MP06 – UF 2 | AUTOMATITZACIÓ DE TASQUES I LLENGUATGES DE GUIONS

RA 1 | Gestiona l'automatització de tasques del sistema, aplicant-hi criteris d'eficiència i utilitzant comandaments i eines gràfiques

RA 2 | Utilitza llenguatge de guions en sistemes operatius, descrivint-ne l'aplicació i administrant-hi serveis del sistema operatiu

1. Bash scripting
   - ACT 1 | Execució de scripts
   - ACT 2 | Variables locals i d'entorn
   - ACT 3 | Paràmetres especials i operacions aritmètiques
   - ACT 4 | Funcionament i opcions de la comanda grep
   - ACT 5 | Classes de caràcters
   - ACT 6 | Treballant amb Stream EDitor
   - ACT 7 | Scripts AWK
   - ACT 8 | Arguments d'entrada: posició i número
   - ACT 9 | Declaracions condicionals
   - ACT 10 | Scripts interactius
   - ACT 11 | Treballant amb bucles
   - ACT 12 | Variables del tipus matriu
   - ACT 13 | Cerca i substitució de variables
   - ACT 14 | Crides a funcions i scripts

PT1 | Una paperera de reciclatge en Bash

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

1. BASH SCRIPTING

Els **Shell** és un programa que interpreta les comandes dels usuaris, que les introdueix directament, o que son llegides des d'un fitxer anomenat **shell script** (o programa d'intèrpret de comandes). Els shell scripts són interpretats, no compilats: el Shell llegeix els scripts línia per línia i busca les comandes allà especificades al sistema, metre que els compiladors converteixen el llenguatge de programació en un codi màquina llegible pel sistema.

A més d'aquesta tasca de passar comandes al nucli, els intèrprets de comandes tenen la funció de proporcionar un entorn de treball pels usuaris. Aquest entorn és configurable a través de fitxers de configuració del propi shell.

1.1. ELS INTÈRPRETS DE COMANDES LINUX

El teu sistema pot interpretar diferents tipus d'**intèrprets de comandes** (shells). Als sistemes Linux (basats en UNIX) podem trobar les següents varietats de shells[1]:

- **sh** o Bourne Shell: és el shell original utilitzat pels sistemes UNIX, i tot i que es tracta de una carcassa bàsica amb poques funcions, és considera el shell estàndard. Linux no l'utilitza per defecte però encara el conserva per mantenir les compatibilitats.
- **bash** o Bourne Again Shell: és el shell estàndard dels sistemes Linux, intuïtiva i flexible. És la més recomanable per a usuaris inicials (nosaltres treballarem en aquest entorn) però també ofereix molta potencialitat als usuaris més experts. Com el seu nom indica, guarda relació amb Bourne Shell, fent que totes les comandes de sh hi estiguin disponibles a bash (tot i que no al contrari). El podríem considerar una gran ampliació de sh mitjançant l'ús de molts complements.
- **csh** o C Shell: és un intèrpret de comandes amb un llenguatge semblant al llenguatge de programació C, cosa que el fa popular entre els programadors.
- **tcsh** o TENEX C Shell: és un "superconjunt" del C Shell comú que millora la facilitat d'ús i la velocitat (a vegades també l'anomenen Turbo C).
- **ksh** o Korn Shell: és un altre "superconjunt" en aquest cas sobre la carcassa bàsica del Bourne Shell, tot i que és força enrevessat i poc útils per a usuaris poc experts.

Com ja sabeu, podreu saber quin shell utilitza per defecte cadascun dels usuaris del vostre sistema examinant el fitxer `/etc/passwd` (últim camp de cadascuna de les línies de cada usuari). Per canviar el nostre shell de treball en un terminal només cal introduir com a comanda el nom del shell al que volem canviar. Això executarà el programa que defineix les funcions del nou intèrpret de comandes concret i presentarà, possiblement, una presentació del prompt o de l'entorn de treball diferent.

[1] Pots conèixer quins intèrprets de comandes (shells) tens disponibles al teu sistema examinant els fitxer `/etc/shells`.

1.2. AVANTATGES DEL BOURNE AGAIN SHELL

BASH (Bourne Again Shell) és un shell compatible amb sh que incorpora funcions útils del Korn Shell (ksh) i del C Shell (csh). S'ajusta a la norma *IEEE POSIX P1003.2 / ISO 9945.2 Shell and Tools estandard*.

Ofereix millores funcionals sobre sh tant per a la programació com per a l'ús interactiu: inclouen l'edició de línia de comandaments, historial de comandes de mida il·limitada, control de feina, funcions de shell i àlies, matrius indexades de mida il·limitada i aritmètica de sencer en qualsevol base de dos a seixanta-quatre. Bash pot executar la majoria dels scripts sh sense modificació.

Igual que els altres projectes de GNU, la iniciativa bash es va iniciar per preservar, protegir i promoure la llibertat d'ús, estudi, còpia, modificació i redistribució de programari. En general, es coneix que aquestes condicions estimulen la creativitat. Aquest també va ser el cas del programa bash, que té moltes funcions addicionals que altres oferir. Algunes de les seves característiques són:

- **Shells interactius:** un shell interactiu generalment llegeix i escriu del/al terminal d'un usuari: l'entrada i la sortida estan connectades a un terminal. El comportament interactiu de Bash s'inicia quan es crida l'ordre bash sense arguments.
- **Fitxers d'arrancada de bash:** els fitxers d'inici són les seqüències d'ordres que llegeix i executa bash al començar el seu mode **interactiu**: `/etc/profile`, `~/.bash_profile` i `~/.bashrc`, per als inicis de sessió, o només `~/.bashrc` en inicis del programa sense iniciar sessió (per exemple, quan obriu un terminal usant una icona, aquest no és un inici de sessió). Bash també es pot invocar de forma **no interactiva**: tots els **scripts** utilitzen shells no interactius. Estan programats per realitzar determinades tasques i no es pot instruir a fer altres treballs diferents als que estan programats. Els fitxers llegits estan definits a la variable `BASH_ENV`. La variable `PATH` no s'utilitza per cercar aquest fitxer (per defecte), i per tant, si el voleu utilitzar caldrà que ús referiu a ell donant la ruta completa i el nom del fitxer. També és pot invocar a través de la **comanda sh:** bash intenta comportar-se com el programa històric Bourne sh al mateix temps que s'ajusta a l'estàndard POSIX. Llegeix els fitxers `/etc/profile` i `~/.profile`.
- **Condicionals:** Bash accepta expressions condicionals dins la seva nomenclatura. Aquestes poden ser unàries (per exemple, examinar l'estat d'un fitxer), o binaries (com comparadors de cadenes o comparadors numèrics) que requereixen dos objectes per fer la comparació.
- **Shell aritmètic:** Bash permet avaluar les expressions aritmètiques, com una de les expansions de l'intèrpret d'ordres o bé mitjançant l'entrada incorporada. L'avaluació es realitza en enters sencers sense comprovar el desbordament, tot i que la divisió per 0 si està atrapada i marcada com un error. Els operadors i la seva prioritat són els mateixos que en el llenguatge C.
- **Àlies:** Com ja hem estudiat, els àlies permeten substituir una cadena per una paraula quan s'utilitza com a primera paraula d'un comandament simple. Bash manté una llista d'àlies que es poden establir i desmarcar amb les ordres `alias` i `unalias`. Dins dels **scripts** cal tenir en compte en definir els àlies al començament per tal que puguin estar disponibles a tot el script i que, si s'assignen dins d'una funció, aquests no estaran disponibles fins que aquesta s'executi.

1.3. EXECUCIÓ DE COMANDES

Bash determina el **tipus de programa que s'ha d'executar**. Els programes normals són comandes del sistema que existeixen en forma compilada al vostre equip. Quan s'executa aquest tipus de programa, es crea un nou procés perquè Bash fa una còpia exacta de si mateix. Aquest procés secundari té el mateix entorn que el pare, només el número d'identificació del procés (PID) és diferent. Aquest procediment s'anomena **forking**.

Després del procés de *forking*, l'espai d'adreces del procés secundari es sobreescriu amb les dades de un nou procés. Això es fa mitjançant una crida **exec** del sistema. El mecanisme **fork-and-exec** modifica d'aquesta manera una antiga comanda amb una nova, mentre que l'entorn en què s'executa el nou programa continua sent el mateix, inclosa la configuració dels dispositius d'entrada i sortida, variables d'entorn i prioritat.

Les **ordres integrades** estan contingudes dins de la mateixa shell. Quan el nom d'una ordre integrada s'utilitza com a primera paraula d'una comanda simple, la shell executa l'ordre directament, sense crear un procés nou. Bash admet els següents tipus d'ordres integrats:

- **Integració de Bourne Shell →** `:`, `.`, `break`, `cd`, `continue`, `eval`, `exec`, `exit`, `export`, `getopts`, `hash`, `pwd`, `readonly`, `return`, `set`, `shift`, `test`, `[`, `times`, `trap`, `umask` i `unset`.
- **Comandes integrats de Bash →** `alias`, `bind`, `builtin`, `command`, `declare`, `echo`, `enable`, `help`, `let`, `local`, `logout`, `printf`, `read`, `shopt`, `type`, `typeset`, `ulimit` i `unalias`.

Quan el programa que s'està executant és un **script**, bash crearà un nou procés bash amb fork. Aquest "subshell" llegeix les línies del script una per una. Les comandes de cada línia es llegeixen, interpreten i executen com si anessin directament des del teclat al shell. Mentre que el "subshell" processa cada línia de la seqüència de comandes, la shell primària espera que el procés secundari finalitzi. Quan no hi ha més línies al script per llegir, el "subshell" finalitza. El shell principal es desperta i mostra un nou indicador (veurem algun exemple a l'ACT 1 | Execució de scripts.

El esquema fork-and-exec anterior només s'aplica després que l'intèrpret d'ordres ha analitzat la informació. Bash llegeix la informació de comandes i la divideix en paraules i operadors, utilitzant les regles de metadades per definir el significat de cada caràcter d'entrada (per exemple, – indica opció). Fa l'expansió d'àlies i a continuació, aquestes paraules i operadors es tradueixen en ordres i altres construccions (arguments). Es comprova l'existència, o no, d'arguments de redirecció i després s'executen les comandes. Aquestes retornen un estat de sortida disponible per a la inspecció o el processament.

Una simple comanda de shell com `touch file1 file2` consisteix en el comandament seguit d'arguments, separats per espais. Les ordres de shell més complexes es componen d'ordres simples disposades de diverses maneres: en una canonada en què la sortida d'una comanda es converteix en l'entrada d'una segona (per exemple, `ls | more`), en un bucle o en una construcció condicional, o en alguna altra agrupació (per exemple, les funcions[2]).

[2] Les funcions de Bash són una manera d'agrupar comandaments per a la seva posterior execució utilitzant un sol nom per al grup. S'executen com una ordre "regular". Quan s'utilitza el nom d'una funció de shell com a nom de comanda simple, s'executa la llista d'ordres associades amb aquest nom de funció. Les funcions de shell s'executen en el context del shell actual; no es crea un nou procés per interpretar-los.

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

1.4. DESENVOLUPANT BONS SCRIPTS

Hauríem de tenir en compte algunes consideracions inicials:

- Un script s'ha d'executar sense errors.
- Hauria de realitzar la tasca per a la qual es pretén.
- La lògica del programa ha d'estar clarament definida.
- Un script no fa treballs innecessaris.
- Els scripts han de ser reutilitzables.

L'**estructura** d'un script shell és molt flexible. Tot i que a Bash es concedeix molta llibertat, heu d'assegurar la lògica correcta, el control del flux i l'eficiència perquè els usuaris que executin el script puguin fer-ho de manera fàcil i correcta. En començar un script nou, feu-vos les següents preguntes:

- Vaig a necessitar informació de l'usuari o de l'entorn de l'usuari?
- Com emmagatzemaré aquesta informació?
- Hi ha fitxers que calgui crear? On i amb quins permisos i propietats?
- Quines ordres utilitzaré? Quan s'utilitza l'script en diferents sistemes, tots aquests sistemes tenen aquestes ordres en les versions requerides?
- L'usuari necessita notificacions? Quan i per què?

En la següent taula trobareu una visió general de **termes de programació** que necessitareu:

Terme	Què és?
Control de comandes	Provar l'estat de sortida d'una ordre per determinar si cal executar una part del programa.
Branca condicional	Punt lògic del programa quan una condició determina què passa després.
Flux lògic	El disseny general del programa. Determina la seqüència lògica de tasques perquè el resultat sigui reeixit i controlat.
Bucle	Part del programa que es fa zero o més vegades.
Entrada de l'usuari	La informació subministrada per una font externa durant l'execució del programa es pot emmagatzemar i recuperar quan sigui necessari.

Per tal d'accelerar el procés de desenvolupament, l'**ordre lògic d'un programa** s'hauria de pensar amb antelació. Aquest és el vostre primer pas en desenvolupar un script. Es poden utilitzar diversos mètodes; un dels més habituals és treballar amb **llistes numerades** de les tasques implicades en un programa, amb les quals podreu descriure cada procés (i referenciar cada comanda, o comandes, amb quina tasca de la llista desenvolupen).

L'ús del vostre propi idioma per definir les tasques que ha d'executar el vostre programa us ajudarà a crear una forma comprensible del vostre programa. Més endavant, podeu reemplaçar les declaracions del llenguatge quotidià amb paraules i construccions del llenguatge shell.

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

1.5. PRIMER EXEMPLE DE BASH SCRIPT

El script **misistema.sh**, que veieu a continuació, executa algunes ordres conegudes (`date`, `w`, `uname`, `uptime`) per mostrar informació sobre tu i la teva màquina:

```
1  #!/bin/bash
2  clear
3  echo "Aquesta és la informació de misistema.sh. El programa començarà ara."
4
5  echo "Hola, $USER"
6  echo
7
8  echo "Avui és `date`, estem a la setmana `date +"%V"` de l'any."
9  echo
10
11  echo "Aquests usuaris estan connectats actualment:"
12  w | cut -d " " -f 1 - | grep -v USER | sort -u
13  echo
14
15  echo "Aquest és 'uname -s' corrent sobre un processador 'uname -m'."
16  echo
17
18  echo "Aquesta és la informació de uptime:"
19  uptime
20  echo
21
22  echo "Això és tot amics!"  #un text de despedia
```

Un script sempre comença amb els mateixos dos caràcters, "`#!`". Després d'això, es definirà l'intèrpret d'ordres que executarà les ordres després de la primera línia (en aquest cas `/bin/bash`). En el cas de misistema.sh comença per esborrar la pantalla a la línia 2. La línia 3 fa que imprimiu un missatge[3], informant a l'usuari del que passarà. La línia 5 saluda a l'usuari pel seu nom a través de la variable `$USER` del sistema. Les línies 6, 9, 13, 16 i 20 només existeixen per a finalitats de visualització ordenades. La línia 8 imprimeix la data actual i el número de la setmana. La línia 11 torna a ser un missatge informatiu, com les línies 3, 18 i 22. La línia 12 forma la sortida de la comanda `w` amb un seguit de "filtres de visualització"; la línia 15 mostra informació sobre el sistema operatiu i la CPU. La línia 19 proporciona informació sobre el temps d'activitat i la càrrega. Per últim a la línia 22 veiem que a més d'un missatge trobem un comentari[4] (darrera de `#`).

[3] La comanda `echo` sempre surt amb un estat 0, i simplement imprimeix arguments seguits d'un caràcter final de línia a la sortida estàndard. Molts cops, en substitució d'aquesta comanda utilitzarem `printf` que permet definir una cadena de format i dóna un codi d'estat de sortida diferent de zero quan es produeixi un error.

[4] DOCUMENTEU ELS VOSTRES SCRIPTS: Heu de ser conscients del fet que és possible que no sigueu l'única persona que llegirà el vostre codi. Molts usuaris i administradors de sistemes executen scripts escrits per altres persones. Si volen veure com ho va fer, els comentaris són útils per il·luminar al lector. Els comentaris també faciliten la vostra vida. Imagineu que heu llegit moltes pàgines de man per aconseguir un resultat determinat en unes determinades ordres que utilitza el vostre script. Possiblement, no recordareu com va funcionar si necessiteu canviar el vostre script després de poques setmanes o mesos, llevat que hàgiu comentat el que heu fet, com ho heu fet i/o per què ho heu fet.

1.6. CREANT I EXECUTNAT SCRIPTS

Un bash script és, com ja sabeu, una seqüència d'ordres que haurem d'utilitzar repetidament. Aquesta seqüència normalment s'executa introduint el nom del script a la línia d'ordres. Alternativament, podeu utilitzar scripts dins del crontab per automatitzar tasques. Un altre ús típic dels scripts es troba al procediment d'arrencada i apagat de del sistema, on es defineix l'operació de dimonis i serveis en scripts d'inici/final.

Per **crear** un bash script, obriu un fitxer buit nou a l'editor. Qualsevol editor de text ho farà, tot i que d'alguns es podran configurar per reconèixer la sintaxis pròpia de Bash i poden ser de gran ajuda alhora de prevenir errors, utilitzant colors, auto-completat de text, etc. Per **donar-li nom** al vostre script procureu que sigui un nom coherent i que doni una idea del que fa el script. Assegureu-vos que el nom del vostre script no entra en conflicte amb les ordres existents (per tal d'assegurar-se que no pugui sorgir cap confusió, els noms d'escriptures sovint acaben en .sh), ni amb cap altre script del sistema (sempre podeu fer un `which -a nomscript`, `whereis nomscript` o `locate nomscript` per assegurar-vos si existeix un altre amb el mateix nom al sistema).

Pot ser una bona idea crear un directori `~/scripts` per guardar els vostres scripts. Així podreu afegir el directori al contingut de la variable `PATH` del sistema: `export PATH = "$PATH:~/scripts"`. Això ens permetrà executar els nostres scripts des de qualsevol ubicació del sistema utilitzant tant sols el seu nom[5].

Si no ubiques els scripts dins d'un directori declarat al PATH, caldrà que per executar-lo t'ubiquis en el directori concret on es troba el script i l'executis mitjançant `./nomscript.sh`. Per últim també podem executar els nostres scripts utilitzant altres intèrprets de comandes (per exemple, `sh nomscript.sh`), o depurant (*debugging*) possibles errors: `bash -x nomscript.sh`.

ACT 1 | **Execució de scripts**
Genera un nou fitxer de text amb el següent contingut i anomena'l act1.sh:

```
#!/bin/bash
clear
echo "Hola $USER"
echo
echo "Vaig a llistar el contingut del directori actual"
echo
ls
echo
echo "Vaig a configurar 2 variables"
COLOR="blau"
VALOR="3"
echo "Aquesta és una cadena de text: $COLOR"
echo "I aquest és un número: $VALOR"
echo
echo "Et retorno el teu prompt"
echo
```

[5] Recorda però que un pas previ a l'execució de qualsevol dels teus scripts és l'assignació de permisos (`chmod`) d'execució (`x`) pels usuaris que vols pugin fer córrer el teu script.

a) Crea un directori scripts dins del teu directori home. Guarda el script act1.sh dins, i dona't permisos d'execució només a tu com a propietari del fitxer act1.sh.
b) Ubicat al teu home (no dins del directori scrips) executa: `act1.sh`. No hauria de funcionar. Inclou el teu nou directori ~/scripts al PATH del sistema. Després torna a executar-lo.
c) En acabar l'execució exeuta: `echo $COLOR` i `echo $VALOR`. Quin resultat obtens? Per què?
d) Prova a executar el teu script de la següent manera: `source act1.sh`. Quin resultat obtens? I si tornes a executar les comandes del apartat c) ?

1.7. DEPURANT SCRIPTS

Com comentava anteriorment, podeu depurar (*debbuging*) un script executant-lo amb l'opció **-x**. En fer-ho veureu que a la sortida trobareu cadascuna de les ordres, i els seus arguments, després que aquestes s'han expandit però abans de la seva execució. Això us pot ajudar a veure perquè el vostre script no funciona com desitgeu, i quina comanda (o argument d'aquesta) està retornant un resultat no esperat.

També teniu la possibilitat de no depurar tot el vostre codi, i fixar-vos només en les zones que ja d'entrada considerem problemàtiques, o que no sabeu ben bé quin resultat aportaran. Per fer-ho dins del propi codi del script encapsularem la comanda "conflictiva" de la següent manera:

```
set -x                  #activar la depuració des d'aquí
codi que volem depurar
set +x                  #aturar la depuració des d'aquí
```

Es poden depurar tantes zones com vulgueu dins del codi del script. Després només s'ha d'executar normalment el vostre script, i només es depurarà aquella zona, o aquelles zones, delimitada pels paràmetres `set -x` i `set +x`. A més a més, pot ser recomanable incloure propis missatges (amb `echo`) a mode de "avisadors" que ens puguin indicar que anem a executar, en quin punt del codi estem, o el valor d'una variable en un moment donat.

1.8. VARIABLES

Bash disposa de dos tipus de variables:

1. **Variables globals:** també conegudes con variables d'entorn estan disponibles en totes les terminals del sistema. Podem visualitzar-les amb les comandes `env` o `printenv`.
2. **Variables locals:** aquestes variables només estan disponibles al terminal actual.

A més de fer la distinció entre globals i locals, també podem dividir les variables en categories segons el seu **contingut**. En aquest cas, trobarem 4 tipus de variables:

1. **Variables de cadena (string)**
2. **Variables d'enters (integer)**
3. **Variables constants (constant)**
4. **Variables de matrius (array)**

Per defecte, les variables són sensibles a majúscules i minúscules. Donar un nom en minúscules a les variables locals és una convenció que de vegades s'aplica. Tanmateix, podeu utilitzar els noms que vulgueu o barrejar casos. Les variables també poden contenir dígits, però no es permet un nom que comenci per un dígit. Amb aquestes premisses, **crear una variable** (al nostre terminal actual i com a variable local d'aquest terminal) serà tant fàcil com assignar-li un nom i un valor: `VARNAME="valor"`. No poseu espais al voltant del signe d'igualtat, provocarà errors!

Per **veure el valor** que té assignat una variable podeu executar `echo $VARNAME` (cal precedir el nom de la variable amb el símbol `$`). Si volem **buidar el contingut d'una variable**, és a dir, netejar el seu contingut, utilitzarem la comanda `unset VARNAME`.

Aquestes variables locals només seran accessibles al terminal actual però no als processos secundaris de l'intèrpret d'ordres actual. Per tal de passar variables a un subshell, necessitem exportar-les mitjançant l'ordre d'exportació incorporada. Les variables que s'exporten són variables d'entorn. Normalment, la configuració i l'exportació es fan en un sol pas: `export VARNAME = "valor"`. Un subshell pot canviar les variables que va heretar del pare, però els canvis fets pel fill no afecten el pare.

ACT 2 | **Variables locals i d'entorn**

Anem a comprovar el funcionament de la creació de variables locals i la seva posterior exportació com a variables d'entorn al nostre sistema.

a) Declara una variable anomenada *nom_complert* amb valor una cadena de text amb el teu nom i cognoms"
b) Mostra el valor de la variable anterior al teu terminal: `echo $nom_complert`.
c) Llançarem un subshell del nostre terminal (executeu `bash`) i torneu a fer l'apartat b). Com veieu queda provat que la variable *nom_complert* és del tipus local.
d) Executeu `exit` per tornar al vostre bash pare. Ara exporteu la variable local *nom_complert* per tal de convertir-la en una variable d'entorn.
e) Torneu a llançar un subshell i comproveu que efectivament ara si podem obtenir el nom de la variable *nom_complert* des d'aquest subshell fill del anterior.
f) Canviarem el valor de la variable de la següent manera: `export nom_complert:"Rafa García"`. Després contesta, quin valor té ara la variable d'entorn *nom_complert* al subshell i al terminal pare?
g) Per últim, neteja el contingut del contingut de la variable *nom_complert*.

A més Bash, com qualsevol altre intèrpret de comandes, té una **llista de variables reservades**. Totes aquestes variables d'entorn tenen un significat especial per l'intèrpret de comandes, i algunes d'elles són només de lectura, i d'altres, se'ls assigna certs valors automàticament, perdent en seu significat si canviem aquests valors.

h) Cerca alguna llista o taula de les variables d'entorn reservades per Bash així com una petita descripció del seu significat i/o valors per defecte.

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

A més de tenir reservats els noms propis de les comandes, i totes les variables d'entorn reservades que heu pogut cercar a l'activitat anterior, també existeixen uns certs **paràmetres especials** que poden ser referenciats, però no se'ls permet l'assignació d'un valor:

Caràcter	Definició
`$*`	Conté els paràmetres de posició, començant per un. Quan el seu valor està entre cometes dobles, s'interpreta com a una sola paraula amb el valor de cada paràmetre separat pel primer caràcter de la variable especial IFS.
`$@`	Conté els paràmetres de posició, començant per un. Quan el seu valor està entre cometes dobles, cada paràmetre s'interpreta a una paraula separada. [6]
`$#`	Conté el número de paràmetres de posició en decimal.
`$?`	Conté l'estat de sortida de la canonada de primer pla més recentment executada.
`$-`	Conté les banderes d'opció actuals tal com s'especifica en la invocació, mitjançant l'ordre integrat del conjunt, o les establertes pel propi intèrpret d'ordres.
`$$`	Conté l'ID del procés de l'intèrpret d'ordres.
`$!`	Conté l'ID de procés de la comanda de fons (asíncrona) més recentment executada.
`$0`	Conté el nom de l'intèrpret d'ordres o del script.
`$_`	Al inici conté el nom de fitxer absolut de l'intèrpret d'ordres o del script que s'executa tal com es passa a la llista d'arguments. Posteriorment, conté a l'últim argument de la comanda anterior.

Veiem alguns exemples on utilitzarem alguns d'aquests paràmetres. Disposem del següent script:

```
#!/bin/bash

# positional.sh
# Aquest script llegeix 3 paràmetres de posició i els escriu per pantalla.

POSPAR1="$1"
POSPAR2="$2"
POSPAR3="$3"

echo "$1 és el primer paràmetre de posició, \$1."
echo "$2 és el segon paràmetre de posició, \$2."
echo "$3 és el tercer paràmetre de posició, \$3."
echo
echo "El número total de paràmetres de posició és $#."
```

[6] La implementació de `$*` sempre ha estat un problema i realment s'hauria de substituir pel comportament de `$@`. En gairebé tots els casos en què els programadors utilitzen `$*`, realment volen dir `$@`. A més a més, `$*` pot causar errors i fins i tot forats de seguretat als vostres scripts.

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

En les següents execucions obtindríem aquests resultats:

```
rgarcia@ubuntu:~$ positional.sh one two three four five
one és el primer paràmetre de posició, $1.
two és el segon paràmetre de posició, $2.
three és el tercer paràmetre de posició, $3.

El número total de paràmetres de posició és 5.

rgarcia@ubuntu:~$ positional.sh one two
one és el primer paràmetre de posició, $1.
two és el segon paràmetre de posició, $2.
 és el tercer paràmetre de posició, $3.

El número total de paràmetres de posició és 2.
```

Observeu ara la següent seqüència de comandes a l'intèrpret de comandes:

```
rgarcia@ubuntu:~$ grep dictionary /usr/share/dict/words
dictionary

rgarcia@ubuntu:~$ echo $_
/usr/share/dict/words

rgarcia@ubuntu:~$ echo $$
10662

rgarcia@ubuntu:~$ mozilla &
[1] 11064

rgarcia@ubuntu:~$ echo $!
11064

rgarcia@ubuntu:~$ echo $0
bash

rgarcia@ubuntu:~$ echo $?
0

rgarcia@ubuntu:~$ ls noexisteix
ls: noexisteix: No such file or directory

rgarcia@ubuntu:~$ echo $?
1
```

L'usuari rgarcia comença a introduir l'ordre `grep`, que resulta en l'assignació de la variable `_`. L'identificador del procés del seu intèrpret d'ordres és 10662 (emmagatzemat a `$$` i que es podria comprovar amb la comanda `ps`). Després de posar un treball en segon pla (`mozilla &`), la variable `!` manté l'ID de procés de la feina de fons. L'execució de l'intèrpret d'ordres és bash com podem veure al valor de la variable `$0`. Quan es fa un error, la variable `?` té un codi de sortida diferent de 0 (zero).

1.9. EXPANSIÓ DELS ARGUMENTS DEL SHELL

Després de que l'ordre s'hagi dividit en fitxes (arguments), aquestes fitxes o paraules s'expandeixen o es resolen. Hi ha vuit tipus d'expansió realitzats que és resolen en el següent ordre:

1. **Expansió de claus:** quan un dels arguments de l'ordre consisteix en una cadena emmarcada per claus `{}`i dividida per comes, aquesta es resol amb els diferents valors possibles per a cadascun dels elements de la cadena, per exemple, `echo pa{pe,pi,po}r`, s'expandiria a la sortida com: `paper papir papor`.

2. **Expansió de titlla:** quan un dels arguments de l'ordre ve precedit per una titlla (~) es resol la cadena corresponent fins a la primera barra (/) que ens indicarà la separació del següent argument de ruta. Si la titlla (~) no va seguit de cap caràcter és substitueix per la variable global `HOME` (directori inicial del usuari que executa el shell), per exemple: `export PATH=~/testdir`, es resoldria com `$HOME/testdir`, que posteriorment (expansió de variables) podria ser traduït per exemple, com `/home/rgarcia/testdir`. Aquest és el seu us més habitual tot i que també hi ha altres opcions interessants: si ens trobem `~+` la expansió es resoldrà com la variable global `$PWD` (directori actual de treball), i si ens trobem `~-` com la variable global `$OLDPWD` (darrer directori abans de l'última comanda `cd`, si està definida)[7].

3. **Expansió de paràmetres del shell i variables:** s'aplica quan els arguments a expandir venen precedits pel caràcter `$`[8]. El nom del paràmetre que segueix a `$` es pot emmarcar entre claus `{}` opcionalment com a mesura de protecció per delimitar clarament quins caràcters formen part del paràmetre. La seva sintaxis habitual per tant seria `${PARAMETRE}` tot i que molts cops ho trobem simplement com `$PARAMETRE` (per exemple, `echo $SHELL`, que s'expandirà com `/bin/bash`). Podem aprofitar l'expansió de variable per crear-les (amb un valor concret) si aquestes no existeixen, per exemple, `echo ${NEWVAR:=259}` que s'expandiria i retornaria directament a la sortida el valor `259`. Es pot realitzar també l'anomenada **expansió indirecta** que vindrà precedida pel símbol d'exclamació (`!`) traduint com a paràmetre el propi nom de la variable en comptes d'expandir el seu valor, per exemple, observeu la següent situació:

```
rgarcia@ubuntu:~$ NAME="VARIABLE"
rgarcia@ubuntu:~$ VARIABLE=259
rgarcia@ubuntu:~$ echo ${NAME} #les claus son opcionals
VARIABLE
rgarcia@ubuntu:~$ echo ${!NAME}
259
```

[7] Quan treballem amb el nostre terminal, a més de les variables `$HOME`, `$PWD` i `$OLDPWD`, es va guardant una llista de directoris visitats. Aquesta llista la podeu visualitzar executant `dirs -v`. Una altra expansió de titlla possible seria `~+N` o `~-N` que s'expandiria amb el valor del directori que es troba a la posició N de llista de directoris visitats, començant per la esquerra (+) o per la dreta (-) i tenint en compte que la primera posició (la posició 0) es correspon amb el directori actual (o `$PWD`).

[8] L'expansió de comandes i l'expansió aritmètica també vindran precedides del caràcter `$`. Per tant, no és un indicador únic per l'expansió de paràmetres i variables.

Autor | RAFAEL GARCÍA SUÁREZ
Contacte | rgarc259@xtec.cat

4. **Expansió de comandes:** s'aplica quan els arguments a expandir venen precedits pel símbol `$` seguit d'una comanda emmarcada entre parèntesis `()`. És equivalent a la notació que emmarca les comandes entre `` ` ` `` (backticks). La substitució d'ordres permet que la sortida d'una ordre substitueixi la mateixa ordre. Per exemple, `echo $(date)`, o `echo` `` `date` `` que s'expandiria a la sortida com `Thu Feb 9 12:54:09 CET 2019` (per exemple).

5. **Expansió aritmètica:** s'aplica quan els arguments a expandir venen precedits pel símbol `$` seguit per una expressió aritmètica emmarcada entre parèntesis dobles `(( ))`. Les expressions aritmètiques es fan amb nombres enters en base decimal[9] d'amplada fixa i sense comprovació desbordaments (tot i que per exemple la divisió entre 0 està capturada com error). El format d'aquestes expressions és molt similar al utilitzat pel llenguatge de programació C. Observeu la següent taula:

Operador	Significat
`VAR++` i `VAR--`	Incrementa i decrementa el valor d'una variable (a posteriori)
`++VAR` i `--VAR`	Incrementa i decrementa el valor d'una variable (a priori)
`-` i `+`	Indicador de número negatiu i positiu
`!` i `~`	Negació lògica i negació bit a bit
`**`	Exponenciació
`*`, `/` i `%`	Multiplicació, divisió i residu
`+` i `-`	Suma i resta
`<<` i `>>`	Desplaçament esquerra i dreta bit a bit
`<=`, `>=`, `<` i `>`	Menor o igual, major o igual, menor i major
`==` i `!=`	Igual o diferent
`&`	AND bit a bit
`^`	XOR bit a bit
`|`	OR bit a bit
`&&`	AND lògica
`||`	OR lògica
`=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=` i `|=`	Assignacions (per exemple, `VAR *= 2`, assignaria el a la variable `VAR` el seu valor anterior multiplicat per 2)
`,`	Separador entre expressions

Els operadors s'avaluen per ordre de prioritat. Les subexpressions entre parèntesis s'avaluen primer i poden anul·lar les regles de precedència anteriors. Així doncs, per exemple, no obtindrem el mateix resultat fent: `echo $((3*4+2))`, amb resultat `14`, que fent: `echo $((3*(4+2)))`, amb resultat `18`.

[9] Podem treballar amb números octals si els escrivim un 0 a l'esquerra del valor (per exemple, 074 s'entendria com un número octal i no el 74 decimal), o amb números hexadecimals si escrivim un 0x o 0X a l'esquerra del valor (per exemple, 0x4B). De fet podem treballar en qualsevol base (de 2 a 64) utilitzant BASE# a l'esquerra del valor (per exemple, 3#21 s'entendria com un número en base 3 i no el 21 decimal).

ACT 3 | **Paràmetres especials i operacions aritmètiques**

Recorda el que has après entorn als paràmetres especials, l'assignació de variables i la realització d'operacions aritmètiques en l'entorn Bash, per generar un script que calculi la superfície d'un rectangle, tenint en compte les següents consideracions:

- Ubica el teu script dins del directori *scripts* del teu directori personal i que està inclòs a la variable global PATH, i anomena'l *rectangle.sh*.
- Les mides de la base i l'altura del rectangle del qual hem de calcular la seva superfície es passaran com a arguments al cridar al nostre script. Per exemple, `rectangle 10 5`.
- El resultat del càlcul de l'àrea l'has d'emmagatzemar en una variable local anomenada RESULT.
- Inclou missatges durant la execució del teu script per fer-ho més descriptiu i elegant de cara a l'usuari que l'executa.

6. **Expansió de processos:** s'aplica quan els arguments a expandir venen precedits pel símbol `>` o el símbol `<` seguits per una llista de comandes emmarcades entre parèntesis `()`. El seu funcionament és similar al que faria una canonada (`|`) connectant la `stdout` d'una comanda amb la `stdin` d'una altra, però ens aporta una funcionalitat més poderosa ja que ens permet fer la canonada des de múltiples processos, és a dir, podem utilitzar la `stdout` de cadascun dels processos de la llista amb la `stdin` d'una comanda, utilitzant per emmagatzemar cadascuna de les sortides els fitxers especials ubicats a a `/dev/fd` (o en la seva absència altres fitxers temporals).

7. **Divisió de paraules:** el shell analitza els resultats de l'expansió de paràmetres, la substitució d'ordres i l'expansió aritmètica que no es van produir entre cometes dobles per dividir paraules. Aquesta divisió es basa en els delimitadors indicats per la variable global `$IFS` (per defecte els delimitadors són `<space><tab><newline>`), que no formaran part de les paraules delimitades, excepte si estan emmarcats entre cometes dobles (per exemple `" "`). Noteu que si no és fa cap expansió de les anteriors, no caldrà fer cap divisió de paraules.

8. **Expansió del nom del fitxer:** després de dividir les paraules, Bash escaneja cada paraula pels caràcters `*`, `?` i `[`. Si apareix un d'aquests caràcters, la paraula es considerarà PATTERN (patró) i s'ha de substituir per una llista ordenada alfabèticament de noms de fitxers que coincideixin amb el patró[10]. La variable global `GLOBIGNORE` es pot utilitzar per restringir el conjunt de noms de fitxer que coincideixen amb un patró. Si s'estableix GLOBIGNORE, s'elimina cada un dels noms del fitxer que coincideixi amb un dels patrons de la llista de coincidències de `GLOBIGNORE`. Els noms dels fitxers `.` i `..` (qui soc i qui és el meu pare) sempre s'ignoren, i els que comencen amb un `.` (fitxers ocults) han de coincidir explícitament amb el patró a expandir o no es consideraran.

[10] En cas de no trobar cap nom de fitxer coincident amb el patró s'actuarà de dues maneres diferents segons si la opció d'enviament `nullglob` està o no activada. Si no ho està la paraula no es modificarà, si està activada la paraula es suprimeix.

1.10. EXPRESSIONS REGULARS

Una **expressió regular** és un patró que descriu un conjunt de cadenes (*strings*). Les expressions regulars es construeixen de forma anàloga a les expressions aritmètiques utilitzant diversos operadors per combinar expressions més petites.

Els elements bàsics són les expressions regulars que coincideixen amb un sol caràcter. La majoria de caràcters, incloses totes les lletres i els dígits, són expressions regulars que coincideixen amb si mateixes. Es pot utilitzar també qualsevol metacaràcter amb un significat especial (*, ?, etc.) precedint-lo amb una barra invertida (\).

Una expressió regular pot ser seguida per un dels diversos **operadors de repetició**. Observeu la següent taula:

Operador	Efecte
`.`	Coincideix amb qualsevol caràcter simple
`?`	L'element que el precedeix és opcional i coincidirà com a molt una vegada
`*`	L'element que el precedeix coincidirà zero o més vegades
`+`	L'element que el precedeix coincidirà una o més vegades
`{N}`	L'element que el precedeix coincidirà exactament N vegades
`{N,}`	L'element que el precedeix coincidirà N o més vegades
`{N,M}`	L'element que el precedeix coincidirà un mínim de N vegades, i no més de M vegades
`-`	Representa un interval si no està situat al principi o final d'una llista, o el final d'un interval si es troba al final d'una llista.
`^`	Coincidirà amb la "cadena buida" al principi d'una línia, o els caràcters que no és troben en una llista
`$`	Coincideix amb la "cadena buida" al final d'una línia
`\b`	Coincideix amb la "cadena buida" al costat d'una paraula
`\B`	Coincideix amb la "cadena buida" sempre que no estigui al costat d'una paraula
`\<`	Fa coincidir la "cadena buida" al començament d'una paraula
`\>`	Fa coincidir la "cadena buida" al final d'una paraula

Es poden **concatenar** dues expressions regulars; l'expressió regular resultant coincideix amb qualsevol cadena formada concatenant dues subcadenes que corresponen respectivament a les subexpressions concatenades.

Es pot **unir** dues expressions regulars amb l'operador `|`. L'expressió regular resultant coincideix amb qualsevol cadena que coincideixi amb qualsevol subexpressió.

La **repetició** té prioritat sobre la concatenació, que al seu torn preval sobre l'alternança. Es pot incloure una subexpressió completa entre parèntesis per anul·lar aquestes regles de precedència.

Veiem alguns exemples d'ús d'expressions regulars a través de la comanda **`grep`**. Recorda (ACT 4) que `grep` cerca els fitxers d'entrada per a línies que continguin una coincidència a una llista de patrons. Quan troba una coincidència, copia tota la línia a la sortida estàndard, o qualsevol altre tipus de sortida indicada.

Autor | RAFAEL GARCÍA SUÁREZ
Contacte | rgarc259@xtec.cat

ACT 4 | **Funcionament i opcions de la comanda grep**

Per tal de recordar el funcionament bàsic de la comanda `grep`, digues quina comanda resol cadascun dels següents punts:

a) Cerca al fitxer /etc/passwd totes les línies que conten la cadena *root*.
b) Torna a obtenir el resultat del apartat a) però incloent el número de línia on localitzem la cadena.
c) Cerca quins usuaris no utilitzen Bash com a intèrpret de comandes.
d) Torna a obtenir el resultat de l'apartat c) però excloent de la cerca aquells que tenen definit com shell *nologin*.
e) Mostra només quants usuaris tenen un shell *FALSE*. Quants resultats obtens? Fes que s'ignorin les majúscules i minúscules en la teva cerca. Quants resultats obtens ara?
f) Cerca la línia del fitxer /etc/fstab que munta l'arrel del sistema (/). Et caldrà l'ajuda de l'opció -w de la comanda grep. Consulta el manual per conèixer el seu significat.

Veieu doncs exemples d'ús d'expressions regulars amb la comanda `grep`:

- Imagineu que voleu realitzar la cerca de l'apartat a) de l'ACT 4, però mostrant només les línies que comencen amb la cadena *root*. Segons la taula anterior, executeu: `grep ^root /etc/passwd`.
- Si volem localitzar els usuaris que no tenen cap shell assignat, podríem pensar en buscar les línies que acaben amb el caràcter `:`, per fer-ho executeu: `grep :$ /etc/passwd`.
- Per comprovar que la variable `PATH` s'exporta al fitxer `/.bashrc`, caldrà buscar les línies que contenen la cadena *export* i després d'aquestes localitzar les que conten la cadena *PATH*. El problema es que si executeu: `grep export ~/.bashrc | grep PATH`, obtindreu altres variables, tipus `MANPATH`, que contenen la cadena *PATH*. Per excloure aquestes sortides et cal utilitzar una expressió regular com la següent: `grep export ~/.bashrc | grep '\<PATH'`.
- Recordant l'ús dels claudàtors per definir llistes, executeu: `grep [yf] /etc/group`, per obtenir totes les línies del fitxer de grups que conten els caràcters `y` i/o `f`. Recordeu que podeu establir intervals, per exemple, `[0-9]` equivaldria a `[0123456789]`, o utilitzant `[^0-9]` correspondria a qualsevol caràcter que no fos un dígit.[11]
- Pel que fa a l'ús de comodins (*wildcards*) disposem del caràcter `.` i el caràcter `*`. Proveu a executar: `grep '\<c..a\>' /usr/share/dict/spanish`. Podeu observar com cercarà totes les paraules del diccionari que comencen pel caràcter `c` seguit per 2, i només 2, caràcters qualsevols i terminen amb el caràcter `a`. Si algun cop voleu buscar línies que continguin el punt (`.`) expressament com a caràcter haureu d'utilitzar la opció `-F` de la comanda `grep`. Si voleu les paraules que comencen per c i acaben en a, tinguin el número de caràcters que tinguin al mig, utilitzarem el caràcter `*`. Proveu el següent: `grep '\<c.*a\>' /usr/share/dict/spanish`. Si algun cop voleu buscar les línies que continguin l'asterisc (`*`) expressament caldrà utilitzar les cometes simples. Veieu que no obteniu el mateix resultat amb aquestes dues comandes: `grep * /etc/profile` i `grep '*' /etc/profile`.

[11] L'equivalència estàndard de `[a-d]` seria `[abcd]` segons la nomenclatura de C. Potser ens trobem en algun sistema on la substitució de l'interval `[a-d]` es correspongui amb `[aAbBcCdD]`. Podeu executar `echo $LC_ALL` per veure si el valor d'aquesta variable d'entorn és "C".

Tenim disponibles també les denominades **classes de caràcters** que podem especificar entre claudàtors amb la següent estructura: [:CLASS:], on CLASS pot prendre els següents valors definits a l'estàndard POSIX: `alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `word` o `xdigit`.

ACT 5 | **Classes de caràcters**
Comprovareu el funcionament d'algunes de les classes de caràcters que heu vist:

a) Dins del teu directori personal genera un directori anomenat *classes*.
b) Situat dins del nou directori *classes* i executa: `mkdir Docs imatges 259`.
c) Ara que tens 3 directoris creats, executa `ls -ld [[:alnum:]]*`, llistant tots els directoris dins de classes de caràcters que continguin lletres i números, és a dir, els tres que existeixen.
d) Executa ara `ls -ld [[:lower:]]*`, hauries d'obtenir només el directori que conté lletres només en minúscules, és a dir, *imatges*. Quina classe de caràcter utilitzaries per tal de retornar *Docs*? I per retornar *259* ?
e) Executa novament l'ordre ara utilitzant la classe d'objecte `xdigit`. Per què obtens aquesta sortida? Cerca quina sortida proporciona cadascuna de les classes d'objecte de la llista POSIX.

1.11. L'EDITOR SED

L'editor **Stream EDitor** (SED) s'utilitza per realitzar transformacions bàsiques de text llegit des d'un fitxer o d'una canonada. El resultat s'envia a la sortida estàndard. La sintaxi per a l'ordre `sed` no té especificació de fitxer de sortida, però els resultats es poden desar a un fitxer mitjançant la redirecció de sortida. L'editor no modifica l'entrada original.

El que distingeix aquests editors d'altres editors, com ara nano o vi, és la seva capacitat per filtrar el text que rep d'una canalització. No necessiteu interactuar amb l'editor mentre s'executa; és per això que, de vegades, SED és conegut com editor de lots. Aquesta característica permet utilitzar comandes d'edició en scripts, facilitant en gran mesura les tasques d'edició repetitives.

El programa `sed` pot realitzar substitucions i supressions de patrons de text utilitzant expressions regulars (com les utilitzades amb l'ordre `grep`). Les ordres d'edició són similars a les que s'utilitzen a l'editor vi. Veieu la següent taula:

Ordre	Resultat
`\a`	Afegeix text per sota de la línia actual
`\c`	Canvia el text de la línia actual per un nou text
`d`	Suprimeix el text
`\i`	Insereix un text per sobre de la línia actual
`p`	Imprimeix el text
`r`	Llegeix un fitxer
`s`	Cercar i substituir text
`w`	Escriure a un fitxer

A més d'editar comandes, podeu donar opcions a `sed`. Teniu una descripció general a la taula següent, recordeu que a les pàgines de manual de `sed` trobareu informació més complerta.

Opció	Efecte
`-e SCRIPT`	Afegeix les ordres en SCRIPT al conjunt d'ordres que s'executaran durant el processament de l'entrada
`-f`	Afegeix les ordres contingudes al fitxer SCRIPT-FILE al conjunt d'ordres que s'executaran durant el processament de l'entrada
`-n`	Mode silenciós
`-V`	Imprimir informació sobre la versió i sortir

Utilitzarem un exemple simple[12] per entendre alguns funcionaments del l'**edició interactiva** mitjançant l'editor SED. Creeu un fitxer de text anomenat *exemple* amb el següent contingut:

```
rgarcia@ubuntu:~$ cat -n example
     1  This is the first line of an example text.
     2  It is a text with erors.
     3  Lots of erors.
     4  So much erors, all these erors are making me sick.
     5  This is a line not containing any errors.
     6  This is the last line.
```

Començarem buscant un patró (cosa que ja sabem fer amb `grep`), en aquest cas "erors". Fixeu-vos que utilitzem l'ordre `p` de l'editor per imprimir el resultat:

```
rgarcia@ubuntu:~$ sed 'erors/p' example
This is the first line of an example text.
It is a text with erors.
It is a text with erors.
Lots of erors.
Lots of erors.
So much erors, all these erors are making me sick.
So much erors, all these erors are making me sick.
This is a line not containing any errors.
This is the last line.
```

Noteu que `sed` imprimeix tot el fitxer *example*, però les línies que contenen la cadena de cerca s'imprimeixen dues vegades.

Possiblement això no és el que volem. Per imprimir només aquestes línies que coincideixin amb el nostre patró, utilitzeu l'opció `-n` (mode silenciós):

[12] Podeu trobar molts exemples de scripts que utilitzen `sed` ubican-te als directoris `/etc/init.d` o `/etc/rc.d/init.d` i executant `grep sed *`.

```
rgarcia@ubuntu:~$ sed -n 'erors/p' example
It is a text with erors.
Lots of erors.
So much erors, all these erors are making me sick.
```

Ara només volem veure les línies que no contenen la cadena de cerca:

```
rgarcia@ubuntu:~$ sed 'erors/d' example
This is the first line of an example text.
This is a line not containing any errors.
This is the last line.
```

Com veieu l'ordre `d` suprimeix el text de la sortida, tot i que recorda que `sed` no modifica el la entrada original, en aquest cas, no eliminarà les línies del fitxer *exemple*.

Podem també cercar les coincidents que comencen amb un patró determinat i acaben en un segon patró es mostren així:

```
rgarcia@ubuntu:~$ sed -n '/^This.*errors.$/p' example
This is a line not containing any errors.
```

Tingueu en compte que l'últim punt s'ha d'escapar (`$`) per poder coincidir realment. En el nostre exemple, l'expressió coincideix amb qualsevol caràcter, inclòs l'últim punt.

Aquesta vegada volem eliminar les línies que contenen els errors (patró *erors*). A l'exemple s'indiquen les línies 2 a 4. Especifiqueu aquest interval per dirigir-lo a la l'ordre `d`:

```
rgarcia@ubuntu:~$ sed '2,4d' example
This is the first line of an example text.
This is a line not containing any errors.
This is the last line.
```

Per imprimir el fitxer fins a una determinada línia del fitxer, utilitzeu una comanda similar a aquesta:

```
rgarcia@ubuntu:~$ sed '3,$d' example
This is the first line of an example text.
This is a line not containing any errors.
This is the last line.
```

Podeu marcar patrons d'inici i final. Per exemple, la següent comanda imprimeix la primera línia que conté el patró "a text", fins la línia que conté el patró "This":

```
rgarcia@ubuntu:~$ sed -n '/a text,/This/p' example
It is a text with erors.
Lots of erors.i
So much erors, all these erors are making me sick.
This is a line not containing any errors.
```

Veieu una funcionalitat més potent que no només cercar, observeu com podem cercar i reemplaçar:

```
rgarcia@ubuntu:~$ sed 's/ erors/errors/' example
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these erors are making me sick.
This is a line not containing any errors.
This is the last line.
```

Com podeu veure, aquest no és exactament l'efecte desitjat: a la línia 4, només s'ha substituït la primera aparició de la cadena de cerca i encara queda un "erors". Utilitzeu l'ordre `g` per indicar a `sed` que hauria d'examinar tota la línia en lloc de parar a la primera aparició de la vostra cadena:

```
rgarcia@ubuntu:~$ sed 's/ erors/errors/g' example
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the last line.
```

Recordeu, un cop més, que l'execució de la comanda *sed* no modificarà l'entrada (el fitxer *example*) i per tant només hem mostrat la correcció modificada però no s'ha guardat a cap lloc. Per fer-ho executaríem la comanda amb la redirecció de sortida: `sed 's/ erors/errors/g' example > example_corregit`.

Per fer substitucions múltiples a un fitxer utilitzeu les ordres de reemplaçament separades individualment per l'opció `-e`:

```
rgarcia@ubuntu:~$ sed -e 's/ erors/errors/g' -e 's/ last/final/g' example
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the final line.
```

Per últim considereu com afegir contingut al vostre text. Observeu els següents exemple on afegirem un caràcter > al començament de cada línia; i al segon exemple, un `EOL` (*end of line*) al final de cadascuna:

```
rgarcia@ubuntu:~$ sed 's/^/> /' example
> This is the first line of an example text.
> It is a text with erors.
> Lots of erors.
> So much erors, all these erors are making me sick.
> This is a line not containing any errors.
> This is the last line.

rgarcia@ubuntu:~$ sed 's/$/EOL/' example
This is the first line of an example text.EOL
It is a text with erors.EOL
Lots of erors.EOL
So much erors, all these erors are making me sick.EOL
This is a line not containing any errors.EOL
This is the last line.EOL
```

ACT 6 | **Treballant amb Stream EDitor**

Realitza els següents exercicis utilitzant l'editor SED per tal de revisar les funcionalitats que heu vist en aquest apartat:

a) Crea un fitxer anomenat *temp.txt* al directori *scripts* del teu directori personal. Després imprimiu una llista dels fitxers del directori *scripts* que finalitzem amb ".sh".
b) Feu una llista de fitxers al directori `/usr/bin` que tinguin la lletra "a" com a segon caràcter. Poseu el resultat al fitxer *temp.txt*.
c) Quines son les teves 3 primeres línies del fitxer *temp*.txt? Creeu un nou fitxer *temp2.txt* que tingui el mateix contingut de *temp.txt* però on s'hagin eliminat les 3 línies anteriors.
d) Imprimiu a la sortida estàndard només les línies del fitxer *temp.txt* que contenen el patró "an".
e) Creu un nou fitxer *temp3.txt* amb la sortida de la comanda del apartat d). Després mostra per la sortida estàndard el fitxer però has d'afegir el patró "*** " davant de les línies que continguin el patró "man". ´
f) Llista el contingut del directori arrel tenint en compte que només has de mostrar les línies corresponents a enllaços simbòlics i afegint al final d'aquestes línies el text " < Això és un link".
g) Creeu un script anomenat *blancs.sh* digui quantes línies té un fitxer de text i que mostri línies que continguin espais blancs al final. El fitxer s'ha de passar com a argument del script. Considereu que heu de donar informació als usuaris sobre la funcionalitat del vostre script.

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

1.12. EL LLENGUATGE DE PROGRAMACIÓ GNU AWK

Gawk és la versió GNU del programa UNIX **awk**, i és un altre editor de flux (*stream editor*) popular. Atès que el programa awk sol ser un enllaç a gawk, ens referirem a ell com awk[13]. La funció bàsica de awk és cercar fitxers per a línies o altres unitats de text que continguin un o més patrons. Quan una línia coincideix amb un dels patrons, es realitzen accions especials en aquesta línia.

Els programes en awk són diferents dels programes de la majoria d'altres llenguatges, ja que els programes awk són "controlats per dades": descriviu les dades amb què voleu treballar i, a continuació, què fer quan les trobeu. La majoria dels altres llenguatges són "procedimentals". Heu de descriure, amb molt de detall, tots els passos que el programa ha de dur a terme. Quan es treballa amb llenguatges de procediment, sol ser molt més difícil de descriure clarament les dades que processarà el programa. Per aquest motiu, els programes awk sovint són molt fàcils de llegir i escriure.

Quan executeu awk, especifiqueu un programa awk que us demana què ha de fer. El programa consisteix en una sèrie de regles. També pot contenir definicions de funcions, bucles, condicions i altres construccions de programació, funcions avançades que actualment ignorarem. Cada regla especifica un patró per cercar i una acció a realitzar quan es troba el patró.

La comanda d'impressió (**`print`**) a awk dona la sortida de les dades seleccionades del fitxer d'entrada. Quan awk llegeix una línia d'un fitxer, divideix la línia en camps segons el separador de camp d'entrada especificat a la variable `FS` de awk. Aquesta variable per defecte està definida per ser un o més espais o tabulacions. Les variables `$1`, `$2`, `$3`, ..., `$N` contenen els valors del primer, el segon, el tercer fins a l'últim camp d'una línia d'entrada. La variable `$0` conté el valor de tota la línia. Això es mostra a la imatge següent, on veiem sis columnes a la sortida de l'ordre `df`:

```
rgarcia@ubuntu:~$ df -h
Filesystem             Size Used Avail Use% Mounted on
/dev/sda1              5.6G 3.2G 2.4G  57%  /
/dev/sda2              9.8G 6.1G 3.7G  62%  /home
192.168.1.1/dades      4.9G 4.1G 800M  84%  /mnt/nfs
rgarcia@ubuntu:~$ df -h | awk '{ print $2 $6 }'
5.6G/
9.8G/home
4.9G/mnt/nfs
```

Veieu que aquesta sortida no és molt llegible, no te format. Incloure alguns separadors i frases pot millorar molt com es llegeixen les dades:

```
rgarcia@ubuntu:~$ df -h | awk '{ print "El punt de muntatge " $6 "\t te una mida de: " $2 }'
El punt de muntatge /          te una mida de: 5.6G
El punt de muntatge /home      te una mida de: 9.8G
El punt de muntatge /mnt/nfs   te una mida de: 4.9G
```

[13] Als anys setanta, tres programadors es van unir per crear aquest llenguatge. Els seus noms eren Aho, Kernighan i Weinberger. Van prendre el primer caràcter de cadascun dels seus noms i els van unir. Així, doncs, el nom del llenguatge podria haver estat "wak.

A l'exemple anterior la utilització de `\t` indica que s'ha d'inserir una tabulació. Altres seqüències habituals són `\n` per inserir un salt de línia, o `\a` per inserir una alarma sonora.

Les cometes, els signes de dòlars i altres metacaràcters s'han d'escapar amb una barra invertida. Per exemple, si voleu buscar el patró "/dev/sd" a l'exemple anterior, per tal de no mostrar els dispositius muntats des de sistemes remots, caldrà escapar la `/` dins del nostre patró de cerca:

```
rgarcia@ubuntu:~$ df -h | awk '/dev\/sd/ { print "El punt de muntatge " $6 "\t te una mida de: " $2 }'
El punt de muntatge /           te una mida de: 5.6G
El punt de muntatge /home       te una mida de: 9.8G
```

Veieu un altre exemple on busquem al directori `/etc` els fitxers que acaben en ".conf" i comencen per "a" o "x", utilitzant expressions regulars esteses:

```
rgarcia@ubuntu:/etc$ ls -l | awk '/\<(a|x).*\.conf$/ { print $9 }
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf
```

Aquest exemple il·lustra el significat especial del punt en les expressions regulars: el primer indica que volem cercar qualsevol caràcter després de la primera cadena de cerca; el segon s'escapa perquè forma part d'una cadena per trobar (el "**.**conf").

Disposeu també d'alguns **patrons especials**. Per exemple, podeu precedir la sortida amb comentaris utilitzant la declaració `BEGIN`:

```
rgarcia@ubuntu:/etc$ ls -l | awk 'BEGIN { print "Fitxers trobats:\n" } /\<(a|x).*\.conf$/ { print $9 }'
Fitxers trobats:
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf
```

De manera semblant utilitzeu la declaració `END` per introduir un comentari al final de la sortida:

```
rgarcia@ubuntu:/etc$ ls -l | awk '/\<(a|x).*\.conf$/ { print $9 } END { print "Cerca finalitzada!" }'
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf
Cerca finalitzada!
```

Com veieu les comandes poden a començar a ser força llargues en la seva escriptura. A més, és possible que vulgueu que siguin reutilitzables i no tenir que escriure-les de nou un altre dia. Podeu, en aquests casos, utilitzar el llenguatge awk dins d'un **script awk** que contindrà les declaracions awk que defineixen patrons i accions. Per exemple, veieu el següent script que cerca les particions més carregades al sistema:

```
rgarcia@ubuntu:~$ cat diskfull.awk
BEGIN { print "*** WARNING WARNING WARNING ***" }
/\<[8|9][0-9]%/ { print "Partició " $6 "\t: " $5 " full!" }
END { print "*** Compra un nou disc URGENTMENT! ***" }

rgarcia@ubuntu:~$ df -h | awk -f diskfull.awk
*** WARNING WARNING WARNING ***
Partition /mnt/nfs  : 84% full!
*** Compra un nou disc URGENTMENT! ***
```

El llenguatge awk accepta **variables** a processar durant l'entrada i sortida. Ja hem comentat que la variable `FS` que s'utilitza com a separador de camps d'entrada (i que es diferent a la varibale *IFS* que ja havíeu vist anteriorment). El valor de la variable separador de camp es pot canviar al programa awk amb l'operador d'assignació =. Sovint, el moment adequat per fer-ho és al principi de l'execució abans de processar qualsevol entrada, de manera que el primer registre es llegeixi amb el separador adequat. Per fer-ho, utilitzeu la definició `BEGIN` que ja heu vist. A l'exemple següent, construïm una ordre que mostra tots els usuaris del vostre sistema amb una descripció:

```
rgarcia@ubuntu:~$ awk 'BEGIN { FS=":" } { print $1 "\t" $5 }' /etc/passwd
root    Usuari administrador
…
rgarcia Usuari suoder principal
…
```

O dins d'un script awk:

```
rgarcia@ubuntu:~$ cat usuaris.awk
BEGIN { FS=":" }
{ print $1 "\t" $5 }

rgarcia@ubuntu:~$ awk -f usuaris.awk /etc/passwd
root    Usuari administrador
…
rgarcia Usuari suoder principal
…
```

Pel que fa a la sortida, els camps normalment estan separats per espais. Això es fa evident quan utilitzeu la sintaxi correcta per a la comanda `print`, on els arguments estan separats per comes per tal de fer aquesta separació per espais per defecte i no es tractin cadascun dels camps com un de sol:

```
rgarcia@ubuntu:~$ cat test
record1         data1
record2         data2
rgarcia@ubuntu:~$ awk '{ print $1 $2 }' test
record1data1
record2data2

rgarcia@ubuntu:~$ awk '{ print $1, $2 }' test
record1 data1
record2 data2
```

Recordeu que teniu modificadors (`\t` o `\n`) per modificar aquest separador. També podeu canviar el separador de camps de sortida per defecte modificant la variable `OFS`. A més, podeu també modificar el separador entre les línies resultants d'un processament awk (normalment és `\n`, és a dir, un salt de línia) a través de la variable `ORS`: [14]

```
rgarcia@ubuntu:~$ awk 'BEGIN { OFS=";" ; ORS="\n-->\n" } { print $1;$2 }' test
record1;data1
-->
record2;data2
-->
```

Disposeu també de la variable `NR` que manté el nombre de registres que es processen. S'incrementa després de llegir una nova línia d'entrada. Es pot utilitzar al final per comptar el nombre total de registres o en cada registre de sortida:

```
rgarcia@ubuntu:~$ cat nr.awk
BEGIN { OFS="-" ; ORS="\n--> fet\n" }
{ print "Registre número " NR ":\t" $1,$2 }
END { print "Número de registres processats: " NR }

rgarcia@ubuntu:~$ awk -f nr.awk test
Registre número 1:      record1-data1
--> fet
Registre número 2:      record2-data2
--> fet
Número de registres processats: 2
--> fet
```

[14] Per a un control més precís del format de sortida que el que normalment es proporciona mitjançant la impressió, utilitzeu **`printf`**. L'ordre `printf` es pot utilitzar per especificar l'amplada del camp que s'utilitzarà per a cada element, així com diverses opcions de format per a números (com quina base de sortida s'utilitzarà, tant si s'ha d'imprimir un exponent, si cal imprimir un signe i quants dígits) per imprimir després del punt decimal). Això es fa subministrant una cadena, anomenada cadena de format, que controla com i on imprimir els altres arguments. La sintaxi és la mateixa que per a la declaració `printf` de llenguatge C.

A part de les variables incorporades, podeu **definir les vostres pròpies variables**. Quan awk troba una referència a una variable que no existeix (que no està predefinida), la variable es crea i s'inicialitza a una cadena nul·la. Per a totes les referències posteriors, el valor de la variable és quin sigui el valor assignat per última vegada. Les variables poden ser una cadena o un valor numèric. El contingut dels camps d'entrada també es pot assignar a variables. Els valors es poden assignar directament mitjançant l'operador = o podeu utilitzar el valor actual de la variable en combinació amb altres operadors:

```
rgarcia@ubuntu:~$ cat ingressos
20021009        20021013        consultoria     BigComp         2500
20021015        20021020        formació        EduComp         2000
20021112        20021123        appdev          SmartComp       10000
20021204        20021215        formació        EduComp         5000

rgarcia@ubuntu:~$ cat total.awk
{ total=total + $5 }[15]
{ print "Enviar factura de " $5 " euros a " $4 }
END { print "--------------------------------\nIngressos totals: " total }

rgarcia@ubuntu:~$ awk -f total.awk ingressos
Enviar factura de 2500 euros a BigCom
Enviar factura de 2000 euros a EduComp
Enviar factura de 10000 euros a SmartComp
Enviar factura de 5000 euros a EduComp
--------------------------------
Ingressos totals: 19500
```

ACT 7 | **Scripts AWK**

El següent exercici us ha de servir per repassar alguns dels conceptes que acabeu d'estudiar entorn al llenguatge awk:

a) Situat al teu directori *scripts* i crea un fitxer de text anomenat *alumnes* amb la següent estructura:
   `Username:Nom:Cognom:Telèfon`
   Que contingui la teva informació personal i la de un parell de companys teus.
b) Ara crea un script awk anomenat *ldapusers.awk* que converteixi cadascuna de les línies del teu fitxer *alumnes* al següent format:
   ```
   dn: uid=Username, dc=exemple, dc=com
   cn: Nom Cognom
   sn: Cognom
   telephoneNumber: Telèfon
   ```

[15] S'admet també la combinació reduïda `VAR+=valor`.

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

1.13. DECLARACIONS CONDICIONALS

De vegades, heu d'especificar **diferents cursos d'acció** que s'han de realitzar en un script depenent de l'èxit o del fracàs d'una ordre. La construcció **if** permet especificar aquestes condicions. La sintaxi més compacta de la comanda és: `if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi`.

S'executa la llista `TEST-COMMAND` i si el seu estat de retorn és zero, s'executa la llista `CONSEQUENT-COMMANDS`. L'estat de retorn és l'estat de sortida de l'última ordre executada o zero si no s'ha provat la veritat.

El `TEST-COMMAND` sovint implica proves de comparació numèriques o de cadenes, però també pot ser qualsevol ordre que retorni un estat de zero quan tingui èxit i un altre estat quan falla. Les expressions unàries sovint s'utilitzen per examinar l'estat d'un fitxer. Si l'argument `FILE` d'una de les primàries és de la forma `/dev/fd/N`, s'ha marcat el descriptor de fitxer "N". `stdin`, `stdout` i `stderr` i els seus respectius descriptors de fitxers també es poden utilitzar per a proves.

La taula següent conté una visió general de les anomenades "primàries" que formen el comandament `TEST-COMMAND` o la llista d'ordres. Aquestes primàries es col·loquen entre claudàtors (`[ ]`) per indicar la prova d'una expressió condicional. Veieu la següent taula d'**expressions primàries**:

Primària	Significat
`[-a FILE]`	True si existeix el fitxer
`[-b FILE]`	True si existeix el fitxer i és un fitxer especial de bloc[16]
`[-c FILE]`	True si existeix el fitxer i és un fitxer especial de caràcter
`[-d FILE]`	True si existeix el fitxer i és un directori
`[-e FILE]`	True si existeix el fitxer
`[-f FILE]`	True si existeix el fitxer i és un fitxer normal
`[-g FILE]`	True si existeix el fitxer i te definit el seu bit SGID[17]
`[-h FILE]`	True si existeix el fitxer i és un enllaç fort
`[-k FILE]`	True si existeix el fitxer i te definit el seu sticky bit
`[-p FILE]`	True si existeix el fitxer i és una pipe
`[-r FILE]`	True si existeix el fitxer i és llegible
`[-s FILE]`	True si existeix el fitxer i té una mida superior a zero
`[-t FD]`	True si el descriptor del fitxer FD està obert i fa referència a un terminal[18]
`[-u FILE]`	True si existeix el fitxer i te definit el seu bit SUID
`[-w FILE]`	True si existeix el fitxer i es pot escriure
`[-x FILE]`	True si existeix el fitxer i es pot executar

[16] Son fitxers que representen dispositius, concretament aquells que fan la transferència d'informació agrupada en blocs, per exemple, els dispositius d'emmagatzematge com `/dev/hda1`. En una sortida "llarga" del `ls` s'identifiquen amb la lletra b. Aquells que fan la transferència byte a byte són dispositius de caràcter, com per exemple, `/dev/tty1`, i a la sortida extensa els trobem amb la lletra c.

[17] Els bits especials `SUID` i `SGID` marquen que aquell usuari que executi un fitxer tindrà els mateixos permisos que té el propietari o grup, respectivament, que va crear el fitxer. A la sortida llarga del `ls` podem detectar si està activat veient que el permís d'execució del usuari o grup ve marcat per una `s` en comptes d'una `x`. Recordeu també l'existència del *sticky bit* que protegeix contra la eliminació de fitxers o directoris, fent que només el propietari (o root) ho puguin eliminar, tot i que alguns usuaris puguin tenir permís d'escriptura. S'identifica a la sortida extensa amb una `t` en comptes d'una `x` al bit d'execució dels permisos per a altres.

[18] Consulteu la següent informació sobre el descriptor de fitxer `FD` i la seva utilitat.

Primària	Significat
`[-O FILE]`	True si existeix el fitxer i és propietat de l'usuari efectiu
`[-G FILE]`	True si existeix el fitxer i és propietat del grup efectiu
`[-L FILE]`	True si existeix el fitxer i és un fitxer enllaç simbòlic
`[-N FILE]`	True si existeix el fitxer i s'ha modificat des de la seva última lectura
`[-S FILE]`	True si existeix el fitxer i és un socket[19]
`[FILE1 -nt FILE2]`	True si FILE1 s'ha modificat més recentment que FILE2 (o FILE2 no existeix)
`[FILE1 -ot FILE2]`	True si FILE1 és més antic que FILE2 (o no existeix FILE1 i si FILE2)
`[FILE1 -ef FILE2]`	True si FILE1 i FILE2 fan referència al mateix dispositiu i números d'inode
`[-o OPTIONNAME]`	True si l'opció de shell especificada està habilitada
`[-z STRING]`	True si la longitud de la cadena és zero
`[-n STRING]`o`[STRING]`	True si la longitud de la cadena no és zero
`[STRING1 == STRING2]`	True si les cadenes són iguals (es podria utilitzar simplement un =)
`[STRING1 != STRING2]`	True si les cadenes no són iguals
`[STRING1 < STRING2]`	True si la primera cadena s'ordena lexicogràficament abans que la segona
`[STRING1 > STRING2]`	True si la primera cadena s'ordena lexicogràficament després que la segona
`[ARG1 OP ARG2]` *ARG1 i ARG2 son números enters*	OP és una de les següents: `-eq`, `-ne`, `-lt`, `-le`, `-gt` o `-ge`. Aquest operadors aritmètics retornen True si ARG1 *és igual*, *diferent*, *menor que*, *menor o igual que*, *major que* o *major o igual que* ARG2, respectivament.

Les expressions anteriors es poden combinar utilitzant els següents operadors, i llistant les primàries en ordre decreixent de prioritat:

Operació	Efecte
`[! EXPR]`	True si la expressió és falsa
`[(EXPR)]`	Retorna el valor de l'expressió. Es pot utilitzar per sobreescriure la prioritat normal dels operadors.
`[EXPR1 -a EXPR2]`	True si les dues expressions són certes
`[EXPR1 -o EXPR2]`	True si una de les dues expressions és certa

La llista `CONSEQUENT-COMMANDS` que segueix la sentència `then` pot ser qualsevol comanda UNIX vàlida, qualsevol programa executable, qualsevol script de l'intèrpret d'ordres executable o qualsevol instrucció de shell, amb l'excepció del `fi` de tancament.

És important recordar que les consideracions `then` i `fi` es consideren declaracions separades al shell, i, per tant, quan s'emeten a la línia d'ordres, queden separades per un punt i coma.

En un script, les diferents parts de la sentència `if` solen estar ben separades. A continuació veurem alguns exemples senzills.

[19] Els *sockets* son un tipus de fitxers que fan una representació abstracta d'un punt de comunicació, permeten establir un canal de comunicació entre dues rutines o programes, normalment orientats a la comunicació d'entrada/sortida dels serveis de xarxa TCP/UDP. S'identifiquen a la sortida llarga de `ls` amb la lletra s.

El primer exemple testeja l'existència d'un fitxer:

```
rgarcia@ubuntu:~$ cat msgcheck.sh
#!/bin/bash

echo "Aquest script testeja l'existència del fitxer messages als logs del equip."
echo "Checking..."
if [ -f /var/log/messages ]
  then
    echo "/var/log/messages existeix."
fi
echo

rgarcia@ubuntu:~$ ./msgcheck.sh
Aquest script testeja l'existència del fitxer messages als logs del equip.
Checking...
/var/log/messages existeix.
```

Sovint volem testejar si alguna opció del shell està activa, per fer-ho podeu utilitzar algunes línies semblants a les del proper exemple dintre dels nostres scripts:

```
# Aquestes línies mostren un missatge si l'opció noclobber está activada:

if [ -o noclobber ]
  then
       echo "Els teus fitxers estan protegits contra la sobreescriptura
accidental a les redireccions."
fi
```

També podríeu revisar aquesta condició des de la pròpia línia de comandes, observeu l'ús dels "**;**":

```
rgarcia@ubuntu:~$ if [ -o noclobber ] ; then echo ; echo "Els teus fitxers están
protegits contra la sobreescriptura accidental a les redireccions." ; echo ; fi

Els teus fitxers están protegits contra la sobreescriptura accidental a les
redireccions.
```

Un altra utilització freqüent és la consulta de la variable **?** que manté l'estat de sortida de l'ordre executat anteriorment (el procés de primer pla completat més recentment). El següent exemple mostra una prova senzilla:

```
rgarcia@ubuntu:~$ if [ $? -eq 0 ]
> then echo "TOT OK!"
> fi
TOT OK!
```

Nota: Observeu que podem dividir l'expressió condicional en varies línies del terminal.

Autor | RAFAEL GARCÍA SUÁREZ
Contacte | rgarc259@xtec.cat

L'exemple següent demostra que `TEST-COMMANDS` pot ser qualsevol comanda UNIX que retorni un estat de sortida i si torna a retornar un estat de sortida de zero:

```
uremot@ubuntu:~$ if ! grep $USER /etc/passwd
> then echo "Aquest no és un usuari local"; fi
Aquest no és un usuari local

uremot@ubuntu:~$ echo $?
0
```

Obtindríeu el mateix resultat de la següent manera:

```
uremot@ubuntu:~$ grep $USER /etc/passwd

uremot@ubuntu:~$ echo if [ $? -eq 0 ]; then echo "Aquest no és un usuari local"; fi
Aquest no és un usuari local
```

El següent exemple mostra comparacions numèriques:

```
rgarcia@ubuntu:~$ num='wc -l work.txt'
rgarcia@ubuntu:~$ echo $num
303
rgarcia@ubuntu:~$ if [ "$num" -gt "100" ]
> then echo "estàs treballant fort avui!"
> echo ; fi
estàs treballant fort avui!
```

Observeu el següent script que executarem a través de `cron` cada diumenge a les 21h:

```
#!/bin/bash

# Calcular el número de la setmana i fer la operación mòdul 2

WEEKNUM=$[ $(date +"%V") % 2 ]

# Comprovar si és una semana parell

if [ $WEEKNUM -eq "0" ]; then
  echo "Diumenge nit, treu la brossa de paper a reciclar."
fi
```

Si consulteu el manual de la comanda `date` veureu que la opció `+%V` ens retorna el número de la setmana dins de l'any. Apliquem l'operació mòdul 2 que si retorna 0 vol dir que no te residu, i per tant, és una setmana parell. El script comprova aquesta circumstancia per tal d'enviar, o no, un missatge recordatori.

Veieu ara un exemple d'una comprovació d'identitat i permisos que podríem utilitzar als nostres scripts. En aquesta comprovació utilitzareu la comparació de cadenes (*strings*):

```
if [ "$(whoami)" != 'root' ]; then
        echo "No tens permisos per executar $0"
        exit 1;
fi
```

Amb Bash, podeu escurçar aquest tipus de construcció. Aquest seria l'equivalent compacte de la comprovació anterior:

```
[ "$(whoami)" != 'root' ] && ( echo No tens permisos per executar; exit 1 )
```

S'utilitza `&&` per indicar si la prova és certa. Podríeu utilitzar `||` per comprovar si la prova és falsa. Fins i tot, molt programadors prefereixen utilitzar la comanda `test` en comptes dels claudàtors per fer la comparació:

```
test "$(whoami)" != 'root'  && ( echo No tens permisos per executar; exit 1 )
```

El següent exemple mostra l'ús d'expressions regulars dins d'una comparació:

```
rgarcia@ubuntu:~$ genere="dona"

rgarcia@ubuntu:~$ if [[ $genere == d* ]]
> then echo "Un plaer coneixer-la, senyora"; fi
Un plaer coneixer-la, senyora
```

Nota: El doble claudàtor `[[ ]]` evita que s'executin expansions de text no desitjades dins de la comparació.

Podem ampliar la capacitat de l'expressió `if` amb l'ús de la declaració **`else`**, que oferirà una acció alternativa si no és compleix la condició. Seguint amb l'exemple anterior:

```
rgarcia@ubuntu:~$ genere="home"

rgarcia@ubuntu:~$ if [[ $genere == d* ]]
> then echo "Un plaer coneixer-la, senyora"
> else echo "Què passa noi!"
> fi
Què passa noi!
```

Igual que la llista `CONSEQUENT-COMMANDS` després de la declaració `then`, la llista `ALTERNATE-CONSEQUENT-COMMANDS` després de la sentència `else` pot contenir qualsevol ordre UNIX.

Recordeu que en lloc de configurar una variable i executar un script, sovint és més elegant posar els valors de les variables a la línia d'ordres. Utilitzeu els paràmetres de posició `$1`, `$2`, ..., `$N` per a aquest propòsit. `$#` fa referència al nombre d'arguments de línia d'ordres i, `$0` fa referència al nom del script. El següent és un exemple senzill:

```
rgarcia@ubuntu:~$ cat pingüi.sh
#!/bin/bash

#Aquest script comprova quins aliments volem donar a Tux, ell només vol peix

if [ "$1" == peix ]; then
   echo "Hmmmmm peix… Tux està content!"
else
   echo "Tux no vol $1, Tux vol peix!"
fi

rgarcia@ubuntu:~$ pingüi.sh poma
Tux no vol poma, Tux vol peix!
rgarcia@ubuntu:~$ pingüi.sh peix
Hmmmmm peix… Tux està content!
```

ACT 8 | **Arguments d'entrada: posició i número**

Volem construir un script que ens permetrà recomanar a un usuari si ha de menjar més o menys en funció del seu pes i alçada que haurà d'introduir com a arguments al script.

a) Situat al teu directori *scripts* i crea un nou script anomenat *pesideal.sh* que ha de comprovar si, segons els arguments passats (primer el pes en quilograms i després l'alçada en centímetres), aquest està per sobre o per sota del seu pes ideal (considereu que el pes ideal es igual a l'alçada, en cm, menys 110). S'han de llançar missatges informatius sobre si ha de menjar més greix, o més verdura, en funció de si està per sota o per sobre del seu pes ideal, respectivament.
b) Millora el script *pesideal.sh* fent que abans de començar comprovi que efectivament s'han passat els dos arguments (pes i alçada) quan es crida el script. Si no és així s'ha d'informar de com utilitzar el script *pesideal.sh* a l'usuari i sortir.

Podeu completar les condicions `if – else`, incorporant una tercera part **`elif`**. Aquesta seria la estructura completa de la declaració `if`:

```
if TEST-COMMANDS; then
CONSEQUENT-COMMANDS;
elif MORE-TEST-COMMANDS; then
MORE-CONSEQUENT-COMMANDS;
else ALTERNATE-CONSEQUENT-COMMANDS;
fi
```

Modificarem el script d'exemple *pingüi.sh* al que ara anomenarem per aportar-li una nova situació:[20]

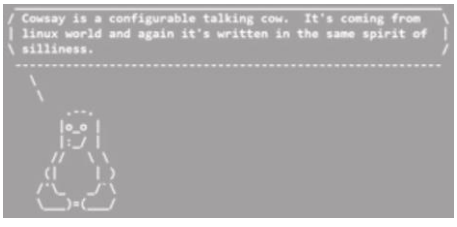
```
rgarcia@ubuntu:~$ cat pingüi.sh
#!/bin/bash

#Aquest script comprova quins aliments volem donar a Tux, ell només vol peix

if [ "$1" == peix ]; then
   echo "Hmmmmm peix… Tux està content!"
elif [ "$1" == pingüi ]; then
   echo "Tux no és un canival, estàs tronat!"
else
   echo "Tux no vol $1, Tux vol peix!"
fi

rgarcia@ubuntu:~$ pingüi.sh poma
Tux no vol poma, Tux vol peix!
rgarcia@ubuntu:~$ pingüi.sh peix
Hmmmmm peix… Tux està content!
rgarcia@ubuntu:~$ pingüi.sh pingüi
Tux no és un canival, estàs tronat!
```

També tenim altres opcions per fer que les nostres condicions if valorin múltiples opcions mitjançant l'ús d'**operadors booleans** dins de la expressió condicional. Aquests operadors poden "sumar" la condició verdadera de dues expressions per tal de fer `true` la condició (operador `&&`), o poden establir a `true` la condició de `if` si una expressió o una altra són certes (operador `||`). Per exemple:

```
rgarcia@ubuntu:~$ cat colors.sh
#!/bin/bash

#Aquest script determina si un color del espectre lluminòs és primari o no

if (( "$1" == blau )) || (( "$1" == verd )) || (( "$1" == vermell" )); then
   echo "$1 és un color primari de l'espectre de llum."
else
   echo "$1 no és un color primari de l'espectre de llum."
fi

rgarcia@ubuntu:~$ color.sh vermell
Vermell és un color primari de l'espectre de llum.
rgarcia@ubuntu:~$ pingüi.sh groc
Groc no és un color primari de l'espectre de llum.
```

[20] Si vols fer-ho més divertit prova d'instal·lar el paquet *cowsay* (`sudo apt install cowsay`).

Un cop instal·lat llença els missatges de sortida del script *pingüi.sh* amb la comanda `cowsay -f tux` *`missatge`*.



Podeu pensar en imbricar diferents declaracions condicionals de `if`, que al seu cop poden contenir condicionals `elif`, per contemplar més opcions d'accions a realitzar segons quina sigui la situació, que a la vegada poden contemplar diferents opcions mitjançant operadors booleans. Tot i això, aquest fet pot acabar provocant que el vostre codi condicional als scripts es vagi complicant i fent difícil la seva comprensió i seguiment.

Per aquest motiu, per fer condicionals més complexos, és recomanable utilitzar l'expressió condicional anomenada **`case`**, que té la següent sintaxis:

```
case EXPRESSION in CASE1) COMMAND-LIST;; CASE2) COMMAND-LIST;; ... CASEN) COMMAND-LIST;; esac
```

Cada cas és una expressió que coincideix amb un patró. S'executen les ordres del `COMMAND-LIST` del primer cas on és trobi coincidència. Es pot utilitzar el símbol `|` per separar múltiples patrons i hem d'utilitzar el símbol `)` per acabar la llista de patrons. Anomenarem **clàusula** a cada cas més les seves ordres. Cada clàusula ha de finalitzar amb `;;`. Cada declaració de `case` s'acaba amb la sentència `esac`.

Per exemple, veieu l'ús del condicional `case` en un script que realitzarà operacions aritmètiques. El script *calcula.sh* rebrà com arguments quina operació hem de realitzar seguida de amb quins dos valors treballarem. Com veureu és habitual establir un `case*)` com a últim cas possible que contempli quina acció o accions realitzar en cas de no haver trobar cap coincidència en els patrons dels casos anteriors:

```
rgarcia@ubuntu:~$ cat calcula.sh
#!/bin/bash

#Aquest script realitza operacions arimètiques amb 2 números

case $1 in
suma)
  echo "$2 + $3 = $(($2+$3))"
  ;;
resta)
  echo "$2 - $3 = $(($2-$3))"
  ;;
multiplica)
  echo "$2 x $3 = $(($2*$3))"
  ;;
divideix)
  echo "$2 / $3 = $(($2/$3))"
  ;;
*)
  echo "Operació no vàlida. Escull suma, resta, multiplica o divideix"
  ;;
esac

rgarcia@ubuntu:~$ calcula.sh suma 42 21
42 + 21 = 63
rgarcia@ubuntu:~$ calcula.sh divide 36 6
Operació no vàlida. Escull suma, resta, multiplica o divideix
rgarcia@ubuntu:~$ calcula.sh divideix 36 6
36 / 6 = 6
```

Un cas habitual d'ús de l'expressió condicional `case`, és la de poder establir diferents valors de sortida dins l'execució d'un script amb la comanda `exit`. Aquesta comanda, a part de terminar amb l'execució del vostre script, retorna al pare un codi de sortida emmagatzemat a la variable `$?`, i que per defecte, tindrà valor zero. Però aquest valor pot ser definit per l'usuari i utilitzat posteriorment. Veieu la següent modificació del script anterior, i com podem utilitzar-lo des d'un nou script utilitzant els diferents codis especificats per a la comanda `exit`:

```
#modifiquem calcula.sh dins l'opció resta per generar estats de sortida diferents

...
resta)
  echo "$2 - $3 = $(($2-$3))"
  if [ "$2" < "$3" ]; then
    exit 1
  elif [ "$2" == "$3" ]; then
    exit 2
  else
    exit 3
  fi
  ;;
...

rgarcia@ubuntu:~$ cat compara.sh
#!/bin/bash

#Aquest script et diu si un número és major, menor o igual que un altre

export num1="$1"
export num2="$2"
crida="/home/rgarcia/calcula.sh"
$crida resta $num1 $num2

case $? In
1)
  echo "$num1 és menor que $num2"
  ;;
2)
  echo "$num1 és igual que $num2"
  ;;
3)
  echo "$num1 és major que $num2"
*)
  echo "Alguna cosa no ha anat bé"
  ;;
esac
```

Nota: fixeu-vos que exportem el valor dels arguments d'entrada per que estiguin disponibles al subshell que és generarà al cridar al script *calcula.sh* des del nostre nou script *compra.sh*.

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

ACT 9 | **Declaracions condicionals**

Ara que ja coneixem les diferents possibilitats alhora de treballar amb declaracions condicionals, realitzeu els següents exercicis revisant els conceptes que acabeu d'estudiar:

a) Crea dins del teu directori *scripts* un script anomenat *dom.sh* que utilitzant construccions `if - then - elif - else`, digui quants dies te el mes actual. A més ha de mostrar informació sobre si és un any de traspàs en cas de que el mes actual sigui febrer.
b) Crea un nou script anomenat *dom2.sh*, que faci el mateix que *dom.sh* però utilitzant ara la construcció `case` i una altra alternativa de la comanda `date`.
c) Modifiqueu el fitxer */etc/profile* per rebre un missatge especial quan ús connecteu com a root.
d) Modifiqueu el script d'exemple *colors.sh* tal de que rebi dos arguments: el color (que ja rebia a l'exemple) i l'espectre. A l'exemple és consideraven els colors primaris de l'espectre de llum, ara a través de l'argument espectre podrem especificar llum o pigment. En el cas d'especificar pigment recordeu que els colors primaris seran cian, magenta i groc.
e) Fes un script anomenat *filetype.sh* que determini diferents sortides per la seva execució segons si un fitxer que rep com argument és un fitxer normal, un directori, un dispositiu de bloc, un dispositiu de caràcter, un socket, una pipe, un enllaç fort o un enllaç simbòlic. Després genera un script anomenat *existeix.sh* que comprovi si s'ha passat un fitxer com argument, cas que comportarà comprovar si existeix al teu directori personal. En cas afirmatiu, cal que cridi al script *filetype.sh* per gestionar missatges de sortida indicant la existència del fitxer i el tipus de fitxer del que es tracta.

1.14. SCRIPTS INTERACTIUS

Alguns scripts funcionen sense cap tipus d'interacció de l'usuari. Els avantatges d'escriptures no interactives són:

- L'escriptura s'executa de manera predicible cada vegada.
- L'escriptura es pot executar en segon pla.

Tanmateix, molts scripts requereixen una entrada de l'usuari o donen sortida a l'usuari a mesura que s'executa l'escriptura. Els avantatges d'aquests **scripts interactius** són, entre d'altres:

- Es poden construir scripts més flexibles.
- Els usuaris poden personalitzar l'escriptura mentre s'executa o que es comporti de diferents maneres.
- L'escriptura pot informar del seu progrés mentre s'executa.

Quan escriviu scripts interactius, no deixeu mai els comentaris. Un script que imprimeix els missatges adequats és molt més fàcil d'usar i es pot depurar amb més facilitat. Un script pot fer un treball perfecte, però obtindreu moltes trucades de suport si no us informa del que està fent. Quan se sol·licita l'entrada d'usuari, també és millor donar informació de més que no de menys de com a de ser aquesta entrada, tot i que en tots els casos serà recomanable fer una comprovació dels arguments rebuts al nostre script com entrada d'usuari abans d'executar res amb aquell valor. Bash té els comandaments de `echo` i `printf` per proporcionar comentaris als usuaris, i ja hauríeu d'estar familiaritzat amb, com a mínim l'ús de `echo`.

Al mateix temps que `echo` i `printf` són comandes d'escriptura a la sortida estàndard (per defecte), disposeu d'una comanda de lectura de l'entrada estàndard. La comanda integrada **`read`** és la contrapart de les ordres de `echo` i `printf`. La seva sintaxi és la següent: `read [opcions] NAME1 NAME2 ... NAMEN`.

Es llegeix una línia de l'entrada estàndard, o des del descriptor de fitxer subministrat com a argument a l'opció `-u`. La primera paraula de la línia s'assigna al nom, `NAME1`, la segona paraula al segon nom, etc., amb paraules restants i els seus separadors intermedis assignats al cognom `NAMEN`. Si hi ha menys paraules llegides des del flux d'entrada que noms hi hagi, els noms restants se'ls assigna valors buits. Un exemple molt senzill:

```
rgarcia@ubuntu:~$ cat firstread.sh
#!/bin/bash

echo -n "Introdueix el teu nom i prem [ENTER]: "
read nom
echo "Hola $nom, benvingut"

rgarcia@ubuntu:~$ firstread.sh
Introdueix el teu nom i prem [ENTER]: Rafa      → el terminal espera l'entrada per part de l'usuari
Hola Rafa, benvingut
```

Els caràcters del valor de la variable `IFS` s'utilitzen per dividir la línia d'entrada en paraules o fitxes. El caràcter de barra invertida es pot utilitzar per eliminar qualsevol significat especial per al següent caràcter de lectura i per a la continuació de línia.

Si no s'ofereix cap nom, la línia llegida s'assigna a la variable `REPLY`. El codi de retorn de l'ordre `read` és zero, tret que s'aconsegueixi un caràcter de final de fitxer, si s'ha llegit un temps de sortida o si s'ofereix un descriptor de fitxer invàlid com a argument a l'opció `-u`. Veieu la següent taula d'opcions compatibles amb l'ordre `read`: Bash:

Opció	Significat
`-a ANAME`	Les paraules s'assignen a índexs seqüencials de la variable matriu `ANAME`, començant per 0. Tots els elements s'eliminen de `ANAME` abans de l'assignació.
`-d DELIM`	El primer caràcter de `DELIM` s'utilitza per acabar la línia d'entrada, en lloc de la nova línia.
`-e`	La línia de lectura es utilitzada per obtenir la línia.
`-n NCHARS`	Llegeix fins a completar la lectura de tants caràcters com indica NCHARS en lloc d'esperar una línia completa d'entrada.
`-p PROMPT`	Mostra el PROMPT, sense una nova línia final, abans d'intentar llegir qualsevol entrada. L'indicador només es mostra si l'entrada prové d'un terminal.
`-r`	La barra invertida no actua com a caràcter d'escapament. Es considera que la barra invertida forma part de la línia. En particular, un parell barra [invertida-línia nova] no es pot utilitzar com a continuació de línia.
`-s`	Mode silenciós. Si l'entrada prové d'un terminal, no es fa `echo` dels caràcters.
`-t TIMEOUT`	Es llegeix si l'entrada es fa abans del temps en segons determinat per `TIMEOUT`, en cas contrari, provoca una fallada. Només funciona si es llegeix d'un terminal o una canonada.
`-u FD`	Llegeix l'entrada des del descriptor de fitxer `FD`.[18]

Al següent exemple s'utilitzen algunes opcions de la comanda `read`:

```
rgarcia@ubuntu:~$ cat navegador.sh
#!/bin/bash
#Aquest script recull el navegador favorit de l'usuari
FAVORITS="/nav/favorits"
echo "Hola, $USER. Aquest script recollirà el teu navegador favorit"
echo
echo "Aquests són els navegadors disponibles:"
cat << BROWSERS
mozilla
chrome
BROWSERS
echo
echo -n "Quin és el teu favorit?"
read navfav
echo "$USER $navfav" >> "$FAVORITS"
echo "S'ha registrat $navfav" com el teu navegador per defecte."
echo

rgarcia@ubuntu:~$ navegador.sh
Hola, rgarcia. Aquest script recollirà el teu navegador favorit"

Aquests son els navegadors disponibles:"
firefox
chrome

Quin és el teu favorit?
mozilla
S'ha registrat firefox com el teu navegador per defecte.

rgarcia@ubuntu:~$ cat /nav/favorits
rgarcia firefox

rgarcia@ubuntu:~$ cat navegar.sh
#!/bin/bash
#Aquest script obre el navegador per defecte de l'usuari
nav=$(grep "$USER" /nav/favorits | awk '{ print $2 }')
echo "Confirmeu que voleu iniciar $nav [s/n]: "
read -t 10 -n 1 yesno
if [ "$yesno" == "s" ]; then
   echo
   echo "Iniciatnt $nav, un moment siuplau..."
   $nav
else
   echo
   echo "Operació cancel·lada"
fi

rgarcia@ubuntu:~$ navegar.sh
Confirmeu que voleu iniciar mozilla [s/n]: s
Inicialitzant mozilla, un momento siuplau...

-- S'inicia el navegador Mozilla Firefox --
```

Al script *navegador.sh* s'utilitza una construcció `cat << BROWSERS` seguida de la llista i finalitzada en `BROWSERS`, aquestes construccions es coneixen com documents *here* i són útils per fer llistes sense tenir que escriure molts `echo` seguits.

Al script *navegar.*sh, noteu que si l'usuari no respon en 10 segons a la pregunta de confirmació formulada pel script, el programa acabarà, de la mateixa manera, això succeirà si és contesta de forma diferent a s (només és llegirà un caràcter).

ACT 10 | **Scripts interactius**
Després d'haver estudiat l'anterior exemple, realitza la següent activitat:

a) Crea dins del teu directori *scripts* un script anomenat *compressio.sh* que comprimirà un directori amb un compressor escollit per l'usuari. Has de tenir en compte les següents consideracions:

- Has de comprovar que *compressio.sh* rep un directori com a paràmetre. Sinó has d'informar a l'usuari i sortir.
- Has de presentar una llista (utilitzant *here document*) indicant les opcions de compressió: `gzip`, `bzip2`, `compress` i `zip`. L'opció s'escollirà amb un sol caràcter (`g`, `b`, `c` i `z`, respectivament).
- La gestió de l'opció escollida l'has de fer amb una construcció del tipus `case`.

b) Com modificaries el script per tal de demanar a l'usuari una ruta de destí pel fitxer comprimit que es generarà? Pensa en demanar la ruta, i si no s'introdueix cap en 15 segons, utilitzi la ruta actual.

1.15. BUCLES

El **bucle for** és la primera de les tres construccions de looping de shell. Aquest bucle permet especificar una llista de valors. S'executa una llista d'ordres per a cada valor de la llista. La sintaxi d'aquest bucle és: `for NAME [in LIST ]; do COMMANDS; done`.[21]

L'estat de retorn és l'estat de sortida de l'última ordre que s'executa. Si no s'executen comandes perquè `LIST` no s'expandeix a cap element, l'estat de retorn és zero.

`NAME` pot ser qualsevol nom de variable, tot i que `i` s'utilitza molt sovint com a variable. `LIST` pot ser qualsevol llista de paraules, cadenes o números, que pot ser literal o generada per qualsevol comanda. Els `COMMANDS` per executar també pot ser qualsevol ordre del sistema operatiu, script, programa o instrucció de shell. La primera vegada que entrem al bucle, `NAME` s'estableix al primer element de `LIST`. La segona vegada, el seu valor s'estableix al segon element de la llista, etc. El bucle finalitza quan `NAME` ha assumit cadascun dels valors de `LIST`.

[21] Si `[in LIST]` no està present, els `COMMANDS` s'executen una vegada per a cada paràmetre posicional establert (`$1`, `$2`, ... `$N`).

Veieu un primer exemple d'ús del bucle for, des de la mateixa línia d'ordres, que realitza una copia de seguretat de tots els fitxers .xml d'un directori:

```
rgarcia@ubuntu:~/articles$ ls *.xml
file1.xml  file2.xml  file3.xml

rgarcia@ubuntu:~/articles$ ls *.xml > list

rgarcia@ubuntu:~/articles$ for i in `cat list`; do cp "$i" "$i".bak ; done

rgarcia@ubuntu:~/articles$ ls *.xml*
file1.xml  file1.xml.bak  file2.xml  file2.xml.bak  file3.xml  file3.xml.bak
```

La construcció **while** permet l'execució repetitiva d'una llista d'ordres, sempre que l'ordre que controli el bucle s'executa amb èxit (estat de sortida zero). La sintaxi és: `while CONTROL-COMMAND; do CONSEQUENT-COMMANDS; done`.

L'estat de retorn és l'estat de sortida de l'última ordre `CONSEQUENT-COMMANDS` o zero si no s'ha executat cap.

El `CONTROL-COMMAND` pot ser qualsevol ordre (o ordres) que pugui sortir amb un estat d'èxit o de fallada. Els `CONSEQUENT-COMMANDS` poden ser qualsevol programa, script o comanda del shell. Tan aviat com el `CONTROL-COMMAND` falla, el bucle surt (en un script, s'executa l'ordre que segueix la sentència `done`).

Aquest primer exemple mostra un script que obre 4 editors de text gràfics de manera seguida:

```
rgarcia@ubuntu:~$ cat 4gedit.sh
#!/bin/bash

# Aquest script obre 4 editors de text gedit.

i="0"

while [ $i -lt 4 ]
do
gedit &
i=$[$i+1]
done
```

Molts cops s'utilitza la condició `while true` per fer bucles que s'executen de manera constant ja que sempre són certs. Aquests molts cops són scripts que executen funcions al sistema en segon pla de manera continua (o fins que es força la seva detenció amb `kill`). Tot i que també es solen trobar quan volem que una sèrie d'ordres es vagin executant fins que l'usuari decideixi finalitzar el bucle, amb `Ctrl+C`, o donant-li una sortida més elegant utilitzant la instrucció `break`.

Veieu el següent exemple que calcula la mitja de les notes d'un alumne:

```
rgarcia@ubuntu:~$ cat mitja.sh
#!/bin/bash

# Calcula la mitja d'una serie de notes.

VALOR="0"
MITJA="0"
SUM="0"
NUM="0"

while true; do

  echo -n "Escriu la teva nota [0-10] ('q' for quit): "; read VALOR;

  if (("$VALOR" < "0"))  || (("$VALOR" > "10")); then
    echo "Nota no vàlida, proba de nou: "
  elif [ "$VALOR" == "q" ]; then
    echo "La teva mitja és: $MITJA"
    break
  else
    SUM=$[$SUM + $VALOR]
    NUM=$[$NUM + 1]
    MITJA=$[$SUM / $NUM]
  fi

done

echo "Sortint..."
```

El bucle **until** és molt similar al bucle while, excepte que el bucle s'executa fins que el `TEST-COMMAND` s'executa amb èxit. Mentre aquesta ordre falla, el bucle continua. La sintaxi és la mateixa que per al bucle while: `until TEST-COMMAND; do CONSEQUENT-COMMANDS; done`.

L'estat de retorn és l'estat de sortida de l'última ordre executada a la llista `CONSEQUENT-COMMANDS` o zero si no s'ha executat cap. `TEST-COMMAND` pot ser, de nou, qualsevol ordre que pugui sortir amb un estat d'èxit o de fallada, i `CONSEQUENT-COMMANDS` pot ser qualsevol comanda, script o comanda del shell.

```
rgarcia@ubuntu:~$ cat llistacompra.sh
#!/bin/bash

# Aquest script crea una llista de la compra

PARAULA="inici"
DATE=`date +%d%m`

echo "Escriu la teva llista de la compra (escriu final per sortir):"

until [ "$PARULA" == "final" ]; do
   read PARAULA
   $PARAULA >> llista_"$DATE"
done
echo "La teva llista d'avui s'ha generat a l'ubicació actual."
```


Autor | RAFAEL GARCÍA SUÁREZ
Contacte | rgarc259@xtec.cat

Ja hem parlat de la instrucció `break`, que s'utilitza per sortir del bucle actual abans del final normal. Recordeu que `break` surt d'un bucle, no d'un script, i per tant, en bucles imbricats permet especificar de quin bucle s'ha de sortir.

Per altra banda, disposeu de la instrucció `continue`, que reprèn una nova iteració d'un bucle for, while, until o select. Quan s'utilitza en un bucle for, la variable de control assumeix el valor de l'element següent de la llista. Quan s'utilitza en while o until, per contra, l'execució es reprèn amb `TEST-COMMAND` a la part superior del bucle.

A l'exemple següent, els noms de fitxers es converteixen en minúscules. Si no cal fer cap conversió, una instrucció `continue` reinicia l'execució del bucle. Aquestes ordres no consumeixen molts recursos del sistema i, probablement, es poden resoldre problemes similars utilitzant `sed` i `awk`. No obstant això, és útil conèixer aquest tipus de construcció:

```
rgarcia@ubuntu:~/test$ ls
test Test2 TEST3
rgarcia@ubuntu:~/test$ cat uppertolower.sh
#!/bin/bash

# Aquest script canvia el nom dels fitxers que conten majúscules a tot minúscules

LIST="$(ls)"

for name in "$LIST"; do

if [[ "$name" != *[[:upper:]]* ]]; then
continue
fi

ORIG="$name"
NEW=`echo $name | tr 'A-Z' 'a-z'`[22]

mv "$ORIG" "$NEW"
echo "El nou nom per $ORIG és $NEW"
done

rgarcia@ubuntu:~/test$ ./uppertolower.sh
El nou nom per Test2 és test2
El nou nom per TEST3 és test3
rgarcia@ubuntu:~/test$ ls
test test2 test3
```

Com veieu, el primer valor que prendrà la variable `name` serà el fitxer "test" i com que no conté majúscules no farem res; i amb `continue`, farem que for actualitzi la variable `name` al següent element de la llista "Test2". Penseu que aquest script pot tenir el desavantatge de sobreescriure fitxers, si per exemple, a la llista de fitxers dels directoris existís un fitxer anomenat "Test" i un altre "test", o "TEST".

[22] La comanda `tr` forma part del paquet *textutils* i pot realitzar tot tipus de transformacions de caràcters (consulteu `mant tr`).

Autor | RAFAEL GARCÍA SUÁREZ
Contacte | rgarc259@xtec.cat

Durant l'explicació anterior heu vist que existeix un tipus de bucle anomenat **select**. Aquesta construcció permet generar fàcilment un menú d'opcions. La sintaxi de select és bastant similar a la del bucle for: `select WORD [in LIST]; do RESPECTIVE-COMMANDS; done`.

`LIST` s'amplia i genera una llista d'elements. L'expansió s'imprimeix a la sortida estàndard; cada element està precedit per un número. Si a `LIST` no hi elements presents, s'imprimeixen els paràmetres de posició (`$1`, `$2`, ... `$N`). `LIST` només s'imprimeix una vegada.

En imprimir tots els elements, s'imprimeix l'indicador del prompt (anomenat `PS3`) i es llegeix una línia de l'entrada estàndard. Si aquesta línia consta d'un número corresponent a un dels elements, el valor de `WORD` s'estableix al nom d'aquest element. Si la línia està buida, es mostraran de nou els elements i l'indicador `PS3`. Si es llegeix un caràcter `EOF` (End Of File), el bucle surt. Atès que la majoria d'usuaris no tenen cap idea de quina combinació de tecles s'utilitza per a la seqüència d'`EOF`, és més fàcil d'usar un ordre de pausa/sortida com un dels elements. Qualsevol altre valor de la línia de lectura establirà `WORD` com una cadena nul·la.

La línia de lectura es desa a la variable `REPLY`. Els `RESPECTORS-COMANDES` s'executen després de cada selecció fins que es llegeix el número que representa l'opció on introduïm la sortida `break`, que surt del bucle.

Veieu el següent exemple senzill:

```
rgarcia@ubuntu:~/test$ cat privat.sh
#!/bin/bash

echo "Aquest script permet donar accés només per a tu als fitxers del directori."
echo "Escull el número del fitxer que vols protegir:"

select FILENAME in *;
do
     echo "Has escollit $FILENAME ($REPLY), ara només tu tens accés."
     chmod go-rwx "$FILENAME"
done

rgarcia@ubuntu:~/test$ ./privat.sh
Aquest script permet donar accés només per a tu als fitxers del directori.
Escull el número del fitxer que vols protegir:
1) test
2) test2
3) test3
#? 2
Has escollit test2 (1), ara només tu tens accés.
#?
```

Com veieu al protegir un fitxer el prompt `PS3` torna aparèixer i només podríem sortir del bucle select utilitzant `Ctrl+C`. A més, PS3 per defecte és bastant "pobre" (`#?`). Veieu com podem millorar el script *privat.sh* afegint una opció de sortida, i millorant l'aspecte de `PS3`:

```
rgarcia@ubuntu:~/test$ cat privat2.sh
#!/bin/bash

echo "Aquest script permet donar accés només per a tu als fitxers del directori."
echo "Escull el número del fitxer que vols protegir:"

PS3="La teva elecció: "
QUIT="Sortir del programa"
touch "$QUIT"

select FILENAME in *;
do
  case $FILENAME in
        "$QUIT")
          echo "Sortint..."
          break
          ;;
        *)
          echo "Has escollit $FILENAME ($REPLY), ara només tu tens accés."
          chmod go-rwx "$FILENAME"
          ;;
  esac
done
rm "$QUIT"   #no oblideu eliminar el "fitxer temporal" que heu creat!

rgarcia@ubuntu:~/test$ ./privat2.sh
Aquest script permet donar accés només per a tu als fitxers del directori.
Escull el número del fitxer que vols protegir:
1) test
2) test2
3) test3
4) Sortir del programa
La teva elecció: 4
Sortint...
```

Qualsevol opció dins d'una construcció select pot ser un altre bucle select, habilitant la possibilitat de crear submenús dins d'un menú. Per defecte, la variable `PS3` no es canvia quan s'introdueix un bucle select imbricat. Si voleu una sol·licitud diferent al submenú, assegureu-vos de definir-la en el moment adequat.

Per acabar amb els bucles, és interessant que coneixeu la comanda **`shif`** de Bash. Aquesta ordre té un argument, un número. Els paràmetres de posició es desplacen cap a l'esquerra per aquest número, N. Així doncs, per exemple, amb `shift 4`, el paràmetre de posició `$4` esdevé `$1`, el `$5` esdevé `$2` i així successivament. Els paràmetres de posició `$1`, `$2` i `$3` aquests es depreciarien.

Si N és zero o major que `$#` (número total d'arguments) els paràmetres de posició no es canvien i l'ordre no té cap efecte. Si N no és present, se suposa que és 1. L'estat de retorn és zero a no ser que N sigui major que `$#` o inferior a zero.

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

Normalment, es fa servir una comanda `shift` quan el nombre d'arguments d'una ordre no es coneix per endavant, per exemple, quan els usuaris poden donar tants arguments com vulguin. En aquests casos, els arguments solen processar-se en un bucle while amb una condició de prova de `(($#))`. Aquesta condició és certa sempre que el nombre d'arguments sigui superior a zero. La variable `$1` i la comanda `shift` processen cada argument. El nombre d'arguments es va reduint cada cop que s'executa i, finalment, es converteix en zero, cosa que provocarà la sortida del bucle while.

L'exemple següent, *cleanup.sh*, utilitza la comanda `shift` per processar cada fitxer d'una llista generada per la comanda `find` que cercarà els fitxers als quals no s'ha accedit des de fa més d'un any:

```
rgarcia@ubuntu:~$ cat cleanup.sh
#!/bin/bash

#Aquest script neteja els fitxers als que no has accredit des de fa més d'un any.

US="Usa així aquest script: $0 dir1 dir2 dir3 ... dirN"

if [ "$#" == "0" ]; then
        echo "$US"
        exit 1
fi

while (( "$#" )); do

if [[ $(ls "$1") == "" ]]; then
        echo "Directori buit, res a fer."
  else
        find "$1" -type f -a -atime +365 -exec rm -i {} \;
fi

shift

done
```

ACT 11 | **Treballant amb bucles**

En aquest apartat, hem parlat de com es poden incorporar ordres repetitives en construccions de bucle. La majoria dels bucles habituals es construeixen utilitzant les declaracions **for**, **while** o **until**, o una combinació d'aquestes ordres.

El bucle **for** executa una tasca un nombre de vegades definit. Si no sabeu quantes vegades s'hauria d'executar una ordre, utilitzeu **while** o **until** per especificar quan ha de finalitzar el bucle.

Els bucles es poden interrompre o reiterar utilitzant les declaracions de **break** i **continue**.

La construcció **select** s'utilitza per imprimir menús en scripts interactius.

La connexió a través dels arguments de la línia d'ordres a un script es pot fer utilitzant la instrucció **shift**.

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

Realitza les següents activitats:

a) Crea dins del teu directori *scripts* un nou script anomenat *copyetc.sh* que faci una copia de seguretat de tots els fitxers del directori `/etc` (fitxers de configuració) dins de la mateixa ubicació amb copies amb el mateix nom acabades en `.bak`. Utilitza un bucle for per dur a terme aquesta tasca.
b) Crea un nou script anomenat *countdown.sh* que rebi com a paràmetre un número de segons a partir del qual faci un compte enrere del temps fins mostrar un missatge de finalització. El script ha d'utilitzar un bucle while i fer la comprovació de que s'ha passat un paràmetre com a temps (si no és així has d'alertar al usuari de com utilitzar *countdown.sh*).
c) Crea un nou script anomenat *alarma.sh* que rebi com a paràmetre una hora i minut (format HHMM) en el que ha de "sonar" l'alarma (en realitat serà un missatge). Utilitza un bucle until i com en l'apartat anterior fes una gestió de que *alarma.sh* es cridat amb un paràmetre.
d) Ara has de generar un script anomenat *rellotge.sh* que tingui el següent aspecte en la seva execució:

```
----------------------------------------
Hola benvingut, son les 18:57:43
----------------------------------------

Aquestes són les opcions del teu rellotge:

1. Compte enrere
2. Alarma
3. Cronòmetre
4. Sortir

Escull una opció:
```

- Has de gestionar que el script executi el que correspon a cadascuna de les opcions i després d'executar-ne una acabi.
- Per les dues primeres podràs reutilitzar part del codi dels scripts *countdown.sh* (per a `1. Compte enrere`) i *alarma.sh* (per a `2. Alarma`) tenint en compte que ara hauràs de demanar el temps o l'hora i minut, respectivament, a través d'un `echo` i un `read`.
- Pel que fa a l'opció `3. Cronòmetre`, has de demanar que és polsi una determinada tecla per iniciar (en cas contrari sortir) un comptador de segons que vagin apareixent per pantalla i que finalitzareu amb `Ctrl+C`.
- La opció `4. Sortir`, ha d'emetre un missatge de comiat i sortir.
- Per últim, has de gestionar el fet de que no s'esculli una opció correcta, notificant-lo abans de sortir.

1.16. MÉS ENTORN A LES VARIBLES

Com ja hem vist, Bash entén molts **tipus diferents de variables** o paràmetres. Fins ara, no ens hem molestat gaire en quin tipus de variables assignem, de manera que les nostres variables poden contenir qualsevol valor que se'ls assignés. Per exemple, `VARIABLE=11; echo $VARIABLE` imprimeix "11", i si fem `VARIABLE=frase; echo $VARIABLE` imprimirà "frase", sense importar que son dos tipus diferents de variables (la primera és un *integer* i la segona un *string*).

Hi ha casos, però, en què es vol evitar aquest tipus de comportament, per exemple, quan es manegen números de telèfon i altres. A part dels enters i de les variables, és possible que també vulgueu especificar una variable que sigui una **constant**. Això es fa sovint al principi d'un script, quan es declara el valor de la constant. Després d'això, només hi ha referències al nom de la variable constant, de manera que quan calgui canviar-la, només s'ha de fer una vegada. Una variable també pot ser una sèrie de variables de qualsevol tipus, l'anomenada **matriu de variables** (VAR0, VAR1, VAR2, ... VARN).

Utilitzant una declaració **`declare`**, podem limitar l'assignació de valor a variables. La sintaxi per declarar és la següent: `declare OPTION(s) VARIABLE=value`.

Les següents opcions s'utilitzen per determinar el tipus de dades que la variable pot contenir i assignar-ne atributs:

Opció	Significat
`-a`	La variable és una matriu (*array*).
`-f`	La variable només pot utilitzar noms de funcions (les veurem més endavant).
`-i`	La variable ha de ser tractada com un enter (*integer*).
`-p`	Mostra els atributs i els valors de cada variable.
`-r`	Fer que les variables siguin de només lectura. És a dir, constants (*constant*).
`-t`	Dona a cada variable l'atribut *trace*.
`-x`	Marca cada variable per exportar a comandes posteriors a través de l'entorn.

Utilitzant `+` en lloc de `-` s'apaga l'atribut. Quan s'utilitza en una funció, `declare` crea variables locals.

El següent exemple mostra com l'assignació d'un tipus a una variable influeix en el valor:

```
rgarcia@ubuntu:~$ declare -i VARIABLE=11
rgarcia@ubuntu:~$ VARIABLE=frase
rgarcia@ubuntu:~$ echo $VARIABLE
0
rgarcia@ubuntu:~$ declare -p VARIABLE
declare -i VARIABLE="0"
```

Tingueu en compte que Bash té una opció per declarar un valor numèric, però cap per declarar valors de cadena. Això es deu al fet que, per defecte, si no es donen especificacions, una variable pot contenir qualsevol tipus de dades:

```
rgarcia@ubuntu:~$ OTRAVAR=blabla
rgarcia@ubuntu:~$ declare -p OTRAVAR
declare -- OTRAVAR="blabla"
```

Tan aviat com restringiu l'assignació de valors a una variable, només pot contenir aquest tipus de dades. Recordeu que les restriccions possibles són **enters** (`-i`), **constants** (`-r`) o **matrius** (`-a`).

Veieu un exemple de variable de tipus *constant*:

```
rgarcia@ubuntu:~$ declare -r TUX=penguinpower
rgarcia@ubuntu:~$ TUX=Mickeysoft
bash: TUX: variable de sólo lectura
```

Pel que fa a les variables tipus matrius (*arrays*) són unes variables que contenen múltiples valors. Qualsevol variable es pot utilitzar com a matriu. No hi ha cap límit màxim per a la mida d'una matriu, ni cap requisit que les variables membres estiguin indexades o assignades de forma contigua. Les matrius són *zero-based*, és a dir, el primer element està indexat amb el número 0.

La **declaració indirecta** es fa amb la següent sintaxi: `ARRAY[INDEXNR]=value`. L'`INDEXNR` es tracta d'una expressió aritmètica que ha d'avaluar-se com un número positiu.

La **declaració explícita** d'una matriu es fa tal i com ja heu vist: `declarare -a ARRAYNAME`.

També s'acceptarà una declaració amb un número d'índex, però s'ignorarà el número d'índex. Els atributs de la matriu es poden especificar utilitzant les declaracions integrades de `declare`, però teniu en compte que aquests atributs s'apliquen a totes les variables de la matriu; no podeu tenir matrius mixtes.

Les variables de matriu també es poden crear utilitzant assignacions compostes en aquest format: `ARRAY=(value1 value2 ... valueN)`. Cada valro (`valueX`) tindrà la forma `[indexnumber=]string` (el número d'índex és opcional). Si se subministra, se li assigna aquest índex; en cas contrari, l'índex de l'element assignat és el número de l'últim índex que s'ha assignat, més un. Aquest format també és acceptat per `declare`. Recordeu que si no s'ofereixen números d'índex, la indexació comença a zero.

Si més endavant necessitem fer una addició de membres "extra" a una matriu es fa utilitzant la sintaxi: `ARRAYNAME[indexnumber]= value`.

Si reviseu les opcions de `read` veureu que proporciona l'opció `-a ANAME`, que permet llegir i assignar valors als membres d'una matriu.

Per fer referència al contingut d'un element d'una matriu, feu servir claus `{}`. Això és necessari, com podeu veure a l'exemple següent, per passar per alt la interpretació d'un intèrpret d'ordres dels operadors d'expansió. Si el nombre d'índex és `@` o `*`, es fa referència a tots els membres d'una matriu:

```
rgarcia@ubuntu:~$ ARRAY=(one two three)
rgarcia@ubuntu:~$ echo ${ARRAY[*]}
one two three
rgarcia@ubuntu:~$ echo $ARRAY[*]
one[*]
rgarcia@ubuntu:~$ echo ${ARRAY[2]}
three
rgarcia@ubuntu:~$ ARRAY[3]=four
rgarcia@ubuntu:~$ echo ${ARRAY[*]}
one two three four
```


Autor | RAFAEL GARCÍA SUÁREZ
Contacte | rgarc259@xtec.cat

Per entendre perquè la comanda `echo $ARRAY[*]` torna el resultat que torna, cal que sabeu que referir-se al contingut d'una variable membre d'una matriu sense proporcionar un nombre d'índex és el mateix que referir-se al contingut del primer element, el que es fa referència amb el número d'índex zero. En l'exemple en no utilitzar les claus `{}` fem escrivim per pantalla `one` que es l'element d'índex 0, seguit del que ara és un text pla `[*]`.

Per tal d'eliminar matrius, o variables membres d'una matriu, utilitzarem **`unset`**. Seguint amb l'exemple anterior:

```
rgarcia@ubuntu:~$ unset ARRAY[1]
rgarcia@ubuntu:~$ echo ${ARRAY[*]}
one three four
rgarcia@ubuntu:~$ unset ARRAY
rgarcia@ubuntu:~$ echo ${ARRAY[*]}

rgarcia@ubuntu:~$
```

Els exemples pràctics de l'ús de matrius són difícils de trobar. Trobareu molts scripts que realment no fan res al vostre sistema, però que utilitzen matrius per calcular sèries matemàtiques, per exemple. I aquest seria un dels exemples més interessants. La majoria dels scripts només mostren el que es pot fer amb un *array* d'una manera simplificada i teòrica.

La raó d'aquesta opacitat és que les matrius són estructures bastant complexes. Trobareu que els exemples més pràctics en els quals es podrien utilitzar matrius que ja s'han implementat al vostre sistema utilitzant matrius, tot i que a un nivell inferior, al llenguatge de programació C en el qual s'escriuen la majoria d'ordres UNIX. Un bon exemple és l'ordre basada en l'historial de Bash.

Una altra raó que són difícils de trobar bons exemples és que no tots els intèrprets de comandes (shells) donen suport a matrius, de manera que trenquen la compatibilitat.

Tot i això, provaré d'utilitzar el següent script com a possible exemple:

```
rgarcia@ubuntu:~$ cat senarsparells.sh
#!/bin/bash

# Aquest script diu si cada element d'una matriu es parell o senar

llistanum=(12 4 57 21 8 93)

count=0
while [ "x${llistanum[count]}" != "x" ]; do
   num=$(( ${llistanum[count]} % 2 ))
   if [ "$num" == "0"]; then
      echo "${llistanum[count]} és parell"
   else
      echo "${llistanum[count]} és senar"
   fi
   count=$(( $count + 1 ))
done
```

La sortida esperada per l'exemple anterior seria:

```
rgarcia@ubuntu:~$ senarsparells.sh
12 és parell
4 és parell
57 és senar
21 és senar
8 és parell
93 és senar
```

ACT 12 | **Variables del tipus matriu**
Basant-te en l'exemple anterior intenta modificar el script *senarsparells.sh* per què els números a avaluar es passin com a paràmetres del script. Per tant, caldrà generar la matriu `llistanum` llegint els paràmetres posicionals de l'entrada (podrà ser qualsevol número de paràmetres d'entrada).

Utilitzant la sintaxi `${#VAR}` calcularà el nombre de caràcters d'una variable. Si VAR és "*" o "@", aquest valor se substitueix pel nombre de paràmetres de posició o nombre d'elements d'una matriu en general. Això es demostra a l'exemple següent:

```
rgarcia@ubuntu:~$ echo $SHELL
/bin/bash
rgarcia@ubuntu:~$ echo ${#SHELL}
9
rgarcia@ubuntu:~$ ARRAY=(one two three)
rgarcia@ubuntu:~$ echo ${#ARRAY}
3
```

Per últim cal que coneixeu diferents tipus de **transformacions de variables**. La primera d'aquestes transformacions és la **substitució**. Utilitzarem la sintaxis `${VAR:-WORD}`. Si `VAR` ja està definida descartarem `WORD`, en canvi, si `VAR` no està definida, o te valor nul, `VAR` utilitzarà el valor `WORD`. Veieu aquest exemple:

```
rgarcia@ubuntu:~$ echo ${var:-25}
25
rgarcia@ubuntu:~$ echo ${#var}

rgarcia@ubuntu:~$ var=10
rgarcia@ubuntu:~$ echo ${var:-25}
10
```

Noteu com la variable var **utilitza** el valor 25 si no te valor definit, o és nul, però no se li **assigna**.

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

Això pot ser útil dins de molts scripts. Imagineu que teniu el típic script que requereix un paràmetre que identifiqui sobre quin directori ha de treballar, sinó es facilita aquest paràmetre s'ha d'utilitzar un per defecte. Una primera idea podria ser construir tot un condicional `if - else`, com el següent:

```
declare -r DEAFULT_PATH=/etc

if [ $1 ]; then
   file_path=$1
else
   file_path=$DEAFULT_PATH
fi
```

Però amb la transformació per substitució que hem vist ho podríem fer més senzill:

```
declare -r DEAFULT_PATH=/etc
file_path=${1:-$DEFAULT_PATH}
```

Si volem **assignar**, i no sols utilitzar, el valor de substitució a la variable substituirem el "-" de la sintaxis per un "=":

```
rgarcia@ubuntu:~$ echo ${var:=25}
25
rgarcia@ubuntu:~$ echo ${#var}
25
rgarcia@ubuntu:~$ var=10
rgarcia@ubuntu:~$ echo ${var:=25}
10
```

La següent transformació que podem considerar és l'**eliminació de subcadenes**. Utilitzarem la següent sintaxis: `${VAR:OFFSET:LENGTH}`. S'eliminaran de VAR el número de caràcters marcats per OFFSET. Si definim també LENGTH marcarem quants caràcters s'han de mantenir després de l'eliminació, si no es defineix la resta de variable és mantindrà fins al final d'aquesta. Observeu l'exemple següents amb diferents possibilitats de l'ús de l'eliminació de subcadenes:

```
rgarcia@ubuntu:~$ frase='Hola bash; Adeu bash'
rgarcia@ubuntu:~$ echo $frase
Hola bash; Adeu bash
rgarcia@ubuntu:~$ echo ${frase:5}
bash; Adeu bash
rgarcia@ubuntu:~$ echo ${frase:5:4}
bash
rgarcia@ubuntu:~$ echo '***'${frase:11}'***'
***Adeu bash***
rgarcia@ubuntu:~$ echo ${frase:{#frase}-16}
bash
```

Una altra opció per eliminar part de les cadenes i fer-ho a través de **patrons** i utilitzant alguna de les següents sintaxis: `${VAR#PATTERN}`, `${VAR##PATTERN}`, `${VAR%PATTERN}` i/o `${VAR%%PATTERN}`. Cal entendre que:

- Totes eliminen el patró `PATTERN` de la variable `VAR`.
- El coixinet busca des del principi de la variable.
- El percentatge busca des del final de la variable.
- Un únic signe cerca la menor coincidència amb el patró `PATTERN` dins de `VAR`, i dos signes cerquen la major coincidència amb el patró `PATTERN` dins de `VAR`.

Veieu com funciona seguint amb l'exemple anterior i la variable `frase='Hola bash; Adeu bash'`:

```
rgarcia@ubuntu:~$ echo ${frase#*ba}
sh; Adeu bash
rgarcia@ubuntu:~$ echo ${frase##*ba}
sh
rgarcia@ubuntu:~$ echo ${frase%ba*}
Hola bash; Adeu
rgarcia@ubuntu:~$ echo ${frase%%ba*}
Hola
rgarcia@ubuntu:~$ echo ${frase%ba*}"****"
Hola Bash; Adeu ****
```

Observeu que en la última instrucció aprofitem per fer una substitució. Al següent exemple una aplicació interessant:

```
rgarcia@ubuntu:~$ cat gif-jpg.sh
#!/bin/Bash

# Aquest script canvia la extensió dels fitxers .gif del teu directori personal

for file in $HOME/*.gif; do
   echo -n "Renombrant "
   basename $file
   mv $file ${file%*gif}jpg
done
```

També ho podem utilitzar per buscar patrons d'eliminació dins d'una matriu. Observeu l'exemple:

```
rgarcia@ubuntu:~$ echo ${MATRIU[*]}
one two one three one four
rgarcia@ubuntu:~$ echo ${MATRIU[*]#one}
two three four
rgarcia@ubuntu:~$ echo ${MATRIU[*]#t}
one wo one hree one four
rgarcia@ubuntu:~$ echo ${MATRIU[*]#t*}
one wo one hree one four
rgarcia@ubuntu:~$ echo ${MATRIU[*]##t*}
one one one four
```


Autor | RAFAEL GARCÍA SUÁREZ
Contacte | rgarc259@xtec.cat

L'última transformació a estudiar és **cerca i substitució** de parts d'una variable. Les sintaxis següents us seran útils per aconseguir-ho: `${VAR/PATTERN/WORD}` i/o `${VAR//PATTERN/WORD}`. En la primera de les sintaxis només es substituirà la primera aparició del patró `PATTERN` dins de la variable `VAR` pel valor de `WORD`. En la segona sintaxis es substituiran totes les aparicions del patró `PATTERN` dins de la variable `VAR` pel valor de `WORD`. Continuant amb `frase='Hola bash; Adeu bash'`:

```
rgarcia@ubuntu:~$ echo ${frase/'bash'/'BASH'}
Hola BASH; Adeu bash
rgarcia@ubuntu:~$ echo ${frase//'bash'/'BASH'}
Hola BASH; Adeu BASH
```

ACT 13 | **Cerca i substitució de variables**

Considerant el que has aprés sobre la transformacions de variables realitza la següent activitat:

Crea un nou script al teu directori *scripts* anomenat *color.sh*. Aquest script utilitzarà aquestes dues variables:

```
colors=(vermell blau verd negre taronja groc blanc lila gris marró)
text='El color és el teu color de la sort per avui'
```

Tenint en compte això genera el script *colors.sh* que demani un número (del 0 al 9) que serà una posició de la matriu `colors` i que determinarà el color de la sort del dia, que s'ha d'imprimir utilitzant la variable `text`.

Investiga com utilitzar la variable `$RANDOM` per generar un número aleatori del 0 al 9, i modifica *color.sh* per tal de que el color de la sort del dia és doni automàticament i de forma aleatòria.

1.17. FUNCIONS

Les **funcions** de shell són una manera de agrupar comandes per a una execució posterior, utilitzant un nom únic per a aquest grup o rutina. El nom de la rutina ha de ser únic dins de shell o script. Totes les comandes que componen una funció s'executen com a comandes regulars. Quan es demana una funció com a nom d'ordre simple, s'executa la llista de comandes associades amb aquest nom de funció. Una funció s'executa dins del shell en què s'ha declarat: no es crea cap procés nou d'intèrpret de comandes. Les comandes integrades especials[23] es troben abans que de les funcions de shell durant el procés d'interpretació d'ordres.

Tenim dos sintaxis vàlides per definir una funció de shell `FUCTION`:

`function FUNCTION { COMMANDS; }` i/o

`FUNCTION () { COMMANDS; }`

[23] Aquestes comandes integrades especials són: `break`, `:`, `.`, `continue`, `eval`, `exec`, `exit`, `export`, `readonly`, `return`, `set`, `shift`, `trap` i `unset`.

Les ordres llistades entre claus `{ COMMANDS; }` formen el cos de la funció. Aquestes ordres s'executen cada vegada que `FUNCTION` s'especifica com el nom d'una comanda. L'estat de sortida és l'estat de sortida de l'última ordre executada al cos de la funció.

Les funcions són com "mini-scripts": poden acceptar paràmetres, poden utilitzar variables conegudes només dins de la funció (utilitzant la shell local integrada) i poden retornar valors a la shell que truca.

Una funció també té un sistema per interpretar paràmetres de posició. Tanmateix, els paràmetres de posició passats a una funció no són els mateixos que els passats a una ordre o un script.

Quan s'executa una funció, els arguments de la funció es converteixen en els paràmetres de posició durant la seva execució. El paràmetre especial `#` que s'expandeix al nombre de paràmetres de posició s'actualitza per reflectir el canvi. El paràmetre de posició `0` no canvia. La variable Bash `FUNCNAME` s'estableix al nom de la funció, mentre s'està executant.

Si el retorn incorporat s'executa en una funció, la funció es completa i l'execució es reprèn amb la següent comanda després de la trucada de funció. Quan es completa una funció, es restauren els valors dels paràmetres de posició i el paràmetre especial als valors que tenien abans de l'execució de la funció. Si es retorna un argument numèric, es retorna aquest estat. Un exemple senzill:

```
rgarcia@ubuntu:~$ cat showparams.sh
#!/bin/bash

echo "Aquest script demostra els arguments de les funcions."
echo

echo "El paràmetre posicional 1 del script és $1."

test ()
{
echo "El paràmetre posicional 1 de la funció es $1."
VALOR_RETORN=$?
echo "El exit code de la funció test és $VALOR_RETORN."
}

test parametrenou

rgarcia@ubuntu:~$ ./showparams.sh parametre1
Aquest script demostra els arguments de les funcions.

El paràmetre posicional 1 del script és parametre1.
El paràmetre posicional 1 de la funció es parametrenou.
El exit code de la funció test és 0.
```


Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

Tingueu en compte que el valor de retorn o el codi de sortida de la funció sovint s'emmagatzemen en una variable, de manera que es pugui sondar en un punt posterior.[24]

Hi ha molts scripts al vostre sistema que utilitzen les funcions com a manera estructurada de gestionar sèries d'ordres. Moltes d'aquestes funcions es troben en algun fitxer de configuració del vostre sistema. Mitjançant aquest mètode, les tasques habituals, com ara comprovar si un procés s'executa (`status`), iniciar (`start`) o aturar (`stop`) un dimoni, etc., només s'ha d'escriure una vegada, de manera general. Si es necessita la mateixa tasca, es recicla el codi.

Podeu crear el vostre propi fitxer `funcions` al vostre directori personal per contenir totes les funcions que utilitzeu regularment en diferents scripts. Després només cal posar la línia `. $HOME/funcions` en algun lloc al principi del script i podeu reciclar funcions que ja teniu definides.

ACT 14 | **Crides a funcions i scripts**

Crea el fitxer `/etc/funcions` que comentàvem fa una estona i dintre genera una funció anomenada *noparam* que enviarà un missatge tipus: "El script que has executat requereix al menys un argument".

a) Genera un script al teu directori *scripts* anomenat *sumaarg.sh* que sumi tots els números que es passin com arguments al script. Aquest a de cridar a la funció *noparm* inicialment per comprovar que al menys hi hagi un número que sumar.
b) Millora el teu script *sumaarg.sh* fent que en acabar demani si voleu sumar uns altres números. En cas afirmatiu heu d'introduir una nova sèrie de valors que serviran per tornar a cridar al teu script[25] amb els nous valors com a paràmetres.

[24] Els scripts d'inici del vostre sistema solen utilitzar la tècnica de sondejar la variable `RETVAL` en una prova condicional, com aquesta:

```
if [ $RETVAL -eq 0 ]; then
        <start the daemon>
```

O com aquest exemple del script /etc/init.d/amd, on s'utilitzen les funcions d'optimització de Bash:

```
[ $RETVAL = 0 ] && touch /var/lock/subsys/amd
```

Les ordres després de `&&` només s'executen quan la prova resulta certa; aquest és un mètode més curt per representar una estructura `if/then/fi`. Veuràs un munt de fitxers dels *initscripts* que acaben en alguna cosa com la `exit $RETVAL`.

[25] Podeu cridar a un script des d'un altre utilitzant `source nomscript.sh` o simplement amb `. nomscript.sh`.

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

PT 1 | UNA PAPERERA DE RECICLATGE EN BASH

En finalitzar aquesta pràctica haurem generat un sistema d'esborrat de fitxers que els emmagatzemi a una paperera de reciclatge des de la qual podrem recuperar fitxers, llistar-los, buidar-los definitivament, etc.

OBJECTIUS

- Crear Bash scripts personalitzats per a la gestió de fitxers al sistema.
- Generar funcions pròpies per la seva utilització posterior dins dels scripts generats.
- Elaborar fitxers de registre (logs) sobre activitats portades a terme.
- Programar l'execució automàtica de scripts a través de l'administrador de tasques (cron).
- Buscar, analitzar i interpretar la documentació tècnica necessària.
- Realitzar manuals tècnics dels procediments realitzats.

Abans de començar genereu a l'arrel del sistema un directori anomenat *trash* amb els subdirectoris següents: *trashscripts* (on ubicaràs els 3 scripts a generar a la pràctica), *trashlogs* (on guardar els fitxers de recolzament i registre amb els que treballarem). Aquest últim subdirectori en tindrà un altre dins anomenat *olds*. Penseu en donar accés de lectura i escriptura per a tots els usuaris del sistema.

El script borra.sh

El primer script que haureu de generar s'anomenarà *borra.sh*, i tindrà com objectiu esborrar el fitxer, o fitxers, passat com a paràmetre. A elecció del usuari aquest esborrat haurà de ser definitiu o s'haurà de simular movent el fitxer, o fitxers, a un nou directori anomenat *paperera*.

S'ha de gestionar si, com a mínim, s'introdueix un fitxer a eliminar, així com si el fitxer, o fitxers, existeixen. Pensa en un sistema que faci una bona comunicació amb l'usuari, oferint ajuda de com utilitzar el teu script *borra.sh*, i informació de quin, o quins, dels fitxers passats no existeixen.

Per últim, considera que has de generar un parell de fitxers de registre. El primer s'anomenarà *restauració.log* que guardi el nom del fitxer i la seva ubicació prèvia a l'ús de *borra.sh* si no és fa una eliminació definitiva. El segon s'anomenarà *trashlog.log* i, des del script *borra.sh* generarà una entrada amb un format similar a aquest:

```
reciclatge:11/04/2019:12.43:/home/rgarcia/test/fitxer1.txt:rgarcia
eliminació:12/04/2019:18.24:/home/abagur/fitxer23.mp3:abagur
```

Com veieu, cal emmagatzemar: el tipus d'eliminació feta (reciclatge si es mou a la paperera, o eliminació si s'elimina definitivament); la data d'execució (en format DD/MM/AA); l'hora d'execució (en format HH.MM); la ruta absoluta al fitxer implicat; i l'usuari que realitza l'acció.

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

El script mytrash.sh

Genereu un segon script anomenat *mytrash.sh*. Aquest script ens servirà com a gestor del directori *paperera*, és a dir, ens permetrà dur a terme diferents accions sobre els fitxers esborrats temporalment a través del script *borra.sh*.

La funció bàsica d'aquest script serà mostrar informació sobre l'estat de la paperera i un menú d'opcions vinculades a les accions possibles a realitzar. El pes del que farà cadascuna de les opcions s'haurà de programar en un fitxer *trashfunctions* (emmagatzemeu-lo a */trash/trashscripts*), que ens servirà com a magatzem de funcions que utilitzarà el nostre script *mytrash.sh*.

En executar *trash.sh* hauríem de trobar una informació i menú semblant al següent:

```
Benvingut a MyTrash, actualment la teva paperera conté 12 elements (381M)
ATENCIÓ! Aquesta paperera es buida automàticament el dia 28 de cada mes.


En que et puc ajudar:


1. Llistar els fitxers eliminats
2. Informació sobre un fitxer eliminat
3. Restaurar un fitxer
4. Restaurar per extensió
5. Buidar la paperera ara
6. Buidar només els més antics
7. Sortir de MyTrash


Que vols fer [1-8]:
```

Observa que a la informació prèvia és mostra el número de fitxers que conté el directori *paperera* així com la mida que ocupen aquest elements. També avisa de que es buidarà automàticament el dia 30 de cada mes, cosa que gestionarem amb un tercer script més endavant.

Pel que fa a les opcions hauran de tenir el següent comportament:

1. Mostar la llista de fitxers eliminats amb el format: `nomfitxer <-- ubicació original`.
2. Demana un fitxer a l'usuari, si aquest existeix a la paperera, mostra informació sobre el tipus de fitxer (comanda `file`).
3. Demana un fitxer a l'usuari, si aquest existeix a la paperera, el retorna a la seva ubicació original.
4. Demana una extensió (*.txt*, *.mp3*, *.pdf*, *.tar*, etc.) al usuari i torna a la seva ubicació original tots els fitxers amb l'extensió demanada. Ha d'informar a l'usuari de quants fitxers s'han restaurat.
5. Elimina tots els fitxers continguts a la paperera. Ha de demanar confirmació a l'usuari.
6. Demana un número de dies a l'usuari i elimina els fitxers continguts a la paperera que portin més dies a la paperera dels indicats per l'usuari. Ha de demanar confirmació a l'usuari i ha d'informar-lo després de quants fitxers s'han eliminat definitivament.
7. Surt del script *mytrash.sh*.

Autor | **RAFAEL GARCÍA SUÁREZ**
Contacte | **rgarc259@xtec.cat**

A més, considera que després de cada opció tornarem a informar a l'usuari dels elements restants a la paperera així com del espai que ocupen, i tronarem a mostrar el menú d'opcions, per tal d'escollir una nova acció (òbviament excepte si escollim la opció 7, que sortirem amb un missatge de comiat).

Per últim pensa que el script *trash.sh* també modificarà el fitxer *trashlog.log* incloent entrades de tipus "eliminació" al utilitzar les opcions 5 i/o 6, i entrades d'un nou tipus "restauració" en escollir opcions 3 i/o 4.

El script neteja.sh

Genereu un tercer script anomenat *neteja.sh*. Aquest script ens servirà per buidar el directori *paperera* automàticament el dia 28 de cada mes. Podríeu considerar fer-ho l'últim dia de cada més, tot i que es una feina més complexa. Per tant, cal programar correctament l'execució de *neteja.sh* dins del cron del sistema.

Aprofita aquest mateix script per buidar el contingut dels fitxers de registre *restauració.log* i *trashlog.log*. Tingués en compte, però, que en el cas de *trashlog.log* haurem de guardar un historial (a la ruta */trash/trashlogs/olds*) de fitxers d'aquest registre, i per tant, abans de buidar-lo haureu de copiar el fitxer a un altre anomenat *trashlog.old.date* on *date* serà la data on es crea el fitxer en format MMAAAA (el dia no caldrà ja que òbviament serà sempre el 28).

RESULTATS

Arribat el final de la pràctica l'alumnat haurà de disposar de 3 scripts *borra.sh*, *mytrash.sh* i *netja.sh*. El primer podrà simular l'eliminació de fitxers movent-los a un directori paperera. El segon donarà un menú d'opcions que han de permetre buidar la paperera, restaurar fitxers, etc. El tercer script haurà de programar-se per funcionar automàticament per tal d'eliminar els fitxers generats durant la utilització dels 2 scripts anteriors així com guardar copies de seguretat d'alguns registres antics. Al mateix temps, haurà generat diferents fitxers de recolzament i registre per dur a terme i/o revisar les tasques efectuades.

A més, haurà de tenir un document on s'especifiqui els diferents codis dels scripts generats. Aquests codis hauran d'estar comentats correctament, i es valorarà que facin una gestió d'errors adequada. Cal a més, aportar captures de pantalla amb demostracions de funcionament correcte dels diferents scripts, funcions i generació de fitxers. Cal lliurar aquesta documentació a través del **Moodle del curs** dins del **termini establert**.