Sistemi Operativi, Secondo Modulo, Canale A–L e Teledidattica
Riassunto delle lezioni del 22/03/2021 e del 24/03/2021

Igor Melatti

La Bash, per davvero

- Pipelining: concatenazione di processi, con collegamento input-output
  - finora, abbiamo visto che si possono dare comandi in sequenza nei seguenti modi: con il `;` o l'andata a capo (equivalenti); con `&&` e il `||`; mettendoli o no dentro parentesi tonde o graffe
  - in tutti questi esempi, quello che succede è semplicemente che prima si esegue un comando, e poi (eventualmente) un altro; input ed output sono scollegati, a meno di redirezioni comuni nel caso di sottoshell o di group command
  - preziosa opportunità un più: collegare gli stdout agli stdin per le sequenze non condizionali (quindi, equivalenti alla separazione con `;` o con l'andata a capo)
  - ovvero, anziché scrivere `cmd1 >file1; cmd2 <file1; rm -f file1`, si scrive `cmd1 | cmd2`
  - lo stdout del processo a sinistra viene rediretto nello stdin del processo a destra
  - scrivendo `|&` anziché `|`, allora anche lo stderr viene rediretto nello stdin
  - se viene mandato in background, con `jobs -l` si vede cosa succede: tutti i processi corrispondenti ai comandi in pipelining sono stati lanciati
  - il pipelining, se applicato ad un group command (comandi tra graffe) o ad una subshell (comandi tra tonde) ha effetto sull'input/output combinato di tutti i comandi del gruppo
  - ad esempio, anziché scrivere `{ cmd1; cmd2; } > out; cmd3 < out; rm out`, si può scrivere `{ cmd1; cmd2; } | cmd3`

- **esercizio:** senza usare `/dev/null`, né alcun file temporaneo, fare in modo che il comando `ls -l fileesistente filenonesistente` scriva solo le informazioni sul file esistente
- **esercizio:** anziché scrivere `awk '{print}' 0< file 1<> file`, risolvere il problema usando il pipelining ed il comando `tee` (vedere il manuale), assumendo che l'esecuzione dei comandi in pipelining sia da sinistra a destra (nota: in realtà questo non è vero; per essere proprio sicuri usare `sponge` anziché `tee`)
- **esercizio:** senza usare un file temporaneo, far sì che awk mostri i soli file della directory attuale che abbiano il permesso di lettura abilitato su tutti e 3 i gruppi, abbiano un nome lungo almeno 10 caratteri ed una dimensione tra i 100 ed i 1000 bytes; il risultato dev'essere la stessa linea ritornata da `ls`
- **esercizio:** senza usare un file temporaneo, far sì che awk mostri i soli file in una qualsiasi sottodirectory della directory attuale che abbiano il permesso di lettura abilitato su tutti e 3 i gruppi, abbiano un nome lungo almeno 10 caratteri ed una dimensione tra i 100 ed i 1000 bytes; il risultato dev'essere la stessa linea ritornata da `ls`, ma con il nome sostituito dal path completo
- **esercizio:** senza usare un file temporaneo, far sì che vengano salvate su un file `risultato.txt` tutte le righe del `man bash` che contengono la stessa parola 2 volte, senza usare `awk`
- **esercizio:** senza usare un file temporaneo, far sì che vengano salvate su un file `risultato.txt` tutte le righe del `man bash` che contengono la stessa parola 2 volte, usando `awk`

Script in Bash

- Primi rudimenti di shell scripting
  - prendere un comando qualsiasi (ad esempio, `ls`), e scriverlo su un file di testo; sia `filename` il nome di tale file
  - è possibile eseguire tale file in 4 modi:
    1. `chmod u+x filename; ./filename` (non proprio ortodosso: ci vorrebbe una prima riga, detta *shabang*, fatta in un certo modo; ci ritorneremo)
    2. `bash filename`
    3. `source filename`
    4. `. filename`
  - `filename` è un *bash script* (spesso, si usa l'estensione `.sh` o `.script`)
  - eseguirlo nei primi 2 modi equivale a lanciare una sottoshell (sempre, anche se c'è dentro un solo comando) che esegue uno alla volta i comandi contenuti nello script

  - invece, nei secondi 2 modi *non* si lancia una sottoshell, e l'esecuzione avviene nel contesto della shell corrente
  - in realtà, la bash permette di avere un vero e proprio linguaggio di programmazione Turing-completo, i cui comandi base sono, essenzialmente, i comandi della shell visti finora più gli assegnamenti e i controllori di flusso (vedere sotto)
  - quindi, è possibile compiere decisioni (ad esempio, con l'`if`) e cicli (ad esempio, con il `for` e il `while`)
  - quindi, alcuni comandi potrebbero non essere eseguiti, o eseguiti più volte
  - se ci sono errori di sintassi: la parte che precede l'errore viene sempre eseguita
  - la parte che segue l'errore potrebbe essere eseguita o no, a seconda della gravità dell'errore
  - tutto quello che si scrive sulla bash interattiva può essere messo in uno script; al posto del `;`, per separare i comandi, si può usare l'andata a capo
  - il viceversa non è vero, ma solo perché le andate a capo possono essere solo negli script
  - infatti, nella shell interattiva, premere invio vuol dire "esegui il comando"...
  - c'è però l'eccezione già menzionata sopra (se ci sono parentesi o apici aperti)
  - altra eccezione: mettendo un backslash alla fine del comando (ma proprio alla fine, senza spazi successivi), l'effetto è lo stesso di quando ci sono apici o parentesi aperte e non chiuse

- Parametri e variabili
  - per poter essere veramente Turing-completi, è necessario avere anche delle *variabili*, esattamente come nei linguaggi di programmazione
  - le variabili sono definite come nei linguaggi di programmazione: un *identificatore* cui si assegna, e dal quale si può recuperare, un valore
  - niente tipi: di default tutte stringhe, ma possono essere interpretate come interi in opportune circostanze; con opportuna sintassi, possono essere anche degli array (associativi o standard)
  - formalmente, in bash ci sono le variabili (definite come sopra), i *parametri posizionali* e i *parametri speciali*
  - inoltre, sempre formalmente, tutte e 3 le categorie appena definite sono indicate genericamente come *parametri*
  - solo le variabili possono essere soggette ad *assegnamento*; per i parametri posizionali, si può usare il comando builtin `set`

  - tutti i parametri possono essere soggetti ad *espansione* (ovvero, si può usare il valore che contengono)

- Variabili
  - si assegnano con `identificatore=espressione`
  - niente spazi prima e dopo l'uguale!!!!
  - si espandono con `$identificatore`, oppure con `${identificatore}`
  - quindi, per vedere quanto valgono, è sufficiente dare un comando come `echo ${identificatore}` (il meccanismo che c'è sotto, chiamato *espansione*, sarà più chiaro nel seguito di questa e delle prossime lezioni)
  - anche, ma più complicato: `echo niente | awk -v var=${identificatore} '{print var}'`, o anche `echo niente | awk '{print "'${identificatore}'"}'`
  - la versione con le graffe è importante quando ci sono 2 variabili, i cui nomi sono l'uno il prefisso dell'altro
  - senza graffe, il nome della variabile finisce non appena non ci sono né caratteri alfanumerici né underscore; altrimenti, il nome della variabile finisce con la }
  - nel caso delle stringhe (default), si possono concatenare a piacimento stringhe costanti e stringhe variabili, senza usare nessun operatore di concatenazione
  - espandere una variabile mai definita prima dà luogo alla stringa vuota (o al valore 0, nelle espansioni aritmetiche)
  - dentro alle parentesi graffe si possono in realtà fare molte cose, ma per ora soprassediamo (vedere "Parameter expansion" nel `man bash`)
  - per le espressioni aritmetiche, ci ritorneremo

- Variabili locali e d'ambiente
  - tutte le variabili assegnate come descritto sopra sono *locali*
  - ovvero, un processo lanciato da uno script, dopo che una variabile è stata settata, non può vedere il valore di quella variabile
  - fanno eccezione le sottoshell tra parentesi tonde (ovviamente anche i compund group tra graffe, ma lì non viene lanciata nessuna sottoshell), nonché ovviamente gli script lanciati con `source` o `.` (che non creano un nuovo processo)
  - se si vuole far sì che un qualsiasi (nuovo) processo lanciato da uno script possa accedere alle variabili $v_1, \ldots, v_n$, precedentemente settate ai valori $a_1, \ldots, a_n$, occorre:
    * usare il comando `export nomevar` (o anche `declare -x nomevar`) prima di lanciare il processo: `export` v_1, ..., `export` v_n

Table 1: Alcune variabili d'ambiente predefinite o comunque usate da bash (vedere `man bash` per la lista completa)

Nome	**Significato**
`PATH`	Lista di directory, ognuna separata da `:`. Ogni qualvolta viene dato un comando `cmd` senza path (ovvero, senza usare neanche uno slash `/`), la bash cerca in ciascuna di tali directory un file eseguibile di nome `cmd`. Viene eseguito il file che si trova nella prima directory (da sinistra) dove il file stesso viene trovato. Se `cmd` non viene trovato, allora l'errore è "Command not found" (exit code 127). Il comando `which` si limita a trovare tale file, senza eseguirlo; `which -a` trova tutte le occorrenze.
`HOME`	Path assoluto della home directory dell'utente attualmente loggato.
`IFS`	Internal Field Separator, usato dal word splitting. Attenzione: quando si lancia uno script (a meno che non lo si faccia con `source` o `.`) questo valore viene sempre resettato al suo valore di default, che è `\t\n`
`BASHPID`	Pid della bash corrente.
`PS1`	Formattatore per il prompt.
`PWD`	Path assoluto dell'attuale cwd; ogni comando `cd` ne cambia il valore.
`OLDPWD`	Path assoluto del cwd precedente quella attuale; ogni comando `cd` ne cambia il valore.

    - oppure, usare la seguente sintassi: v_1=a_1 ... v_n=a_n `comando`
    - nel primo caso, le variabili $v_1, \ldots, v_n$ saranno disponibili a tutti i processi che verranno lanciati in seguito
    - nel secondo caso, le variabili saranno visibili solo al processo creato da `comando`
    - in ogni caso, eventuali modifiche non avranno effetto sulla bash padre
- il comando `export` ha proprio l'effetto di trasformare una variabile locale in variabile d'ambiente
- il comando `unset nomevar` annulla una `export nomevar`
- esempio: `bash lancia_var_ambiente.sh`
- inizialmente, ci sono svariate variabili predefinite nell'ambiente (e altre se ne possono aggiungere mettendole nei file di configurazione); le più importanti sono descritte in Tabella 1
- **esercizio:** scrivere uno script che modifichi la variabile d'ambiente `PATH`, cancellando il precedente contenuto e facendo sì che contenga solo la directory corrente, e poi ne stampi il valore. Verificare che, lanciandolo come nuovo processo, la modifica avviene effettivamente

all'interno dello script, ma nella bash di partenza (quella che invoca lo script) la variabile `PATH` non è stata modificata. Verificare anche che, lanciandolo all'interno dello stesso processo della bash, la modifica risulta permanere anche nella bash invocante. Modificare lo script in modo tale che, dopo la modifica, la variabile `PATH` venga ripristinata al suo valore precedente.

- Parametri posizionali
  - si vedono solo negli script, o nei corpi delle funzioni
  - vengono espansi con il carattere `$` seguito da un numero intero positivo
  - `$n` sta per l'`n`-esimo argomento dato allo script o alla funzione
  - se `n` necessita più di una cifra decimale per essere scritto, allora l'espansione va fatta con le parentesi graffe: `${n}` (altrimenti...)
  - con `set {valore}` si possono cambiare i valori di questi parametri: se vengono specificati `n` valori, allora il primo viene assegnato a `$1`, il secondo a `$2`, ..., l'`n`-esimo a `$n`
  - con `shift [n]` si possono cambiare i valori di questi parametri: quelli da `$1` a `$n` avranno come valore la stringa vuota (quindi, saranno non assegnati), mentre `$(n+1)` avrà il vecchio valore di `$1`, `$(n+2)` avrà il vecchio valore di `$2` e così via fino all'ultimo. L'argomento di default di `shift` è 1.
  - nelle chiamate a funzione, i valori dei parametri posizionali vengono modificati solo temporaneamente, in modo da avere i valori passati alla chiamata stessa
  - una volta finita la chiamata, riprendono i valori precedenti (ci ritorneremo)
  - **esercizio:** scrivere uno script che stampi il valore dei suoi argomenti, usando solo `$1` come parametro posizionale. Assumere che verranno sempre passati esattamente 5 argomenti.
  - **esercizio:** scrivere uno script che stampi il valore di alcuni suoi argomenti, usando solo `$1` come parametro posizionale. Quali argomenti stampare è deciso dal primo argomento: assumendo che sia un intero positivo n, occorrerà stampare gli argomenti in posizione $1 + in$, con $i \in \{1, 2, 3, 4, 5\}$.
- Parametri speciali, ci limitiamo ai seguenti
  - `$0`: il nome dello script, come è stato avviato (o quasi, rivedere Tabella 1 della lezione 7)
  - `$*`: lista di tutti i parametri posizionali (da 1)
  - `$@`: lista di tutti i parametri posizionali (da 1); la differenza con il precedente la fa il word splitting (ci ritorneremo)

  - `$#`: numero di parametri posizionali (da 1), separati da uno spazio
  - `$?`: exit status dell'ultimo processo terminato
  - `$!`: pid dell'ultimo processo avviato in background (terminato o no)
  - `$$`: pid della bash (o del parent della bash, se si tratta di una sottoshell)
  - **esercizio:** scrivere uno script che stampi il valore del suo ultimo argomento.
  - **esercizio:** scrivere 2 script `a.sh` e `b.sh`, con `a.sh` che lancia `b.sh`. Fare in modo che `a.sh` stampi il valore del suo i-esimo argomento (che esista o no), dove i è l'exit status di `b.sh`

- Per le variabili è possibile (ma obbligatorio solo per gli array associativi) specificare un tipo tra:
  - stringa: tipo di default, accetta qualsiasi assegnamento
  - intero: `declare -i`. Se gli si assegna una stringa, la variabile prende il valore 0; per essere precisi: se gli si assegna un valore, allora viene invocata la valutazione aritmetica (vedere più avanti);
  - array indicizzato da interi: `declare -a`
  - array indicizzato da stringhe (array associativo): `declare -A` (dichiarazione obbligatoria)
  - variabile costante (ossimoro...): `declare -r` (occorre averci assengato un valore in precedenza...); attenzione, non è reversibile!

- Variabili di tipo array
  - `varArray=(`v_0 `...`v_{n-1}`)`, dove i valori v_i sono separati da spazi (e possono essere eterogenei, con stringhe mischiate ad interi); in questo caso, gli indici partono da 0
  - si accede all'elemento di indice i così: `${varArray[`i`]}`
  - si può assegnare anche il singolo elemento: `varArray[`i`]=`v_i (non è necessario aver settato i precedenti i: gli array sono *sparsi*)
  - si possono specificare anche gli indici: `varArray=([`i_0`]=`v_0 `...[`i_{n-1}`]=`v_{n-1}`)`
  - per gli array associativi, gli indici sono stringhe; per il resto è tutto uguale
  - gli indici possono essere espansioni di variabili

- Metacaratteri e *quoting*
  - i metacaratteri sono caratteri speciali, che sono interpretati da bash in un modo peculiare per ciascuno di essi
  - sono solo i seguenti: `(`, `)`, `|`, `&`, `;`, `<`, `>`

  - l'interpretazione è già stata discussa: inizio e fine sottoshell, pipelining o esecuzione condizionale, esecuzione in background o esecuzione condizionale o redirezioni, terminatore di comando, redirezioni
  - altri caratteri hanno significato speciale in alcuni contesti: `$`, `!`, `,`, {, }, `*`, `+` (alcuni già visti, altri da vedere)
    * per esempio: `$` è un carattere speciale solo se non è seguito da spazi; provare a digitare `echo ciao $ ciao` e `echo ciao $ciao`
  - per usarli nel loro senso letterale, occorre il quoting:
    * mettendo davanti a ciascun metacarattere un \; se si va a capo subito dopo un \, il comando continua nella riga sottostante
    * usando la sintassi speciale `$'carattere'`; se si vuole stampare `'`, occorre scrivere `$'\''` oppure `"'"`; se si vuole stampare \, occorre scrivere `$'\\'`; si possono usare le escape sequence dei comandi `printf`: `\n` è l'andata a capo (carriage return + line feed), `\r` solo il carriage return, `\f` solo il line feed, `\t` è la tabulazione
    * racchiudendoli tra single quote `'`. All'interno, non ci possono essere altri `'`, nemmeno con il backslash davanti; tutti gli altri caratteri speciali perdono il loro essere speciali e vengono scritti così come appaiono
      · per la precisione: `echo '\'` stampa semplicemente il \ (il quale, essendo dentro degli apici singoli non preceduti da $, non è più un metacarattere); quindi `echo '\''` vorrebbe dire *stampa il \ e poi...* ...aspetta che arrivi la chiusura del terzo apice, che è aperto e non chiuso
    * racchiudendoli tra double quote `"`. All'interno, alcuni caratteri speciali, come `!` (history expansion, ci ritorneremo), `$` (espansione dei parametri, descritta in questa lezione, più altre espansioni che vedremo in seguito) e `` ` `` (command substitution, ci ritorneremo) mantengono il loro significato speciale; anche \, ma solo se è seguito da uno di questi 3 caratteri o da `"`.
  - **esercizio:** Scrivere uno script che, usando il comando `echo`, si fa stampare tutti i metacaratteri riportati sopra, usando tutti i metodi descritti. Alla fine, farsi stampare anche le seguenti stringhe: `'ciao'`, `"ciao"`, `"'ciao'"`, `'"ciao"'`
  - **esercizio:** come sopra ma, anziché `ciao`, stampare 2 volte `ciao`, con una tabulazione in mezzo
  - **esercizio:** sperimentare la differenza tra carriage return, line feed e la combinazione dei 2

- Funzionamento di bash: per ogni comando, *tokenizzazione* ed *expansion*
  - la tokenizzazione, essenzialmente, scompone il comando in *words* e *operators*

  - l'espansione trasforma le sole word secondo le regole descritte sotto

- Tokenizzazione (vedere anche `http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_01_04.html#sect_01_04_01_01`)
  - ottiene i cosiddetti *building blocks*, ovvero words ed operators
  - gli operatori sono i metacaratteri già visti in precedenza, ovvero: `(`, `)`, `|`, `&`, `;`, `<`, `>`
  - da non confondere con il *word splitting* descritto sotto: per separare le word serve almeno uno spazio, e non c'è modo di cambiare questo comportamento
  - gli operatori possono anche essere scritti attaccati (senza spazi di separazione), sia tra loro che alle word
  - se un operatore è preceduto da \, oppure è tra apici singoli o doppi, oppure ancora è all'interno del costrutto `$''`, diventa una word; lo stesso succede se si trova all'interno di un costrutto come quello dell'arithmetic expansion (vedere sotto) o delle condizioni aritmetiche (vedere lezione 11)
  - come risultato della tokenizzazione, il comando potrebbe essere diventato una lista di comandi (per esempio: se in mezzo c'è un `;`)

- Expansion (vedere anche `http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_04.html`)
  - alcune già viste: il globbing o filename expansion (espansione delle wildcard per formare un filename), l'espansione delle variabili, la tilde che indica la home directory
  - ce ne sono anche altre; l'elenco completo è in Tabella 2, che mostra anche l'ordine con cui vengono applicate
  - quello che accade è che la bash, prima di eseguire un comando, effettua in ordine le espansioni riportate in Tabella 2
  - quindi, quale comando viene effettivamente eseguito (e con quali argomenti/opzioni) dipende da tali espansioni
  - per esempio: il comando `ls ~/*.txt` dapprima diventa `ls /home/utente/*.txt` (per la tilde expansion) e poi `ls /home/utente/file.txt` (filename expansion, assumendo che l'unico file con estensione `.txt` nella home di `utente` sia `file.txt`)
  - se l'ordine di espansione fosse stato l'inverso, si sarebbe ottenuto lo stesso risultato?
  - le espansioni possono essere in cascata, ovvero al risultato di una espansione si applicano le seguenti (ma nell'ordine di Tabella 2)
    - eccezione: non può succedere di applicare la variable expansion al risultato di una tilde expression, perché la tilde expression dentro delle parentesi graffe (senza spazi) non è riconosciuta come tale (deve essere staccata da metacaratteri o caratteri speciali)

Table 2: I tipi di espansione di bash, e loro ordinamento. Nella terza colonna sono riportati i casi in cui l'espansione avviene anche se all'interno di double quotes `""`; se vengono usate le single quotes, allora non avviene mai alcuna espansione. Attenzione però, vincono le quotes più esterne: se le double quotes sono dentro delle single quotes, nessuna espansione; se le single quotes sono dentro delle double quotes, c'è l'espansione per quei 3 casi che la prevedono

Espansione	**Ordine**	**Double quotes**
Brace expansion	1	NO
Tilde expansion	2	NO
Parameter and variables expansion	3	SÌ
Arithmetic expansion	4	SÌ
Command expansion (o substitution)	5	SÌ
Word splitting	6	NO
Filename expansion (o globbing, o wildcarding)	7	NO
Process substitution, non sempre disponibile	8	NO
(Quote removal)	9	

- Brace expansion: espansione delle parentesi graffe (*brace expression*)
  - si tratta di una mini-espressione regolare, che riconosce un linguaggio *finito*, ma con una sintassi diversa da quella vista finora
  - inoltre, qui lo scopo non è il matching, come per `grep`: qui il risultato è proprio la generazione di tutte le stringhe del linguaggio dell'espressione regolare
  - due forme sintattiche: $\{p_1\texttt{,}\ldots\texttt{,}p_n\}$ e $\{v..w\}$ (attenzione: niente spazi all'interno delle parentesi)
  - prima forma: le p_i sono stringhe qualsiasi; se serve che contengano {, } o `,` occorre quotarli
  - seconda forma: v e w devono essere valori interi (altrimenti, non è più una brace expression)
  - la prima forma viene espansa in una stringa che contiene $p_1 \ldots p_n$, separati da uno spazio
  - la seconda forma viene espansa in una stringa che contiene $v\ v+1 \ldots w$ (sempre separati da uno spazio), oppure $v\ v-1 \ldots w$, a seconda che valga $v < w$ oppure il contrario
  - c'è una terza forma, variante della seconda: $\{v..w..k\}$, che viene espansa in $v\ v+k \ldots v+ik$ (con i più grande intero tale che $v+ik \leq w$), oppure $v\ v-k \ldots v-ik$ (con i più grande intero tale che $v-ik \geq w$), a seconda se $v < w$ oppure il contrario
  - è possibile combinare più di una brace expansion. In generale, se si usano n brace expansion, ciascuna risultante in $i_1, \ldots, i_n$ stringhe, allora il numero totale di stringhe generate sarà $\prod_{j=1}^{n} i_j$

Table 3: Tilde expressions e tilde expansions

Tilde expression	**Espansione**
(stringa vuota)	`$HOME`
`nomeutente`	Home directory di `nomeutente` (se esistente)
`+`	`PWD`
`-`	`OLDPWD`
`+n`	output di `dirs +n`
`-n`	output di `dirs -n`

  - **esercizio:** usando la brace expansion, farsi stampare tutti i numeri tra 0.0 e 10.0, con step di 0.1
  - **esercizio:** usando la brace expansion, farsi stampare tutti i numeri tra 0.0 e 10.0, con step di 0.5

- Tilde expansion: espansione del carattere tilde (*tilde expression*)
  - non solo home: vedere Tabella 3
  - `dirs` è un comando usato in concomitanza con `pushd` e `popd`, che mantengono uno stack di directory (vedere `man bash`)
  - **esercizio:** assegnare ad una variabile la home di `utente1`, e poi stamparla

- Espansione di parametri e variabili, è quella con più possibilità
  - in Tabella 4 vengono riportate le principali
  - **esercizio:** creare un array dove sono definiti i seguenti indici: `3`, `5`, `10`, `100`, con valore pari al doppio dell'indice; farsi poi stampare il valore agli indici `3` e `4`; usando la Tabella 4, farsi stampare anche la lunghezza (come numero di caratteri) di tali valori; farsi inoltre stampare tutti gli indici così settati, e farsi anche stampare il numero totale di indici settati
  - **esercizio:** creare un array dove sono definiti i seguenti indici: `3`, `5a`, `10df`, `100`, con valore pari al doppio della parte numerica dell'indice; farsi poi stampare il valore agli indici `5a` e `4`; usando la Tabella 4, farsi stampare anche la lunghezza (come numero di caratteri) di tali valori; farsi inoltre stampare tutti gli indici così settati, e farsi anche stampare il numero totale di indici settati

Table 4: Variable and parameter expressions e expansions (laddove vengano nominati i pattern solo gli stessi per il file globbing, vedere fine della lezione 5)

Espressione	**Espansione (var definita)**	**Espansione (var non definita)**
`$nomevar`	valore di `nomevar`	stringa vuota o 0
`${nomevar}`	valore di `nomevar`	stringa vuota o 0
`${nomevar:-stringa}`	`${nomevar}`	`stringa`
`${nomevar:+stringa}`	`stringa`	stringa vuota o 0
`${nomevar:?stringa}`	`${nomevar}`	scrive `stringa` su stderr; se la shell è interattiva, abortisce la parte restante del comando; se la shell non è interattiva, abortisce l'intera shell
`${nomevar:=stringa}`	`${nomevar}`	prima assegna `stringa` a `nomevar`, e poi espande `nomevar`
`${nomevar:offset}`	`${nomevar}` dalla posizione `offset` in poi (si comincia da 0); se `offset` è negativo (attenzione: allora mettere uno spazio prima del `-`) allora la posizione è data dalla lunghezza di `${nomevar}` cui si sottrae offset	stringa vuota o 0
`${nomevar:offset:length}`	come sopra, ma espande al più `length` caratteri	come sopra
`${#nomevar}`	lunghezza dell'espansione di `${nomevar}`; se si tratta di un array, allora lunghezza dell'espansione di `${nomevar[0]}`	0
`${#nomevar[i]}`	lunghezza dell'espansione di `${nomevar[i]}`	0
`${!nomevar[*]}`	lista degli indici assegnati nell'array `nomevar`	0
`${!nomevar[@]}`	lista degli indici assegnati nell'array `nomevar`; come per `$@` e `$*`, la differenza la fa il word splitting	0
`${!pref*}`	lista dei nomi di variabili definite che iniziano con `pref`	
`${!pref@}`	lista dei nomi di variabili definite che iniziano con `pref`; come per `$@` e `$*`, la differenza la fa il word splitting	

Continuazione di Tabella 4 (laddove vengano nominati i pattern solo gli stessi per il file globbing, vedere fine

Espressione	**Espansione (var definita)**	**Espansione (var non definita)**
`${nomevar#pattern}`	prima espande `nomevar`, poi toglie la più corta sequenza iniziale che faccia match con `pattern`	stringa vuota o 0
`${nomevar##pattern}`	prima espande `nomevar`, poi toglie la più lunga sequenza iniziale che faccia match con `pattern`	stringa vuota o 0
`${nomevar%pattern}`	prima espande `nomevar`, poi toglie la più corta sequenza finale che faccia match con `pattern`	stringa vuota o 0
`${nomevar%%pattern}`	prima espande `nomevar`, poi toglie la più lunga sequenza finale che faccia match con `pattern`	stringa vuota o 0
`${nomevar/pattern/string}`	simile al replace di `sed`: prima espande `nomevar`, poi sostituisce con `string` la più lunga sottostringa che faccia match con `pattern`	stringa vuota o 0
`${nomevar//pattern/string}`	prima espande `nomevar`, poi sostituisce con `string` tutte le sottostringhe che facciano match con `pattern`	stringa vuota o 0
`${nomevar/#pattern/string}`	prima espande `nomevar`, poi sostituisce con `string` la più lunga sottostringa iniziale che faccia match con `pattern`	stringa vuota o 0
`${nomevar/%pattern/string}`	prima espande `nomevar`, poi sostituisce con `string` la più lunga sottostringa finale che faccia match con `pattern`	stringa vuota o 0
`${nomevar^pattern}`	prima espande `nomevar`, poi converte il primo carattere del risultato, trasformandolo in maiuscolo, purché faccia match con `pattern`. Attenzione, questi pattern devono fare match con un singolo carattere (ad esempio, un pattern che fa la concatenazione di due letterali non avrà effetto)	stringa vuota o 0

Continuazione di Tabella 4 (laddove vengano nominati i pattern solo gli stessi per il file globbing, vedere fine

Espressione	**Espansione (var definita)**	**Espansione (var non definita)**
`${nomevar,pattern}`	come sopra, ma trasforma in minuscolo	stringa vuota o 0
`${nomevar^^pattern}`	prima espande `nomevar`, poi converte tutti i caratteri che facciano match con `pattern`, trasformandoli in maiuscolo	stringa vuota o 0
`${nomevar,,pattern}`	prima espande `nomevar`, poi converte tutti i caratteri che facciano match con `pattern`, trasformandoli in minuscolo	stringa vuota o 0

- **esercizio:** verificare che una variabile intera, se non assegnata precedentemente, viene espansa con 0
- **esercizio:** assegnare alle variabili `v1` e `v2` i valori 1 e 2, rispettivamente, e poi farsi stampare il loro valore separato da _
- **esercizio:** farsi stampare il valore della variabile `v1`, oppure `la variabile v1 non e' definita` se alla variabile non è stato ancora assegnato un valore (attenzione al quoting...)
- **esercizio:** farsi stampare `la variabile v1 e' definita` se alla variabile è già stato assegnato un valore, e la stringa vuota altrimenti
- **esercizio:** scrivere uno script che stampa per 3 volte, su 3 righe consecutive, il valore di una variabile d'ambiente `var_ambiente`, preceduta dalla stampa di una riga `inizio` e seguita dalla stampa della parola `fine`; se tale variabile non è stata definita, allora solo la prima riga dev'essere scritta. Non devono essere stampati altri messaggi. Verificare il corretto funzionamento dello script.
- **esercizio:** come sopra, ma se la variabile d'ambiente `var_ambiente` non è definita, allora assegnarle il valore della variabile `PATH`
- **esercizio:** scrivere uno script che stampi, del valore di `PATH`: i) gli ultimi 10 caratteri; ii) i caratteri dal decimo in poi; iii) i caratteri dal ventesimo al quarantesimo (compresi); iv) la lunghezza (come numero di caratteri)
- **esercizio:** farsi stampare i nomi di tutte le variabili d'ambiente che cominciano con `BASH_`
- **esercizio:** farsi stampare `PATH`, ma: i) senza la prima directory; ii) senza l'ultima directory; iii) con la sola prima directory; iv) con la sola ultima directory; v) con `lib` al posto di `usr`; vi) con `lib` al posto di `usr`, tranne che per la prima occorrenza di `usr`, che dev'essere sostituita con `var`; vii) solo l'ultima directory, e al posto di quelle precedenti deve scrivere `altre`; viii) solo la prima directory, e al posto di quelle precedenti deve scrivere `altre`; ix) con tutti i caratteri maiuscoli; x) con le sole lettere `u, s, r` messe in maiuscolo

- Arithmetic expansion
  - tutte le operazioni sugli *interi*
  - vengono valutate, e il risultato viene sostituito all'interno del comando
  - sintassi: `$((expr))`, dove `expr` è scritta come una normale espressione algebrica (anche con le parentesi); con `%` che è il modulo
  - si possono fare anche confronti (ma stavolta 0 è falso, diverso da 0 è vero)
  - anche operazioni bit-a-bit, come nel C, e operazioni logiche (OR, AND, NOT; per il bit-a-bit c'è anche lo XOR con ^)
  - si possono usare anche le variabili (quelle locali o d'ambiente della bash), tanto la variable expansion avviene prima... in più, si possono riferire le variabili anche senza `$`
  - si possono fare assegnamenti, anche con operatori di modifica (del tipo `+=`), e di incremento/decremento (`++`, `--`), sia pre che post
  - esiste l'espressione condizionale con il `?`
  - è possibile fare assegnamenti e poi dare un'espressione da valutare: `,`
  - se viene usata una variabile che non è stata definita, viene espansa a 0
  - se una variabile dentro un'espressione aritmetica espande come una stringa, l'espansione prova ad espandere quella stringa come se fosse una variabile (il processo è ricorsivo; se si arriva ad una variabile non definita, il suo valore è 0; se si arriva ad una variabile con un nome non valido, dà errore)!
    - esempio: `a=b; b=10; echo $a $((a));`
  - per inciso: la sintassi `(())` può essere usata anche senza `$`: è un comando che può essere usato per assegnare valori
  - **esercizio:** verificare se la lunghezza di `PATH` è maggiore della lunghezza di `PWD` oppure la vecchia current directory ha più caratteri di quella attuale; dipendentemente da questo, assegnare alla variabile `var` il valore 10 (se è vero) o 20 (se è falso)
  - **esercizio:** verificare se il valore di `BASH_PID` è pari o dispari
- Command substitution
  - due sintassi: `` `comando` `` (backticks) e `$(comando)`
  - si possono fare annidamenti (nel primo caso, occorrerà usare \`)
  - l'espansione consiste nell'eseguire il comando, prendere ciò che viene scritto in stdout e sostiturlo al comando stesso

  - **esercizio:** come si può, con un solo comando, cercare dei file che corrispondono ad un certo pattern, e poi cercare nei files di tale insieme una certa stringa? Farlo in due modi: i) usando una variabile d'appoggio e la concatenazione tra comandi; ii) usando un comando solo e la command substitution
  - **esercizio:** creare un array dove sono definiti i seguenti indici: `3`, `5a`, `10df`, `100`, con valore pari al doppio della parte numerica dell'indice; assegnare ad una variabile `var` il numero di indici assegnati in tal modo

- Word splitting: attenzione, si applica solo al risultato di parameter, arithmetic e command expansion, e solo se non double quoted (se fosse single quoted, niente espansione...)
  - semplicemente, ogni diversa parola è passata separatamente al comando
  - normalmente, le parole sono separate da spazi o tab
  - nel caso della bash, sono in realtà separate dai caratteri dentro la variabile d'ambiente `IFS`
  - esempio: supponiamo di settare `IFS=:`
  - il comando `echo a:b` scrive `a:b`
  - il comando `echo "a:b"` scrive `a:b`
  - ma questo solo perché il word splitting qui non avviene, in quanto non c'è stata nessuna delle 3 espansioni (di variabili, aritmetica o di comando) richieste
  - infatti, il comando `echo ‘echo a:b‘` scrive `a b`
  - mentre il comando `echo "‘echo a:b‘"` scrive `a:b` (dentro i double quotes, niente word splitting)
  - un risultato analogo lo si può ottenere con il comando `var="a:b"; echo $var; echo "$var"` (dove si usa l'espansione dei parametri anziché la command substitution)
  - analogamente, `echo $IFS` non scrive nulla: infatti dapprima la bash espande `$IFS` in `:`, e poi viene fatto il word splitting, che sostituisce il `:` con uno spazio
  - invece, `echo "$IFS"` scrive `:`, perché il word splitting non avviene
  - si può ora capire la differenza tra `$*` e `$@`: nel primo caso, vengono scritti tutti gli argomenti separati dal primo carattere di `IFS`, nel secondo invece vengono scritti tutti gli argomenti separati da spazi (indipendentemente dal valore di `IFS`)
  - se non sono double quoted, il risultato finale è quindi lo stesso: `$*` va soggetto a word splitting e il carattere `IFS` viene sostituito da spazi; `$@` va anch'esso soggetto a word splitting, ma non c'è nulla da sostituire, in quanto sono già tutti spazi

- se sono double quoted: non andando soggetto a word splitting, `$*` rimarrà separato dal carattere `IFS`, mentre `$@` rimane separato da spazi
- quindi, nel caso siano double quoted, il risultato sarà diverso non appena `IFS` non contenga lo spazio
- una precisazione importante: diversamente da altre variabili d'ambiente, `IFS` viene sempre resettata al suo valore di default (che è `$' \t\n'`) quando viene invocato uno script
- **esercizio:** verificare che il contenuto di `IFS` sia effettivamente quello sopra menzionato, usando opportunamente il comando `tr`
- esempio: sia `var.sh` il seguente script:

```
echo $HOME
echo $PATH
echo $PS4
echo "$IFS"
```

- dopo averlo reso eseguibile, provare ad invocarlo, per esempio, così: `HOME="ciao" PATH="aho" PS4="ehi" IFS=";" ./vars.sh`
- l'unica variabile non settata come indicato è proprio `IFS`; notare che vengono scritte 2 andate a capo: una è dovuta ad `IFS` stesso, l'altra ad `echo`
- **esercizio:** nella stampa ottenuta con il comando precedente, in realtà la quarta riga contiene prima uno spazio, poi un tab e poi un'andata a capo: trovare il modo per provare questa affermazione (senza usare il mouse...)
- per provarlo, scrivere dentro un file `chiocc.sh` la riga `IFS=";"; echo $@; echo "$@"`, e dentro un file `ast.sh` la riga `IFS=";"; echo $*; echo "$*"`
- supponendo che `IFS` valga `":"`, provare a chiamare `bash chiocc.sh a:b` e confrontare con `bash ast.sh a:b`
- il risultato è lo stesso: il word splitting non avviene, quindi ai due script viene passato `a:b`, che è un argomento unico; pertanto, né `$*` e `$@` hanno necessità di mettere separatori
- provare a chiamare `bash chiocc.sh $(echo a:b)` e confrontare con `bash ast.sh $(echo a:b)`
- il risultato sarà diverso nella seconda linea, come predetto; notare che il carattere usato per separare il risultato di `"$*"` è l'IFS settato *dentro* lo script
- infatti, il word splitting fa sì che i due script vengano chiamati con `a b`, quindi con 2 argomenti; dopodiché, il discorso fatto sopra fa il resto

  - supponendo che `IFS` contenga `:`, provare a chiamare `bash chiocc.sh "$(echo a:b)"` e confrontare con `bash ast.sh "$(echo a:b)"`
  - il risultato sarà uguale: non c'è word splitting nel passare gli argomenti, quindi ai due script viene passato `a:b`, che è un argomento unico
  - **esercizio:** si supponga di voler vedere le informazioni standard sulle directory contenute in `PATH`, settando opportunamente `IFS`. Funziona il comando `ls -ld $PATH`? Perché? E il comando `ls -ld "$PATH"`? Perché? E il comando `ls -ld $(echo $PATH)`? Perché? Come si può correggere, minimalmente, quest'ultimo comando?
  - **esercizio:** Si supponga che il contenuto di `PATH` venga messo su un file. Come si può ottenere lo stesso risultato dell'esercizio precedente?
  - **esercizio:** confrontare il risultato dei comandi `echo ‘ls‘` e `echo "‘ls‘"`. Cosa succede, e come si spiega?
  - **esercizio:** anziché "supporre" che `IFS` sia settato in un certo modo, come scritto nei test di sopra, come si possono eseguire gli stessi test da riga di comando, e fare in modo che poi `IFS` ritorni quello originario? (si può fare in 2 modi)

- Filename expansion (o globbing); `man 7 glob`
  - vedere lezione 5
  - **esercizio:** qual è il risultato del comando `ls index.{html,p*}`?
  - Creare 2 file in una directory inizialmente vuota: `filename.txt` e `file name.txt`. Che differenza c'è tra i comandi `ls *`, `ls ‘ls‘`, `ls "*"`, `ls "‘ls‘"`?

- Process substition
  - sintassi: `<(comando)` (lettura) e `<(comando)` (scrittura)
  - `cat, awk, sed, head`... se non si dà l'argomento come file, leggono da standard input
  - ma non tutti i comandi sono così: per esempio, `diff` vuole per forza 2 file
  - se per esempio si volesse confrontare il risultato (su standard output) di due comandi, usare `diff` è complicato: `cmd1 > file1; cmd2 > file2; diff cmd[12]; rm cmd[12]`
  - non si può usare il pipelining
  - viene in soccorso l'espansione detta *process substitution*
  - `diff <(cmd1) <(cmd2)`
  - quindi la semantica è: prendi ciò che è delimitato da `<()`, eseguilo, ridirigi lo standard output su un file temporaneo, e sostituisci `<()` con il nome di questo file temporaneo

  - si sottintende che il comando che usa `<()`, come `diff` nell'esempio di sopra, prenderà questo file per leggerne il contenuto
  - molto utile in lettura, un po' meno in scrittura
  - prendi il comando C che è delimitato da `>()`, crea un file temporaneo e sostituisci `>()` con il nome di questo file temporaneo; poi esegui C, dandogli come standard input il contenuto del file temporaneo
  - si sottintende che il comando che usa `>()` prenderà questo file per scriverci un qualche contenuto

- Quote removal
  - alla fine di tutte le espansioni, una diminuzione: tutte le occorrenze dei caratteri \, `"` e `'` vengono eliminate
  - a meno che non siano state prodotte da una qualche expansion: `echo ‘echo \"‘`
  - il che include anche il caso in cui abbiano un \ davanti: `echo \\`, o abbiano i single quotes, o i double quotes per i casi in cui anche i double quotes hanno effetto

- Relazioni tra tokenizzazione ed espansione
  - non si possono scrivere delle espansioni il cui risultato sia un metacarattere, e poi pretendere che sia interpretato come metacarattere
    * quando si arriva all'espansione, i metacaratteri sono già stati interpretati
    * sarà come se fossero stati opportunamente quotati
    * ad es.:
      `var=';'; echo ciao $var echo ciao`
      non è come
      `echo ciao ; echo ciao`
      ma bensì come
      `echo ciao ';' echo ciao`
    * ad es.:
      `var='> file'; echo ciao $var`
      non è come
      `echo ciao > file`
      ma bensì come
      `echo ciao '>' file`
  - ... a meno di non usare il comando built-in `eval`

- Esecuzione del comando: dopo tokenizzazione ed espansione, si salvano a parte le redirezioni e gli assegnamenti eventualmente presenti, e la prima parola di ciò che resta è il comando da eseguire

- da cercare tra i comandi built-in (anche funzioni o alias), oppure nelle directory della variabile `PATH`
- da notare che ora le parole possono anche contenere spazi al loro interno: il quote removal, prima di agire, prende nota di eventuali word con dentro degli spazi
- per esempio, scrivere
  `ls file1 file2` cerca 2 file, mentre
  `ls file1' 'file2` (o anche `ls file1\ file2`) cerca un solo file il cui nome è `file1 file2` (con lo spazio in mezzo)
- se non resta niente, gli assegnamenti eventualmente presenti hanno valore sulla shell corrente (in pratica: si possono scrivere sequenze di assegnamenti anche senza separarle con il `;`)