

Operatori su stringhe

Minimale...

Accesso alle variabili

- Operatori disponibili:

`$<var>` o `${<var>}`

ritorna il valore di `<var>`

`${<var>:-<val>}`

se `<var>` esiste ed ha valore non vuoto, ritorna il suo valore, altrimenti ritorna `<val>`

`${<var>:=<val>}`

se `<var>` esiste ed ha valore non vuoto, ritorna il suo valore, altrimenti assegna `<val>` a `<var>` e ritorna `<val>`

`${<var>:?<message>}`

se `<var>` esiste ed ha valore non vuoto, ritorna il suo valore, altrimenti stampa il nome e `<message>` su `stderr`

Accesso alle variabili (2)

- Operatori disponibili (cont.):

```
${<var>:+<val>}
```

se **<var>** esiste ed ha valore non vuoto, ritorna il valore **<val>**

- Esempi:

```
bash:~$ echo $PIPP0
```

-- variabile non definita

```
bash:~$ echo ${PIPP0:-ii}
```

```
ii
```

```
bash:~$ echo $PIPP0
```

-- ancora non definita

```
bash:~$
```

Accesso alle variabili (3)

- Esempi:

```
bash:~$ echo $PIPP0
```

```
iiii -- variabile definita
```

```
bash:~$ echo ${PIPP0:+kk}
```

```
kk
```

```
bash:~$ echo $PIPP0
```

```
iiii -- ancora stesso valore
```

```
bash:~$
```

Accesso alle variabili (4)

- Esempi:

```
bash:~$ echo $PIPP0
```

```
iiii -- variabile definita
```

```
bash:~$ echo ${PIPP0:+kk}
```

```
kk
```

```
bash:~$ echo $PIPP0
```

```
iiii -- stesso valore
```

```
bash:~$ echo ${PIPP0:=kkll}
```

```
iiii -- non modificata
```

```
bash:~$
```

Sottostringhe

```
${<var>:<offset>}
```

```
${<var>:<offset>:<length>}
```

ritorna la sottostringa di **<var>** che inizia in posizione **<offset>** (NOTA: il primo carattere è in posizione 0)

Nella seconda forma la sottostringa è lunga **<length>** caratteri. Esempio:

```
bash:~$ A=armadillo
```

```
bash:~$ echo ${A:5}
```

```
illo
```

```
bash:~$ echo ${A:5:2}
```

```
il
```

```
bash:~$
```

Lunghezza

`${#<var>}`

consente di ottenere la lunghezza (in caratteri) del valore della variabile `<var>` (NOTA: la lunghezza è comunque una stringa)

– Esempio:

```
bash:~$ A=armadillo
```

```
bash:~$ echo ${#A}
```

```
9
```

```
bash:~$ echo ${A:${#A}-4)}
```

```
illo
```

```
bash:~$ B=${A:3:3}
```

```
bash:~$ echo ${#B}      -- $B=adi
```

```
3
```

```
bash:~$
```

Pattern matching

- È possibile selezionare parti del valore di una variabile sulla base di un pattern (modello)
- I pattern possono contenere *,?, e [] e sono analoghi a quelli visti per l'espansione di percorso
- Occorrenze iniziali
 - `${<var>#<pattern>}`
 - `${<var>##<pattern>}`
 - se `<pattern>` occorre all'inizio di `$<var>` ritorna la stringa ottenuta eliminando da `$<var>` la più corta / la più lunga occorrenza *iniziale* di `<pattern>`

Pattern matching (2)

- Occorrenze finali

`${<var>%<pattern>}`

`${<var>%%<pattern>}`

se `<pattern>` occorre alla fine di `$<var>` ritorna la stringa ottenuta eliminando da `$<var>` la più corta / la più lunga occorrenza *finale* di `<pattern>`

- esempi:

- `outfile=${infile%.pcx}.gif`

- rimuove l'eventuale estensione `.pcx` dal nome del file (in `infile`) e ci aggiunge `.gif` (`pippo.pcx` → `pippo.gif`)

Pattern matching (3)

- Esempi (cont):

- `basename=${fullpath##*/}`

- rimuove dal `fullpath` il prefisso più lungo che termina con `'/'` (cioè estrae il nome del file dal path completo)

- `dirname=${fullpath%/*}`

- rimuove dal `fullpath` il suffisso più corto che inizia per `'/'` (cioè estrae il nome della directory dal path completo)

```
bash:~$ fullpath=/home/s/susanna/myfile.c
```

```
bash:~$ echo ${fullpath##*/}
```

```
myfile.c
```

```
bash:~$ echo ${fullpath%/*}
```

```
/home/s/susanna
```

Pattern matching (4)

- Esempi (cont):
 - **SCRIPTNAME=\${0##*/}**
 - Seleziona dal pathname dello script in esecuzione il nome del file
 - Può essere utile per parametrizzare i messaggi stampati es:
echo "\${SCRIPTNAME}: Error"

Sostituzione di sottostringhe

- È possibile sostituire le occorrenze di un pattern nel valore di una variabile

`${<var>/<pattern>/<string>}`

`${<var>//<pattern>/<string>}`

- l'occorrenza più lunga di **pattern** in **var** è sostituita con **string**.
- La prima forma sostituisce solo la prima occorrenza, la seconda le sostituisce tutte
- se **string** è vuota le occorrenze incontrate sono eliminate
- se il primo carattere è **#** o **%** l'occorrenza deve trovarsi all'inizio o alla fine della variabile
- se **var** è ***** o **@** l'operazione è applicata ad ogni parametro posizionale, e viene ritornata la lista risultante

Sostituzione di sottostringhe (2)

- Esempi:

```
bash:~$ echo $A
```

```
unEsempioDiSostituzione
```

```
bash:~$ echo ${A/e/eee}
```

```
unEseeeempioDiSostituzione
```

```
bash:~$ echo ${A//e/eee}
```

```
unEseeeempioDiSostituzioneeee
```

```
bash:~$ echo ${A/%e/eee}
```

```
unEsempioDiSostituzioneeee
```

```
bash:~$ ${A/#*n/eee}
```

```
eeeEsempioDiSostituzione
```

```
bash:~$
```

Esempio: **pushd** **popd**

- **pushd** e **popd**

- sono dei builtin che implementano uno stack di directory

- pushd** **<dir>** cambia la working directory in **<dir>** e mette **<dir>** sullo stack

- popd** elimina la directory sul top dello stack e si sposta nella nuova directory top

```
bash:~$ pushd didattica
```

```
/home/s/susanna/didattica /home/s/susanna
```

```
bash:~/didattica$ popd
```

```
/home/s/susanna
```

```
bash:~$
```

Esempio: pushd popd (2)

- Discutiamo come implementarli come funzioni bash:

- una prima soluzione

#DIRSTACK variabile che implementa lo stack

```
function pushd ()
```

```
{
```

```
    DIRNAME=${1:?}"missing directory name"}
```

```
    cd $DIRNAME &&
```

```
    DIRSTACK="$PWD ${DIRSTACK:-$OLDPWD}"
```

```
    echo $DIRSTACK
```

```
}
```

Esempio: pushd popd (3)

#DIRSTACK variabile che implementa lo stack

```
function popd ()
```

```
{
```

```
    DIRSTACK=${DIRSTACK#* }
```

```
    cd ${DIRSTACK%% *} }
```

```
    echo $PWD
```

```
}
```

Esempio: `pushd` `popd` (4)

- Problemi:
 - non gestisce o gestisce male errori come
 - ‘directory non esistente’
 - stack vuoto (nella `popd`)
 - non permette di trattare directory che hanno uno spazio nel nome
 - l’implementazione dei builtin veri è molto più complessa

Esempio: pushd popd (5)

- Esempio: miglioriamo la pushd

```
function pushd ()
{
  DIRNAME=${1:? "missing directory name"}
  if cd $DIRNAME; then
    DIRSTACK="$PWD ${DIRSTACK:-$OLDPWD}"
    echo $DIRSTACK
  else
    echo "Error still in $PWD"
  fi
}
```

Esempio: overloading

- Esempio: ridefiniamo `cd`

```
function cd ()
{
    builtin cd "$@"
    es=$?
    echo "cd: $OLDPWD --> $PWD"
    return $es
}
```

Esempio: overloading (2)

- Esempio: ridefiniamo cd

```
function cd ()  
{  
    builtin cd "$@"  
    es=$?  
    echo "cd: $OLDPWD --> $PWD"  
    return $es  
}
```

Richiede l'esecuzione di un builtin specifico (altrimenti chiamerebbe la funz. che stiamo definendo)

Da esplicitamente l'*exit status* del cd
Altrimenti è quello dell'ultimo comando eseguito (echo)

Esempio: mygunzip

```
#!/bin/bash
file=$1
if [ -z "$file" ] || ! [ -e "$file" ] ; then
    echo "Usage: mygunzip filename"
    exit 1
else
    ext=${file##*.} #determina il suffisso
    if ! [ $ext = gz ] ; then
        mv $file $file.gz
        file=$file.gz #se non è gz lo aggiunge
    fi
    gunzip $file
fi
```

Test - Operatori logici

- Diverse condizioni su stringhe e file possono essere combinate all'interno di un test tramite gli *operatori logici* :

-a (and) -o (or) ! (not)

- All'interno di una condizione (**test 0 [...]**) la sintassi è

\(expr1 \) -a \(expr2 \)

\(expr1 \) -o \(expr2 \)

! expr1

Operatori logici : esempi

- Miglioriamo ancora la pushd

```
function pushd ()
{ DIRNAME=$1
if [ -d "$DIRNAME" -a
  -x "$DIRNAME" ]; then
  cd $DIRNAME;
  DIRSTACK="$PWD ${DIRSTACK:-$OLDPWD}"
  echo $DIRSTACK
else
  echo "Error still in $PWD"; return 1
fi }
```

Test - Interi

– Si possono effettuare test su stringhe interpretate come valori interi

`-lt` (minore)

`-gt` (maggiore)

`-le` (min uguale)

`-ge` (maggiore uguale)

`-eq` (uguale)

`-ne` (diverso)

– Attenzione, questi test sono utili solo per mescolare test su stringhe e interi, altrimenti `$ ((<cond>))` è più efficiente ed espressivo

- include =, <, >, >=, <=, ==, !=, &&, ||

Esempio: change

- Es.

change <old> <new>

- ridenomina ogni file con suffisso ‘.<new>’ nella directory corrente sostituendo il suffisso con ‘.<old>’

```
bash:~$ ls
```

```
h.txt g.fig r.txt
```

```
bash:~$ change txt txtnew
```

```
bash:~$ ls
```

```
h.txtnew g.fig r.txtnew
```

```
bash:~$
```

Esempio: change (2)

```
#!/bin/bash  
OLD=$1  
NEW=$2  
for FILE in *.$OLD ; do  
    mv $FILE ${FILE%$OLD}.$NEW  
done
```

Esempio: mylsR

- Es. funzione `mylsR`

- si comporta come `ls -R`
- discende ricorsivamente le directory fornite come argomento evidenziandone la struttura

```
bash:~$ mylsR
```

```
dir1
```

```
  subdir1
```

```
    file1
```

```
    ...
```

```
  subdir2
```

```
    subdir3
```

```
      file2
```

```
...
```

Esempio: mylsR (2)

- Vediamo prima una funzione analoga a `ls -R` senza strutturazione

```
function tracedir () {  
  for file in "$@" ; do  
    echo $file  
    if [ -d "$file" ]; then  
      cd $file  
      tracedir $(command ls)  
      cd ..  
    fi  
  done
```

Esempio: mylsR (3)

È ricorsiva!!!!

```
function tracedir () {  
  for file in "$@" ; do  
    echo $file  
    if [ -d "$file" ]; then  
      cd $file  
      tracedir $(command ls)  
      cd ..  
    fi  
  done
```

Esegue il comando ls e non
eventuali funzioni

Esempio: mylsR (4)

```
function mylsR () {  
  singletab= "\t"  
  for file in "$@" ; do  
    echo -e $tab$file  
    if [ -d "$file" ]; then  
      cd $file  
      tab=$tab$singletab  
      mylsR $(command ls)  
      cd ..  
      tab=${tab%"\t"}  
    fi  
  done
```

Esempio: mylsR (5)

```
function mylsR () {  
  singletab= "\t"  
  for file in "$@" ; do  
    echo -e $tab$file  
    if [ -d "$file" ]; then  
      cd $file  
      tab=$tab$singletab  
      mylsR $(command ls)  
      cd ..  
      tab=${tab%"\t"}  
    fi  
  done
```

Con `-e` echo interpreta `\t` come un carattere di escape TAB

Esempio: ancora cd

- La funzione

cd old new

- che con 1 o 0 parametri si comporta come il builtin **cd**
- mentre con 2 parametri cerca nel pathname della directory corrente la stringa **old**, se la trova la sostituisce con **new** e cerca di spostarsi nella directory corrispondente

Esempio: ancora cd (2)

```
function cd () {
    case "$#" in
        0|1 ) builtin cd $1;;
        2   ) newdir="${PWD}/${1}/${2}"
              case "$newdir" in
                  $PWD) echo "bash: cd: bad \
                          substitution" 1>&2
                      return 1;;
                  * ) cd $newdir;;
                  esac;;
        *   ) echo "bash: cd: too many args" 1>&2
              return 1;;
    esac }

```

Esempio: icd

- La funzione

icd

- che elenca le directory presenti in quella corrente
- e a scelta dell'utente si sposta in una di queste

Esempio: icd (2)

```
function icd () {  
PS3="Scelta?"  
select dest in $(command ls -aF | grep "/"); do  
    if [ $dest ]; then  
        cd $dest  
        echo "bash; icd; Changed to $dest"  
        break  
    else  
        echo "bash; icd; wrong choice"  
    fi  
done  
}
```

C'è molto di più....

- Si possono trattare opzioni sulla riga di comando (builtin **shift**, **getopts**)
- è possibile definire array
- è possibile leggere l'input dell'utente (builtin **read**)
- è possibile costruire comandi all'interno dello script ed eseguirli (comando **eval**)
- e molto altro ...
-