Come funziona Linux:
le strutture di controllo della shell

La shell mette a disposizione del programmatore strutture di controllo comparabili con quelle di altri linguaggi di programmazione ad alto livello, che possono essere utilizzati come un "collante" per mettere assieme la potenza dei comandi di Linux.

Sesta parte

di Giuseppe Zanetti

Spesso si preferisce scrivere uno script nel linguaggio di shell invece che in un linguaggio più ad alto livello perché esso è molto più semplice da realizzare e mantenere. L'idea base di uno shell script è quella di evitare di reinventare ogni volta la ruota, ma di compiere invece le operazioni base richiamando i diversi programmi esterni alla shell. La shell funge pertanto da "collante" fra i diversi comandi/programmi di UNIX, i quali, salvo pochissime eccezioni, non sono costruiti direttamente dentro la shell stessa (builtin), ma vengono ogni volta richiamati da disco ed eseguiti passando loro nella linea di comando i parametri adeguati.

Scrivere uno shell script significa essenzialmente capire come funzionano i (relativamente pochi) comandi di UNIX e metterli assieme in un modo logico al fine di realizzare l'operazione desiderata. Per vedere come questo metodo di lavoro sia conveniente, è sufficiente pensare a quanto codice sarebbe necessario scrivere in linguaggio C solamente per ordinare alfabeticamente le linee di un file. Per compiere la stessa operazione mediante uno script è invece sufficiente richiamare il comando sort con gli opportuni parametri.

La potenza dei comandi UNIX, unita al fatto di non dover compilare lo script ad ogni minima modifica, offre degli innegabili vantaggi in termini di velocità di sviluppo e di manutenzione, che si pagano però in termini di prestazioni: è infatti necessario eseguire la maggior parte dei comandi caricandoli da disco e questa operazione (fork) risulta abbastanza pesante in quanto è necessario avviare un nuovo processo e distruggerlo al termine dell'operazione. In realtà ogni strumento, compresi i linguaggi di programmazione, ha un suo ambito ottimale di utilizzo e nessuno si sognerebbe di scrivere in un linguaggio interpretato un programma che esegua centinaia di calcoli.

Un ulteriore vantaggio della programmazione in shell è quello di poter scrivere ed eseguire il programma anche su macchine su cui non sia installato - perché "macchina di produzione" o semplicemente per le poche risorse disponibili - un sistema di sviluppo per altri linguaggi (ad esempio il compilatore C). Per scrivere uno shell script è sufficiente infatti solamente un qualunque editor di testi. Mi è capitato di scrivere semplici script addirittura utilizzando l'output del comando "echo" ridiretto su un file.

Ultimo, ma non per questo meno importante, motivo che rende interessante lo scrivere uno shell script è la possibilità di utilizzarlo senza o con poche modifiche in qualunque sistema Linux o UNIX. Lo standard POSIX (IEEE Std. 1003.1-1990 Standard for Information Technology, Portable Operating System Interface for uniX systems), al fine di permettere la compatibilità fra sistemi operativi diversi, definisce, fra le altre cose, delle specifiche minime a cui devono uniformarsi i programmi fondamentali. Ciò vale per i programmi UNIX più importanti e, a maggior ragione, per la shell. Su ogni sistema operativo che si dichiari "POSIX compliant", un programma dovrà perciò risiedere in una determinata posizione nel filesystem (esempio: /bin/ls), dovrà accettare nella linea di comando gli stessi parametri e dovrà fornire in output risultati coerenti.

Prima di analizzare le diverse strutture di controllo messe a disposizione negli shell script, vediamo come è possibile per la shell accedere alle informazioni generate dai diversi programmi.

Passaggio di informazioni dai programmi alla shell

I risultati delle elaborazioni eseguite dai comandi generalmente vengono scritti nello standard output. Oltre a questo il programma ritorna al chiamante (in questo caso alla shell) un valore di ritorno che, per convenzione, di solito indica l'eventuale presenza di errori. Gli shell script possono a loro volta ritornare un valore utilizzando il comando "exit valore".

All'interno di uno script i risultati delle operazioni possono essere passati direttamente ad un altro comando mediante un operatore di pipe oppure possono essere utilizzati dalla shell. Uno dei modi per far ciò è quello di far creare al pro-

gramma un file contenente i risultati, da cui poi andare a rileggerli. Ciò si può fare in due modi: o scrivendo opportunamente il software in modo che crei il file desiderato, oppure ridirezionando in modo opportuno lo standard output mediante l'operatore di ridirezione >. La shell potrà poi accedere ai dati utilizzando l'apposito comando read. Il seguente programma di esempio crea in un file temporaneo la lista degli utenti attualmente connessi al sistema e la utilizza per stampare sullo schermo l'elenco dei processi lanciati da ognuno di essi.

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.1   1104    72 ?        S    09:07   0:03 init [3]
root         2  0.0  0.0      0     0 ?        SW   09:07   0:00 [kflushd]
root         3  0.0  0.0      0     0 ?        SW   09:07   0:00 [kupdate]
root         4  0.0  0.0      0     0 ?        SW   09:07   0:00 [kpiod]
root         5  0.0  0.0      0     0 ?        SW   09:07   0:03 [kswapd]
news       180  0.0  2.0   5576  1324 ?        S    09:07   0:03 innd -p5
bin        358  0.0  0.0   1200     0 ?        SW   09:08   0:00 [portmap]
root       374  0.0  0.0   1088     0 ?        SW   09:08   0:00 [apmd]

...
daemon     534  0.0  0.1   1128   104 ?        S    09:08   0:00 atd
nobody     668  0.0  0.0   3968     0 ?        SW   09:08   0:00 [httpd]
nobody     669  0.0  0.0   3968     0 ?        SW   09:08   0:00 [httpd]
beppe     2740  0.0  1.5   1760   996 tty3     S    10:36   0:00 -bash
beppe     2751  0.0  3.9   3376  2516 tty3     S    10:36   0:03 mutt
lorenz    2753  0.0  3.5   3376  5516 tty4     S    10:38   0:04 -bash
```

```
#!/bin/sh

TMP=/tmp/miofile.$$.tmp

who|cut -f1 -d" "|sort|uniq >$TMP

(
        while read utente
        do
             echo "Processi dell'utente $utente"
             ps axu | grep "^$utente "
        done
) <$TMP

rm -f $TMP
```

Per prima cosa viene definito nella variabile $TMP un nome univoco di file da usare per tenere i risultati temporanei delle elaborazioni. E' buona norma crearlo nella directory /tmp e cancellarlo al termine dello script (l'opzione -f del comando rm forza la cancellazione del file senza richiederne conferma all'utente). Per creare un nome univoco si è utilizzata la metavariabile $$, che contiene il pid del processo corrente.

Il comando who crea la lista degli utenti attualmente collegati al sistema, nel seguente formato:

```
root     tty4     May  5 10:17
beppe    tty5     May  5 11:45
root     pts/0    May  5 09:09
root     pts/1    May  5 11:17
```

Per ricavare il solo nome dell'utente si è utilizzato il comando cut, il quale con i parametri indicati ricava dall'output di who il primo (-f1) di più campi separati dal simbolo di spazio (-d" "). Il resto della pipeline serve per eliminare eventuali utenti collegati alla macchina da più sessioni di lavoro (nell'esempio, root). Il metodo utilizzato consiste nell'ordinare la lista mediante il comando sort e nell'utilizzare uniq per eliminare i nomi ripetuti. L'ordinamento è necessario per il corretto funzionamento di uniq.

Una volta che l'output è stato inserito nel file temporaneo, è possibile andare a rileggerlo mediante il comando read. Esso è inserito all'interno di una struttura while, la quale ripete l'operazione fino a quando il valore ritornato da read vale 0, ovvero fino alla fine del file. I comandi racchiusi fra parentesi tonde vengono eseguiti utilizzando una nuova shell figlia di quella che gestisce lo script principale. In questo modo è possibile ridirezionare senza problemi lo standard input solo per quella sezione di codice senza influire su tutto il programma. Il comando read spezza l'input utilizzando il contenuto della variabile d'ambiente IFS, la quale come valore predefinito contiene "a capo", lo spazio e altri simboli di "blank".

Il nome di ogni utente della lista viene poi utilizzato, a turno, per filtrare l'output del comando ps alla ricerca dei processi ad esso appartenenti. L'output di partenza è il seguente:

(vedi tabella in alto)

Esso viene filtrato cercando mediante grep per trovare tutte e sole le righe che iniziano (il simbolo ^ in una espressione regolare indica l'inizio della riga) col nome dell'utente seguito da uno spazio.

L'esempio precedente, essendo utilizzato a scopo didattico, non è ottimizzato. Una cosa che si potrebbe fare per migliorarlo è quella di passare l'output dei comandi grep...uniq direttamente alla shell che esegue la ricerca senza passare attraverso il file temporaneo, nel seguente modo:

```
#!/bin/sh

who|cut -f1 -d" "|sort|uniq | (
        while read utente
        do
             echo "Processi dell'utente $utente"
             ps axu | grep "^$utente "
        done
)
```

Un altro metodo, forse ancora più utilizzato, per passare dati dai programmi alla shell, è quello di ricorrere all'operatore di sostituzione del comando con il suo output, che abbiamo visto nelle lezioni precedenti. Possiamo utilizzare questo meccanismo per riscrivere il programma precedente in modo ancora più semplice ed efficiente.

```
#!/bin/sh

for utente in `who|cut -f1 -d" "|sort|uniq`
do
        echo "Processi dell'utente $utente"
        ps axu | grep "^$utente "
done
```

In seguito alla sostituzione è come se il comando for venisse richiamato nel seguente modo:

```
for utente in beppe lorenz root
```

Questa linea esegue tutte le operazioni comprese fra le parole chiave "do" e "done", assumendo come valore della variabile $utente ad uno ad uno tutti gli elementi della lista specificata (beppe, lorenz, root).

Il meccanismo di sostituzione può essere utilizzato anche per assegnare valori alle variabili. Ad esempio per ottenere il tipo e il numero di versione del sistema operativo utilizzato si ricorre all'output generato dal comando uname (l'opzione -s scrive in output il tipo del sistema operativo, mentre -r scrive la versione):

```
OS=`uname -s`
RELEASE=`uname -r`
```

Un altro esempio consiste nell'ottenere la lunghezza della stringa contenuta in una variabile:

```
nome="Edoardo"
LUNGHEZZA=`${#nome}`
```

Alla fine dell'operazione nella variabile $LUNGHEZZA è contenuto il valore 7.

La struttura condizionale if...then...else

Abbiamo già visto la volta scorsa un semplice esempio di valutazione di una condizione mediante il costrutto if...then...else. La sintassi generale è la seguente (le parentesi quadre identificano parti facoltative):

if condizioni **then** comandi [**else** comandi2] **fi**

Per una migliore leggibilità è anche possibile riscrivere il tutto nel seguente modo:

```
if condizioni
then
        comandi
else
        comandi2
fi
```

La sintassi è del tutto analoga a quella della stessa struttura presente in altri linguaggi di programmazione. Per prima cosa vengono eseguiti i comandi che seguono immediatamente l'if. Se il valore ritornato è zero (ricordo che per convenzione il valore di uscita 0 indica che il comando è andato a buon fine, mentre un valore diverso identifica un errore) vengono eseguiti i comandi che seguono la parola then. In caso contrario vengono eseguiti, se specificati, i comandi che seguono la parola chiave else (altrimenti). Il valore di stato ritornato dalla struttura if nella metavariabile $? corrisponde al valore ritornato dall'ultimo comando eseguito oppure è zero se la condizione non è stata verificata.

Una possibile estensione alla struttura if è la seguente:

if condizioni **then** comandi [**elif** condizioni2 **then** comandi2] ... [**else** comandi3] **fi**

In questo caso se il valore ritornato è diverso da zero, vengono valutati in sequenza anche i valori di ritorno dei comandi che seguono gli elif. Appena una condizione risulta verificata, vengono eseguiti i comandi che seguono il then corrispondente. Se nessuna condizione è verificata viene eseguito anche in questo caso l'eventuale else. Un metodo alternativo allo specificare la lista dei comandi da eseguire è quello di valutare, mediante il comando test, il valore di ritorno di un comando precedente, che è contenuto nella variabile $?.

```
comando

if test $? = 0
then
        comandi
else
        comandi2
fi
```

L'esempio appena visto può essere riscritto nel seguente modo:

```
comando

if [ $? = 0 ]
then
        comandi
else
        comandi2
fi
```

Non ci si lasci ingannare dal paragone con altri linguaggi di programmazione: in questo caso la parentesi non è parte della sintassi della struttura, ma semplicemente il comando [è definito come un alias del comando test. Lo si può vedere listando il contenuto della directory /usr/bin

```
# ls -ls /usr/bin/[
    0 lrwxrwxrwx    1 root root   4 Feb 13 13:03
/usr/bin/[ -> test
```

In realtà per ragioni di efficienza la Bash implementa direttamente al proprio interno questo comando; non è però detto che tutte le shell lo facciano e per questo motivo esso è presente anche a parte.

Test è un comando multiuso, che permette di valutare diversi tipi di espressione. Facendo precedere l'espressione dal simbolo ! se ne valuta la "falsità":

```
if [ ! "$nome" = "Mario Rossi" ]
then
        echo "Lo script è eseguibile solo
dall'utente Mario Rossi."
        exit 1
fi
```

Si noti che nel confronto fra stringhe, per evitare problemi, si è quotata la variabile $nome mediante le virgolette. Infatti essa viene sostituita col suo valore prima di eseguire il confronto e il fatto che la stringa contiene degli spazi avrebbe al-

trimenti causato problemi al confronto (Mario spazio Rossi sono due parametri invece dell'unico parametro che il comando si aspetta). Come buona norma per evitare possibili problemi difficili da comprendere e risolvere è bene prendere l'abitudine di quotare comunque una variabile.

Si noti che per segnalare la condizione di errore, lo script restituisce un valore di ritorno diverso da zero.

Volendo è possibile utilizzare i connettori logici AND (-a) e OR (-o) per valutare assieme più espressioni:

```
if [ "$nome" = "Mario" -a "$cognome" = "Rossi" ]
then
        echo "Ciao Mario Rossi"
fi
```

oppure

```
if [ "$USER" = "beppe" -o "$USER" = "lorenz" ]
then
        echo "Benvenuto $USER"
fi
```

Alcuni possibili utilizzi di test sono riassunti nella tabella seguente, mentre altri si possono trovare sul manuale in linea.

sintassi	condizione testata
condizione1 -a condizione2	AND logico fra due condizioni: verifica il test solo se entrambe le condizioni sono verificate
condizione1 -o condizione2	OR logico fra due condizioni: verifica il test se almeno una delle condizioni è verificata
-n stringa	la lunghezza della stringa è maggiore di zero
-z stringa	la stringa ha lunghezza nulla
stringa1 = stringa2	le due stringhe coincidono
stringa1 != stringa2	le due stringhe sono diverse
intero1 -eq intero2	i due interi sono uguali
intero1 -gt intero2	intero1 è maggiore di intero2
intero1 -lt intero2	intero1 è minore di intero2
intero1 -ne intero2	intero1 è diverso da intero2
file1 -nt file2	file1 è più recente di file2
-d file	il file esiste ed è una directory
-f file	il file esiste ed è un file normale
-e file	il file esiste
-s file	il file esiste ed ha lunghezza diversa da zero
-w file	il file è scrivibile dall'utente

Eventuali variabili o espressioni valutate dalla shell vengono sostituite prima di eseguire il comando test. Il seguente esempio mostra come valutare se la lunghezza di una variabile eccede un determinato valore:

```
if [ ${#user} -gt 8 ]
then
        echo "Non puoi usare un nome maggiore di 8
        caratteri"
fi
```

La struttura case...esac

Nel caso si debba scegliere fra più alternative, può essere consigliabile ricorrere alla struttura di selezione case:

case parola **in** [pattern [| pattern] ...) list ;;] ... **esac**

Essa confronta "parola" con tutte le espressioni regolari specificate come pattern. Appena la parola soddisfa una espressione regolare, viene eseguita la lista di comandi associata a quest'ultima (terminata dal simbolo ;;) e l'elaborazione continua con la prima riga dopo l'esac. Lo stato di uscita della struttura case è zero se non viene trovato alcun pattern che si accoppi con la stringa, altrimenti corrisponde al valore di uscita ritornato dall'ultimo comando eseguito nella lista associata al pattern trovato. Le regole utilizzate per l'espansione delle espressioni sono le stesse che abbiamo visto le volte scorse. E' possibile utilizzare il carattere | per eseguire un OR logico fra più scelte:

```
case "$nome" in
     [Gg]iuseppe|[Ll]orenzo)  echo "Benvenuto !"
                              ;;

      *) echo "Non puoi usare questo programma."
         exit 1
         ;;
esac
```

Nell'esempio si notano due cose: per verificare la variabile $nome non sono state specificate delle semplici stringhe, bensì delle espressioni regolari, in modo che il programma funzioni correttamente anche se si inseriscono i nomi con l'iniziale maiuscola ("Giuseppe" e "giuseppe").

In questo caso si sarebbe anche potuto scrivere "giuseppe|Giuseppe|lorenzo|Lorenzo)", ma il metodo non sarebbe stato "scalabile"..

La seconda cosa da notare è che si è specificata come ultima scelta l'espressione regolare *, che intercetta qualunque stringa che non abbia passato i controlli precedenti.

Vediamo ora un possibile utilizzo dei valori OS e RELEASE trovati nell'esempio precedente: in un programma di installazione di un pacchetto software è spesso necessario verificare il tipo del sistema operativo utilizzato. In uno shell script ciò è possibile mediante una struttura case, o eventualmente due nidificate se si desidera verificare anche il numero di versione. Si noti che si è utilizzata l'espressione regolare 2.2.* per ricercare tutte le stringhe di tipo 2.2.0, 2.2.1, ...

```
# Programma di esempio: verifica il tipo di s.o.
usato

OS=`uname -s`
RELEASE=`uname -r`

case    "$OS" in
       Linux) echo "Stai usando Linux versione
$RELEASE"

       case "$RELEASE" in
              2.2.*) echo "Ottima scelta"
                     ;;

              *)     echo "Occorre Linux in ver
                     sione 2.2.x."
                     exit 1 # ritorna uno stato
                     di errore
              ;;
```

```
                esac
                ;;

        HP-UX) echo "Stai utilizzando HP-UX ver-
sione $RELEASE"
                ;;

        *)      echo "Stai usando un sistema opera
                tivo che non conosco."
                ;;
esac
```

I cicli: while, until e for

Abbiamo già visto degli esempi di cicli while e for. Analizziamone ora in modo più formale la sintassi:

while list **do** list **done**
until list **do** list **done**

Il ciclo while esegue di continuo la lista di comandi racchiusa fra le parole chiave "do" e "done", fino a quando l'ultimo comando della lista ritorna uno stato di uscita pari a zero. Il ciclo di tipo until è identico, solo che esce quando tale valore è pari a 1. Lo stato di uscita ritornato dalla struttura corrisponde al valore ritornato dall'ultimo comando eseguito nella lista, oppure è zero se non è stato eseguito alcun comando. E' possibile uscire anticipatamente da un ciclo (vale anche per le altre strutture) mediante la parola chiave "break" oppure saltare l'esecuzione di una parte di lista mediante "continue".

```
while read nome
do
        if [ "$nome" = "fine" ]
        then
                break
        fi

        if [ "$nome" = "ignorami" ]
        then
                continue
        fi

        echo $nome
done
```

La sintassi del ciclo for è:

for name **[in** valori; **] do** list ; **done**

Una scrittura più leggibile è la seguente:

for name **[in** valori; **]**
do
        list
done

In questo caso il simbolo di ; non è necessario.

In un ciclo di tipo for l'espressione che segue la parola chiave "in" può essere una lista di valori predefiniti oppure derivare da una operazione come una sostituzione di comandi oppure l'espansione di una variabile o di una espressione regolare. Alla variabile viene assegnato di volta in volta uno dei valori ottenuti.

Nel caso che segue, la lista viene ottenuta a partire da una espressione regolare. Ad essa vengono sostituiti i nomi dei file che la soddisfano (in questo caso essi sono cercati nella directory corrente).

```
for i in *.GIF
do
        NUOVO=`basename $i .GIF`.gif
        mv $i $NUOVO
done
```

Un metodo più veloce per fare la stessa cosa è il seguente:

```
for i in *.GIF
do
        mv $i `basename $i .GIF`.gif
done
```

Il prossimo esempio, molto simile, trasforma tutti i nomi di file nella directory corrente in lettere minuscole:

```
for i in *
do
        mv $i `echo $i | tr "[a-z]" "[A-Z]" `
done
```

I diversi valori derivanti da una espansione vengono spezzati considerando come separatori i caratteri contenuti nella variabile d'ambiente IFS (come valori predefiniti essa contiene lo spazio e altri simboli di "blank"). Il seguente esempio spezza un indirizzo di e-mail nelle sue componenti.

```
IFS=".@"

EMAIL="beppe@profuso.com"

for i in $EMAIL
do
        echo $i...
done
```

Le funzioni

Nel linguaggio della shell bash è possibile definire delle funzioni, che possono essere richiamate variandone i parametri. La sintassi è la seguente:

[function] name **() {** list; **}**

Questa linea definisce una funzione di nome "name". Il corpo della funzione è costituito da una lista di parametri racchiusa fra parentesi graffe. Per richiamare la funzione è sufficiente scriverne il nome seguito dagli eventuali parametri, che vengono passati, in modo analogo ai parametri di uno script, all'interno delle metavariabili ${1}, ${2}, ..., ${n}. Lo stato di uscita della funzione è quello dell'ultimo comando eseguito.

E' possibile ritornare un determinato valore di uscita utiliz-

zando il comando "return valore".

Quando la funzione termina, l'esecuzione dello script continua dalla prima linea successiva al punto di chiamata della funzione stessa.

```
function dimmiciao
{
        echo "Ciao $1."
}

dimmiciao Giuseppe
dimmiciao Mario
```

Eventuali variabili possono essere definite come "locali" alla funzione mediante il comando local:

```
function paperino
{
        local i

        for i in 1 2 3
        do
                echo $i
        done
}
```

Le funzioni definite nello script principale possono essere esportate ad eventuali shell figlie mediante il comando export, come avviene per le variabili. Le funzioni possono essere ricorsive (ovvero richiamare se stesse) e non esistono limiti al livello di ricorsione utilizzato.

Ancora sui parametri di chiamata

Riprendiamo il discorso della volta scorsa sui parametri di chiamata, che - lo abbiamo appena visto - vale anche per le funzioni. Per ottenere la lista dei parametri di chiamata è possibile utilizzare le metavariabili $* o $@. In entrambi i casi l'espressione viene espansa in tutti i parametri di chiamata partendo dal primo.

La differenza si nota quando si racchiude la metavariabile fra virgolette. Nel primo caso viene generata un'unica stringa in cui i valori dei parametri vengono separati mediante il primo carattere contenuto nella variabile d'ambiente IFS (oppure spazio se IFS non è definita).

Nel secondo caso si ottengono più parole, ognuna racchiusa fra virgolette.

```
echo "$@" Æ echo "Nonna Papera" "Pluto" "Paolino
Paperino"
```

Tale funzione è molto utile negli script.

```
function listaparametri
{
        for i in "$@"
        do
                echo "Parametro=$i"
        done
}
```

La metavariabile $# viene invece espansa nel numero di parametri passati. Ciò è utile ad esempio per verificare se ad uno script è stato passato un numero sufficiente di parametri:

```
if [ $# -lt 3 ]
then
        echo "Usage: $0 vecchiaext nuovaext file
        [file...]" >&2
        exit 1
fi
```

Il controllo precedente se non ci sono sufficienti parametri scrive un messaggio di errore ridirezionandolo nello standard output. La metavariabile $0 viene espansa nel nome del programma.

Come nell'esempio precedente, spesso si può voler scrivere un programma che prenda come parametri i primi valori passati nella linea di comando e poi compia delle operazioni usando i parametri seguenti alla stregua di nomi di file. In questo caso è utile il comando shift, che sposta in avanti (o in indietro) l'indicatore del primo parametro. Nel caso in esame si desidera scrivere un piccolo script che cambi l'estensione di un file, da richiamare come

```
cambiaestensione vecchiaestensione nuovaestensione
file1 file2 ... fileN
```

Per far ciò risulta comodo salvare i primi due parametri e poi usare shift per traslare l'indice del primo parametro, in modo che $* venga espansa nella sola lista dei file:

```
#!/bin/sh

VECCHIA=${1}
NUOVA=${2}

shift 2

for i in $*
do
        mv $i `basename .${VECCHIAEXT}`.${NUOVAEXT}
done
```

Ricordo che la sostituzione avviene prima di passare i parametri al programma.

```
./rinomina jpeg jpg *jpeg
Æ ./rinomina jpeg jpg pippo.jpeg pluto.jpeg min-
ni.jpeg
```

Conclusioni

In questo corso stiamo parlando di un particolare linguaggio che è quello della shell, in quanto esso è disponibile "di serie" su tutte le installazioni di Linux, anche le più minimali, e su tutti i sistemi UNIX, ma, ovviamente, i vantaggi e svantaggi di cui abbiamo parlato si trovano anche in altri linguaggi interpretati, ad esempio in perl.

Le idee contenute in queste lezioni perciò sono interessanti anche per chi non avesse immediato interesse a programmare shell script.

MC