

Laboratorio di Informatica di Base

Progetto Tandem 2007/2008

Docente: Carlo Drioli

Lucidi a cura di

Andrea Colombari,
(colombari@sci.univr.it)

Carlo Drioli
(drioli@sci.univr.it)

e Barbara Oliboni
(oliboni@sci.univr.it)

Procedure (script)

Testo di riferimento:
V. Manca
“Metodi Informazionali”
Bollati Boringhieri

Procedure shell (shell script)

- Vengono usate nei programmi che interagiscono con il sistema operativo
 - Esempio: per semplificare le operazioni di installazione e /o configurazione di pacchetti software
- Il linguaggio shell comprende:
 - variabili locali e d'ambiente
 - operazioni di lettura/scrittura
 - strutture per il controllo del flusso di esecuzione: sequenziale, decisionale e iterativa
 - richiamo di funzioni con passaggio di parametri

Creare una procedura (script)

- Una **procedura o script**, non è altro che un file di testo contenente una serie di istruzioni e comandi da far interpretare/eseguire alla shell.
- I seguenti passi sono necessari per creare ed eseguire uno script:
 - Elaborare lo script all'interno di un file di testo, che chiamiamo `MIO_SCRIPT`, mediante un elaboratore di testi (es: `joe MIO_SCRIPT`)
 - Una volta creato lo script, settare i permessi per la sua esecuzione (es: `chmod +x MIO_SCRIPT`)
 - Far interpretare lo script alla shell. Per fare ciò, supposto che il nome del file contenente lo script sia **script**, si scrive:

```
$ ./MIO_SCRIPT  
$
```

Creare una procedura (script) (2)

- Il “./” davanti al nome serve per specificare il fatto che lo script si trova all’interno della cartella corrente.
- Se la directory corrente è nella variabile d’ambiente PATH, allora per eseguire lo script possiamo scrivere semplicemente:

```
$ MIO_SCRIPT  
$
```

Esempio di script

- Come primo esempio, vediamo uno script per scrivere sul terminale video la scritta “Ciao Mondo” avendo cura precedentemente di ripulire lo schermo. Creiamolo con `cat`:

```
$ cat > MIO_SCRIPT  
clear  
echo "Ciao Mondo"
```

- A questo punto premendo CTRL+C si interrompe il `cat` e quanto inserito sullo standard input verrà trasferito sul file `MIO_SCRIPT`.

```
Andrea@nyx ~  
$ cat MIO_SCRIPT  
clear  
echo "Ciao Mondo"  
Andrea@nyx ~  
$_
```

Esempio di script (2)

- Ora facendo `ls -l` si può notare che i permessi di esecuzione mancano:

```
$ ls -l
totale 1
-rw-r--r-- 1 Andrea Nessuno 24 Sep 24 14:34 MIO_SCRIPT
```

- E' quindi necessario cambiare i permessi per poter eseguire lo script. Usiamo allora `chmod` per aggiungere il permesso di esecuzione e rifacciamo `ls -l`, si vedrà:

```
Andrea@nyx ~
$ chmod +x MIO_SCRIPT

Andrea@nyx ~
$ ls -l
totale 1
-rwxr-xr-x 1 Andrea Nessuno 24 Sep 24 14:34 MIO_SCRIPT
```

Esempio di script (3)

- A questo punto è possibile eseguire lo script:

```
Andrea@nyx ~  
$ ./MIO_SCRIPT
```



Quello che accade è che scompare tutto il testo a video e poi compare la scritta in alto a sinistra, e subito dopo il prompt:

```
Ciao Mondo  
Andrea@nyx ~  
$ _
```


Script per inizializzare l'ambiente

- L'utente può personalizzare le operazioni di inizializzazione dell'ambiente effettuate dal sistema ad ogni connessione.
- Ad ogni connessione la shell esegue una procedura.
 - La **bash** controlla nella home dell'utente la presenza del file **.bash_profile** e:
 - se esiste ne esegue tutte le righe
 - se non esiste la **bash** cerca nella home dell'utente il file **.profile** e se esiste ne esegue tutte le righe
- Quindi, personalizzando tale script è possibile personalizzare il proprio ambiente di lavoro.

Script per inizializzare l'ambiente (2)

- Un esempio di `.bash_profile`

```
# aggiunge al PATH la directory /etc e la directory
# bin contenuta nella propria home
PATH=$PATH:/etc:$HOME/bin

# crea la variabile MAIL, o se esiste la rimpiazza,
# inserendovi la directory mail presente nella propria home
MAIL=$HOME/mail

# imposta il prompt personalizzato con il nome utente
PS1=${LOGNAME}"> "

# directory contenente le mie lettere
MIE_LETTERE=$HOME/lettere
export MIE_LETTERE

# creazione di un alias del comando rm in modo che venga
# eseguito sempre con l'opzione -i
alias rm="rm -i"
```

Script per inizializzare l'ambiente (3)

- Si noti che il carattere **#** è utilizzato per inserire dei **commenti**, cioè testo che non viene interpretato dalla shell ma che serve al programmatore per rendere più chiaro il proprio codice.
- Il **prompt** può essere personalizzato grazie all'impostazione della variabile **PS1**. Alcuni pattern utilizzabili:
 - **"\u"**: visualizza il nome utente.
 - **"\h"**: visualizza il nome della macchina (hostname).
 - **"\w"**: visualizza il percorso di dove vi trovate.
 - In generale è possibile usare l'output di un qualsiasi comando con la notazione **"\$(comando [opzioni] [argomenti])"**. Per esempio, è possibile inserire l'ora, i minuti e i secondi sfruttando il comando **date**:
"\$(date +%H:%M:%S) "

Il comando **read**

- Il comando **read** legge una riga da standard input fino al ritorno a capo e assegna ogni parola della linea alla corrispondente variabile passata come argomento

```
$ read a b c
111 222 333 444 555
$ echo $a
111
$ echo $b
222
$ echo $c
333 444 555
$
```

- Il carattere separatore è contenuto nella variabile **IFS** che per default contiene lo spazio

Uso di **read** in una procedura

■ Esempio:

File **prova_read**

```
echo "dammi il valore di x"
read x
echo "dammi il valore di y"
read y
echo "x ha valore" $x
echo "y ha valore" $y
```

```
$ sh prova_read
dammi il valore di x
15
dammi il valore di y
ottobre
x ha valore 15
y ha valore ottobre
$
```

Uso degli apici

- Una stringa racchiusa tra apici singoli non subisce espansione

```
$ echo '*$HOME*'
*$HOME*
```

- Una stringa racchiusa tra apici doppi subisce l'espansione delle sole variabili

```
$ echo "$*$HOME*"
*/home/pippo*
```

- Una stringa racchiusa tra apici singoli rovesciati viene interpretata come comando

```
$ lista=`ls -la`
$ echo $lista
Cartella1 Cartella2 file1.txt ...
```

Uso degli apici (2)

- Un apice singolo o doppio può essere racchiuso tra apici sole se preceduto dal carattere \

```
$ echo 'Oggi e\' una bella giornata'  
Oggi e' una bella giornata  
$ echo "Il linguaggio \"C\""  
Il linguaggio "C"  
$
```

- Un apice può essere passato come argomento di un comando sole se preceduto dal carattere \

```
$ echo \  
'  
$
```

Parametri posizionali

- Valori passati alle procedure come argomenti sulla riga di comando
- Gli argomenti devono seguire il nome della procedura ed essere separati da almeno uno spazio
- **Esempio:** File **posizionali**

```
echo nome della procedura "$0"  
echo numero di parametri "$#"   
echo parametri "[" $1 $2 $3 $4 $5 "]"
```

```
$ ./posizionali uno due tre  
nome della procedura [./posizionali]  
numero di parametri [3]  
parametri [ uno due tre ]  
$
```


Variabili `$*` e `$@`

- La variabile `$*` contiene una stringa con tutti i valori dei parametri posizionali
- La variabile `$@` contiene tante stringhe quanti sono i valori dei parametri posizionali
- Esempio: File `argomenti`

```
./posizionali "$*"
./posizionali "$@"
```

```
$ ./argomenti uno due tre
nome della procedura [./posizionali]
numero di parametri [1]
parametri [ uno due tre ]
nome della procedura [./posizionali]
numero di parametri [3]
parametri [ uno due tre ]
$
```

Calcoli

- La bash consente di valutare espressioni aritmetiche
- Le espressioni vengono considerate come se fossero racchiuse tra doppi apici, quindi le variabili vengono espansive prima dell'esecuzione dei calcoli
- Il risultato viene tornato come stringa
- Formati ammessi:

```
$((espressione_aritmetica))
```

```
$(espressione_aritmetica)
```

```
$(expr espressione_aritmetica)
```

```
`expr espressione_aritmetica`
```

- Esempio:

```
$ b=7
$ echo $(( $b * 3 ))
21
$
```

Calcoli (2)

- In realtà **expr** è un comando che prende in input una espressione e restituisce il risultato del computo su standard output. Per usarlo in uno script bisogna che sia eseguito incapsulandolo all'interno di `$ ()` oppure `` ``
- Esempi di **expr** come comando da linea:

```
$ expr 1 + 3          # <- addizione
$ expr 2 - 1          # <- sottrazione
$ expr 10 / 2          # <- divisione
$ expr 20 \% 3         # <- modulo, ovvero il resto
$ expr 10 \* 3         # <- moltiplicazione
```

- Esempi di **expr** il cui risultato è usato da un altro comando:

```
$ echo `expr 6 + 3`   # <- stampa il risultato
$ echo $(expr 6 + 3)  # <- stampa il risultato
```

Codice di uscita di un comando

- Numero intero positivo compreso tra 0 e 255
 - Il codice di uscita è 0 se il comando svolge correttamente i propri compiti
 - Il codice di uscita è diverso da 0 altrimenti
- Il codice di uscita dell'ultimo comando lanciato dalla shell viene memorizzato nella variabile speciale **\$?**
- Esempio:

```
$ ls -l frase
-rw-r--r-- 1 pippo stud 332 Feb 23 17:40 frase
$ echo $?
0
$ ls -l canzone
ls: canzone: No such file o directory
$ echo $?
1
```

Lista di comandi

- Gruppo di comandi che la shell esegue in sequenza
- Connessione di comandi **incondizionata**
 - Tutti i comandi della lista vengono sempre eseguiti (a meno della terminazione della procedura)
 - Comandi su righe differenti o separati da ;

```
comando1; comando2; ...
```

- Connessione di comandi **condizionata**
 - Operatori && e ||

```
comando1 && comando2  
comando1 || comando2
```

Operatori **&&** e **||**

- **comando1 && comando2**

comando2 viene eseguito se e solo se **comando1** restituisce un codice di uscita **uguale** a 0

- **comando1 || comando2**

comando2 viene eseguito se e solo se **comando1** restituisce un codice di uscita **diverso** da 0

- Supponedo esista il file di testo **frase** contenente la frase “Il sole splende”, accade che:

```
$ grep sole frase && echo " -->frase contiene 'sole'"
Il sole splende.
-->frase contiene 'sole'
$ grep luna frase || echo " -->frase non contiene 'luna'"
-->frase non contiene 'luna'
$
```

Operatori && e || (2)

- La shell scandisce sempre tutti i comandi, ma condiziona l'esecuzione verificando il codice di uscita
- Esempio:

```
$ grep luna frase &&  
> echo " -->frase contiene 'luna'" ||  
> echo " -->frase non contiene 'luna'"  
-->frase non contiene 'luna'  
$
```

Costrutti del linguaggio di shell

- Le strutture per il controllo del flusso sono di due tipologie:
 - I costrutti di **alternativa** che permettono di fare delle scelte in base a delle condizioni: **if ... then ... fi** (e sue varianti) e **case ... esac**.
 - I costrutti iterativi o **cicli** che permettono di ripetere delle azioni per un certo numero di volte. Tale numero può essere fisso o dipendere da delle condizioni che ne determinano lo stop: **for ... do ... done**, **while ... do ... done** e **until ... do ... done**.

Costrutti del linguaggio di shell:

if ... then ... fi

- Il flusso di esecuzione può essere regolato in base alla valutazione di una CONDIZIONE: se essa si verifica eseguo un qualche comando altrimenti non viene fatto nulla.
- La sintassi è la seguente:

```
if CONDIZIONE
then
    COMANDO
fi
```

- Esempio:

```
# Script per visualizzare un file
cat $1
if [ $? -eq 0 ]
then
    echo $1 ", file trovato e visualizzato"
fi
```

Costrutti del linguaggio di shell:

if ... then ... fi (2)

- Il flusso di esecuzione può essere regolato in base alla valutazione di una CONDIZIONE: se essa si verifica eseguo un qualche comando altrimenti non viene fatto nulla.
- La sintassi è la seguente:

```
if CONDIZIONE
then
    COMANDO
fi
```

Si noti che \$1, \$2, ecc. corrispondono alle stringhe che seguono il nome dello script quando viene eseguito. Mentre \$0 contiene il nome stesso dello script.

- Esempio:

```
# Script per visualizzare un file
cat $1
if [ $? -eq 0 ]
then
    echo $1 ", file trovato e visualizzato"
fi
```

Costrutti del linguaggio di shell:

if ... then ... fi (3)

- Il flusso di esecuzione può essere regolato in base alla valutazione di una CONDIZIONE: se essa si verifica eseguo un qualche comando altrimenti non viene fatto nulla.
- La sintassi è la seguente:

```
if CONDIZIONE
then
    COMANDO
fi
```

Si tratta di un confronto tra il contenuto dell'exit status ? e lo zero. Si sta verificando se sono uguali `-eq`. Altre possibilità sono `-gt (>)`, `-lt (<)`, `-ge (>=)` e `-le (<=)`.

- Esempio:

```
# Script per visualizzare un file
cat $1
if [ $? -eq 0 ]
then
    echo $1 ", file trovato e visualizzato"
fi
```

Costrutti del linguaggio di shell:

if ... then ... else ... fi

- Estensione di **if ... then ... fi**, l'unica differenza è che se la condizione è falsa si esegue ciò che è compreso tra **else** e **fi**
- La sintassi è la seguente:

```
if CONDIZIONE
then
    COMANDO1
else
    COMANDO2
fi
```

- Esempio:

```
# Script per verificare se il primo argomento dello
# script è un numero positivo
if [ $1 -gt 0 ]
then
    echo $1 "positivo"
else
    echo $1 "negativo"
fi
```

Costrutti del linguaggio di shell:

if ... then ... else ... fi (2)

- Esempio: File **if1**

```
if ls $1
then
    echo "il file $1 esiste ..."
    if grep $2 $1
    then
        echo "... e contiene la parola $2!"
    else
        echo "... ma non contiene la parola $2!"
    fi
else
    echo "il file $1 non esiste!"
fi
```

Costrutti del linguaggio di shell:

if ... then ... else ... fi (3)

- **Esempio:** Esecuzione del file **if1**

```
$ if1 frase sole
il file frase esiste ...
Il sole splende.
... e contiene la parola sole!
$ if1 frase luna
il file frase esiste ...
... ma non contiene la parola luna!
$
```

Costrutti del linguaggio di shell:

test o [expr]

- Comando per valutare se una espressione è vera
- La sintassi è la seguente:

`test ESPRESSIONE`

`[ESPRESSIONE]`

- **ESPRESSIONE** è una combinazione di valori, operatori di relazione (-eq, -lt, -gt, ...), oppure operatori matematici (+,-,/,...)
- Con questo costrutto si possono usare argomenti di tipo:
 - Intero
 - File
 - Stringa di caratteri

Costrutti del linguaggio di shell:

test o [expr] (2)

- Esempio: File **prova_test**

```
if test "$1" = si
then
    echo Risposta affermativa
else
    if test "$1" = no
    then
        echo Risposta negativa
    else
        echo Risposta indeterminata
    fi
fi
fi
```

```
$ prova_test si
Risposta affemativa
```


Espressioni logiche su stringhe

- `stringa1 = stringa2`
vero se le stringhe sono uguali
- `stringa1 != stringa2`
vero se le stringhe sono diverse
- `-z stringa1`
vero se `stringa1` ha lunghezza 0
- `[-n] stringa1`
vero se `stringa1` ha lunghezza maggiore di 0

Composizione di espressioni logiche

- Operatori:
 - **-a** mette in AND due espressioni
 - **-o** mette in OR due espressioni
 - **!** nega l'espressione che segue

- Esempio: File **prova_test2**

```
if [ "$1" = si -o "$1" = SI ]
then
    echo Risposta affermativa
else
    if [ "$1" != no -a "$1" != NO ]
    then
        echo Risposta indeterminata
    else
        echo Risposta negativa
    fi
fi
fi
```

Costrutti del linguaggio di shell:

case

- Alternativa a **if ... then ... else ... fi** multi-livello
- Consente di confrontare molti valori con una variabile
- La sintassi è la seguente:

```
case $VARIABILE in
    pattern1) COMANDO
        ...
        COMANDO;;
    pattern2) COMANDO
        ...
        COMANDO;;
    ...
    patternN) COMANDO
        ...
        COMANDO;;
    *)
        COMANDO
        ...
        COMANDO;;
esac
```

Costrutti del linguaggio di shell:

case (2)

- Il contenuto di `VARIABILE` è confrontato con i vari pattern, se nessuno corrisponde allora viene eseguito il caso `*` di **default**.
- Esempio: script che dato un tipo di vettura passato come primo argomento, restituisce il numero di versioni disponibili:

```
tipo $1
case $tipo in
    "auto") echo "Per $tipo disponibili 10 versioni";;
    "van")  echo "Per $tipo disponibili 3 versioni";;
    "jeep") echo "Per $tipo disponibili 2 versioni";;
    *)      echo "Nessuna disponibilit\ 'a"
esac
```

- Esempio di uso:

```
$ ./script jeep
Per jeep disponibili 2 versioni
$
```

Costrutti del linguaggio di shell:

for ... do ... done

- Sintassi:

```
for VARIABLE in LISTA
do
    COMANDO1
    COMANDO2
    ...
    COMANDOn
done
```

- Esegue i comandi 1, ..., n una volta per ogni iterazione
- Per ogni iterazione VARIABLE assumere, uno dopo l'altro, i valori che compaiono in LISTA

Costrutti del linguaggio di shell:

for ... do ... done (2)

- Esempio:

Creiamo lo script **tabella** come segue:

```
n=$1
for i in 1 2 3 4 5 6 7 8 9 10
do
    echo $n "*" $i "=" `expr $i \* $n`
done
```

- Eseguiamo lo script passando il valore 7

```
$ ./tabella 7
```

- Quello che appare a video è quanto segue:

```
7 * 1 = 7
7 * 2 = 14
...
7 * 10 = 70
```

Costrutti del linguaggio di shell:

while ... do ... done

- Sintassi:

```
while CONDIZIONE
do
    COMANDO1
    COMANDO2
    ...
    COMANDOn
done
```

- Esegue i comandi 1, ..., n una volta per ogni iterazione
- All'inizio di ogni iterazione viene valutata la CONDIZIONE: se è vera viene effettuata un'altra iterazione, altrimenti si prosegue con gli eventuali comandi che seguono **done**.

Costrutti del linguaggio di shell:

while ... do ... done (2)

- Esempio:

Lo script `tabella` può essere fatto anche così:

```
n=$1
i=1
while [ $i -le 10 ]
do
    echo $n "*" $i "=" `expr $i \* $n`
    i=`expr $i + 1`
done
```

- L'output di tale script coincide con il precedente, cioè si comporta allo stesso modo.
- Si noti che la mancanza dell'incremento provoca una situazione di **ciclo infinito**, in quanto la condizione sarebbe sempre verificata perché i rimarrebbe sempre pari a 1 e quindi sempre minore o uguale a 10.


```
# procedura ins_agenda
# permette l'inserimento di nome, cognome e telefono
# nel file agenda
RISPOSTA=si
while [ "$RISPOSTA" = si ]
do
    echo "Inserisci il cognome:"
    read COGNOME
    echo "Inserisci il nome:"
    read NOME
    echo "Inserisci il telefono"
    read TELEFONO
    if grep "$COGNOME,$NOME,$TELEFONO" agenda
    then
        echo "Dati gia' inseriti!"
    else
        echo "$COGNOME,$NOME,$TELEFONO">>>agenda
        echo "Inserimento effettuato"
    fi
    echo "Altro nominativo da inserire? (si/no)"
    read RISPOSTA
done
$
```

Costrutti del linguaggio di shell:

until ... do ... done

- Permette di creare cicli condizionati
- Forma:

```
until lista_di_comandi1  
do  
    lista_di_comandi2  
done
```

- I comandi di **lista_di_comandi2** vengono eseguiti fino a quando l'esecuzione dell'ultimo comando in **lista_di_comandi1** restituisce 0