

# Introduzione a UNIX shell e file comandi

# Shell

Programma che permette di far *interagire l'utente (interfaccia testuale) con SO tramite comandi*

- ❑ resta in attesa di un comando...
- ❑ ... mandandolo in esecuzione alla pressione di <ENTER>

In realtà (lo vedremo ampiamente) *shell è un interprete comandi evoluto*

- ❑ potente *linguaggio di scripting*
- ❑ interpreta ed esegue comandi da *standard input* o da *file comandi*

# Differenti shell

- La shell non è unica, un sistema può metterne a disposizione varie
  - *Bourne shell* (standard), C shell, Korn shell, ...
  - L'implementazione della *bourne shell* in Linux è **bash** (/bin/bash)
- Ogni utente può indicare la shell preferita
  - La scelta viene memorizzata in **/etc/passwd**, un file contenente le informazioni di tutti gli utenti del sistema
- La shell di login è quella che richiede inizialmente i dati di accesso all'utente
  - Per *ogni utente connesso* viene generato un **processo dedicato** (che esegue la shell)

# Ciclo di esecuzione della shell

```
loop forever
  <LOGIN>
    do
      <ricevi comando da file di input>
      <interpreta comando>
      <esegui comando>
    while (! <EOF>)
  <LOGOUT>
end loop
```

# Accesso al sistema: login

Per accedere al sistema bisogna possedere una coppia ***username e password***

- NOTA: UNIX è case-sensitive

SO verifica le credenziali dell' utente e manda in esecuzione la sua ***shell di preferenza***, posizionandolo in un ***direttorio di partenza***

- Entrambe le informazioni si trovano in `/etc/passwd`

## Comando `passwd`

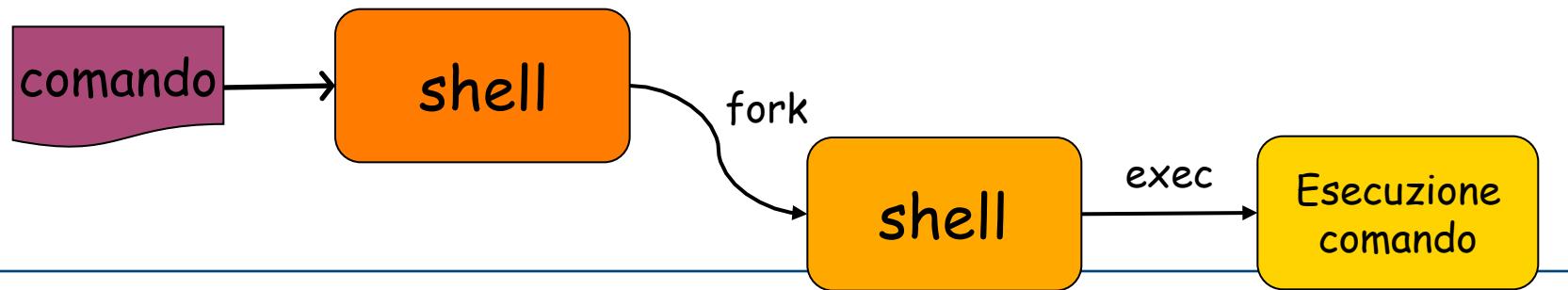
- È possibile ***cambiare la propria password*** di utente, mediante il comando `passwd`
- Se ci si dimentica della password, bisogna chiedere all' amministratore di sistema (utente `root`)

# Uscita dal sistema: logout

- Per uscire da una shell qualsiasi si può utilizzare il comando **exit** (che invoca la system call `exit()` per quel processo)
- Per uscire dalla shell di login
  - `logout`
  - `CTRL+D` (che corrisponde al carattere `<EOF>`)
  - `CTRL+C`
- Per rientrare nel sistema bisogna effettuare un nuovo login

# Esecuzione di un comando

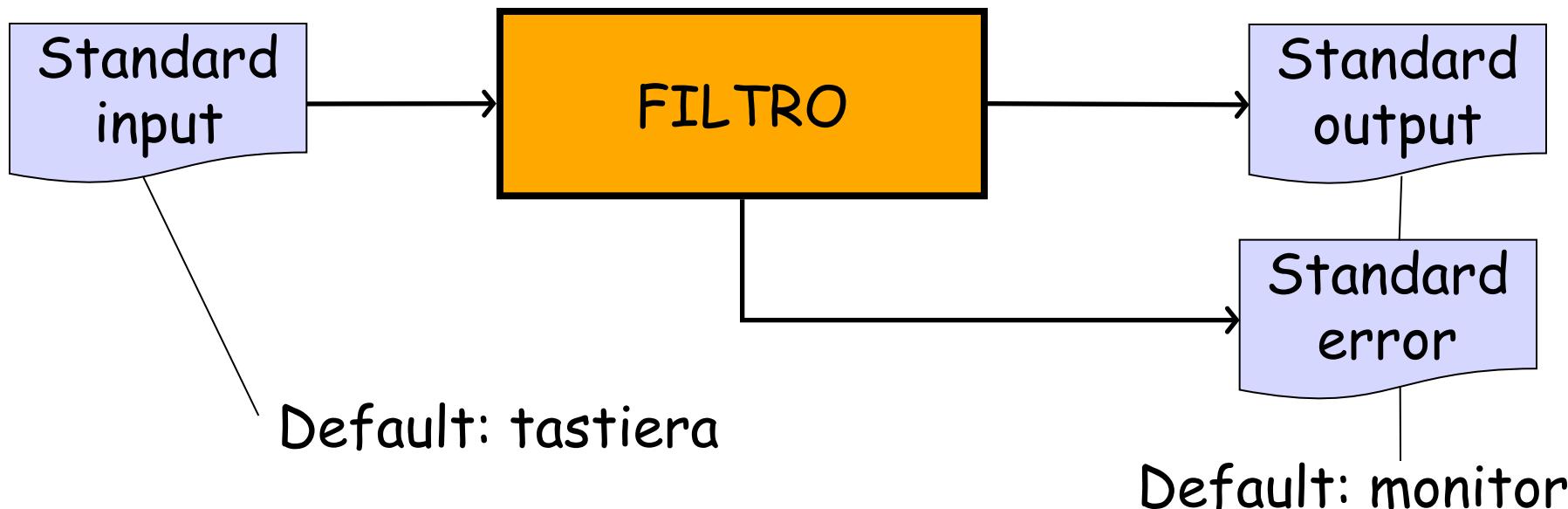
- Ogni comando richiede al SO l' esecuzione di una particolare azione
- I *comandi principali* del sistema si trovano nella directory `/bin`
- Possibilità di *realizzare nuovi comandi (scripting)*
- Per ogni comando, shell *genera un processo figlio dedicato alla sua esecuzione*
  - Il processo padre *attende la terminazione del comando* (foreground) o *prosegue in parallelo* (background)



# Comandi e input/output

*I comandi UNIX si comportano come FILTRI*

- ❑ un filtro è un programma che riceve un ingresso da un input e produce il risultato su uno o più output



# Comandi shell Linux: filtri

Alcuni esempi:

- ❑ **grep <text> [<file>...]**

Ricerca di testo. Input: (lista di) file. Output: video

- ❑ **tee <file>**

Scrive l'input sia su file, sia sul canale di output

- ❑ **sort [<file>...]**

Ordina alfabeticamente le linee. Input: (lista di) file. Output: video

- ❑ **rev <file>**

Inverte l'ordine delle linee di file. Output: video

- ❑ **cut [-options] <file>**

Seleziona colonne da file. Output: video

# Ridirezione di input e output

Possibile ridirigere input e/o output di un comando facendo sì che non si legga da stdin (e/o non si scriva su stdout) **ma da file**

- ❑ *senza cambiare il comando*
- ❑ *completa omogeneità tra dispositivi e file*
- Ridirezione dell' input
  - ❑ `comando < file_input`
- Ridirezione dell' output
  - ❑ `comando > file_output`
  - ❑ `comando >> file_output`

Aperto in lettura

Aperto in scrittura  
(nuovo o sovrascritto)

Scrittura in  
append

# Esempi

- `ls -l > file`

File conterrà il risultato di `ls -l`

- `sort < file > file2`

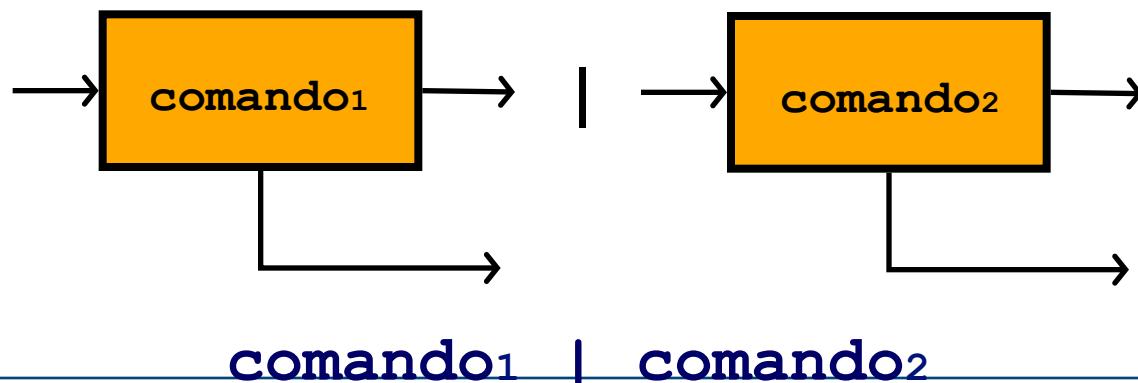
Ordina il contenuto di `file` scrivendo il risultato su `file2`

- Cosa succede con `> file` ?

# piping

*L'output di un comando può esser diretto a diventare l'input di un altro comando (piping)*

- In DOS: *realizzazione con file temporanei* (primo comando scrive sul file temporaneo, secondo legge da questo)
- In UNIX: *pipe come costrutto parallelo* (l'output del primo comando viene reso disponibile al secondo e consumato appena possibile, non ci sono file temporanei)
- Si realizza con il carattere speciale '|'



# Esempi di piping

- `who | wc -l`
  - Conta gli utenti collegati
- `ls -l | grep ^d | rev | cut -d' ' -f1 | rev`
  - Che cosa fa? Semplicemente mostra i nomi dei sottodirettori della directory corrente
    - `ls -l` lista i file del direttorio corrente
    - `grep` filtra le righe che cominciano con la lettera d (pattern `^d`, vedere il `man`) ovvero le directory (il primo carattere rappresenta il tipo di file)
    - `rev` rovescia l' output di `grep`
    - `cut` taglia la prima colonna dell' output passato da `rev`, considerando lo spazio come delimitatore (vedi `man`)
      - ✓ quindi, poiché `rev` ha rovesciato righe prodotte da `ls -l`, estrae il nome dei direttori 'al contrario'
    - `rev` raddrizza i nomi dei direttori

**Suggerimento:** aggiungere i comandi uno alla volta (per vedere cosa viene prodotto in output da ogni pezzo della pipe)

# Metacaratteri ed espansione

# Metacaratteri

Shell riconosce **caratteri speciali (wild card)**

- **\*** una qualunque stringa di zero o più caratteri in un nome di file
- **?** un qualunque carattere in un nome di file
- **[zfc]** un qualunque carattere, in un nome di file, compreso tra quelli nell'insieme. Anche **range** di valori: **[a-d]**:

**Per esempio:** `ls [q-s]*` lista i file con nomi che iniziano con un carattere compreso tra q e s

- **#** commento fino alla fine della linea
- **\** escape (segnala di *non interpretare* il Carattere successivo come speciale)

# Esempi con metacaratteri

**ls [a-p,1-7]\*[c,f,d] ?**

- elenca i file i cui nomi hanno come iniziale un carattere compreso tra 'a' e 'p' oppure tra 1 e 7, e il cui penultimo carattere sia 'c', 'f', o 'd'

**ls \*\\*\***

- Elenca i file che contengono, in qualunque posizione, il carattere \*

# Variabili nella shell

In ogni shell è possibile **definire un insieme di variabili** (trattate come stringhe) con **nome e valore**

- i riferimenti ai **valori delle variabili** si fanno con il **carattere speciale \$** (\$nomevariabile)
- si possono fare **assegnamenti**  
nomevariabile=\$nomevariabile  
    *l-value*                                  *r-value*

## Esempi

- **x=2**
- **echo \$x** (visualizza 2)
- **echo \$PATH** (mostra il contenuto della variabile PATH)
- **PATH=/usr/local/bin:\$PATH** (aggiunge la directory /usr/local/bin alle directory del path di default)

# Ambiente di esecuzione

Ogni comando esegue *nell'ambiente associato (insieme di variabili di ambiente definite) alla shell* che esegue il comando

- ogni shell *eredita l'ambiente dalla shell* che l'ha creata
- nell'ambiente ci sono **variabili** alle quali il comando può fare riferimento:
  - **variabili con significato standard:** PATH, USER, TERM, ...)
  - **variabili user-defined**

# Variabili

Per vedere tutte le variabili di ambiente e i valori loro associati si può utilizzare il comando **set**:

```
$ set
```

```
BASH=/usr/bin/bash
HOME=/space/home/wwwlia/www
PATH=/usr/local/bin:/usr/bin:/bin
PPID=7497
PWD=/home/Staff/AnnaC
SHELL=/usr/bin/bash
TERM=xterm
UID=1015
USER=anna
```

# Espressioni

Le **variabili shell** sono **stringhe**!

È comunque possibile forzare l'interpretazione numerica di stringhe che contengono la codifica di valori numerici

- comando **expr**:

```
expr 1 + 3
```

Esempio:

```
var = 5
```

```
echo risultato: $var+1
```

\$var+1 è il risultato della corrispondente espressione?

a che cosa  
serve?

```
echo risultato: `expr $var + 1`
```

# Esempio

```
#!/bin/bash
A=5
B=8
echo A=$A, B=$B
C=expr $A + $B
echo C=$C
D=`expr $A + $B`
echo D=$D
```

*file somma*

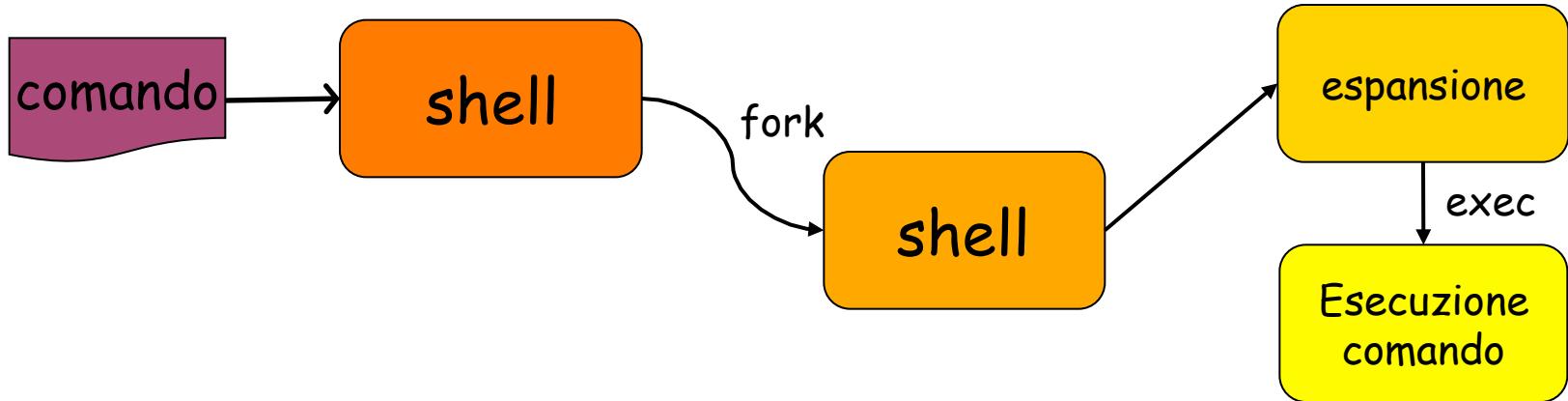
bash-2.05\$ ./somma

A=5, B=8  
C=expr 5 + 8  
D=13

invocazione

output

# Espansione



Prima della esecuzione:

- La shell *prima prepara i comandi come filtri*: ridirezione e piping di ingresso uscita
- Successivamente, se shell trova caratteri speciali, *produce delle sostituzioni (passo di espansione)*

# Passi di sostituzione

## Sequenza dei passi di sostituzione

### 1) Sostituzione dei comandi

- comandi contenuti tra `` (backquote) sono eseguiti e sostituiti dal risultato prodotto

### 2) Sostituzione delle variabili e dei parametri

- *nomi delle variabili* (\$nome) sono espansi nei valori corrispondenti

### 3) Sostituzione dei metacaratteri in nomi di file

- metacaratteri \* ? [ ] sono espansi nei *nomi di file* secondo un meccanismo di *pattern matching*

# Inibizione dell'espansione

In alcuni casi è necessario *privare i caratteri speciali del loro significato*, considerandoli come caratteri normali

- ❑ \carattere successivo è considerato come un normale carattere
- ❑ ' ' (apici): proteggono da qualsiasi tipo di espansione
- ❑ " " (doppi apici) proteggono dalle espansioni con l'eccezione di \$ \ ` ` (backquote)

# Esempi sull' espansione

- `rm '*$var'*`
  - Rimuove i file che cominciano con \*\$var
- `rm "*$var"*`
  - Rimuove i file che cominciano con \*<contenuto della variabile var>
- `host203-31:~ anna$ echo "<`pwd`>"</Users/AnnaC>"`
- `host203-31:~ anna$ echo '<`pwd`>'<`pwd`>"`
- `A=1+2 B=`expr 1 + 2``
  - In A viene memorizzata la stringa 1+2, in B la stringa 3 (expr forza la valutazione aritmetica della stringa passata come argomento)

# Riassumendo: passi successivi del parsing della shell

R ridirezione dell'input/output

`echo hello > file1` # crea `file1` e # collega a `file1` lo `stdout` di `echo`

1. sostituzione dei comandi (backquote)

``pwd`` → `/temp`

2. sostituzione di variabili e parametri

`$HOME` → `/home/staff/AnnaC`

3. sostituzione di metacaratteri

`plu?o*` → `plutone`

# Esempio

- **Provare:**

C=1

```
echo ciao > '*pippo'$C
```

```
C=`expr $C + 1`
```

```
echo hello > '*pippo'$C
```

```
C=`expr $C + 1`
```

```
echo hola > '*pippo'$C
```

```
C=`expr $C + 1`
```

```
ls '*pippo'* | grep "*pippo`expr $C - 1`" > ciccio
```

Che cosa fa questa sequenza di comandi? Quale sarà il contenuto di ciccio?

# Scripting: realizzazione file comandi

# File comandi

- Shell è un *processore comandi* in grado di interpretare *file sorgenti in formato testo e contenenti comandi* -> *file comandi (script)*

**Linguaggio comandi** (vero e proprio linguaggio programmazione)

- Un *file comandi* può comprendere
  - *statement per il controllo di flusso*
  - *variabili*
  - *passaggio dei parametri*

NB:

- *quali statement* sono disponibili dipende da *quale shell* si utilizza
- *file comandi* viene *interpretato* (non esiste una fase di compilazione)
- *file comandi deve essere eseguibile* (usare *chmod*)

# Scelta della shell

La prima riga di un file comandi deve specificare *quale shell si vuole utilizzare*:  
**`#! <shell voluta>`**

- Es: `#!/bin/bash`
- `#` è visto dalla shell come un commento ma...
- `#!` è visto da SO come identificatore di un file di script
  - SO capisce così che l'interprete per questo script sarà `/bin/bash`
- Se questa riga è assente viene scelta la shell di preferenza dell'utente

# File comandi

È possibile memorizzare *sequenze di comandi all'interno di file eseguibili*:

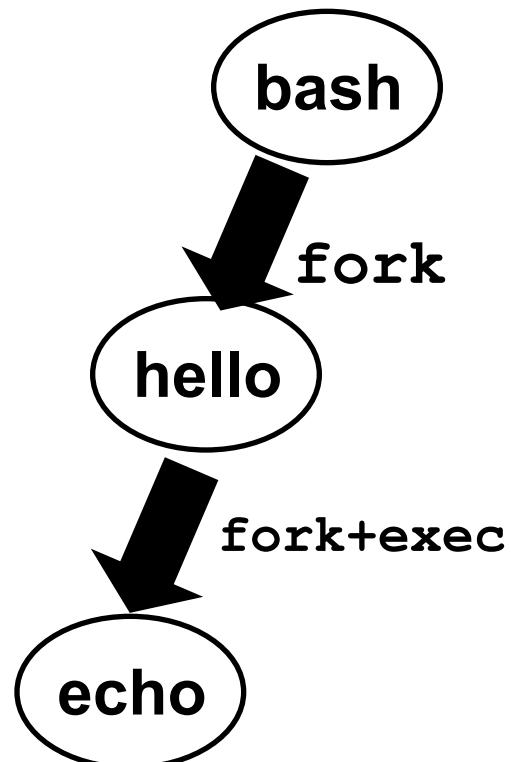
*file comandi (script)*

Ad esempio:

```
#!/bin/bash
echo hello world!
```

*file hello*

```
bash-2.05$ hello
hello world!
```



# Passaggio parametri

```
./nomefilecomandi arg1 arg2 ... argN
```

Gli argomenti sono **variabili posizionali** nella linea di invocazione contenute nell'ambiente della shell

- \$0 rappresenta il comando stesso
- \$1 rappresenta il primo argomento ...
- è possibile far scorrere tutti gli argomenti verso sinistra  
→ **shift**

\$0 non va perso, solo gli altri sono spostati (\$1 perso)

	\$0	\$1	\$2
prima di <b>shift</b>	nomefilecom	-w	/usr/bin
dopo <b>shift</b>	nomefilecom	/usr/bin	

- è possibile riassegnare gli argomenti → **set**
  - **set exp1 exp2 exp3 ...**
  - gli argomenti sono assegnati secondo la posizione

# Altre informazioni utili

Oltre agli argomenti di invocazione del comando

- **\$\*** insieme di *tutte le variabili posizionali*, che corrispondono arg del comando: \$1, \$2, ecc.
- **\$#** *numero di argomenti* passati (**\$0 escluso**)
- **\$?** valore (int) restituito dall'ultimo comando eseguito
- **\$\$** id numerico del processo in esecuzione (pid)

## Semplici forme di input/output

- **read var1 var2 var3** **#input**
- **echo var1 vale \$var1 e var2 \$var2** **#output**
  - **read** la stringa in ingresso viene attribuita alla/e variabile/i secondo corrispondenza posizionale

# Strutture di controllo

Ogni comando in uscita restituisce un **valore di stato**, che indica il suo **completamento o fallimento**

Tale valore di uscita è posto nella variabile ?

- `$?` può essere riutilizzato in espressioni o per controllo di flusso successivo

Stato vale usualmente:

- zero: comando OK
- valore positivo: errore

Esempio

```
host203-31:~ daniela$ cp a.com b.com
cp: cannot access a.com
host203-31:~ daniela$ echo $?
```

2

# test

Comando per la **valutazione di una espressione**

- **test -<opzioni> <nomefile>**

Restituisce uno stato uguale o diverso da zero

- valore **zero → true**
- valore **non-zero → false**

**ATTENZIONE: convenzione opposta rispetto al linguaggio C!**

- Motivo: i codici di errore possono essere più di uno e avere significati diversi

# Alcuni tipi di test: stringhe

## Test

- ❑ **-f <nomefile>** esistenza di file
- ❑ **-d <nomefile>** esistenza di direttori
- ❑ **-r <nomefile>** diritto di lettura sul file (**-w** e **-x**)
- ❑ **test <stringa1> = <stringa2>** uguaglianza stringhe
- ❑ **test <stringa1> != <stringa2>** diversità stringhe

## ATTENZIONE:

- gli **spazi intorno a =** (o **a !=**) sono **necessari**
- stringa1 e stringa2 possono contenere metacaratteri (attenzione alle espansioni)
- ❑ **test -z <stringa>** vero se **stringa nulla**
- ❑ **test <stringa>** vero se **stringa non nulla**

# Alcuni tipi di test: valori numerici

- ❑ `test <val1> -gt <val2>` ( $\text{val1} > \text{val2}$ )
- ❑ `test <val1> -lt <val2>` ( $\text{val1} < \text{val2}$ )
- ❑ `test <val1> -le <val2>` ( $\text{val1} \leq \text{val2}$ )
- ❑ `test <val1> -ge <val2>` ( $\text{val1} \geq \text{val2}$ )

# Strutture di controllo: alternativa

```
if <lista-comandi>
  then
    <comandi>
  [elif <lista_comandi>
    then <comandi>]
  [else <comandi>]

  fi
```

## ATTENZIONE:

- le parole chiave (do, then, fi, ...) devono essere o **a capo o dopo il separatore** ;
- if controlla il valore in uscita **dall'ultimo comando di <lista-comandi>**

# Esempio

```
# fileinutile
# risponde "sì" se invocato con "sì" e un numero
< 24
if test $1 = sì -a $2 -le 24
  then echo sì
  else echo no
fi
```

---

```
#test su argomenti
if test $1; then echo OK
  else echo Almeno un argomento
fi
```

---

# Alternativa multipla

```
# alternativa multipla sul valore di var
case <var> in
  <pattern-1>
    <comandi>;;
  ...
  <pattern-i> | <pattern-j> | <pattern-k>
    <comandi>;;
  ...
  <pattern-n>
    <comandi> ;;
esac
```

**Importante: nell' alternativa multipla si possono usare metacaratteri per fare pattern-matching**

# Esempi

```
read  risposta
case $risposta in
  S* | s* | Y* | y* ) <OK>;;
  * ) <problema>;;
esac
```

---

```
# append: invocazione  append [dove]  adove
case $# in
  1) cat >> $1;;
  2) cat < $1 >> $2;;
  *) echo uso: append [dove]  adove;
      exit 1;;
esac
```

---

# Cicli enumerativi

```
for <var> [in <list>] # list=lista di
  stringhe
do
  <comandi>
done
```

- scansione della lista <list> e *ripetizione del ciclo per ogni stringa presente nella lista*
- scrivendo solo **for i** si itera con valori di **i** in \$\*

# Esempi

- **for i in \***
  - esegue per tutti i file nel directory corrente
- **for i in `ls s\*`**  
**do <comandi>**  
**done**
- **for i in `cat file1`**  
**do <comandi per ogni parola del file file1>**  
**done**
- **for i in 0 1 2 3 4 5 6**  
**do**  
**echo \$i**  
**done**

# Ripetizioni non enumerative

```
while <lista-comandi>
do
  <comandi>
done
```

Si ripete se il valore di stato dell'ultimo comando della lista è zero  
(successo)

```
until <lista-comandi>
do
  <comandi>
done
```

Come while, ma inverte la condizione

## Uscite anomale

- vedi C: **continue**, **break** e **return**
- **exit [status]**: system call di UNIX, anche comando di shell

# Esempi di file comandi

## Esercizio:

- scrivere un file comandi che ogni 5 secondi controlli se sono stati *creati o eliminati file in una directory*. In caso di cambiamento, si deve visualizzare un messaggio su stdout (quanti file sono presenti nella directory)
- il file comandi deve poter essere invocato con *uno e un solo parametro*, la directory da porre sotto osservazione (fare opportuno controllo dei parametri)

Suggerimento: uso di un file temporaneo, in cui tenere traccia del numero di file presenti al controllo precedente

---

# Esempi di file comandi

```
#! /bin/bash
echo 1 > loop.$$.tmp
OK=0
while [ $OK = 0 ]
do
    sleep 5s
    new=`ls $1|wc -w`
    old=`cat loop.$$.tmp`
    if [ $new -ne $old ]
    then
        echo $new > loop.$$.tmp
        echo in $1 sono presenti `cat loop.$
        .tmp` file
    fi
done
```

# Esempi di file comandi (1)

```
echo `pwd` > "f1"
# R: crea il file di nome f1, poi stdout= f1>; echo `pwd`  
# 1:      echo /usr/bin
# 2:      nessuna operazione ulteriore di parsing
# 3:      nessuna operazione ulteriore di parsing
```

---

```
test -f `pwd`/$2 -a -d "$HOME/dir?"
# R:      nessuna operazione di parsing
# 1:      test -f /temp/$2 -a -d "$HOME/dir?"
# 2:      test -f /temp/pluto -a -d "/home/staff/
#                  pbellavis/dir?"
# 3:      nessuna operazione ulteriore di parsing
#                  test -f /temp/pluto -a -d /home/staff/
#                  pbellavis/dir?
```

---

# ESERCIZIO PROPOSTO

- Realizzare un file comandi che preveda la seguente sintassi:

**Comando D S**

dove D è una directory, e S una stringa.

- Il file comandi deve individuare nel directory D il file che contiene il maggior numero di occorrenze della stringa S.