Unix shell

Shell

Programma che interpreta il linguaggio a linea di comando attraverso il quale l'utente utilizza le risorse del sistema. Permette la gestione di variabili e dispone di costrutti per il controllo del flusso delle operazioni.

Viene generalmente eseguito in modalità interattiva, all'atto del login, restando attivo per tutta la durata della sessione di lavoro ed effettuando le seguenti operazioni:

- Gestione del "main command loop";
- Analisi sintattica;
- Esecuzione di comandi ("built-in", file eseguibili) e programmi in linguaggio di shell (script);
- Gestione dello standard I/O e dello standard error ;
- Gestione dei processi da terminale.

Shell interattiva

La comunicazione tra l'utente e la shell interattiva avviene mediante i comandi o il nome di uno script.

La prima parola deve essere il nome di un comando built-in della shell

oppure

il nome di un file eseguibile, cioè il nome di un file presente nel file system contenente un programma in formato ELF (Executable and Linking Format)

oppure

il nome di uno script, cioè di un file ASCII presente nel file system, scritto in linguaggio di shell e dotato del permesso di esecuzione

Le shell di UNIX

I sistemi di tipo UNIX offrono la possibilità di lavorare con diverse shell, ed ogni utente può modificare la propria shell nella sessione d'uso.

/bin/sh	Bourne Shell	UNIX Sistem V dist.
/bin/csh	C Shell	Berkeley Sw. dist.
/bin/ksh	Korn Shell	Bell Labs
/bin/bash	Bourne Again Shell	Free Software Foundation
....		

Tutti i sistemi di tipo UNIX offrono il linguaggio e le funzionalità offerte dalla Bourne shell, o mediante implementazione diretta, oppure per il tramite di un'altra shell (p. es. bash).

La Bourne shell

Caratteri speciali (metacaratteri):

|	pipe
&&	and logico
||	or logico
;	separatore di comandi
;;	delimitatore per il costrutto case
&	operatore di background
( )	raggrupamento di comandi
{ }	delimitatori
<	redirezione dell'input
<<	input da un documento "here"
>	creazione di output
>>	accodamento di output
*****	corrisponde ad una stringa qualunque (anche vuota)
?	corrisponde ad un carattere singolo qualsiasi
[...]	corrisponde ad uno dei caratteri racchiusi
$*nome*	sostituisce il valore per la variabile di shell ***nome***
`...`	sostituisce un comando con l'output di questo
****	elimina il significato speciale del carattere che lo segue
'...'	elimina il significato di tutti i caratteri inclusi, eccetto ' stesso
"..."	elimina il significato speciale dei caratteri inclusi, eccetto: **$**, **`**, **** e **"**

Parole riservate:

if then else elif fi	costrutto if
case in esac	costrutto case
for while until do done	costrutti for, while e until

Profilo di allocazione

Quando ci si alloca su un terminale di un sistema UNIX, una shell interattiva viene attivata per gestire la seduta di lavoro relativa a quel terminale.

La prima azione della shell è quella di cercare se esiste un file contenente il profilo di allocazione dell'utente; se tale file esiste tutti i comandi in esso memorizzati vengono eseguiti dalla shell prima di restituire il prompt.

Questo file va dunque utilizzato dagli utenti per personalizzare il comportamento di una shell interattiva.

Per la Bourne shell il profilo di allocazione dell'utente deve trovarsi nel file .profile, sotto la directory home.

Variabili di shell predefinite

Esistono delle variabili di shell predefinite (variabili di ambiente), che permettono di caratterizzare il comportamento della shell.

Per convenzione, il nome di tali variabili è in caratteri tutti maiuscoli:

- **HOME** argomento di default per il comando **cd**, inizializzato da login con il path della home directory, letto dal file **/etc/passwd**;
- **PATH** Il path di ricerca degli eseguibili;
- **PS1** stringa del prompt, di default " $ " per l'utente normale e "#" per il super-user;
- ....

Variabili di shell predefinite

L'ambiente di shell può essere visualizzato grazie al comando **printenv**:

```
$ printenv
!C:=C:\cygwin\bin
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\Administrator\Dati applicazioni
COMMONPROGRAMFILES=C:\Programmi\File comuni
COMPUTERNAME=BAAL
COMSPEC=C:\WINNT\system32\cmd.exe
CVS_RSH=/bin/ssh
HOME=/home/Administrator
HOMEDRIVE=C:
HOMEPATH=\Documents and Settings\Administrator
....
```

Sintassi dei comandi

comando [argomento ...]

Gli argomenti possono essere:

- opzioni o flag (-)
- parametri

separati da almeno un separatore

Nota: Il separatore di default è il carattere spazio; per alcune shell può essere modificato grazie alla ridefinizione di una variabile d'ambiente opportuna (cfr. seg.).

Una volta interpretata la prima parola sulla linea di comando, la shell ricerca nel file system un file con il nome uguale a tale prima parola.

La ricerca avviene ordinatamente all'interno delle directory elencate nella variabile d'ambiente PATH

(Ri)definizione di variabili di shell

La shell offre all'utente sia la possibilità di ridefinire alcune variabili d'ambiente, sia di definire delle nuove variabili a proprio piacimento.

Esempio 1

```
$ frutto=mela
$ verbo=mangia
$ nome=Stefania
$ echo $nome $verbo una $frutto
Stefania mangia una mela
$
```

(Ri)definizione di variabili di shell

Esempio 2

```
$ echo $PATH
$ /usr/bin:/home/gio:.
$ ps
sh: ps: No such file or directory
$ PATH=$PATH:/bin
$ ps
PID TTY TIME CMD
2487 ttyp1 00:00:00 sh
2488 ttyp1 00:00:00 ps
$
```

(Ri)definizione di variabili di shell

Esempio 3

```
$ frutto=mela
$ frutto=${frutto}banana
$ echo $frutto
melabanana
$ tipo="mela banana"
$ echo $tipo
mela banana
$
```

Redirezione std I/O

- ***comando [argomento..] > file***

L'output di comando viene scritto in file. Se file non esiste viene creato, altrimenti viene sovrascritto

- ***comando [argomento..] >> file***

L'output di comando viene scritto in file. Se file non esiste viene creato, altrimenti l'output viene accodato

- ***comando [argomento..] < file***

Il contenuto di file viene passato in input a comando

Redirezione std error

Da ricordare:
input standard = 0
output standard = 1
errore standard = 2

- ***comando [argomento..] 2> file***

Lo standard error di comando viene scritto in file. Se file non esiste viene creato, altrimenti viene sovrascritto

- ***comando [argomento..] 2>> file***

Lo standard error di comando viene scritto in file. Se file non esiste viene creato, altrimenti l'output viene accodato

- ***comando [argomento..] > /dev/null 2>&1***

Lo std O viene cestinato. Lo std error viene rediretto sullo std O (e quindi anch'esso su /dev/null)

Sostituzione e pipeline di comandi

Sostituzione di un comando:

Gli apici ` ` sostituiscono la chiamata di un comando con il suo standard output . Da non confondere con gli apici ' '; su molte tastiere si ottengono digitando insieme i tasti AltGr e ' :

```
$ echo `who`
gio   pts/0    Oct 22 00:52
$ echo 'who -r'
who -r
$
```

Pipeline di due o più comandi:

Lo standard output di com1 funge da input a com2...

- ***com1 [arg ..] | com2 [arg ..]..|..***

Gestione dei processi da terminale

Quando si richiede alla shell di eseguire un comando, questo viene lanciato come un processo a se stante. I processi lanciati dalla shell si distinguono in due tipi:

- I processi in foreground sono quelli che sottraggono alla shell il controllo del terminale durante la loro esecuzione;
- Nei processi in background, il controllo del terminale rimane alla shell: il prompt appare immediatamente dopo che il processo è stato avviato.

Per default, l'esecuzione di un comando o di uno script genera un processo in foreground.
Per segnalare alla shell che si vuole lanciare il comando (o lo script) in background, è necessario farne seguire il nome dal carattere & :

- ***comando [argomento..] &***

Caratteri di escape

Tutti i metacaratteri tra gli apici ' ' non sono interpretati dalla shell, sono cioè trattati come caratteri standard.

```
$ echo 'Quanto vale $number ?'
Quanto vale $number ?
```

I metacaratteri tra gli apici doppi " " non sono interpretati dalla shell, eccetto $, \ e gli apici di sostituzione ` `

```
$ echo "Is today `date` ? "
Is today  Fri 21 12:00:00 CEST 2001 ?
```

Il carattere \ previene la shell dall'interpretare il carattere seguente (anche il carattere newline).

```
$ echo La data di oggi e\' `date` \?
La data di oggi e' Fri 21 12:00:00 CEST 2001 ?
```

Shell ed espressioni regolari

Molti comandi per l'elaborazione di testi di UNIX (ad esempio grep, ed, sed, ..) consentono la definizione di espressioni regolari, ossia di schemi per la ricerca di testo basati sull'impiego di metacaratteri:

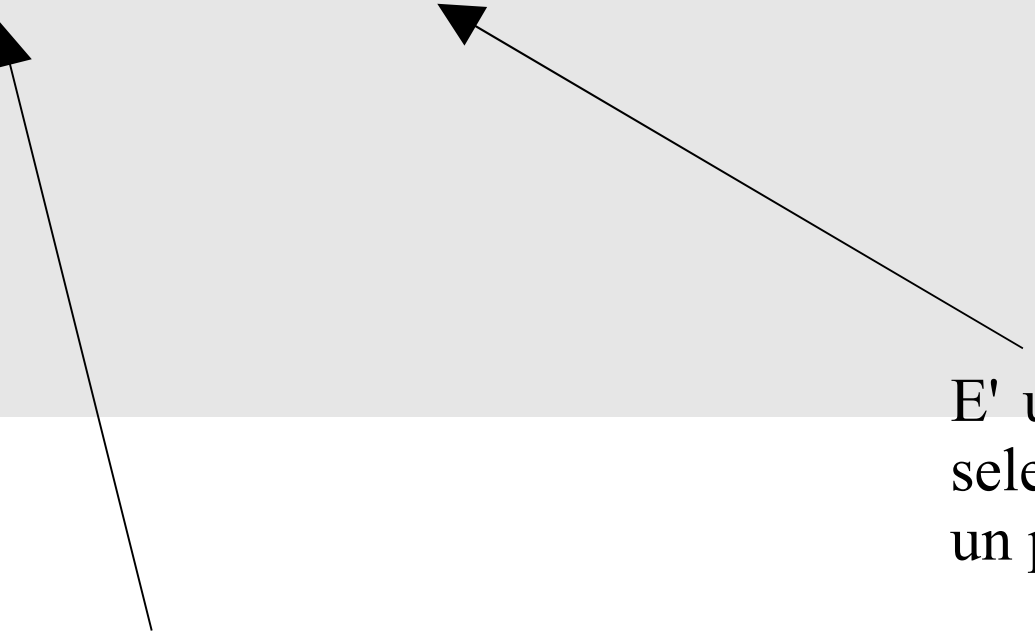
- Generalmente, i metacaratteri usati da tali comandi non coincidono con i metacaratteri impiegati dalla shell per identificare i nomi dei file
- Molti caratteri che hanno un significato speciale nelle espressioni regolari hanno pure un significato speciale per la shell

- Attenzione a non confondere i metacaratteri di shell con quelli che non lo sono
- Utilizzare gli apici ' ' o i doppi apici " " per racchiudere le espressioni regolari

Shell ed espressioni regolari

Esempio

```
$ for file in `ls`
do
if [ -f -a  -r $file]
then
cat $file | grep  '\.$' >> ultimerighe
fi
done
$
```



E' un metacarattere usato da grep per selezionare le linee che terminano con un punto.

E' usato dalla shell per ottenere il valore della variabile file

Script di shell

Script di shell

E' possibile scrivere file di testo nel linguaggio della shell (script). Quando uno script viene dato in input ad una shell, le sue istruzioni sono lette una alla volta ed eseguite dalla shell.

Se dotato dei permessi di esecuzione, uno script rappresenta un programma che può essere eseguito dalla shell interattiva.

E' possibile passare agli script degli argomenti all'atto della chiamata, come avviene per i comandi.

All'interno degli script di shell è possibile utilizzare:

- strutture di controllo
- comandi built-in
- comandi esterni

Programmazione della shell

Le operazioni di configurazione e gestione

di un sistema operativo sono

operazioni ripetitive

Automatizzare tali operazioni

Script di shell

Esempio

```
$ cat myscript
#! /bin/sh
# A differenza di questa, la riga precedente non è un commento
syl1=to
syl2=po
both_syl=$syl1$syl2
echo Le prime due sillabe sono: $both_syl
echo La parola e\': ${both_syl}nomastica
echo La parola e\': $both_sylnomastica
$
$ ./myscript
$ Le prime due sillabe sono: topo
$ La parola e': toponomastica
$ La parola e':
```

Da ricordare:
Potete scrivere i vostri script con un qualsiasi editor di testi, ma *non* con un word processor.
L'editor a video disponibile su *tutti* i sistemi UNIX e' **vi.** Per informazioni sul suo uso, consultate la documentazione in linea (comando **man vi**) o la *Guida minimalista*.

Argomenti degli script

Gli argomenti di uno script sono posizionali e vengono indicati mediante un numero (**0, 1, 2, 3..., 9**):

- Il numero **0** rappresenta il nome del file dello script
- I numeri **1,..., 9** rappresentano, nell'ordine, gli argomenti passati in input allo script
- **#** rappresenta il numero di argomenti passati in input allo script
- @ rappresenta l'insieme di tutti gli argomenti dati in input allo script

Se si passano meno di nove argomenti, a quelli mancanti corrispondono stringhe vuote

Esempio

```
$ cat myscript
#!/bin/sh
echo Il nome dello script e\': $0
echo Il primo argomento e\': $1
echo Il secondo argomento e\': $2
echo Il numero di argomenti e\': $#
echo Entrambi gli argomenti sono: $@
$ ./myscript start stop
Il nome dello script e': myscript
Il primo argomento e': start
Il secondo argomento e': stop
Il numero di argomenti e': 2
Entrambi gli argomenti sono: start stop
```

Da ricordare:
Se volete che lo script possa essere eseguito come un programma ,dovete aggiungere al file i permessi in esecuzione col comando **chmod**.

Da ricordare:
Se volete evitare di premettere il path relativo ./ per i vostri eseguibili, inserite tale path nella variabile di ambiente PATH, modificandone il valore grazie al file .profile.

Cicli for

```
for variabile [in lista di valori]
do lista di comandi
done
```

Da ricordare:
Il costrutto for permette nello stesso tempo di definire la variabile di shell di nome ***variabile*** e di assegnarle dei valori. Si noti la differenza sotto questo aspetto col costrutto case, illustrato in seguito.

variabile assume in successione i valori di tutti gli argomenti dati in input; oppure, se è presente il costrutto opzionale "**in** *lista di valori*", tutti i valori in *lista di valori*. Una lista di valori è un insieme di stringhe, separate da uno spazio e terminata da newline.

Per ogni valore viene eseguita la *lista di comandi* compresa tra **do** e **done**. Una lista di comandi è un insieme di comandi separati da ; o newline.

Esempi

```
for colore in giallo verde rosso
do
    echo colore $colore
done
```

Nota:
Mostra a video la scritta:
colore giallo
colore verde
colore rosso

```
for i
do
    >$i
done
```

Nota:
Permette di creare nella directory corrente file (vuoti), aventi nomi uguali agli argomenti passati in input allo script che lo contiene.

Costrutto if

```
if comando
then lista di comandi
else [lista di comandi 2]
fi
```

Da ricordare:
L'istruzione **if** va sempre fatta seguire da **then** e conclusa con **fi**

Si basa sul principio che un comando restituisce alla shell il valore zero se la sua esecuzione avviene con successo; un valore diverso da zero in caso contrario.

Se *comando* è eseguito con successo, allora viene eseguita la lista di comandi che segue l'istruzione **then**.

Se *comando* è eseguito con esito negativo, e se è presente il costrutto opzionale "**else** *lista di comandi 2*" , allora viene eseguita *lista di comandi 2*.

Costrutto if (cont.)

I costrutti **if** possono essere nidificati, ed è inoltre possibile usare l'istruzione **elif** per semplificarne la sintassi.

```
if comando_1
then lista_comandi_1
else
   if comando_2
   then lista_comandi_2
   fi
fi
```

```
if comando_1
then lista_comandi_1
elif comando_2
then lista_comandi_2
fi
```

Esempi

```
if grep "mystring" /tmp/myfile
then
  echo mystring trovata
else
  echo mystring non trovata
fi
```

Nota:
Intende determinare se il file ***/tmp/myfile*** contiene o meno la stringa ***mystring***, mostrando a video dei messaggi opportuni (corretto).

```
if find . -name "$1" 2>/dev/null
then
  echo Il file $1 esiste
fi
```

Nota:
Inserito in uno script, intende determinare se il file passato come input allo script esiste nel sottoalbero radicato nella directory corrente (errato).

"find exits with status 0 if all files are processed successfully, greater than 0 if errors occur".

Comando test

```
test condizione
```

Il comando **test** serve a valutare *condizione*.

Restituisce il valore 0 (successo) se *condizione* è vera, il valore 1(insuccesso) altrimenti.

Viene utilizzato spesso con i costrutti di controllo.

condizione può essere relativa a:

1. confronti tra valori numerici
2. tipi di file
3. confronti tra stringhe di caratteri

test: valori numerici

Se *condizione* è del tipo: **N <primitiva> M**

test controlla la relazione che intercorre tra i due numeri **N** ed **M**, rappresentabili anche mediante variabili di shell. Le primitive sono:

-eq	i valori di M e N sono uguali
-ne	i valori di N ed M sono diversi
-gt	N>M
-lt	N<M
-ge	N>=M
-le	N<=M

Esempi

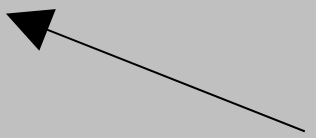
```
#!/bin/sh
utenti=`who | wc -l`
if  test $utenti -lt 8
then echo Sono allocati meno di 8 utenti
fi
```

Nota:
Il comando **test** può essere anche richiamato includendo la condizione tra parentesi quadre.

```
#!/bin/sh
utenti=`who | wc -l`
if [ $utenti -ge 8 ]
then echo Sono allocati almeno 8 utenti
fi
```

Esempio

```
$ cat myscript
#!/bin/sh
hour=`date | cut -c12-13`
echo Sono le $hour
if  test $hour -ge 17
then
   echo Buona sera
elif  test $hour -ge 12
then
   echo Buon pomeriggio
else
   echo Buon giorno
fi
$ ./myscript
Sono le 12
Buon pomeriggio
```



Nota:
Serve ad estrarre il campo “ora” dall' output di **date**. Dipende dalla implementazione di **date**

test: tipi di file

Se *condizione* è del tipo: ***<primitiva>* nomefile**

test permette di controllare l'esistenza e le caratteristiche del file nomefile. Le primitive più comuni sono:

-e	il file esiste
-s	il file esiste e non è vuoto
-f	è un file standard
-x	è un file eseguibile
-d	è una directory
-w	si hanno i permessi di scrittura
-r	si hanno i permessi di lettura

Esempi

```
if test -d $1

   then echo $1 rappresenta una directory

fi
```

```
echo File eseguibili nella directory `pwd`
for file in `ls`
do
    if  [ -x $file ]
    then
       echo $file
    fi
done
```

Da ricordare:
Lasciate sempre degli spazi

test: stringhe

Se *condizione* è del tipo: **S <primitiva> R**

test permette di confrontare le stringhe **S** ed **R**. Le primitive disponibili sono:

- **=** le due stringhe sono uguali
- **!=** le due stringhe sono diverse

Se *condizione* è del tipo: **<primitiva> S**

test controlla l'esistenza della stringa **S**. Le primitive disponibili sono:

- **-z** la stringa S ha lunghezza nulla
- **-n** la stringa S ha lunghezza non nulla

Esempio

```
$ numero=1
$ number=01
$ nombre=" 1"
$ if test $numero -eq $number ; then echo ok; fi
ok
$ if test $numero = $number ; then echo ok; fi
$ if test $numero -eq $nombre ; then echo ok; fi
ok
$ if test $numero= $nombre ; then echo ok; fi
$ echo $numero
1
$ echo $nombre
 1
$
```

test: **Operatori logici**

Per creare condizioni composte di **test** si possono utilizzare gli operatori logici **-a** , **-o** e **!**

L'operatore **-a** indica la funzione logica AND tra due espressioni: il risultato di tale funzione è vero solo se entrambe le espressioni sono vere

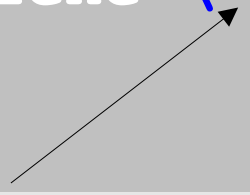
L'operatore **-o** indica la funzione logica OR tra tra due espressioni: il risultato di tale funzione è vero solo se almeno una delle due espressioni è vera

L'operatore **!** indica la negazione logica: il risultato di tale funzione è vero solo se l'espressione che segue è falsa

Esempi

```
#!/bin/sh
if [ -w $2 -a -r $1 ]
then cat $1 >> $2
else echo impossibile accodare
fi
```

```
#!/bin/sh
if [ $1 -lt  -1 -o $1 -gt  1 ]
then echo "Il numero inserito non appartiene \
                all'intervallo [-1,1]"
fi
```



Nota:
C'è ma non si vede...
(è il carattere **newline**)

Costrutto while

```
while comando
do
    lista di comandi
done
```

Se *comando* è eseguito con successo, allora viene eseguita la lista di comandi che segue l'istruzione **do** ed il ciclo viene ripetuto; altrimenti lista di comandi non viene eseguita e si esce dal ciclo.

Esempio

```
If [ $# -eq 0 ]
    then echo Uso: $0 file1...»&2
    exit
fi
while test $# -gt 0
do
   if test ! -s $1
   then echo $1 non esiste»&2
   else cat $1
   fi
   shift
done
```

Nota:
Serve a “scalare” sui parametri di input

Costrutto until

```
until comando
do
    lista di comandi
done
```

Se *comando* è eseguito senza successo, allora viene eseguita la lista di comandi che segue l'istruzione **do** ed il ciclo viene ripetuto; altrimenti lista di comandi non viene eseguita e si esce dal ciclo.

Esempi

```
until test $# -eq 0
do
    if [ ! -s $1 ]
    then echo $1 non esiste»&2
    else cat $1
    fi
    shift
done
```

```
until test -f file
do
    sleep 60
done
```

Nota:
Attende fintanto che **file** non viene creato, verificando l'esistenza del file ad ogni minuto.

Comandi true e false

I comandi **true** e **false** si limitano a restituire, rispettivamente il valore VERO (successo) e FALSO (insuccesso).

Sono utili, ad esempio, per realizzare cicli infiniti

```
while true
do
   sleep 300
   lpstat
done
```

```
until false
do
   sleep 300; lpstat
done
```

Costrutto case

```
case stringa in
stringa caso 1) lista di comandi 1 ;;
stringa caso 2) lista di comandi 2 ;;
...
esac
```

Se *stringa* è uguale a *stringa caso 1*, allora viene eseguita *lista di comandi 1* ed esce dal costrutto; altrimenti *lista di comandi 1* non viene eseguita, e passa ad elaborare in modo analogo il caso successivo.

poiché * rappresenta una stringa qualunque, essa può essere utilizzata per rappresentare "tutti gli altri casi".

Esempio

```
case $word in
hello) echo English ;;
howdy) echo American ;;
gday) echo Australian ;;
bonjour) echo French ;;
"guten tag") echo German ;;
*) echo Unknown Language: $word ;;
esac
```

Comando expr

expr considera i suoi argomenti come un'espressione aritmetica da valutare

Il risultato viene visualizzato sullo standard output

Gli operatori aritmetici che si possono usare sono:

- **+** addizione
- **-** sottrazione
- ***** moltiplicazione
- **/** divisione (parte intera del quoziente)
- **%** resto di una divisione

Esempio

```
$ expr 2 * 3
expr: syntax error
$ expr 2*3
2*3
$ expr "2 * 3"
2 * 3
$ expr 2 "*" 3
6
$ expr 2 \* 3
6
$ expr 2 '*' 3
6
$ expr 2\*3
2*3
$
```

Da ricordare:
Trattandosi anche di un metacarattere di shell, l'operatore di moltiplicazione * va sempre utilizzato con i caratteri di escape. Il carattere **spazio** permette invece a **expr** di individuare i propri operandi, e non va "escaped".

Esempi

Si possono utilizzare altri operatori con expr, che consentono, ad es., di confrontare valori numerici o stringhe. Nell'esempio seguente, l'operatore **:** è utilizzato per verificare se il primo argomento dello script è una singola lettera preceduta da un'opzione

```
if  expr $1 : -? >/dev/null
then echo L\'argomento costituisce una opzione
else echo "L'argomento non costituisce una opzione"
fi
```

Script interattivi

E' possibile creare degli script interattivi grazie all'uso del comando **read** . Questo comando attende l'inserimento di una linea di caratteri da parte dell'utente, e assegna la stringa corrispondente ad una variabile di shell.

```
$ cat pappagallo
#!/bin/sh
echo "Dimmi qualcosa: \c"
read cosa
echo "Ti faccio l'eco: $cosa"
$ ./pappagallo
Dimmi qualcosa: copione
Ti faccio l'eco: copione
$
```

Funzioni

Le funzioni di shell, come gli script, sono costituite da sequenze di istruzioni scritte nel linguaggio della shell utilizzata.

Tuttavia, mentre uno script viene eseguito richiamando il file dove è memorizzato, l'esecuzione di una funzione richiede una chiamata alla funzione tramite il nome assegnatole.

Una funzione può essere definita nello stesso file relativo allo script che la richiama, oppure in un file separato.

Esempio

```
#!/bin/sh
conferma() {  # inizio del corpo della funzione
   echo  "Sei sicuro? (S)i/(N)o? [S] "
   read answer
   case $answer in
   s|S|"")
      return 0
      ;;
   n|N)
      return 1
      ;;
   *)
      conferma # richiede in caso di risposta sbagliata
      ;;
   esac
} # fine del corpo della funzione

if conferma
then echo "L'utente ha detto SI"
else echo "L'utente ha detto NO"
fi
```

Esempio

```
#!/bin/sh

factorial()
{
  if [ "$1" -gt "1" ]; then
    i=`expr $1 - 1`
    j=`factorial $i`
    k=`expr $1 \* $j`
    echo $k
  else
    echo 1
  fi
}

while :
do
  echo "Enter a number:"
  read x
  factorial $x
done
```

Nota:
Attende l'inserimento dell'input da parte dell'utente

Verifica degli script

La Bourne shell accetta delle opzioni per la verifica delle procedure:

-v visualizza i comandi prima di eseguirlo

-x visualizza i comandi ed il valore delle variabili

-n stampa i comandi senza eseguirli

Riferimenti ulteriori

- S. Bourne - *An Introduction to the UNIX shell*
  http://www.sektorn.mooo.com/era/unix/shell.html

- D. Giacomini – *Appunti di Informatica Libera*
  http://a2.swlibero.org/a2.html