

Sistemi Operativi, Secondo Modulo, Canale A–L
e Teledidattica
Riassunto della lezione del 29/03/2021

Igor Melatti

Gli Script Bash

- Come detto, sono file di testo che contengono comandi bash
 - gli stessi che possono essere scritti anche sulla shell interattiva
 - il comando `exit n` fa sì che l'exit code dell'esecuzione di uno script sia `n` (tra 0 e 255)
 - i commenti si fanno con il `#`, analogamente al `//` del Java (quindi, da dove si trova fino all'andata a capo)
 - eccezione 1: nell'espansione di parametri (vedere lezione 10)
 - eccezione 2 (*shabang*): nella prima riga dello script, se seguito dal carattere `!`, ciò che segue viene interpretato come il programma (l'interprete, da fornire col percorso *assoluto*) tramite il quale lanciare lo script stesso (questo solo se lo si rende eseguibile e lo si lancia direttamente)
 - in assenza di tale riga, si usa la bash stessa
 - <https://it.wikipedia.org/wiki/Shabang>
 - lo si può usare per rendere eseguibile qualsiasi file di testo per qualsiasi interprete: ad esempio, Python, Perl, ...
 - non Java, perché non prende direttamente in input il file da eseguire
 - per `awk`, dato che per passargli un file con il programma occorre l'opzione `-f`, occorre scrivere `#!/usr/bin/awk -f`
 - formalmente, l'effetto è questo: eseguire il comando `cmd lista_args`, dove `cmd` ha una shabang `#!/interprete opt` ha l'effetto di eseguire `interprete opt cmd lista_args` (per essere massimamente portabile, `opt` dev'essere una sola opzione)
 - **esercizio:** trasformare tutti gli esercizi su `awk` e `sed` nelle lezioni 7 ed 8 in script eseguibili; si può fare anche con `uniq`?

- Comandi built-in e di sistema: distinguibili o dal `man` o più semplicemente con il comando (built-in!) `type`
 - provare ad esempio `type cd` e `type which`
- Come detto in lezione 2, la bash ha la `history`
 - interattivamente, la si può sfruttare con le frecce posizione su/giù, o con il `CTRL+r` per cercare una sottostringa
 - una volta trovato un comando precedente, lo si può modificare
 - tutto ciò lo si può fare anche da comando
 - *history expansion*: ovviamente, avviene prima di ogni altra expansion
 - `!n` viene espanso nell'`n`-esimo comando dall'inizio della shell (se `n` è positivo), o nell'`n`-esimo comando dato in precedenza (se negativo)
 - `!0` dà errore; `!-1` si può scrivere anche `!!`
 - in realtà, la `history` della bash può memorizzare un numero finito N (configurabile) di comandi, e una volta raggiunto il limite, l' $(N + 1)$ -esimo comando più vecchio viene cancellato dalla `history`, quindi `!1` non è necessariamente il primo comando dato...
 - comando (built-in) `history`, mostra tutti i comandi nella `history`, con il loro numero progressivo
 - `!prefix` trova ed esegue l'ultimo comando che iniziava con `prefix` (senza spazi...)
 - `!!:s/search/replace/` se l'ultimo comando che conteneva `search`, prima di eseguirlo sostituisci `search` con `replace` (si può fare la stessa cosa con `^search^replace ^`)
 - si può fare molto altro; come al solito, vedere `man bash`
 - **esercizio**: farsi stampare su `stdout` (senza eseguirli): i) il primo e l'ultimo comando della `history`; ii) il comando a metà della `history`, ma se contiene il comando `ls` sostituirlo con `stat`
- Le condizioni negli script (e nella bash interattiva)
 - sono usate nelle istruzioni condizionali `if-then` ed `if-then-else`, nonché nei cicli `while` ed `until`
 - si possono scrivere in due modi (quattro contandone anche altri 2 non standard): `[condizione]` (occhio agli spazi: uno dopo la parentesi aperta e uno prima di quella chiusa) e `test condizione`
 - si possono fare anche combinazioni logiche di condizioni: `-a` per l'AND, `-o` per l'OR, `!` per il NOT; per raggruppare, `\(e \)`
 - tutte le principali condizioni sono riportate in Tabella 1
 - formalmente, viene avviato un processo che effettua il controllo della condizione, e poi ritorna come exit status 0 (condizione vera) o 1 (condizione falsa); quindi le condizioni sono dei comandi veri e propri

Table 1: Condizioni in bash

Operatore	Operandi	Vero se...	Binario
-d p	pathname	p è una directory	unario
-e p	pathname	p esiste	unario
-f p	pathname	p è un file regolare	unario
-h p	pathname	p è un link simbolico	unario
-s p	pathname	p ha dimensione non nulla	unario
p1 -nt p2	pathname	p1 è più recente di p2	binario
p1 -ot p2	pathname	p2 è più recente di p1	binario
s1 == s2	string	s1 ed s2 sono uguali	binario
s1 != s2	string	s1 ed s2 sono diverse	binario
s1 \> s2	string	s1 è lessicograficamente maggiore di s2	binario
s1 \< s2	string	s1 è lessicograficamente minore di s2	binario
-z s	string	s ha lunghezza 0	unario
-n s	string	s ha lunghezza maggiore di 0; -n può anche essere omissso	unario
i1 -eq i2	integer	i1 è uguale ad i2	binario
i1 -ne i2	integer	i1 è diverso da i2	binario
i1 -gt i2	integer	i1 è maggiore di i2	binario
i1 -lt i2	integer	i1 è minore di i2	binario
i1 -ge i2	integer	i1 è maggiore o uguale a i2	binario
i1 -le i2	integer	i1 è minore o uguale a i2	binario

– niente vieta di scrivere comandi come `if ls > /dev/null; then echo ciao; fi`

– o comandi come `test -f file && echo il file esiste`

– o comandi come `test -f file: provare a dare poi il comando echo $?`

– sintassi alternativa 1: `[[`

* rispetto a `[`, cambiano alcune cosette: le combinazioni logiche sono come nel C (quindi con `&&`, `||` e `!`); i confronti si fanno con `<` e `>`, e si usa `==` anche tra interi

* ammette anche confronti con espressioni regolari: `[[expr =~ regex]]` (sono quelle estese, ma senza backreference)

– sintassi alternativa 2: `((`, vale solo per gli operatori aritmetici

– **esercizio:** Farsi stampare 1 se esistono due file di nome `file1.txt` e `file2.txt`, con il secondo più recente del primo, oppure se `file1.txt` è un link simbolico; altrimenti, farsi stampare 10 (usare solo comandi e/o espansioni visti fino a questo punto).

- Esecuzione condizionale: `if test-command; then commands1; else commands2; fi`

- l’**else** può non esserci
 - o può essere sostituito da un **elif**
 - qui come altrove: occhio ai punti e virgola, vanno esattamente lì
 - ogni punto e virgola può essere sostituito da un’andata a capo (nella shell interattiva, verrà atteso che si completi il comando)
 - si può andare a capo anche in alcuni punti dove non c’è il punto e virgola (ad esempio, dopo le keyword)
 - **commands** è un qualsiasi comando della shell (quindi anche una sequenza, condizionale o no, di comandi)
 - l’exit status dell’intero comando è 0 se nessun **test-command** è risultato vero, altrimenti è l’exit status dell’ultimo **commands** eseguito
 - **esercizio**: rifare l’esercizio precedente usando un **if**
- Esecuzione condizionale: `case word in listapattern1) commands1;; ...listapatternn) commandsn;; esac`
 - i pattern sono quelli del filename expansion
 - ci possono essere liste di pattern, separati da |
 - si esegue il comando del primo pattern che fa match
 - ciò permette di mettere come ultimo pattern *, ad indicare “qualsiasi altra cosa”
 - nota: i pattern non vanno quotati (né single né double), altrimenti hanno il loro valore letterale
 - **esercizio**: scrivere uno script che accetti 2 argomenti e 3 opzioni. Come di consueto, si supponga che le opzioni, introdotte da **-a**, **-b** e **-c**, vengano prima degli argomenti. Le opzioni **-a** e **-b** necessitano un argomento (per esempio, come l’opzione **-o** di **ps**), mentre **-c** è solo un flag (senza argomenti, come ad esempio l’opzione **-E** di **grep**). Come risultato, lo script deve avere il seguente output:

```
Opzione -a: A
Opzione -b: B
Opzione -c: C
Argomento 1: A1
Argomento 2: A2
```
- dove *A*, *B* e *C* possono essere o **assente** (se l’opzione non è stata data) oppure il valore passato all’opzione stessa (nel caso di *C*, dev’essere semplicemente **presente**). Se viene data un’opzione non tra quelle elencate, oppure gli argomenti non sono esattamente 2, occorre dare un messaggio d’errore. Provare a fare questo script sia usando quanto visto sin qui, sia usando il comando built-in **getopts** (vedere il **man bash**)

- Ciclo `until test-command; do commands; done`
 - `commands` viene eseguito tutte le volte che `test-command` fallisce (ovvero, torna qualcosa di diverso da 0), finché il test non esce con 0 (successo)
 - l'exit status dell'intero comando è 0 se nessun `commands` è stato eseguito, altrimenti è l'exit status dell'ultimo `commands` eseguito
 - **esercizio:** Scrivere uno script che prende 2 argomenti e copia il primo nel secondo. La copia deve avvenire solo se: i) il primo file esiste ed è accessibile in lettura; ii) il secondo file è un file con i permessi di scrittura oppure il secondo file è una directory con i permessi di scrittura. La copia deve avvenire tramite un ciclo `until` in cui la copia avviene al ritmo di una riga per iterazione, fino al completamento della copia. L'exit status dello script deve coincidere con quello del ciclo `until`. Verificare tale exit status nel caso in cui il primo file sia vuoto. Modificare poi lo script in modo da prendere un'opzione `-r r` (se non dato, $r = 1$): il ritmo di copiatura sarà ad r righe per iterazione.
- Ciclo `while test-command; do commands; done`
 - come sopra, ma questa volta `commands` è eseguito se il test ha successo, e si esce dal ciclo al primo fallimento
 - **esercizio:** rifare l'esercizio precedente, aggiungendo una quarta opzione (flag) `-w`. Se `-w` viene dato, allora dev'essere usato un ciclo `while`, altrimenti usare il ciclo `until` come sopra.
- Ciclo `for`, 2 sintassi:
 1. `for name [in listaparole]; do commands; done`
 - le parole nella lista di parole sono sempre separate da spazi, ma la lista stessa può essere il risultato di un word splitting
 - se la lista di parole non c'è, si intende essere "\$@"
 - viene eseguito `command` tante volte quante sono le parole in `listaparole` (attenzione al word splitting...); ogni volta, prima di eseguire `command`, alla variabile `name` viene assegnata una parola dalla lista, nell'ordine specificato
 - exit status: quello dell'ultimo comando in `commands`, oppure 0 se `listaparole` viene espansa alla lista vuota
 - **esercizio:** eseguire `stat` su ogni directory elencata in `PATH`.
 2. `for ((expr1; expr2; expr3)); do commands; done`
 - praticamente, come il C (e il Java)
 - `expr1` è l'inizializzazione (eseguita una sola volta, prima di eseguire `commands` la prima volta)

- `expr2` è la condizione da controllare prima di ogni esecuzione di `commands`
 - `expr3` è il comando da effettuare dopo ogni esecuzione di `commands`
 - in queste 3 espressioni, vale quanto detto per le espressioni aritmetiche (vedere lezione 13)
 - exit status: quello dell'ultimo comando in `commands`, oppure 1 se c'è qualche errore in una delle 3 espressioni
 - **esercizio:** rifare l'esercizio precedente, cambiando la quarta opzione in `-w w`. Se `w` è `while`, allora dev'essere usato un ciclo `while`, se è `until` usare il ciclo `until`, se è `for1` usare il ciclo `for` nella prima forma (difficile...), se è `for2` usare il ciclo `for` nella seconda forma, altrimenti dare un messaggio d'errore e non fare nulla.
- Le cattive abitudini restano: `break` e `continue`
 - `break` forza l'uscita da un ciclo
 - `continue` forza la continuazione di un ciclo (nel caso del `for` con 3 espressioni, viene prima eseguita `expr3`)
- Script interattivo
 - un po' ossimorico, ma c'è il comando `read {nomevar}`
 - mette in `nomevar` la stringa letta da tastiera (o meglio, da `stdin`)
 - se ci sono più variabili date come argomento, allora assegna ciascuna diversa parola ad una variabile
 - le parole sono separate da IFS
 - **esercizio:** Scrivere uno script che, senza usare `awk` o `sed`, legga un file (di testo) riga per riga; per ogni riga letta, sostituisca la parola `ciao` con `ehi`, e ogni numero con il suo successivo.
- Si possono anche dichiarare e chiamare delle funzioni
 - al solito, la sintassi è alquanto particolare: per la *dichiarazione*, occorre scrivere `function nomefunzione () { commands; }`
 - la parte tra parentesi graffe è esattamente un group command, con tutte le limitazioni sintattiche del caso (spazi obbligatori dopo `{` e prima di `}`, occorre il `;` o l'andata a capo prima di `}`)
 - è possibile aggiungere delle redirezioni; verranno effettuate solo al tempo della chiamata
 - non si scrivono mai gli argomenti: dopo `nomefunzione` c'è sempre `()` (tra l'altro, sono sintatticamente opzionali)
 - gli argomenti, come anticipato in altre lezioni, sono gestiti come gli argomenti degli script, usando i parametri posizionali

- quindi, i parametri posizionali, quando usati nel corpo di una funzione, non sono più gli argomenti dello script ma quelli della funzione
 - una volta terminata una chiamata, i parametri posizionali tornano ad essere quelli dello script (a meno che non si trattasse di una funzione che ne aveva chiamata un'altra..)
 - sintassi per la *chiamata*: `nomefunzione arg1 ...argn` (separati da spazi)
 - quindi nessun controllo sugli argomenti, né su quanti sono né sul tipo
 - il nome di una funzione è un identificatore, ma si tratta di identificatori separati da quelli per le variabili
 - si possono definire funzioni con lo stesso nome dei comandi; nel qual caso, hanno precedenza le funzioni (per ripristinare la situazione precedente, usare `unset`)
 - si possono fare funzioni ricorsive:

```
function fib () { local fm1;
local fm2; if [ $1 -lt 3 ]; then return 1 ; else fib
$(( $1 - 1 )) ; fm1=$?; fib $(( $1 - 2 )); fm2=$?; return
$((fm1 + fm2)); fi; }
```
 - notare le parole chiave `return` e `local`
 - senza `local`, si suppone che le variabili siano globali, quindi la funzione di cui sopra non funzionerebbe
 - il `return` è come l'`exit` degli script (in effetti, può essere usato anche dentro uno script, ed è uno dei pochi casi, se non l'unico caso, di comando che si può usare solo in uno script e non nella shell interattiva); provare quindi a chiamare `fib 13` e `fib 14`...
 - **esercizio:** usando una variabile d'appoggio `funres` anziché `return`, fare in modo che la funzione `fib` possa andare oltre il valore 13 (ovviamente, si arriva fino ad un certo massimo che dipende dal sistema, e dovrebbe essere tra 90 e 100...)
 - **esercizio:** riconsiderare l'esercizio sul `case` dato nella lezione 14, e far sì che il parsing della riga di comando (opzioni ed argomenti) sia fatto da una funzione
- Nota: il comando `source` può essere usato, in pratica, per includere uno script dentro un altro
 - *Here documents*
 - altra tecnica di redirection, vale solo per l'input
 - anziché leggere da file, si legge direttamente dalla riga di comando
 - sintassi:

```
comando << parolaDiFineInput
riga di input 1
```

- ...
`riga di input n`
`parolaDiFineInput`
- semanticamente, è come se si fosse fatto il redirect in input di un file che contiene le n righe:
`riga di input 1`
...
`riga di input n`
- con una importante differenza: nell'here documents avvengono le espansioni dei parametri (anche aritmetiche) e del command substitution
- ma in modo “strano”: sia i single che i double quotes restano così come sono (niente quote removal), quindi ad esempio se `var` vale 10 allora `'$var'` viene espansa in `'10'`
- l'unico modo per evitare l'espansione dei parametri è escapparli con il backslash
- ovviamente, 2 backslash risultano in un backslash solo
- inoltre, per l'espansione dei parametri *non* avviene il word splitting
- invece, per il command substitution avviene *sempre* il word splitting (anche dentro i doppi apici)
- solitamente, `parolaDiFineInput` viene scelta in modo che non appaia (come riga singola) all'interno dell'input voluto (altrimenti, l'input verrebbe troncato...)
- scelte comuni sono EOF, o il comando al contrario (ad esempio, `tac` se il comando è `cat`)

- *Here strings*

- come here documents, ma con una singola parola
- se serve mettere spazi, allora occorrono i quotes, singoli o doppi (ovviamente, occhio al word splitting)
- sintassi:
`comando <<< parolaInput`
- per quanto riguarda le espansioni, sono di nuovo “normali” (`'$var'` viene espansa in `$var`); oltre a parametri, aritmetiche e command substitution, c'è anche la tilde expansion

- Vedere esempi allegati