BASH

Breve introduzione alla programmazione della shell

La shell, oltre ad essere una comoda interfaccia per il sistema, è un potente linguaggio di scripting. Le sequenze di comandi possono infatti essere salvate in un file e successivamente richiamate come comandi.

Il presente documento è un estratto delle innumerevoli funzioni della shell BASH, a titolo di introduzione alla programmazione ed all'uso. Per una trattazione completa si veda il manuale GNU. ogni osservazione o correzione è benvenuta (matteo$_{chiocciola}$matteolucarelli$_{punto}$net)

Importante: lo shell-scripting, come molti altri linguaggi di script, è case-sensitive (cioè distingue tra maiuscole e minuscole) e NON IGNORA I BLANK, attenzione quindi a spazi, tab, return, ecc.

Indice

Formato del file di programma

Un programma per shell è un semplice file di testo (eventualmente con estensione .sh). E' utilizzabile come un qualsiasi altro comando se reso eseguibile

```
chmod 755 ./mioscript
./mioscript
```

Altrimenti può venire invocato passandolo come argomento alla shell (in tal caso non deve necessariamente essere eseguibile).

```
sh mioscript
```

La prima riga deve contenere il percorso dell'interprete, quindi ogni script inizierà con la dicitura:

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

Alcuni Unix richiedono anche uno spazio dopo il "!".

Il segno $ e le variabili

Il segno "$" fa sì che il valore seguente venga "espanso" (valutato). Per le variabili questo significa che al momento dell'assegnazione utilizzeremo il solo nome mentre al momento dell'utilizzo dovremo far precedere il nome dal segno di dollaro. La valutazione delle variabili avviene anche nelle stringhe tra virgolette. L'espansione ha molti altri effetti, che si apprenderanno gradualmente.

```
VALORE=3
echo $VALORE
```

Va notato che le variabili di shell non richiedono dichiarazione di tipo. Ad una stessa variabile è possibile assegnare successivamente valori di ogni tipo (numerico, stringa, ecc). E' quindi possibile realizzare array misti.

Nei nomi delle variabili sono ammessi tutti i caratteri alfanumerici e l'underscore. Il primo carattere deve essere alfabetico. Per convenzione si utilizzano normalmente caratteri maiuscoli.

Per evitare ambiguità la variabile può essere delimitata dalle parentesi graffe

```
VALORE=3
echo ${VALORE}alpha
```

NOTA: Non vanno lasciati spazi intorno al segno "=" di un assegnamento.

Altri caratteri speciali

- **\ (backslash)**

Carattere di escape, serve ad fare in modo che il carattere seguente non venga considerato come speciale.
es: **\$** è il cattere dollaro
(mentre il solo **$** è il carattere speciale già visto).
Posto come ultimo carattere di una linea serve a continuare il comando sulla linea successiva (deve essere l'ultimo, quindi attenzione ai blank!)

- **# (cancelletto)**

ad inizio riga identifica i commenti (la linea viene ignorata)

- **; (puntoevirgola)**

separa più comandi sulla stessa riga (sostituisce il return)
es: **for F in *; do echo $F; done**

- **| (pipe)**

l'output del primo comando va in input al secondo
es: **ls -l | grep .txt**

- **> (maggiore)**

lo standard output del comando viene scritto su un file (redirezione)
es: **ls -l > ls.txt**

- **2>**

lo standard error del comando viene scritto su un file

- **2>&1**

lo standard error del comando viene redirezionato sullo standard output (è possibile anche il contrario)

- **&>**

standard output e standard error vengono scritti su file
es: **kwrite &> /dev/null**

- **< (minore)**

l'input del comando viene letto da file

- **&& (and logico)**

il secondo comando viene eseguito solo se il primo ritorna valore zero (cioè, per convenzione, se ha successo). Ha funzione di AND logico.

- **|| (or logico)**

il secondo comando viene eseguito solo se il primo ritorna valore non zero. Ha funzione di OR logico.

- **& (ecommerciale)**

utilizzato in fondo ad un comando lo invoca in background, cioè in modo asincrono (l'esecuzione continua senza attendere la fine del comando)

- **‘ (apice inverso)**

Un comando messo tra apici inversi viene eseguito e gli viene sostituito il suo output (l'apice inverso si ottiene sulla tastiera italiana premendo AltGr+')
‘comando‘ equivale quindi a a **$(comando)**
La seconda forma, oltre ad essere più leggibile, permette l'annidamento, ed è quindi preferibile.

Raggruppamento dei comandi

I comandi possono essere raggruppati tramite le parentesi tonde () o tramite le parentesi graffe {}. I comandi raggruppati sono eseguiti come se fossero un singolo comando (quindi, ad esempio possono essere redirezionati in blocco). Il raggruppamento nelle parentesi tonde fa si che i comandi vengano eseguiti da una nuova shell figlia (subshell).
Nota: le parentesi graffe si ottengono dalla tastiera italiana premendo AltGr+7 e AltGr+0.

Parametri posizionali

I parametri posizionali sono gli argomenti passati allo script o alla funzione al momento della chiamata. Quindi se lo script "ordina.sh" viene chiamato:

```
ordina.sh /home/user/documenti
```

All'interno dello script il primo parametro posizionale ($0) varrà "ordina.sh" il secondo ($1) varrà "home/user/documenti".

- **$n** : n-esimo parametro posizionale ($0,$1,..)
- **$*** : lista dei parametri posizionali in un unica stringa
- **$@** : lista dei parametri posizionali come array di stringhe
- **$#** : numero dei parametri posizionali

Definizione di funzioni

All'interno di uno script possono essere definite delle funzioni tramite una sintassi analoga al linguaggio C:

```
[ function ] name ()
{
        command-list;
}
```

Gli argomenti della chiamata saranno i parametri posizionali all'interno della funzione. Non vanno quindi specificati nella definizione della funzione, e il loro numero può essere variabile. Le variabili all'interno della funzione hanno scope locale (cioè non sono definite all'esterno).

ES: tipica verifica di inzio script: se non sono presenti due parametri, oppure i due parametri non sono quelli previsti, stampa una breve nota di utilizzo ed esce.

```
print_usage() {
        echo "Questo script va chiamato con 2 parametri"
        echo "Che possono essere \"-c path\""
        echo "            oppure \"-l path\""
}

[ $# != 2 ] && print_usage && exit 1
[ $1 != "-l" ] && [ $1 != "-c" ] && print_usage && exit 2
```

Principali variabili predefinite

- **GROUPS** : elenco dei gruppi dell'utente corrente
- **HOME** : home dell'utente corrente
- **HOSTNAME,HOSTTYPE** : nome e tipo macchina
- **IFS** : lista dei caratteri separatori utilizzati
- **OLDPWD** : directory precedente
- **PATH** : percorsi di ricerca comandi
- **PPID** : pid del processo padre
- **PS1** : prompt
- **PWD** : directory corrente
- **RANDOM** : numero casuale (tra 0 e 32767)

- **`UID`** : id dell'utente corrente

Altre variabili automatiche

- **`$?`** : valore di ritorno dell'ultimo processo eseguito
  NOTA: tutti i comandi dovrebbero evere un valore di ritorno, ed in particolare, secondo lo standard POSIX, valore 0 significa "nessun errore"
- **`$$`** : PID (id del processo) della shell corrente
- **`$!`** : PID dell'ultimo processo lanciato in background

Principali comandi interni

I comandi che seguono sono un estratto dei comandi built-in della shell. Per ottenere maggiori informazioni utilizzare **`man bash`**.

- **`alias|unalias [name[=value] ...]`**
  assegna un alias, cioè la sostituzione di un comando. eseguito senza argomenti elenca gli alias definiti.
  es: **`alias ls="ls -l"`** assegna di default il flag -l al comando ls.
- **`break`**
  Interrompe (conclude) un ciclo.
- **`cd [dir|-]`**
  Cambia la directory corrente (quella di riferimento per i successivi percorsi relativi). Senza argomenti porta alla HOME, con "-" porta alla directory precedente.
- **`continue`**
  All'interno di un ciclo passa all'iterazione successiva.
- **`declare [-a] [name[=value]]`**
  serve a dichiarare una variabile. Nella bash le dichiarazioni sono implicite, quindi non necessarie per le semplici variabili. Il comando si utilizza in effetti solo per gli array (flag -a)
- **`echo [-neE] [arg ...]`**
  Manda in output la stringa e gli argomenti successivi (come un "print").
  -e abilita le seguenti sequenze di escape:
  - \a : bell
  - \b : backspace
  - \n : newline
  - \t : tab
  - \nnn : carattere con valore nnn (ottale)
  - \xHH esadecimale

  -E disabilita le sequenze di escape
  -n non aggiuge un newline alla fine della stringa stampata
- **`exec [-c] [-a name] [command [arguments]]`**
  Rimpiazza il processo della shell corrente con l'esecuzione del comando specificato. Il flag -c fa eseguire il comando in un ambiente vuoto.
- **`exit [n]`**
  Termina l'esecuzione, n è il valore di uscita.
- **`logout [n]`**
  Termina l'esecuzione di una shell di login, n è il valore di uscita.
- **`pwd`**
  Ritorna il valore della directory corrente (impostata con "cd").
- **`read [-a arr] [-d ch] [-n n] [-p str] [-t t] [-u fd] [name ...]`**
  legge da standard input, separa le parole e le assegna in sequenza alle variabili *name*. All'ultima variabile viene assegnata l'eventuale rimanenza. -a assegna gli input all'array arr. -d separa utilizzando il carattere ch [il default è lo spazio] -n legge max n caratteri -p mostra str come prompt -t attende max t sec -u legge da file descriptor
- **`umask [mode]`**
  Assegna il valore della maschera dei permessi di default. I file creati successivamente avranno i permessi determinati dal complemento a 7 del valore di umask (umask=022 -> permessi=755)
- **`source|. scriptfile`**
  Legge ed esegue lo script specificato. Può replicare il meccanismo degli header file (file di definizioni comuni, utili in sistemi complessi di script) o essere usato per funzioni di libreria.

Principali comandi esterni

I comandi che seguono sono eseguibili indipendenti dalla shell. Vengono generalmente considerati presenti in ogni sistema, e quindi comunemente utilizzati all'interno degli script. Ognuno dei comandi presenta molte opzioni,

per approfondire si vedano le corrispondenti pagine di manuale. I comandi illustrati risiedono generalmente in /bin e /usr/bin. Nella maggioranza dei sistemi esiste una pagina di manuale specifica per ogni comando visibile con **`man comando`**.

- **`awk`**
  Più che un comado è un vero e proprio linguaggio di programmazione. Serve a processare pattern, ad esempio con espressioni regolari.
  es:**`ls -l | awk '{print $6}'`** stampa solo il sesto campo (la data) dell'output di ls
  es:**`echo $A | awk '{print tolower($0)}'`** converte $A in caratteri minuscoli
- **`cat`**
  Lista uno o più file sullo standard output. Utilizzato per unire file.
  es: **`cat file1 file2 file3 > file123`** (unisce i tre file in uno)
- **`cdrecord/mkisofs/growisofs`**
  Comandi per masterizzare CD (cdrecord) creare ISO (mkisofs) e DVD (growisofs)
- **`chmod/chown/chgrp`**
  cambiano permessi e proprietario a file e directory
- **`cp`**
  copia file e directory
- **`date`**
  stampa o imposta data e ora, è possibile specificare il formato
- **`df [-h]`**
  Visualizza l'ammontare di spazio libero su disco (-h : formato "comprensibile")
- **`du [file o directory]`**
  Visualizza la quantità di spazio utlizzato su disco.
- **`grep`**
  filtra le righe che corrispondono ad un modello, è un sistema di parsing molto potente. Nel modo più semplice si usa per filtrare elementi da una lista (Es: ps aux | grep root)
- **`gzip/bzip2`**
  utility di compressione (rispettive decompressioni bunzip2/gunzip)
- **`host`**
  Risolve indirizzi IP (richiede un DNS configurato).
- **`lp/lpr`**
  Inviano contenuti alla stampante. Possono stampare in pipe o inviare file.
  es: **`lp doc.ps`** stampa il file postscript
  es: **`comando|lp`** invia alla stampante l'otput del comando
- **`ln [-s] file destinazione`**
  crea un link (-s simbolico) a file (o directory) nella specificata destinazione.
- **`ls [-al]`**
  visualizza i contenuti del filesystem (-a visualizza anche i file nascosti, -l visualizza le caratteristiche).
- **`mail/mailto`**
  Inviano e ricevono e-mail. Mailto può inviare anche contenuti multimediali.
- **`mkdir [-p] path`**
  Crea directory. con -p crea tutto il percorso se non esiste.
- **`mv`**
  sposta o rinomina file e directory
- **`ps`**
  elenca i processi (con tutte le relative informazioni)
- **`rm [-r]`**
  elimina file e (con -r) directory
- **`seq [-s STR][-f FRM] first [incr] last`**
  stampa una sequenza di numeri da *first* a *last* con incremento *incr* separati dalla stringa *STR* (default newline). Utile per realizzare cicli. Tramite -f FRM si può definire il formato, con sintassi simile alla printf del C.
- **`sleep n[smhd]`**
  Attende n secondi/minuti/ore/giorni. In alcuni sistemi è possibile assegnare intervalli decimali, quindi inferioiri al secondo. Altri hanno il comando usleep che accetta come argomento un valore in microsecondi.
- **`sort [file]`**
  Ordina alfabeticamente le linee di un file e le visualizza. Tramite pipe può leggere da standard input.
  es: **`echo -e "alpha\ngamma\nbeta\nkappa\ndelta" | sort`**
- **`sync`**
  Forza il flush del filesystem (cioè obbliga il sistema ad eseguire le operazioni eventualmente "rimandate").
- **`tar`**
  Archivia files e directory in un solo file (non compresso!) aggiungendo le opzioni "z" (gzip) o "j" (bzip2) lavora insieme alle utility di compressione, creando o leggendo archivi compressi
  es: **`tar xf archivio.tar`** scompatta l'archivio nella dir corrente
  es: **`tar cjf archivio.tar.bz2 ./dir`** crea l'archivio compresso con la directory dir
  attenzione: un archivio contiene i path relativi, quindi non è detto il contenuto resti nella directory corrente.

- **tr**

Traduce caratteri.
es: **tr -d '\015' < file.txt > file.2.txt** elimina i CR dal contenuto del file (converte un testo da formato DOS a formato UNIX)
es: **tr --squeeze-repeats ' ' < file** elimina tutte le ripetizioni di spazi, lasciando solo il primo

- **wget UrlSorgente [path_destinazione]**

Permette di effettuare download non interattivi sia da ftp che http.

Controllo dei jobs e dei processi

- **bg [job]**

riattiva un job in background

- **fg [job]**

riattiva un job in foreground

- **jobs [-rs]**

lista i job (-r: processi attivi,-s processi stoppati)

- **kill [job]**

invia un segnale ad un processo o ad un job (default=SIGTERM) utilizzare "kill -l" per avere una lista dei segnali possibili

- **wait [job o pid]**

attende il termine del processo

Test

Un test si può realizzare con una delle seguenti forme (gli spazi sono necessari!):

```
[ test ]
test test
```

Alcuni dei test possibili sono:

espressioni su file (es: **-a filename**):

- **-a|-e** : se esiste
- **-b** : se è un block device
- **-c** : se è un char device
- **-d** : se è una directory
- **-f** : se è un file normale
- **-h|-L** : se è un link simbolico
- **-p** : se è una pipe
- **-r** : se è leggibile
- **-s** : se non è vuoto (0 bytes)
- **-w** : se è scrivibile
- **-x** : se è eseguibile
- **-S** : se è una socket
- **-N** : se è stato modificato dopo l'ultima lettura

confronto tra file (es: **file1 -nt file2**):

- **-nt** : se file1 è più nuovo di file2 (modifica)
- **-ot** : se file1 è più vecchio di file2
- **-ef** : se file1 e file2 hanno lo stesso inode (cioè sono hard link ad uno stesso contenuto)

espressioni su stringhe (es: **-Z string**):

- **-z** : se è una stringa di lunghezza zero
- **-n** : se è una stringa di lunghezza non zero

confronto tra stringhe (es: **str1 == str2**):

- **==** : se sono uguali
- **!=** : se sono diverse

Note:

- la negazione di un test si ottiene con il punto esclamativo:
`[ ! -f nomefile ]` : vero se *file* NON è un file normale.

- Sono molto utilizzate forme sintetiche che sfruttano la priorità di valutazione:
`test && comando` : se *test* è vero esegui *comando*
`test || comando` : se *test* non è vero esegui *comando*
- Il test di una variabile ritorna vero se è definita e falso in caso contrario
`[ $VAR ] && echo "VAR=$VAR"`

Es: controlla il corretto numero di argomenti:

```
[ $# != 2 ] && echo "sintassi errata" && exit 1
```

Es: se non esiste l'eseguibile *mysqld* termina:

```
test -x /usr/sbin/mysqld || exit 0
```

Cicli e strutture condizionali

In tutte le strutture esposte i ritorni a capo sono necessari, nel caso in cui si voglia autilizzare una sola riga, vanno sostituiti con ; (puntoevirgola).

- Esecuzione condizionale:

```
if test
then
        commands
[elif test; then commands]
[else commands]
fi
```

- Selezione multipla:

```
case word in
        pattern [| pattern]...) command
                command
                ;;
        pattern [| pattern]...) command-list ;;
        .
        .

esac
```

- Assegnamento interattivo (apparirà un menu interattivo):

```
select name [in words ...]
do
        commands
done
```

- Iterazione classica (notare l'assenza di $ nel parametro MAX):

```
for (( i=0 ; i<MAX ; i++ ))
do
        commands
done
```

- Iterazione su lista (word è una lista di elementi):

```
for name [in words ...]
do
        commands
done
```

- Ciclo condizionale (eseguito mentre test è falso):

```
until test
do
        commands
done
```

- Ciclo condizionale (eseguito mentre test è vero):

```
while test
do
        commands
done
```

Es: se /bin/comando è eseguibile eseguilo, altrimenti stampa un errore

```
if [ -x /bin/comando ]; then
  /bin/comando
else
  echo "ERRORE: non posso eseguire /bin/comando"
fi
```

Es: stampa numeri crescenti all'infinito

```
COUNT=0
while [ 0 ]; do
  echo $COUNT
  (( ++COUNT ))
done
```

Es: attesa di 5 sec con (semplice) progress-bar

```
for loop in 3 2 1 0 ; do sleep 1; echo -n "$loop"; done
```

Es: memorizza il nome dei file presenti nella directory config nell'array TPCONF

```
TPCOUNT = 0
declare -a TPCONF
for CONFFILE in ./config/*
do
    TPCONF[$TPCOUNT]=$CONFFILE
    (( TPCOUNT++ ))
done
```

Es: countdown

```
COUNT=20
until [  $COUNT -lt 10 ]; do
     echo count: $COUNT
     let COUNT-=1
done
```

Es: stampa una stringa descrittiva dell'architettura di processore

```
case $( uname -m ) in
  i[3456]86 ) echo "architettura Intel o simili";;
  alpha     ) echo "architettura Alpha";;
  arm       ) echo "architettura Arm";;
  *         ) echo "Altro";;
esac
```

Es: Selezione di un file con menù

```
select SCELTA in ./*
do
  echo Hai scelto $SCELTA
  break
done
```

Pattern Matching

Una delle funzionalità più potenti della shell è nella ricerca di file e directory. E' possibile utilizzare nei path alcuni caratteri speciali:

- ***** : qualsiasi stringa, anche nulla
- **?** : qualsiasi singolo carattere, non nullo
- **?(lista)** : nessuno o uno dei path in lista
- ***(lista)** : nessuno, uno o più dei path in lista
- **+(lista)** : uno o più dei path in lista
- **@(lista)** : solo uno dei path in lista
- **!(lista)** : tutti eccetto i path in lista
- **[*xxx..*]** : uno qualsiasi tra i caratteri specificati

ES: conteggia e memorizza in un array i file presenti nella directory ./conf/

```
declare -a TPCONF
TPCOUNT=0
for CONFFILE in ./conf/*
do
                TPCONF[$TPCOUNT]=$CONFFILE
                (( TPCOUNT++ ))
done
```

Particolari espansioni

Alcune delle espansioni illustrate hanno una sintassi particolarmente criptica. Permettono, con un po' di pratica, sostituzioni quasi impossibili con altri linguaggi. Per una migliore comprensione si vedano gli esempi. NOTA: Le espansioni sono quasi sempre annidate, quindi i parametri vengono a loro volta espansi.

- **{*xxx..*}**
  espande nelle possibili combinazioni dei caratteri inclusi
- **~** (tilde)
  path della HOME (il carattere tilde si ottiene premendo AltGr+ì)
- **${parameter:-word}**
  se definito espande *parameter*, altrimenti *word*. Utile per definire eventuali valori di default.
- **${parameter:=word}**
  come il precedente ma a *parameter* viene assegnata l'espansione di *word*.
- **${parameter:?word}**
  Se definito vale *parameter*, altrimenti *word* viene stampato a std error (si utilizza per verificare i parametri iniziali)
- **${parameter:+word}**
  Se *parameter* è definito vale null. Altrimenti vale *word*.
- **${#parameter}**
  Espande nella lunghezza, in caratteri di *parameter*. Se *parameter* è un array espande nel numero di elementi.

```
Es: echo a{d,c,b}e  =output=> ade ace abe
```

Es: sostituisce gli spazi con underscore all'interno dei nomi dei file nella directory corrente (rinominandoli)

```
for FILE in *
do
        NUOVONOME=${FILE// /_}
        mv "$FILE" $NUOVONOME
done
```

Es: suddivide un path nei suoi componenti e li assegna ad un array: (l'espressione sembra complessa, ma è veramente potente)

```
PATH="/home/utente/doc/documento.pdf"
declare -a COMP=( "\""${PATH//\//"\" \""}"\"" )
```

```
# ciclo di stampa dell'array ottenuto
for i in '/usr/bin/seq 1 $((${#COMP[*]}-1))'
do
        echo ${COMP[$i]}
done
```

Stringhe

La Bash incorpora molte funzioni di parsing delle stringhe, anche se con una certa mancanza di omogeneità. Alcune delle seguenti funzioni sono utilizzate anche in altri contesti (la prima ad esempio, applicata ad un array ne restituisce il numero di elementi). Molte altre operazioni sono possibili tramite i comandi *tr* ed *expr*.

- **${#stringa}**
  Restituisce la lunghezza della stringa in caratteri
- **${stringa%regexp}**
  Elimina il più lungo match (regexp è un'espressione regolare) partendo dalla fine della stringa
- **${stringa##regexp}**
  Elimina il più lungo match (regexp è un'espressione regolare) partendo dall'inizio della stringa
- **${stringa%%regexp}**
  Elimina il più breve match (regexp è un'espressione regolare) partendo dalla fine della stringa
- **${stringa#regexp}**
  Elimina il più breve match (regexp è un'espressione regolare) partendo dall'inizio della stringa
- **${string:offset:length} ${parameter:offset}**
  Espande nei *lenght* caratteri della stringa a partire dal carattere *offset*, nella seconda forma estrae i caratteri fino alla fine della stringa
- **${string/sub1/sub2} ${string//sub1/sub2}**
  Sostituisce le occorrenze di *sub1* con *sub2* all'interno della stringa Nella prima forma viene sostituita solo un'occorrenza, nella seconda tutte
- **${string/#sub1/sub2} ${string/%sub1/sub2}**
  se *sub1* (che è un'espressione regolare) corrisponde alla parte diniziale della stringa viene sostituito con sub2. Nella seconda forma opera in modo analogo ma con la parte finale della stringa (utile, ad esempio per cambiare l'estensione)

```
foo=/tmp/my.dir/filename.tar.gz

len = ${#foo}          # ritorna la lunghezza della stringa
path = ${foo%/*}       # ritorna /tmp/my.dir
file = ${foo##*/}      # ritorna filename.tar.gz
base = ${file%%.*}     # ritorna filename
ext = ${file#*.}       # ritorna tar.gz

${file/%.tar.gz/.tgz} # ritorna filename.tgz

$(echo $file | tr '[A-Z]' '[a-z]') # ritorna FILENAME.TAR.GZ
```

Aritmetica

Un'espressione aritmetica si può valutare utilizzando la seguente forma:

```
(( expr ))
```

Esistono inoltre i due comandi **expr** e **bc**, per il cui approfondimento si rimanda ai rispettivi manuali.

Le operazioni possibili sono:

- **x++ ++x** : post e pre incremento
- **x-- --x** : post e pre decremento
- **+,-,*,/** : somma,sottrazione,prodotto,divisione
- ****,%** : elevamento a potenza,resto
- **! ~** : negazione logica e bitwise

- **<< >>** : shift bitwise
- **<= >= < >** : confronto
- **== !=** : uguaglianza e disuguaglianza
- **&,|,^** : bitwise AND,OR,XOR
- **&&,||** : AND e OR logici
- **test?b:c** : valutazione condizionale (se *test* allora vale b, altrimenti c)

Per parentesi annidate si utilizzano le parentesi semplici:

```
(( 5 * ( 2 - $NUM ) ))
```

il **$** è necessario, come sempre, per le assegnazioni e per l'output:

```
SUM=$(( $A + $B ))
echo $(( 5+3 ))
```

mentre non è necessario per incremento e decremento:

```
(( COUNTER++ ))
```

La doppia parentesi tonda va utilizzata anche per test di tipo numerico:

```
(( $A >= $B )) && echo "B non è più grande di A"
```

Anche se per i test di confronto funziona anche la parentesi quadra (che però, in realtà, effettua un confronto tra stringhe):

```
[ $A == $B ] && echo "A è uguale a B"
```

Array

La bash permette l'uso di array monodimensionali, che possono essere anche misti (cioè contenere valori di tipo differente).

```
# definizione
ARY=( uno due tre quattro )

# stampa il numero di elementi di un array
echo "${#ARY[*]}"

# stampa due elementi dopo il terzo
echo "${ARY[@]:3:2}"

#stampa il 2 elemento
echo "${ARY[1]}"

# stampa l'intero array
echo "${ARY[*]}"

# ciclo di stampa sugli elementi di un array
for (( I=0 ; I<${#ARY[*]} ; I++ ))
do
        echo "elemento[$I]: ${ARY[$I]}"
done
```

Trucchi e consigli

- Provare ogni sequenza alla riga di comando. Le sequenze complesse (ed anche i cicli) possono essere provate inline (cioè su una sola linea), separando le istruzioni con il carattere **;**.
- Utilizzare gli echo separando errori da semplici output:

```
echo "errore" >> /dev/stderr
echo "output" >> /dev/stdout
```

Quando non si vorranno più vedere gli avvisi (ma solo gli errori) sarà sufficiente avviare lo script con:

```
script.sh 1>> /dev/null
```

- Per fare il parsing dei flag di comando (si può utilizzare anche il builtin **getopt**):

```
while [ $# -gt 0 ]
do
        case "$1" in
                -h | --help )  print_usage && exit 0;;
                -v | --version )  echo "$VERSION" && exit 0;;
                -t | --test )  TESTFLAG=1;;
                -c | --config ) CONFIGFILE=$2;;
                * ) break ;;
                # altri flag e azioni possibili
        esac
        shift
done
```

- Per fugare eventuali dubbi sintattici, i pattern matching possono essere agevomente provati con il comando "ls" senza alcun rischio per il filesystem.
- Un problema tipico è il riempimento di liste con i nomi file, senza incappare in indesiderate divisioni dovute ad eventuali spazi presenti nei nomi. Il seguente codice utilizza IFS per memorizzare in un array i nomi dei file in ordine temporale:

```
OLDIFS=$IFS
IFS=$'\x0A'
FILEARRAY=( $(ls -dtr *) )
IFS=$OLDIFS
```

- Per debuggare uno script è possibile far stampare a video ogni linea prima che venga eseguita. E' sufficiente lanciare lo script con:
/bin/bash -x nomescript
L'opzione -x può essere anche posta direttamente all'interno dello script nella prima riga:
#!/bin/bash -x
