LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica [0514G] - 2 - Scienze e Tecnologie Informatiche (LT) andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

Espansione aritmetica

- La sintassi base per una subshell è da non confondere con l'espansione aritmetica che utilizza le doppie parentesi tonde.

- All'interno delle doppie parentesi tonde si possono rappresentare varie espressioni matematiche inclusi assegnamenti e confronti.

Alcuni esempi:

```
(( a = 7 )) (( a++ )) (( a < 10 )) (( a = 3<10?1:0 ))
```

Confronti logici - costrutti

I costrutti fondamentali per i confronti logici sono il comando `test` e i raggruppamenti tra parentesi quadre singole e doppie: `test … , [ … ], [[ … ]]`

- `test …` e `[ … ]` sono *built-in* equivalenti
- `[[ … ]]` è una coppia di *shell-keywords*

In tutti i casi il blocco di confronto genera il codice di uscita 0 in caso di successo, un valore differente (tipicamente 1) altrimenti.

NOTA. *built-in* e *shell-keywords*: i *builtins* sono sostanzialmente dei comandi il cui corpo d'esecuzione è incluso nell'applicazione shell direttamente (non sono eseguibili esterni) e quindi seguono sostanzialmente le "regole generali" dei comandi, mentre le *shell-keywords* sono gestite come marcatori speciali così che possono "attivare" regole particolari di parsing. Un caso esemplificativo sono gli operatori "<" e ">" che normalmente valgono come redirezionamento, ma all'interno di `[[ … ]]` valgono come operatori relazionali.

Confronti logici - tipologia operatori

Le parentesi quadre singole sono POSIX-compliant, mentre le doppie sono un'estensione bash. Nel primo caso gli operatori relazionali "tradizionali" (*minore-di*, *maggiore-di*, etc.) non possono usare i termini comuni (<, >, etc.) perché hanno un altro significato (*) e quindi se ne usano di specifici che però hanno un equivalente più "tradizionale" nel secondo caso.

Gli operatori e la sintassi variano a seconda del tipo di informazioni utilizzate: una distinzione sottile c'è per confronti tra stringhe e confronti tra interi.

(*) salvo eventualmente utilizzare il raggruppamento con doppie parentesi tonde per le espansioni aritmetiche

Confronti logici - interi e stringhe

interi		
	`[ … ]`	`[[ … ]]`
uguale-a	`-eq`	`==`
diverso-da	`-ne`	`!=`
minore-di minore-o-uguale-a	`-lt` `-le`	`<` `<=`
maggiore-di maggiore-o-uguale-a	`-gt` `-ge`	`>` `>=`

stringhe		
	`[ … ]`	`[[ … ]]`
uguale-a	= o ==	
diverso-da	!=	
minore-di (ordine alfabetico)	`\<`	`<`
maggiore-di (ordine alfabetico)	`\>`	`>`
nota: occorre lasciare uno spazio prima e dopo i “simboli” (es. non “=” ma “ = “)		

Confronti logici - operatori unari

Esistono alcuni operatori unari ad esempio per verificare se una stringa è vuota o meno oppure per controllare l'esistenza di un file o di una cartella.

Alcuni esempi:

`[[ -f /tmp/prova ]]` : è un file?

`[[ -e /tmp/prova ]]` : file esiste?

`[[ -d /tmp/prova ]]` : è una cartella?

Confronti logici - negazione

Il carattere “!” (punto esclamativo) può essere usato per negare il confronto seguente.

Alcuni esempi:

```
[[ ! -f /tmp/prova ]]

[[ ! -e /tmp/prova ]]

[[ ! -d /tmp/prova ]]
```

ESERCIZI - 1

Scrivere delle sequenze di comandi (singola riga da eseguire tutta in blocco) che utilizzano come “input” il valore della variabile DATA per:

1. Stampa “`T`” (per True) o “`F`” (per False) a seconda che il valore rappresenti un file o cartella esistente
2. Stampa “`file`”, “`cartella`” o “`?`” a seconda che il valore rappresenti un file (esistente), una cartella (esistente) o una voce non presente nel file-system
3. Stampa il risultato di una semplice operazione aritmetica (es: ‘$1 < 2$’) contenuta nel file indicato dal valore di DATA, oppure “`?`” se il file non esiste

SCRIPT/BATCH

È possibile raccogliere sequenze di comandi in un file di testo che può poi essere eseguito:

- Richiamando il tool “bash” e passando il file come argomento
- Impostando il bit “x” e:
  - specificando il path completo (relativo o assoluto)
  - Indicando il solo nome se il percorso è presente in $PATH

Esempio 1 - subshell e PID

SCRIPT “bashpid.sh”:

```
# bashpid.sh
echo $BASHPID
echo $( echo  $BASHPID)
```

CLI:

```
chmod +x ./bashpid.sh ; echo $BASHPID ; ./bashpid.sh
```

Elementi particolari negli SCRIPT

Le righe vuote e i commenti sono ignorati.

I commenti sono porzioni di righe che cominciano con “#” (non in una stringa)

La prima riga può essere un metacommento (detto hash-bang, she-bang e altri nomi simili): `#!application [opts]` che identifica un’applicazione cui passare il file stesso come argomento (tipicamente usato per identificare l’interprete da utilizzare)

Sono disponibili variabili speciali in particolare $@, $# e $0, $1, $2, ...

Alri costrutti

For loop:

```
for i in ${!lista[@]}; do
    echo ${lista[$i]}
done
```

While loop:

```
while [[ $i < 10 ]]; do
    echo $i ; (( i++ ))
done
```

If condition:

```
if [ $1 -lt 10 ]; then
    echo less than 10
elif [ $1 -gt 20 ]; then
    echo greater than 10
else
    echo between 10 and 20
fi
```

Esempio 2 - argomenti

SCRIPT “args.sh”:

```
#!/usr/bin/env bash
nargs=$#
while [[ $1 != "" ]]; do
    echo "ARG=$1"
    shift
done
```

CLI:

```
chmod +x ./args.sh
./args.sh uno
./args.sh uno due tre
```

ESERCIZI - 2

- Scrivere uno script che dato un qualunque numero di argomenti li restituisca in output in ordine inverso.
- Scrivere uno script che mostri il contenuto della cartella corrente in ordine inverso rispetto all'output generato da "`ls`" (che si può usare ma senza opzioni)

CONCLUSIONI

L’utilizzo di BASH - tramite CLI o con SCRIPT - è basilare per poter interagire attraverso comandi per l’uso del file-system e delle altre risorse e per poter invocare tools e applicazioni.

Esistono (numerose) alternative ed è possibile anche sfruttare più strumenti in cooperazione (ad esempio scripts con interpreti differenti).