versione 1.0
20121117

Sistemi Operativi

II Facoltà di Ingegneria - Cesena
a.a 2012/2013
docenti: Santi / Ricci

[modulo lab 1c]

SHELL SCRIPTING

SHELL PROGRAMMING

- La shell in quanto processore comandi è in grado di elaborare comandi prendendoli da un file, chiamato **file comandi**
  - Un file comandi è un semplice file di testo, che contiene un programma che segue la grammatica del linguaggio di programmazione della shell
  - Il file comandi viene letto ed interpretato / eseguito dalla shell

- In realtà ogni file comandi è di per sé un programma sorgente (**shell script**) scritto nel linguaggio delle shell, con statement per il controllo di flusso, variabili e passaggio parametri

- Lo scripting language delle shell è utilizzato come linguaggio prototipale per rapido sviluppo di piccole applicazioni / macro comandi, di solito ottenute dalla composizione di comandi / programmi esistenti

- A seconda delle shell si hanno statement diversi: qui faremo riferimento alla Bourne shell (`/bin/sh`)

SPECIFICA DELL'INTERPRETE

- Di solito un file comandi ha estensione pari al nome della shell: nel nostro caso `.sh`

- Dato un file comandi `XXX.sh`, per mandarlo in esecuzione è necessario prima rendere eseguibile il file, poi lanciarlo:

  ```
  chmod +x XXX.sh ; XXX.sh
  ```

  (il punto e virgola serve per impartire comandi in sequenza). Oppure si può invocare direttamente una shell che lo esegua:

  ```
  sh XXX.sh
  ```

- La prima linea di uno script deve essere per convenzione

  ```
  #!<Nome Interprete>
  ```

  quindi nel nostro caso

  ```
  #!/bin/sh
  ```

  In questo modo si specifica quale interprete usare per interpretare lo script

VARIABILI

- Anche negli script è possibile definire / accedere variabili come visto in precedenza
  - per accedere il valore di una variabile `VAR` si può usare sia la notazione `$VAR`, sia `${VAR}`
    - La seconda torna utile quando si specificano variabili attaccate a testo

- Le variabili possono essere locali (default) oppure definite a livello di ambiente
  - Per estendere lo scope di una variabile all'ambiente si deve usare il comando export

```
CLASSPATH=/usr/classes
export CLASSPATH
```

PASSAGGIO DEI PARAMETRI

- Gli argomenti passati ad un file comandi sono contenuti per default nelle variabili **$N**, dove N è un intero che rappresenta l'indice dell'argomento (ovvero si dice che gli argomenti sono variabili posizionali nella linea di invocazione):
  - **$0** rappresenta il nome del file comandi
  - **$1** rappresenta il primo argomento...e così via
- Esempio (file comandi `dir.sh` che vuole un parametro)

```
#!/bin/sh
echo argomento: $1
ls $1
```

- Altre variabili significative:
  - **$*** rappresenta l'insieme di tutte le variabili posizionali, ovvero l'insieme di tutti gli argomenti `$1`,`$2`,... passati (`$0` escluso)
  - **$#** contiene il numero degli argomenti passati (`$0` escluso)
  - **$?** valore (intero) dell'ultimo comando eseguito
  - **$$** identificatore numerico del processo in esecuzione (PID)

COMANDI BUILT-IN (1/2)

- I comandi builtin sono comandi che non corrispondono a nessun programma nel file system (in `/usr/bin`), ma sono forniti direttamente dall'interprete
  - L'utilizzo di un `;` fra comandi evita che venga creata una nuova subshell per l'esecuzione dei comandi
- Tra i comandi builtin (in parte già visti):
  - `cd <Dir>` (cambia directory),
  - `pwd` (stampa dir corrente)
  - `echo <Args>` (stampa in standard output gli argomenti)
  - `eval <Args>` (valutazione degli argomenti)
  - `. <FileName>` (legge il file specificato e lo interpreta)
  - `:` (comando 'nullo': non fa nulla, ritorna 0)
  - `test <Expr>` (valuta l'espressione booleana e ritorna 0 se vera)
  - `shift` (sposta gli argomenti di una posizione, mangiando il primo)
  - `exit <N>` (uscita dalla shell con il valore di ritorno N)
  - `set <Arg>` (setta gli argomenti della linea di comando senza `Arg`, visualizza l'elenco delle variabili)

COMANDI BUILT-IN (2/2)

- Comandi builtin (continua)
  - `read var1 var2`... (lettura dalla standard input)
  - `exec <Command>` (esegue il comando, rimpiazzando la shell corrente)
  - `kill -<SIG> %<JOB>` (invia segnale SIG al JOB specificato)
  - `wait  <Pid>` (aspetta la terminazione del processo specificato)
  - `trap <Cmd Sig>` (esegue il comando `Cmd` se il segnale `Sig` è inviato alla shell)
  - `bg [<Pid>], fg[<Pid>]` (rende il processo specificato ad essere in background o in foreground)

ALTRO ESEMPIO (`test1.sh`)

```
#!/bin/sh
# this is a comment
echo The number of arguments is $#
echo The arguments are $*
echo The first is $1
echo My process number is $$
echo Type something from the keyboard:
read something
echo You typed $something
```

COMANDO TEST (1/2)

- Il comando per la verifica di una condizione è test:

  `test -``<Opzioni> <target>`

- Restituisce uno stato uguale a zero (significa che la condizione è true) o diverso da zero (è falsa).
  - **Da notare che ha una convenzione opposta a quella del C**
- Tra i tipi di test possibili abbiamo:
  - `test -e <NomeFile>   # true se f (file o directory) esiste`
  - `test -f <NomeFile>   # true se il file f esiste`
  - `test -d <NomeDir>    # true se la directory specificata esiste`
  - `test -r <NomeFile>   # true se il file è leggibile (-w scrivibile, -x eseguibile)`
  - `test -n <Stringa>   # true se la lunghezza della stringa è > 0`
  - `test -z <stringa>   # true se la stringa è nulla`
  - `test <stringa>      # true se la stringa non è nulla`
  - `test <String1> = <String2>    # true se stringhe sono uguali`

COMANDO TEST (2/2)

- (tipi di test...) :
  - `test <String1> != <String2> # true se le stringhe sono diverse`
  - `test <Numero1> [-eq -ne -gt -lt –ge -le] <Numero2>  # confronto fra stringhe numeriche`
  - `test <Condizione> [ -a -o ] <Condizione>  # per espressioni booleane di and e or`
  - `test ! <Condizione> # not booleano`

- NOTE:
  - gli spazi attorno a = **sono necessari**.
  - String1 e String2 possono contenere metacaratteri

- Equivalente a `test` è `[ ]`:
  - `test $a = $b` è equivalente a `[ $a = $b ]`

VALUTAZIONE ESPRESSIONI (`test2.sh`)

- Il comando **expr** forza la valutazione di espressioni interpretate come espressioni aritmetiche, producendo un valore

- Esempi:

```
expr 1 + 2 + 3
x=303; y="$x + 2";echo $y
expr $y
```

- Cosa viene stampato in stdout?

VALORE DI RITORNO DEGLI STATEMENT

- Come per i comandi, ogni statement in uscita restituisce un valore di stato, che indica il completamento o meno del comando.

- Tale valore di uscita è posto nella variabile **`$?`**
  - `$?` può essere utilizzato in espressioni e per il controllo di flusso:
    - Se lo stato vale zero, il comando è andato a buon fine
    - Se contiene un valore positivo, significa che c'è stato un errore

COSTRUTTO DI SELEZIONE `if`

- Costrutto di selezione:

```
if <Condizione> ; then
  <Comandi>
elif <Condizione> ; then
  <Comandi>...
else
  <Comandi>
fi
```

- La condizione viene espressa generalmente con `test` o `[ ]`

`If`: ESEMPIO 1 (`test_if1.sh`)

- Esempio 1: verifica che il terzo argomento passato in input sia la somma dei primi due:

```
#!/bin/sh
if test $# -ne 3; then
  echo Errore: servono 3 argomenti
else
  sum=`expr $1 + $2`
  if test $sum -eq $3; then
    echo si
  else
    echo no
  fi
fi
```

`If`: ESEMPIO 2 (`test_if2.sh`)

- Esempio 2: check nome utente...

```
#!/bin/sh
myname=`whoami`
if [ $myname = root ]; then
  echo Welcome
else
  echo You must be root to run the $0 script
  exit 1
fi
```

If: ESEMPIO 3 (test_if3.sh)

- Come condizione può esser specificato qualsiasi comando: se restituisce zero (come exit status) la condizione è vera, altrimenti è considerata falsa

```
#!/bin/sh
if test $# -ne 1; then
  echo Errore: serve un argomento
else
  if grep -w $1 /etc/passwd > /dev/null ; then
    echo $1 ha un account
  else
    echo $1 non ha un account
  fi
fi
```

COSTRUTTO DI SELEZIONE MULTIPLA

- Il costrutto per alternative multiple è **`case`**, la cui sintassi è:

```
case <Espressione> in
  <Pattern-1> )  <Comandi> ;;
  <Pattern-2> )  <Comandi> ;;
  <Pattern-3> | <Pattern-4> | <Pattern-5> ) <comandi> ;;
  ...
esac
```

- L'espressione è una stringa, generalmente contenuta in una variabile o risultato di una espressione backquoted

`case:` ESEMPIO (`test_case.sh`)

- Esempio:

```
#!/bin/sh
read answer
case $answer in
  Y* | y* | S* | s* ) echo YES;;
  N* | n* ) echo NO ;;
  *) echo UNKNOWN;;
esac
```

RIPETIZIONI ENUMERATIVE: `for`

- Il costrutto per le ripetizioni enumerative è il **`for`**:

```
for <Var>  [  in <List> ]; do
  <Comandi>
done
```

- Dove `<List>` è una lista di stringhe

- Il costrutto quindi scandisce gli elementi della lista `<List>` e ripete il ciclo per ogni stringa presente nella lista
  - Esempio:

```
#!/bin/sh
for animale in cane gatto topo; do
  echo $animale
done
```

`for`: ESEMPI (1/2)

- Per specificare la lista di stringhe è possibile usare metacaratteri, facendo riferimento a file nel file system. Esempio: (visualizza tutti i file nella directory corrente):

```
#!/bin/sh
for i in *; do
  echo $i
done
```

- Se `<List>` è omesso, si considera come lista di stringhe la lista degli argomenti (`$*`). Esempio `test_for1.sh`, visualizzazione degli argomenti, con il loro indice:

```
#!/bin/sh
j=0
for i ; do
  echo argument $j is $i
  j=`expr $j + 1`
done
```

`for`: ESEMPI (2/2) (`test_for2.sh`)

- Esempio: itera su tutti i file della directory corrente cercando il file con dimensioni maggiori, il cui nome viene stampato in stdout

```
#!/bin/sh
max=0
for file in * ; do
  if test -f $file; then
    size=`cat $file | wc -c`
    if test $size -gt  $max; then
      max=$size
      maxfile=$file
    fi
  fi
done
echo Il File con dimensione maggiore nella directory
corrente è: $maxfile - dimensione $max caratteri
```

RIPETIZIONI NON ENUMERATIVE: `while`

- Per le ripetizioni non enumerative si usa il costrutto **`while`**

```
while <Condizione>; do
  <Comandi>
done
```

- Si ripete il ciclo per tutto il tempo che `<Condizione>` è zero (true): termina quando tale valore diventa false

while: ESEMPIO (test_while.sh)

```
#!/bin/sh
# cicla fino alla
# comparsa di un file di nome $1
while test ! -f $1 ; do
  sleep 1
done
echo $1 appeared!
```

- Lanciare lo script in parallelo alla shell con

```
$ sh test_while.sh  pippo.dat &
```

- poi creare il file pippo.dat mediante:

```
$ touch pippo.dat
```

RIPETIZIONI NON ENUMERATIVE: `until`

- Altro costrutto che permette ripetizioni non enumerative è until:

```
until <Condizione>; do
    <Comandi>
done
```

- Si ripete il ciclo fino a quando la condizione è diversa da zero (insuccesso, false)
- Esempio:

```
#!/bin/sh
# wait-for-user.sh
until  who | grep $1 ; do
  sleep 1
done
echo User $1 logged in!
```

ESEMPIO SCRIPT PER RIMOZIONE RICORSIVA DIRECTORY

- `test_del_dir.sh <DirSource> <DirName>`
  - Invocare lo script: ./`test_del_dir.sh base_del_dir del_dir`

```
#!/bin/sh
case $# in
  2)  ;;
  *)  echo "Wrong no. parameters: <DirSource> <DirName>";
        exit 1;;
esac
find "$1" -name "$2" |
while read dir
  do
    if [ -d $dir ]; then
      echo removing directory: $dir
      rm -fR $dir
    fi
done
```

ALTRO ESEMPIO: CAT, WHILE E READ

- `readpipe.sh`

```
#!/bin/sh
# readpipe.sh
# This example contributed by BjonEriksson.

cat $0 |
while read line
  do
    echo "{$line}"
  done
exit 0 # End of code.
```

FUNZIONI (`myfunc.sh`)

- In uno script è possibile definire funzioni nel seguente modo:

```
<nome funzione>(){
   comandi
}
```

  - I parametri passati alla funzione vengono identificati dalle variabili `$N`, con N in [1...NArgs], `$@` contiene la lista intera
  - Si richiama specificando il nome, seguita dai parametri
- Esempio di script con definizione e uso funzione `myfunc`:

```
#!/bin/sh

myfunc() {
  echo "Function called.. $@"
  i=0
  for arg in $@
  do
    echo arg $i: $arg
    i=`expr $i + 1`
  done
}
myfunc pippo paperino
myfunc pluto
```

YATTA (yatta.sh)

```
#!/bin/bash
# by Marco Fabbri (mrfabbri@gmail.com)
#jump
#
# \0/
# _|_
#
#
#stand
#  0
# /|\
# / \
#
yattaa(){
  clear;
  head -n12 $1 | tail -n3 ;
  sleep 1;
  clear;
  echo -e '# yattaaa!!!\n#\n#';
  sleep 1;
  clear;
  head -n7 $1 | tail -n3 ;
  sleep 1;
}
yes | while read i;
do yattaa $0;
done;
```

ESERCIZIO SULLE FUNZIONI (FORK BOMB 1/2)

- Nello script ad una linea seguente si definisce ed invoca una funzione... qual è l'effetto dell'esecuzione ?

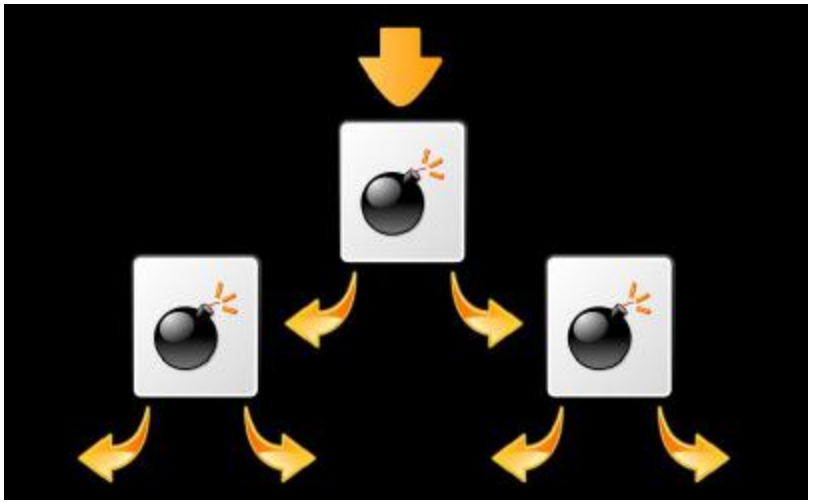
```
$ :(){ :|:& };:
```

($ si suppone sia il prompt della shell)

ESERCIZIO SULLE FUNZIONI (FORK BOMB 2/2)

- In forma più leggibile:

```
bomb ( ) {
   bomb | bomb &
};
bomb
```



TRATTAMENTO DEI SEGNALI

- La shell mette a disposizione comandi sia per inviare, sia per reagire all'arrivo di segnali, che manifestano l'occorrenza di eventi asincroni
- Per specificare quali comandi eseguire inseguito all'arrivo di un segnale:

  trap <Comandi> <Segnale>
- All'arrivo del segnale <Segnale> vengono eseguiti i comandi specificati
  - I segnali sono identificati con il loro valore numerico: 0 fine del file, 2 <CTRL-C> (interrupt da tastiera), 3 <CTRL-Z> (stop da tastiera), 15 kill...
- Esempi:

  `trap 'rm /tmp/*.*;  exit' 2`

  All'arrivo del segnale 2 (interruzione), si ripulisce il direttorio tmp

  `trap 'ls; exit' 15`

  All'arrivo del segnale kill, viene eseguito il listing dei file contenuti nel direttorio corrente e poi si esce
- Per ignorare un segnale allora:

  `trap '' 2`

  Il segnale di interruzione viene ignorato

PROGETTAZIONE DI UNO SCRIPT

- Definire l'interfaccia dello script (file comandi), in particolare dei parametri passati
  - Controllo degli argomenti (fondamentale)

- Individuare l'insieme dei comandi / filtri disponibili utili per risolvere il problema, precisando il loro flusso di informazioni standard input / standard output

- Realizzare la 'colla' algoritmica fra i comandi e i flussi

ESERCIZIO #1

- Elencare utenti presenti nel sistema

```
#! /bin/sh

## elenca gli utenti presenti nel sistema

echo `who`
```

ESERCIZIO #2 1/2 (`fileAppend.sh`)

- Progettare uno script che permetta di fare l'append di due file ad un terzo

```
#!/bin/sh
# file Append
#  concatena il contenuto di due file ad un terzo

case $# in
  0)  echo "[Usage:] fileAppend srcUno srcDue dest"; exit 2;;
  1)  echo "[Usage:] fileAppend srcUno srcDue dest"; exit 2;;
  2)  echo "[Usage:] fileAppend srcUno srcDue dest"; exit 2;;
  3)  echo "Append started" ;;
  *)  echo "[Troppi argomenti, ne bastano 3]"; exit 3;;
esac
...
```

ESERCIZIO #2 2/2 (`fileAppend.sh`)

```
...

if test ! -f $1; then
  echo "Il primo argomento deve essere un file";
  echo "[Usage:] fileAppend srcUno srcDue dest";
  exit 3;
elif test ! -f $2; then
  echo "Il secondo argomento deve essere un file";
  echo "[Usage:] fileAppend srcUno srcDue dest";
  exit 3;
else
  more  $1 >> $3 ; more  $2 >> $3 ;
fi
echo "Append finished";
exit 0;
```

ESERCIZIO #3 (`userIn1.sh`)

- Realizzare uno script che visualizzi le sessioni attive di un dato utente nel sistema

```
#!/bin/sh
## se un dato user è loggato nel sistema
## elenca le sue sessioni attive le shell

# controllo sugli argomenti

case $# in
  0)  echo "[Usage:] userIn1 <UserName>"; exit 2;;
  1)  echo "Started" ;;
  *)  echo "[Num argomenti non valido]"; exit 3;;
esac


echo "Session(s) Found for $1:"
echo `who | grep $1`
```

ESERCIZIO #4 (`userIn2.sh`)

- Realizzare uno script che visualizzi le sessioni attive di un dato utente nel sistema, aggiornate ogni 10 secondi

```
#! /bin/sh
# userIn nameUser

# controllo sugli argomenti
case $# in
  0)  echo "[Usage:] userIn2 NameUser "; exit 1;;
  1)  echo "Started " ;;
  *)  echo "[Numero arg. non valido]"; exit 2;;
esac
while : ; do
  who | grep $1;
  sleep 10;
done
```