ambiente. Per la precisione, '**getopts**' analizza gli argomenti finali della sua stessa riga di comando (quelli che sono stati indicati nello schema sintattico come un elemento facoltativo) e in mancanza di questi utilizza il contenuto del parametro **@**. Si osservi l'esempio:

getopts *stringa_di_opzioni* *nome_di_variabile*

Quanto appare sopra è esattamente uguale a:

getopts *stringa_di_opzioni* *nome_di_variabile* $@

Il comando '**getopts**' dipende in particolare dalla variabile ***OPTIND***, la quale contiene l'indice di scansione di questi argomenti. Il suo valore iniziale predefinito è pari a uno, corrispondente al primo elemento, incrementato successivamente ogni volta che si utilizza '**getopts**'. Se per qualche motivo si dovesse ripetere una scansione (degli stessi, o di altri argomenti), occorrerebbe inizializzare nuovamente tale variabile al valore uno.

Per funzionare, '**getopts**' richiede due informazioni: una stringa contenente le lettere delle opzioni previste; il nome di una variabile di ambiente da creare e inizializzare di volta in volta con il nome dell'opzione individuata. Se è previsto che un'opzione di quelle da scandire sia seguita da un argomento, quell'argomento viene inserito nella variabile di ambiente ***OPTARG***. La stringa che definisce le lettere delle opzioni è composta proprio da quelle stesse lettere, le quali possono essere seguite dal simbolo due punti ('**:**') se si vuole specificare la presenza di un argomento.

Per cominciare, si osservi l'esempio seguente, in cui viene mostrato uno script elementare anche se piuttosto lungo:

```
#!/bin/sh
echo "Indice opzioni: $OPTIND"
getopts a:b:c:defg OPZIONE -a ciao -b come -c stai -d -e -f -g -h -i
echo "Opzione \"${OPZIONE}\" con argomento ${OPTARG}."

echo "Indice opzioni: $OPTIND"
getopts a:b:c:defg OPZIONE -a ciao -b come -c stai -d -e -f -g -h -i
echo "Opzione \"${OPZIONE}\" con argomento ${OPTARG}."

echo "Indice opzioni: $OPTIND"
getopts a:b:c:defg OPZIONE -a ciao -b come -c stai -d -e -f -g -h -i
echo "Opzione \"${OPZIONE}\" con argomento ${OPTARG}."

echo "Indice opzioni: $OPTIND"
getopts a:b:c:defg OPZIONE -a ciao -b come -c stai -d -e -f -g -h -i
echo "Opzione \"${OPZIONE}\" con argomento ${OPTARG}."

echo "Indice opzioni: $OPTIND"
getopts a:b:c:defg OPZIONE -a ciao -b come -c stai -d -e -f -g -h -i
echo "Opzione \"${OPZIONE}\" con argomento ${OPTARG}."

echo "Indice opzioni: $OPTIND"
getopts a:b:c:defg OPZIONE -a ciao -b come -c stai -d -e -f -g -h -i
echo "Opzione \"${OPZIONE}\" con argomento ${OPTARG}."

echo "Indice opzioni: $OPTIND"
getopts a:b:c:defg OPZIONE -a ciao -b come -c stai -d -e -f -g -h -i
echo "Opzione \"${OPZIONE}\" con argomento ${OPTARG}."

echo "Indice opzioni: $OPTIND"
getopts a:b:c:defg OPZIONE -a ciao -b come -c stai -d -e -f -g -h -i
echo "Opzione \"${OPZIONE}\" con argomento ${OPTARG}."

echo "Indice opzioni: $OPTIND"
getopts a:b:c:defg OPZIONE -a ciao -b come -c stai -d -e -f -g -h -i
echo "Opzione \"${OPZIONE}\" con argomento ${OPTARG}."
```

Come si può notare, '**getopts**' viene avviato sempre nello stesso modo (soprattutto con gli stessi argomenti da scandire); inoltre, subito prima viene visualizzato il contenuto della variabile ***OPTIND*** e dopo viene visualizzato il risultato della scansione. Ecco cosa si ottiene:

```
Indice opzioni: 1
Opzione "a" con argomento ciao.
Indice opzioni: 3
Opzione "b" con argomento come.
Indice opzioni: 5
Opzione "c" con argomento stai.
Indice opzioni: 7
Opzione "d" con argomento .
Indice opzioni: 8
Opzione "e" con argomento .
Indice opzioni: 9
Opzione "f" con argomento .
Indice opzioni: 10
Opzione "g" con argomento .
Indice opzioni: 11
./prova.sh: illegal option -- h
```

```
Opzione "?" con argomento .
Indice opzioni: 11
./prova.sh: illegal option -- i
Opzione "?" con argomento .
```

In pratica, sono valide solo le opzioni dalla lettera «a» alla lettera «g», inoltre le prime tre (dalla «a» alla «c») richiedono un argomento. Si può osservare che le opzioni '**-h**' e '**-i**', che sono state aggiunte volutamente, sono in più e '**getopts**' ne ha segnalato la presenza come un errore.

Vale la pena di osservare anche l'andamento dell'indice rappresentato dalla variabile ***OPTIND***: nel caso delle opzioni '**-a**', '**-b**' e '**-c**', l'incremento è di due unità perché c'è anche un argomento di queste.

Il comando '**getopts**' restituisce un valore diverso da zero (*Falso*) tutte le volte che si verifica un errore. In questo modo, diventa agevole il suo inserimento al posto di un'espressione condizionale, come nell'esempio seguente, in cui si fa la scansione delle opzioni fornite allo script, ovvero quelle contenute nel parametro **@**:

```
#!/bin/sh
while getopts a:b:c:defg OPZIONE
do
    echo "Opzione \"${OPZIONE}\" con argomento ${OPTARG}."
done
```

Al primo errore, il ciclo termina e non viene mostrato il messaggio relativo.

In condizioni normali, è più probabile che si utilizzi una struttura '**case**' per analizzare la scansione delle opzioni, come nell'esempio seguente, dove è stata aggiunta anche l'inizializzazione della variabile ***OPTIND*** a titolo precauzionale, per garantire che la scansione parta dall'inizio:

```
#!/bin/sh
OPTIND=1
while getopts :a:b:c:defg OPZIONE
do
    case $OPZIONE in
        a) echo "Opzione \"a\" con argomento $OPTARG."      ;;
        b) echo "Opzione \"b\" con argomento $OPTARG."      ;;
        c) echo "Opzione \"c\" con argomento $OPTARG."      ;;
        d) echo "Opzione \"d\" che non richiede argomento." ;;
        e) echo "Opzione \"e\" che non richiede argomento." ;;
        f) echo "Opzione \"f\" che non richiede argomento." ;;
        g) echo "Opzione \"g\" che non richiede argomento." ;;
        *) echo "È stata indicata un'opzione non valida."   ;;
    esac
done
```

Questo esempio è diverso da quelli precedenti, soprattutto per la stringa di definizione delle opzioni da scandire: questa stringa inizia con il simbolo due punti ('**:**'). In questo modo, si vuole evitare che '**getopt**' restituisca *Falso* quando si verifica un errore negli argomenti.

Il comando '**getopts**' utilizza anche un'altra variabile di ambiente: ***OPTERR***. Questa variabile contiene normalmente il valore uno; se le viene assegnato zero, si inibiscono tutte le segnalazioni di errore.

17.3.4.2 Comando «set»

Il comando '**set**', se usato senza argomenti, emette l'impostazione generale della shell, nel senso che vengono visualizzate tutte le variabili di ambiente e le funzioni. Se si indicano degli argomenti si intendono alterare alcune modalità (opzioni) legate al funzionamento della shell.

```
set [{-|+}x]...
```

```
set {-|+}o [modalità]
```

```
set valore_param_1 [valore_param_2 ... [valore_param_n]]
```



```
set -- [valore_parametro_1 [valore_parametro_2...]]
```

Quasi tutte le modalità in questione sono rappresentate da una lettera alfabetica, con un qualche significato mnemonico. L'attivazione di queste modalità può essere verificata osservando il contenuto del parametro **-**:

```
$ echo $- [Invio]
```

Si potrebbe ottenere una stringa come quella seguente:

```
imH
```

Le lettere che si vedono («i», «m» e «H») rappresentano ognuna l'attivazione di una modalità particolare, dove però '**set**' può intervenire solo su alcune di queste. Gli argomenti normali del comando '**set**' sono le lettere delle modalità che si vogliono attivare o disattivare: se le lettere sono precedute dal segno '**-**' si specifica l'attivazione di queste, mentre se sono precedute dal segno '**+**' si specifica la loro disattivazione.

Il comando '**set**' può essere usato per modificare le modalità di funzionamento anche attraverso l'opzione '**-o**', oppure '**+o**', che deve essere seguita da una parola chiave che rappresenta la modalità stessa. I segni '**-**' e '**+**' rappresentano ancora l'attivazione o la disattivazione della modalità corrispondente. Le modalità a cui si accede attraverso l'opzione '**-o**' (o '**+o**') non sono esattamente le stesse che si possono controllare altrimenti.

Il comando '**set**' può servire anche per modificare il contenuto dei parametri posizionali. Per questo, se ci sono degli argomenti che seguono la definizione dell'ultima modalità, vengono interpretati come i valori da assegnare ordinatamente a questi parametri. In particolare, se si utilizza la forma '**set --**', si interviene su tutti i parametri: se non si indicano argomenti, si eliminano tutti i parametri; se ci sono altri argomenti, questi diventano ordinatamente i nuovi parametri, mentre tutti quelli precedenti vengono eliminati.

Se viene utilizzato il comando '**set -o**', si ottiene l'elenco delle impostazioni attuali.

Il comando '**set**' restituisce *Vero* se non viene incontrata un'opzione errata.

Segue la descrizione di alcune modalità, che potrebbero essere riconosciute dalla maggior parte delle shell POSIX.

Modalità	Descrizione
{-|+}a {-|+}o allexport	Le variabili che vengono modificate o create, sono marcate automaticamente per l'esportazione verso l'ambiente per i comandi avviati dalla shell.
{-|+}b {-|+}o notify	Fa in modo che venga riportato immediatamente lo stato di un gruppo di elaborazione (*job*) sullo sfondo che termina. Altrimenti, questa informazione viene emessa subito prima dell'invito primario successivo.
{-|+}e {-|+}o errexit	Termina immediatamente se un comando qualunque conclude la sua esecuzione restituendo uno stato diverso da zero. La shell non esce se il comando che fallisce è parte di un ciclo '**until**' o '**while**', di un'istruzione '**if**', di una lista '**&&**' o '**||**', o se il valore restituito dal comando è stato invertito per mezzo di '**!**'.
{-|+}f {-|+}o noglob	Disabilita l'espansione di percorso (quello che riguarda i caratteri jolly, o metacaratteri, nei nomi di file e directory).
{-|+}m {-|+}o monitor	Abilita il controllo dei gruppi di elaborazione. Questa modalità è attiva in modo predefinito per le shell interattive.
{-|+}n {-|+}o noexec	Legge i comandi, ma non li esegue. Ciò può essere usato per controllare gli errori di sintassi di uno script di shell. Questo valore viene ignorato dalle shell interattive.

Modalità	Descrizione
{-|+}u {-|+}o nounset	Fa in modo che venga considerato un errore l'utilizzo di variabili non impostate (predisposte) quando si effettua l'espansione di una variabile (o di un parametro). In tal caso, quindi, la shell emette un messaggio di errore e, se il funzionamento non è interattivo, termina restituendo un valore diverso da zero.
{-|+}v {-|+}o verbose	Emette le righe inserite nella shell appena queste vengono lette.
{-|+}x {-|+}o xtrace	Nel momento in cui si eseguono dei comandi, viene emesso il comando stesso attraverso lo standard output preceduto da quanto contenuto nella variabile ***PS4***.
{-|+}C {-|+}o noclobber	Disabilita la sovrascrittura dei file preesistenti a seguito di una ridirezione dell'output attraverso l'uso degli operatori '**>**', '**>&**' e '**<>**'. Questa impostazione può essere scavalcata (in modo da riscrivere i file) utilizzando l'operatore di ridirezione '**>|**' al posto di '**>**'. In generale sarebbe meglio evitare di intervenire in questo modo, dal momento che ciò non è conforme all'utilizzo normale.

L'esempio seguente mostra come modificare il gruppo dei parametri posizionali:

```
$ set -- ciao come stai? [Invio]
$ echo $1 [Invio]

  ciao
$ echo $2 [Invio]

  come
$ echo $3 [Invio]

  stai?
```

17.4 Accesso ai file

La shell POSIX ha una capacità limitata ad accedere ai file in modo sequenziale. Ciò consente di fare qualcosa di interessante negli script.

17.4.1 Utilizzo dei descrittori

Generalmente, si utilizzano i tre flussi standard in modo intuitivo, senza la necessità di aprirli o di chiuderli. Quando si vogliono gestire più flussi di dati simultaneamente, occorre attivare altri descrittori.

Tabella 17.60. Descrittori standard.

Descrittore	Apertura	Standard
0	lettura	standard input
1	scrittura	standard output
2	scrittura	standard error

I descrittori dei flussi standard risultano aperti senza bisogno di una richiesta esplicita; per aprire dei descrittori ulteriori, occorre dare delle istruzioni appropriate nella riga di comando:

Comando	Descrizione
comando ... n< file	Apre il file in lettura e ne consente l'accesso al comando attraverso il descrittore numero ***n***.
comando ... n> file	Apre il file in scrittura, azzerandolo inizialmente, concedendone l'accesso al comando attraverso il descrittore numero ***n***.
comando ... n>> file	Apre il file in scrittura, per l'aggiunta di dati in coda, concedendone l'accesso al comando attraverso il descrittore numero ***n***.

Generalmente, i descrittori aperti durante l'esecuzione di uno script, vengono chiusi automaticamente al termine del funzionamento dello stesso; in alternativa possono essere chiusi esplicitamente con la sintassi seguente:

```
n<&-
```

È da osservare che in questo modo si chiude il descrittore ***n***, indipendentemente dal fatto che questo rappresenti un flusso in ingresso (lettura) o in uscita (scrittura).

All'interno di uno script è possibile aprire dei descrittori alla chiamata di una funzione, come nell'esempio seguente:

```
#!/bin/sh
…
function fa_qualcosa () {
    …
}
…
fa_qualcosa 3< mio_file
…
```

Teoricamente, è possibile aprire un descrittore **in lettura** in modo più semplice, senza che ciò debba avvenire necessariamente alla chiamata di un comando, ma l'uso di una shell poco amichevole, sotto questo punto di vista, potrebbe rendere la cosa del tutto inutile. Pertanto, pur essendo un procedimento sconsigliabile, ecco come si potrebbe procedere:

```
#!/bin/sh
…
3< mio_file    # apre il file nel descrittore n. 3
…
# qui si fa qualcosa con il descrittore n. 3
…
3<&-    # chiude il file associato al descrittore n. 3
…
```

17.4.2 Lettura e scrittura

La lettura da un descrittore si ottiene con il comando '**read**':

```
read [-p prompt] variabile...
```

Il comando '**read**' legge dallo standard input, pertanto, per leggere un flusso di dati proveniente attraverso un altro descrittore, occorre inserirlo nello standard input:

```
read [-p prompt] variabile... <&n
```

Per leggere i dati provenienti da un file, aperto attraverso il descrittore ***n***, si può usare un ciclo simile a quello seguente, dove si deve elaborare una riga alla volta del file originale:

```
while read RIGA_DEL_FILE <&n
do
    # Fa qualcosa con il contenuto della variabile di
    # ambiente RIGA_DEL_FILE.
    …
done
```

Per scrivere qualcosa all'interno di un descrittore, si usano normalmente i comandi '**echo**' o '**printf**'. Entrambi questi comandi scrivono attraverso lo standard output, pertanto, anche in questo caso, occorre ridirigere il flusso verso il descrittore desiderato, se necessario. L'esempio seguente legge un file attraverso il descrittore ***n*** e lo emette, tale e quale, attraverso il descrittore ***m***, elaborando i dati riga per riga:

```
while read RIGA_DEL_FILE <&n
do
    echo "$RIGA_DEL_FILE" >&m
done
```

L'esempio successivo legge dallo standard input e scrive attraverso lo standard output, dopo aver trasformato il testo in maiuscolo (considerando soltanto l'alfabeto latino senza lettere accentate):

```
#!/bin/sh
while read RIGA
do
    echo $RIGA | tr a-z A-Z
done
```

L'esempio seguente fa la stessa cosa utilizzando il descrittore numero 3 per la lettura del file e il descrittore numero 4 per la scrittura:

```
#!/bin/sh
while read RIGA <&3
do
    echo $RIGA | tr a-z A-Z >&4
done
```

L'esempio seguente utilizza una funzione per la trasformazione dei dati, rendendo esplicita l'apertura e la chiusura dei descrittori:

```
#!/bin/sh
function elabora () {
    while read RIGA <&3
    do
        echo $RIGA | tr a-z A-Z >&4
    done
}
elabora 3< file_da_leggere 4> file_da_creare
3<&-
4<&-
```

L'esempio seguente **non funziona**, perché non c'è modo di aprire il descrittore in scrittura secondo la modalità che qui viene mostrata:

```
#!/bin/sh
3< file_da_leggere
4> file_da_creare    # Non funziona!
while read RIGA <&3
do
    echo $RIGA | tr a-z A-Z >&4
done
3<&-
4<&-
```

17.4.2.1 Gestione degli spazi

Il comando '**read**' interpreta il contenuto delle righe in base alla configurazione stabilita con la variabile di ambiente ***IFS***. In pratica, per quanto riguarda gli esempi proposti e l'impostazione usuale di questa variabile, ciò significa che gli spazi orizzontali presenti all'inizio e alla fine delle righe, vengono eliminati.

Per evitare questo tipo di trattamento degli spazi occorrerebbe intervenire nella variabile di ambiente ***IFS***, ma non è detto che il risultato che si ottiene sia corretto. Pertanto, conviene limitarsi all'uso «normale» del comando '**read**', considerando la perdita degli spazi orizzontali iniziali e finali.

17.4.3 Contenuto senza file

Invece di aprire un file in lettura per fornirlo a un descrittore, è possibile inviare al descrittore direttamente il contenuto, attraverso il meccanismo noto come *here document*. Vengono messe a confronto le due forme:

```
n< file
```

```
n<< "marcatore_conclusivo"
    testo
    ...
marcatore_conclusivo
```

L'esempio seguente ne riprende uno già proposto in precedenza:


```
#!/bin/sh
function elabora () {
    while read RIGA <&3
    do
        echo $RIGA | tr a-z A-Z >&4
    done
}
elabora 4> file_da_creare 3<< "FINE_DEL_TESTO"
    bla bla bla bla bla
    bla bla bla bla bla
    bla bla bla bla bla
    bla bla bla bla bla
FINE_DEL_TESTO
3<&-
4<&-
```

17.5 Traduzione dei messaggi

È possibile fare in modo che i messaggi generati all'interno di uno script vengano tradotti automaticamente, attraverso Gettext, descritto nella sezione 71.[6] Per ottenere questo, si incorpora in uno script il codice contenuto nel file '`gettext.sh`', il quale potrebbe risultare installato nella directory '`/usr/bin/`', come parte del pacchetto che compone proprio Gettext; quindi, nello script si fa uso delle funzioni '**`gettext`**' e '**`eval_gettext`**', per ottenere la traduzione dei messaggi.

17.5.1 Esempio iniziale

Per cominciare a comprendere il meccanismo, conviene partire da un esempio molto semplice, vedendo dall'inizio alla fine il procedimento. Si suppone che il file seguente sia denominato '`bye-bye.sh`':

```
1  #!/bin/sh
2  TEXTDOMAINDIR=/tmp
3  TEXTDOMAIN=bye-bye
4  export TEXTDOMAINDIR
5  export TEXTDOMAIN
6
7  . /usr/bin/gettext.sh
8
9  tizio="daniele"
10
11 eval_gettext "bye bye \$tizio"
12 echo ""
13 gettext      "bye bye \$tizio"
14 echo ""
```

Si può osservare che nella settima riga viene inserito il codice contenuto nel file '`/usr/bin/gettext.sh`', quindi si rendono disponibili le funzioni '**`eval_gettext`**' e '**`gettext`**'. Nelle righe numero 11 e numero 13, si vede l'uso delle due funzioni, però con gli stessi argomenti; lì si deve osservare che in entrambi i casi, il dollaro che precede il nome della variabile '**`tizio`**' è stato protetto in modo da non essere espanso dalla shell.

Una volta realizzato il file, si utilizza '**`xgettext`**' per generare il file '`messages.po`', da tradurre. In questo caso, il nome '`bye-bye.sh`' è stato scelto appositamente con l'estensione '**`.sh`**', per facilitare a '**`xgettext`**' il riconoscimento del contesto:

```
$ xgettext bye-bye.sh [Invio]
```

Se non si commettono errori, si ottiene così il file '`messages.po`', con il contenuto seguente:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2006-04-04 19:24+0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
```

```
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: bye-bye.sh:10 bye-bye.sh:12
#, sh-format
msgid "bye bye $tizio"
msgstr ""
```

Il file va modificato, soprattutto per ciò che riguarda la traduzione. In tal caso, conviene creare il file ‘bye-bye.po’:

```
# bye-bye.sh PO file.
# Copyright (C) 2006 Pinco Pallino
# Pinco Pallino <ppinco@dinkel.brot.dg>, 2006.
#
msgid ""
msgstr ""
"Project-Id-Version: 0.1\n"
"Report-Msgid-Bugs-To: Pinco Pallino <ppinco@dinkel.brot.dg>\n"
"POT-Creation-Date: 2006-04-04 18:59+0200\n"
"PO-Revision-Date: 2006-04-04 18:59+0200\n"
"Last-Translator: Pinco Pallino <ppinco@dinkel.brot.dg>\n"
"Language-Team: Italian <it@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: bye-bye.sh:10 bye-bye.sh:12
#, sh-format
msgid "bye bye $tizio"
msgstr "ciao ciao $tizio"
```

Si passa quindi alla compilazione del file, con la quale si vuole ottenere il file ‘bye-bye.mo’:

```
$ msgfmt -vvvv -o bye-bye.mo bye-bye.po [Invio]
```

Il file ‘bye-bye.mo’, tenuto conto che è stato realizzato per la configurazione locale italiana, va collocato all’interno del percorso ‘it/LC_MESSAGES/’ che, a sua volta, deve partire da quanto contenuto nella directory indicata nella variabile ***TEXTDOMAINDIR***, oppure nella collocazione predefinita che potrebbe essere ‘/usr/share/locale/’. In questo caso, nello stesso script appare la dichiarazione e l’esportazione della variabile di ambiente ***TEXTDOMAINDIR*** (pertanto l’eventuale collocazione predefinita non viene considerata), con un valore tale per cui il file ‘bye-bye.mo’ deve trovarsi nella directory ‘/tmp/it/LC_MESSAGES/’. Inoltre, la variabile ***TEXTDOMAINDIR*** stabilisce che sia proprio il file ‘bye-bye.mo’ quello che deve essere cercato.

Una volta collocato correttamente il file ‘bye-bye.mo’, se la configurazione locale è quella della lingua italiana, lo script funziona mostrando i messaggi tradotti:

```
$ LANG=it_IT.UTF-8 [Invio]

$ export LANG [Invio]

$ ./bye-bye.sh [Invio]

  ciao ciao daniele
  ciao ciao $tizio
```

Il risultato che si ottiene mostra il comportamento delle due funzioni: ‘**eval_gettext**’ e ‘**gettext**’. Entrambe le funzioni restituiscono una stringa tradotta, senza aggiungere un codice di interruzione di riga; pertanto, nello script si manda a capo il testo con due comandi ‘**echo**’ vuoti. Nel caso della funzione ‘**eval_gettext**’, la stringa originaria viene scandita alla ricerca di variabili da espandere, mentre la funzione ‘**gettext**’ si limita a lasciare inalterata la stringa tradotta.

Si deve osservare che la chiamata delle funzioni ‘***gettext**’ è stata fatta volutamente proteggendo il dollaro davanti al nome della variabile ‘**tizio**’, per evitare che la stringa passata alle funzioni stesse venga espansa preliminarmente dalla shell. Infatti, se ciò accadesse, sarebbe inutile tentare di tradurre i messaggi. Naturalmente, l’uso di una variabile con la funzione ‘**gettext**’ diventa del tutto inutile, ma qui serve a dimostrare la differenza di comportamento tra le due funzioni.

17.5.2 utilizzo più sofisticato

Non è semplice usare le funzioni ‘***gettext**’ così come sono, all’interno di uno script di shell, rispetto a come si può fare invece con un linguaggio di programmazione normale. Qui si propone l’uso di una funzione che serve a estendere le capacità di ‘**printf**’, con la dimostrazione dei raggiri necessari a ottenere il funzionamento del sistema di Gettext. Si suppone che il file seguente sia denominato ‘bye-bye-2.sh’:

```
1  #!/bin/sh
2  TEXTDOMAINDIR=/tmp
3  TEXTDOMAIN=bye-bye-2
4  export TEXTDOMAINDIR
5  export TEXTDOMAIN
6
7  . /usr/bin/gettext.sh
8
9  printf_gettext () {
10     local string="$1"
11     shift
12     string=‘gettext "$string"‘
13     printf "$string" "$@"
14 }
15
16 tizio="daniele"
17
18 printf_gettext "bye bye %s\n" $tizio
```

Per ottenere il file ‘messages.po’, occorre imbrogliare ‘**xgettext**’, fingendo che la funzione ‘**printf_gettext**’ sia invece soltanto ‘**gettext**’:

```
$ cat bye-bye-2.sh ↩
↪     | sed "s/printf_gettext/gettext/" ↩
↪     | xgettext -L Shell - [Invio]
```

Se non si commettono errori, si ottiene così il file ‘messages.po’, con il contenuto seguente:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2006-04-05 10:10+0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: standard input:19
msgid "bye bye %s\\n"
msgstr ""
```

Come già nella sezione precedente, il file va modificato, soprattutto per ciò che riguarda la traduzione. In tal caso, conviene creare il file ‘bye-bye-2.po’:

```
# bye-bye-2.sh PO file.
# Copyright (C) 2006 Pinco Pallino
# Pinco Pallino <ppinco@dinkel.brot.dg>, 2006.
#
msgid ""
msgstr ""
"Project-Id-Version: 0.1\n"
"Report-Msgid-Bugs-To: Pinco Pallino <ppinco@dinkel.brot.dg>\n"
```

```
"POT-Creation-Date: 2006-04-04 18:59+0200\n"
"PO-Revision-Date: 2006-04-04 18:59+0200\n"
"Last-Translator: Pinco Pallino <ppinco@dinkel.brot.dg>\n"
"Language-Team: Italian <it@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: standard input:19
msgid "bye bye %s\\n"
msgstr "ciao ciao %s\\n"
```

Si passa quindi alla compilazione del file, con la quale si vuole ottenere il file '`bye-bye-2.mo`':

```
$ msgfmt -vvvv -o bye-bye-2.mo bye-bye-2.po [Invio]
```

Anche in questo caso, data la configurazione delle variabili ***TEXTDOMAINDIR*** e ***TEXTDOMAIN***, il file va collocato nella directory '`/tmp/it/LC_MESSAGES/`'.

```
$ LANG=it_IT.UTF-8 [Invio]

$ export LANG [Invio]

$ ./bye-bye-2.sh [Invio]

  ciao ciao daniele
```

17.6 Libreria Readline

Diversi programmi che funzionano in modo interattivo mostrando un invito all'inserimento dei comandi (un *prompt*) e offrendo una riga di comando, sfruttano la libreria Readline[7] per la gestione di uno storico dei comandi e per offrire altre funzionalità come il completamento automatico.

La libreria Readline è sottoposta alle condizioni della licenza GNU GPL, pertanto i programmi che la incorporano vengono distribuiti alle stesse condizioni.

Questa libreria offre funzionalità così raffinate che spesso chi utilizza programmi interattivi che se ne avvalgono, si limita a sfruttarne una porzione minima. Questo capitolo dà solo una visione limitata delle funzionalità disponibili; chi desidera approfondire lo studio può cercare la sua documentazione che potrebbe essere disponibile nella pagina di manuale *readline(3)*, o in sua mancanza nella documentazione della shell Bash, che potrebbe essere disponibile come *info bash* oppure *bash(1)*.

Nel capitolo vengono mostrate diverse tabelle che descrivono l'uso dei comandi comuni disponibili; si osservi però che molte cose possono essere ridefinite attraverso la configurazione che avviene normalmente attraverso il file '`~/.readline`'.

17.6.1 Comandi

Quando la configurazione della libreria Readline è realizzata nel modo corretto, i tasti freccia consentono di scorrere all'interno dello storico e all'interno di un comando per consentirne la modifica, così come altri tasti di spostamento funzionano in modo intuitivo; diversamente sono disponibili delle combinazioni di tasti standard, secondo lo schema delle tabelle successive.

Tabella 17.79. Alcuni comandi di spostamento e affini.

Comando	Significato mnemonico	Descrizione
[*Ctrl a*]		Sposta il cursore all'inizio della riga. Di solito si predispone la configurazione in modo che il tasto [*Inizio*] svolga questa funzione.
[*Ctrl e*]	*end*	Sposta il cursore alla fine della riga. Di solito si predispone la configurazione in modo che il tasto [*Fine*] svolga questa funzione.

Comando	Significato mnemonico	Descrizione
[*Ctrl f*]	*forward*	Sposta il cursore a destra di un carattere. Di solito si predispone la configurazione in modo che il tasto [*freccia-destra*] svolga questa funzione.
[*Ctrl b*]	*backward*	Sposta il cursore a sinistra di un carattere. Di solito si predispone la configurazione in modo che il tasto [*freccia-sinistra*] svolga questa funzione.
[*Ctrl l*]		Ripulisce lo schermo.

Tabella 17.80. Alcuni comandi per la modifica del testo della riga di comando.

Comando	Significato mnemonico	Descrizione
[*Ctrl d*]	*delete*	Cancella il carattere sopra il cursore. Di solito si predispone la configurazione in modo che il tasto [*Canc*] svolga questa funzione.
[*Backspace*]		Cancella il carattere a sinistra del cursore.
[*Ctrl v*]	*verbatim*	Dopo questa sequenza è possibile inserire un carattere in modo letterale, quando non è possibile farlo in condizioni normali.

Tabella 17.81. Alcuni comandi per l'utilizzo dello storico. Vale la pena di precisare che il termine «storico», in un contesto come questo, intende fare riferimento a un «archivio storico» o un «registro storico» di qualcosa.

Comando	Significato mnemonico	Descrizione
[*Invio*]		Conferma l'inserimento del contenuto della riga, anche se il cursore non si trova alla fine della stessa. Ciò che viene inserito è accumulato automaticamente nello storico.
[*Ctrl p*]	*previous*	Recupera dallo storico l'ultimo comando inserito o comunque quello precedente a quello che si vede sulla riga di comando. Di solito si predispone la configurazione in modo che il tasto [*freccia-su*] svolga questa funzione.
[*Ctrl n*]	*next*	Scorre in avanti l'elenco dei comandi nello storico. Di solito si predispone la configurazione in modo che il tasto [*freccia-giù*] svolga questa funzione.
[*Ctrl r*]	*search backward*	Inizia la ricerca di un comando all'indietro.

17.6.2 Completamento automatico

Durante l'inserimento di un comando, si può usare il tabulatore, [*Tab*], per ottenere il completamento di questo in base a qualche criterio, dipendente dall'applicazione in cui la libreria viene usata.

Generalmente, la pressione del tasto [*Tab*] porta al completamento di qualcosa, se il contesto non permette di avere dubbi, altrimenti il completamento può essere parziale o imposibile. Quando il completamento è ambiguo, la ripetizione del comando produce la visualizzazione dell'elenco delle alternative disponibili.

17.6.3 Configurazione

La configurazione usata dalla libreria Readline avviene normalmente attraverso il file '`/etc/inputrc`' in modo generale, mentre per i singoli utenti attraverso il file '`~/.inputrc`'.

La cosa più comune che viene definita nel file di configurazione è l'uso di tasti per lo spostamento del cursore e per lo scorrimento nello storico, oltre alle combinazioni già previste. L'esempio seguente si riferisce alla configurazione necessaria per l'uso ottimale di una console virtuale, in un sistema GNU/Linux, su un elaboratore con architettura x86:

```
# Abilita l'inserimento di caratteri a 8 bit.
set meta-flag           on
# Disabilita la conversione dei caratteri con l'ottavo bit
# attivo in sequenze di escape.
set convert-meta        off
# Abilita la visualizzazione di caratteri a 8 bit.
set output-meta         on
# Modifica l'abbinamento con i tasti rispetto a determinati
# comportamenti.
"\e[1~": beginning-of-line      # [home]            era C-a
"\e[4~": end-of-line            # [fine]            era C-e
"\e[3~": delete-char            # [canc]            era C-d
"\e[5~": backward-word          # [pagina su]       era M-b
"\e[6~": forward-word           # [pagina giù]      era M-f
```

Come si intuisce, non sono stati abbinati i tasti [*freccia-sinistra*] e [*freccia-destra*], che in condizioni normali funzionano al pari delle combinazioni [*Ctrl b*] e [*Ctrl f*].

17.6.4 Utilizzo di «cle»

Il programma '**cle**',[8] ovvero *Command line editor*, è un involucro per i programmi interattivi che funzionano attraverso una riga di comando, ma non dispongono di funzionalità simili a quelle offerte dalla libreria Readline:

```
cle [opzioni] programma [argomenti]
```

In pratica, si usa '**cle**' per avviare un altro programma, il quale può avere bisogno dei suoi argomenti, controllando l'inserimento dei dati provenienti dallo standard input.

Si può tentare di capire cosa fa questo programma con un esempio realizzato con comandi comuni:

```
$ tee prova [Invio]

Ciao, [Invio]

  Ciao,

come stai? [Invio]

  come stai?

[Ctrl d]
```

Al termine il file '`prova`' contiene esattamente il testo:

```
Ciao,
come stai?
```

Durante l'inserimento del testo, non è possibile correggere la riga se non a partire dalla cancellazione dalla fine; con l'aiuto di '**cle**' si ottiene tutta la potenza della libreria Readline, compresa la gestione dello storico:

```
$ cle tee prova [Invio]

...
```

In questo caso, lo storico viene accumulato precisamente nel file '`~/.tee_history`', pertanto si intuisce che controllando un altro programma si ottiene un file con il prefisso che richiama il nome dello stesso.

Bisogna tenere presente però che '**cle**' non è perfetto per tutte le circostanze: prima di tutto è necessario che il programma che viene controllato riceva i dati dallo standard input, perché se usa invece una tecnica differente, il meccanismo non può funzionare; inoltre,

se il programma controllato mostra un invito, quando si scorre lo storico questo viene eliminato, perché '**cle**' non ne è consapevole.

17.7 Tabelle riepilogative

17.7.1 Particolarità della shell Bash

Tabella 17.29. A seconda del modo con cui l'eseguibile della shell Bash viene avviato si utilizzano diversi tipi di file di configurazione.

Comando	Tipo	All'avvio	Alla conclusione
`bash`	login	'`/etc/profile`', più '`~/.bash_profile`', oppure '`~/.bash_login`', oppure '`~/.profile`'	'`~/.bash_logout`'
`bash`	interattiva	'`~/.bashrc`'	
`bash`	non interattiva	il file indicato nella variabile di ambiente ***BASH_ENV***	
`sh`	login	'`/etc/profile`', più '`~/.profile`'	
`sh`	interattiva	il file indicato nella variabile di ambiente ***ENV***	
`sh`	non interattiva	--	
`bash --posix`	login	il file indicato nella variabile di ambiente ***ENV***	
`bash --posix`	interattiva	il file indicato nella variabile di ambiente ***ENV***	
`bash --posix`	non interattiva	--	

Tabella 17.33. Elenco di alcuni codici speciali per definire l'invito con la shell Bash.

Codice	Descrizione
`\t`	Orario attuale nel formato ***hh***:***mm***:***ss*** (ore, minuti, secondi).
`\d`	Data attuale.
`\n`	Interruzione di riga.
`\s`	Nome della shell.
`\w`	Percorso assoluto della directory corrente.
`\W`	Nome finale del percorso della directory corrente (*basename*).
`\u`	Utente.
`\h`	Nome dell'elaboratore.
`\#`	Numero del comando attuale.
`\!`	Numero del comando nello storico.
`\$`	'`#`' se UID = 0; '`$`' se UID > 0.
`\`***nnn***	Carattere corrispondente al numero ottale indicato.
`\\`	Una barra obliqua inversa singola ('`\`').
`\[`	Inizio di una sequenza di controllo.
`\]`	Fine di una sequenza di controllo.

17.7.2 Parametri comuni

Parametro	Descrizione
n	Un parametro posizionale è definito da una o più cifre numeriche a eccezione dello zero che ha invece un significato speciale. I parametri posizionali rappresentano gli argomenti forniti al comando: '**$1**' si espande nel primo, '**$2**' si espande nel secondo e così di seguito. Quando si utilizza un parametro composto da più di una cifra numerica, è indispensabile racchiuderlo tra parentesi graffe; per esempio: '**${10}**', '**${11}**',...
`0`	Restituisce il nome della shell o dello script. Se la shell viene avviata con un file di comandi, '**$0**' si espande nel nome di quel file. Se la shell viene avviata con l'opzione '**-c**', '**$0**' si espande nel primo argomento dopo la stringa dei comandi (sempre che ce ne sia uno).
`*`	L'asterisco rappresenta l'insieme di tutti i parametri posizionali a partire dal primo. Quando viene utilizzato all'interno di apici doppi, rappresenta un'unica parola composta dal contenuto dei parametri posizionali, spaziati dal primo carattere contenuto nella variabile speciale ***IFS***. Se questa variabile non è definita, viene utilizzato uno spazio singolo. Per esempio, se ***IFS*** contenesse la sequenza '**xyz**', '**"$*"**' sarebbe equivalente a '**"$1x$2x…"**'. La variabile di shell ***IFS*** contiene di solito la sequenza: *<SP><HT><LF>* (corrispondente a uno spazio normale, un carattere di tabulazione e al codice di interruzione di riga nella maggior parte dei sistemi Unix). Di conseguenza, viene utilizzato normalmente il carattere spazio (*<SP>*) per staccare i vari parametri posizionali. Per cui, di solito, '**"$*"**' equivale a '**"$1 $2…"**'.
`@`	Rappresenta l'insieme di tutti i parametri posizionali a partire dal primo. Quando viene utilizzato all'interno di apici doppi, rappresenta delle parole, ognuna composta dal contenuto del parametro posizionale rispettivo. Di conseguenza, '**"$@"**' equivale a '**"$1" "$2" … "$*n*"**'. Questo comportamento rappresenta un'eccezione rispetto agli altri parametri che invece si limitano a generare una sola parola.
`#`	Rappresenta il numero di parametri posizionali esistenti.
`?`	Rappresenta il valore restituito dall'ultimo condotto eseguito in primo piano (*foreground*). In pratica, restituisce il valore dell'ultimo comando eseguito.
`-`	Il trattino rappresenta la serie di lettere corrispondenti alle modalità configurabili attraverso il comando interno '**set**' o con opzioni particolari della riga di comando.
`$`	Restituisce il numero PID della shell. Se viene utilizzato all'interno di una subshell, cioè tra parentesi tonde, restituisce il numero PID della shell principale e non quello della subshell.
`!`	Restituisce il numero PID del processo avviato più di recente e messo sullo sfondo.

17.7.3 Variabili di ambiente comuni

Tabella 17.9. Elenco delle variabili più importanti di una shell POSIX.

Variabile	Contenuto
`PWD`	La directory corrente. Il contenuto della variabile viene modificato dal comando '**cd**'.
`OLDPWD`	La directory corrente visitata precedentemente. Il contenuto della variabile viene modificato dal comando '**cd**'.
`PPID`	Il numero PID del processo genitore della shell attuale.
`IFS`	*Internal field separator*. Il contenuto predefinito della variabile dovrebbe essere: *<SP><HT><LF>*.
`PATH`	I percorsi di ricerca per i comandi, separati dal carattere '**:**'.
`HOME`	La directory personale dell'utente.
`CDPATH`	Il percorso di ricerca per il comando '**cd**' (di solito la variabile contiene la stringa nulla).
`MAIL`	Il percorso del file che rappresenta la cartella di posta in entrata dell'utente.

Variabile	Contenuto
`MAILCHECK`	La frequenza, in secondi, con cui si deve verificare la presenza di messaggi nuovi nella cartella corrispondente alla variabile ***MAIL***. Se ***MAILCHECK*** è vuota o contiene il valore zero, il controllo avviene ogni volta che deve essere emesso un nuovo invito.
`MAILPATH`	Questa variabile, se definita, prevale su ***MAIL*** e definisce un elenco di percorsi per altrettante cartelle di posta elettronica alternative. L'elenco è separato con il carattere '**:**'.
`OPTIND`	Contiene l'indice del prossimo argomento da elaborare dal comando '**getopts**'.
`OPTARG`	Il valore dell'ultimo argomento elaborato da '**getopts**'.
`PS1`	L'invito primario. Di solito, il valore predefinito di questa variabile fa sì che sia rappresentato un dollaro o un cancelletto a seconda che si tratti di un utente comune o dell'utente '**root**'.
`PS2`	L'invito secondario, che appare quando si deve completare un comando. Il valore predefinito è normalmente '**>** '.
`ENV`	Il nome di un file di configurazione per una shell POSIX.

17.7.4 Espansione e sostituzione

Espansioni e sostituzioni relative a parametri, variabili, comandi ed espressioni:

Modello	Descrizione
$***parametro*** | ${***parametro***} $***variabile*** | ${***variabile***}	In uno di questi modi si ottiene la sostituzione del parametro o della variabile con il suo contenuto.
$(***comando***) ‘ ***comando*** ‘	Sostituzione di comando: quanto emesso attraverso lo standard output dal comando viene usato nell'espansione.
$((***espressione***))	Esegue l'espressione aritmetica e si espande nel suo risultato.

Tabella 17.46. Operatori aritmetici.

Operatore e operandi	Descrizione
`+`***op***	Non ha alcun effetto.
`-`***op***	Inverte il segno dell'operando.
op1 `+` ***op2***	Somma i due operandi.
op1 `-` ***op2***	Sottrae dal primo il secondo operando.
op1 `*` ***op2***	Moltiplica i due operandi.
op1 `/` ***op2***	Divide il primo operando per il secondo.
op1 `%` ***op2***	Modulo: il resto della divisione tra il primo e il secondo operando.
var `=` ***valore***	Assegna alla variabile il valore alla destra.
op1 `+=` ***op2***	***op1*** `=` ***op1*** `+` ***op2***
op1 `-=` ***op2***	***op1*** `=` ***op1*** `-` ***op2***
op1 `*=` ***op2***	***op1*** `=` ***op1*** `*` ***op2***
op1 `/=` ***op2***	***op1*** `=` ***op1*** `/` ***op2***
op1 `%=` ***op2***	***op1*** `=` ***op1*** `%` ***op2***

Espansione relativa a nomi di file e di directory:

Modello	Descrizione
`~`	Corrisponde al contenuto della variabile di ambiente ***HOME*** (la directory personale dell'utente che sta usando la shell.
`~`***utente***	Corrisponde alla directory personale dell'utente.

Modello	Descrizione
*	Corrisponde a qualsiasi stringa, compresa la stringa nulla.
?	Corrisponde a un carattere qualsiasi (uno solo).
[...]	Corrisponde a uno qualsiasi dei caratteri racchiusi tra parentesi quadre.
[!...]	Corrisponde a tutti i caratteri esclusi quelli indicati.
[*a*-*z*]	Corrisponde a uno qualsiasi dei caratteri compresi nell'intervallo da ***a*** a ***z***.
[!*a*-*z*]	Corrisponde a tutti i caratteri esclusi quelli appartenenti all'intervallo indicato.

17.7.5 Comandi e *job*

Sintassi	Descrizione
[!] ***comando_1*** [| ***comando_2***...]	Condotto.
comando_1 ; ***comando_2***	Esegue il primo comando e al termine avvia il secondo.
comando &	Avvio sullo sfondo (*background*).
comando_1 & ***comando_2***	Avvia sullo sfondo il primo comando e avvia immediatamente il secondo comando.
comando_1 && ***comando_2***	Esegue il primo comando e se ciò avviene con successo, esegue anche il secondo comando.
comando_1 || ***comando_2***	Esegue il primo comando e se questo restituisce *Falso* esegue anche il secondo comando.
(***comando_1*** ; ***comando_2*** ; ...)	Lista di comandi da eseguire in una subshell.
{ ***comando_1*** ; ***comando_2*** ; ... ; }	Lista di comandi da eseguire normalmente concatenando l'output generato (il contenuto deve essere separato dalle parentesi graffe).

Riferimento ai gruppi di elaborazione	Descrizione
%***n***	Il simbolo '**%**' seguito da un numero fa riferimento al gruppo di elaborazione con quel numero.
%***prefisso***	Il simbolo '**%**' seguito da una stringa fa riferimento a un gruppo di elaborazione con un nome che inizia con quel prefisso. Se esiste più di un gruppo di elaborazione sospeso con lo stesso prefisso si ottiene una segnalazione di errore.
%?***stringa***	Il simbolo '**%**' seguito da '**?**' e da una stringa fa riferimento a un gruppo di elaborazione con una riga di comando contenente quella stringa. Se esiste più di un gruppo di elaborazione del genere si ottiene una segnalazione di errore.
%% %+	Le notazioni '**%%**' o '**%+**' fanno riferimento al gruppo di elaborazione corrente dal punto di vista della shell, il quale corrisponde all'ultimo a essere stato sospeso quando questo si trovava a funzionare in primo piano.
%-	La notazione '**%-**' fa riferimento al penultimo gruppo di elaborazione sospeso. Utilizzando i comandi '**bg**' e '**fg**', in mancanza di un riferimento esplicito al gruppo di elaborazione, viene preso in considerazione quello «corrente» dal punto di vista della shell.

17.7.6 Ridirezione

Tabella 17.22. Sintassi per la ridirezione.

Sintassi	Descrizione
[***n***]< ***file***	La ridirezione 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 zero.

Sintassi	Descrizione
[***n***]> ***file***	La ridirezione 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 numero uno. Di solito, se il file da aprire in scrittura esiste già, viene sovrascritto.
[***n***]>> ***file***	La ridirezione dell'output fatta in questo modo fa sì che se il file da aprire in scrittura esiste già, questo non sia sovrascritto, ma gli siano semplicemente aggiunti i dati.
<<[-] ***parola_di_delimitazione*** ***testo*** ... ***parola_di_delimitazione***	Si tratta di un tipo di ridirezione particolare e poco usato. Istruisce la shell di leggere le righe successive fino a quando viene incontrata la parola indicata (senza spazi iniziali); successivamente invia quanto accumulato in questo modo allo standard input del comando indicato. In pratica, la parola indica la fine della fase di lettura. Non è possibile fare giungere l'input da una fonte diversa. Se la parola viene racchiusa tra virgolette, quelle usate per la protezione delle stringhe, si intende che il testo contenuto non deve essere espanso. Altrimenti, il testo viene espanso come di consueto. Se si usa il trattino ('**<<-**'), significa che le tabulazioni iniziali nel testo vengono eliminate.
[***n***]<&***m***	In questo modo si unisce il descrittore ***m*** al descrittore ***n*** di ingresso oppure, in mancanza dell'indicazione di ***n***, si unisce allo standard input.
[***n***]<&-	Chiude il descrittore ***n*** oppure, in mancanza dell'indicazione di ***n***, chiude lo standard input.
[***n***]>&***m***	In questo modo si unisce il descrittore ***n*** al descrittore ***m*** di uscita oppure, in mancanza dell'indicazione di ***n***, si unisce lo standard output.
[***n***]>&-	Chiude il descrittore ***n*** oppure, in mancanza dell'indicazione di ***n***, chiude lo standard output.
[***n***]<> ***file***	In questo modo si apre il file indicato in lettura e scrittura, collegando i due flussi al descrittore ***n***. Se questo descrittore non è specificato si intende l'utilizzo di entrambi standard input e standard output.

17.7.7 Strutture di controllo

Sintassi	Descrizione
for ***variabile*** [in ***valore***...] do ***lista_di_comandi*** done	La variabile indicata dopo '**for**' viene posta, di volta in volta, al valore di ciascun elemento della lista che segue la sigla '**in**', eseguendo ogni volta la lista di comandi che segue '**do**' (una volta per ogni valore disponibile). Se la sigla '**in**' (e i suoi argomenti) viene omessa, il comando '**for**' esegue la lista di comandi ('**do**') una volta per ogni parametro posizionale esistente. In pratica è come se venisse usato: '**in $@**'.

Sintassi	Descrizione

`case` ***parola*** `in`
`[`***modello*** `[ | ` ***modello*** `]...  )` ↩
↪***lista_di_comandi*** `;; ]`
...
`[*)` ***lista_di_comandi*** `;; ]`
`esac`
	La parola che segue '**`case`**' viene confrontata con ognuno dei modelli, usando le stesse regole dell'espansione di percorso (i nomi dei file). La barra verticale ('**`|`**') viene usata per separare i modelli quando questi rappresentano possibilità diverse di un'unica scelta.
	Quando viene trovata una corrispondenza, viene eseguita la lista di comandi corrispondente. Dopo il primo confronto riuscito, non ne vengono controllati altri dei successivi. L'ultimo modello può essere '**`*)`**', corrispondente a qualunque valore, che si può usare come alternativa finale in mancanza di altro.

`if` ***lista_condizione***
`then`
lista_di_comandi
`[elif` ***lista_condizione***
`then`
lista_di_comandi`]`
...
`[else`
lista_di_comandi`]`
`fi`
	Inizialmente viene eseguita la lista che segue '**`if`**' che costituisce la condizione. Se il valore restituito da questa lista è zero (cioè *Vero*), allora viene eseguita la lista seguente '**`then`**' e il comando termina. Altrimenti viene eseguita ogni '**`elif`**' in sequenza, fino a che ne viene trovata una la cui condizione si verifica. Se nessuna condizione si verifica, viene eseguita la lista che segue '**`else`**', sempre che esista.

`while` ***lista_condizione***
`do`
lista_di_comandi
`done`
	Il comando '**`while`**' esegue ripetitivamente la lista che segue '**`do`**' finché la lista che rappresenta la condizione continua a restituire il valore zero (*Vero*).

`until` ***lista_condizione***
`do`
lista_di_comandi
`done`
	Il comando '**`until`**' è analogo a '**`while`**', cambia solo l'interpretazione della lista che rappresenta la condizione nel senso che il risultato di questa viene invertito (negazione logica). In generale, per avere maggiori garanzie di compatibilità conviene utilizzare solo il comando '**`while`**', invertendo opportunamente la condizione.

`[function]` ***nome*** `() {`
lista_di_comandi
`}`
	La lista di comandi viene eseguita ogni volta che il nome della funzione è utilizzato come comando.
	Quando viene eseguita una funzione, i parametri posizionali contengono gli argomenti di questa funzione e anche **#** restituisce un valore corrispondente alla situazione.
	È possibile utilizzare il comando interno '**`return`**' per concludere anticipatamente l'esecuzione della funzione.

17.7.8 Comando «echo»

Comando	Descrizione
`echo [-n] [`***argomento***`...]`	Emette gli argomenti separati da uno spazio. Restituisce sempre il valore zero. '**`echo`**' riconosce alcune sequenze di escape che possono essere utili per comporre il testo da visualizzare. L'opzione '**`-n`**' consente di impedire che alla fine del testo visualizzato sia inserito il codice di interruzione di riga finale, in modo che il testo emesso successivamente prosegua di seguito. Si osservi che la shell Bash non riconosce le sequenze di escape se non si aggiunge espressamente l'opzione '**`-e`**', oppure si abilita l'opzione '**`xpg_echo`**' con il comando: '**`shopt -s xpg_echo`**'.

Tabella 17.48. Alcune sequenze di escape che possono essere riconosciute dal comando '**`echo`**'.

Codice	Descrizione
`\\`	Inserisce la barra obliqua inversa ('\').
`\a`	Inserisce il codice <*BEL*> (avvisatore acustico).
`\b`	Inserisce il codice <*BS*> (*backspace*).
`\c`	Alla fine di una stringa previene l'inserimento di una nuova riga.
`\f`	Inserisce il codice <*FF*> (*formfeed*).
`\n`	Inserisce il codice <*LF*> (*linefeed*).
`\r`	Inserisce il codice <*CR*> (*carriage return*).
`\t`	Inserisce una tabulazione normale (<*HT*>).
`\v`	Inserisce una tabulazione verticale (<*VT*>).
`\0`***n***	Inserisce il carattere corrispondente al codice ottale ***n***.

17.7.9 Comando «set»

Comando	Descrizione

`set [{-|+}`*x*`]...`
`set {-|+}o [`***modalità***`]`
`set` ***valore_param_1*** ↩
↪`[`***valore_param_2***`...]`
`set -- [`
valore_param_1 ↩
↪`[`***valore_param_2***`...]]`
	Questo comando, se usato senza argomenti, emette l'impostazione generale della shell, nel senso che vengono visualizzate tutte le variabili di ambiente e le funzioni.
	Se si indicano degli argomenti si intendono alterare alcune modalità (opzioni) legate al funzionamento della shell.

Alcune modalità, che potrebbero essere riconosciute dalla maggior parte delle shell POSIX:

Modalità	Descrizione
`{-|+}a` `{-|+}o allexport`	Le variabili che vengono modificate o create, sono marcate automaticamente per l'esportazione verso l'ambiente per i comandi avviati dalla shell.
`{-|+}b` `{-|+}o notify`	Fa in modo che venga riportato immediatamente lo stato di un gruppo di elaborazione (*job*) sullo sfondo che termina. Altrimenti, questa informazione viene emessa subito prima dell'invito primario successivo.

Modalità	Descrizione
{-|+}e {-|+}o errexit	Termina immediatamente se un comando qualunque conclude la sua esecuzione restituendo uno stato diverso da zero. La shell non esce se il comando che fallisce è parte di un ciclo '**until**' o '**while**', di un'istruzione '**if**', di una lista '**&&**' o '**||**', o se il valore restituito dal comando è stato invertito per mezzo di '**!**'.
{-|+}f {-|+}o noglob	Disabilita l'espansione di percorso (quello che riguarda i caratteri jolly, o metacaratteri, nei nomi di file e directory).
{-|+}m {-|+}o monitor	Abilita il controllo dei gruppi di elaborazione. Questa modalità è attiva in modo predefinito per le shell interattive.
{-|+}n {-|+}o noexec	Legge i comandi, ma non li esegue. Ciò può essere usato per controllare gli errori di sintassi di uno script di shell. Questo valore viene ignorato dalle shell interattive.
{-|+}u {-|+}o nounset	Fa in modo che venga considerato un errore l'utilizzo di variabili non impostate (predisposte) quando si effettua l'espansione di una variabile (o di un parametro). In tal caso, quindi, la shell emette un messaggio di errore e, se il funzionamento non è interattivo, termina restituendo un valore diverso da zero.
{-|+}v {-|+}o verbose	Emette le righe inserite nella shell appena queste vengono lette.
{-|+}x {-|+}o xtrace	Nel momento in cui si eseguono dei comandi, viene emesso il comando stesso attraverso lo standard output preceduto da quanto contenuto nella variabile ***PS4***.
{-|+}C {-|+}o noclobber	Disabilita la sovrascrittura dei file preesistenti a seguito di una ridirezione dell'output attraverso l'uso degli operatori '**>**', '**>&**' e '**<>**'. Questa impostazione può essere scavalcata (in modo da riscrivere i file) utilizzando l'operatore di ridirezione '**>|**' al posto di '**>**'. In generale sarebbe meglio evitare di intervenire in questo modo, dal momento che ciò non è conforme all'utilizzo normale.

17.7.10 Comando «test»

Comando	Descrizione
test *espressione_condizionale* [*espressione_condizionale*]	Risolve (valuta) l'espressione indicata (la seconda forma utilizza semplicemente un'espressione racchiusa tra parentesi quadre). Il valore restituito può essere *Vero* (corrispondente a zero) o *Falso* (corrispondente a uno) ed è pari al risultato della valutazione dell'espressione. Le espressioni possono essere unarie o binarie. Le espressioni unarie sono usate spesso per esaminare lo stato di un file. Vi sono operatori su stringa e anche operatori di comparazione numerica. Ogni operatore e operando deve essere un argomento separato. Se si usa la forma tra parentesi quadre, è indispensabile che queste siano spaziate dall'espressione da valutare.

Tabella 17.49. Espressioni per il comando '**test**'.

Espressione	Descrizione
-e *file*	*Vero* se il file esiste ed è di qualunque tipo.
-b *file*	*Vero* se il file esiste ed è un dispositivo a blocchi.
-c *file*	*Vero* se il file esiste ed è un dispositivo a caratteri.

Espressione	Descrizione
-d *file*	*Vero* se il file esiste ed è una directory.
-f *file*	*Vero* se il file esiste ed è un file normale.
-h *file*	*Vero* se il file esiste ed è un collegamento simbolico.
-p *file*	*Vero* se il file esiste ed è un file FIFO (*pipe* con nome).
-S *file*	*Vero* se il file esiste ed è un socket (socket di dominio Unix).
-t *descrittore*	*Vero* se lo standard output è aperto su un terminale.
-g *file*	*Vero* se il file esiste ed è impostato il suo bit SGID.
-u *file*	*Vero* se il file esiste ed è impostato il suo bit SUID.
-k *file*	*Vero* se il file ha il bit Sticky attivo.
-r *file*	*Vero* se il file esiste ed è leggibile.
-w *file*	*Vero* se il file esiste ed è scrivibile.
-x *file*	*Vero* se il file esiste e dispone del permesso di esecuzione, oppure se la directory esiste e c'è il permesso di attraversamento.
-O *file*	*Vero* se il file esiste e appartiene all'UID efficace dell'utente attuale.
-G *file*	*Vero* se il file esiste e appartiene al GID efficace dell'utente attuale.
-s *file*	*Vero* se il file esiste e ha una dimensione maggiore di zero.
file1 -nt *file2*	*Vero* se il primo file ha la data di modifica più recente.
file1 -ot *file2*	*Vero* se il primo file ha la data di modifica più vecchia.
file1 -et *file2*	*Vero* se i due nomi corrispondono allo stesso inode.
stringa	*Vero* se la stringa è diversa dalla stringa nulla.
-z *stringa*	*Vero* se la lunghezza della stringa è zero.
-n *stringa*	*Vero* se la lunghezza della stringa è diversa da zero.
stringa1 = *stringa2*	*Vero* se le stringhe sono uguali.
stringa1 != *stringa2*	*Vero* se le stringhe sono diverse.
stringa1 < *stringa2*	*Vero* se la prima stringa è lessicograficamente precedente.
stringa1 > *stringa2*	*Vero* se la prima stringa è lessicograficamente successiva.
op1 -eq *op2*	*Vero* se gli operandi hanno valori numerici uguali.
op1 -ne *op2*	*Vero* se gli operandi hanno valori numerici differenti.
op1 -lt *op2*	*Vero* se il primo operando ha un valore numerico inferiore al secondo.
op1 -le *op2*	*Vero* se il primo operando ha un valore numerico inferiore o uguale al secondo.
op1 -gt *op2*	*Vero* se il primo operando ha un valore numerico maggiore del secondo.
op1 -ge *op2*	*Vero* se il primo operando ha un valore numerico maggiore o uguale al secondo.
! *espressione*	Inverte il risultato logico dell'espressione.
espressione -a *espressione*	*Vero* se entrambe le espressioni danno un risultato *Vero*.
espressione -o *espressione*	*Vero* se almeno un'espressione dà un risultato *Vero*.
(*espressione*)	*Vero* se il risultato dell'espressione è *Vero*.

17.7.11 Comando «ulimit»

Comando	Descrizione
`ulimit` [*opzioni*] [*limite*]	Fornisce il controllo sulle risorse disponibili per la shell e per i processi avviati da questa, sui sistemi che permettono un tale controllo. Il valore del limite può essere un numero nell'unità specificata per la risorsa, o il valore '**`unlimited`**'. Se l'indicazione dell'entità del limite viene omessa, si ottiene l'informazione del valore corrente. Quando viene specificata più di una risorsa, il nome del limite e l'unità vengono emessi prima del valore. Il controllo pratico dei limiti impostati in questo modo dipende dal sistema operativo, che potrebbe anche ignorarne alcuni, per carenze realizzative nelle funzioni che dovrebbero attuare questi compiti. Se il limite viene espresso, questo diventa il nuovo valore per la risorsa specificata. Se non viene espressa alcuna opzione, si assume normalmente '**`-f`**'. I valori, a seconda dei casi, sono espressi in multipli di 1024 byte, o in «blocchi» da 512 byte, tranne per '**`-t`**' che è riferito a secondi e '**`-n`**' che rappresenta una quantità precisa. Il valore restituito è zero se non vengono commessi errori.

Tabella 17.50. Opzioni per l'uso del comando '**`ulimit`**'.

Opzione	Descrizione
`-H`	Viene impostato il limite fisico (*hard*) per la data risorsa. Un limite fisico non può essere aumentato una volta che è stato impostato. Se non viene specificata questa opzione, si intende l'opzione '**`-S`**' in modo predefinito.
`-S`	Viene impostato il limite logico (*soft*) per la data risorsa. Un limite logico può essere aumentato fino al valore del limite fisico. Questa opzione è predefinita se non viene specificata l'opzione '**`-H`**'.
`-a`	Sono riportati tutti i limiti correnti.
`-c`	La grandezza massima dei file '`core`' creati, espressa in blocchi che dovrebbero essere di 512 byte.
`-d`	La grandezza massima del segmento dati di un processo, in multipli di 1024 byte.
`-f`	La grandezza massima dei file creati dalla shell, espressa in blocchi che dovrebbero essere di 512 byte.
`-m`	La grandezza massima della memoria occupata, in multipli di 1024 byte.
`-s`	La grandezza massima della pila del processo (*stack*), in multipli di 1024 byte.
`-t`	Il massimo quantitativo di tempo di CPU in secondi.
`-n`	Il numero massimo di descrittori di file aperti (la maggior parte dei sistemi non permette che questo valore sia impostato, consentendo solo la sua lettura).

17.7.12 Altri comandi interni

Tabella 17.47. Descrizione sintetica di alcuni comandi interni di una shell POSIX.

Comando	Descrizione
`:` [*argomenti*]	Comando nullo. Ciò che inizia con il simbolo '`:`' non viene eseguito. Si ottiene solo l'espansione degli argomenti e l'esecuzione della ridirezione. Il valore restituito alla fine è sempre zero.

Comando	Descrizione
`.` *file_script*	Vengono letti ed eseguito il contenuto del file indicato, il quale è sufficiente sia leggibile. Se il nome del file non fa riferimento a un percorso, questo viene cercato all'interno dei vari percorsi elencati dalla variabile ***PATH*** (ci sono shell che cercano il file anche nella directory corrente). Il valore restituito dallo script è: quello dell'ultimo comando eseguito al suo interno; zero (*Vero*) se non vengono eseguiti comandi; *Falso* (un valore diverso da zero) se il file non è stato trovato.
`alias` [*nome*[*=valore*]]...	Il comando '**`alias`**' permette di definire un alias, oppure di leggere il contenuto di un alias particolare, o di elencare tutti gli alias esistenti. Se viene utilizzato senza argomenti, emette attraverso lo standard output la lista degli alias nella forma '***nome=valore***'. Se viene indicato solo il nome di un alias, ne viene emesso il nome e il contenuto. Se si utilizza la sintassi completa si crea un alias nuovo. La coppia '***nome=valore***' deve essere scritta senza lasciare spazi prima e dopo del segno di uguaglianza ('**`=`**'). Il comando '**`alias`**' restituisce il valore *Falso* quando è stato indicato un alias inesistente senza valore da assegnare, negli altri casi restituisce *Vero*.
`bg` [*specificazione_del_job*]	Mette sullo sfondo il gruppo di elaborazione (*job*) indicato, come se fosse stato avviato aggiungendo il simbolo e-commerciale ('**`&`**') alla fine. Se non viene specificato il gruppo di elaborazione, viene messo sullo sfondo quello corrente, dal punto di vista della shell. Se l'operazione riesce, il valore restituito è zero.
`break` [*n*]	Interrompe un ciclo '**`for`**', '**`while`**' o '**`until`**'. Se viene specificato il valore numerico ***n***, l'interruzione riguarda ***n*** livelli. Il valore ***n*** deve essere maggiore o uguale a uno. Se ***n*** è maggiore dei cicli annidati in funzione, vengono semplicemente interrotti tutti. Il valore restituito è zero purché ci sia un ciclo da interrompere.
`cd` [*directory*]	Cambia la directory corrente. Se non viene specificata la destinazione, si intende la directory contenuta nella variabile ***HOME*** (che di solito corrisponde alla directory personale dell'utente). Il funzionamento di questo comando può essere alterato dal contenuto della variabile ***CDPATH*** che può indicare dei percorsi di ricerca per la directory su cui ci si vuole spostare. Di norma, la variabile ***CDPATH*** è opportunamente vuota, in modo da fare riferimento semplicemente alla directory corrente.

Comando	Descrizione
`command` *comando* [*argomento*...]	Esegue un «comando» con degli argomenti eventuali. Il comando che si avvia può essere solo un comando interno oppure un programma, mentre sono escluse espressamente le funzioni.
`continue` [*n*]	Riprende, a partire dall'iterazione successiva, un ciclo '**for**', '**while**' o '**until**'. Se viene specificato il valore numerico ***n***, il salto riguarda ***n*** livelli. Il valore ***n*** deve essere maggiore o uguale a uno. Se ***n*** è maggiore dei cicli annidati in funzione, si fa riferimento al ciclo più esterno. Il valore restituito è zero, a meno che non ci sia alcun ciclo da riprendere.
`echo` [`-n`] [*argomento*...]	Emette gli argomenti separati da uno spazio. Restituisce sempre il valore zero. '**echo**' riconosce alcune sequenze di escape che possono essere utili per comporre il testo da visualizzare. Queste sono elencate nella tabella 17.48. L'opzione '**-n**' (non prevista dallo standard, ma generalmente disponibile) consente di impedire che alla fine del testo visualizzato sia inserito il codice di interruzione di riga finale, in modo che il testo emesso successivamente prosegua di seguito. Si osservi che la shell Bash non riconosce le sequenze di escape se non si aggiunge espressamente l'opzione '**-e**', specifica di Bash, oppure si abilita l'opzione '**xpg_echo**' con il comando: '**shopt -s xpg_echo**'.
`eval` [*argomento*...]	Esegue gli argomenti come parte di un comando unico. Restituisce il valore restituito a sua volta dal comando rappresentato dagli argomenti. Se non vengono indicati argomenti, o se questi sono vuoti, restituisce *Vero*.
`exec` [*comando* [*argomenti*]]	Se viene specificato un comando (precisamente deve essere un programma), questo viene eseguito rimpiazzando la shell, in modo da non generare un nuovo processo ulteriore. Se sono stati indicati degli argomenti, questi vengono passati regolarmente al comando. Il fatto di rimpiazzare la shell implica che, al termine dell'esecuzione del programma, non c'è più la shell. Se si utilizza questo comando da una finestra di terminale, questa potrebbe chiudersi semplicemente, oppure, se si tratta di una shell di *login* potrebbe essere riavviata la procedura di accesso.
`exit` [*n*]	Termina l'esecuzione della shell restituendo il valore ***n***. Se viene omessa l'indicazione esplicita del valore da restituire, viene utilizzato quello dell'ultimo comando eseguito.

Comando	Descrizione
`export` *nome*...	Le variabili elencate vengono segnate per l'esportazione, nel senso che vengono trasferite all'ambiente dei programmi eseguiti successivamente all'interno della shell stessa.
`fg` [*job*]	Pone il gruppo di elaborazione (*job*) indicato in primo piano, ovvero in *foreground*. Se non viene specificato il gruppo di elaborazione, si intende quello attuale, ovvero, l'ultimo a essere stato messo sullo sfondo (*background*).
`getopts` *stringa_di_opzioni* ↩ ↪*nome_di_variabile* [*argomenti*]	Il comando interno '**getopts**' serve per facilitare la realizzazione di script in cui si devono analizzare le opzioni della riga di comando. Ogni volta che viene chiamato, '**getopts**' analizza l'argomento successivo nella riga di comando, restituendo le informazioni relative attraverso delle variabili di ambiente. Per la precisione, '**getopts**' analizza gli argomenti finali della sua stessa riga di comando (quelli che sono stati indicati nello schema sintattico come un elemento facoltativo) e in mancanza di questi utilizza il contenuto del parametro **@**. L'utilizzo di '**getopts**' può risultare complesso, pertanto viene descritto meglio in una sezione apposita.
`hash` [`-r`] [*comando*...]	Per ciascun comando indicato, viene determinato e memorizzato il percorso assoluto. Se non viene dato alcun argomento, si ottiene l'elenco dei comandi memorizzati. Se si usa l'opzione '**-r**' si cancellano i percorsi memorizzati.
`jobs` [*job*...]	Elenca i gruppi di elaborazione attivi. Se viene indicato esplicitamente un gruppo di elaborazione, l'elenco risultante è ristretto alle sole informazioni sullo stesso.
`kill` [`-s` *segnale*] *pid*|*job*... `kill -l` [*numero_del_segnale*]	Invia il segnale indicato al processo corrispondente al numero del PID o del gruppo di elaborazione (*job*). Il segnale viene definito attraverso un nome, come per esempio '**KILL**' (senza il prefisso '**SIG**'), o un numero di segnale. Spesso viene tollerata l'assenza dell'indicazione del segnale, ma in tal caso si intende '**TERM**'. Un argomento '**-l**' elenca i nomi dei segnali.
`local` [*nome*[=*valore*]]	All'interno di una funzione, dichiara una variabile con un campo di azione limitato alla funzione stessa. Il comando non è previsto dallo standard POSIX ma è diffuso tra le shell che sono comunque aderenti allo standard.
`pwd` [`-P`]	Emette il percorso assoluto della directory corrente. Se viene usata l'opzione '**-P**', i percorsi che utilizzano collegamenti simbolici vengono tradotti in percorsi reali. Restituisce zero se non si verifica alcun errore mentre si legge il percorso della directory corrente.

Comando	Descrizione
`read [-p prompt] variabile...`	Viene letta una riga dallo standard input, assegnando la prima parola di questa riga alla prima variabile indicata come argomento, assegnando la seconda parola alla seconda variabile e così via. All'ultima variabile indicata nella riga di comando viene assegnata la parte restante della riga dello standard input che non sia stata attribuita diversamente. Per determinare la separazione in parole della riga dello standard input si utilizzano i caratteri contenuti nella variabile ***IFS***. L'opzione '**-p**' permette di definire un invito particolare. Questo viene visualizzato solo se l'input proviene da un terminale.
`readonly [variabile[=valore]...]` `readonly -p`	Le variabili indicate vengono marcate per la sola lettura e i valori di queste non possono essere cambiati dagli assegnamenti successivi. Se viene indicata l'opzione '**-p**', si ottiene una lista di tutti i nomi a sola lettura.
`return [n]`	Termina l'esecuzione di una funzione restituendo il valore ***n***. Se viene omessa l'indicazione di questo valore, la funzione che termina restituisce il valore restituito a sua volta dall'ultimo comando eseguito al suo interno. Se il comando '**return**' viene utilizzato al di fuori di una funzione, ma sempre all'interno di uno script, termina l'esecuzione dello script stesso.

`set [{-|+}x]...`
`set {-|+}o [modalità]`
`set valore_param_1 ↩`
`↪[valore_param_2...]`
`set -- [valore_param_1 ↩`
`↪[valore_param_2...]]`
	Questo comando, se usato senza argomenti, emette l'impostazione generale della shell, nel senso che vengono visualizzate tutte le variabili di ambiente e le funzioni. Se si indicano degli argomenti si intendono alterare alcune modalità (opzioni) legate al funzionamento della shell.
	Dal momento che si tratta di un comando molto complesso, il suo utilizzo viene descritto in una sezione apposita.

`shift [n]`	I parametri posizionali da ***n***+1 in avanti sono spostati a partire dal primo in poi (il parametro zero non viene coinvolto). Se ***n*** è 0, nessun parametro viene cambiato. Se ***n*** non è indicato, il suo valore predefinito è uno. Il valore di ***n*** deve essere un numero non negativo minore o uguale al parametro **#** (cioè al numero di parametri posizionali esistenti). Se ***n*** è più grande del parametro **#**, i parametri posizionali non vengono modificati. Restituisce *Falso* se ***n*** è più grande del parametro **#** o minore di zero; altrimenti restituisce *Vero*.

Comando	Descrizione
`test espressione_condizionale` `[ espressione_condizionale ]`	Risolve (valuta) l'espressione indicata (la seconda forma utilizza semplicemente un'espressione racchiusa tra parentesi quadre). Il valore restituito può essere *Vero* (corrispondente a zero) o *Falso* (corrispondente a uno) ed è pari al risultato della valutazione dell'espressione. Le espressioni possono essere unarie o binarie. Le espressioni unarie sono usate spesso per esaminare lo stato di un file. Vi sono operatori su stringa e anche operatori di comparazione numerica. Ogni operatore e operando deve essere un argomento separato. Se si usa la forma tra parentesi quadre, è indispensabile che queste siano spaziate dall'espressione da valutare. Nella tabella 17.49 vengono elencate le espressioni elementari che possono essere utilizzate in questo modo. Non si tratta necessariamente di un comando interno.
`times`	Emette i tempi di utilizzo accumulati.
`trap [argomento segnale...]`	Il comando espresso nell'argomento deve essere letto ed eseguito quando la shell riceve il segnale, o i segnali indicati. Se non viene fornito l'argomento, o viene indicato un trattino ('**-**') al suo posto, tutti i segnali specificati sono riportati al loro valore originale (i valori che avevano al momento dell'ingresso nella shell). Se l'argomento fornito corrisponde a una stringa nulla, questo segnale viene ignorato dalla shell e dai comandi che questo avvia. Se il segnale è '**EXIT**', pari a zero, il comando contenuto nell'argomento viene eseguito all'uscita della shell. Se viene utilizzato senza argomenti, '**trap**' emette la lista di comandi associati con ciascun numero di segnale. I segnali intercettati sono riportati al loro valore originale in un processo discendente quando questo viene creato.
`true`	Si tratta di un comando nullo che restituisce zero, pari a *Vero*.
`type nome...`	Determina le caratteristiche di uno o più comandi indicati come argomento. Restituisce *Vero* se uno qualsiasi degli argomenti viene trovato, *Falso* se non ne viene trovato alcuno.

Comando	Descrizione
`ulimit [opzioni] [limite]`	Fornisce il controllo sulle risorse disponibili per la shell e per i processi avviati da questa, sui sistemi che permettono un tale controllo. Il valore del limite può essere un numero nell'unità specificata per la risorsa, o il valore '**unlimited**'. Se l'indicazione dell'entità del limite viene omessa, si ottiene l'informazione del valore corrente. Quando viene specificata più di una risorsa, il nome del limite e l'unità vengono emessi prima del valore. Il controllo pratico dei limiti impostati in questo modo dipende dal sistema operativo, il quale potrebbe anche ignorarne alcuni, per carenze realizzative nelle funzioni che dovrebbero attuare questi compiti. Se il limite viene espresso, questo diventa il nuovo valore per la risorsa specificata. Se non viene indicata alcuna opzione, si assume normalmente '**-f**' (l'unica a essere prevista dallo standard POSIX). I valori, a seconda dei casi, sono espressi in multipli di 1024 byte, o in «blocchi» da 512 byte, tranne per '**-t**' che è riferito a secondi e '**-n**' che rappresenta una quantità precisa. Il valore restituito è zero se non vengono commessi errori. La tabella 17.50 riepiloga le opzioni e i limiti più comuni che possono essere impostati con '**ulimit**'.
`umask [modalità]`	La maschera dei permessi per la creazione dei file dell'utente viene modificata in modo da farla coincidere con la modalità indicata. Generalmente può essere inserita la modalità soltanto in forma di numero ottale. Se la modalità viene omessa si ottiene il valore corrente della maschera.
`unalias nome_di_alias...` `unalias -a`	Rimuove l'alias indicato dalla lista degli alias definiti. Se viene fornita l'opzione '**-a**', sono rimosse tutte le definizioni di alias.
`unset [-v] nome_variabile...` `unset -f nome_funzione...`	Vengono rimosse le variabili o le funzioni indicate. Se viene utilizzata l'opzione '**-f**', si fa riferimento espressamente a funzioni; se si indica l'opzione '**-v**' ci si riferisce espressamente a variabili. Se non si indicano opzioni e ci può essere ambiguità tra i nomi, vengono rimosse le variabili.
`wait [n]`	Attende la conclusione del processo specificato e restituisce il suo valore di uscita. Il numero ***n*** può essere un PID o un gruppo di elaborazione (*job*); se viene indicato un gruppo di elaborazione, si attende la conclusione di tutti i processi nel condotto relativo. Se ***n*** non viene indicato, si aspetta la conclusione di tutti i processi discendenti ancora attivi, restituendo il valore zero. Se ***n*** specifica un processo o un gruppo di elaborazione che non esiste, viene restituito il valore *Falso*, altrimenti il valore restituito è lo stesso dell'ultimo processo o gruppo di elaborazione di cui si è attesa la conclusione.

17.8 Riferimenti

- Mendel Cooper, *Advanced Bash-Scripting Guide*, *Appendix I. Localization*
  http://www.tldp.org/LDP/abs/html/localization.html

[1] **Bash** GNU GPL

[2] **Ash** UCB BSD

[3] Trattandosi di un «registro storico», l'abbreviazione del termine al solo aggettivo, viene fatta al maschile: «storico».

[4] *Vero* inteso come conclusione corretta del comando; *Falso* inteso come fallimento, parziale o totale del comando. Quando un comando termina senza un successo totale del tuo compito, il valore restituito serve a comunicare il tipo di problema che si è verificato; per questa ragione, dal momento che nei sistemi Unix può esistere un solo esito soddisfacente e tanti tipi di esito insoddisfacente, è corretto che lo zero sia stato associato al successo. Di conseguenza, per i fini della shell è bene ragionare in termini di zero=*Vero*, ma senza dimenticare che nell'algebra di Boole, vale il contrario.

[5] Per negazione logica si intende che zero viene commutato in uno, mentre un qualunque valore diverso da zero viene tramutato in zero.

[6] **Gettext** GNU GPL

[7] **Readline GNU** GNU GPL

[8] **Cle** GNU GPL