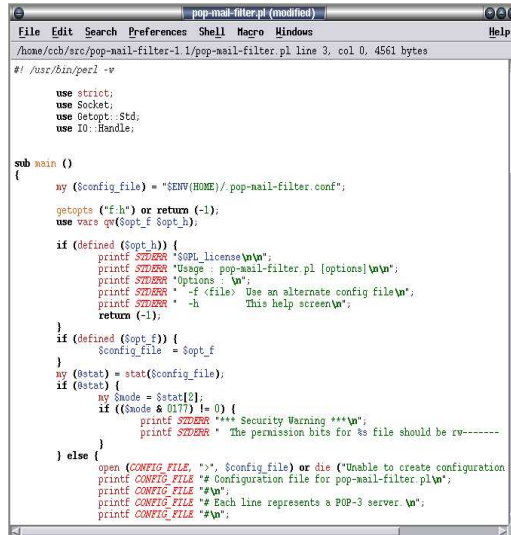


NEdit

```
$ nedit mon_script.sh
```



```
pop-mail-filter.pl (modified)
File Edit Search Preferences Shell Macro Windows Help
/home/cch/src/pop-mail-filter-1.1/pop-mail-filter.pl line 3, col 0, 4561 bytes
#!/usr/bin/perl -v

use strict;
use Socket;
use Getopt::Std;
use IO::Handle;

sub main ()
{
    my ($config_file) = "$ENV{HOME}/pop-mail-filter.conf";

    getopts ("f:h") or return (-1);
    use vars qw($opt_f $opt_h);

    if (defined ($opt_h)) {
        printf STDERR "SPL license\n";
        printf STDERR "Usage : pop-mail-filter.pl [options]\n";
        printf STDERR "Options : \n";
        printf STDERR "  -f <file> Use an alternate config file\n";
        printf STDERR "  -h          This help screen\n";
        return (-1);
    }

    if (defined ($opt_f)) {
        $config_file = $opt_f;
    }
    my ($stat) = stat($config_file);
    if ($stat) {
        my $mode = $stat[2];
        if (($mode & 0177) != 0) {
            printf STDERR "*** Security Warning ***\n";
            printf STDERR "  The permission bits for %s file should be rw-----\n";
        }
    } else {
        open (CONFIG_FILE, ">", $config_file) or die ("Unable to create configuration\n");
        printf CONFIG_FILE "# Configuration file for pop-mail-filter.pl\n";
        printf CONFIG_FILE "#\n";
        printf CONFIG_FILE "# Each line represents a POP-3 server.\n";
        printf CONFIG_FILE "#\n";
    }
}
```

Voir aussi :

Web

– Site principal de NEdit (<http://www.nedit.org/>)

Invocation de l'interpréteur

On peut exécuter un script en appelant explicitement l'interpréteur :

```
$ csh mon_script.sh  
$ source mon_script.sh
```

```
$ sed mon_script.sed  
$ awk -f mon_script.awk
```

Sous Unix, les suffixes (.sh, .sed, .awk, .pl, ...) n'ont qu'un rôle d'information pour l'utilisateur, mais n'ont aucune signification pour le système.

À vous...

Utilisez l'un des éditeurs disponibles sur votre système pour écrire le script suivant :

test.sh

```
set VAR=25
```

Exécutez-le successivement avec les commandes suivantes :

```
$ csh test.sh ; echo $VAR
```

```
$ source test.sh ; echo $VAR
```

Voyez-vous une différence ?

Pour pouvoir invoquer le script directement par son nom, il faut d'abord que le noyau sache où le trouver.

Essayez d'appeler directement :

```
$ test.sh
```

Quel est le résultat ?

On peut placer le script dans un endroit connu par le noyau. Ce dernier ira le chercher dans l'ensemble des répertoires indiqués dans la variable d'environnement `PATH`.

Affichez le contenu de votre variable d'environnement :

```
$ echo $PATH
```

Quels sont les répertoires parcourus pour rechercher les fichiers ?

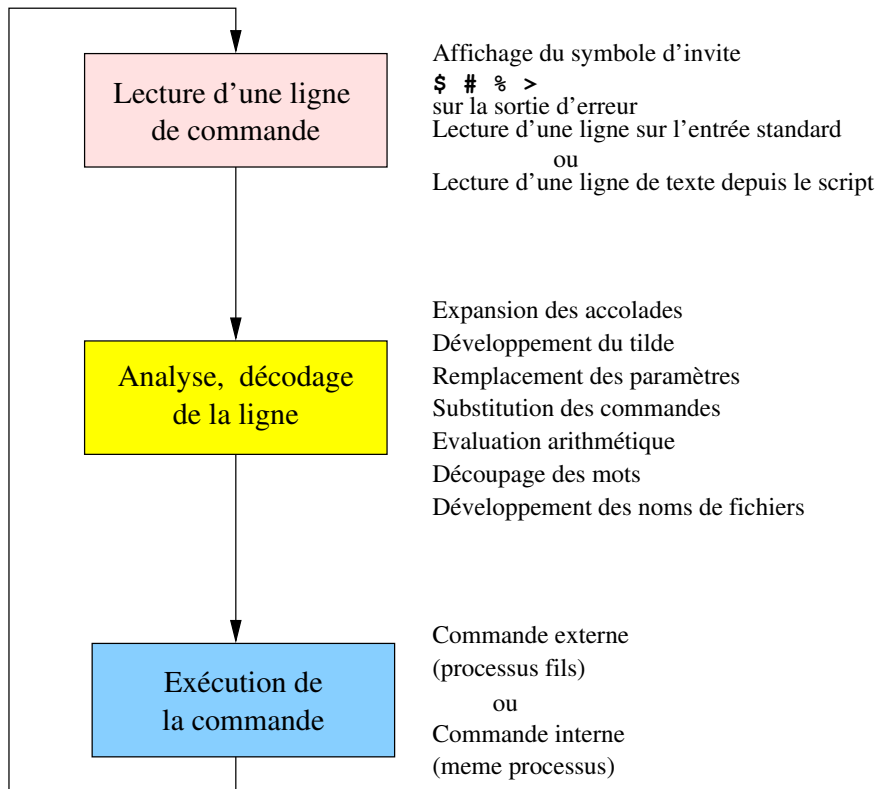
On peut indiquer explicitement que le fichier se trouve dans le répertoire en cours :

```
$ ./test.sh
```

Quel est le résultat ?

Interprétation d'une commande

Boucle générale du shell



Expansion des accolades

À vous...

Que donnent les lignes suivantes :

```
$ echo image_{05,06,07}.gif
$ echo images-{juillet,aout,{septem,octo,novem,decem}bre}.jpg
$ echo images{}.jpg
$ echo images}.jpg
$ echo images{.jpg
```

Conclusion ?

```
prefixe_{nom1,nom2,nom3}_suffixe
```

est développé ainsi :

```
prefixe_nom1_suffixe prefixe_nom2_suffixe prefixe_nom3_suffixe
```

Ne pas confondre ces accolades et celles qui servent à regrouper les commandes composées que nous verrons plus loin.

NB : le shell Bourne original n'effectue pas l'expansion des accolades, ni les versions du shell Korn antérieures à Ksh 93.

Développement du tilde

A vous...

Qu'affichent les lignes suivantes :

```
$ echo ~
```

```
$ echo ~userd/bin
```

```
$ echo ~inexistant
```

```
$ echo a~
```

```
~nom/suite/du/chemin  
est remplacé par  
/home/nom/suite/du/chemin
```

La chaîne ~/ est remplacée par le contenu de la variable d'environnement HOME si elle existe (et le répertoire personnel de l'utilisateur sinon).

Le répertoire personnel est déterminé en consultant `/etc/passwd`.

Si le nom d'utilisateur n'existe pas, la chaîne est inchangée.

Le remplacement n'a lieu que si le ~ est en début de chaîne.

Remplacement des paramètres

À vous...

Qu'affichent les lignes suivantes :

```
$ set a = chaine      (avec des espaces autour du =)
$ echo $a
```

```
$ set a=azert        (sans espaces)
$ echo $a
```

```
$ echo $ay
```

```
$ echo ${a}y
```

`${nom}`

est remplacé par la valeur de la variable ou du paramètre nom.

Lorsqu'il n'y a pas d'ambiguïté possible, on peut utiliser \$nom à la place de \${nom}.

À vous...

```
$ set a=azert  
$ echo $?a
```

```
$ set vide=""  
$ echo $?vide
```

```
$ echo $?indefinie
```

`$?` est un opérateur permettant de connaître le statut d'une variable.

À vous...

```
$ set a=azertyuiop  
$ echo $a          (ou echo ${%a})
```

```
$ expr $a : 'aze\(.*\).'
```

```
$ expr $a : '\(.*\)iop'
```

```
$ expr $a : '.\{3\}\(..\{4\}\)'
```

```
$ expr $a : '.\{3\}\(.*\)'
```

```
${%variable}
```

est remplacé par le nombre de caractères contenus dans la variable.

```
expr $variable : 'prefixe\(.*\).'
```

est remplacé par le contenu de la variable une fois qu'on a éliminé le prefixe.

```
expr $variable : '\(.*\)suffixe'
```

est remplacé par le contenu de la variable une fois qu'on a éliminé le suffixe.

```
expr $variable : '.\{debut\}\(.\{longueur\}\)'
```

est remplacé par la sous-chaîne commençant à la position debut (comptée à partir de zéro) et ayant la longueur indiquée. Avec * en guise de longueur on va jusqu'à la fin de la chaîne.

Les shells Korn antérieurs à Ksh 93 n'offrent pas « non plus » l'extraction de sous-chaîne.

PAGE BLANCHE !

Substitution de commandes

Qu'affichent les lignes suivantes :

```
$ set a=`date`  
$ echo $a
```

```
$ set a=`date +%Y`  
$ echo $a
```

```
$ set a=`ls`  
$ echo $a
```

```
$ set a=`ls -l`  
$ echo $a  
$ echo "$a"
```

Syntaxe :

``commande``

La commande est exécutée, et la construction ``...`` est remplacée par le résultat de sa sortie standard.

CHAQUE saut de ligne est remplacé par un espace à l'exception du saut l'éventuel saut de ligne final de la sortie standard est simplement éliminé lors de la substitution de commande

2 difficultés potentielles dans l'emploi de cette syntaxe :

- risque de confusion entre ``commande`` et les apostrophes encadrant une 'chaîne',
- difficulté d'imbriquer des appels (utiliser des backslashes `\`).

Evaluation arithmétique

Syntaxe :

```
@ expression
```

Le contenu de l'expression est évalué selon les règles de l'arithmétique entière, et la construction est remplacée par le résultat du calcul.

À vous...

```
$ @ a= ((12 + 3 * (5 + 3) * 5)) ; echo $a
```

```
$ @ a=10  
$ @ a=$a + 1  
$ echo $a
```

```
$ @ a+=1  
$ echo $a
```

```
$ @ a++  
$ echo $a
```

```
$ @ a=($a << 4)  
$ echo $a
```

L'évaluation est limitée à l'arithmétique entière. Pour des calculs avec des nombres réels, voir l'utilitaire bc(1).

Les parenthèses ajoutent des priorités dans les expressions.

Les constantes sont exclusivement des entiers.

ATTENTION aux espaces, parfois nécessaires !
--

Le shell Bourne original n'offre pas d'arithmétique, il faut invoquer la commande externe expr(1).

Opérateurs arithmétiques

+ -	plus et moins unaires
! ~	négations logique et binaire
* / %	multiplication, division, reste
+ -	addition, soustraction
< < >>	décalages binaires
<= >= < >	comparaisons
== !=	égalité, différence
&	ET binaire
^	OU exclusif binaire
	OU binaire
&&	ET logique
	OU logique
= *= /= %= += -= <<= >>= &= ^= =	affectations

Protections et découpage en mots

À vous...

```
$ a=azerty  
$ echo $a  
$ echo "$a"  
$ echo '$a'
```

```
$ echo "`ls`"  
$ echo '`ls`'
```

```
$ echo "ab{0,1}cd"  
$ echo "~/bin"
```

Les chaînes entre apostrophes simples ' ne sont jamais modifiées.

Dans une chaîne entre guillemets droits ", les modifications suivantes ont lieu :

- remplacement des paramètres et variables
- substitution de commande

mais ni l'expansion des accolades, ni le développement du tilde.

Les guillemets vont servir à préserver l'intégrité d'une chaîne sans qu'elle soit découpée en autant d'arguments qu'elle comporte de mots.

À vous...

```
$ touch "un fichier"  
$ ls -l un fichier  
$ ls -l "un fichier"  
$ ls -l un\ fichier
```

```
$ set a=un fichier
```

```
$ set a="un fichier"  
$ ls -l $a  
$ ls -l "$a"
```

PAGE BLANCHE !

```
ls -l /mnt/dos/Mes Documents/Mes Images/
```

```
ls
```

```
-l
```

```
/mnt/dos/Mes
```

```
Documents/Mes
```

```
Images
```

```
ls -l /mnt/dos/Mes\ Documents\Mes\ Images/
```

```
ls -l "/mnt/dos/Mes Documents/Mes Images/"
```

```
ls -l '/mnt/dos/Mes Documents/Mes Images/'
```

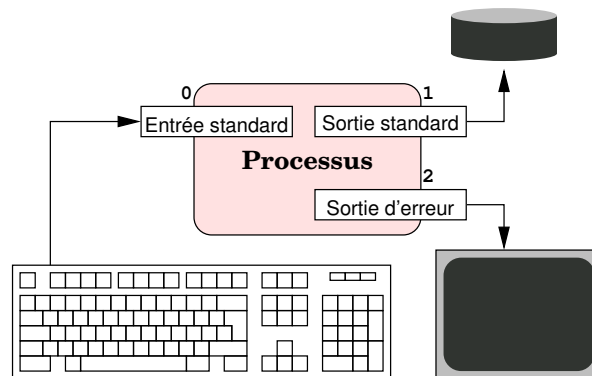
```
ls
```

```
-l
```

```
/mnt/dos/Mes Documents/Mes Images
```

Redirections de la sortie standard

Avec la notation `commande > fichier` la sortie standard du processus est redirigée vers le fichier plutôt que sur le terminal.

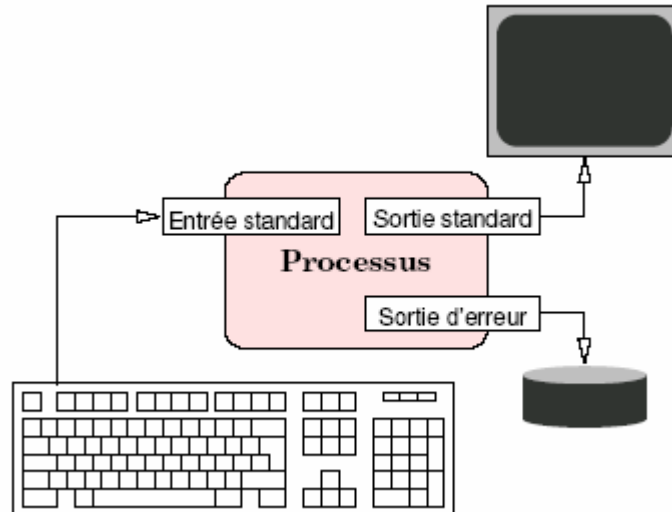


La redirection `commande >> fichier` fait un **ajout** en fin de fichier sans écrasement des données précédentes.

```
$ grep root /etc/*  
$ grep root /etc/* > standard  
$ grep root /etc/* >> standard
```

Redirection de la sortie d'erreur

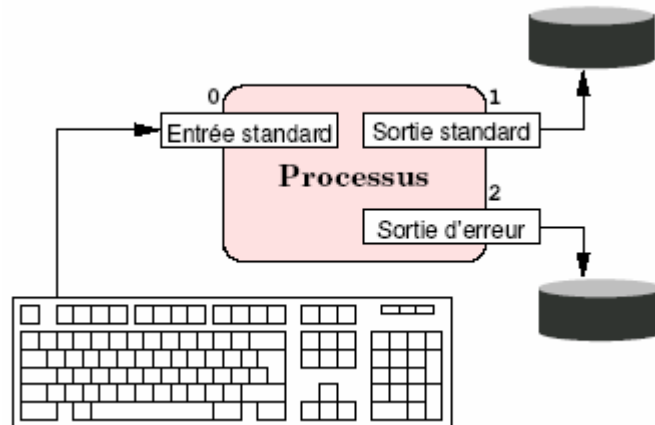
La notation `(commande > /dev/tty) >& fichier` redirige la sortie d'erreur du processus.
Et `(commande2 > /dev/tty) >>& fichier` ajoute en fin de fichier.



```
$ grep root /etc/*  
$ (grep root /etc/* > /dev/tty) >& erreur  
$ (grep root /etc/* > /dev/tty) >& /dev/null
```

Redirections des deux sorties

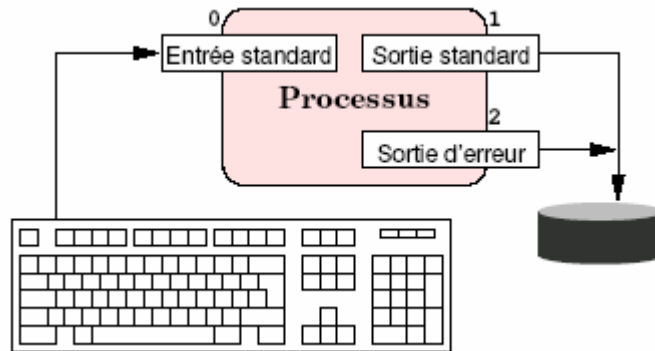
Les sorties standard et d'erreur peuvent être redirigées simultanément avec
`(commande > fichier) >& erreur`.



```
$ (grep root /etc/* > standard) >& erreur
```

Regroupement des deux sorties

La notation `commande >& fichier` redirige les sortie standard et sortie d'erreur vers une unique destination.

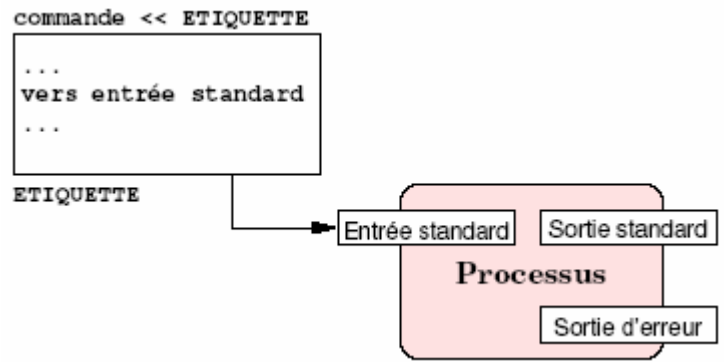


Il n'est plus possible de distinguer les deux sorties une fois le regroupement effectué.

```
$ grep root /etc/* >& sorties
```

Document en ligne

La redirection des documents en ligne permet d'automatiser des tâches interactives.



Variables du shell

Affectation d'une variable avec la notation

```
set variable=valeur
```

Lors d'une affectation encadrer les chaînes contenant des espaces par des guillemets ou des apostrophes.

Tableaux

Il existe des tableaux dont les cases sont remplies avec

```
set tableau=(valeur1 valeur2 valeur3)
```

et consultées avec l'expression

```
$tableau[$i]          (la première cellule porte l'indice 1)
$tableau              renvoie l'ensemble des éléments.
$tableau[$n-]        renvoie du énième jusqu'au dernier élément.
$tableau[-$n]        renvoie du premier jusqu'au énième élément.
```

L'expression \$#tableau correspond au nombre de cases dans le tableau.

NB : les tableaux n'existent que dans les shells modernes (Bash 2, Ksh 93).

Paramètres non-modifiables

\$0	Nom du script ou du shell
\$1 \$2... \$9 \${10}...	Paramètres positionnels (arguments ligne de commande)
\$#	Nombre de paramètres sauf \$0
\$*	Tous les paramètres sauf \$0
\$\$	PID du shell en cours
#!	PID du dernier processus lancé à l'arrière-plan
\$?	Code de retour de la dernière commande

\$* correspond à \$1 \$2 ... \$n

"\$*" correspond à "\$1 \$2 ... \$n"

Paramètres utilisables en lecture

PWD	Répertoire en cours
HOSTNAME	Nom de la machine hôte
	...

Paramètres de configuration

PATH	Chemin de recherche des commandes
HOME	Répertoire personnel de l'utilisateur
prompt	Symbole d'invite principal
prompt2, prompt3	Symboles d'invite supplémentaires
LANG	Langue de l'utilisateur
LC_ALL	Localisation générale

path, home, tty, uid, user, gid, group, VENDOR, MACHTYPE, OSTYPE, ...

Environnement d'un processus**À vous...**

```
$ set variable=abc
$ /bin/csh
[csh]$ echo $variable
```

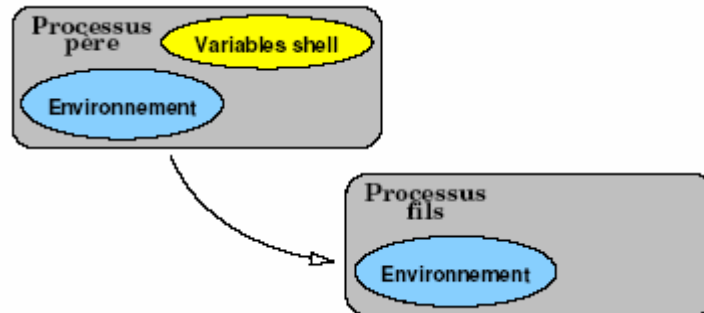
```
[csh]$ exit
$ echo $variable
```

```
$ setenv variable abc
$ /bin/csh
[csh]$ echo $variable
```

```
[csh]$ set variable=def
[csh]$ echo $variable
[csh]$ exit
$ echo $variable
```

```
$ /bin/csh
[csh]$ echo $variable
[csh]$ setenv variable def
[csh]$ echo $variable
[csh]$ exit
$ echo $variable
```

Pour qu'une variable du shell soit visible par ses futurs processus fils, il faut rendre accessible la variable dans l'environnement du père. Ce que l'on fait avec la commande `setenv`.



Le processus fils reçoit une copie de l'environnement de son père. Il ne peut en aucun cas modifier l'environnement de son père.

La commande `env(1)` sans argument affiche l'environnement du processus.

Commandes simples

Une commande simple du shell est décrite par le schéma suivant :

```
[affectations de variables] commande [arguments...] [redirections]
```

Chaque processus se terminant normalement sous Unix renvoie un code de retour, indiquant ses conditions de réussite ou d'échec.

Le code de retour est consultable dans le paramètre \$? du shell après terminaison de la commande.

À vous...

```
$ gvim
$ echo $LANG
$ set LANG=en_US
$ gvim
```

```
$ setenv LANG en_US
$ gvim
```

```
$ cd /etc
$ echo $?
```

```
$ cd /azertyuiop
$ echo $?
$ echo $?
```

```
$ cd /azertyuiop
$ echo $status
$ echo $status
```

```
$ grep root /etc/passwd  
$ echo $?
```

```
$ grep abcdefg /etc/passwd  
$ echo $?
```

```
$ grep root /etc/inexistant  
$ echo $?
```

Voir aussi :
Manuel Unix
– grep(1)

Pipelines

Les commandes simples peuvent être organisées en *pipelines* :

```
commande [ | commandes... ]
```

La sortie standard de chaque commande – sauf la dernière – est redirigée via un *tube* Unix vers l'entrée standard de la commande suivante.

Les sorties d'erreur ne sont pas concernées.

Le code de retour est celui de la dernière commande à se terminer (en principe la dernière du pipeline).

Alternative

```
commande_1 || commande_2
```

La seconde commande n'est exécutée que si la première a échoué.

Symétriquement l'échec d'une commande est indiqué par un code de retour **non-nul**.

Essais successifs :

```
cd /var/tmp || cd /usr/tmp || cd /tmp
```

Vérification d'erreur :

```
mount /mnt/cdrom || ( echo "CD absent du lecteur ?"; exit 1 )
```

Commandes composées

Les accolades permettent d'isoler une portion de liste de pipelines, en ayant priorité sur les opérateurs &, &&, || et ; se trouvant à l'extérieur des parenthèses.

```
mount /mnt/cdrom || ( echo "CD dans le lecteur ?" ; exit 1 )
```

PAGE BLANCHE !

PAGE BLANCHE !

Fonctions

Les fonctions permettent de regrouper des commandes que l'on exécutera ensuite avec une invocation simple.

Intérêts :

- appel depuis plusieurs emplacements du script,
- découpage du script en unités fonctionnelles simples,
- possibilité d'invocations itératives (récursivité).

SOUCI ! Le concept de fonction n'existe pas en csh. Un pis-aller ? la commande intégrée source :

```
source fichier-fonction [arguments]
```

Au sein du "fichier-fonction", les arguments sont alors disponibles dans les paramètres positionnels \$1, \$2, etc (ou via le tableau \$argv).

PAGE BLANCHE !

PAGE BLANCHE !

PAGE BLANCHE !

Sélections

Deux structures de contrôle pour les sélections :

Le test `if/then/else` permet d'organiser le déroulement du script en fonction du code de retour d'une commande exécutée.

La sélection `switch/case` compare une chaîne avec divers motifs et réagit en conséquence.

La commande `test` permet de vérifier des conditions sur des chaînes, des nombres ou des fichiers. (Voir aussi la puissante fonction intégrée `filetest`.)

La notation `! condition` permet d'inverser le résultat logique de la condition

ATTENTION, pour la structure `if`, la valeur associée à FAUX est 0 tandis que n'importe quelle valeur autre que 0 équivaut à VRAI.

Structure if-then-else

```
if (condition) then
commandes...
else if (condition) then
commandes...
else
commandes...
endif
```

Exemples de tests :

```
echo Invocation : $0 $argv[1]
if (`filetest -d $argv[1]`) then
echo $argv[1] existe et est bel et bien un dossier
else if (`filetest -f $argv[1]`) then
echo $argv[1] existe et est contre toute attente un fichier
else
echo -n "soit $argv[1] n'existe pas, "
echo "soit ce n'est ni un dossier ni meme un simple fichier"
endif
```

Exemples de tests (suite)

```
if (`ping -c 1 $nom_serveur >& /dev/null && echo 1 || echo
0`) then
echo "Serveur $nom_serveur injoignable"
endif
```

Réécriture ce qui précède :

```
ping -c 1 $nom_serveur >& /dev/null
if (! $status) then
echo "Serveur $nom_serveur injoignable"
endif
```

Une troisième formulation :

```
if (! { ping -c 1 $nom_serveur }) then
echo "Serveur $nom_serveur injoignable"
endif
```

```
test expression
[ expression ]
```

Expression	vraie si...
[! <i>condition</i>]	la <i>condition</i> est fausse
[<i>condition_1</i> -a <i>condition_2</i>]	les deux conditions sont vraies
[<i>condition_1</i> -o <i>condition_2</i>]	une au moins des conditions est vraie

Toutefois, on préfère en général les constructions :

- ! [*condition*]
- [*condition_1*] && [*condition_2*]
- [*condition_1*] || [*condition_2*]

qui sont plus lisibles.

Voir aussi :
Manuel Unix
- test(1)

Structure switch-case

```
switch (chaine)
case motif:
    action
    ...
    breaksw
case motif:
case motif:
case motif:
    action
    ...
    breaksw
default:
    action
    ...
    breaksw
endsw
```

Les motifs peuvent contenir des caractères génériques comme ? ou *.

Exemple

```
switch ($nom_fichier)
case *.gif:
case *.jpeg:
case *.tif{f,}:
    echo "Fichier graphique"
    breaksw
case *.txt:
case *.htm{,1}:
    echo "Fichier texte"
    breaksw
default:
    echo "Format de fichier inconnu..."
    echo "Continuer quand meme ?"
    set reponse=$<
    switch ($reponse)
    case [Oo]*:
    case [Yy]*:
        echo "Ok"
        breaksw
    default:
        exit 1
        breaksw
    endsw
    breaksw
endsw
```

Itérations

Il existe deux structures itératives :

La boucle `foreach` répète un bloc de code en donnant successivement à une variable toutes les valeurs d'une liste.

La boucle `while` répète une portion de script tant qu'une condition est vraie.

Les instructions `break` et `continue` permettent de modifier l'exécution d'une structure de boucle en provoquant une sortie ou une reprise prématurées.

Structure foreach

```
foreach variable (liste)
commandes...
end
```

À vous...

```
foreach F (im*)
  echo -n "Le fichier $F appartient a "
  ls -l $F | awk '{ print $3 }'
end
```

Ou encore :

```
foreach F (`find . -size +20000 -type f`)
  echo -n "Le gros fichier $F appartient a "
  ls -l $F | awk '{ print $3 }'
end
```

```
@ nl=1
foreach ligne ( " `cat $argv[1] ` " )
  printf "%5d  %s\n" $nl " $ligne "
  @ nl++
end
```

```
#!/bin/csh

if ($#argv == 2) then
  foreach fichier (*. $argv[1])
    mv $fichier `echo $fichier | sed 's/\.'$1'$/.'$2'/'`
  # OU mv $fichier `expr $fichier : '\(.*\)\.'$1`.$2
  # OU mv $fichier `basename $fichier .$1`.$2
  end
else
  echo Nombre d'argument(s) fourni : $#
  echo Usage correct : $0 suffixe_courant nouveau_suffixe
endif
```

Structure while

```
while (condition)
  commandes...
end
```

À vous...

```
@ cpt=10
while ($cpt)
  echo plus que $cpt seconde(s)
  sleep 1
  @ cpt--
end
```

```
@ i=1
while ($i <= 10)
  echo $i
  @ i++
end
```

```
while (1)
  date
  sleep 1
end
```

Voir aussi :
Manuel Unix
– sleep(1)

PAGE BLANCHE !

Ruptures de séquences

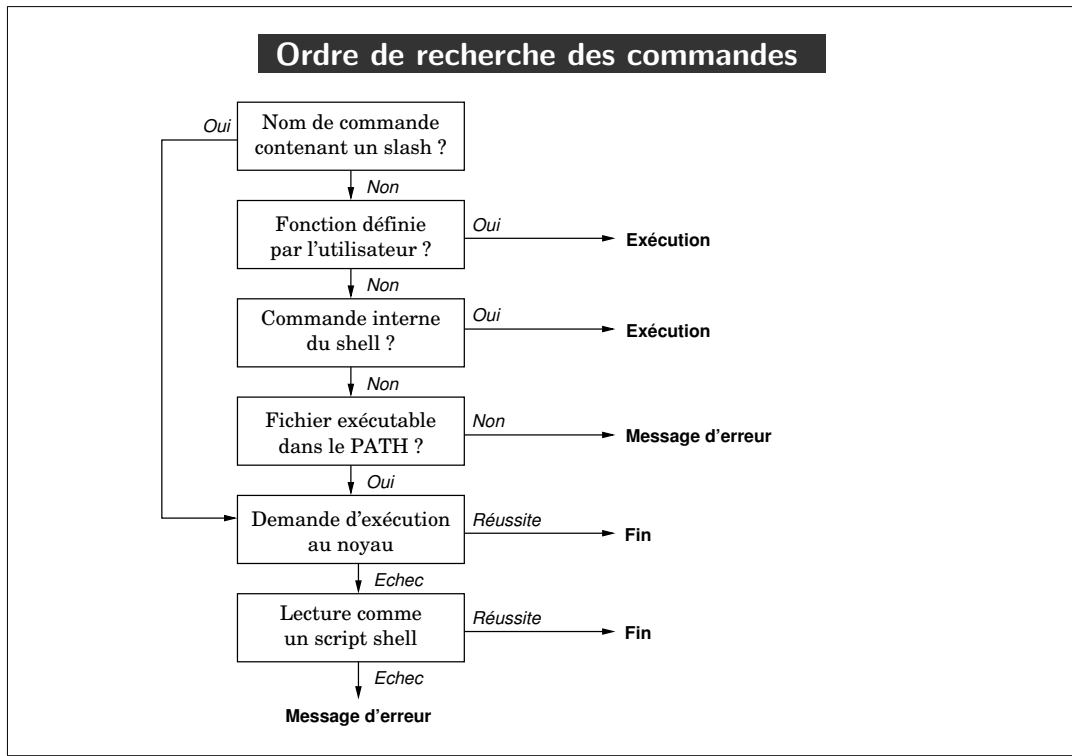
L'instruction `break` permet de sortir immédiatement de boucles `foreach` ou `while`.

L'instruction `continue` fait passer immédiatement à l'itération suivante d'une boucle `foreach` ou `while`.

Exemples...

```
while (1)
...
[ "$reponse" = "quitter" ] && break
end
```

```
foreach fichier (*)
[ ! -f "$fichier" ] && continue
...
end
```



	sh	csh	bash	ksh	SUSv3
.	*		*	*	*
:	*	*	*	*	*
[*	X	*	*	*
alias		*	*	*	*
bg		*	*	*	*
bind			*		
bindkey		*			
break	*	*	*	*	*
builtin			*	*	
cd	*	*	*	*	*
command			*	*	*
continue	*	*	*	*	*
declare			*		
echo	X	*	*	*	*
enable		X	*	X	
eval	*	*	*	*	*
exec	*	*	*	*	*
exit	*	*	*	*	*
export	*		*	*	*
false	X	X	X	*	*
fc			*	*	*
fg		*	*	*	*
getopts	*		*	*	*
hash	*		*	*	*
rehash		*			
jobs		*	*	*	*

* = commande interne, X = commande externe

	sh	csH	bash	ksh	SUSv3
kill	X	*	*	*	*
let			*	*	
local			*	*	
print				*	
printf		X	*	X	*
pwd	*	X	*	*	*
read			*	*	*
readonly	*		*	*	*
return	*		*	*	*
set		*	*	*	*
setenv		*			
shift	*	*	*	*	*
source		*	*		
test	*	X	*	*	*
filetest		*			
times	*		*	*	*
trap	*		*	*	*
true	X	X	X	*	*
type			*	*	*
typeset			*	*	
ulimit			*	*	*
unlimit		*			
limit		*			
umask	*	*	*	*	*
unalias		*	*	*	*
unset	*	*	*	*	*
wait		*	*	*	*
whence				*	

* = commande interne, X = commande externe

Les commandes `which` et `where` indiquent si leur argument est une fonction, un alias, une commande interne (builtin) ou une commande externe.

À vous...

```
$ which ls  
$ which echo
```

```
$ where echo
```

Entrées-sorties

Les entrées-sorties pour un script shell peuvent se faire avec diverses commandes.

Affichage de données :

- `echo` : affichage classique de message,
- `cat` : affichage du contenu d'un fichier ou document en-ligne.
- `printf` : mise en forme d'arguments numériques, chaînes, etc.

Saisie de données :

- `set VAR=$<` : lecture d'une seule valeur (délimitée par le premier blanc),
- `set VAR="$<"` : lecture d'une ligne de texte,
- `cat` : lecture d'un bloc de texte.

echo`echo [options] arguments...`

affiche les arguments séparés par un espace, avec un saut-de-ligne final.

Options		Séquences spéciales	
<code>-n</code>	pas de saut-de-ligne final	<code>\a</code>	sonnerie (<i>alarm</i>)
<code>-E</code>	ignorer les séquences spéciales	<code>\b</code>	retour en arrière (<i>backspace</i>)
<code>-e</code>	traiter les séquences spéciales	<code>\c</code>	pas de saut-de-ligne final
		<code>\f</code>	saut-de-page (<i>form feed</i>)
		<code>\n</code>	saut-de-ligne (<i>new line</i>)
		<code>\r</code>	début de ligne (<i>carriage return</i>)
		<code>\t</code>	tabulation horizontale
		<code>\v</code>	tabulation verticale
		<code>\\</code>	caractère <i>backslash</i>
		<code>\nnn</code>	caractère Ascii <i>nnn</i> (octal)

Les options de `echo` n'étant pas disponibles dans toutes les implémentations, il est parfois nécessaire de faire appel à `printf(1)`.

À vous...

```
$ echo abc    def
$ echo "abc   def"
```

```
$ echo -n "message"
```

```
$ echo "\007"
$ set echo_style=sysv
$ echo "\007"
$ set echo_style=bsd (ou set echo_style=none)
$ echo "\007"
```

```
$ set echo_style=both
$ while (1)
while? echo -n `date +%X`
while? sleep 1
while? echo "\r\c"
while? end
```

```
set
```

```
set VAR=$<
```

Lit une valeur depuis l'entrée standard.

```
set VAR="$<"
```

Lit une ligne depuis l'entrée standard.

PAGE BLANCHE !

PAGE BLANCHE !

cd

```
cd répertoire
```

`cd` modifie le répertoire de travail du processus pour celui indiqué en argument.

pwd

La commande `pwd` affiche le répertoire de travail du processus sur la sortie standard.

PWD

La variable `PWD` est mise à jour à chaque déplacement dans un répertoire.

La modification de la variable `PWD` est autorisée, mais elle ne change pas le répertoire de travail du processus.

Configuration du shell

set option

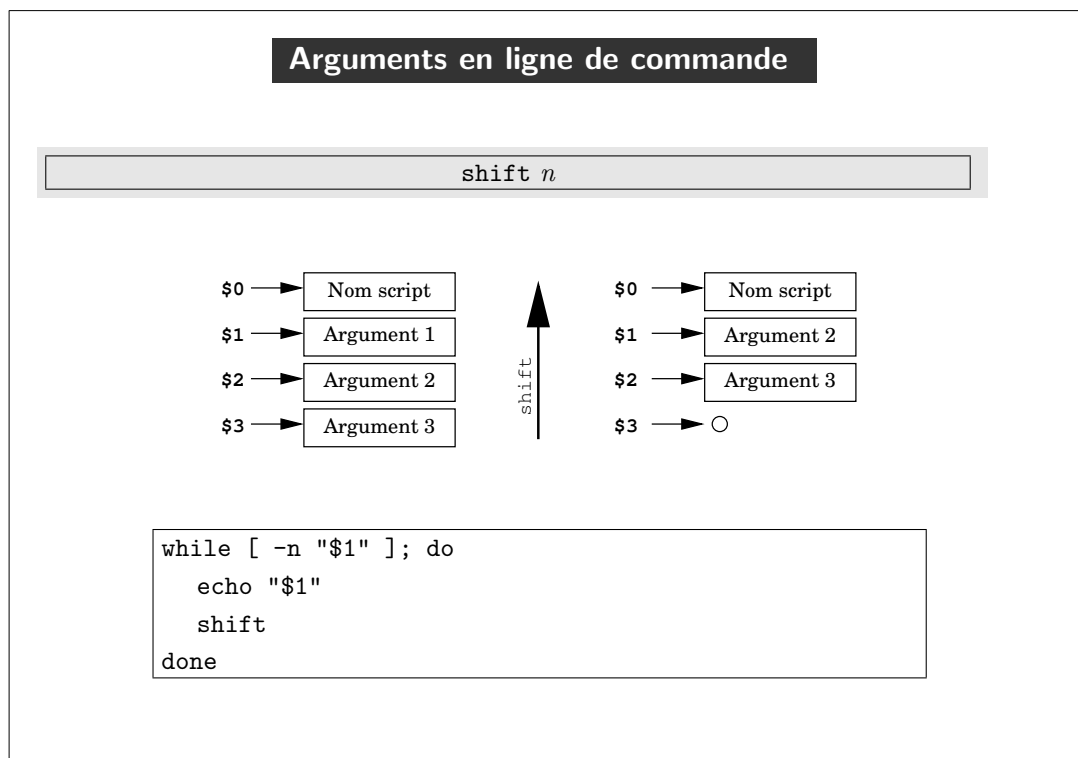
notify	signaler les terminaisons des jobs en arrière-plan
noclobber	ne pas écraser de fichier lors des redirections
noglob	ne pas développer les noms de fichiers
verbose	afficher les lignes de commande avant exécution

Exemple :

```
$ echo premier contenu > fichier  
$ echo autre contenu > fichier
```

```
$ set noclobber  
$ echo troisieme contenu > fichier
```

```
$ unset noclobber  
$ echo dernier contenu > fichier
```



Choisir avec soin les noms de variables

`fichier`, `repertoire`, `reponse`, `nb_lignes`, sont préférables à `f1`, `r`, `r1` ou `nl`.

Utiliser des noms détaillés pour les variables globales, utilisées à différents endroits du script.

Utiliser des noms brefs pour des variables employées localement dans des portions très courtes.

Habituellement, sous Unix, les variables contenant des paramètres fixes (chemin d'accès à l'application, nom de l'utilisateur, etc.) sont écrites en majuscules.

À la lecture, encadrer les variables par des guillemets

Pour protéger les éventuels espaces (noms de fichier, etc.)

Indenter le code

L'organisation logique du code doit être visible au premier coup d'œil.

Utiliser plutôt des tabulations que des espaces.

Les blocs de code dépendant de structures de contrôle (boucles, tests...) doivent être clairement identifiés.

Mettre les fonctions en évidence

Les définitions de fonctions se trouvent en début de script.

Chaque fonction doit commencer par un commentaire décrivant son rôle, ses arguments et sa valeur de retour.

Limiter avec le mot-clé `local` la portée des variables utilisées dans les fonctions.

```
oui_ou_non()
{
    # Affiche la question passée en argument. Attend une réponse.
    # Renvoie Vrai si c'est Oui, Faux si c'est Non, et recommence sinon.

    local rep

    while true
    do
        echo "$@"
        printf "O/N ? "
        read rep
        case "$rep" in
            [Oo]* ) return 0 ;;
            [Nn]* ) return 1 ;;
        esac
    done
}

[...]

if oui_ou_non "Voulez-vous quitter le programme"; then exit 0; fi
```

Commandes avancées pour script shell

Affichage formaté

`printf format arguments...`

met en forme les arguments en fonction des directives de conversion.

Conversion :

`%[attribut][largeur][.precision]indicateur`

Attributs		Indicateurs	
		diu	entier décimal
-	aligner à gauche	o	entier octal
+	afficher le signe	xX	entier hexadécimal
espace	espace pour +	feEgG	réel format décimal et scientifique
0	remplir avec des zéros	c	caractère
		s	chaîne
		%	caractère %

```
$ annee=2004
$ printf "%d %f\n" 2004 2004
2004 2004,000000
$ printf "%07d %X \n" $annee $annee
0002004 7D4
$ pi=3,14159264
$ printf "%f %e %g\n" $pi $pi $pi
3,141593 3,141593e+00 3,14159
$ printf "%.2f %.4f \n" $pi $pi
3,14 3,1416
$
```

PAGE BLANCHE !

PAGE BLANCHE !

Configuration du terminal

stty options...

configure le terminal connecté sur son entrée standard.

Quelques options courantes :

- `stty -a` : affiche la configuration courante
- `stty sane` : ramène le terminal dans son état par défaut
- `stty -echo` : supprime l'écho des caractères saisis (mot de passe par exemple)
- `stty echo` : restitue l'écho des caractères
- `stty -icanon min 0 time 1` : permet de lire les touches à la volée (sans presser *Entrée*)

`stty` est normalisé par *SUSv3*.

```
#!/bin/sh
table[0]='|'
table[1]='/'
table[2]='-'
table[3]='\'
i=0
touche=""
echo "pressez 'q' pour quitter"
echo
stty -echo -icanon time 1 min 0
while [ "$REPLY" != "q" ] ; do
    read
    if [ -n "$REPLY" ]; then touche=$REPLY; fi
    echo -ne "\r${table[$i]} Dernière touche : ${touche}"
    i=$(( (i+1) % 4))
done
echo
stty sane
```

En début de ligne, une barre « tourne » sur elle-même, indiquant que le programme se déroule et capture les caractères au vol.

```
$ ./script.sh
pressez 'q' pour quitter
/ Dernière touche : a
```

parler de onintr [- | label]

F.A.Q.

On trouvera ici la réponse à des questions posées par des stagiaires au cours de différentes sessions de la formation *Programmation Shell et Langages de Scripts*.

Certains problèmes sont récurrents, et les réponses pourront éventuellement servir dans d'autres situations.

Comment convertir en majuscules une chaîne de caractères contenue dans une variable ?

```
chaine=$(echo "$chaine" | tr '[:lower:]' '[:upper:]')
```

On peut aussi invoquer Awk, comme dans la réponse suivante.

Peut-on passer en minuscules tous les noms de fichiers d'un répertoire ?

```
for fic in * ; do
  mv $fic $(echo $fic | awk '{print tolower($0)}')
done
```

Comment arrondir une valeur réelle à l'entier le plus proche ?

```
entier=$(echo "($reel + 0.5)/1" | bc)
```

La division par 1 force bc à recalculer les décimales.

Quel est le répertoire dans lequel se trouve le script en cours d'exécution ?

```
# ! /bin/sh
REP=$(eval echo $(echo $(dirname $0)/ | sed 's/^\.\.?\\/$PWD\\/\0/'))
```

Comment lire de manière efficace la réponse à une question ?

```
oui_ou_non ()
{ # Cette routine renvoie Vrai ou Faux suivant la réponse à
  # la question fournie en argument.
  while true ; do
    echo -n "$@" " >" >&2
    if ! read < /dev/tty; then return 1 ; fi
    case $REPLY in
      [oOyY]* ) return 0 ;;
      [nN]* ) return 1 ;;
    esac
  done
}
```

```
echo Usage : $0 $argv[1]

if (`filetest -d $argv[1]`) then
    echo $argv[1] existe et est bel et bien un dossier
else if (`filetest -f $argv[1]`) then
    echo $argv[1] existe et est contre toute attente un fichier
else
    set res=`filetest -l $argv[1]`
    if (! $res) then
        if (! { test -b $argv[1] }) then
            if (-c $argv[1]) then
                echo -n $argv[1] est en fait un fichier
                echo " special en mode caractere"
            else
                echo "soit $argv[1] n'existe pas, soit ce n'est ni un
dossier ni meme un simple fichier"
            endif
        else
            echo $argv[1] est un fichier special en mode bloc
        endif
    else
        echo $argv[1] existe et est un lien symbolique
    endif
endif

@ taille= -Z $argv[1]
@ nombreLiensEnDur= -N $argv[1]
echo $argv[1] fait $taille octets et possede $nombreLiensEnDur au total
```