BASH Programming - Introduction HOWTO

di Mike G mikkey@dynamo.com.ar Lunedì 17 Luglio 11:47:00 ART 2000

Questo articolo si propone di aiutarti ad iniziare a programmare script di shell di livello base-intermedio. Non vuole essere un documento avanzato (vedi il titolo). Io NON sono un esperto o un guru della programmazione della shell. Ho deciso di scrivere questo HOWTO perché imparerò molto e potrebbe essere utile ad altre persone. Qualsiasi tipo di riscontro sarà apprezzato, specialmente se in forma di patch :) . Traduzione di *William Ghelfi* a.k.a. Wiz of Id, Mercoledì 19 Luglio 2000.

Indice

1 Introduzione

1.1 Ottenere l'ultima versione

http://www.linuxdoc.org/HOWTO/Bash-Prog-Intro-HOWTO.html

In italiano

http://www.pluto.linux.it/ildp/HOWTO/Bash-Prog-Intro-HOWTO.html

1.2 Requisiti

Familiarità con la riga di comando GNU/Linux e con i concetti di base della programmazione, saranno d'aiuto. Nonostante questa non sia una introduzione alla programmazione, spiega (o almeno ci prova) molti concetti di base.

1.3 Usi di questo documento

Questo documento vuol essere utile nei seguenti casi

- Hai un'infarinatura di programmazione e vuoi iniziare a scrivere qualche script di shell.
- Hai una vaga idea della programmazione della shell e desideri un qualche tipo di riferimento.
- Vuoi vedere qualche script di shell ed alcuni commenti per cominciare a scriverne di tuoi.
- Stai passando da DOS/Windows (o l'hai già fatto) e vuoi preparare dei processi batch.
- Sei un nerd totale (complete nerd, ndt) e leggi ogni how-to disponibile.

1.4 Traduzioni

Koreano: Chun Hye Jin *sconosciuto*

C'erano altre traduzioni, ma non le ho incluse perché non avevo alcun URL verso cui puntarle. Se qualcuno di voi ne conoscesse, è pregato di inviarmeli via email.

1.5 Note sulla traduzione

Nel caso vi venisse l'idea (ottima, :-) di inviarmi correzioni in formato diff, vi sarei grato se per crearle utilizzaste il comando *diff -u vecchio.sgml nuovo_e_corretto.sgml*; questo perché la mia dabbenaggine mi impedisce di imparare a destreggiarmi con diff che non siano stati creati con l'opzione -u. Grazie e buona lettura!

2 Script molto facili

Questo HOW-TO tenterà di darti alcuni consigli sulla programmazione della shell basandosi principalmente su esempi.

In questa sezione troverai qualche piccolo script che si spera ti sia d'aiuto per comprendere alcune tecniche.

2.1 Il tradizionale script hello world

```
#!/bin/bash
echo Hello World
```

Questo script ha solamente due righe. La prima indica al sistema quale programma utilizzare per eseguire il file.

La seconda riga è l'unica azione compiuta dallo script, che stampa 'Hello World' sul terminale.

Se ottieni qualcosa come *./hello.sh: Command not found.* Probabilmente la prima riga '#!/bin/bash' è errata, controlla dove si trova bash o vedi 'trovare bash' per sapere come dovresti modificare tale riga.

2.2 Uno script di backup molto semplice

```
#!/bin/bash
tar -cZf /var/my-backup.tgz /home/me/
```

In questo script, invece di stampare un messaggio sul terminale, creiamo una tar-ball (archivio tar) della home directory di un utente. Questo NON è pensato per essere usato, uno script di backup più utile sarà presentato più avanti in questo documento.

3 Tutto sulla redirezione

3.1 Teoria e riferimento veloce

Esistono 3 descrittori di file: stdin, stdout e stderr (std=standard).

Basilarmente tu puoi:

1. redirigere stdout verso un file
2. redirigere stderr verso un file
3. redirigere stdout verso stderr

4. redirigere stderr verso stdout
5. redirigere stderr e stdout verso un file
6. redirigere stderr e stdout verso stdout
7. redirigere stderr e stdout verso stderr

1 'rappresenta' stdout e 2 stderr.

Una piccola nota per vedere queste cose: con il comando less puoi visualizzare sia stdout (che resterà nel buffer) che lo stderr che verrà stampato sullo schermo, ma eliminato non appena tenterai di 'sfogliare' il buffer.

3.2 Esempio: stdout verso file

Questo farà sì che l'output di un programma venga scritto su un file.

```
ls -l > ls-l.txt
```

Qui, un file chiamato 'ls-l.txt' verrà creato e conterrà ciò che vedresti sullo schermo digitando il comando 'ls -l' ed eseguendolo.

3.3 Esempio: stderr verso file

Questo farà sì che l'output di stderr di un programma venga scritto su un file.

```
grep da * 2> grep-errors.txt
```

Qui, un file chiamato 'grep-errors.txt' sarà creato e conterrà ciò che vedresti come porzione di stderr dell'output del comando 'grep da *'.

3.4 Esempio: stdout verso stderr

Questo farà sì che l'output di stderr di un programma venga scritto sul medesimo filedescriptor di stdout.

```
grep da * 1>&2
```

Qui, la porzione di stdout del comando è inviata a stderr, puoi accorgertene in diversi modi.

3.5 Esempio: stderr verso stdout

Questo farà sì che l'output di stderr di un programma venga scritto sul medesimo filedescriptor di stdout.

```
grep * 2>&1
```

Qui, la porzione di stderr del comando è inviata a stdout, se fai una pipe verso less, noterai che righe le quali normalmente 'scomparirebbero' (poiché sono scritte su stderr) ora vengono tenute (perché si trovano su stdout).

3.6 Esempio: stderr e stdout verso file

Questo porterà ogni output di un programma su un file. Può risultare a volte utile per cron, se vuoi che un comando passi in assoluto silenzio.

```
rm -f $(find / -name core) &> /dev/null
```

Questo (pensando a cron) eliminerà ogni file chiamato 'core' in qualsiasi directory. Osserva che dovresti essere piuttosto sicuro di cosa sta facendo un comando, prima di eliminarne ogni output.

4 Le pipe

Questa sezione mostra in maniera molto semplice e pratica come usare le pipe, e per quale motivo potresti volerlo fare.

4.1 Che cosa sono e perché vorrai utilizzarle

Le pipe ti permettono di usare (molto semplice, insisto) l'output di un programma come input di un altro.

4.2 Esempio: semplice pipe con sed

Questo è un modo molto semplice di usare le pipe.

```
ls -l | sed -e "s/[aeio]/u/g"
```

Qui, succede questo: prima è eseguito il comando ls, ed il suo output, invece di essere stampato, è inviato (mandato in pipe) al programma sed, che a sua volta, stampa quello che ha da stampare.

4.3 Esempio: una alternativa a ls -l *.txt

Probabilmente, questo è il modo più difficile per fare ls -l *.txt, ma è qua per illustrare le pipe, non per risolvere un tale dilemma di elencazione.

```
ls -l | grep "\.txt$"
```

Qui, l'output del programma ls -l è inviato al programma grep, che, a sua volta, stamperà le righe che corrispondono alla regex \.txt$.

5 Variabili

Puoi usare le variabili come in ogni linguaggio di programmazione. Non esistono tipi di dati. Una variabile nella bash può contenere un numero, un carattere, una stringa di caratteri.

Non hai bisogno di dichiarare una variabile, il solo atto di assegnare un valore al suo riferimento farà sì che venga creata.

5.1 Esempio: Hello World! usando le variabili

```
#!/bin/bash
STR="Hello World!"
echo $STR
```

La riga 2 crea una variabile chiamata STR e le assegna la stringa Hello World!. Poi il VALORE di questa variabile è recuperato inserendo il simbolo '$' all'inizio (del riferimento, ndt). Osserva (provaci!) che se non usi il segno '$', l'output del programma sarà differente, e probabilmente non quello che avresti voluto fosse.

5.2 Esempio: Uno script di backup molto semplice (un poco migliore)

```
#!/bin/bash
OF=/var/my-backup-$(date +%Y%m%d).tgz
tar -cZf $OF /home/me/
```

Questo script introduce un'altra cosa. Prima di tutto, dovresti aver dimestichezza con la creazione e l'assegnazione di variabile alla riga 2. Osserva l'espressione '$(date +%Y%m%d)'. Se esegui lo script noterai che lancia il comando incluso tra le parentesi, catturando il suo output.

Osserva che in questo script, il nome del file di output sarà diverso ogni giorno, a causa dell'opzione di formattazione del comando date (+%Y%m%d). Puoi cambiarlo specificando una differente formattazione.

Altri esempi:

echo ls

echo $(ls)

5.3 Variabili locali

Le variabili locali possono essere create utilizzando la keyword *local*.

```
#!/bin/bash
HELLO=Hello
function hello {
        local HELLO=World
        echo $HELLO
}
echo $HELLO
hello
echo $HELLO
```

Questo esempio dovrebbe essere sufficiente a mostrarti come utilizzare una variabile locale.

6 Condizionali

Le (espressioni, ndt) condizionali ti permettono di decidere se compiere o no un'azione. Tale decisione è presa valutando un'espressione.

6.1 Pura Teoria

Le (espressioni, ndt) condizionali hanno varie forme. La forma più basilare è: **if** *espressione* **then** *istruzione* dove 'istruzione' viene eseguita solamente se 'espressione' ha valore vero. '2<1' è una espressione che ha valore falso, mentre '2>1' ha valore vero.

Le condizionali hanno altre forme come: **if** *espressione* **then** *istruzione1* **else** *istruzione2*. Qui 'istruzione' è eseguita se 'espressione' è vera, altrimenti viene eseguita 'istruzione2'.

Ancora un'altra forma di (espressione, ndt) condizionale è: **if** *espressione1* **then** *istruzione1* **else** **if** *espressione2* **then** *istruzione2* **else** *istruzione3*. In questa forma è stato aggiunto solamente ELSE IF 'espressioen2' THEN 'istruzione2' che fa eseguire istruzione2 se espressione2 vale vero. Il resto è come ti puoi immaginare (vedi le forme precedenti).

Una parola sulla sintassi:

La base per i costrutti 'if' nella bash è questa:

if [espressione];

then

codice eseguito se 'espressione' è vera.

fi

6.2 Esempio: Esempio basilare di condizionale if .. then

```
#!/bin/bash
if [ "foo" = "foo" ]; then
   echo expression evaluated as true
fi
```

Il codice da eseguire se l'espressione tra parentesi quadre è vera può trovarsi solamente dopo la parola 'then' e prima del 'fi' che indica la fine del codice eseguito sotto condizione.

6.3 Esempio: Esempio basilare di condizionale if .. then ... else

```
#!/bin/bash
if [ "foo" = "foo" ]; then
   echo expression evaluated as true
else
   echo expression evaluated as false
fi
```

6.4 Esempio: Condizionali con variabili

```
#!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" = "$T2" ]; then
    echo expression evaluated as true
else
    echo expression evaluated as false
```

```
        fi
```

7 Cicli for, while e until

In questa sezione troverai cicli for, while e until.

Il ciclo **for** è leggermente diverso da quello degli altri linguaggi di programmazione. Basilarmente, ti permette un'iterazione su una serie di 'parole' in una stringa.

Il **while** esegue una porzione di codice se l'espressione di controllo è vera, e si ferma esclusivamente quando è falsa (o viene raggiunta un'interruzione esplicita all'interno del codice eseguito).

Il ciclo **until** è all'incirca uguale al ciclo while, solo che il codice è eseguito finchè l'espressione di controllo ha valore falso.

Se hai il sospetto che while e until siano molto simili hai ragione.

7.1 Per esempio

```
#!/bin/bash
for i in $( ls ); do
    echo item: $i
done
```

Sulla seconda riga, dichiariamo i come la variabile che prenderà i differenti valori contenuti in $(ls).

La terza riga potrebbe essere più lunga se necessario, o ci potrebbero essere più righe prima del done (4).

'done' (4) indica che il codice che ha utilizzato il valore di $i è terminato e $i può ricevere un nuovo valore.

Questo script ha veramente poco senso, ma un modo più utile per utilizzare il ciclo for sarebbe di usarlo per isolare (to match, ndt) solo certi file nell'esempio precedente.

7.2 For simil-C

fiesh ha suggerito di aggiungere questo modo di eseguire un ciclo. Si tratta di un ciclo for più simile al for dei linguaggi C/perl...

```
#!/bin/bash
for i in `seq 1 10`;
do
        echo $i
done
```

7.3 Esempio di while

```
#!/bin/bash
COUNTER=0
while [  $COUNTER -lt 10 ]; do
    echo The counter is $COUNTER
    let COUNTER=COUNTER+1
```

```
done
```

Questo script 'emula' la ben conosciuta struttura 'for' dei linguaggi C, Pascal, perl, etc.

7.4 Esempio di until

```
#!/bin/bash
COUNTER=20
until [  $COUNTER -lt 10 ]; do
    echo COUNTER $COUNTER
    let COUNTER-=1
done
```

8 Funzioni

Come in quasi ogni linguaggio di programmazione, puoi utilizzare le funzioni per raggruppare porzioni di codice in modo più logico oppure praticare la divina arte della ricorsione (ricorsività, ndt).

Dichiarare una funzione è giusto questione di scrivere function mia_funzione { mio_codice }.

Chiamare una funzione è proprio come chiamare un altro programma, semplicemente scrivi il suo nome.

8.1 Esempio di funzioni

```
#!/bin/bash
function quit {
    exit
}
function hello {
    echo Hello!
}
hello
quit
echo foo
```

Le righe 2-4 contengono la funzione 'quit'. Le righe 5-7 contengono la funzione 'hello'. Se non sei assolutamente sicuro di cosa faccia questo script, sei pregato di provarlo!.

Osserva che le funzioni non hanno bisogno di essere dichiarate in alcun ordine particolare.

Lanciando questo script lo noterai per la prima volta: la funzione 'hello' è chiamata, per seconda la funzione 'quit', e il programma non raggiunge mai la riga 10.

8.2 Esempio di funzioni con parametri

```
#!/bin/bash
function quit {
   exit
}
function e {
```

```
        echo $1
    }
    e Hello
    e World
    quit
    echo foo
```

Questo script è praticamente identico al precedente. La differenza principale è la funzione 'e'. Tale funzione, stampa il primo argomento che riceve. Gli argomenti, nell'ambito delle funzioni, vengono trattati nella stessa maniera degli argomenti passati allo script.

9 Interfacce utente

9.1 Utilizzo di select per la creazione di semplici menù

```
#!/bin/bash
OPTIONS="Hello Quit"
select opt in $OPTIONS; do
    if [ "$opt" = "Quit" ]; then
     echo done
     exit
    elif [ "$opt" = "Hello" ]; then
     echo Hello World
    else
     clear
     echo bad option
    fi
done
```

Se lanci questo script vedrai che si tratta di quel che i programmatori sognano per i menù testuali. Probabilmente noterai che è molto simile al costrutto 'for', solo che invece di eseguire il ciclo per ogni 'parola' in $OPTIONS, richiede input all'utente.

9.2 Usare la riga di comando

```
#!/bin/bash
if [ -z "$1" ]; then
    echo usage: $0 directory
    exit
fi
SRCD=$1
TGTD="/var/backups/"
OF=home-$(date +%Y%m%d).tgz
tar -cZf $TGTD$OF $SRCD
```

Ciò che fa questo script ti dovrebbe essere chiaro. L'espressione nella prima condizionale controlla se il programma ha ricevuto un argomento ($1) e ed esce in caso negativo, mostrando all'utente un breve messaggio di utilizzo. A questo punto il resto dello script dovrebbe esserti chiaro.

10 Varie

10.1 Leggere l'input dell'utente con read

In molte occasioni potresti voler richiedere l'utente un certo input, Ci sono diversi modi per raggiungere tale scopo. Eccone uno:

```
#!/bin/bash
echo Please, enter your name
read NAME
echo "Hi $NAME!"
```

Come variante, puoi ottenere valori multipli con read, questo esempio dovrebbe chiarire il concetto.

```
#!/bin/bash
echo Please, enter your firstname and lastname
read FN LN
echo "Hi! $LN, $FN !"
```

10.2 Valutazione aritmetica

Dalla riga di comando (o da una shell) prova questo:

echo 1 + 1

Se ti aspettavi di vedere '2' sarai dispiaciuto. Che fare se vuoi che BASH processi dei numeri che hai? Ecco la soluzione:

echo $((1+1))

Questo produrrà un output più 'logico'. Questo per valutare espressione aritmetica. Puoi ottenere lo stesso risultato con qualcosa come:

echo $[1+1]

Se hai bisogno di usare le frazioni, o operazioni più complesse (more math, ndt), o semplicemente perché ne hai voglia, puoi utilizzare bc per processare le espressioni aritmetiche.

Se eseguissi echo $[3/4] al prompt dei comandi, mi restituirebbe 0 poiché bash usa solamente interi in fase di risposta. Eseguendo echo 3/4|bc -l, ti restituirebbe un più adeguato 0.75.

10.3 Trovare bash

Da un messaggio di mike (vedi Grazie a)

tu usi sempre #!/bin/bash .. potresti fornire un esempio di come

scoprire dove si trovi bash.

'locate bash' è preferibile, ma non tutte le macchine hanno locate.

'find ./ -name bash' dalla root directory (quella indicata con /,

ndt) funziona, in genere.

Suggerimenti su dove cercare:

ls -l /bin/bash

ls -l /sbin/bash

ls -l /usr/local/bin/bash

ls -l /usr/bin/bash

ls -l /usr/sbin/bash

ls -l /usr/local/sbin/bash

(non me vengono in mente altri al momento. (l'ho trovata nella

maggior parte di questi posti in sistemi diversi).

Puoi provare anche 'which bash'.

10.4 Prendere il valore di ritorno da un programma

Nella bash, il valore di ritorno di un programma è memorizzato in variabile speciale chiamata $?.

Questo mostra come catturare il valore restituito da un programma; faccio conto che la directory *dada* non esista. (Anche questo è stato suggerito da mike)

```
#!/bin/bash
cd /dada &> /dev/null
echo rv: $?
cd $(pwd) &> /dev/null
echo rv: $?
```

10.5 Catturare l'output di un programma

Questo piccolo script mostra tutte le tabelle da tutti i database (assumendo che tu abbia MySQL installato). Inoltre, considera la possibilità di modificare il comando 'mysql' per aggiungere uno username ed una password validi.

```
#!/bin/bash
DBS=`mysql -uroot  -e"show databases"`
for b in $DBS ;
do
        mysql -uroot -e"show tables from $b"
done
```

10.6 File a sorgenti multipli

Puoi usare più di un file per volte tramite il comando source.

__TO-DO__

11 Tavole

11.1 Operatori di confronto tra stringhe

(1) s1 = s2

(2) s1 != s2

(3) s1 < s2

(4) s1 > s2

(5) -n s1

(6) -z s1

(1) s1 corrisponde a s2

(2) s1 non corrisponde a s2

(3) __TO-DO__

(4) __TO-DO__

(5) s1 non è vuota (contiene uno o più caratteri)

(6) s1 è vuota

11.2 Esempi di confronto tra stringhe

Confrontare due stringhe.

```
#!/bin/bash
S1='string'
S2='String'
if [ $S1=$S2 ];
then
        echo "S1('$S1') is not equal to S2('$S2')"
fi
if [ $S1=$S1 ];
then
        echo "S1('$S1') is equal to S1('$S1')"
fi
```

Riporto qui una nota da una mail, inviata da Andreas Beck, in riferimento all'uso di *if [$1 = $2]*.

Non è proprio una buona idea, dato che se una tra $S1 ed $S2 è vuota, riceverai un errore di sintassi. x$1=x$2 oppure $1=$2 vanno meglio.

11.3 Operatori aritmetici

+

-

*

/

% (resto della divisione)

11.4 Operatori aritmetici relazionali

-lt (<)

-gt (>)

-le (<=)

-ge (>=)

-eq (==)

-ne (!=)

I programmatori in C dovrebbero limitarsi a mappare l'operatore alla parentesi corrispondente.

11.5 Comandi utili

Questa sezione è stata riscritta da Kees (vedi Grazie a...)

Alcuni di questi comandi quasi prevedono completi linguaggi di programmazione. Per tali comandi saranno spiegate soltanto le basi. Per una descrizione più dettagliata, ti consiglio una lettura più approfondita alle pagine man di ciascun comando.

sed (stream editor)

Sed è un editor non interattivo. Invece di alterare un file muovendo il cursore sullo schermo, usi uno script di istruzioni di editing per sed, più il nome del file da editare. Puoi considerare sed anche come un filtro. Diamo un'occhiata ad alcuni esempi:

```
$sed 's/vecchio_testo/testo_che_lo_sostituisce/g' /tmp/dummy
```

Sed rimpiazza la stringa 'vecchio_testo' con la stringa 'testo_che_lo_sostituisce' e legge dal file /tmp/dummy. Il risultato sarà inviato a stdout (normalmente la consolle) ma puoi anche aggiungere '> cattura' alla fine della riga qua sopra così che sed invii l'output al file 'cattura'.

```
$sed 12, 18d /tmp/dummy
```

Sed mostra tutte le righe tranne quelle da 12 a 18. Il file originale non è alterato da questo comando.

awk (manipolazione di datafile, recuperare testo e processarlo)

Esistono molte implementazioni del linguaggio di programmazione AWK (gli interpreti più conosciuti sono gawk della GNU e 'new awk' mawk.) Il principio è semplice: AWK ricerca un modello, e per ogni corrispondenza verrà compiuta una azione.

Di nuovo, ho creato un file dummy contente le seguenti righe:

test123

test

tteesstt

```
$awk '/test/ {print}' /tmp/dummy
```

test123

test

Il modello cercato da AWK è 'test' e l'azione che compie quando trova una riga nel file /tmp/dummy con la stringa 'test' è 'print' (stampa, ndt).

```
$awk '/test/ {i=i+1} END {print i}' /tmp/dummy
```

3

Quando stai cercando più di un modello, sarebbe meglio se sostituissi il testo tra apici con '-f file.awk' così da poter inserire tutti i modelli e le azioni nel file 'file.awk'.

grep (stampa righe che corrispondono ad un modello di ricerca)

Abbiamo già incontrato un paio di comandi grep nei capitoli precedenti, che mostrano le righe corrispondenti ad un modello. Ma grep sa fare di più.

```
$grep "la sto cercando" /var/log/messages -c
```

12

La stringa la sto cercando è stata trovata 12 volte nel file /var/log/messages.

[ok, questo esempio era uno scherzo, il file /var/log/messages era preparato :-)]

wc (conta righe, parole e byte)

Nell'esempio seguente, notiamo che l'output non è quello che ci aspettavamo. Il file dummy, così come è usato in questo esempio, contiene il seguente testo: *bash introduction howto test file*

```
$wc --words --lines --bytes /tmp/dummy
```

2 5 34 /tmp/dummy

Wc non si cura dell'ordine dei parametri. Wc li stampa sempre nell'ordine standard, cioè, come puoi vedere: <righe><parole><byte><nomefile>.

sort (ordina le righe dei file di testo)

Questa volta il file dummy contiene il seguente testo:

b

c

a

```
$sort /tmp/dummy
```

Ecco come dovrebbe apparire l'output:

a

b

c

I comandi non dovrebbero essere così semplici :-)

bc (un linguaggio di programmazione che fa da calcolatrice)

Bc accetta calcoli dalla riga di comando (input da un file. Non da un operatore di redirezione e da una pipe), ma anche da una interfaccia utente. La seguente dimostrazione presenta alcuni dei comandi. Osserva che io lancio

bc usando il parametro -q per evitare un messaggio di benvenuto.

```
$bc -q
```

1 == 5

0

0.05 == 0.05

1

5 != 5

0

2 ^ 8

256

sqrt(9)

3

while (i != 9) {

i = i + 1;

print i

}

123456789

quit

tput (inizializza un terminale o interroga il database di terminfo)

Una piccola dimostrazione delle capacità di tput:

```
$tput cup 10 4
```

Il prompt appare a (y10,x4).

```
$tput reset
```

Pulisce lo schermo e il prompt appare a (y1,x1). Nota che (y0,x0) è l'angolo in alto a sinistra.

```
$tput cols
```

80

Mostra il numero di caratteri possibili in direzione x.

È vivamente raccomandato di familiarizzare con questi programmi (al meno). Ci sono tonnellate di piccoli programmi che ti permetteranno di fare delle vere magie dalla riga di comando.

[alcuni esempi sono tratti da pagine man o FAQ]

12 Altri Script

12.1 Applicare un comando a tutti i file in una directory.

12.2 Esempio: Uno script di backup molto semplice (ancora un poco migliore)

```
#!/bin/bash
SRCD="/home/"
TGTD="/var/backups/"
OF=home-$(date +%Y%m%d).tgz
tar -cZf $TGTD$OF $SRCD
```

12.3 Rinominatore di file

```
#!/bin/sh
# renna: rename multiple files according to several rules
# written by felix hudson Jan - 2000

#first check for the various 'modes' that this program has
#if the first ($1) condition matches then we execute that portion of the
#program and then exit

# check for the prefix condition
if [ $1 = p ]; then

#we now get rid of the mode ($1) variable and prefix ($2)
  prefix=$2 ; shift ; shift

# a quick check to see if any files were given
# if none then its better not to do anything than rename some non-existent
# files!!

  if [$1 = ]; then
     echo "no files given"
     exit 0
  fi

# this for loop iterates through all of the files that we gave the program
# it does one rename per file given
  for file in $*
    do
    mv ${file} $prefix$file
  done

#we now exit the program
  exit 0
```

```
fi

# check for a suffix rename
# the rest of this part is virtually identical to the previous section
# please see those notes
if [ $1 = s ]; then
  suffix=$2 ; shift ; shift

   if [$1 = ]; then
    echo "no files given"
   exit 0
   fi

 for file in $*
  do
   mv ${file} $file$suffix
 done

 exit 0
fi

# check for the replacement rename
if [ $1 = r ]; then

  shift

# i included this bit as to not damage any files if the user does not specify
# anything to be done
# just a safety measure

  if [ $# -lt 3 ] ; then
    echo "usage: renna r [expression] [replacement] files... "
    exit 0
  fi

# remove other information
  OLD=$1 ; NEW=$2 ; shift ; shift

# this for loop iterates through all of the files that we give the program
# it does one rename per file given using the program 'sed'
# this is a sinple command line program that parses standard input and
# replaces a set expression with a give string
# here we pass it the file name ( as standard input) and replace the nessesary
# text

  for file in $*
  do
    new=`echo ${file} | sed s/${OLD}/${NEW}/g`
    mv ${file} $new
  done
exit 0
fi

# if we have reached here then nothing proper was passed to the program
# so we tell the user how to use it
```

```
        echo "usage;"
        echo " renna p [prefix] files.."
        echo " renna s [suffix] files.."
        echo " renna r [expression] [replacement] files.."
        exit 0

        # done!
```

12.4 Rinominatore di file (semplice)

```
#!/bin/bash
# renames.sh
# basic file renamer

criteria=$1
re_match=$2
replace=$3

for i in $( ls *$criteria* );
do
    src=$i
    tgt=$(echo $i | sed -e "s/$re_match/$replace/")
    mv $src $tgt
done
```

13 Quando qualcosa va male (debugging)

13.1 Modi di chiamare BASH

Una cosa carina da fare è di aggiungere alla prima riga

```
#!/bin/bash -x
```

Ciò produrrà un po' di interessanti informazioni di output

14 Informazioni sul documento

Sentiti libero di proporre suggerimenti/correzioni, o qualunque cosa tu pensi che potrebbe essere interessante vedere in questo documento. Io cercherò di aggiornarlo al più presto possibile.

14.1 (no) warranty

This documents comes with no warranty of any kind. and all that

14.2 Traduzioni

Italiano: a cura di Willy *is here*

Francese: a cura di Laurent Martelli *is missed*

Sono convinto che esistano altre traduzioni, ma non ne ho alcuna notizia; se voi le avete, per piacere, mandatemele via email cosi potrò aggiornare questa sezione.

14.3 Grazie a

- Le persone che hanno tradotto questo documento in altre lingue (sezione precedente).
- Nathan Hurst per avermi mandato un sacco di correzioni.
- Jon Abbott per aver inviato commenti sulla valutazione delle espressioni aritmetiche.
- Felix Hudson per aver scritto lo script *renna*.
- Kees van den Broek (per aver inviato molte correzioni, riscritto la sezione dei comandi utili).
- Mike (pink) ha avanzato qualche suggerimento su come trovare bash e testare i file.
- Fiesh ha avanzato un buon suggerimento per la sezione dei cicli.
- Lion ha suggerito di menzionare un errore comune (./hello.sh: Command not found.).
- Andreas Beck ha fatto diverse correzioni e commenti.

14.4 History

Aggiunta la sezione comandi utili riscritta da Kess.

Inclusi molti suggerimenti e correzioni.

Aggiunti esempi sul confronto tra stringhe.

v0.8 abbandonata la numerazione delle versioni, credo che la data sia abbastanza.

v0.7 Altre correzioni e alcune vecchie sezioni TO-DO riscritte.

v0.6 Correzioni minori.

v0.5 Aggiunta la sezione sulla redirezione.

v0.4 Scomparsa dalla sua locazione a causa del mio ex-capo e questo documento ha trovato il suo nuovo posto all'opportuno url: www.linuxdoc.org.

precedenti: non mi ricordo e non usavo rcs o cvs :(

14.5 Altre risorse

Introduction to bash (under BE) *http://org.laol.net/lamug/beforever/bashtut.htm*

Bourne Shell Programming http://207.213.123.70/book/