Programació d'ordres combinades (*shell scripts*)

Remo Suppi Boldrito

PID_00215356

UOC
Universitat Oberta de Catalunya

Índex

Introducció

En aquest mòdul veurem la importància fonamental que té l'intèrpret d'ordres o instruccions (*shell*) i sobretot analitzarem amb cert detall les seves possibilitats per a executar fitxers d'ordres seqüencials i escrits en text pla (ASCII), que seran interpretats pel *shell*. Aquests fitxers, anomenats *shell scripts*, són l'eina fonamental de qualsevol usuari avançat, i imprescindibles per a un administrador de sistemes *nix.

Un coneixement pràctic de *shell scripting* serà necessari, ja que el mateix GNU/Linux es basa en aquest tipus de recurs per a iniciar el sistema (per exemple, executant els arxius de */etc/rcS.d*), per la qual cosa els administradors han de tenir els coneixements necessaris per a treballar-hi, entendre el funcionament del sistema, modificar-los i adequar-los a les necessitats específiques de l'entorn en el qual treballin.

Molts autors consideren que fer *scripting* és un art, però, segons el nostre parer, no és difícil d'aprendre, ja que es poden aplicar tècniques de treballar per etapes (*divide & conquer*), ja que s'executarà de manera seqüencial i el conjunt d'operadors/opcions no és tan extens perquè generi dubtes sobre quin recurs s'ha d'utilitzar. Sí que la sintaxi serà dependent del *shell* utilitzat, però en ser un llenguatge interpretat amb encadenament de seqüències d'ordres serà molt fàcil de depurar i posar en funcionament.

Avui dia l'*scripting* ha arribat a molts àmbits amb la potencialitat presentada per alguns llenguatges com PHP per a desenvolupar codi del costat del servidor (originalment dissenyat per al desenvolupament web de contingut dinàmic); Perl, que és un llenguatge de programació (1987) que pren característiques de C, de *Bourne shell*, AWK, `sed`, Lisp entre d'altres; Phyton (1991), la filosofia del qual posa l'accent en una sintaxi que afavoreixi un codi llegible, essent un llenguatge de programació (interpretat) que suporta orientació a objectes, programació imperativa i, en menys mesura, programació funcional; i amb tipus dinàmics o com Ruby (1993-1995), que és un llenguatge de programació (també interpretat), reflexiu i orientat a objectes, que combina una sintaxi inspirada en Python i Perl amb característiques de programació orientada a objectes similars a Smalltalk.

També, un *shell script* és un mètode *quick-and-dirty* (que es podria traduir com a ràpid i en brut) de prototipatge d'una aplicació complexa per a aconseguir fins i tot un subconjunt limitat de la funcionalitat útil en una primera etapa del desenvolupament d'un projecte. D'aquesta manera, l'estructura de l'aplicació i

les grans línies es poden provar i es pot determinar quines seran les dificultats principals abans de procedir al desenvolupament de codi en C, C++, Java, o un altre llenguatge estructurat i/o interpretat més complex.

D'acord amb M. Cooper hi ha una sèrie de situacions en les quals ell categòricament indica que "no s'han d'utilitzar *shell scripts*", si bé al nostre entendre diríem que s'haurà de ser prudent i analitzar en profunditat el codi desenvolupat sobretot des del punt de vista de la seguretat per a evitar que es puguin dur a terme accions més enllà d'aquelles per a les quals es van dissenyar (per exemple, en *scripting* per la banda del servidor en aplicacions web és una de les causes principals d'intrusió o execució no autoritzades). Entre les causes podem enumerar: ús intensiu de recursos, operacions matemàtiques d'alt rendiment, aplicacions complexes en què es necessiti estructuració (per exemple llistes, arbres) i tipus de variables, situacions en les quals la seguretat sigui un factor determinant, accés a arxius extensos, matrius multidimensionals, treballar amb gràfics, accés directe al maquinari/comunicacions, No obstant això, per exemple, amb la utilització de Perl, Phyton o Rugbi, moltes d'aquestes recomanacions deixen de tenir sentit i en alguns àmbits científics (per exemple, en genètica, bioinformàtica, química, etc.) l'*scripting* és la forma habitual de treball.

Aquest capítol està orientat a recollir les característiques principals de Bash (*bourne-again shell*), que és el *shell script* estàndard en GNU/Linux, però molts dels principis explicats es poden aplicar a Korn Shell o C Shell.

1. Introducció: l'intèrpret d'ordres

L'intèrpret (*shell*) és una peça de programari que proporciona una interfície per als usuaris en un sistema operatiu i que proveeix accés als serveis del nucli. El seu nom anglès prové de l'embolcall extern d'alguns mol·luscs, ja que és la "part externa que protegeix el nucli".

Els intèrprets (o *shells*) es divideixen en dues categories: línia d'ordres i de gràfics, depenent de si la interacció es fa mitjançant una línia d'instruccions (CLI, *command line interface*) o en mode gràfic per mitjà d'una GUI (*graphical user interface*). En qualsevol categoria l'objectiu principal de l'intèrpret és invocar o "llançar" un altre programa; tanmateix, solen tenir capacitats addicionals, com ara veure el contingut dels directoris, interpretar ordres condicionals, treballar amb variables internes, gestionar interrupcions, redirigir entrada/sortida, etc.

Si bé un intèrpret gràfic és agradable per a treballar i permet a usuaris sense gaires coneixements exercir-se amb certa facilitat, els usuaris avançats prefereixen els de mode text, ja que permeten una manera més ràpida i eficient de treballar. Tot és relatiu, ja que en un servidor és probable que un usuari administrador no utilitzi ni tan sols interfície gràfica, mentre que un usuari d'edició de vídeo mai no treballarà en mode text. El principal atractiu del sistema *nix és que també en mode gràfic es pot obrir un terminal i treballar en mode text com si estigués en un intèrpret de text, o canviar interactivament del mode gràfic i treballar en mode text (fins amb 6 terminals diferents amb Ctrl-Alt-F1... F6, generalment) i després tornar al mode gràfic amb una simple combinació de tecles (Ctrl-Alt-F7). En aquest apartat ens dedicarem a l'intèrpret en mode text i a les característiques avançades en la programació de *shell scripts*.

Entre els intèrprets més populars (o històrics) en els sistemes *nix tenim:

- Bourne *shell* (*sh*).
- Almquist *shell* (*ash*) o la seva versió de Debian (*dash*).
- Bourne-Again *shell* (*bash*).
- Korn *shell* (*ksh*).
- Z *shell* (*zsh*).
- C *shell* (*csh*) o la versió Tenex C *shell* (*tcsh*).

El Bourne *shell* ha estat l'estàndard *de facto* en els sistemes *nix, ja que va ser distribuït per Unix Version 7 el 1977, i Bourne-Again (Bash) és una versió millorada del primer escrita pel projecte GNU sota llicència GPL, la qual s'ha transformat en l'estàndard dels sistemes GNU/Linux. Com ja hem comentat, Bash serà l'intèrpret que analitzarem, ja que, a més de ser l'estàndard, és molt

potent, té característiques avançades, recull les innovacions plantejades en altres intèrprets i permet executar sense modificacions (generalment) scripts fets per a qualsevol intèrpret amb sintaxi compatible amb Bourne *shell*. L'ordre `cat /etc/shells` ens proporciona els intèrprets coneguts pel sistema Linux, independentment de si estan instal·lats o no. L'intèrpret per defecte per a un usuari s'obtindrà de l'últim camp de la línia corresponent a l'usuari del fitxer */etc/passwd* i canviar d'intèrpret simplement significa executar el nom de l'intèrpret sobre un terminal actiu, per exemple:

```
RS@DebSyS:~$ echo $SHELL
/bin/bash
RS@DebSyS:~$ tcsh
DebSyS:~>
```

Una part important és el que es refereix als modes d'execució d'un intèrpret. S'anomena *login* interactiu o *entrada interactiva* quan prové de l'execució d'una entrada, fet que implicarà que s'executaran els arxius */etc/profile*, *~/.bash_profile*, *~/.bash_login* o *~/.profile* (la primera de les dues que existeixi i es pugui llegir) i *~/.bash_logout* quan acabem la sessió. Quan és de no-connexió interactiu (s'executa un terminal nou) s'executarà *~/.bashrc*, ja que un *bash* interactiu té un conjunt d'opcions habilitades respecte a si no ho és (consulteu el manual). Per a saber si l'intèrpret és interactiu, executeu `echo $-` i ens haurà de respondre **himBH**, en què ***i*** indica que sí que ho és.

Hi ha un conjunt d'ordres internes[1] en l'intèrpret, és a dir, integrats amb el codi d'intèrpret, que per a Bourne és:

(1)Consulteu el manual per a una descripció completa.

`:, ., break, cd, continue, eval, exec, exit, export, getopts, hash, pwd, readonly, return, set, shift, test, [, times, trap, umask, unset.`

I a més el Bash inclou:

`alias, bind, builtin, command, declare, echo, enable, help, let, local, logout, printf, read, shopt, type, typeset, ulimit, unalias.`

Quan el Bash executa un *shell script*, crea un procés fill que executa un altre Bash, el qual llegeix les línies de l'arxiu (una línia per vegada), les interpreta i executa com si vinguessin del teclat. El procés Bash pare espera mentre el Bash fill executa l'script fins al final, en què el control torna al procés pare, el qual torna a posar l'indicador o *prompt* novament. Una ordre d'intèrpret és una cosa tan simple com `touch arxiu1 arxiu2 arxiu3`; consisteix en l'ordre seguida d'arguments, separats per espais, *arxiu1* és el primer argument i així successivament.

1.1. Redireccions i *pipes*

Hi ha tres descriptors de fitxers: *stdin, stdout* i *stderr* (l'abreviatura *std* significa 'estàndard'), i en la majoria dels intèrprets (fins i tot en Bash) es pot redirigir *stdout* i *stderr* (junts o separats) a un fitxer, *stdout* a *stderr* i viceversa. Totes es representen per un número; 0 representa *stdin*, 1, *stdout*, i 2, *stderr*.

Per exemple, enviar *stdout* de l'ordre *ls* a un fitxer serà `ls -l > dir.txt`, en què es crearà un fitxer anomenat *dir.txt*, que contindrà el que es veuria a la pantalla si s'executés `ls -l`

Per a enviar la sortida *stderr* d'un programa a un fitxer, escriurem `grep xx yy 2> error.txt`. Per a enviar l'*stdout* a l'*stderr*, farem `grep xx yy 1>&2` i a la inversa simplement intercanviant l'1 per 2, és a dir, `grep xx yy 2>&1`.

Si volem que l'execució d'una ordre no generi activitat per pantalla, la qual cosa es denomina *execució silenciosa*, només hem de redirigir totes les seves sortides a */dev/null*; per exemple, pensant en una ordre del `cron` que volem que esborri tots els arxius acabats en .mov del sistema: `rm -f $(find / -name "*.mov") &> /dev/null`. Però s'ha d'anar amb compte i estar molt segur, ja que no tindrem cap sortida per pantalla.

Els *pipes* permeten utilitzar de manera simple tant la sortida d'una ordre com l'entrada d'una altra; per exemple, `ls -l | sed -i "s/[aeio]/u/g"`, en què s'executa l'ordre `ls`, i la seva sortida, en comptes d'imprimir-se a la pantalla, s'envia (per un tub o *pipe*) al programa `sed`, que imprimeix la seva sortida corresponent. Per exemple, per a buscar en el fitxer */etc/passwd* totes les línies que acabin amb *false* podríem fer `cat /etc/passwd | grep false$`, en què s'executa el `cat`, i la seva sortida es passa al `grep` (el símbol $ al final de la paraula indica al `grep` que és final de línia).

1.2. Aspectes generales

Si l'entrada no és un comentari, és a dir (deixant de banda els espais en blanc i tabulador), la cadena **no** comença per #, l'intèrpret llegeix i el divideix en paraules i operadors, que es converteixen en ordres, operadors i altres construccions. A partir d'aquest moment, es fan les expansions i substitucions, les redireccions, i finalment, l'execució de les ordres.

En el Bash es podran tenir funcions, que són una agrupació d'ordres que es poden invocar posteriorment. I quan s'invoca el nom de la funció d'intèrpret (s'usa com un nom d'ordre simple), la llista d'ordres relacionades amb el nom de la funció s'executarà tenint en compte que les funcions s'executen en el context de l'intèrpret en curs, és a dir, no es crea cap procés nou per a això.

Un **paràmetre** és una entitat que emmagatzema valors, que poden ser un nom, un nombre o un valor especial. Una **variable** és un paràmetre que emmagatzema un nom i té atributs (1 o més o cap) i es creen amb la sentència `declare` i es treuen amb `unset;` i si no els donem dóna valor, tenen assignada la cadena nul·la.

Finalment, hi ha un conjunt d'expansions que fa l'intèrpret que es du a terme en cada línia i que poden ser enumerades com: d'accent/cometes, paràmetres i variables, substitució d'ordres, d'aritmètica, de separació de paraules i de noms d'arxius.

2. Elements bàsics d'un *shell script*

2.1. Què és un *shell script*?

Un *shell script* és simplement un arxiu (mysys.sh per a nosaltres) que té el contingut següent (hem numerat les línies per a referir-nos-hi com el `cat -n` però no formen part de l'arxiu):

```
RS@debian:~$ cat -n mysys.sh

1 #!/bin/bash
2 clear; echo "Informació donada pel shell script mysys.sh."
3 echo "Hola, $USER"
4 echo "La data és `date`, i aquesta setmana `date +"%V"`."
5 echo -n "Usuaris connectats:"
6 w | cut -d " " -f 1 - | grep -v USER | sort -u
7 echo "El sistema és `uname -s` i el processador és `uname -m`."
8 echo "El sistema està engegat des de fa:"
9 uptime
10 echo
11 echo "Això és tot!"
```

Per a executar-lo podem fer `bash mysys.sh` o bé canviar-li els atributs per a fer-lo executable i executar-lo com una ordre (al final es mostra la sortida després de l'execució):

```
RS@debian:~$ chmod 744 mysys.sh

RS@debian:~$ ./mysys.sh

Informació donada pel shell script mysys.sh.

Hola, RS

La data és Sun Jun 8 10:47:33 EDT 2014, i aquesta és la setmana 23.

Usuaris connectats: RS

El sistema és Linux i el processador és x86_64.

El sistema està engegat des de fa:
10:53:07 up 1 day, 11:35, 1 user, load average: 0.12, 0.125, 0.33
```

```
Això és tot!
```

L'script comença amb "#!", que és una línia especial per a indicar amb quin intèrpret s'ha de llegir aquest script. La línia 2 és un exemple de dues ordres (una esborra la pantalla i l'altra imprimeix el que hi ha a la dreta) en la mateixa línia, i per això han d'estar separades per ";". El missatge s'imprimeix amb la sentència `echo` (ordre interna) però també es podria haver utilitzat la sentència `printf` (també ordre interna) per a una sortida amb format. Quant als aspectes més interessants de la resta, la línia 3 mostra el valor d'una variable, la 4 forma una cadena de caràcters (*string*), la 5 forma una cadena amb la sortida d'una ordre (reemplaçament de valors), la 6 executa la seqüència de 4 ordres amb paràmetres encadenats per *pipes* "|" i la 7 (similar a la 4) també reemplaça valors d'ordres per a formar una cadena de caràcters abans d'imprimir-se per pantalla.

#!/bin/bash

Així evitem que si l'usuari té un altre intèrpret, l'execució doni errors de sintaxi, és a dir, garantim que aquest script sempre s'executarà amb *bash*.

Per a depurar un *shell script* podem executar l'script amb `bash -x mysys.sh` o incloure en la primera línia *-x:* `#!/bin/bash -x`.

També es pot depurar una secció de codi incloent-hi:

```
set -x # activa depuració des d'aquí
codi per depurar
set +x # per a la depuració
```

2.2. Variables i matrius (*arrays*)

Es poden usar variables però no hi ha tipus de dades. Una variable de Bash pot contenir un nombre, un caràcter o una cadena de caràcters; no es necessita declarar una variable, ja que es crearà només assignar-li un valor. També es pot declarar amb `declare` i després assignar-li un valor:

```
#!/bin/bash
a="Hola UOC"
echo $a
declare b
echo $b
b="Com esteu?"
echo $B
```

Com es pot veure, es crea una variable *a* i se li assigna un valor (per a delimitar-lo s'ha de posar entre " ") i es recupera el VALOR d'aquesta variable posant-li un $ al principi, i si no se li posa $, només imprimirà el nom de la variable, no el valor. Per exemple, per a fer un script que faci una còpia (*backup*) d'un directori, incloent-hi la data i hora, en el moment de fer-ho, en el nom de l'arxiu (intenteu fer això tan fàcil amb ordres en un sistema W i acabareu frustrats):

```
#!/bin/bash
OF=/home/$USER-$(date +%d%m%Y).tgz
```

```
tar -cZf $OF /home/$USER
```

Aquest script introdueix una cosa nova, ja que estem creant una variable i assignem un valor (resultat de l'execució d'una ordre en el moment de l'execució). Fixeu-vos en l'expressió *$(date +%d%m%Y)*, que es posa entre () per a capturar-ne el valor. USER és una variable d'entorn i només es reemplaçarà pel seu valor. Si volguéssim agregar (concatenar) a la variable USER, per exemple, una cadena més, hauríem de delimitar la variable amb {}. Per exemple:

```
RS@debian:~$OF=/home/${USER}uppi-$(date +%d%m%Y).tgz
RS@debian:~$ echo $OF
/home/RSuppi-17042010.tgz
```

Així, el nom del fitxer serà diferent cada dia. És interessant veure el reemplaçament d'ordres que fa el Bash amb els parèntesis; per exemple, `echo $(ls)`.

Les variable locals es poden declarar anteposant *local*, i això en delimita l'àmbit.

```
#!/bin/bash
HOLA=Hola
function uni {
local HOLA=UOC
echo -n $HOLA
}
echo -n $HOLA
uni
echo $HOLA
```

La sortida serà *HolaUOCHola*, en què hem definit una funció amb una variable local que recupera el seu valor quan s'acaba d'executar la funció.

Les variables les podem declarar com a **ARRAY[INDEXNR]=valor**, en què *INDEXNR* és un nombre positiu (començant des de 0) i també podem declarar una matriu com `declare -a ARRAYNAME`, i es poden fer assignacions múltiples amb: **ARRAY=(value1 value2... valueN)**

```
RS@debian:~$ array=( 1 2 3)
RS@debian:~$ echo $array
1
RS@debian:~$ echo ${array[*]}
1 2 3
RS@debian:~$ echo ${array[2]}
3
RS@debian:~$ array[3]=4
RS@debian:~$ echo ${array[*]}
1 2 3 4
```

```
RS@debian:~$ echo ${array[3]}
4
RS@debian:~$ unset array[1]
RS@debian:~$ echo ${array[*]}
1 3 4
```

2.3. Estructures condicionals

Les estructures condicionals ens permeten decidir si es fa una acció o no; aquesta decisió es pren avaluant una expressió.

- La més bàsica és: **if expressió; then sentència; fi**, en què *sentència* només s'executa si *expressió* s'avalua com a verdadera. 2 < 1 és una expressió que s'avalua falsa, mentre que 2 > 1 s'avalua verdadera.

- Els condicionals tenen altres formes, com ara: **if expressió; then sentència1; else sentència fi**. Aquí *sentència1* s'executa si *expressió* és verdadera. D'altra manera, s'executa *sentència2*.

- Una altra forma més de condicional és: **if expressió1; then sentència1; elif expressió2; then sentència2; else sentència3**. En aquesta forma només s'afegeix *else if expressió2 then sentència2*, que fa que *sentència2* s'executi si *expressió2* s'avalua verdadera. La sintaxi general és:

```
if [expressió];
   then
   codi si 'expressió' és verdadera.
   fi
```

Un exemple en el qual el codi que s'executarà si l'expressió entre claudàtors és verdadera es troba entre la paraula *then* i la paraula *fi*, que indica el final del codi executat condicionalment:

```
#!/bin/bash
if [ "pirulo" = "pirulo" ]; then
echo expressió avaluada com a verdadera
fi
```

Un altre exemple amb variables i comparació per igualtat i per diferència (= o !=):

```
#!/bin/bash
T1="pirulo"
T2="pirulon"
if [ "$T1" = "$T2" ]; then
echo expressió avaluada com a verdadera
else
```

```
echo expressió avaluada com a falsa
fi
if [ "$T1" != "$T2" ]; then
echo expressió avaluada com a verdadera
else
echo expressió avaluada com a falsa
fi
```

Un exemple d'expressió de comprovació si hi ha un fitxer (cal vigilar amb els espais després de [i abans de]):

```
FILE=~/.basrc
if [ -f $FILE ]; then
echo el fitxer $FILE existeix
else
echo fitxer no trobat
fi
if [ 'test -f $FILE' ]
echo Una altra manera d'expressió per a saber si existeix o no
fi
```

Hi ha versions curtes d'**if**, com per exemple:

```
[ -z "${COLUMNS:-}" ] && COLUMNS=80
```

És la versió curta de:

```
if [ -z "${COLUMNS:-}" ]; then
COLUMNS=80
fi
```

Un altre exemple d'ús de la sentència **if** i variables:

```
#!/bin/bash
FILE=/var/log/syslog
MYMAIL="adminp@master.hpc.local"
SUBJECT="Error - $(hostname)"
BODY="Existeixen ERRORS en $(hostname) @ $(date). Veure syslog."
OK="OK: No es detecten errors en Syslog."
WARN="Possibles Errors. Analitzar"
# Verificar que $FILE existeix
if test ! -f "$FILE"
then
   echo "Error: $FILE no existeix. Analitzar quin és el problema."
   exit 1
fi
# Busquem errors
```

```
errors=$(grep -c -i "rror" $FILE)
# Si?
if [ $errors -gt 0 ]
then     # Si ...
            echo "$BODY" | mail -s "$SUBJECT" $MYMAIL
            echo "Mail enviat ..."
else      # No
            echo "$OK"
fi
```

La sentència `if then else` es pot resumir com:

```
if list; then list; [ elif list; then list; ] ... [ else
list; ] fi
```

S'ha de parar esment als ";" i a les paraules clau, i no és necessari posar tota la sentència en una sola línia.

2.4. Els bucles

El bucle **for** té dues formes, una d'elles és diferent a la d'altres llenguatges de programació, ja que permet iterar sobre una sèrie de "paraules" contingudes dins d'una cadena. La segona forma de **for** segueix els patrons dels llenguatges de programació habituals. El bucle **while** executa una secció de codi si l'expressió de control és verdadera, i només s'atura quan és falsa (o es troba una interrupció explícita dins del codi en execució; per exemple, un **break**). El bucle **until** és gairebé idèntic al bucle **while**, tret que el codi s'executa mentre l'expressió de control s'avalua com a falsa. Vegem-ne uns exemples i observem-ne la sintaxi:

```
#!/bin/bash
for i in $( ls ); do
echo element: $i
done
```

En la segona línia declarem *i* com la variable que rebrà els diferents valors continguts a `$( ls )`, que seran els elements del directori obtingut en el moment de l'execució. El bucle es repetirà (entre *do* i *done*) tantes vegades com elements tingui la variable i en cada iteració la variable *i* adquirirà un valor diferent de la llista.

Una altra sintaxi acceptada podrà ser:

```
#!/bin/bash
for i in `seq 1 10`;
do
```

```
echo $i
done
```

En la segona forma del `for`, la sintaxi és `for (( expr1 ; expr2 ; expr3 )) ; do list ;` on primer s'avalua l'expressió `expr1;` després s'avalua l'expressió repetidament fins que és 0. Cada vegada que la `expr2` és diferent de 0, s'executa `list` i l'expressió aritmètica `expr3` és avaluada. Si alguna de les expressions s'omet, aquesta sempre s'avaluarà com a 1. Per exemple:

```
#!/bin/bash
for (( c=1; c<=5; c++ ))
do
   echo "Repetició Núm. $c "
done
```

Un bucle infinit podria ser:

```
#!/bin/bash
for (( ; ; ))
do
   echo "Bucle infinit [ introduir CTRL+C per a finalitzar]"
done
```

La sintaxi de **while** és:

```
#!/bin/bash
comptador=0
while [ $comptador -lt 10 ]; do
echo El comptador és $comptador
let comptador=comptador+1
done
```

Com podem veure, a més de **while** hem introduït una comparació en l'expressió *per menor* "-lt" i una operació numèrica amb *let comptador=comptador +1*. La sintaxi d'**until** és equivalent:

```
#!/bin/bash
comptador=20
until [ $comptador -lt 10 ]; do
echo Comptador-Until $comptador
let comptador-=1
done
```

En aquest cas veiem l'equivalència de la sentència i a més una altra manera de fer operacions sobre les mateixes variables amb "-=".

Les sentències repetitives es poden resumir com:

```
for name [ [ in [ word ... ] ] ; ] do list ; done
for (( expr1 ; expr2 ; expr3 )) ; do list ;
while list-1; do list-2; done
until list-1; do list-2; done
```

Com en la sentència de `if` s'han de tenir en compte els ";" i les paraules clau de cada instrucció.

2.5. Funcions, *select*, *case*, arguments i altres qüestions

Les funcions són la manera més fàcil d'agrupar seccions de codi, que es podran tornar a utilitzar. La sintaxi és *function nom { codi }*, i per a cridar-les només és necessari que siguin dins del mateix arxiu i escriure el seu nom.

```
#!/bin/bash
function sortir {
exit
}
function hola {
echo Hola UOC!
}
hola
sortir
echo Mai no arribareu a aquesta línia
```

Com es pot veure tenim dues funcions, *sortir* i *hola*, que després s'executaran (no és necessari declarar-les en un ordre específic) i com la funció *sortir* executa un **exit** en l'intèrpret mai no arribarà a executar l'*echo* final. També podem passar arguments a les funcions, que seran llegits per ordre (de la mateixa manera com es llegeixen els arguments d'un script): *$1* el primer, *$2* el segon i així successivament:

```
#!/bin/bash
function sortir {
exit
}
function hola-amb-parametres {
echo $1
}
hola-amb-parametres Hola
hola-amb-parametres UOC
sortir
echo Mai no arribareu a aquesta línia
```

El `select` és per fer a opcions i llegir des de teclat:

```
#!/bin/bash
OPCIONS="Sortir Opció1"
select opt in $OPCIONS; do
if [ "$opt" = "Sortir" ]; then
echo Sortir. Adéu.
exit
elif [ "$opt" = "Opció1" ]; then
echo Hola UOC
else
clear
echo opció no permesa
fi
done
```

El `select` permet fer menús en mode text i és equivalent a la sentència `for`, només que itera sobre el valor de la variable indicada en el `select`. Un altre aspecte important és la lectura d'arguments, com per exemple:

```
#!/bin/bash
if [ -z "$1" ]; then
echo ús: $0 directori
exit
fi
A=$1
B="/home/$USER"
C=backup-home-$(date +%d%m%Y).tgz
tar -cZf $A$B $C
```

Si el nombre d'arguments és 0, escriu el nom de l'ordre ($0) amb un missatge d'ús. La resta és similar al que hem vist anteriorment excepte per al primer argument ($1). Fixeu-vos amb atenció en l'ús de " i la substitució de variables.

El `case` és una altra forma de selecció; mirem aquest exemple que ens alerta en funció de l'espai de disc que tenim disponible:

```
#!/bin/bash
space=`df -h | awk '{print $5}' | grep % \
| grep -v Use | sort -n | tail -1 | cut -d "%" -f1 -`

   case $space in
   [1-6]*)
   Message="tot bé."
   ;;
   [7-8]*)
```

```
   Message="Podríeu començar a esborrar alguna cosa en $space %"
   ;;
   9[1-8])
   Message="Uff. Millor un disc nou. Partició $space % fins a dalt."
   ;;
   99)
   Message="Pànic! No teniu espai en $space %!"
   ;;
   *)
   Message="No teniu res..."
   ;;
   esac
   echo $Message
```

A més de parar atenció a la sintaxi del **case**, podem mirar com s'escriu una ordre en dues línies amb "\", el filtratge de seqüència amb diversos `grep` i | i una ordre interessant (filtre) com `awk` (la millor ordre de tots els temps).

Com a resum de les sentències abans vistes tenim:

```
select name [ in word ] ; do list ; done
case word in [ [(] pattern [ | pattern ] ... ) list ;; ] ... esac
```

Un exemple de la sentència `break` podria ser el següent, que busca un arxiu en un directori fins que el troba:

```
#!/bin/bash
for file in /etc/*
do
if [ "${file}" == "/etc/passwd" ]
then
   nroentradas=`wc -l /etc/passwd | cut -d" " -f 1`
   echo "Existeixen ${nroentrades} entrades definides en ${file}"
   break
fi
done
```

L'exemple següent mostra com s'utilitza el `continue` per a continuar amb el bucle següent del llaç:

```
#!/bin/bash
for f in `ls`
do
   # if l'arxiu .org existeix llegeixo la següent
   f=`echo $f | cut -d"." -f 1`
   if [ -f ${f}.org ]
```

```
then
   echo "Següent $f arxiu..."
   continue  # llegeixo la següent i salto l'ordre cp
fi
   # no tenim l'arxiu .org llavors el copio
   cp $f $f.org
done
```

Moltes vegades, podeu voler sol·licitar a l'usuari alguna informació, i hi ha diverses maneres per a fer-ho:

```
#!/bin/bash
echo Si us plau, escriviu el nom
read nom
echo "Hola, $nombre!"
```

Com a variant, es poden obtenir múltiples valors amb *read:*

```
#!/bin/bash
echo Si us plau, escriviu el nom i el primer cognom
read NO AP
echo "Hola $AP, $NO!"
```

Si fem `echo 10 +10` i esperàveu veure un 20 quedareu desil·lusionats; la forma d'avaluació directa és amb `echo $((10+10))` o també `echo $[10+10]` i si necessiteu operacions com fraccions o d'altres podeu utilitzar `bc`, ja que l'anterior només serveix per a nombres enters. Per exemple, `echo $[3/4]` donarà 0, però `echo 3/4|bc -l` sí que funcionarà. També recordem l'ús del `let;` per exemple, `let a=75*2` assignarà 150 a la variable *a*.

Un exemple més:

```
RS@debian:~$ echo $date
20042011
RS@debian:~$ echo $((date++))
20042011
RS@debian:~$ echo $date
20042012
```

És a dir, com a operadors podem utilitzar:

- VAR++ VAR-- variable postincrement i postdecrement.
- ++VAR --VAR com l'anterior però abans.
- + - operadors unaris.
- ! ~ negació lògica i negació en *string*.
- ** exponent.
- / + - % operacions.

- < < > > desplaçament a esquerra i dreta.
- < = >= < > comparació.
- == != igualtat i diferència.
- & ^ | operacions AND, OR exclusiu i OR.
- && || operacions AND i OR lògics.
- *expr* ? *expr* : *expr* avaluació condicional.
- = *= /= %= += -= < < = >>= &= ^= |= operacions implícites.
- , separador entre expressions.

Per a capturar la sortida d'una ordre, és a dir, el resultat de l'execució, podem utilitzar el nom de l'ordre entre accents oberts ` `.

```
#!/bin/bash
A=`ls`
for b in $A ;
do
file $b
done
```

Per a la comparació tenim les opcions següents:

- s1 = s2 verdader si s1 coincideix amb s2.
- s1 != s2 verdader si s1 no coincideix amb s2.
- s1 < s2 verdader si s1 és alfabèticament anterior a s2.
- s1 > s2 verdader si s1 és alfabèticament posterior a s2.
- -n s1 s1 no és nul (conté un o més caràcters).
- -z s1 s1 és nul.

Per exemple, cal anar amb compte perquè si S1 o S2 són buits, ens donarà un error d'interpretació (*parse*):

```
#!/bin/bash
S1='cadena'
S2='Cadena'
if [ $S1!=$S2 ];
then
echo "S1('$S1') no és igual que S2('$S2')"
fi
if [ $S1=$S1 ];
then
echo "S1('$S1') és igual que S1('$S1')"
fi
```

Quant als operadors aritmètics i relacionals, podem utilitzar:

a) Operadors aritmètics:

- + (addició).
- - (sostracció).
- * (producte).
- / (divisió).
- % (mòdul).

b) Operadors relacionals:

- -lt (<).
- -gt (>).
- -li (< =).
- -ge (>=).
- -eq (==).
- -ne (!=).

Fem un exemple de script que ens permetrà rebatejar fitxers S1 amb una sèrie de paràmetres (prefix o sufixos) d'acord amb els arguments. La manera serà *p* [*prefix*] *fitxers*, o si no, *s* [*sufix*] *fitxers*, o també *r* [*patró-antic*] [*patró-nou*] *fitxers* (i com sempre diem, els scripts són millorables):

```
#!/bin/bash -x
# renom: reanomena múltiples arxius d'acord amb certes regles
# Basat en un original de F. Hudson. Gener de 2000
# comprova si vull reanomenar amb prefix i deixo només els noms
# d'arxius
if [ $1 = p ]; then
      prefix=$2 ; shift ; shift

      # si no hi ha entrades d'arxius acabo.
      if [ "$1" = '' ]; then
         echo "no s'han especificat arxius"
         exit 0
      fi
      # Interacció per a reanomenar
      for arxiu in $*
         do
         mv ${arxiu} $prefix$arxiu
         done
      exit 0
fi
# comprova si vull reanomenar amb prefix i deixo només els noms
# d'arxius
if [ $1 = s ]; then
      sufix=$2 ; shift ; shift
      if [ "$1" = '' ]; then
         echo "no s'han especificat arxius"
         exit 0
```

```
      fi
      for arxiu in $*
         do
         mv ${arxiu} $arxiu$sufix
         done
      exit 0
fi
# comprova si és una substitució de patrons
if [ $1 = r ]; then
shift
# s'ha inclòs això com a mesura de seguretat
      if [ $# -lt 3 ] ; then
         echo "ús: renom r [expressió] [substitut] arxius... "
         exit 0
      fi
      VELL=$1 ; NOU=$2 ; shift ; shift
      for arxiu in $*
         do
         nou=`echo ${arxiu} | sed s/${VELL}/${NOU}/g`
         mv ${arxiu} $nou
         done
      exit 0
fi
# si no mostro l'ajuda
echo "ús:"
echo " renom p [prefix] arxius..."
echo " renom s [sufix] arxius..."
echo " renom r [patró-antic] [patró-nou] arxius..."
exit 0
```

Cal parar una atenció especial a "", '', `` {}, etc, per exemple:

```
RS@debian:~$ date=20042010
RS@debian:~$ echo $date
20042010
RS@debian:~$ echo \$date
$date
RS@debian:~$ echo '$date'
$date
RS@debian:~$ echo "$date"
20142005
RS@debian:~$ echo "`date`"
Mon Jun 9 00:33:42 CEST 2014
RS@debian:~$ echo "el que se m'ocorre és: \"Estic cansat\""
el que se m'ocorre és: "Estic cansat"
RS@debian:~$ echo "\"
>
```

```
(espera més entrades: feu Ctrl-C)
remo@debian:~$ echo "\\"
\
RS@debian:~$ echo grand{et,ot,às}
grandet grandot grandàs
```

Una combinació d'entrada de teclat i operacions/expressions per a calcular un any de traspàs:

```
#!/bin/bash
clear;
echo "Entreu l'any que cal verificar (4 dígits), i després [ENTER]:"
read year
if (( ("$year" % 400) == "0" )) || (( ("$year" % 4 == "0") \
&& ("$year" % 100 !="0") )); then
      echo "$year és de traspàs."
else
      echo "$year l'any no és de traspàs."
fi
```

Una ordre interessant que combina **read** amb delimitador en una expressió (està posat tot en una línia i per això la sintaxi està separada per ;). El resultat serà "interessant" (ens mostrarà totes les cadenes donades pel **find** en una línia i sense /:

```
find "$PWD" -name "m*" | while read -d "/" file; do echo $file; done
```

Un aspecte interessant per a treballar amb string (vegeu el manual de Bash: `man bash`) és ${*VAR*:OFFSET:*LENGTH*}.

```
RS@debian:~$ export str="unstring molt llarg"
RS@debian:~$ echo $str
unstring molt llarg
RS@debian:~$ echo ${str:4}
ring molt llarg
RS@debian:~$ echo $str
unstring molt llarg
RS@debian:~$ echo ${str:9:3}
mol
RS@debian:~$ echo ${str%llarg}
unstring molt
```

La instrucció `trap` ens permetrà capturar un senyal (per exemple, de teclat) dins d'un *script*.

```
#!/bin/bash
# Per a acabar feu des d'un altre terminal kill -9 pid
```

```
trap "echo ' Ja!'" SIGINT SIGTERM
echo "El pid és $$"

while : # això és el mateix que "while true".
      do
      sleep 10 # L'script no fa res.
done
```

2.6. Filtres: *grep*

El *grep* és un filtre de patrons molt útil i versàtil; a continuació en podem veure alguns exemples:

```
RS@debian:~$ grep root /etc/passwd    busca patró root
root:x:0:0:root:/root:/bin/bash

RS@debian:~$ grep -n root /etc/passwd    a més numera les línies
1:root:x:0:0:root:/root:/bin/bash

RS@debian:~$ grep -v bash /etc/passwd | grep -v nologin
     mostra els que no tenen el patró i bash ni el patró nologin
daemon:x:1:1:daemon:/usr/sbin: /bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
...

RS@debian:~$ grep -c false /etc/passwd    compta els que tenen false
7
RS@debian:~$ grep -i ps ~/.bash* | grep -v history
     busca el patró ps en tots els arxius que comencen per .bash en el directori /home
     excloent-hi els que tinguin history en l'arxiu
/home/RS/.bashrc:[ -z "$PS1" ] && return
/home/RS/.bashrc:export HISTCONTROL=$HISTCONTROL${HISTCONTROL+,}ignoredups
/home/RS/.bashrc:# ... or force ignoredups and ignorespace
...

RS@debian:~$ grep ^root /etc/passwd     busca l'inici de línia
root:x:0:0:root:/root:/bin/bash

RS@debian:~$ grep false$ /etc/passwd    busca el final de línia
Debian-exim:x:101:105::/var/spool/exim4:/bin/false
statd:x:102:65534::/var/lib/nfs:/bin/false

RS@debian:~$ grep -w / /etc/fstab    busca /
/dev/hda1 / ext3 errors=remount-ro 0 1
```

```
RS@debian:~$ grep [yf] /etc/group    busca y o Y
sys:x:3:
tty:x:5:
....

RS@debian:~$ grep '\<c...h\>' /usr/share/dict/words    busca començant per c i acabant per h amb
                                                                    un màxim de tres.
catch
cinch
cinch's
....
RS@debian:~$ grep '\<c.*h\>' /usr/share/dict/words     busca començant per c i acabant per h totes.
caddish
calabash
...
```

2.7. Filtres: Awk

Awk, o també la implementació de GNU *gawk*, és una ordre que accepta un llenguatge de programació dissenyat per a processar dades basades en text, ja siguin fitxers o fluxos de dades, i ha estat la inspiració de Larry Wall per a escriure Perl. La seva sintaxi més comuna és `awk 'programa' arxius...` i programa pot ser: **patró {acció} patró {acció}...** *awk* llegeix l'entrada d'arxius una línia alhora. Cada línia es compara amb cada patró en ordre; per a cada patró que concordi amb la línia s'efectua l'acció corresponent. Un exemple pregunta pel primer camp si és *root* de cada línia de l'arxiu */etc/passwd* i la imprimeix considerant com a separador de camps el ":" amb -F:; en el segon exemple reemplaça el primer camp per *root* i imprimeix el resultat canviat (a la primera ordre utilitzem '=='; a la segona, només un '=').

```
RS@debian:~$ awk -F: '$1=="root" {print}' /etc/passwd
root:x:0:0:root:/root:/bin/bash
RS@debian:~$ awk -F: '$1="root" {print}' /etc/passwd
root x 0 0 root /root /bin/bash
root x 1 1 daemon /usr/sbin /bin/sh
root x 2 2 bin /bin /bin/sh
root x 3 3 sys /dev /bin/sh
root x 4 65534 sync /bin /bin/sync
root x 5 60 games /usr/games /bin/sh
root x 6 12 man /var/cache/man /bin/sh
```

L'*awk* divideix automàticament l'entrada de línies en camps, és a dir, una cadena de caràcters que no siguin blancs separats per blancs o tabuladors; per exemple, el `who` té 5 camps i `awk` ens permetrà filtrar cada un d'aquests camps.

```
RS@debian:~$ who
```

```
RS tty7 2010-04-17 05:49 (:0)
RS pts/0 2010-04-17 05:50 (:0.0)
RS pts/1 2010-04-17 16:46 (:0.0)
remo@debian:~$ who | awk '{print $4}'
05:49
05:50
16:46
```

L'*awk* anomena aquests camps $1 $2... $NF, en què NF és una variable igual que el nombre de camps (en aquest cas NF = 5). Per exemple:

```
ls -al | awk '{print NR, $0}    Agrega nombre d'entrades (la variable NR compta el nombre de línies,
                                          $0 és la línia sencera.
ls -al | awk '{printf "%d %s\n", NR, $9}'   %d significa un nombre decimal (NR) i %s un
                                                   string ($9) i  un new line
awk -F: '$2 == "" ' /etc/passwd      donarà els usuaris que en l'arxiu passwd no tinguin posada
                                                   la contrasenya
```

El patró es pot escriure de diverses maneres:

```
$2 == ""      si el segon camp és buit
$2 ~ /^$/     si el segon camp coincideix amb la cadena buida
$2 !~ /./      si el segon camp no concorda amb cap caràcter (! és negació)
length($2) ==     si la longitud del segon camp és 0, length és una funció interna de l'awk
~                indica coincidència amb una expressió
!~               significa el contrari (sense coincidència)
NF % 2 != 0        mostra la ratlla només si hi ha un nombre parell de camps
awk 'lenght ($0) 32 {print "Línia", NR, "llarga", substr($0,1,30)} ' /etc/passwd       avalua las
                     línies de /etc/passwd i genera un substring</i>
```

Hi ha dos patrons especials, BEGIN i END. Les accions BEGIN es fan abans que la primera línia s'hagi llegit; es pot usar per a inicialitzar les variables, imprimir encapçalaments o posicionar separadors d'un camp assignant-los a la variable FS, per exemple:

```
awk 'BEGIN {FS = ":"} $2 == "" ' /etc/passwd         igual que l'exemple d'abans
awk 'END { print NR } ' /etc/passwd          imprimeix el nombre de línies processades al final de la
                                                          lectura de l'última ratlla.
```

L'`awk` permet operacions numèriques en forma de columnes, per exemple, per a sumar tots els nombres de la primera columna:

```
{ s=s + 1} END { print s}          i per a la suma i la mitjana END { print s, s/NR}
```

Les variables s'inicialitzen a zero en declarar-se, i es declaren en utilitzar-se, per la qual cosa resulta molt simple (els operadors són els mateixos que a C):

```
{ s+=1} END {print s}                Per exemple, per a comptar línies, paraules i caràcters:
{ nc += length($0) +1 nw +=NF } END {print NR, nw, nc}
```

Hi ha variables predefinides en l'*awk:*

- FILENAME nom de l'arxiu actual.
- FS caràcter delimitador del camp.
- NF nombre de camps del registre d'entrada.
- NR número del registre de l'entrada.
- OFMT format de sortida per a nombres (%g per defecte).
- OFS cadena separadora de camp a la sortida (blanc per defecte).
- ORS cadena separadora de registre de sortida (*new line* per defecte).
- RS cadena separadora de registre d'entrada (*new line* per defecte).

Operadors (ídem C):

= += -= *= /= %= || && ! >> >>= << <<= == != ~ !~
+ - * / %
++ --

Funcions predefinides a *awk:*

cos(), exp(), getline(), index(), int(), length(), log(), sin(), split(), sprintf(), substr().

A més suporta dins d'acció sentències de control amb la sintaxi similar a C:

```
if (condició)
proposició1
else
proposició2

for (exp1;condició;exp2)

while (condició) {
proposició
expr2
}
continue     continua, avalua la condició novament
break          trenca la condició
next            llegeix la línia d'entrada següent
exit             salta a END
```

També l'`awk` maneja arranjaments, per exemple, per a fer un `head` de */etc/passwd*:

```
awk '{ line[NR] = $0} \
```

```
END { for (i=NR; i>2; i--) print line[i]} ' /etc/passwd
```

Un altre exemple:

```
awk 'BEGIN { print "Usuari UID Shell\n------- --- -----" } $3 >= 500 { print $1, $3, $7 |
"sort -r"}' FS=":" /etc/passwd
Usuari UID Shell
------- --- -----
RS 1001 /bin/bash
nobody 65534 /bin/sh
debian 1000 /bin/bash

RS@debian:~$ ls -al | awk '
BEGIN { print "File\t\t\tOwner" }
{ print $8, "\t\t\t", \
$3}
END { print "done"}
'
File Owner
. RS
.. root
.bash_history RS
.bash_logout RS
.bashrc RS
.config RS
...
```

2.8. Exemples complementaris

1) Un exemple complet per a esborrar els registres (esborra */var/log/wtmp* i es queda amb 50 línies –o les que passi l'usuari en línia d'ordres– de */var/log/messages*)

```
#!/bin/bash
#Variables
LOG_DIR=/var/log
ROOT_UID=0      # només el pot executar el root
LINES=50            # Línies por defecte.
E_XCD=86           # No em puc canviar de directori
E_NOTROOT=87 # Sortida de no-root error.

if [ "$UID" -ne "$ROOT_UID" ]   # Sóc root?
      then
      echo "Heu de ser root. Em sap greu."
      exit $E_NOTROOT
      fi
```

```
if [ -n "$1" ]            # Nombre de línies per preservar?
      then
      lines=$1
      else
      lines=$LINES   # valor per defecte.
      fi

cd $LOG_DIR

if [ `pwd` != "$LOG_DIR" ]
# també pot ser if [ "$PWD" != "$LOG_DIR" ]
      then
      echo "No puc anar a $LOG_DIR."
      exit $E_XCD
      fi #

# Una altra manera de fer-ho seria:
# cd /var/log || {
# echo "No puc !" >&2
# exit $E_XCD;
# }

tail -n $lines messages > mesg.temp     # Deso temporalment
mv mesg.temp messages                      # Moc.

cat /dev/null > wtmp                              #Esborro wtmp.
echo "Registres esborrats."
exit 0
# Un zero indica que tot ha anat bé.
```

2) Utilització de l'*expr*.

```
#!/bin/bash
echo "Aritmètics"
echo
a=`expr 5 + 3`
echo "5 + 3 = $a"
a=`expr $a + 1`
echo
echo "a + 1 = $a"
echo "(increment)"
a=`expr 5 % 3`
# mòdul
echo
echo "5 mod 3 = $a"
# Lògics
# 1 si true, 0 si false,
```

```
#+ oposat a normal Bash convention.

echo "Lògics"
x=24
y=25
b=`expr $x = $y` # per igual.
echo "b = $b" # 0 ( $x -ne $y )
a=3
b=`expr $a \> 10`
echo 'b=`expr $a \> 10`, per la qual cosa...'
echo "If a > 10, b = 0 (false)"
echo "b = $b" # 0 ( 3 ! -gt 10 )
b=`expr $a \< 10`
echo "If a < 10, b = 1 (true)"
echo "b = $b" # 1 ( 3 -lt 10 )
echo
# Compte amb els operadors d'escapada \.
b=`expr $a \<= 3`
echo "If a <= 3, b = 1 (true)"
echo "b = $b" # 1 ( 3 -le 3 )
# S'utilitza un operador "\>=" operator (greater than or equal to).

echo "String"
echo
a=1234zipper43231
echo "String base \"$a\"."
b=`expr length $a`
echo "Long. de \"$a\" es $b."
# index: posició del primer caràcter que satisfà en la cadena
#
b=`expr index $a 23`
echo "Posició numèrica del \"2\" en \"$a\" és \"$b\"."
# substr: extract substring, starting position & length specified
b=`expr substr $a 2 6`
echo "Substring de \"$a\", començant a 2,\
i de 6 chars és \"$b\"."
# Using Regular Expressions ...
b=`expr match "$a" '[0-9]*'` # comptador numèric.
echo Nombre de dígits de \"$a\" és $b.
b=`expr match "$a" '\([0-9]*\)'` # compte amb els caràcters d'escapada
echo "Els dígits de \"$a\" són \"$b\"."
exit 0
```

3) Un exemple que mostra diferents variables (amb lectura des del teclat), sentències i execució d'ordres:

```
#!/bin/bash
```

```
echo -n "Introduir el % de CPU del procés que més consumeix a supervisar, l'interval[s],
         Nre. repeticions [0=sempre]: "
read lim t in
while true
do
           A=`ps -e -o pcpu,pid,comm | sort -k1 -n -r | head -1`
           load=`echo $A | awk '{ print $1 }'`
           load=${load%.*}
           pid=`echo $A | awk '{print $2 }'`

           if [ $load -gt $lim ]
           then
              # verifico cada t segons la CPU load
              sleep $t
              B=`ps -e -o pcpu,pid,comm | sort -k1 -n -r | head -1`
              load2=`echo $B | awk '{ print $1 }'`
              load2=${load2%.*}
              pid2=`echo $B | awk '{print $2 }'`
              name2=`echo $B | awk '{print $3 }'`
              [ $load2 -gt $lim ] && [ $pid = $pid2 ] && echo $load2, $pid2, $name2
           let in--
              if [ $in == 0 ]; then  echo "Fi"; exit 0; fi
fi
done
```

4) Moltes vegades en els *shell scripts* és necessari fer un menú perquè l'usuari esculli una opció. En el primer codi s'ha d'instal·lar el paquet `dialog (apt-get install dialog)`.

a. Menú amb l'ordre `dialog`:

```
#!/bin/bash
# original http://serverfault.com/questions/144939/multi-select-menu-in-bash-script

cmd=(dialog --separate-output --checklist "Select options:" 22 76 16)
options=(1 "Opció 1" off    # Si se'n desitja qualsevol
                            # es pot posar en on
         2 "Opció 2" off
         3 "Opció 3" off
         4 "Opció 4" off)
choices=$("${cmd[@]}" "${options[@]}" 2>&1 >/dev/tty)
clear
for choice in $choices
do
    case $choice in
        1)
            echo "Primera Opció"
```

```
            ;;
        2)
            echo "Segona Opció"
            ;;
        3)
            echo "Tercera Opció"
            ;;
        4)
            echo "Quarta Opció"
            ;;
    esac
done
```

b. Menú només amb sentències:

```
#!/bin/bash
# Basat en original de D. Williamson
# http://serverfault.com/questions/144939/multi-select-menu-in-bash-script

toogle () {
    local choice=$1
    if [[ ${opts[choice]} ]]
    then
        opts[choice]=
    else
        opts[choice]="<-"
    fi
}

PS3='Seleccionar opció: '
while :
do
    clear
    options=(Opció A ${opts[1]}" "Opció B ${opts[2]}" "Opció C ${opts[3]}" "Sortir")
    select opt in "${options[@]}"
    do
        case $opt in
            "Opció A ${opts[1]}")
                toogle 1
                break
                ;;
           "Opció B ${opts[2]}")
                toogle 2
                break
                ;;
           "Opció C ${opts[3]}")
                toogle 3
```

```
                break
                ;;
            "Sortir")
                break 2
                ;;
            *) printf '%s\n' 'Opció no vàlida';;
        esac
    done
done

printf '%s\n' 'Opcions seleccionades:'
for opt in "${!opts[@]}"
do
    if [[ ${opts[opt]} ]]
    then
        printf '%s\n' "Opció $opt"
    fi
done
```

5) De vegades és necessari filtrar una sèrie d'arguments passats en la línia d'ordres i `getop` o `getopts` poden ajudar. Les ordres `getopt` i `getopts` s'utilitzen per a processar i validar arguments d'un *shell script* i són similars però no idèntics. A més, la funcionalitat pot variar en funció de la seva implementació per la qual cosa és necessari llegir les pàgines del manual amb atenció. En el primer exemple utilitzarem `getopts` i filtrarà `nombre_cmd -a valor`:

```
#!/bin/bash
# Més opcions a
while getopts ":a:" opt; do
   case $opt in
   a)
      echo "-a va ser introduït, Paràmetre: $OPTARG" >&2
      ;;
   \?)
      echo "opció invàlida: -$OPTARG" >&2
      exit 1
      ;;
   :)
      echo "Opció -$OPTARG requereix un argument." >&2
      exit 1
      ;;
   esac
done
```

I amb `getopt`:

```
#!/bin/bash
args=`getopt abc: $*`
if test $? != 0
   then
      echo 'Ús: -a -b -c valor'
      exit 1
fi
set -- $args
for i
do
   case "$i" in
        -c) shift;echo "Opció c com a argument amb $1";shift;;
         -a) shift;echo "Opció a com a argument";;
         -b) shift;echo "Opció b com a argument";;
   esac
done
```

Activitats

1. Creeu un script interactiu que calculi el nombre de dies entre dues dates introduïdes per l'usuari segons el format dia/mes/any. També haureu de calcular el nombre d'hores si l'usuari introdueix hora d'inici i hora de final, en el format hh:mm. Si introdueix només una hora, es calcularà fins a les 23:59 del dia donat com a final del període per calcular.

2. Creeu un script que es faci una còpia (recursiva) dels arxius a /etc, de manera que un administrador del sistema pugui modificar els arxius d'inici sense temor.

3. Escriviu un script anomenat *homebackup* que automatitzi el **tar** amb les opcions correctes i en el directori /var/backups per a fer una còpia de seguretat del directori principal de l'usuari amb les condicions següents:

a) Fer un test del nombre d'arguments. L'script s'ha d'executar sense arguments, i si n'hi ha, ha d'imprimir un missatge d'ús.
b) Determinar si el directori de còpies de seguretat té suficient espai lliure per a emmagatzemar la còpia de seguretat.
c) Preguntar a l'usuari si vol una còpia completa (tot) o una còpia incremental (només els arxius que hagin canviat). Si l'usuari no té una còpia de seguretat completa, s'haurà de fer, i en cas d'una còpia de seguretat incremental, només es pot fer si la còpia de seguretat completa no té més d'una setmana.
d) Comprimir la còpia de seguretat utilitzant qualsevol eina de compressió.
e) Informar a l'usuari que es farà en cada moment, ja que això pot trigar algun temps i així s'evitarà que l'usuari es posi nerviós si la sortida no apareix a la pantalla.
f) Imprimir un missatge que informi l'usuari sobre la mida de la còpia de seguretat sense comprimir i comprimida, el nombre d'arxius desats o actualitzats i el nombre de directoris desats o actualitzats.

4. Escriviu un script que executi un navegador web simple (en mode text), utilitzant **wget** i **links -dump** per a mostrar les pàgines HTML en un terminal.

L'usuari té 3 opcions: introduir una adreça URL, entrar **b** per a retrocedir (*back*) i **q** per a sortir. Les 10 últimes URL introduïdes per l'usuari s'emmagatzemen en una matriu, des d'on l'usuari pot restaurar l'URL utilitzant la funcionalitat **b** (*back*).

Bibliografia

És recomanable complementar la lectura en què s'ha basat part d'aquesta documentació, sobretot per a aspectes avançats, ja que aquí només s'ha vist un breu resum de l'essencial:

Barnett, Bruce. "AWK" http://www.grymoire.com/Unix/Awk.html. © General Electric Company.

"Bash Reference Manual" http://www.gnu.org/software/bash/manual/bashref.html.

Cooper, Mendel. "Advanced Bash-Scripting Guide" http://tldp.org/LDP/abs/html/index.html.

Garrels, Machtelt. "Bash Guide for Beginners" http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html.

Mike, G. "Programación en BASH - COMO de introducción" http://es.tldp.org/COMO-INSFLUG/COMOs/Bash-Prog-Intro-COMO/.

Robbins, Arnold. "UNIX in a Nutshell" (3.ª ed.) http://oreilly.com/catalog/unixnut3/chapter/ch11.html.

"The GNU awk programming language" http://tldp.org/LDP/Bash-Beginners-Guide/html/chap_06.html.

"The Shell Scripting Tutorial" http://bash.cyberciti.biz/guide/Main_Page.

van Vugt, Sander. "Beginning Ubuntu Server Administration: From Novice to Professional" (Disponible a Safari Books).