La shell `bash`

- Bash contiene alcune caratteristiche che derivano da altre shell molto usate ma anche alcune caratteristiche precipue.

- Alcune delle shell le cui caratteristiche sono state usate in Bash sono la Bourne Shell (`sh`), la Korn Shell (`ksh`), e la C-shell (`csh` insieme al suo successore, la TC-shell `tcsh`).

- Bash è anche un interprete di un linguaggio di comandi. Il nome è un acronimo di 'Bourne-Again SHell', un riconoscimento a Steve Bourne, autore di un diretto antenato della shell UNIX originaria `/bin/sh`.

- Bash è una shell `sh`-compatibile e rispetto ad essa offre miglioramenti funzionali sia per ciò che riguarda l'uso interattivo che per ciò che riguarda l'uso programmativo.

Cos'è una shell

- Una shell ha questo nome, che significa *conchiglia*, perché racchiude il kernel ed è di fatto la superficie con cui l'utente entra in contatto quando vuole interagire con il sistema.

- Fondamentalmente una shell è un macro processor che esegue comandi.

- Una shell UNIX è sia un interprete di comandi, che fornisce all'utente un'interfaccia verso un ricco insieme di utility, che un linguaggio di programmazione, che permette di combinare queste utility.

- Le shell possono essere utilizzate sia in maniera interattiva che non: a seconda dei casi, l'input è accettato dalla tastiera del terminale utente o da un file.

- Si possono infatti creare file che contengono comandi. Questi file, detti *script*, diventano comandi alla stessa stregua dei comandi di sistema che si trovano in directory come `/bin`, permettendo così agli utenti ed ai gruppi di personalizzare l'ambiente di lavoro.

Alcune caratteristiche

- I costrutti di "redirezione" permettono un controllo a grana fine dell'input e dell'output dei comandi, e la shell a sua volta permette un controllo sul loro ambiente d'esecuzione.

- Le shell UNIX forniscono anche un piccolo insieme di comandi *built-in* che implementano funzionalità altrimenti difficili da realizzare (come, ad esempio, `history`, `getopts`, `kill`, e `pwd`) se non addirittura impossibili (come, ad esempio, `cd`, `break`, `continue`, e `exec`).

- Mentre eseguire comandi è essenziale, molto del potere (e della complessità) della shell deriva dal linguaggio di programmazione incorporato. Come qualsiasi linguaggio di alto livello, esso fornisce variabili, funzioni e costrutti per il controllo del flusso.

- Nel tempo, le shell sono state equipaggiate con caratteristiche orientate sempre più verso l'uso interattivo che verso la programmazione. Alcune di tali caratteristiche sono il controllo dei job, l'editor da linea di comando, lo 'storico' (history) e gli alias.

Operazioni della shell

Quando legge ed esegue comandi, la shell fa sostanzialmente le seguenti operazioni:

1. Legge l'input da un file (*script*), o da una stringa passata come argomento (se invocata con l'opzione `-c`), o dalla tastiera del terminale utente.
2. Scompone l'input in parole ed operatori, rispettando le regole di quoting. I token riconosciuti sono separati da *metacaratteri*. In questo passo viene anche effettuata l'espansione degli alias.
3. Analizza i token in comandi semplici e composti.
4. Esegue vari tipi di espansioni, scomponendo i token espansi in liste di nomi di file, comandi ed argomenti.
5. Esegue ogni redirezione necessaria e rimuove gli operatori di redirezione ed i loro operandi dalla lista degli argomenti.
6. Esegue il comando.
7. Aspetta che l'esecuzione termini e quindi ne rileva lo stato.

Bash

- L'eseguibile della shell Bash è `bash`, che si trova normalmente nella directory `/bin/`.

  `bash` [*opzioni*] [*file-script*] [*argomenti*]

- Si distinguono fondamentalmente due tipi di modalità di funzionamento della shell Bash: interattiva e non interattiva.

- Quando l'eseguibile `bash` viene avviato con l'indicazione del nome di un file, `bash` 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/bash`). Gli eventuali argomenti che possono seguire il nome del file, vengono passati allo script in forma di parametri (numerati a partire da 1).

- La shell Bash è potenzialmente compatibile con diversi altri tipi di shell e questo crea una sostanziale differenza di comportamento nel momento dell'avvio, a seconda del tipo di compatibilità a cui ci si riferisce.

Shell interattiva

- La shell è interattiva quando interagisce con l'utente e di conseguenza mostra un invito (prompt) a inserire dei comandi.

- L'eseguibile `bash` può essere avviato eventualmente in modo esplicitamente interattivo utilizzando l'opzione `-i`.

- Quando la shell Bash funziona in modo interattivo, se già non esiste, crea la variabile di ambiente `PS1` che serve a contenere l'invito.

- 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.

Shell interattiva di login

- Una shell di login dovrebbe essere quella che si ha di fronte quando è stata completata la procedura di accesso (*login*, appunto). Alternativamente, può essere esplicitamente avviata utilizzando l'opzione `-login`.

- In una shell di login il parametro zero (`$0`) contiene un trattino (`-`) come primo carattere (di solito contiene esattamente il valore `-bash`).

- La shell Bash messa in funzione a seguito di un accesso, se non è stata specificata l'opzione `-noprofile`, tenta di leggere ed eseguire il contenuto:
  - del file `/etc/profile`;
  - del file `~/.bash_profile`, e, 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 il file esiste, legge ed esegue il contenuto di `~/.bash_logout`.

Shell interattiva di login

I file di configurazione letti da una shell di login vengono solitamente utilizzati per

- stabilire il path
- definire funzioni (ed alias)
- stabilire i parametri del terminale (con `stty`)
- stabilire il tipo del terminale (con `eval ‘tset ...‘`)
- stabilire i permessi di default sui file (con `umask`)

Shell interattiva normale

Se non è stata specificata una delle opzioni `-norc` o `-rcfile`, sempre che esista, viene letto ed eseguito il contenuto di `~/.bashrc`.

- Collegamenti tra file di configurazione
  - Spesso si invoca l'esecuzione del contenuto del file `~/.bashrc` anche nel caso di shell di login e questo attraverso un accorgimento molto semplice: all'interno del file `~/.bash_profile` si includono le righe seguenti.

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

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

Shell non interattiva

- Una shell non interattiva è di norma dedicata a eseguire uno script.

  Nel momento dell'avvio in questa modalità, la shell Bash 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 la variabile è vuota, oppure non esiste affatto.

Interpretazione degli argomenti successivi

- Se restano degli argomenti dopo le opzioni e non sono state usate le opzioni `-c` e `-s`, il primo di questi argomenti viene interpretato come il nome di un file di comandi di shell: uno script di shell.
  - `-c` *stringa*: vengono eseguiti i comandi contenuti nella stringa. Eventuali argomenti successivi vengono passati ai parametri posizionali a partire da `$0`.
  - `-s`: la shell legge i comandi dallo standard input.

- Se la shell viene avviata in questo modo, viene assegnato al parametro `$0` il nome dello script ed ai parametri posizionali (`$1`, `$2`, `$3`, ecc.) il resto degli argomenti.

- La shell legge ed esegue i comandi dello script e quindi termina l'esecuzione.

- Il valore restituito alla fine della sua esecuzione è quello dell'ultimo comando eseguito dallo script.

Bash: compatibilità

La tabella seguente riepiloga la sequenza di file di configurazione utilizzati a seconda del tipo di emulazione preferito.

	Tipo	File eseguito all'avvio
bash	login	/etc/profile + {~/.bash_profile| ~/.bash_login|~/.profile}
bash	interattiva	~/.bashrc
bash	non inter.	$BASH_ENV
sh	login	/etc/profile + ~/.profile
sh	interattiva	$ENV
sh	non inter.	

Nel caso di una shell bash di login, alla conclusione viene eseguito il file `/.bash_logout`, se esiste.

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 ('\'). Per esempio `\w` corrisponde alla directory corrente, `\u` al nome dell'utente e `\h` a quello del computer.

- 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.

Conclusione

- La conclusione del funzionamento della shell, quando si trova in modalità interattiva, si ottiene normalmente attraverso il comando interno `exit`, oppure eventualmente con il comando interno `logout` se si tratta di una shell di login.

- Se invece si tratta di una shell avviata per interpretare uno script, questa termina automaticamente alla conclusione dello script stesso.

Commenti

In una shell non interattiva, o anche in una shell interattiva in cui l'opzione `interactive_comments` è abilitata, una parola che comincia col simbolo `#` fa si che quella parola e tutta la parte restante della linea siano ignorati.

In una shell interattiva, l'opzione `interactive_comments` è abilitata di default. Una shell interattiva con l'opzione `interactive_comments` non abilitata non permette commenti.

Parametri, espansione e sostituzione

- Un compito molto importante delle shell UNIX è quello di rimpiazzare variabili e simboli speciali con quello che rappresentano.

- A fianco di questo problema si pone la necessità di proteggere ciò che si vuole evitare sia espanso dalla shell.

- Anche i simboli che proteggono contro l'espansione sono soggetti a loro volta a un procedimento di sostituzione: quando la shell ha terminato l'interpretazione di un'istruzione, questi devono essere rimossi in modo da non lasciarne traccia per un eventuale programma che dovesse ricevere questi dati in forma di argomenti.

Quoting

- Il termine *quoting* viene usato per rendere l'idea del verbo inglese 'quote' e fa riferimento all'azione di racchiudere parti di testo all'interno di delimitatori (virgolette o apici) per evitare confusione nei comandi, o per poter utilizzare un simbolo che altrimenti avrebbe un significato speciale.

- Il metodo del quoting viene quindi usato per togliere il significato speciale che può avere un carattere o una parola per la shell. Ci sono tre meccanismi di quoting:
  - il carattere di escape (rappresentato dalla barra obliqua inversa \),
  - gli apici semplici (') e
  - gli apici doppi (", o virgolette).

Escape e continuazione

- La barra obliqua inversa (‘\’) rappresenta il carattere di escape. Serve per preservare il significato letterale del carattere successivo, cioè per evitare che venga interpretato diversamente da quello che è veramente.
- 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 (bisogna fare bene attenzione a non lasciare spazi dopo questo simbolo, altrimenti non si comporta più come segno di continuazione).
- Alcuni esempi

```
/home/rossi$ ls -l \*
```

```
#!/bin/bash
echo "Questa e' una frase \
su una sola riga"
```

Apici singoli

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

- **Attenzione**: l'apice inclinato nel modo opposto (‘) viene usato con un altro significato che non rientra nel quoting.

- Vediamo alcuni esempi

```
/home/rossi$ echo 'Attenzione: $0 $1 ... restano "inalterati".'
Attenzione: $0 $1 ... restano "inalterati".
/home/rossi$
```

Apici doppi

- 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 solo quando è seguita da $, ‘, ” e \; 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 (cioè sequenze di caratteri tra una coppia di apici doppi) in cui si possono inserire:
  - variabili e parametri
  - comandi da sostituire

  le cui espansioni non vengono inibite.

- ```
  /home/rossi$ echo "Il parametro \$0 contiene: \"$0\""
  Il parametro $0 contiene: "-bash"
  /home/rossi$
  ```

  ```
  /home/rossi$ echo "oggi e' bello"
  oggi e' bello
  /home/rossi$
  ```
  ```

Parametri e variabili

- L'elemento comune tra i parametri e le variabili è il modo con cui essi devono essere identificati quando si vuole leggerne il contenuto: occorre il simbolo `$` davanti al nome (o al simbolo) dell'entità in questione.

- Il simbolo `$` non deve essere indicato quando si vuole assegnare un valore all'entità (sempre che ciò sia possibile).

- Un parametro (o una variabile) è definito, cioè esiste, quando contiene un valore, compresa la stringa vuota.

Parametri

- Un parametro è una variabile speciale che può essere solo letta e rappresenta alcuni elementi particolari dell'attività della shell.

- Ci sono due tipi di parametri: *posizionali* e *speciali*.

- Quando nelle documentazioni si fa riferimento a un parametro, si utilizza quasi sempre il suo nome (o il simbolo che rappresenta) preceduto dal simbolo `$` (dire, ad esempio, che il primo parametro posizionale si chiama `$1` non è esatto).

Parametri posizionali

- Un parametro posizionale è definito da una o più cifre numeriche a eccezione del parametro $0 che ha invece un significato speciale. I parametri posizionali rappresentano gli argomenti forniti al comando: $1 è il primo, $2 è il secondo, e così di seguito.

- Quando viene eseguita una funzione, questi parametri vengono rimpiazzati temporaneamente con gli argomenti forniti alla funzione.

- Quando si utilizza un parametro composto da più di una cifra numerica, occorre racchiudere questo numero tra parentesi graffe: ${10}, ${11}, ...

Parametri speciali

$0 Restituisce il nome della shell o dello script. Se la shell viene avviata con un file di comandi, $0 conterra' il nome del file. In una shell di login, $0 contiene solitamente la stringa '-bash'. Se la shell viene avviata con l'opzione '-c', $0 conterra' il primo argomento dopo la stringa dei comandi (se esiste).

$* Rappresenta l'insieme di tutti i parametri posizionali a partire dal primo. 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' (che di solito contiene la sequenza: <Sp><Tab><NL>).

$@ Rappresenta l'insieme di tutti i parametri posizionali a partire dal primo. Quando viene utilizzato all'interno di apici doppi, rappresenta una serie di parole, ognuna composta dal contenuto del rispettivo parametro posizionale. Di conseguenza, "$@" equivale a "$1" "$2"... .
Questo rappresenta un'eccezione rispetto agli altri parametri: l'espansione di questo parametro genera diverse parole.

$# Restituisce il numero di parametri posizionali.

$? Restituisce lo stato dell'ultima pipeline eseguita in primo piano (foreground).

$- Restituisce la serie di lettere corrispondenti alle modalita' configurabili attraverso il comando interno 'set'.

$$ Restituisce il PID della shell. Se viene utilizzato all'interno di una subshell, cioe' tra parentesi tonde, restituisce il PID della shell principale e non quello della subshell.

$! Restituisce il valore dell'errore.

$_ Restituisce l'ultimo argomento del comando precedente.

Variabili di shell

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

  $$nome\_di\_variabile=[valore]$$

  Se la variabile a sinistra dell'assegnamento non esiste allora viene creata ed inizializzata, altrimenti il suo valore precedente viene sovrascritto.

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

- Se il valore da assegnare non viene fornito, si intende la stringa vuota. La lettura del contenuto di una variabile si ottiene facendone precedere il nome dal simbolo `$`.

  ```
  /home/rossi$ echo $PIPPO
  ```

  Mostra sul video il valore della variabile `PIPPO`.

- Alcune variabili (ad esempio, `BASH`, `PPID`, `PWD`, `SHELL`, `UID`) vengono inizializzate direttamente dalla shell stessa.

Variabili di shell

- Tra le variabili restanti ci sono ad esempio
  - `PATH`: un elenco di directory separato da due punti verticali (':'). Il valore predefinito dipende dalla configurazione della shell. Un valore comune potrebbe essere:

    ```
    /usr/local/bin:/bin:/usr/bin:.:
    ```

  - `PS1`: invito primario. Un valore possibile è

    ```
    \u@\h:\w\$
    ```

    che, una volta interpretato, protrebbe restituire

    ```
    rossi@dsiII:/home/rossi$
    ```

    Provate ora ad assegnare a `PS1` il valore

    ```
    \u\w\$
    ```

  - `PS2`: invito secondario. Solitamente il suo valore è >.

- Le definizioni di variabili valide ad un dato momento possono essere mostrate tramite i comandi `env` e `printenv`.

Esportazione

- Quando si creano o si assegnano delle variabili, queste hanno una validità limitata all'ambito della shell stessa: i comandi interni sono al corrente di queste variazioni mentre i programmi che vengono avviati non ne risentono.

- Affinché anche i programmi ricevano le variazioni fatte sulle variabili, queste devono essere "esportate".

- L'esportazione delle variabili si ottiene con il comando interno `export`.

- Vediamo alcuni esempi
  - ```
    /home/rossi$ PIPPO="ciao"
    /home/rossi$ export PIPPO
    /home/rossi$
    ```

    Crea ed inizializza la variabile `PIPPO` e quindi la esporta.
  - ```
    /home/rossi$ export PIPPO="ciao"
    /home/rossi$
    ```

    Esegue la stessa operazione dell'esempio precedente, ma in un colpo solo.
  - ```
    /home/rossi$ export
    /home/rossi$
    ```

    Mostra tutte le variabili attualmente dichiarate "esportabili".
    ```

Subshell

Ci sono varie circostanze in cui la shell *corrente* crea una nuova shell *figlia*, detta appunto **subshell**, per eseguire determinati task.

- Quando viene eseguito un raggruppamento di comandi. Per esempio, `( ls ; pwd ; date )`. Se il raggruppamento non è eseguito in background, la shell corrente si sospende in attesa della terminazione della shell figlia.

- Quando viene eseguito uno script. Se lo script non è eseguito in background, la shell corrente si sospende in attesa della terminazione della shell figlia.

- Quando viene eseguito un job in background. In questo caso, la shell corrente continua la sua esecuzione concorrentemente alla shell figlia.

Una subshell eredita una copia delle variabili esportate dalla shell creante, dette anche *variabili d'ambiente*.

Le altre variabili, sono dette *variabili locali*.

Espansione

- Traduzione di parametri, variabili e altre entità analoghe, nel loro risultato. L'espansione viene eseguita sulla riga di comando, dopo averla scomposta in parole.

- Esistono sette tipi di espansione eseguite in quest'ordine
  1. parentesi graffe;
  2. tilde;
  3. parametri e variabili;
  4. comandi;
  5. aritmetica (da sinistra a destra);
  6. parole;
  7. percorso o pathname.

- Solo l'espansione attraverso parentesi graffe, 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.

- Dopo tutte le fasi di espansione e sostituzione, i simboli utilizzati per il quoting (che servivano appunto a prevenire certe espansioni) vengono eliminati.

Espansione delle parentesi graffe

- L'espansione delle parentesi graffe deriva dalla shell C. Si parte da una stringa del tipo

  `<prefisso>{<elenco>}<suffisso>`

  L'elenco indicato all'interno delle parentesi graffe è formato di elementi separati da virgola. Il risultato è una serie di parole composte tutte dal prefisso e dal suffisso indicati e all'interno uno degli elementi della lista.

  Per esempio, $a\{b, c, d\}e$ genera esattamente le tre parole '*abe ace ade*'.

- Vediamo alcuni esempi
  - `/home/rossi$ mkdir pippo/{vecchio,nuovo,dist,bachi}`
    Crea quattro directory: `vecchio/`, `nuovo/`, `dist/` e `bachi/` a partire da `/home/rossi/pippo/`.
    N.B.: `pippo` deve già esistere.
  - `/home/rossi# chown verdi /usr/{paperino/{qui,quo,qua},\`
    `topolino/{t??.*,minnie}}`
    cambia il proprietario di una serie di file:
    `/usr/paperino/qui`
    `/usr/paperino/quo`
    `/usr/paperino/qua`
    `/usr/topolino/t??.*`
    `/usr/topolino/minnie`

Espansione della tilde

- L'espansione della tilde deriva dalla shell C.
- Se una parola inizia con il simbolo tilde (`~`) si cerca di interpretare quello che segue, fino alla prima barra obliqua (`/`), come uno username, facendo in modo di sostituire questa prima parte con il nome della home directory 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`, cioè la directory personale dell'utente attuale.
- Vediamo alcuni esempi
  - `/home/rossi/report$ cd ~`
    `/home/rossi$`
  - `/home/rossi$ cd ~bianchi`
    `/home/bianchi$`
- Il carattere tilde viene usato anche per altri tipi di sostituzioni. Precisamente: la coppia `~+` viene sostituita con il contenuto di `PWD`, mentre, la coppia `~-` viene sostituita con il contenuto di `OLDPWD`.

Espansione di parametri e variabili

- Si ottiene facendo precedere il parametro o la variabile, eventualmente racchiuso tra parentesi graffe, dal simbolo `$`. Così facendo, il nome del parametro o della variabile sarà sostituito dal suo contenuto.

- Vediamo alcuni esempi
  - ```
    #!/bin/bash
    echo " 1 arg. = $1"
    echo " 2 arg. = $2"
    echo " 3 arg. = $3"
    ...
    echo "10 arg. = ${10}"
    echo "11 arg. = ${11}"
    ```
    Visualizza in sequenza l'elenco degli argomenti ricevuti, fino all'undicesimo.
  - ```
    #!/bin/bash
    UNO="Dani"
    echo "${UNO}ele"
    ```
    Compone la parola `Daniele` unendo il contenuto di una variabile con un suffisso costante.

- Con la shell Bash, i modi in cui è possibile ottenere la sostituzione di parametri e variabili sono numerosi e generalmente derivati dalla shell Korn.

Sostituzione dei comandi

- La sostituzione dei comandi consente di utilizzare quanto emesso da un comando attraverso lo standard output. Ci sono due forme possibili, la prima derivata dalla shell Korn e la seconda dalla shell Bourne.

  `$(<comando>)`
  `‘<comando>‘`

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

- La sostituzione dei comandi può essere annidata. Se si utilizza il metodo degli apici inversi, occorre fare precedere a quelli più interni il simbolo di escape.

- Se la sostituzione appare tra apici doppi, la suddivisione in parole e l’espansione di percorso non sono eseguite nel risultato.

Sostituzione dei comandi: esempi

- `/home/rossi$ ELENCO=$(ls)`

  Crea e assegna alla variabile `ELENCO` l'elenco dei file della directory corrente.

- `/home/rossi$ ELENCO=$(ls "a*")`

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

- `/home/rossi$ rm $( find / -name "*.tmp" )`

  Elimina da tutto il filesystem i file che hanno l'estensione `.tmp`.

  N.B.: segnala un errore `too few arguments` dovuto a `rm` se non ci sono file da eliminare (cioè se `find` non restituisce alcun file).

Espansione delle espressioni aritmetiche

- Le espressioni aritmetiche consentono la valutazione delle espressioni stesse e la loro sostituzione con il relativo risultato.

- Esistono due forme per rappresentare la sostituzione tramite espressione aritmetica:

  ```
  $[<espressione>]
  $((<espressione>))
  ```

- L'espressione viene trattata come se fosse racchiusa tra doppi apici, ma un doppio apice 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 il quoting.

- La sostituzione aritmetica può essere annidata.

- Se l'espressione aritmetica non è valida, si ottiene una segnalazione di errore senza alcuna sostituzione.

Espansione delle espressioni aritmetiche

- Vediamo alcuni esempi
  - `/home/rossi$ echo "$((123+23))"`
    Emette il numero 146 corrispondente alla somma di 123 e 23.
  - `/home/rossi$ VALORE=$[123+23]`
    Assegna alla variabile `VALORE` la somma di 123 e 23.
  - `/home/rossi$ echo "$[123*$VALORE]"`
    Emette il prodotto di 123 per il valore contenuto nella variabile `VALORE`.

Suddivisione in parole

- Una parola è una sequenza di caratteri che rappresenta qualcosa di diverso da un operatore o da un'entità da valutare. In altri termini, è una stringa che viene presa così com'è e rappresenta una cosa sola. Per esempio, un argomento fornito a un programma è una parola.

- La shell esegue la suddivisione in parole dei risultati delle espansioni di parametri e variabili, della sostituzione di comandi e di espressioni aritmetiche, che non siano avvenuti all'interno di un quoting di apici doppi.

- La shell considera ogni carattere contenuto all'interno della variabile `IFS` (*Internal Field Separator*) 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><newline>` (ovvero `<SP><HT><LF>`).

  La variabile `IFS` è quindi importantissima: non può mancare o essere vuota.

Suddivisione di parole: esempi

- ```
  /home/rossi$ echo b* d*
  bin diapositive
  /home/rossi$
  ```

  mostra i file il cui nome comincia per `b` o `d`, se c'è ne sono, altrimenti stampa `b*` e `d*`.

- ```
  /home/rossi$ Pippo="b* d*"
  /home/rossi$ echo $Pippo
  bin diapositive disegni
  /home/rossi$
  ```

  il risultato dell'espansione della variabile `Pippo` è suddiviso in parole.

- ```
  /home/rossi$ Pippo="b* d*"
  /home/rossi$ echo "$Pippo"
  b* d*
  /home/rossi$
  ```

  all'interno della coppia di doppi apici avviene la sostituzione (del nome `Pippo` con il suo valore) ma non la suddivisione in parole, quindi non può avvenire la successiva espansione di percorso.

- ```
  /home/rossi$ Pippo="b* d*"
  /home/rossi$ echo '$Pippo'
  $Pippo
  /home/rossi$
  ```

  all'interno della coppia di apici semplici, non avviene alcuna sostituzione della variabile `Pippo`.

Espansione di percorso (globbing)

- Dopo la suddivisione in parole, la shell Bash 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 è tale per cui la parola resta immutata. Tuttavia, quando si vuole indicare espressamente un nome che contiene effettivamente un asterisco o un punto interrogativo, sarebbe meglio essere precisi ed usare il 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 dal globbing.

Espansione di percorso

Caratteri jolly, o metacaratteri

***** Corrisponde a qualsiasi stringa, compresa la stringa vuota.

? Corrisponde a un qualsiasi carattere (uno solo).

[**...**] Corrisponde a uno qualsiasi dei caratteri racchiusi tra parentesi quadre.

[**a-z**] Corrisponde a uno qualsiasi dei caratteri compresi nell'intervallo da `a` a `z`.

[**!...**] Corrisponde a tutti i caratteri esclusi quelli indicati.

[**!a-z**] Corrisponde a tutti i caratteri esclusi quelli appartenenti all'intervallo indicato.

Per includere il trattino o la parentesi quadra chiusa in un raggruppamento tra parentesi quadre, occorre che questi simboli siano i primi o gli ultimi.

Eliminazione dei simboli di quoting

Al termine dei vari processi di espansione, tutti i simboli usati per il quoting (\, ’ e "`) che non siano stati protetti attraverso l’uso della barra obliqua inversa o di virgolette di qualche tipo, vengono rimossi.

Comandi

Con questo termine si intendono diversi tipi di entità che hanno in comune il modo con cui vengono utilizzate:

attraverso un nome seguito eventualmente da alcuni argomenti.

- *Comandi interni*

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

- *Funzioni*

  Dette anche funzioni di shell, sono funzioni definite 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.

Exit status

- Un comando che termina la sua esecuzione restituisce 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.
- Quando il risultato di un'esecuzione di un comando viene utilizzato in un'espressione booleana, si considera lo zero come equivalente a Vero, mentre un qualunque altro valore viene considerato equivalente a Falso.
- In casi particolari è la shell che assegna i valori di uscita di un comando:
  - quando un programma viene interrotto a causa di un segnale, il suo valore di uscita è pari a 128 più il valore di quel segnale;
  - quando un programma non viene trovato e quindi non può essere avviato, il suo valore di uscita è 127;
  - quando un presunto programma viene trovato, ma non risulta eseguibile, il suo valore di uscita è 126.
- Quando termina, la shell restituisce il valore di uscita dell'ultimo comando eseguito, se non riscontra un errore di sintassi, altrimenti genera un valore diverso da zero.

Pipeline

- Una pipeline è una sequenza di uno o più comandi separati dal simbolo *pipe*, ovvero la barra verticale (`|`).

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

- Lo standard output del primo comando è diretto allo 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 pipeline basta anche un solo comando.

- Normalmente, il valore restituito dalla pipeline corrisponde a quello dell'ultimo comando che viene eseguito all'interno di questa.

- Se all'inizio della pipeline viene posto un punto esclamativo (`!`), il valore restituito corrisponde alla negazione logica del risultato normale.

- La shell attende che tutti i comandi della pipeline siano terminati prima di restituire un valore.

- Ogni comando in una pipeline è eseguito come un processo separato (cioè, in una subshell).

Pipeline: esempi

Due usi molto comuni delle pipeline sono in connessione con i comandi `more` e `grep`.

- `/home/rossi$ ls -al | more`

  Mostra la lista particolareggiata dei file (anche quelli nascosti) della directory corrente, una pagina per volta.

- `/home/rossi$ who | more`

  Mostra, una pagina alla volta, la lista degli utenti attualmente collegati al sistema.

- `/home/rossi$ ps -aux | grep rossi`

  Mostra la lista dei processi relativi all'utente `rossi`.

Lista di comandi

- Una lista di comandi è una sequenza di una o più pipeline separate da `;`, `&`, `&&` o `||`, e terminata da `;`, `&` o dal codice di interruzione di riga.

- Parti della lista sono raggruppabili tramite parentesi (tonde o graffe) per controllarne la sequenza di esecuzione.

- Il valore di uscita della lista corrisponde a quello dell'ultimo comando della stessa lista che ha potuto essere eseguito.

Il separatore ‘;’

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

- – `/home/rossi$ latex lucidi.tex ; dvips lucidi.dvi`
  Avvia in sequenza una serie di comandi per la compilazione di un file latex e la generazione di un file postscript.

  – `/home/rossi$ echo "uno" ; echo "due"`
  `/home/rossi$ echo "uno" ; echo "due" ;`
  I due comandi sono equivalenti: nel secondo la lista viene conclusa con un punto e virgola, ma ciò non produce alcuna differenza di comportamento.

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

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

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

L'operatore di controllo `&&`

- L'operatore di controllo `&&` si comporta come l'operatore booleano AND: se il valore di uscita di ciò che sta alla sinistra è zero (Vero), viene eseguito anche quanto sta alla destra.

- Dal punto di vista pratico, viene eseguito il secondo comando solo se il primo ha terminato il suo compito con successo.

- `/home/rossi$ mkdir ./prova && echo "directory prova creata"`

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

  Provate ad eseguire il comando due volte.

L'operatore di controllo ||

- L'operatore di controllo `||` si comporta come l'operatore booleano OR: se il valore di uscita di ciò sta alla sinistra è zero (Vero), il comando alla destra non viene eseguito.

- Dal punto di vista pratico, viene eseguito il secondo comando solo se il primo non ha potuto essere eseguito, oppure se ha terminato il suo compito riportando un qualche tipo di insuccesso.

- `/home/rossi$ mkdir ./prova || mkdir ./prova1`

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

  Provate ad eseguire il comando due volte.

Avvio sullo sfondo con &

- I comandi seguiti dal simbolo `&` vengono messi in esecuzione sullo sfondo (*background*), ed eseguiti da una sub-shell ma senza la sospensione della shell corrente. Dal momento che non si attende la loro conclusione per passare all'esecuzione di quelli successivi, il valore restituito è sempre zero.

- — `/home/rossi$ yes > /dev/null & echo "yes sta funzionando"`
  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.

  — `/home/rossi$ echo "yes sta per essere avviato" ; yes > /dev/null &`
  In questo caso viene prima emesso il messaggio e quindi viene avviato `yes` sullo sfondo.

  — `/home/rossi$ ghostview lucidi.ps &`
  Avvia sullo sfondo il programma `ghostview` per visualizzare file postscript.

  — `/home/rossi$ xemacs note.txt &`

Delimitatori di lista `(...)`

- Le liste, o parti di esse, possono essere racchiuse utilizzando delle parentesi tonde.

- La lista racchiusa tra parentesi tonde viene eseguita in una subshell. Gli assegnamenti di variabili e l'esecuzione di comandi interni che influenzano l'ambiente della shell non lasciano effetti dopo che il comando composto è completato.

- Il valore restituito è quello dell'ultimo comando eseguito all'interno delle parentesi.

- `/home/rossi$ (mkdir ./prova || mkdir ./prova1) && echo "Creata"`

  Crea `prova` o `prova1`; se ci riesce, visualizza il messaggio.

- `/home/rossi$ echo $saluto`

```
/home/rossi$ saluto="ciao"
/home/rossi$ echo $saluto
ciao
/home/rossi$ (saluto="arrivederci")
/home/rossi$ echo $saluto
ciao
/home/rossi$
```

  Questo esempio mostra che i cambiamenti effettuati in una subshell non influenzano l'ambiente della shell corrente.

Delimitatori di lista { . . . }

- All'interno di uno script di shell, le liste di comandi possono essere raggruppate utilizzando parentesi graffe.

- Le liste contenute tra parentesi graffe vengono eseguite nell'ambiente di shell corrente. Si tratta quindi di un semplice raggruppamento di comandi su più righe.

- Il valore restituito è quello dell'ultimo comando eseguito all'interno delle parentesi.

- Gli esempi seguenti sono equivalenti.

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

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

Alias

- La gestione degli alias deriva dalla shell Korn.
- Attraverso i comandi interni `alias` e `unalias` è possibile definire ed eliminare degli alias, ovvero dei sostituti ai comandi.
- Il nome dell'alias non può contenere il simbolo `=`.
- Prima di eseguire un comando di qualunque tipo, la shell cerca la prima parola di questo comando (quella che lo identifica) all'interno dell'elenco degli alias; se la trova, la sostituisce con il suo alias. La sostituzione non avviene se il comando o la prima parola di questo è protetta attraverso il quoting, ovvero se è tra virgolette.
- 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.

Alias

- Se l'ultimo carattere del testo di rimpiazzo dell'alias è uno spazio o una tabulazione, allora anche la parola successiva viene controllata per una possibile sostituzione attraverso gli alias.

- Gli alias non vengono espansi quando la shell non funziona in modalità interattiva; di conseguenza, non sono disponibili durante l'esecuzione di uno script.

- A differenza della shell C, non c'è modo di utilizzare argomenti attraverso gli alias. Se necessario, bisogna utilizzare le funzioni. In generale, l'utilizzo di alias è superato dall'uso delle funzioni.

- 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`.

Alias: esempi

- `/home/rossi$ alias rm="rm -i"`

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

- Esempi simili sono:

  ```
  /home/rossi$ alias cp="cp -i"

  /home/rossi$ alias mv="mv -i"
  ```

- `/home/rossi$ dir`

  `dir` è un alias di `ls -l`, ma se si digita

  `/home/rossi$ "dir"`

  l'espansione dell'alias non avviene ed è in realtà eseguito il comando `/usr/bin/dir` che si comporta come `ls`.

- `/home/rossi$ unalias mv`

  Rimuove l'alias `mv` definito precedentemente.

- `/home/rossi$ alias`

  Mostra la lista degli alias attualmente validi.

Standard Input e Standard Output

- UNIX semplifica l'interazione tra programmi/comandi e utenti. Molti comandi prendono l'input da ciò che è chiamato *standard input* (`stdin`) e mandano l'ouput a ciò che è chiamato *standard output* (`stdout`).

  Solitamente la shell fa sì che `stdin` corrisponda alla tastiera e `stdout` allo schermo del terminale utente.

- Per esempio, il comando `ls` stampa il listato di una directory sullo standard output, che è normalmente "connesso" allo schermo. Un comando interattivo, come la shell `bash`, legge i comandi dallo standard input cioè dalla tastiera.

- Un programma può anche scrivere gli errori generati sullo *standard error* (`stderr`) che è quasi sempre connesso allo schermo, in modo che si possano leggere i messaggi d'errore.

Ridirezione

- "Ridirezione" fa riferimento al cambiamento del metodo normale con cui la shell tratta lo standard input (`stdin`), lo standard output (`stdout`) e lo standard error (`stderr`) (per default collegati al terminale utente).

- Prima che un comando sia eseguito, si può ridirigere il suo input e il suo output utilizzando una speciale notazione interpretata dalla shell.

- 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 viene segnalato un errore.

Ridirezione

- Per effettuare la ridirezione è necessario poter fare riferimento a `stdin`, `stdout` e `stderr`. Questi sono trattati come file speciali accessibili tramite i relativi descrittori.

- *Descrittore di file.* Si distinguono tre tipi di descrittori standard di file:
  - `0` = standard input;
  - `1` = standard output;
  - `2` = standard error.

- *N.B.* Una ridirezione riguarda un comando ed un file, mentre una pipeline riguarda due comandi.

- Altri descrittori di file, dal `3` al `9`, possono essere definiti dall'utente ed utilizzati per ridirigere input/output/error (questa caratteristica deriva dalla shell Bourne).

Ridirezione dell'input

`[n]< <parola>`

- 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 (descrittore di file `0`).

- `/home/rossi$ sort < ./elenco`

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

- `/home/rossi$ sort 0< ./elenco`

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

Ridirezione dell'output

`[n]> <parola>`

- 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 (descrittore di file `1`).

- Di solito, se il file da aprire in scrittura esiste già, viene sovrascritto, sempre che non sia attiva la modalità `noclobber` (comando interno `set`). Se invece è attiva la modalità `noclobber`, si ottiene l'aggiunta di dati al file eventualmente esistente.

- Per ottenere una sicura sovrascrittura di un file eventualmente preesistente, si può utilizzare l'operatore di ridirezione `>|`.

Ridirezione dell'output

- `/home/rossi$ ls > ./dir.txt`

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

- `/home/rossi$ ls 1> ./dir.txt`

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

- `/home/rossi$ ls 1>| ./dir.txt`

  Esegue la stessa operazione del primo esempio, ma si indica in maniera inequivocabile che il file `dir.txt` deve essere creato, anche se è attiva la modalità `noclobber`.

- `/home/rossi$ ls XtgEWSjhy * 2> ./errori.txt`

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

Ridirezione dell'output "in aggiunta"

`[n]>> <parola>`

- 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.

- `/home/rossi$ ls >> ./dir.txt`

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

- `/home/rossi$ ls 1>> ./dir.txt`

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

- `/home/rossi$ ls XtgEWSjhy * 2>> ./errori.txt`

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

Ridirezione simultanea

```
&> <parola>
>& <parola>
```

- La shell Bash consente la ridirezione simultanea di standard output e standard error in un unico file di destinazione rappresentato dalla parola che segue il simbolo di ridirezione (la prima delle due notazioni è preferibile).

- Non è possibile sfruttare questo meccanismo per aggiungere dati alla fine di un file esistente.

- `/home/rossi$ ls XtgEWSjhy * &> ./tutto.txt`

  Crea il file `tutto.txt` e gli inserisce il messaggio di errore causato dal file `XtgEWSjhy` inesistente e l'elenco dei file della directory corrente.

Ridirezione: esempi

- `/home/rossi$ sort < ./elenco > ./elenco_ordinato`

  Riordina il contenuto del file `elenco` nella directory corrente e ne emette il risultato nel file `elenco_ordinato`.

  `/home/rossi$ sort ./elenco > ./elenco_ordinato`

  è equivalente (dato che `sort` può prendere il suo input anche da un file).

- `/home/rossi$ more filename 2> /dev/null`

  Il file `/dev/null` è un "buco nero" per i bit. L'effetto è di scaricare qualsiasi messaggio di errore.

- `/home/rossi$ (dir AAA VARIE > outfile) &> errfile`

  Ridirige `stdout` e `stderr` in due file separati, `outfile` e `errfile` rispettivamente. Per far ciò, l'output è prima ridiretto all'interno di una subshell.

- `/home/rossi$ dir AAA VARIE 1> outfile 2> errfile`

  `/home/rossi$ dir AAA VARIE > outfile 2> errfile`

  Hanno lo stesso effetto del comando precedente.

Ridirezione: esempi

- `n>&m`

  ridirige il decrittore di file `n` nel decrittore di file `m`.

- Per esempio

  ```
  /home/rossi$ (cat file1 3>&2 2>&1 1>&3) > file2
  ```

  usa il descrittore di file `3` per scambiare i descrittori standard `1` e `2`.

  L'effetto del comando è che se `file1` può essere letto il suo contenuto è mostrato sullo schermo, mentre se il file non può essere letto un messaggio di errore viene registrato in `file2`.

Controllo dei processi

- Il controllo dei processi si riferisce alla possibilità di sospendere e ripristinare selettivamente l'esecuzione dei processi (programmi in esecuzione).

- UNIX mette a disposizione due utility ed un comando interno:
  - `ps` genera una lista di processi con i loro attributi, tra cui il nome, il PID, il terminale di controllo ed il proprietario;
  - `kill` *-signal* {*PID*}+ permette di inviare segnali ad un processo basandosi sul suo PID;
  - `wait` [*PID*] permette ad una shell di attendere fino a che uno dei suoi processi "figli" termini.

Controllo dei processi

`ps` genera una lista delle informazioni di stato dei processi.

`ps l` mostra informazioni più dettagliate sui processi.

`ps e` mostra anche l'ambiente dopo la linea di comando.

`ps f` mostra una famiglia di alberi indicanti le dipendenze dei vari comandi.

```
\home\rossi$ ps f
20742   1 S    0:01  \_ xterm -ls
20743  p4 S    0:00      \_ -bash
20756  p4 S    0:08          \_ xemacs lez8.tex
20774  p4 S    0:00          \_ xdvi.bin -name xdvi lez8.dvi
20824  p4 S    0:01          \_ xemacs serve.tex
20887  p4 R    0:00          \_ ps f
\home\rossi$
```

```
\home\rossi$ ps l
 FLAGS UID   PID  PPID PRI NI SIZE  RSS WCHAN   STA TTY TIME COMMAND
   100 500 20743 20742  17  0 1804 1128  117951 S   p4  0:00 -bash
100000 500 20756 20743   5  0 8268 6464  12dded S   p4  0:08 xemacs lez
100000 500 20774 20743   0  0 3348 2360  12dded S   p4  0:00 xdvi.bin ...
     0 500 20824 20743   0  0 7432 5600  12dded S   p4  0:01 xemacs ut
100000 500 20904 20743  19  0  880  368  0      R   p4  0:00 ps l
\home\rossi$
```

Provate anche `ps aux`.

Controllo dei job

- Il controllo dei job si riferisce alla possibilità di sospendere e ripristinare selettivamente l'esecuzione dei job.

- La shell associa un *job* a ogni pipeline e mantiene una tabella dei job in esecuzione, che può essere letta attraverso il comando interno `jobs`.

- Si distinguono due tipi di job. Un job è
  - in primo piano, o in *foreground*, quando è collegato alla tastiera e al video del terminale che si sta utilizzando.
  - sullo sfondo, o in *background*, quando lavora in modo indipendente e asincrono rispetto all'attività del terminale.

- Quando la shell avvia un processo in background, emette una riga simile alla seguente,

  ```
  [1] 12432
  ```

  che indica rispettivamente il numero di job (tra parentesi quadre) e il numero di processo (PID) dell'ultimo processo della pipeline associato al job.

Controllo dei job

- Un job in esecuzione in primo piano può essere sospeso immediatamente attraverso l'invio del carattere di sospensione, di solito Ctrl-z, in modo da avere di nuovo a disposizione l'invito della shell.

- In alternativa si può sospendere con ritardo un job in esecuzione in primo piano attraverso il carattere di sospensione con ritardo, di solito 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.

- Per far direttamente terminare un job in esecuzione in primo piano, si può usare il carattere di terminazione, di solito Ctrl-c.

Controllo dei job

- Per vedere la lista dei job di una shell si usa il comando interno `jobs`.

```
rossi@dsiII:/home/rossi/LEZIONI > jobs
[1]   Running                 xemacs lez8.tex &
[2]-  Running                 xemacs serve.tex &
[3]+  Running                 xdvi lez8.dvi &
rossi@dsiII:/home/rossi/LEZIONI >
```

  - Il numero tra parentesi è il *numero del job*: usato per riferire in modo specifico il job.
  - Il carattere `+` che segue le parentesi comunica che quello è il "job corrente": il job spostato per ultimo dal foreground al background.
  - Il carattere `-` che segue le parentesi comunica che quello è il penultimo job spostato dal foreground al background.
  - Lo stato può essere:
    * `Running`: in esecuzione.
    * `Stopped`: sospeso pronto a tornare in azione appena qualcuno lo richieda.
    * `Terminated`: ucciso da un segnale.
    * `Done`: terminato con codice d'uscita uguale a 0.
    * `Exit`: terminato con codice d'uscita diverso da 0.

Controllo dei job

- Con `jobs -l`, vengono anche mostrati i PID dei jobs.

```
rossi@dsiII:/home/rossi/LEZIONI > jobs -l
[1]  20647 Running                 xemacs lez8.tex &
[2]- 20650 Running                 xemacs serve.tex &
[3]+ 20662 Running                 xdvi lez8.dvi &
rossi@dsiII:/home/rossi/LEZIONI >
```

- Il comando interno `kill` consente di eliminare definitivamente un job o un processo (specificato dal suo PID).

  `kill` [`-l`] [*-signal*] {*PID* | *%job*}+

  L'opzione `-l` fornisce la lista dei segnali possibili. Questi possono essere specificati anche con codici numerici. Ad esempio, (9) SIGKILL e (15) SIGTERM.

  I processi possono proteggersi da tutti i segnali ad esclusione del segnale KILL (che non consente al processo di terminare normalmente come farebbe alla ricezione di un segnale TERM).

Controllo dei job

Per fare riferimento ad un job si utilizza il carattere `%`.

`%n` fa riferimento al job numero `n`.

`%<prefisso>` fa riferimento a un job il cui nome inizia con `prefisso`. Se esiste più di un job sospeso con lo stesso prefisso si ottiene una segnalazione di errore.

`%?<stringa>` fa riferimento a un job con una riga di comando contenente `stringa`. Se esiste più di un job del genere si ottiene una segnalazione di errore.

`%+` (e `%%`) fa riferimento al job corrente dal punto di vista della shell, quello marcato con `+`.

`%-` fa riferimento al job marcato con `-`.

Controllo dei job

- È possibile gestire i job sospesi attraverso i comandi interni `bg` e `fg`.
- `bg` consente di fare riprendere sullo sfondo l'esecuzione di un job sospeso. Se nessun job gli viene passato come argomento, farà riferimento al job marcato con `+`.

```
/home/rossi$ bg %1
/home/rossi$
```

```
/home/rossi$ %1 &
/home/rossi$
```

- `fg` consente di fare riprendere in primo piano l'esecuzione di un job sospeso. Se nessun job gli viene passato come argomento, farà riferimento al job marcato con `+`.

```
/home/rossi$ fg %1
```

```
/home/rossi$ %1
```

Controllo dei job: esempio

- Usiamo il comando `yes` al prompt.

  ```
  /home/rossi$ yes
  ```

  Questo comando ha lo strano effetto di creare una lunga colonna di `y` nel bordo sinistro dello schermo, più veloce di quello che riuscite a vedere.

- Sopendiamo il comando con Ctrl-z; prima del prompt, viene stampato un messaggio del tipo:

  ```
  [1]+  Stopped                 yes
  ```

  – Il numero tra parentesi è il *numero del job* usato per riferire in modo specifico il job

  – Il carattere `+` che segue le parentesi comunica che quello è il "job corrente": il job spostato per ultimo dal foreground al background.

  – La parola `Stopped` significa che il job è sospeso; Linux lo ha salvato in uno speciale stato di sospensione, pronto a tornare in azione appena qualcuno lo richieda.

  – `yes` è il nome del job che è stato fermato.

Controllo dei job: esempio

- Possiamo far ripartire (magari dopo aver eseguito altri comandi) il job sospeso con il comando `fg` che lo riporterà in primo piano. Una volta ritornato in foreground, ricominceranno le `y`.

- Cancelliamo ora questo job usando `kill`.

```
/home/rossi$ kill %1
[1]+  Terminated                    yes
```

- Adesso avviamo `yes` nuovamente, così:

```
/home/rossi$ yes > /dev/null
```

  cioè mandando l'output di `yes` nel file speciale `/dev/null`.

- Sospendiamo il job premendo Ctrl-z ed otterremo di nuovo il prompt.

```
[1]+  Stopped                       yes > /dev/null
/home/rossi$
```

Controllo dei job: esempio

- Tramite il comando `bg` lo riattiviamo in background, così da lasciarci il prompt per altri lavori interattivi.

```
/home/rossi$ bg
[1]+  yes > /dev/null  &
/home/rossi$
```

- Ci sono adesso due modi diversi per cancellarlo: con il comando `kill` o rimettendo il job in foreground e premendo il tasto di interruzione Ctrl-c.

- Proviamo il secondo metodo, solo per capire meglio le relazioni tra `fb` e `bg`:

```
/home/rossi$ fg
yes>/dev/null
```

  Per riottenere il prompt è necessario far terminare il job in foreground premendo Ctrl-c.

Controllo dei job: esempio

- Adesso avviamo un pò di job contemporaneamente.

```
/home/rossi$ yes > /dev/null   &
[1] 1024
/home/rossi$ yes | sort > /dev/null   &
[2] 1026
/home/rossi$ yes > /dev/null
```

  e premiamo Ctrl-z per sospendere l'ultimo job.

```
[3]+  Stopped                 yes > /dev/null
```

- Mettere un `&` dopo un comando comunica alla shell di avviarlo fin dall'inizio in background (è soltanto un modo per evitare di dover avviare il programma, premere Ctrl-z e digitare `bg`).

- Ogni job ha comunicato il suo numero di job (i primi due hanno mostrato anche il loro PID).

Controllo dei job: esempio

- Per eliminare il secondo job, potremmo semplicemente scrivere `kill %2`, oppure possiamo portarlo in foreground con

  ```
  /home/rossi$ fg %2
  yes | sort > /dev/null
  ```

  e poi premere Ctrl-c. Quindi `fg` prende anche i parametri che cominciano con %.

- In effetti, si poteva anche digitare:

  ```
  /home/rossi$ %2
  yes | sort > /dev/null
  ```

Controllo dei job: esempio

- Adesso digitiamo `jobs` per vedere quali job sono ancora in esecuzione:

```
/home/rossi$ jobs
[1]- Running                 yes > /dev/null &
[3]+ Stopped                 yes > /dev/null
```

  Il "+" significa che il job in questione è il primo della lista — un `fg` senza parametri porterà in primo piano il job numero 3.

  Il "-" significa che il job numero 1 è il secondo job pronto ad essere messo in foreground.

- Per mandare il job 1 in foreground si può usare

```
/home/rossi$ fg %1
yes > /dev/null
```

  Adesso premiamo Ctrl-z per sospenderlo.

```
[1]+  Stopped                 yes > /dev/null
```

Controllo dei job: esempio

- Essere passati al job numero 1 ed averlo sospeso ha cambiato la priorità dei job, come si può vedere con `jobs`:

```
/home/rossi$ jobs
[1]+  Stopped                 yes > /dev/null
[3]-  Stopped                 yes > /dev/null
```

- Il job 1 può essere ora rimesso in esecuzione con

```
/home/rossi$ bg
[1]+ yes > /dev/null  &
/home/rossi$ jobs
[1]-  Running                 yes > /dev/null
[3]+  Stopped                 yes > /dev/null
```

- Ora eliminiamo tutti questi job per poter usare meglio la macchina:

```
/home/rossi$ kill %1 %3
[1]+   Terminated     yes > /dev/null
[3]    Terminated     yes > /dev/null
/home/rossi$ jobs
/home/rossi$
```

History

- La shell mantiene informazioni sui comandi eseguiti al suo interno (caratteristica derivata dalle shell C e Korn).
- Alcune delle variabili usate sono le seguenti:
  - `HISTSIZE` denota il numero dei comandi per il quale si vuole che la shell mantenga informazioni (valore di default: 500);
  - `HISTFILE` denota il nome del file in cui la lista dei comandi è memorizzata (valore di default: `~/.bash_history`).

  N.B.: non sempre il file `~/.bash_history` viene utilizzato.
- Il comando `history` può essere utilizzato per mostrare o modificare la lista di history o per modificare il file di history. Se usato senza argomenti, mostra gli ultimi `$HISTSIZE` comandi eseguiti, altrimenti gli ultimi `n`.

```
/home/rossi$ history 3
  628  xdvi lez9.dvi
  629  latex lez9.tex
  630  history 3
/home/rossi$
```

History

- L'espansione della history introduce parole estratte dalla lista di history nello stream di input, facilitando
  - la ripetizione dei comandi,
  - l'inserimento di precedenti argomenti nella riga di input corrente e
  - la correzione di errori in comandi precedenti.

- L'espansione è fatta in due tempi:
  - prima si determina il comando all'interno della history che deve essere utilizzato
  - quindi si determina la porzione della riga di comando effettivamente da includere.

History

Alcuni modi di riferire comandi nella lista di history

- `!!` ripete l'ultimo comando
- `!n` ripete il comando numerato `n`
- `!string` ripete il comando più recente che comincia con `string`
- `!?string[?]` il comando più recente che contiene `string` (il secondo `?` si può omettere se `string` è seguito immediatamente da un NL).
- `^string1^string2^` ripete l'ultimo comando rimpiazzando `string1` con `string2`.
- I tasti freccia [↓] e [↑] possono essere usati per scorrere la lista dei comandi salvati e selezionare quello che si vuole rieseguire.

Esecuzione dei comandi

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

- Se il nome del comando contiene una o più barre (`/`), allora viene interpretato come un percorso del filesystem, 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 un alias o una funzione di shell con quel nome, questa viene eseguita;
  - altrimenti, se esiste un comando interno con quel nome, questo viene eseguito;
  - altrimenti, viene cercato all'interno del percorso di ricerca degli eseguibili contenuto nella variabile `PATH`.

Se la ricerca fallisce o se si tratta di una directory, si ottiene una segnalazione di errore e la restituzione di un valore di uscita diverso da zero.

Esecuzione dei comandi

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

- La shell tenta di avviare il programma configurando gli argomenti nel modo consueto: il primo, cioè 0, contiene il nome del programma, quelli successivi, contengono gli argomenti forniti eventualmente nella riga di comando.

- Se non si tratta di un programma, si intende che sia uno script di shell. Viene allora generata una 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 eseguirà quindi quel programma dando come argomenti il nome dello script e altri eventuali argomenti ricevuti nella riga di comando originale.

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>`

- 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 generati durante il funzionamento della shell stessa.

- È possibile eliminare delle variabili attraverso il comando interno `unset`.

Configurazione di ambiente

- 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 `latex` sullo sfondo con un diverso percorso di ricerca, senza però influenzare lo stato generale della configurazione di ambiente della shell.

```
/home/rossi$ PATH=/bin:/usr/bin/TeX latex slides.tex &
/home/rossi$
```

Caratteristiche di uno script

- Uno *script* è un file di testo contenente una serie di istruzioni che possono essere eseguite attraverso un interprete.

- Per eseguire uno script occorre avviare il programma interprete e informarlo di quale script questo deve eseguire.

  Per esempio, il comando

  ```
  /home/rossi$ bash pippo
  ```

  avvia `bash` come interprete dello script `pippo`.

- Per evitare questa trafila, nei sistemi UNIX esiste una convenzione attraverso la quale si automatizza l'esecuzione dei file script: si può dichiarare all'inizio del file script il programma che deve occuparsi di interpretarlo.

Caratteristiche di uno script

- Per questo si comincia il file da eseguire usando la sintassi seguente:

  `#! <nome-del-programma-interprete>`

  e gli si attribuisce il permesso di esecuzione.

- 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 Bash (`/bin/bash`), la prima riga sarà la seguente:

  `#!/bin/bash`

- 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.

Strutture di Controllo

- Per la formulazione di comandi complessi si possono usare le tipiche strutture di controllo 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 separazione e può essere rimpiazzato da uno o più codici di interruzione di riga.

Bash: comandi interni

:	Non fa nulla: semplicemente espande gli argomenti ed esegue le ridirezioni.
.	Legge ed esegue, nella shell corrente, i comandi del file indicato come argomento.
break	Termina un ciclo for, while, until o select.
cd	Cambia la directory corrente.
continue	Riprende la prossima iterazione di un ciclo for, while, until o select.
eval	Concatena ed esegue gli argomenti come un unico comando.
exec	Esegue un comando rimpiazzando la shell.
exit	Termina il funzionamento della shell.
export	Marca le variabili in modo che siano passate all'ambiente dei processi figli.
getopts	Analizza i parametri posizionali di uno script o funzione.
hash	Determina e memorizza i percorsi completi dei programmi indicati.
kill	Invia un segnale a un processo.
pwd	Emette il percorso della directory attuale.
readonly	Protegge le variabili contro la scrittura.
return	Termina una funzione restituendo un valore ben preciso.
shift	Fa scalare verso sinistra i parametri posizionali di un certo numero di posizioni.
test	Valuta un'espressione condizionale.
times	Emette i tempi di utilizzo accumulati.

Bash: comandi interni

trap	Specifica i comandi da eseguire quando la shell riceve segnali.
umask	Determina la maschera dei permessi per la creazione dei file.
unset	Elimina variabili e funzioni.
wait	Attende la conclusione dei processi figli.
logout	Termina l'esecuzione di una shell di login.
source	Esegue la stessa funzione del punto singolo (.).
fc	Recupera dei comandi dallo storico.
let	Esegue dei calcoli aritmetici con le variabili.
alias	Crea un alias di un comando.
unalias	Elimina un alias di un comando.
bind	Visualizza o modifica la configurazione della tastiera.
builtin	Esegue un comando interno in modo esplicito.
command	Esegue un comando interno o un programma.
declare	Dichiara delle variabili.
echo	Emette gli argomenti attraverso lo standard input.
enable	Abilita o disabilita dei comandi interni.
help	Emette informazioni sui comandi interni.
local	Crea delle variabili locali ad una funzione.
logout	Termina l'esecuzione di una shell di login.
read	Legge una riga dallo standard input e la assegna ad un insieme di variabili.
type	Determina il tipo di comando.
ulimit	Fornisce il controllo sulle risorse disponibili.
set	Configura una grande quantita' di elementi.