

Corso di Laboratorio di Sistemi Operativi

prima parte
la shell

obiettivi del corso

- uso di macchine con sistema operativo UNIX/Linux
- realizzazione di script per la shell sh
- conoscenza dei processi di UNIX e dei sistemi di sincronizzazione e comunicazione
- programmazione in C con invocazione dei servizi di sistema su file e processi

gli utenti

- la conoscenza minima del sistema è quella che ci permette di:
 - accedere ad una sessione di lavoro ed abbandonarla
 - visualizzare il contenuto del filesystem, copiare, spostare, cancellare, creare file e directory
 - visualizzare e modificare file
 - eseguire programmi e comandi
- il possesso di queste capacità ci rende utenti o *user* del sistema
- per ottenere l'accesso ad un sistema UNIX bisogna eseguire il LOGIN indicando il proprio *username* e la relativa *password*.
- UNIX controlla la corrispondenza e, in caso affermativo, avvia normalmente una shell con lo username specificato
- l'uscita dalla sessione si ottiene con *exit* o forzando un carattere di fine file con la sequenza di tasti CTRL-D

il superutente *root*

- chi amministra un sistema è l'utente *root* che ha particolari privilegi ma anche particolari responsabilità
- deve quindi possedere una conoscenza maggiore del sistema in merito a:
 - avvio ed arresto *ordinato*
 - gestione degli utenti e dei permessi
 - sicurezza
 - installazione di applicazioni condivise
 - servizi di rete e di comunicazione
- Anche l'utente *root* deve eseguire una operazione di login con username *root* e relativa password.
- E' possibile ottenere i diritti di *root* anche nella sessione di un utente standard con il comando *su*. Avendo la password!

la shell

- la shell è un programma che ‘ascolta’ le richieste degli utenti e si occupa dell’invocazione del sistema operativo per l’esecuzione dei comandi
- la modalità di lettura e validazione dei comandi è specificata dalla shell mediante la sintassi
- esistono diverse shell per UNIX/Linux. questo corso farà uso della Bourne Shell sh.
 - Nelle installazioni Linux viene normalmente preferita la shell BASH che rappresenta una estensione, comunque compatibile, di sh.
- la shell esegue un ciclo di:
 - lettura command line
 - esecuzione della command line
- fino alla ricezione di CTRL-D o di exit

richiesta di un comando

- l'utente chiede alla shell l'esecuzione di un comando digitando la *command line*
- la shell indica lo stato di *pronto* con una sequenza di caratteri detta *prompt*
- la command line è una sequenza di caratteri digitati al prompt seguiti dal tasto *invio*
- una command line può contenere l'invocazione di un comando esterno o di un *built-in* o entrambe più, eventualmente, degli argomenti
 - un built-in è un simbolo proprio della shell
 - un comando esterno è il nome di un file eseguibile
 - gli argomenti sono caratteri aggiuntivi *passati* al built-in o al comando esterno

un esempio di comando

- ls elenca i file presenti nella directory corrente
- l'invocazione più semplice è:

L **S** **INVIO**

- visualizza (lista) i file presenti nella directory corrente.

gli argomenti della linea di comando

- secondo una prassi consolidata e supportata dallo standard Posix: gli argomenti che iniziano con il simbolo '-' sono dette opzioni
- le opzioni sono costituite da un unico carattere e possono essere seguite da un argomento relativo all'opzione:

```
gcc -o eseguibile sorgente.c
```

- 'o' è una opzione, 'eseguibile' è l'argomento di '-o', 'sorgente.c' è un argomento di 'gcc'

file system

- il file system di UNIX è una struttura gerarchica di file, rappresentata dal grafo dei direttori
- in UNIX il file è una astrazione unificante; tutto ciò che si trova nel file system è file:
 - file ordinari
 - file direttori
 - file speciali
- un file può essere indicato:
 - in modo assoluto: `/home/alex/iso/esempio1.sh`
 - in modo relativo, dal direttorio corrente: `iso/esempio1.sh`
 - in modo relativo semplice, dal direttorio in cui si trova il file: `esempio1.sh`
- in UNIX i nomi dei file sono case-sensitive

lay-out tipico del filesystem

- solitamente il filesystem di UNIX presenta le seguenti directory principali:
- /bin eseguibili principali di sistema
- /boot sistema di avvio della macchina
- /dev file speciali per i device
- /etc file di configurazione
- /lib librerie di sistema
- /tmp file temporanei
- /usr file utente condivisi da più computer
 - nota: capita spesso, per esigenze di velocità e di sicurezza, di ‘montare’ la directory /usr in read-only e, in tal caso, i file scrivibili per log, spool, ecc. che si trovano in /usr vengono spostati in /var.

la home directory

- è una directory dedicata all'utente che normalmente ne possiede i diritti e la proprietà
- l'abbinamento fra nome utente e directory home è indicato nel file `/etc/passwd`
- la modifica, l'inserimento o la cancellazione di file dalla propria home non altera il funzionamento del sistema ma, eventualmente, solo quello dell'utente.
- l'utente può tipicamente creare sottodirectory nella propria home

proprietà dei file

- un file UNIX è caratterizzato da alcune informazioni di accesso/protezione
 - identificativo del proprietario
 - identificativo del gruppo del proprietario
 - flag dei diritti
- è possibile ottenere queste informazioni con il comando `ls` seguito dall'opzione `'l'`:

```
alex@gateway:~/work$ ls -l
```

```
total 3
```

```
-rw-rw-r--  1 alex    alex      40 Nov 30 04:08 file
drwxrwxr-x  2 alex    alex    1024 Dec 13 10:01 getopt
drwxrwxr-x  2 alex    alex    1024 Nov 30 03:52 pippo
```

diritti	nl	user	group	size	time	nome
----------------	-----------	-------------	--------------	-------------	-------------	-------------

permessi

- il primo carattere indica il tipo di file, - ordinario, d directory, ...
- i rimanenti 9 indicano i permessi per:
 - primi 3: utente proprietario
 - secondi 3: utenti appartenenti al gruppo
 - ultimi 3: altri utenti (tutti)
- i permessi sono codificati in un bitfield di 12 bit, 9 di protezione e 3 speciali, che vengono così decodificati da ls -l:
 - 'r' permesso in lettura, '-' lettura impedita
 - 'w' permesso in scrittura, '-' scrittura impedita
 - 'x' permessi di esecuzione, '-' non eseguibile
 - nella posizione della 'x' è possibile trovare anche 's' per indicare che l'esecuzione 'promuove' l'utente ai privilegi dell'utente o del gruppo proprietario
 - sempre nella posizione della 'x' è possibile trovare 't' per richiedere il salvataggio del segmento TEXT dell'eseguibile sul device di swap.

il comando ls

- la lista prodotta dal comando ls può essere parametrizzata mediante opzioni:
 - l abbina al nome le informazioni associate al file, ovvero tipo, permessi, numero di link, proprietario e gruppo, dimensione e data/ora dell'ultima modifica
 - a non nasconde i nomi dei file che iniziano con .
 - A come -a ma esclude i file particolari '.' e '..' per le directory corrente e superiore
 - F postpone il carattere '*' agli eseguibili e '/' ai direttori
 - d lista il nome delle directory senza listarne il contenuto
 - R percorre ricorsivamente la gerarchia
 - i indica gli i-number dei file oltre al loro nome
 - r inverte l'ordine dell'elenco
 - t lista i file in ordine di modifica
- Le opzioni di ogni comando possono essere combinate assieme: ls -l -a = ls -la

il comando chmod

- è possibile modificare la maschera dei permessi di un file, sia esso ordinario o di direttorio, mediante il comando *chmod*

- la sintassi di invocazione del comando è:

`chmod [u g o] [+ -] [r w x s t] nomefile`

`chmod numero_ottale nomefile`

- nel primo caso, specificazione simbolica:
 - si operano le variazioni della maschera per il proprietario (u-ser), il gruppo a cui appartiene il proprietario (g-roup) o per i rimanenti utenti (o-thers)
 - concedendo (+) o rimuovendo (-) i privilegi di
 - lettura (r), scrittura (w), esecuzione per file o esplorazione per directory (x); è possibile anche richiedere che l'eseguibile ottenga lo user id del proprietario o il group id del gruppo (s) o permettere la copiatura del segmento text dell'eseguibile sullo swap device (t)
 - è possibile combinare più specificazioni simboliche separandole con la virgola
- nel secondo caso, specificazione ottale:
 - si forza la maschera di bit del file al valore indicato

i comandi mkdir e cd

- la creazione di una directory si ottiene con il comando:
mkdir nomedirectory
- se nomedirectory è specificato in modo *relativo semplice* viene creata una directory sotto alla directory corrente
- in caso contrario, viene creata una directory sotto quella corrente (*relativo*) o sotto la radice (*assoluto*), nel percorso indicato solo se questo esiste
- la directory corrente è visibile con il comando
pwd
- è possibile spostare la directory corrente con
cd nuovadirectory

i comandi cat e more

- cat è un comando che permette di visualizzare il contenuto di un file, riversandone il contenuto sul proprio standard output che, normalmente, corrisponde con il video della console
 - se invocato senza argomenti, legge i caratteri da tastiera e li riversa sul terminale fino al particolare carattere di 'fine file' ottenibile con CTRL-D
 - se si specifica un argomento, cat cerca un file con quel nome e lo visualizza.
- se il file da visualizzare non può essere contenuto nello schermo, è possibile usare il comando more che arresta lo scorrimento delle linee a fine schermo.
- esiste una estensione di more, denominata less, che permette anche di ritornare alle pagine precedenti

documentazione *in linea*

- il comando 'man' visualizza la pagina del manuale associata al comando passato come argomento.
- 'man man' fornisce le istruzioni d'uso di man
- normalmente, in un sistema UNIX/Linux, sono installate le pagine man per i comandi di shell e per le funzioni di libreria
- in caso di omonimia è possibile specificare la *sezione* del manuale in cui cercare:
 - man 1 mkdir
 - man 2 mkdir
 - Attenzione: sulle Sun la sintassi di man per richiedere una sezione specifica è diversa, richiedendo l'inserimento dell'opzione 'section'
- nei sistemi 'GNU' esiste anche il comando 'info'.
- si esce da man e info premendo 'q'

le utility

- sono programmi di utilità generalmente distribuiti con il sistema operativo per:
- cercare file:
 - find
 - grep
- modificare file in modo ‘automatico’:
 - sed
 - awk
- archiviare:
 - tar/gzip/bz2
 - rcs/cvs

le utility (2)

- generare programmi
 - make
 - cc/gcc/as
 - lex/flex/yacc/bison
- creare e modificare file di testo
 - vi/ex/vim
 - emacs
- manualistica
 - man
 - info
- tipicamente, ogni utente può invocare queste utility

ridirezione

- ogni comando utilizza dei file standard per l'input, l'output ed i messaggi di errore.
- questi file sono indicati con *stdin*, *stdout*, *stderr*
- normalmente, un comando eseguito da terminale preleva l'input da tastiera ed emette output e messaggi d'errore a video
- questo comportamento può essere modificato con la ridirezione.
 - è possibile, ad esempio, riversare il risultato di un comando su un file:
 - `ls > lsfile`
 - *lsfile* contiene l'intero output del comando *ls*
 - `ls >> lsfile`
 - appende l'output di *ls* al file *lsfile* esistente.
 - è possibile ridirigere l'input in modo da prelevare i dati da un file
 - `cat < lsfile`
 - i descrittori dei file *standard* (0 per *stdin*, 1 per *stdout*, 2 per *stderr*) possono essere utilizzati per eseguire delle ridirezioni specifiche
 - `ls file_non_esistente 2> lsfile`
 - ridirige il messaggio di errore di *ls* al file *lsfile*

ridirezione (2)

- i simboli per la ridirezione vanno inseriti dopo l'invocazione del comando e dopo gli eventuali argomenti:
 - `>` ridirige lo standard output
 - `<` ridirige lo standard input
 - `2>` ridirige lo standard error
 - `n>` ridirige il descrittore *n*
- la ridirezione può avvenire verso un file, specificandone il nome dopo il simbolo di ridirezione, o ad un altro handler aperto:
 - `2>&1` ridirige lo standard error allo standard output
 - Attenzione: l'ordine della ridirezione è importante
 - comando `> file 2>&1` ridirige il canale 1 (`>`) a 'file' ed il canale 2 (`2>`) al canale 1 (`&1`) dopo che questo è stato ridiretto a 'file'. Il risultato è che tutto viene riversato in 'file'
 - comando `2>&1 >file` ridirige il canale 2 (`2>`) al canale 1 (`&1`) che è attualmente la console, poi ridirige il canale 1 (`>`) a 'file'. Il risultato è che lo standard output va a 'file' e lo standard error va a console.

pid

- i processi sono entità astratte proprie di UNIX e dei suoi derivati, identificate da un numero intero detto *pid*, da un proprio spazio di memoria, da una propria copia delle variabili d'ambiente e da un elenco di descrittori di file.
- la shell è un processo; l'esecuzione di un comando viene delegata ad un nuovo processo creato dalla shell (figlio) che può attenderne la conclusione (processo in foreground) o no (processo in background)
- un comando viene eseguito da un processo in background se si postpone al comando stesso il carattere &
- l'elenco dei processi si ottiene con il comando ps che visualizza anche l'identificatore del processo o pid
- i processi possono interagire fra loro mediante il file system (risorsa condivisa), mediante comunicazioni inter-processo e mediante segnali

pipe

- la pipe rappresenta un metodo di comunicazione inter-processo fornito dal sistema operativo; consiste in una particolare ridirezione che 'collega' lo standard input di un comando/processo allo standard output di un altro comando/processo
- la ridirezione semplice consente il concatenamento di più comandi mediante un file intermedio:
 - `ps ax > tmp.file; grep apache < tmp.file; rm tmp.file`
- la concatenazione con una pipe non richiede il file intermedio in quanto è il sistema operativo stesso a gestire il canale di comunicazione senza appoggiarsi su file (come invece avviene per la pipe di MSDOS) e sincronizzando opportunamente i due processi
- `ps ax | grep apache`

fifo

- è possibile intercettare i dati in una pipeline mediante l'inserimento di *derivazioni a T* con il comando *tee*

```
cat index.html | tee inspector.file | grep a
```

- è anche possibile creare delle pipe manualmente ed associarvi un nome nel filesystem con *mkfifo*. una *named-pipe* è denominata *fifo*

```
mkfifo np; grep a <np & cat index.html>np ; rm np
```

- la *fifo np*, a differenza di un file ordinario temporaneo, sfrutta i meccanismi di sincronizzazione del sistema operativo

segnali

- i segnali sono oggetti che il sistema operativo mette a disposizione per sincronizzare i processi
- un segnale particolare è quello di KILL che permette ad un processo *dotato di certi requisiti* di eliminare un altro.
- kill è anche un built-in della shell che permette l'invio di segnali ad un processo di cui si conosce il pid; il segnale inviato è normalmente TERM(15) se non si specifica diversamente.
- i segnali, ad eccezione di KILL, possono essere intercettati da uno shell script per eseguire certe operazioni
- l'intercettazione di un segnale si ottiene con il built-in *trap*
`trap 'echo "CTRL-c non ammesso"' 2`
- blocca il segnale di break (2) impedendo il blocco dello script

segnali (2)

- un processo può inviare un segnale ad un altro processo solo se:
 - il processo *inviante* è del super-user (UID=0)
 - l'utente proprietario del processo *inviante* è il proprietario anche del processo *ricevente*. un processo può segnalare ai propri discendenti
- trap può essere usato anche per tracciare uno script

```
#!/bin/sh
# debug trapping
trap 'echo "Il valore e'''' stato modificato a:" $valore' DEBUG
# viene invocato ad ogni comando
valore=1
echo valore inizializzato
a=`expr $a + 1`
echo valore modificato
```

l'editor vi

- vi è un text editor che è normalmente presente in ogni macchina UNIX, eventualmente in una delle sue evoluzioni *nvi*, *elvis*, *vim*.
- il comportamento di vi è *modale*:
 - all'avvio vi si trova, normalmente, in modalità comandi
 - la digitazione del testo si esegue in modalità input, dalla quale si esce premendo ESC
- quasi tutte le versioni attuali di vi accettano i comandi di movimento anche dai tasti freccia ma, storicamente, lo spostamento del cursore è ottenibile con i tasti hjkl in modalità comandi:

k = su

h = sinistra

l = destra

j = giu'

comandi interattivi e batch

- la shell può interpretare comandi digitati direttamente dall'utente o comandi depositati in un file.
- in questo secondo caso, il file con i comandi viene denominato *shell script*.
- l'invocazione di uno shell script può essere richiesta con:
sh shell_script
sh <shell_script
./shell_script
- la seconda forma utilizza la redirectione (v.oltre), la terza richiede che *shell_script* abbia il flag 'eseguibile' settato (v.oltre)
- nel terzo caso, lo script deve contenere un *codice* per selezionare l'interprete (bash, csh, tcsh, perl, ...)

magic number

- un file in unix è eseguibile se ha il corrispondente flag settato.
- la modalità di esecuzione non dipende dal nome del file o dall'estensione (come per DOS e Win) ma da un codice 'magic' posto all'inizio del file
- i file contenenti shell script sono marcati con i due caratteri iniziali:

`#!`

- questa coppia è spesso indicata come *sha-bang*
- *di seguito è normalmente indicato l'interprete:*

`#!/bin/bash`

- la shell interpreta `#` come inizio commento, quindi la prima linea non interferisce con il contenuto dello script

Il primo script

- La creazione di uno script richiede un editor
- un editor normalmente presente è **vi**, o una sua estensione come **vim** o **elvis**
- editor alternativi sono **emacs** della FSF o **mcedit** distribuito con la suite **midnight commander** che clona il **Norton Commander**,
- esistono anche editor commerciali e/o dedicati all'esecuzione in ambiente grafico **X**.
- Gli esempi che seguono fanno riferimento all'editor **vi**.
- Per creare un file con vi è sufficiente invocare:

```
vi nomefile
```

Il primo script (2)

- Vi mostra le linee vuote del file con il simbolo ‘~’
- lo spostamento del cursore si ottiene con i tasti *freccia* o con i quattro tasti **h j k l**
- l’inserimento di caratteri si ottiene entrando in **modalità inserimento** con il tasto **i**
- si esce dalla modalità inserimento con il tasto **esc**
- si salva il file digitando **:w**, ovviamente non in modalità inserimento,
- si esce digitando **:q**
- per salvare ed uscire si può usare **ZZ**
- *in lab: leggere la man page di vi (o vim) e sperimentare le principali opzioni*

Il primo script (3)

- Posizionarsi nella propria home: `cd[invio]`
- Creare il file: `vi miols[invio]`
- inserire lo sha-bang: `i#!/bin/sh[invio]`
- inserire il comando da eseguire: `/bin/ls -la[invio]`
- salvare: `[esc]:w`
- uscire da vi: `:q`
- rendere lo script eseguibile: `chmod a+x miols[invio]`
- provare lo script: `./miols[invio]`
- *in lab: variare le opzioni di ls e sostituire ls con altri comandi*

parametrizzare lo script

- il comportamento di uno shell script può dipendere da variabili:
 - inserite nella linea di comando dello script ed indicate appunto come *argomenti della linea di comando*
 - definite nell'ambiente di esecuzione mediante il built-in *export* e denominate *variabili di ambiente*
- l'accesso a queste variabili si ottiene con:
 - i-esimo argomento: $\$1$, $\$2$, $\$3$, ..., $\${10}$, ... $\$0$ =nome comando
 - numero di argomenti: $\$#$
 - valore della variabile di ambiente *nomevar*.
- il *parsing* degli argomenti della linea di comando può essere eseguita elegantemente con il comando *getopt*.
- uno shell script può creare o modificare variabili di ambiente e variabili locali.

esempi

```
#!/bin/sh
#visualizza il primo arg
echo lo script $0 ha \
rilevato l\'argomento \
$1
```

- il carattere apice ' deve essere preceduto dalla backslash (escaping) altrimenti verrebbe interpretato dalla shell.
- il backslash a fine linea serve ad evitare (escaping) il salto linea.

```
#!/bin/sh
#visualizza il valore di
#PATH
echo il valore della \
variabile PATH e\' : $PATH
```

- il backslash non evita il fine linea se si trova in un commento

esempi

```
#!/bin/sh
#esegue il comando
#indicato come primo
#argomento
$1
```

- la shell interpreta i comandi quindi può espandere i valore delle variabili a 'run-time'

```
#!/bin/sh
#crea una var e la
#visualizza
miavariabile=valore
echo $miavariabile
```

- la shell esegue lo script in un processo (v.oltre) figlio della shell stessa, non è possibile esportare una variabile al processo 'padre'

caratteri speciali e quoting

- i caratteri “*” e “?” sono interpretati dalla shell in modo particolare:
 - * sostituisce ogni sequenza di qualsiasi caratteri
 - ? sostituisce un qualsiasi carattere
- la shell li espande automaticamente salvo quando vengono ‘quotati’ *quoting*

echo * #mostra tutti i nomi di file nella dir.

- il quoting può bloccare l’interpretazione anche di altri caratteri speciali, come \$, ` e \.
- esiste il full quoting con apici ‘ ed il partial quoting con le virgolette “

strutture di controllo della shell

le strutture di controllo sono quegli elementi del linguaggio che permettono di codificare decisioni e ripetizioni di comandi

- la verifica di una condizione si ottiene con il built-in *test* o equivalentemente con la parentesi quadra aperta *[*.
- i cicli si ottengono con *for*, *while*, e *until*; elementi opzionali nei cicli sono *break* e *continue*
- la scelta fra più possibilità si ottiene con *case*
- un insieme di istruzioni può essere racchiuso in una funzione *myfunc ()*
- più comandi possono essere ‘collegati’ fra loro mediante la ridirezione su file ed il piping.

if

if condizione

then

comandi

fi

- *condizione* è un qualsiasi comando o built-in del quale viene verificato il valore di ritorno: 0=vero, 1=falso
- *comandi* è un qualsiasi comando, built-in o sequenza di questi
- il ritorno a capo dopo *then* è opzionale
- il ritorno a capo dopo *condizione* e prima di *fi* è necessario ma può essere sostituito con un punto e virgola ;

if condizione; then comando; fi

test

- un built-in molto utile nel costrutto if è il test o [
`if test -z "$1"; then echo OK; fi`
`if [-z "$1"]; then echo OK; fi`
- la parentesi quadra di chiusura non è necessaria anche se consigliata per leggibilità
- bash mette a disposizione la keyword `[[]]` che esegue il test come `[]` permettendo costruzioni logiche con `&&` e `||`
- le opzioni di test sono numerose, man `builtin` ne da un'elenco dettagliato
- è possibile inserire un `else` o `elif`
- è possibile annidare più condizioni if

espressioni condizionali per test o [

- `-d file` true se file esiste ed è una directory
- `-e file` true se file esiste
- `-f file` true se file esiste ed è un file regolare
- `-r file` true se file esiste ed è leggibile
- `-s file` true se file esiste ed ha dimensione maggiore di 0 (non vuoto)
- `-w file` true se file esiste ed è scrivibile
- `-x file` true se file esiste ed è eseguibile
- `file1 -nt file2` true se file1 ha data di modifica posteriore di file2 (è più 'nuovo')
- `file1 -ot file2` true se file1 è più vecchio di file2
- `-z string` true se la lunghezza di string è zero (vuota)
- `-n string` true se string è non vuota
- `string1 = string2` true se le stringhe sono uguali
- `string1 != string2` true se sono diverse
- `arg1 OP arg2` verifiche aritmetiche sugli operandi arg1 e arg2: OP può valere:
 - `-eq` uguaglianza numerica
 - `-ne` diversità numerica
 - `-lt` arg1 minore (strettamente) di arg2
 - `-le` arg1 minore o uguale ad arg2
 - `-gt` arg1 maggiore (strettamente) di arg2
 - `-ge` arg2 maggiore o uguale ad arg2

liste

- leggendo la documentazione della shell `man bash` si trovano molti richiami al concetto di *list*; uno script è una lista.
- una lista è una sequenza di uno o più comandi separati fra loro da `;` `&` `&&` `||` o fine linea, eventualmente terminata da `;` `&` o fine linea
- il risultato di una lista che non contiene compositori logici `&&` o `||` è il risultato dell'ultimo comando
- gli operatori `&&` e `||` eseguono l'AND e l'OR rispettivamente dei comandi, ma non è garantita l'esecuzione di entrambe.
- `;` e fine linea sono equivalenti
- `&` esegue il comando che precede in background senza attenderne il completamento; il valore di ritorno è 0 (=vero)

pipeline

- il termine *comandi* indicato nel lucido precedente non è quello indicato nel manuale della shell: si parla di *pipeline*
- una pipeline è la generalizzazione di un comando, essendo la composizione di uno o più comandi in sequenza separati dal carattere | (pipe)
- ogni comando in una pipeline ottiene in ingresso l'uscita del comando che lo precede e fornisce il proprio output in ingresso al comando che lo segue
- il primo comando prende l'input (standard input) dall'attuale standard input, l'ultimo emette i risultati sullo standard output corrente.
- il valore di ritorno di una pipeline è quello dell'ultimo comando

for

- il costrutto for serve ad eseguire una lista di comandi ripetutamente con una variabile che assume valori presi da un elenco

```
#!/bin/sh
#esempio di for
for a in 1 2 3 4 5 6
do
    echo \$a vale $a
done
```

- a assume i valori 1,2,3,4,5 e 6 e, ad ogni valore, viene eseguito il comando echo
- se viene omesso 'in ...' la variabile assume i valori degli argomenti della command line

elenchi *particolari*

- la shell mette a disposizione delle agevolazioni per accedere ai file

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

```
#semplice clone di ls
```

```
for filename in *
```

```
do
```

```
    echo $filename
```

```
done
```

- * espande tutti i nomi di file ad eccezione di quelli che iniziano con un dot .
- `shopt -s dotglob` abilita l'espansione anche dei dot files (solo bash)

while

- il costrutto while serve ad eseguire ripetutamente dei comandi fintanto che è vera una condizione (test)

```
#!/bin/sh
# esempio di while
i=1
while [ $i -le 10 ]
do
    echo $i
    i=`expr $i + 1`
done
```

until

- il costrutto `until` è simile a `while`, con l'unica eccezione che la ripetizione dei comandi avviene se la condizione è falsa.

```
#!/bin/sh
# esempio di until
i=1
until [ $i -gt 10 ]
do
    echo $i
    i=`expr $i + 1`
done
```

break, continue

- all'interno di qualsiasi ciclo for, while e until possono essere invocate le keyword break e continue
- break provoca l'immediata uscita dal ciclo
- continue provoca l'interruzione della lista di comandi nel ciclo per l'iterazione corrente, passando alla successiva esecuzione del ciclo, ad esempio con il successivo valore del for

```
if [ condizione ]  
then  
continue # oppure break  
fi
```

case

- il costrutto case serve a selezionare una lista di comandi fra più possibilità in funzione del valore di una variabile

```
#!/bin/sh
# esempio case
case "$1" in
    "1" )
        echo uno;;
    "2" )
        echo due;;
    *) #default
        echo altro;;
esac
```

case (2)

- ogni condizione del case può essere indicata anche come elenco di possibilità (separate da |) o come range di valori (rappresentate da [a-z])
- le linee di condizione devono essere terminate da)
- la lista di comandi corrispondente ad una condizione deve essere terminata da ;;
- l'intero costrutto case deve essere terminato dalla keyword esac, ovvero case al contrario (anche if è terminato da fi).
- la condizione di *default* nella quale si ricade se nessuna delle precedenti ha avuto successo, è *)
- la prima condizione che risulta vera viene eseguita ed il controllo esce dal case; se vi sono più condizioni vere, solo la prima viene eseguita.

esempi di pattern per 'case'

- L'uso del case può essere 'potenziato' utilizzando, nella specificazione dell'argomento di confronto, dei pattern.
- Ecco alcuni esempi:

```
case string in
```

```
[123]          ) echo string vale "1", "2" o "3" ;;
```

```
?             ) echo string ho lunghezza unitaria ;;
```

```
[123][456]    ) echo string vale "14" o 15,16,24,25,26,34,35,36 ;;
```

```
[^123][456]   ) echo string vale "44", "w5", ... ;;
```

```
aa*           ) echo string inizia per "aa" ;;
```

```
*            ) echo string non ricade in nessuna delle precedenti;;
```

- case può essere utilizzato, ad esempio, dentro un for con globbing per verificare la corrispondenza del nome a certi criteri
- in alternativa è possibile utilizzare il 'grep' in un 'if' nella forma:

```
if echo $filename | grep 'aa*';
```

```
then ...
```

- la pipeline echo | grep torna il risultato dell'ultimo (grep) che vale vero (0) se trovato

read

- per richiedere input all'operatore tramite lo standard input si usa `read a`

- è la forma più semplice di `read`, mette tutti i caratteri in 'a' fino al fine linea

```
read a b c
```

- permette di leggere più variabili separate da spazi

```
read -p "Digita: " a
```

- inserisce anche il messaggio specificato
- se le parole separate da spazi sono più dei nomi di variabili, l'ultima variabile contiene anche le parole eccedenti.

select

- questo è una estensione di bash per richiedere all'operatore una scelta fra varie opzioni fissate.
- permette la selezione di una sola voce di un *menu* personalizzabile, al quale l'utente risponde con il numero corrispondente
- il *prompt* in questo caso si modifica (PS3)

```
#!/bin/sh
# esempio di select
select a in uno due tre
do
    echo $a
    break
done
```

command substitution

- serve sostituire ad una espressione il suo output, si ottiene includendo l'espressione fra *backticks* o apostrofo rovesciato (ascii 96)

```
echo ls
```

- scrive i caratteri 'l' e 's'

```
echo `ls`
```

- scrive l'output del comando ls
- si può ottenere anche nella forma

```
echo $(ls)
```

- spesso si trova il built in *eval* come comando da eseguire

```
echo `eval ls`
```

set

- set visualizza le variabili di ambiente se invocato senza argomenti
- set ammette numerosi parametri:

`set -e`

- forza l'uscita dallo script appena un comando restituisce una condizione falsa, utile per invocare un elenco di comandi dipendenti

`set -n`

- visualizza i comandi ma non li esegue, utile per verificare uno script
- altri usi sono indicati nel manuale

funzioni

- è possibile strutturare la programmazione di uno shell script definendo delle funzioni

```
#!/bin/sh
# funzioni
function f1
{
    echo sono f1
}
f2 () {
    echo sono f2
}
f1 #invoca f1
f2 #invoca f2
```

variabili locali

- all'interno di una funzione è possibile creare delle variabili visibili solo dall'interno.
- queste sono dette *variabili locali*

```
#!/bin/sh
# variabili locali
f1 () {
    local a=10
    echo $a
}
f1 # scrive '10'
echo $a # non scrive nulla
```

ricorsione

- è possibile utilizzare la ricorsione in uno script
- la ricorsione si può ottenere creando funzioni, dette *ricorsive*, che invocano loro stesse.

```
#!/bin/sh
# fattoriale
myfact () {
    if [ $1 -gt 1 ]; then
        echo $(expr $1 \* `myfact $(expr $1 - 1)` )
    else
        echo 1
    fi
}
echo `myfact $1`
```

ricorsione per processi

- la ricorsione può essere ottenuta anche invocando ripetutamente l'intero shell script.
- in questo caso, ogni iterazione genera (almeno) un processo

```
#!/bin/sh
# fattoriale con ricorsione
#+per processi
if [ $1 -gt 1 ]; then
    echo $(expr $1 \* `sh factp.sh $(expr $1 - 1)` )
else
    echo 1
fi
```

approccio ricorsivo a strutture ricorsive

- l'utilizzo più *naturale* della ricorsione è relativo alle operazioni sulle directory
- esempio: elencazione di tutti i file in una directory e nelle relative sottodirectory

```
#!/bin/sh
# tree
visualizza () {
    for filename in *; do
        if [ -d $filename ]
        then
            echo In $filename :
            cd $filename
            visualizza
            cd ..
        else
            echo $filename
        fi
    done
}
visualizza
```

ridirezione (3)

- la ridirezione può essere limitata a blocchi di comandi di uno script:

```
#!/bin/sh
# ridirezione di un blocco
a="qualcosa"
while [ "$a" != "" ]
do
    read a
    echo $a
done < tmp.file > tmpfile.copy
```

here document

- E' possibile inserire in un file dei caratteri presenti nello script mediante l'uso dello 'here document'

- Esempio:

```
cat > tmp.file <<-EOF
```

```
Questa e' una prova  
di here document
```

```
EOF
```

- Tutto ciò che segue l'istruzione con '<<' viene riversato sullo standard input del comando fino alla lettura della stringa di terminazione (nell'esempio "EOF").
- Il '-' opzionale serve ad eliminare le tabulazioni all'inizio di ogni linea dal file in modo da permettere comunque l'identazione.

debugging

- non esistono tool di debugging per shell script
- l'autore di uno script deve porre attenzione alla *debuggabilità*
- con:
 - tracing disabilitabile
 - assert
 - verifiche esaustive
- si possono usare:
 - gli switch nell'invocazione di sh (o bash):
 - -n: non vengono eseguiti i comandi, controllo sintattico
 - -v, -x: visualizza i comandi prima di eseguirli
 - copie dei dati temporanei con tee o named pipe
 - signal trapping DEBUG, EXIT (0).

sicurezza

- ogni operazione sul filesystem operata da uno shell script non è sicura!
- il filesystem è un oggetto condiviso e non protetto da meccanismi di sincronizzazione:

```
if [ -e $filename ]  
then  
    # qualche altro processo cancella $filename  
    rm $filename # errore  
fi
```

- uno script affidabile deve intercettare tutte le possibili cause di malfunzionamento in quanto non è possibile escluderle del tutto