

# Introduzione alla Programmazione con la Shell

v. aggiornata al 12/03/2005

(C) 2005 Christian Mair <chris@1006.org>

1

## Indice

1. Requisiti e Check Up
2. Introduzione e Note Storiche
3. Input, Output e Pipeline
4. Variabili
5. Condizionali
6. Operatori ed Aritmetica
7. Cicli
8. Funzioni
9. Semplici Interfacce Utenti
- x. Ringraziamenti e Copyright

2

# I. Requisiti e Check Up

- Familiarità con la riga di comando su un sistema Linux, in particolare la conoscenza dei comandi più importanti per:
  - spostarsi e orientarsi nella gerarchia del file system: `cd`, `pwd`, `ls`;
  - gestire i file e le cartelle: `cp`, `mv`, `rm`, `mkdir`, `rmdir`;
  - elaborare i file testuali: `cat`, `head`, `tail`, `grep`, `wc`, `sort`.
- Familiarità con le basi dell'amministrazione di sistema Linux, in particolare:
  - login da terminale testuale e da interfaccia grafica, accesso alla shell;
  - sistema dei diritti (utenti, gruppi, permessi sui file), comandi `chown` e `chmod`;
  - uso di un editor testuale (`emacs` o `vi`) oppure grafico (p.e. `gedit`);
  - gestire processi con la shell (`CTRL-Z`, `CTRL-C`, `bg`, `fg`, `&`) o con i comandi del sistema (`ps`, `kill`).

3

## Esercizio I.1

- a. Qual'è il significato dei seguenti comandi elencati nello slide precedente?

```
cd
pwd
ls
cp
mv
rm
cat
tail
grep
wc
sort
```

4

## Esercizio 1.2

- a. Creare un file testuale con il seguente contenuto:  

```
#!/bin/sh  
echo "ciao mondo!"
```
- b. Salvare il file con il nome `hello.sh` e renderlo eseguibile con il comando:  

```
chmod 755 hello.sh
```
- c. Eseguire il file dando il comando  

```
./hello.sh
```
- d. Domande:
  - A che cosa serve la prima riga?
  - Quali permessi vengono indicati dal numero (in base 8) 755?
  - Quando si esegue `./hello.sh`, a cosa serve il `./`?

5

## 2. Introduzione e Note Storiche

- Su un sistema Unix la **shell** è l'interfaccia primaria tra il sistema operativo e l'utente.
- Al momento del login (testuale) la shell viene lanciata e permette all'utente di eseguire i comandi del sistema operativo.
- La shell introduce una serie di comandi e meccanismi propri per rendere più potente il dialogo col sistema.  
Se p.e. diamo il comando  

```
ls *.txt
```

la shell espande `*.txt` con i nomi di tutti i file che finiscono in `.txt`. È importante capire che *non* è il comando `ls` che fa questo lavoro di espansione ma la shell.
- La più antica shell è la Bourne Shell, che prende il nome dal programmatore S. R. Bourne che l'ha ideata. Si ritrova in tutti i sistemi Unix come `/bin/sh`.
- Sotto Linux usiamo la `bash` del progetto GNU, che è compatibile alla Bourne Shell. Il nome sta per *Bourne Again Shell* - un gioco di parole ;-)

6

- **Programmare con la shell** vuole dire creare un file (detto programma) che contiene i comandi che altrimenti bisognerebbe ripetere manualmente.
- La difficoltà sta però nello scrivere il programma in modo sicuro e robusto. Se per esempio eseguiamo i seguenti comandi per creare un backup su server remoto:

```
mount backup-server:/backup /backup
tar cfz /backup/2005-01-01.tgz /home
umount /backup
```

non possiamo semplicemente copiare queste istruzioni in un file, perché introdurremmo una serie di problematiche:

- se `mount` fallisce (p. e. perché la rete non è disponibile), `tar` farebbe il backup sul mount point vuoto, magari riempiendo il disco locale;
- il nome del file `2005-01-01.tgz` dovrebbe essere variabile;
- manca la segnalazione di eventuali problemi all'amministratore.
- Sebbene la shell si possa considerare un linguaggio di programmazione completo il suo uso di solito è limitato al campo dell'amministrazione di sistema dove eccelle. Per lo sviluppo di applicazioni si preferiscono linguaggi come C++, Java, PHP, Python, ecc.

7

## Esercizio 2.1

- Vedere quale versione di sh (bash) è installato sul proprio PC:
 

```
sh --version
```
- Spostarsi in una cartella dove *non* sono presenti file che finiscono in `.txt` e dare il comando:
 

```
ls *.txt
```

 creare un paio di file con nomi che finiscono in `.txt` e ridare lo stesso comando:
 

```
ls *.txt
```

 Infine, sempre nella stessa cartella, con i file `.txt` ancora presenti dare:
 

```
ls '*.txt'
```
- Domande:
  - Che significato ha l'apostrofo nel comando precedente?
  - Per quanto riguarda la selezione dei file se il comando fosse stato `rm` anziché `ls` cambierebbe qualcosa?

8

## 3. Input, Output e Pipeline

- Ogni programma in esecuzione (chiamato processo) su un sistema Unix può aprire file per la lettura o scrittura. Ci sono tre file “speciali” già aperti dal sistema che ogni processo possiede:  
`stdin` (*standard input*) è di solito assegnato all’input da tastiera mentre `stdout` (*standard output*) e `stderr` (*standard error output*) sono di solito assegnati all’output su terminale.
- È possibile **ridirigere** questi file con degli operatori speciali!
  - `cmd > file` (lo `stdout` di `cmd` viene scritto su `file`, non sul terminale)
  - `cmd 2> file` (idem per `stderr`)
  - `cmd &> file` (`stdout` e `stderr` insieme vengono scritti su `file`)
  - `cmd < file` (`stdin` viene letto da `file`, non da tastiera)
  - Infine, aggiungendo `2>&1` alla fine di una riga di comando si ridirige `stderr` su `stdout`, indipendentemente da dove va `stdout`.

9

## Esercizio 3.1

- Provare gli operatori di ridirezione:
  - `ls > elenco.txt`
  - `cat > testo.txt`  
(dare qualche riga di testo e finire con `CTRL-D`)
  - Togliere errori superflui: confrontare l’output dei due comandi eseguiti come utente non privilegiato:  
`find /etc -type f`  
`find /etc -type f 2> /dev/null`
  - Creare un file `conto.txt` contenente `1+2` ed eseguire  
`bc < conto.txt`
- Domande:
  - Cosa fa il comando `find`?
  - Che proprietà ha il file speciale `/dev/null`?

10

- Avete notato che molti comandi Unix disponibile sono molto semplici, come p.e. il comando `wc` (che conta caratteri, parole o righe in file testuali). La potenza di questi comandi sta nella possibilità di connetterli per formare comandi più complessi!
- L'operatore `|` (chiamato *pipeline*) connette lo `stdout` di un comando allo `stdin` di un altro comando:  

```
cmd1 | cmd2
```
- Molti comandi possono lavorare su `stdin` anziché su un file dato come argomento. Così la pipeline rende possibile compattare operazioni che altrimenti richiederebbero passaggi intermedi. Anziché:  

```
ls > tmp.txt
wc -l tmp.txt
```

 si scrive  

```
ls | wc -l
```

 per avere il numero di file nella cartella in uso.
- Menzioniamo infine gli operatori `>>` e `2>>` che servono a aggiungere ad un file invece di crearlo ex novo come fanno `>` e `2>`.

11

## Esercizio 3.2

- Che significato dà l'argomento `-l` al comando `wc`?
- Quanti processi sono in esecuzione in questo momento sul vostro sistema?
- Provare a eseguire:  

```
cd /etc
file -b * | sort | uniq -c | sort -rn
```

 Il comando vi dà una statistica sui tipi di file che avete in `/etc`. Sapete spiegare passo per passo come funziona? Sarà necessario consultare le pagine del manuale (comando `man`).
- Abbiamo brevemente utilizzato il comando `find` semplicemente per farci elencare tutti i file a partire da una certa cartella. `find` sa fare molto di più! Riuscite a:
  - elencare tutti i file della vostra cartella home che sono stati modificati nelle ultime 24 ore?
  - elencare tutte le cartelle (solo cartelle!) nel percorso `/etc`?

12

## 4. Variabili

- **Variabili** sono contenitori temporanei di informazioni quali numeri o stringhe di caratteri a cui si da un nome che serve a richiamare l'informazione altrove.
- Variabili vengono assegnati con l'operatore uguale:  

```
FILE="ciao.txt"  
MAX="4"
```
- Il nome della variabile contiene i caratteri A-Z, i numeri 0-9 o il carattere `_` (*underscore*). Altri caratteri come lettere accentate sono da evitare. Un numero non può apparire all'inizio del nome. Infine, come sempre con la shell, occhio alle minuscole o maiuscole! `MAX` e `max` sono variabili diverse.
- Notare che sebbene non sia strettamente necessario useremo sempre apostrofi o doppi apici quando assegneremo valori costanti a variabili e utilizzeremo lettere maiuscole per i nomi di variabili. Queste sono solo convenzioni utili.
- È invece un errore aggiungere spazi prima o dopo il segno uguale.

13

- Una variabile può essere "usata" - cioè l'informazione che ha memorizzata può essere richiamata - con l'operatore `$`. Per esempio così:  

```
echo $FILE  
echo $MAX
```
- Una variabile può essere utilizzata direttamente all'interno di un'altra espressione.  
Ecco alcuni usi di variabili:  

```
cd $HOME/Desktop  
echo "raggiunto $MAX tentativi"
```
- Supponiamo di voler comporre il nome di un file che deve essere o `2D.txt` o `3D.txt` a secondo se la variabile `$DIM` è uguale a 2 o 3. Evidentemente  

```
FILE="$DIMD.txt"
```

non va, perché la variabile `$DIMD` non è definita (o è un'altra variabile). La soluzione è di usare le parentesi graffe nel modo seguente:  

```
FILE="{DIM}D.txt"
```

così da indicare esplicitamente il nome della variabile `$DIM`.

14

- L'apostrofo è invece utilizzato quando si vuole evitare che la shell espanda nomi di variabili. Esempio:  

```
echo 'attenzione: variabile $HOME non è definita'
```

In genere gli apici evitano l'espansione di tutti i caratteri speciali della shell, quali \* e \$.
- Se si vuole rendere disponibile una variabile ad altri programmi, p.e. quelli eseguiti dall'interno di un programma shell, bisogna "esportarli" con il comando `export`:  

```
PATH=/bin:/usr/bin:/$HOME/bin
export PATH
```
- Le variabili `$PATH` e `$HOME` fanno parte di una serie di variabili predefinite, le variabili d'ambiente (*environment variables*).  
`$PATH` indica l'insieme dei percorsi da valutare per trovare un comando quando non viene dato un percorso assoluto. `$HOME` indica la cartella home dell'utente.
- Il comando `env` senza argomenti può essere utilizzato per elencare tutte le variabili d'ambiente definite.

15

## Esercizio 4.1

- Editare il file `$HOME/.bashrc`. Aggiungere alla variabile d'ambiente `$PATH` il percorso dato dal punto (`.`), cioè la cartella attuale. Questo renderà superfluo specificare `./` quando si chiamano i propri programmi shell.  
Nota: root non dovrebbe mai fare questo per motivi di sicurezza.
- Il comando `which` indica la posizione di un eseguibile nel percorso (dato da `$PATH`). Provare ad eseguire:  

```
which ls
which rm
which cd
which export
```

Evidentemente `ls` e `rm` sono programmi esterni mentre `cd` e `export` non lo sono.
  - Avete una spiegazione del perché di questo?
  - Sapete trovare altri cosiddetti *shell builtin commands*?
- Elencare tutte le variabili d'ambiente definite per il proprio utente.

16

## Esercizio 4.2

- a. È possibile assegnare ad una variabile l'output di un comando anziché un valore costante. Lo si può fare in due modi.
- si usa l'operatore `$` insieme a parentesi tonde:  
`ELENCO=$(ls)`
  - oppure si usano gli accentini:  
`ELENCO=`ls``
- Scrivere un programma che genera un file `"back-2005-02-15.tgz"`, dove però la data è quella attuale. N.B.: serve il comando `date`.
- b. Oltre alle variabili definite da noi o dal sistema ci sono variabili "speciali" assegnati implicitamente della shell. Così p. e. le variabili `$1`, `$2`, `$3`, ecc... contengono gli argomenti passati al programma. Se p. e. si chiama `./script.sh chris` all'interno di `script.sh` è disponibile il valore `$1` che contiene `"chris"`.
- Realizzare un programma, che dato come argomento il nome di un file stampa informazioni su quel file (usare i comandi `file`, `stat`, ...).

17

## 5. Condizionali

- Per eseguire istruzioni solo in certi casi si usano i **condizionali**. Nel seguente programma `cmd` viene eseguito se e solo se `condizione` è vera:  

```
if condizione ; then
    cmd
fi
```
- `condizione` assume una delle seguenti forme:  

```
test $VAR          (vero se $VAR è non nullo)
test $S1 = $S2     (vero se $S1 è uguale a $S2)
test $S1 != $S2    (vero se $S1 non è uguale a $S2)
test $S1 > $S2     (vero se $S1 appare dopo $S2 in ord. alfabetico)
test $N1 -eq $N2   (vero se il numero $N1 è uguale a $N2)
test $N1 -ne $N2   (vero se i numeri $N1 e $N2 sono diversi)
test $N1 -gt $N2   (vero se il numero $N1 è più grande di $N2)
test -f $F         (vero se $F indica un file esistente)
test -d $F         (vero se $F indica una cartella esistente)
```

e molte altre (vedere `man test`).

18

- A volte è utile specificare dei comandi *alternativi* da eseguire se una data condizione *non* è vera. Perciò è possibile estendere la costruzione `if` con un blocco `else`:

```
if condizione ; then
  cmd1
else
  cmd2
fi
```

Se *condizione* è vera, viene eseguito `cmd1`, altrimenti `cmd2`.

- Notare che i blocchi dati da `cmd1` o `cmd2`, possono a loro volta consistere di una sequenza arbitrari di comandi. In particolare possono contenere blocchi `if`. In quel caso l'uso dello spazio per indentare le istruzioni sebbene non sia obbligatorio è veramente utile per aumentare la leggibilità del codice.
- Un sinonimo per il comando `test` è il comando parentesi quadra aperta (`[`). Come ultimo argomento aspetta la parentesi quadra chiusa (`]`). Con questa notazione un `if` con il primo test dello slide precedente viene scritto nel modo seguente, preferito da molti:

```
if [ $VAR ] ; then
```

19

## Esercizio 5.1

- Migliorare il programma dell'esercizio 4.2b che stampa informazioni su un file! Se viene chiamato senza argomento il programma dovrà stampare delle istruzioni di come usarlo. Se viene chiamato con il nome di un file non esistente dovrà stampare un messaggio d'errore.
- Scrivere un programma che gioca a "morra cinese". Il programma accetta come argomenti le giocate dei due giocatori A e B in ordine. Come risultato stampa il giocatore vincente. Esempio:
 

```
./gioca.sh carta sasso
vince A
./gioca.sh carta forbice
vince B
```
- Scrivere un programma che emette un messaggio di attenzione se il file system di sistema supera il 90% di occupazione (bisognerà utilizzare i comandi `df` e `cut`). Sapete utilizzare cron? Allora impostatelo in modo che il vostro programma venga eseguito ogni 5 minuti.

20

- Per evitare di dover scrivere una lunga fila di `if` quando ci sono tante scelte da fare, la shell ha un'ulteriore costruzione condizionale: il `case`:

```
case $VAR in
  valore1)
    cmd1
    ;;
  valore2)
    cmd2
    ;;
  valore3)
    cmd3
    ;;
  *)
    cmdx
    ;;
esac
```

Se `$VAR` assume uno dei valori da `valore1` a `valore3` vengono eseguiti i comandi da `cmd1` a `cmd3` rispettivamente. Altrimenti viene eseguito il comando `cmdx`.

21

## Esercizio 5.2

- a. Scrivere un programma start/stop nello stile dei programmi in `/etc/init.d` che servono a gestire i servizi.
  - Il programma può essere lanciato con uno di due argomenti: "start" o "stop".
  - Se viene eseguito con l'argomento "start" lancia in background un programma innocuo (p.e. `sleep 10000 &`) e registra in un file il PID (*process id*) del programma appena lanciato.
  - Se viene eseguito con l'argomento "stop" recupera il PID e termina il processo (usando il comando `kill`).
  - Nota: il PID dell'ultimo comando lanciato è disponibile nella variabile speciale `#!`.
  - Il programma dovrebbe essere robusto nel senso che dovrebbe dare un errore se si vuole far partire il "servizio" più di una volta o se si vuole terminare il "servizio" prima di averlo lanciato.

22

## 6. Operatori ed Aritmetica

- Nel esercizio 2.1 abbiamo osservato che la shell espande l'espressione data `*.txt` a tutti i nomi di file presenti che finiscono in `.txt`. Questa espansione è chiamata **file globbing**. Ci sono altri caratteri usati per il file globbing. Ecco un elenco:
  - `*` indica una stringa qualsiasi di zero o più caratteri
  - `?` indica un singolo carattere arbitrario
  - `[ ]` indica un singolo carattere arbitrario tra quelli contenuti nelle `[ ]`, in particolare:
    - `-` è usato per indicare un carattere di una successione, p.e. `[a-z]`
    - `!` o `^` all'inizio sono usati per invertire la selezione.
- Esempi di file globbing:
  - `*txt` indica `hello.txt` e `hellotxt`, ma non `hello.txtx`
  - `ch*` indica `chris` e `chros`, ma non `Chris`
  - `[a-zA-Z]*` indica `chris` e `Chris`, ma non `_chris`
  - `[^0-9]*` indica `ad.txt`, ma non `2d.txt`
  - `hello.?` indica `hello.c` e `hello.h`, ma non `hello.txt`

23

- Operatori **booleani**:
  - La *negazione* è indicata con il punto esclamativo (`!`). In un espressione di test `!condizione` è vera se e solo se `condizione` è falsa.
  - L'operazione *AND* è indicata con il simbolo `&&`. La condizione composta `condizione1 && condizione2` è vera se e solo se entrambe le condizioni sono vere.
  - L'operazione *OR* è indicata con il simbolo `||`. La condizione composta `condizione1 || condizione2` è vera se e solo se almeno una delle due condizioni è vera.
- Esempio 1: verifica che entrambi i file sono presenti:

```
if [ -f hello.c ] && [ -f hello.h ] ; then
  cc hello.c
fi
```
- Esempio 2: ferma l'esecuzione del programma se la cartella `install` manca:

```
if [ ! -d install ] ; then
  echo "manca la cartella \"install\""
  exit 1
fi
```

24

## Esercizio 6.1

- a. Nella vostra cartella `$HOME` ci sono file “nascosti” che iniziano con un punto. Ricordiamo che per listarli bisogna dare l’opzione `-a` al comando `ls`.
  - Usare `echo` per fare elencare tutti i file che iniziano con un punto nella cartella `$HOME`, escluso i file speciali “`..`” e “`.`” che indicano, rispettivamente, la cartella superiore e quella attuale nella gerarchia del file system.
- b. Scrivere un programma `myps` che a secondo del numero di processi del sistema stampa un messaggio:
  - se i processi sono meno di 20, stampa “pochi processi”,
  - se i processi sono compreso da 20 e 50, stampa “qualche processo”
  - se i processi sono più di 50, stampa “tanti processi”
- c. Scrivere un programma `mytar` che usa `tar` per creare un archivio:
  - chiamato come `mytar x.tar y` crea l’archivio `x.tar` della cartella `y`,
  - chiamato invece come `mytar y` sottintende il nome `backup.tar`.

25

- Ogni comando termina con un **exit code** (“codice d’uscita”). Per convenzione zero indica il buon esito del comando, mentre un numero diverso da zero indica un errore.  
Il comando `test` in particolare termina con 0 se la condizione data è valida e vera e con 1 altrimenti.  
Questo vuole dire che possiamo usare l’`if` direttamente con un comando qualsiasi per verificarne il buon esito.  
Nel seguente esempio verifichiamo se la creazione della cartella `/xxx` riesce:

```
if mkdir /xxx ; then
    echo "ho creato la cartella"
else
    echo "errore: non ho creato la cartella"
fi
```
- Risulta a volte utile la variabile speciale  `$?`  che contiene l’exit code dell’ultimo comando eseguito dalla shell.
- Si può usare il comando `exit` per terminare l’esecuzione di un programma. Di solito si segue la convenzione di cui sopra e si usa `exit 0` per indicare il buon esito del programma, e `exit 1` (o un altro numero diverso da 0) per indicare che c’è stato un problema.

26

## Esercizio 6.2

- a. Gli operatori `&&` e `||` nella shell vengono valutati da sinistra a destra e in modo più efficiente possibile.

Per l'AND (`&&`) questo vuole dire che se la prima condizione è falsa tutta l'espressione è falsa e la seconda condizione non viene più valutata.

Per l'OR (`||`) questo vuole dire che se la prima condizione è vera tutta l'espressione è vera e la seconda condizione non viene più valutata.

Questo introduce delle sottigliezze.

- Il seguente check è corretto:

```
if [ $FILE ] && [ -f $FILE ] ; then
    echo "$FILE ok"
fi
```

Spiegare perché bisogna fare i test proprio nell'ordine dato!

- A volte si vedono dei comandi di questo genere:

```
mkdir /install && cp file /install
```

Spiegarne il funzionamento!

27

- La shell è in grado di eseguire operazioni aritmetiche sugli interi.

Esempi:

```
let A=7/3
echo $A
```

- Alcune operazioni date in ordine di precedenza sono:

<code>N++</code>	<code>N--</code>	post-increment	
<code>++N</code>	<code>--N</code>	pre-increment	
<code>**</code>		potenza	
<code>*</code>	<code>/</code>	<code>%</code>	moltiplicazione, divisione e modulo
<code>+</code>	<code>-</code>		addizione e sottrazione

- In alternativa a `let` si può utilizzare la seguente notazione:

```
A=$(( 7 / 3 ))
echo $A
```

28

## 7. Cicli

- **Cicli** (*loops*) sono un importante strumento per la programmazione. I cicli permettono di ripetere operazioni.

Una delle più comuni espressioni per scrivere un ciclo è il ciclo `for` che assume la seguente forma:

```
for variabile in lista ; do
    cmd
done
```

Qui `cmd` viene eseguito una volta per ogni elemento di `lista` e `variabile` assume di volta in volta il valore di ognuno degli elementi.

- Ecco un esempio per cui la lista è semplicemente data da stringhe costanti:

```
for VAR in "bart" "lisa" "maggie" ; do
    echo "Ho visto $VAR."
done
```

29

- La lista può anche essere data dall'output di un comando:

```
for VAR in $(seq 1 10) ; do
    echo "Ho visto $VAR"
done
```

o da un operazione di *file globbing*:

```
for VAR in *.txt ; do
    echo "Ho visto $VAR"
done
```

o dagli argomenti al programma con la variabile speciale `$@`, che è la lista di tutti gli argomenti dati `$1, $2, $3...`:

```
SOMMA=0
for ARG in $@ ; do
    SOMMA=$((SOMMA+ARG))
done
echo $SOMMA
```

30

- A volte si preferisce ripetere la operazione condizionalmente. È possibile farlo con il ciclo `while`. A differenza del ciclo `for`, al posto dell'iterazione sugli elementi di una lista appare una condizione. Il ciclo viene eseguito mentre la condizione è vera. Il primo esempio stampa i numeri da 0 a 99:

```
I=0
while [ $I -lt 100 ] ; do
    echo $I
    I=$((I+1))
done
```

Il secondo esempio continua a chiedere all'utente se vuole continuare fino a quando non risponde con un input diverso da "s":

```
RISPOSTA="s"
while [ $RISPOSTA = "s" ] ; do
    echo -n "vuoi continuare (s/n)? "
    read RISPOSTA
done
```

31

## Esercizio 7.1

- Estendere l'esempio del programma che calcola la somma dei suoi argomenti in modo che accetti anche numeri decimali.

Nota: bisognerà utilizzare il programma esterno `bc -l`.

- Scrivere un programma che data come argomento una lista di file (in senso generico: file regolari, cartelle, link simbolici...), per ogni file dato determina se è una cartella o meno. L'output dovrebbe avere il seguente formato:

```
Music è una cartella
pippo.txt non è una cartella
```

Versione avanzata: formattare meglio output (allineare le colonne) con il comando `printf`.

- Scrivere un programma `potenza` che, se chiamato come `potenza A B` calcola `A` elevato alla `B`-esima potenza senza però usare l'operatore `**`. In altre parole bisogna moltiplicare `A` per se stesso per `B` volte.

32

## 8. Funzioni

- **Funzioni** (chiamate anche *metodi* o *procedure*) servono per organizzare meglio i propri programmi. Procedure ripetute possono essere definite e nominate per poi essere usate in più parti del programma.

La definizione di una funzione nella shell assume la seguente forma:

```
function nome() {  
    cmd  
}
```

Ogni volta che viene dato il comando *nome* la shell chiama la funzione omonima e il comando (o la sequenza di comandi) *cmd* viene eseguito.

- Note sulle funzioni:
  - nella funzione le variabili speciali \$1, \$2, ... prendono il valore dei parametri con cui la funzione è chiamata,
  - si può usare il comando `local` per definire una variabile all'interno della funzione non visibile al di fuori della funzione stessa.

33

- Ecco un esempio: `stampa_somma` stampa la somma degli argomenti dati:

```
function stampa_somma() {  
    local SUM=0  
    for N in $@ ; do  
        SUM=$((SUM+$N))  
    done  
    echo $SUM  
}  
  
stampa_somma 3 4      # prima chiamata - stampa 7  
stampa_somma 1 2 3   # seconda chiamata - stampa 6
```

- Notare che la funzione `stampa_somma` deve essere definita prima del punto dove viene chiamata.
- Sarebbe possibile implementare la stessa funzionalità in un programma esterno. Con le funzioni però si evita la creazione di un altro processo shell alla chiamata (e quindi le funzioni sono più performanti) e si evita di avere il programma sparso in troppi file singoli.

34

## 9. Semplici Interfacce Utenti

- Il comando **dialog** permette di creare interfacce utenti in modalità testuale. Vedere **man dialog** per tutte le possibilità.
- Esempi:
  - Il seguente comando mostra il messaggio "Ciao" all'utente in una box alta 8 righe e larga 40 caratteri e aspetta che l'utente prema il pulsante *Ok*:

```
dialog --msgbox "Ciao" 8 40
```
  - Il seguente comando chiede all'utente di inserire del testo in una maschera, il testo inserito viene stampato su **stderr**. L'exit code di **dialog** è 0 se l'utente ha dato *Ok* o diverso da 0 se ha dato *Cancel*:

```
dialog --inputbox "Inserire informazione:" 8 40
```
- Esercizio: scrivere un programma **indovinello** che genera un numero random (comando **jot -r 1 1 100**) e continua a chiedere all'utente di indovinare il numero ripetutamente finché non lo indovina o preme *Cancel*. Dopo ogni tentativo viene data un indicazione all'utente: se il numero da indovinare è minore o maggiore di quello dato.

35

## x. Ringraziamenti e Copyright

- Ringraziamenti:  
Ringrazio i ragazzi del corso "Sviluppatori in sistemi liberi - Open Source" di Kantea S.c.r.l. per i loro suggerimenti e correzioni ortografiche ;-)
- Copyright:  
Quest'opera è stata rilasciata sotto la licenza Creative Commons Attribuzione-StessaLicenza.
  - Tu sei libero:
    - di utilizzare l'opera per scopi commerciali.
    - di distribuire, comunicare al pubblico, rappresentare o esporre in pubblico l'opera
    - di creare opere derivate
  - Alle seguenti condizioni:
    - Attribuzione. Devi riconoscere la paternità dell'opera all'autore originario.
    - Condividi sotto la stessa licenza. Se alteri, trasformi o sviluppi quest'opera, puoi distribuire l'opera risultante solo per mezzo di una licenza identica a questa.
    - In occasione di ogni atto di riutilizzo o distribuzione, devi chiarire agli altri i termini della licenza di quest'opera.
    - Se ottieni il permesso dal titolare del diritto d'autore, è possibile rinunciare a ciascuna di queste condizioni.
    - Le tue utilizzazioni libere e gli altri diritti non sono in nessun modo limitati da quanto sopra.
  - Questo è un riassunto in lingua corrente dei concetti chiave della licenza completa (codice legale) sul sito <http://creativecommons.org/licenses/by-sa/2.0/it/legalcode>.

36