

Keep it simple, stupid!

- La filosofia alla base dei programmi Unix (solitamente non grafici) è semplice:
 - ogni programma è specializzato per uno scopo ben preciso;
 - funzionalità avanzate si ottengono dalla combinazione dei programmi semplici.
- tramite i meccanismi messi a disposizione dalle shell dei sistemi Unix (redirezione dell'input ed dell'output, pipeline) è possibile utilizzare i programmi quasi come dei mattoncini Lego, per costruire un sistema più complesso !

Alias

- Scorciatoie utili per evitare di scrivere comandi complessi.
 - `alias nomealias=comandi`
 - `alias ll="ls -l"`
 - `alias ls="ls -l"`
- Come le variabili, non vengono salvati alla fine della sessione di lavoro.
- Per cancellare un alias
 - `unalias nomealias`
 - `unalias -a`
 - cancella tutti gli alias

Variabili

- Create al volo senza dichiarazione
 - `nomevariabile=valore`
- Accesso al valore della var: premettendo un "\$":
 - `echo $nomevariabile`
- Alcune variabili sono fondamentali per il sistema:
 - PATH, PS1, HOME, MAIL
 - Si possono visualizzare con `set`
- Per permettere ad altri programmi (avviati dalla stessa shell) di accedere ad una variabile, va esportata.
 - `export nomevariabile`

Metacaratteri

- Alcuni caratteri hanno un significato particolare per la shell: sono detti *metacaratteri*.
 - ad esempio `< > | * ? [] ; &`
- È possibile comunque togliere significato ai metacaratteri, ed utilizzarli come caratteri normali.

```
Es:          $ ./prova saluti&baci
              saluti
#!/bin/sh    [2] 3887
echo $1      bash: baci: command not found
              [2] Done          ./prova saluti
```

Quoting

- Facendo precedere il carattere di escape `\` ad un carattere qualsiasi, questo perde il suo significato particolare.
 - `\$, \\`
- Racchiudendo una sequenza di caratteri tra apici singoli, viene preservato il significato letterale di ogni carattere.
- Racchiudendo una sequenza di caratteri tra apici doppi, viene preservato il significato letterale di ogni carattere AD ECCEZIONE di `$`, ``` e `\`

Apice inverso

- Carattere ``` (apice inverso): se si inserisce al suo interno un comando (semplice o composto), questo viene eseguito e “sostituito” dal relativo output.

```
$ ls -hs prova.txt
4,0K prova.txt
```

```
$ echo "informazioni sul file: ls -hs prova.txt "
informazioni sul file: ls -hs prova.txt
```

```
$ echo "informazioni sul file: `ls -hs prova.txt` "
informazioni sul file: 4,0K prova.txt
```

```
$ echo "informazioni sul file: $(ls -hs prova.txt) "
informazioni sul file: 4,0K prova.txt
```

Quoting

- In questo modo è comunque possibile (con i doppi apici) inserire delle variabili ed espanderne il valore.

```
$ var1="provola provoletta"
$ echo $var1
provola provoletta
$ echo var1 contiene $var1
var1 contiene provola provoletta
$ echo "var1 contiene $var1"
var1 contiene provola provoletta
$ echo 'var1 contiene $var1'
var1 contiene $var1
$ echo var1 contiene \$var1
var1 contiene $var1
```

Script

- Commenti: carattere `#`
- La prima riga contiene normalmente un commento speciale (bang path).
 - `#!/bin/sh`
 - `#!/bin/bash --login`
- È il nome del programma (e relativi parametri) che viene richiamato per eseguire lo script.
- È un meccanismo generale:
 - `#!/usr/bin/perl`
 - `#!/usr/bin/python`

Parametri

- La shell mette a disposizione degli script una serie di parametri predefiniti, identificati da "\$" seguito da un'altro carattere.
 - \$n (n intero): l'n-mo parametro passato da linea di comando allo script. Se n contiene almeno di 2 cifre, va racchiuso tra { } (ad es. \${12}).
 - \$0: il nome dello script.
 - \$* : se utilizzato all'interno di apici doppi, un'unica parola contenente tutti i parametri passati a linea di comando, separati dal contenuto del primo carattere della variabile d'ambiente IFS.
 - IFS: contiene normalmente <SP><HT><LF>

if

- Permette di specificare else if in cascata.

```
if list1
then
    comandi
[elif list2
then
    comandi]
...
[else
    comandi]
fi
```

Parametri

- @\$: comportamento simile a quello di \$*; a differenza di quest'ultimo, se utilizzato all'interno di apici doppi, rappresenta una lista di parole, una per ogni parametro.
- \$# : numero dei parametri passati.
- shift num
 - trasla i parametri da [\$#-num+1, \$#] in [\$1, \$#-num].
 - ad es. con 5 parametri, shift 2 sposta il 3° al posto del 1°, il 4° al posto del 2°, il 5° al posto del 3°, cancella gli altri due, e pone \$# pari a 3.

Condizioni

- Quali sono le condizioni utilizzabili all'interno dell'if?
- Dato un qualsiasi comando (o blocco di comandi), si considera il valore restituito.
 - Se questo è 0, la condizione è vera;
 - Altrimenti, la condizione è falsa.
- Quindi list1 può essere semplicemente una serie di comandi, legati da &&, ||, & (e separati da <newline> o da punto e virgola ";").

Condizioni

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

```
if ls $1 &>/dev/null
then
    echo "trovato"
fi
```

- `/dev/null`: dispositivo speciale, utile tra le altre cose per sopprimere l'output di un programma (come in questo caso).

Espressioni condizionali

- Sono delimitate da `[...]` (modo equivalente di scrivere il comando `test`).
- Vari tipi di condizioni:
 - "classiche" (`==`, `!=`, `>`, `<`)
 - altre sulle stringhe:
 - `-z` stringa (controlla se stringa vuota)
 - sui file
 - `-e` file (controlla se il file esiste)
 - `-d` file (il file esiste ed è una directory)
 - `file1 -nt file2` (file1 modificato dopo file2)
 -
- Vedere pagina di manuale del comando `test` !!! (parte del programma)

Espressioni aritmetiche

- Comando interno `let`: permette di eseguire espressioni aritmetico/logiche.
- Sono supportati i normali operatori aritmetici (`+`, `-`, `*`, `/`, `**`, `<<`, `>>`, `&`, `^`, `|`) e logici (`&&`, `||`).
- Possono essere utilizzate come espressioni condizionali, delimitando le espressioni con `((espressione))`

for

- Doppia sintassi

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

- Se `in lista` è omessa, vengono utilizzati i parametri posizionali.

```
for ((expr1; expr2; expr3))
do
    comandi
done
```

for

```
#!/bin/bash

listafile=`ls`
for nomefile in $listafile
do
    echo "file: $nomefile"
done
```

while/until

Esempio: attesa attiva

```
#!/bin/bash

while pidof $1 &>/dev/null
do
    sleep 5
done

echo "Comando completato"
play suono.wav
```

while/until

- Condizioni di uscita simmetriche

```
while listacomandirestituiscevero
do
    comandi
done
```

```
until listacomandirestituiscefalso
do
    comandi
done
```

Case

- La parola specificata viene confrontata con i vari modelli. Se viene trovata una corrispondenza, viene eseguita la lista di comandi corrispondente, e non vengono eseguiti altri controlli..
 - *): selezionato se nessun modello corrisponde.

```
case parola in
    [modello [ | modello]... )
        lista_di_comandi ;; ]
    ...

    [*) lista_di_comandi ;; ]
esac
```

Case

- modello è un pattern (può contenere caratteri jolly come * o ?).

```
#!/bin/bash
pattern=`file $1`
case $pattern in
  *:*html*)
    lynx $1 ;;
  *:*ASCII*)
    vi $1 ;;
  *:*mp3*)
    mpg123 $1 ;;
  *)
    echo "Non riconosco il tipo di $1"
    ;;
esac
```

Case: script di avvio (2/2)

```
restart)
  echo -n "Stopping sshd: sshd"
  kill `cat /var/run/sshd.pid`
  echo "."
  echo -n "Starting sshd: sshd"
  /usr/sbin/sshd
  echo "."
  ;;
*)
  echo "Usage:/etc/init.d/sshd start|stop|restart"
  exit 1
  ;;
esac
```

Case: script di avvio (1/2)

```
#!/bin/sh

test -f /usr/sbin/sshd || exit 0

case "$1" in
  start)
    echo -n "Starting sshd: sshd"
    /usr/sbin/sshd
    echo "."
    ;;
  stop)
    echo -n "Stopping sshd: sshd"
    kill `cat /var/run/sshd.pid`
    echo "."
    ;;

```

funzioni

- Più subroutine che funzioni.

```
[function] nomefunzione() {
    corpofunzione
}
```
- In questo caso, le variabili \$1, \$2 ... , (ma non \$0), sono relative non al programma principale, ma rappresentano i parametri della funzione.
- Condividono le variabili con il programma principale.
- Variabili locali: direttiva local.

funzioni: esempio

```
#!/bin/sh

function doppio
{
    local prova=1
    dp=$1$1
}

doppio "ciao"
echo "dp: $dp"
```

Select

- Genera in modo semplice menù interattivi.
- La scelta va avanti finché non viene richiamato break.

```
select nomescelta [in lista]
do
    comandi
done
```

- Se "in lista" non è specificato, vengono utilizzati i parametri posizionali (\$1, \$2, ...).

Input

- read (comando interno della bash) permette di assegnare valori alle variabili, inserendoli da standard input.
 - read [opzioni] [nomevar1 nomevar2 ...]
 - le parole inserite in input vengono assegnate in ordine a nomevar1, nomevar2,...
 - Se non sono state specificate variabili, vengono utilizzati i parametri posizionali.
 - opzione -s: silent mode (come per password)
 - opzione -n num: si ferma dopo l'inserimento di num caratteri

Select

```
#!/bin/bash

echo -n "In quale razza è particolarmente "
echo "sviluppato il senso dell'onore?"
lista="klingon romulani vulcaniani cardassiani"

select scelta in $lista
do
    if [ $scelta = klingon ]
    then
        echo "esatto!"
        break
    else
        echo "sbagliato!"
    fi
done
```

Comandi utili

- **sort**
 - ordinamento sul testo in input.
- **find**
 - ricerca file, ed esecuzione comandi in corrispondenza dei medesimi.
- **grep**
 - ricerca sottostringhe nei file; supporta espressioni regolari.
- **sed**
 - stream editor: permette modifiche al volo sul testo in arrivo dallo standard input.