

# Ambienti Operativi

SUPSI Dipartimento Tecnologie Innovative

Gianni Grasso

26 novembre 2024

**Classe:** I1B

**Anno scolastico:** 2024/2025

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Tipi di file . . . . .	3
1.2	SSH . . . . .	3
1.3	Struttura di un filesystem . . . . .	3
<b>2</b>	<b>Bash</b>	<b>4</b>
2.1	Comandi . . . . .	4
2.2	Globbering . . . . .	7
2.3	Redirezione . . . . .	7
2.3.1	Pipe . . . . .	7
2.4	Esecuzione sequenziale e condizionale . . . . .	8
2.5	Gestione dei permessi . . . . .	9
2.5.1	Cambiare i permessi . . . . .	9
2.6	Elaborazione parallela . . . . .	10
2.6.1	Approccio manuale . . . . .	10
2.6.2	GNU Parallel . . . . .	10
<b>3</b>	<b>Scripting</b>	<b>11</b>
3.1	Primi passi . . . . .	11
3.2	Variabili . . . . .	11
3.2.1	Variabili predefinite . . . . .	11
3.3	Shift . . . . .	12
3.4	Read . . . . .	12
3.5	Catturare l'output di un comando . . . . .	12
3.6	Test . . . . .	13
3.6.1	Esempio . . . . .	13
3.7	Funzioni . . . . .	14
3.7.1	Esempio . . . . .	14
3.8	Condizioni . . . . .	14
3.8.1	Con <code>test</code> . . . . .	14
3.9	Cicli . . . . .	15
3.9.1	<code>while</code> . . . . .	15
3.9.2	<code>break</code> e <code>continue</code> . . . . .	15
3.9.3	<code>for</code> . . . . .	15
3.10	<code>switch</code> . . . . .	16
3.11	Here document . . . . .	16
3.12	Elaborazione di stringhe . . . . .	17
3.12.1	Esempi . . . . .	17
<b>4</b>	<b>Espressioni regolari</b>	<b>18</b>
4.1	Pattern semplici . . . . .	18
4.2	Caratteri speciali . . . . .	18
4.2.1	Alternative . . . . .	18
4.3	Esempi . . . . .	19

# 1 Introduzione

Per riuscire a capire cos'è un ambiente operativo, dobbiamo innanzitutto riuscire a vedere il computer come uno strumento di elaborazione dei dati alla quale vengono passati degli input e date delle istruzioni. L'ambiente operativo è il luogo nella quale vengono date queste istruzioni.

È importante non fare confusione e non confondere il sistema operativo con l'ambiente operativo, il primo ha lo scopo di nascondere i meccanismi di gestione della macchina, rendendo l'utente in grado di poterla utilizzare senza conoscerne il funzionamento a basso livello, l'ambiente operativo invece fa da tramite tra gli utenti ed il sistema operativo e può essere visto come l'interfaccia nella quale si danno istruzioni alla macchina.

## 1.1 Tipi di file

Tutti i dati che si trovano su un computer sono rappresentati da una sequenza binaria, con **file binari** intendiamo che i dati non sono direttamente comprensibili da una persona mentre per con il termine **file personali** si intendono i file che possono essere compresi da una persona sotto forma di testo.

Per codificare i dati testuali si associa ogni carattere a una sequenza binaria, non esiste però un'unica codifica dei caratteri, di seguito sono riportati alcuni esempi:

- ASCII
- Windows code pages
- ISO 8859
- Unicode
- ...

Dobbiamo poi fare distinzione tra documenti testuali semplici e documenti strutturati, i primi sono quei documenti in cui la struttura logica non è facilmente distinguibile mentre nel secondo caso parliamo di documenti di testo in cui c'è una struttura che stabilisce il contenuto del file (ad esempio i file **csv**).

## 1.2 SSH

SSH, ovvero Secure Shell, è un protocollo di rete crittografico che ci consente di utilizzare servizi di rete in modo sicuro su una rete non protetta. Le sue applicazioni più comuni sono il login remoto e l'esecuzione da riga di comando, noi useremo il SSH per connetterci ad un server didattico.

Per connetterci al server da macOS/Linux dobbiamo digitare a terminale il seguente comando:

```
ssh linux1-didattica.supsi.ch -l nome.cognome@supsi.ch
```

## 1.3 Struttura di un filesystem

Il filesystem di un sistema ha una struttura ad albero, la radice è la cartella **root** (**/**), tutte le altre directory sono sottostanti ad essa.

Tra le directory principali che ci interessano ci sono **/home**, cartella che contiene le directory degli utenti locali, e **/tmp** cartella nella quale risiedono i file temporanei.

Un percorso può essere assoluto o relativo, nel primo caso specifichiamo l'intero percorso di una directory o un file, indipendentemente da dove ci troviamo, mentre nel secondo indichiamo il percorso per raggiungere un file a partire dalla posizione corrente.

## 2 Bash

La prima volta che apriremo un terminale potremo notare che il cursore è preceduto da: `utente@host:path$`, dove utente sta per il nome dell'utente connesso alla macchina, host il nome del server e path il percorso corrente.

È importante ricordare che in bash è tutto **case-sensitive**, sia i comandi che i nomi dei file e delle directory.

### 2.1 Comandi

Ecco una lista di comandi utili visti durante il corso:

- **history**

Stampa la cronologia dei comandi

```
history
```

`ctrl + r` per cercare

- **touch**

Aggiorna la data dell'ultima modifica del file, se il file non esiste ne crea uno

```
touch filename
```

- **rm**

Cancella un file o una directory

```
rm filename #cancella un file
```

```
rmdir directory #cancella una directory vuota
```

```
rm -r #cancella una directory e tutti i file al suo  
interno
```

- **mv e cp**

Rinomina, copia o sposta un file

```
mv filename newFilename #rinomina un file
```

```
cp filename copyFilename #copia un file
```

```
cp -r directory copyDirectory #copia una cartella
```

```
mv filename directory #sposta un file
```

- **cat**

Visualizza il contenuto di un file

```
cat filename
```

- **seq**

Stampa una sequenza di numeri

```
# inizia da 1 e finisce a 100 (incrementa di 1 di default)
```

```
.
```

```
seq 1 100 > file.txt
```

```
cat file.txt
```

```
# 1
```

```
# 2
```

```
# ...
```

```
# 100
```

- **find**

Cerca dei file secondo i criteri imposti. Argomenti comuni

- **-name**: il nome dei file
- **-iname**: il nome dei file (non case sensitive)
- **-user**: il proprietario dei file
- **-size**: la dimensione dei file
- **-exec**: esegue un comando per ogni file trovato. **-exec rm {} \;**

```
# trova i file che finiscono con .jpg e li copia nella
# cartella /destinazione
find / -name '*.jpg' -exec copy {} /destination \;
```

- **head**

Stampa le prime  $n$  righe di un file

```
head -n 50 filename #stampa le prime 50 righe del file

cat filename | head -n 1 #stampa la prima riga del file

cat filename | head -n -3 #stampa tutto il contenuto del
# file tranne le prime 3 righe
```

- **tail**

Stampa le ultime  $n$  righe di un file

```
tail -n 50 filename #stampa le ultime 50 righe del file

cat filename | tail -n 1 #stampa l'ultima riga del file
```

- **wc**

Conta le parole passate come input

```
# stampa il numero di linee di file.txt
cat file.txt | wc -l
```

- **tr**

Sostituisce una parola con un'altra

```
# sostituisce tutte le lettere `a` con `b`
cat file.txt | tr a b
```

- **cut**

Estrae porzioni di testo, utilizzando delimitatori o posizioni specifiche.

```
# estrae il secondo campo di testo dividendo il file a
# ogni `;`
cat file.txt | cut -d ";" -f2
```

- **grep**

Filtra le righe di un file

```
# Visualizza le righe di file.txt che contengono la
# parola `ciao`
grep ciao file.txt
```

- **tee**

Redirige l'output

```
find / -name "z*" 2>&1 | tee filename #cerca dei file e
# redirige l'output sia nel file che sullo schermo
```

- **sort**

Ordina l'output di un comando o visualizza il contenuto ordinato di un file

```
cat filename | sort #ordine alfabetico
```

```
cat filename | sort -r #ordine alfabetico al contrario
```

- **uniq**

Stampa a schermo le righe senza duplicati (se le righe sono adiacenti)

```
cat filename | sort | uniq
```

- **groups**

Visualizza a quali gruppi appartengo

```
groups
```

- **newgrp**

Cambio il mio gruppo corrente

```
newgrp adm
```

- **chmod**

Cambia i permessi di un file

```
chmod u=rw,g=w,o=- filename
```

```
chmod ugo=rw filename
```

- **chown e chgrp**

Cambia gruppo o proprietario di un file

```
chown user filename
```

```
chgrp group filename
```

- **parallel**

Esegue processi in parallelo

```
cat filename | parallel rm {} #elimina ogni file che ha  
il nome uguale ad una riga del file "filename"
```

```
cat filename | parallel -n 2 echo "Riga" #-n 2 prende due  
righe alla volta
```

```
cat filename | parallel --pipe -n2 wc -l
```

- **expr**

Fa la somma di due argomenti

```
expr 10 + 20
```

```
expr 10 '*' 20 #per evitare il globbing
```

## 2.2 Globbing

Il Globbing consiste nell'utilizzare un pattern con uno o più caratteri "wildcard" per trovare o fare azioni sui file.

- `*`, corrispondenza con zero o più caratteri qualsiasi
- `?`, corrispondenza con esattamente un carattere
- `[a-zA-Z9-0]`, corrispondenza tra un gruppo di caratteri
- `[^abc]`, non-corrispondenza tra un gruppo di caratteri

Ad esempio per trovare tutti i file che iniziano con un numero e hanno un'estensione di tre caratteri:

```
find [0-9]*.???
```

Facciamo un esempio più complicato, il seguente codice trova tutti i file che hanno un nome che inizia con un numero compreso tra 10 e 20.

```
ls 1[0-9][^0-9]* 20[^0-9]*
```

Notiamo innanzitutto che il comando `ls` prende due intervalli, il primo trova i file che iniziano con il carattere 1 e che hanno come secondo carattere un numero da 0 a 9, il terzo carattere però non può essere un altro numero, in questo modo troviamo tutti i file che iniziano con un numero compreso tra 10 e 19. Il secondo intervallo invece trova i file che iniziano con il numero 20 (il terzo carattere) non può essere un numero.

## 2.3 Redirezione

Sui sistemi Unix i programmi hanno accesso a tre stream di input e output.

- `stdin [0]`, input dalla tastiera

```
> cat > greetings.txt
```

- `stdout [1]`, output su schermo

```
cat file > /dev/null
```

- `stderr [2]`, output degli errori su schermo

```
cat nonexistentfile 2> errors.txt
```

**Nota:** Di default `1>` non include gli errori e sovrascrive il file di destinazione. Per fare l'append usare `>>`.

È anche possibile redirezionare `stdout` e `stderr` insieme, tuttavia non è buona pratica farlo in un file come nell'esempio sottostante:

```
cat file nonexistent >& results.txt
```

### 2.3.1 Pipe

La pipe è un buffer che permette di passare lo `stdout` di un comando come `stdin` di un altro, da sinistra a destra. Il seguente comando ad esempio prende l'input del comando `cat` e lo passa al comando `grep`:

```
cat A.txt | grep "ABC"
```

È possibile redirezionare lo `stderr` **nello** `stdout`, in questo caso possiamo usare `'2>&1'`:

```
ls inesistente 2>&1 >/dev/null | grep impossibile
```

## 2.4 Esecuzione sequenziale e condizionale

È possibile eseguire sequenzialmente una serie di comandi inserendoli in un'unica linea di comando, ogni istruzione deve essere separata da un `;`. Ad esempio:

```
utente@host:~$ echo "ciao"; echo "mondo"; pwd
ciao
mondo
/home/utente
```

È anche possibile eseguire più comandi inserendoli sulla stessa linea di comando legandoli insieme con un'operazione booleana, `OR` `||` oppure `AND` `&&`. Ad esempio:

```
cat miofile || date || ls
```

L'esecuzione termina quando uno dei comandi (eseguiti da sinistra verso destra) termina correttamente senza errori.

```
cat miofile && date && ls
```

L'esecuzione termina quando uno dei comandi (eseguiti da sinistra verso destra) produce un errore.



## 2.5 Gestione dei permessi

Linux è un sistema operativo **multiutente**. Per autenticarsi c'è bisogno di un nome utente (che è univoco) e una password. La lista di tutti gli utenti è disponibile nel file `/etc/passwd`. L'utente `root` ha tutti i privilegi di sistema.

Gli utenti possono essere organizzati in gruppi, la lista di tutti i gruppi è disponibile nel file `/etc/groups`, di base ogni utente viene anche inserito in un gruppo con il suo nome. Ovviamente un utente può appartenere a più gruppi ma si è attivi in un unico gruppo alla volta.

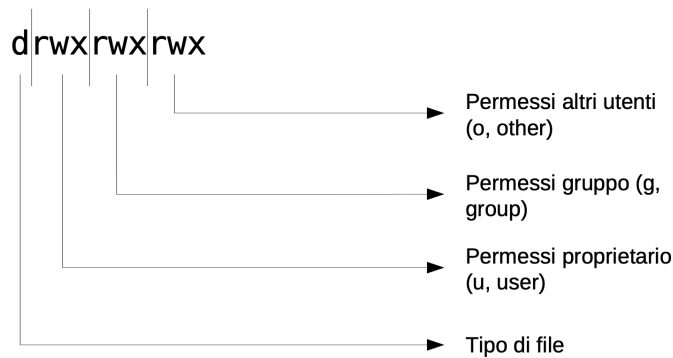


Figura 1: Struttura dei permessi

**Nota:** ci sono diversi tipi di file ma generalmente ci sarà `d` (directory) o `-` (file regolare).

Se l'immagine di prima rappresentava la struttura dei permessi, ora cerchiamo di descrivere i permessi di base e come si comportano su file e directory:

- **Lettura (r)**
  - *File:* permette di leggerne il contenuto
  - *Directory:* permette di elencarne il contenuto (file, sotto-directory)
- **Scrittura (w)**
  - *File:* permette di modificarne il contenuto
  - *Directory:* permette di aggiungere o rimuovere contenuto
    - \* posso cancellare un file solo se ho permessi sulla directory che lo contiene, non basta il permesso di scrittura sul file
- **esecuzione (x)**
  - *File:* permette di eseguirli (su un file `.txt` ad esempio è inutile)
  - *Directory:* permette di attraversarle per accedere a file e sotto-directory, non permette di elencare il contenuto

### 2.5.1 Cambiare i permessi

Ci sono due modi per cambiare i permessi:

- Modalità simbolica, `chmod [-R] [ugo] [+ -=] [rwxst] files`
- Modalità ottale, `chmod [-R] [N] [N] [N] [N] files`

**Nota:** `ugo` indica (user, group, others)

**Nota:** nella modalità ottale i permessi vengono rappresentati con dei bit, in generale `r` vale 4, `w` 2 e `x` 1

È inoltre possibile cambiare il proprietario di un file con `chown` e cambiare il gruppo di un file con `chgrp`

## 2.6 Elaborazione parallela

Per visualizzare ed elencare i processi presenti sul sistema possiamo usare:

- `ps`
- `top`
- `htop`

Le informazioni sulla cpu a disposizione sono presenti nel file `/proc/cpuinfo`.

Per vedere le informazioni relative al tempo di un comando possiamo usare il comando `time` prima dell'effettivo comando.

### 2.6.1 Approccio manuale

Con il comando `split` possiamo dividere un file in  $n$  file in base al numero di linee che esso contiene. Ad esempio:

```
$ wc -l file.tsv
9804311 file.tsv #Il numero di righe del file

$ split -l 4902155 file.tsv #Separa il file ogni 4902155 righe (
    la metà), verranno creati due file
```

### 2.6.2 GNU Parallel

Il comando `parallel` ci permette di eseguire un comando per ogni riga di un file e lo fa in modo parallelo. Funziona in modo simile al comando `exec` con `find`.

Il comando `parallel` riceve lo `stdin` e lo passa al comando successivo, è utile con comandi che funzionano con lo `stdin`, come ad esempio `rm` o `touch`. Facciamo qualche esempio:

```
cat lista | parallel rm {}

cat lista | parallel echo {} "Riga"
```

Si può anche specificare quante righe alla volta usare come parametro, ad esempio:

```
cat lista | parallel -n 2 echo "Ciao {1} mondo {2}"
```

Infine, possiamo usare il comando `--pipe` per passare le righe tramite pipe. Questo è utile solo quando vogliamo eseguire operazioni con più righe alla volta:

```
cat lista | parallel --pipe -n2 wc -l
```

## 3 Scripting

Lo scripting in **bash** è utile per automatizzare compiti ripetitivi, combinando i comandi che abbiamo già visto e che da soli non sempre possono fare quello che ci serve.

Uno script è un file di testo che contiene dei comandi shell. Un file di questo tipo ha estensione **.sh** ma può anche essere omessa.

**Nota:** Bash è un linguaggio interpretato. In caso di errori quindi lo script andrà avanti.

### 3.1 Primi passi

```
#!/bin/bash

# Questo è un commento, e inizia con il cancelletto
echo "Ciao "mondo"
```

- La prima linea di codice specifica che si sta utilizzando **bash**
- I commenti si fanno utilizzando il carattere **#**
- Uno script può essere eseguito in due modi
  - **bash script.sh**
  - **./script.sh** (solo con permessi di esecuzione)
  - **script** (solo se lo script è nella directory **/bin**, accessibile solo da amministratore)

### 3.2 Variabili

Le variabili possono essere solo **interi**, **stringhe**, o **array**, **bash** non supporta i numeri decimali.

```
#!/bin/bash
i=4
x="ciao mondo"
echo $x
```

**Nota:** non ci sono spazi intorno a **=**.

**Nota:** il carattere **\$** espande la variabile e ne restituisce il valore, questa operazione ha la precedenza sul globbing. Per utilizzare invece il carattere **'\$'** si usa **\\$** o si mette la stringa tra singoli apici.

#### 3.2.1 Variabili predefinite

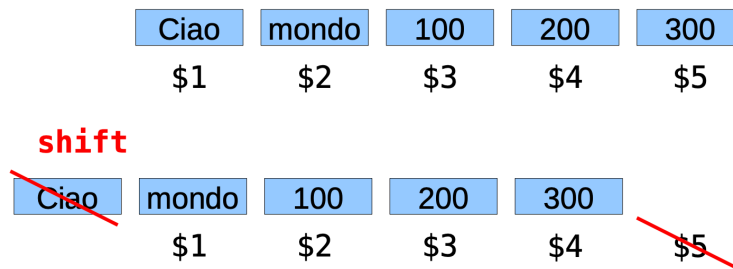
- **\$0**, il nome dello script
- **\$1 .. \$n**, i parametri (per parametri **> 10** usare **\${nn}**)
- **\$#**, il numero dei parametri
- **\$@**, tutti i parametri (è un array)
- **\$\$**, il PID del processo della shell corrente
- **\$\_**, il PID dell'ultimo processo mandato in background
- **\$\_**, il valore di ritorno dell'ultimo comando eseguito (**0 = true**, altrimenti ha dato errore)

**Nota:** le variabili predefinite si possono sovrascrivere assegnando un altro valore.

**Nota:** con il comando **exit** si esce dallo script, possiamo specificare se il programma è andato a buon fine usando ad esempio **exit 0** per la buona riuscita e **exit 1** per un'uscita che ha generato errori.

### 3.3 Shift

Il comando `shift` sposta i valori dei parametri posizionali



### 3.4 Read

Il comando `read` legge l'input dell'utente e lo memorizza in una variabile

```
#!/bin/bash

#Va a capo
echo "Ciao come ti chiami?"
read nome
echo "Ciao $nome!"

#Resta sulla riga corrente
read -p "Come ti chiami"? nome
echo "Ciao $nome!"

#Resta sulla riga corrente e non mostra quello che l'utente
#inserisce da tastiera
read -s -p "Password"? segreto
echo "Conosco il tuo $segreto!"
```

### 3.5 Catturare l'output di un comando

Con `$()` posso salvare l'output di un comando in una variabile.

```
#!/bin/bash

risultato=$(ls | grep "A") #Attennzione a non mettere spazi dove
#non servono

echo "risultato: $risultato"
```

### 3.6 Test

Per verificare una condizione si utilizza il comando `test`. Un'espressione vera (true) ritornerà 0 mentre un'espressione falsa ritornerà un valore diverso da zero.

```
#Equal
test 5 -eq 5

#Not equal
[ 5 -ne 6 ] #Sintassi alternativa

#Se la condizione è vera stampa 'OK'
test 5 -eq 5 && echo "OK"
```

<code>-n op</code>	Controlla se la stringa <code>op</code> contiene qualcosa
<code>-z op</code>	Controlla se la stringa <code>op</code> è vuota
<code>-d op</code>	Esiste una directory <code>op</code>
<code>-e op</code>	Esiste il file/directory <code>op</code>
<code>a -eq b</code>	<code>a</code> e <code>b</code> sono numeri interi e sono uguali
<code>a -ne b</code>	Opposto di <code>-eq</code>
<code>a = b</code>	<code>a</code> è uguale a <code>b</code> (come stringa)
<code>a != b</code>	Opposto di <code>=</code>
<code>a -lt b</code>	<code>a &lt; b</code> ( <code>a</code> , <code>b</code> numeri interi)
<code>a -gt b</code>	<code>a &gt; b</code> ( <code>a</code> , <code>b</code> numeri interi)
<code>a -le b</code>	<code>a &lt;= b</code> ( <code>a</code> , <code>b</code> numeri interi)
<code>a -ge b</code>	<code>a &gt;= b</code> ( <code>a</code> , <code>b</code> numeri interi)
<code>a -a b</code>	AND logico
<code>a -o b</code>	OR logico
<code>! a</code>	NOT logico

#### 3.6.1 Esempio

Scriviamo uno script che accetta esattamente due parametri. Se il numero di parametri non è corretto termina l'esecuzione con un codice di errore, altrimenti scrive "OK"

```
( test $# -eq 2 && echo "OK" && exit 0 ) || exit 1

test $# -ne 2 && echo "errore" && exit 1
```

**Nota:** è importante lasciare uno spazio dopo la parentesi tonda.

### 3.7 Funzioni

```
function saluta() {  
    echo "Hello, world"  
}  
  
saluta
```

**Nota:** una funzione si usa con il suo nome senza l'aggiunta di parentesi tonde dopo.

#### 3.7.1 Esempio

```
function somma() {  
    echo $(expr $1 + $2)  
}  
  
somma 10 20
```

oppure utilizzando il `return`

```
function somma() {  
    risultato=$(expr $1 + $2)  
    return $risultato  
}  
  
echo $?
```

**Nota:** in questo caso `$1` e `$2` sono i parametri della funzione.

### 3.8 Condizioni

```
if mkdir nuovacartella 2>/dev/null; then  
    echo "cartella creata"  
elif touch nuovofile 2>/dev/null; then  
    echo "file creato"  
else  
    echo "non posso creare ne cartella ne il file"  
fi
```

**Nota:** il comando `mkdir nuovacartella` restituisce `true` o `false` a dipendenza dell'esito del comando. La condizione può essere un qualsiasi comando bash.

#### 3.8.1 Con test

```
if [ $# -ne 2 ] && [ $1 -eq 5 ]; then  
    echo "Devi passarmi due parametri e il primo deve essere 5"  
    exit 1  
fi  
echo "tutto OK"  
  
test=""  
  
if [ -n "$testo" ]; then  
    echo "la stringa non è vuota"  
fi
```

**Nota:** usare sempre le virgolette quando si usa una variabile di testo.

## 3.9 Cicli

### 3.9.1 while

```
i=1
while [ $i -lt 10 ]; do
    echo "dentro il while, i vale: $i"

    i=$(expr $i + 1)
done
```

Scorrere tutti i parametri (con `shift`)

```
while [ $# -gt 0 ]; do
    echo "parametro: $1"
    shift
done
```

Viene letto il contenuto del file `dati.txt` linea per linea

```
while read line; do
    echo "linea: $line"
done < dati.txt
```

### 3.9.2 break e continue

```
while [ $# -gt 0 ]; do
    echo $1
    if [ "$1" = "x" ]; then
        break;
    fi
    shift
done
```

### 3.9.3 for

Scorre tutti i parametri

```
for i in $@; do
    echo $i
done
```

Scorre il contenuto di tutti file in base a un pattern

```
for file in $(ls /tmp); do
    while read linea; do
        echo "linea: $linea"
    done < $file
done
```

### 3.10 switch

```
for $file in *.txt; do
    case $file in
        a*) echo "inizia con a" ;;

        b*) echo "inizia con b" ;;

        *t*) echo "contiene una t" ;;

        *) echo "caso di default" ;;
    esac
done
```

### 3.11 Here document

- Redirige una stringa come input (stdin) di un comando con un <<TAG
- La forma <<"TAG" evita l'espansione di variabili all'interno della stringa

```
if [ ! -e $1.java ]; then
    cat <<FINE > $1.java
class $1 {
    public $1() {

        public static void main (String args[]) {
            new $1();
        }
    }
}
FINE
    echo "File $1.java creato"
else
    echo "Errore: il file $1.java esiste già" 1>&2
    exit 1
fi
```



### 3.12 Elaborazione di stringhe

- Usato per modificare delle stringhe secondo alcuni criteri
- Supporta il globbing

<code>\${var#pattern}</code>	Se pattern coincide con l'inizio di var rimuovi la parte più corta che coincide
<code>\${var##pattern}</code>	Se pattern coincide con l'inizio di var, elimina la parte più lunga che coincide
<code>\${var%pattern}</code>	Se pattern coincide con la fine di var, elimina la più corta parte che coincide
<code>\${var%%pattern}</code>	Se pattern coincide con la fine di var, elimina la più lunga parte che coincide
<code>\${var/pattern/str}</code>	La parte più lunga di var che coincide con pattern è sostituita da str
<code>\${var//pattern/str}</code>	Tutte le volte che pattern coincide con una parte di var, sostituisci con str
<code>\${var:offset:l}</code>	Ritorna la sottostringa della lunghezza l specificata di var a partire dall'offset. Se lunghezza non è specificata, continua fino alla fine, valori negativi terminano al n-esimo carattere
<code>\${#var}</code>	Ritorna la lunghezza della stringa

#### 3.12.1 Esempi

```
testo="Ambienti Operativi"
echo ${testo#Amb}
#stampa: ienti Operativi

testo2="Ambienti Operativi"
echo ${file%???}
#stampa: Ambienti Operat

file="miofile.jpg"
echo ${file//.jpg/.png}
#stampa: miofile
```

## 4 Espressioni regolari

Le espressioni regolari permettono di cercare delle sequenze di caratteri all'interno di stringhe di testo. Fino ad ora abbiamo visto ad esempio il comando `grep`, tuttavia è piuttosto limitato. Per testare utilizzare il sito [regex101](https://regex101.com).

### 4.1 Pattern semplici

- **Singoli caratteri:** `b`, cerca il carattere `b` all'interno del testo
- **Sequenze di più caratteri:** `, mani` cerca la sequenza `mani` all'interno del testo (`mani`, `romani`, `umani`, ...)

### 4.2 Caratteri speciali

Alcuni caratteri assumono un significato speciale se sono preceduti da `\`:

<code>\b</code>	Delimitatore di una parola
<code>\B</code>	Inverso di <code>\b</code> (carattere che non è un delimitatore di parola)
<code>\d</code>	Cifra
<code>\D</code>	Inverso di <code>\d</code> (carattere che non è una cifra)
<code>\s</code>	Spazio
<code>\S</code>	Inverso di <code>\s</code> (tutto tranne uno spazio)
<code>\w</code>	Carattere di una parola
<code>\W</code>	Inverso di <code>\w</code>
<code>\A</code>	Inizio del testo (!)
<code>\Z</code>	Fine del testo (!)

Altri sono speciali senza l'utilizzo del `\`:

<code>^</code>	Inizio della stringa (o della riga) *
<code>\$</code>	Fine della stringa (o della riga) *
<code>*</code>	Zero o più (quantificatore)
<code>+</code>	Uno o più (quantificatore)
<code>?</code>	Zero o uno (quantificatore)
<code>.</code>	Qualsiasi carattere tranne ritorno a capo
<code>()</code>	Delimitatori di gruppo
<code>{ }</code>	Delimitatori per il numero di ripetizioni
<code>[ ]</code>	Insiemi di caratteri
<code> </code>	Alternativa

**Nota:** per trovare il carattere effettivo bisogna mettere prima `\`, ad esempio `\*`.

#### 4.2.1 Alternative

Trova `ciao` oppure `cibo`, inoltre crea dei sottomatch per `ia` e `ib`.

```
c(ia | ib)o
```

### 4.3 Esempi

Di base per cercare con le `regex` usiamo `grep -E "regex" a.txt`.

- Trovare tutte le parole di un testo potremmo usare

```
\b[A-Za-z]+\b
```

*#oppure*

```
\b[[:alpha:]]+\b
```

- Trovare tutte le parole palindrome di quattro lettere

```
\b(\w)(\w)\2\1\b
```