

Christophe Blaess

Fonctionnement du shell

Ecriture de scripts shell

Ce document a été réalisé sur système Linux avec les logiciels libres :

- **Formation+** pour préparer le cours complet,
- **L^AT_EX** pour produire les transparents et le support de cours imprimé,
- **Xfig** pour les graphiques vectoriels,
- **The Gimp** pour les images et les photos.

Support de cours : version 17

© CHRISTOPHE BLAESS 2001-2007 – Tous droits réservés

Aucune partie de ce cours ne peut être reproduite ou transmise à quelque fin ou par quelque moyen que ce soit, électronique ou mécanique, sans l'autorisation expresse et écrite de l'auteur.

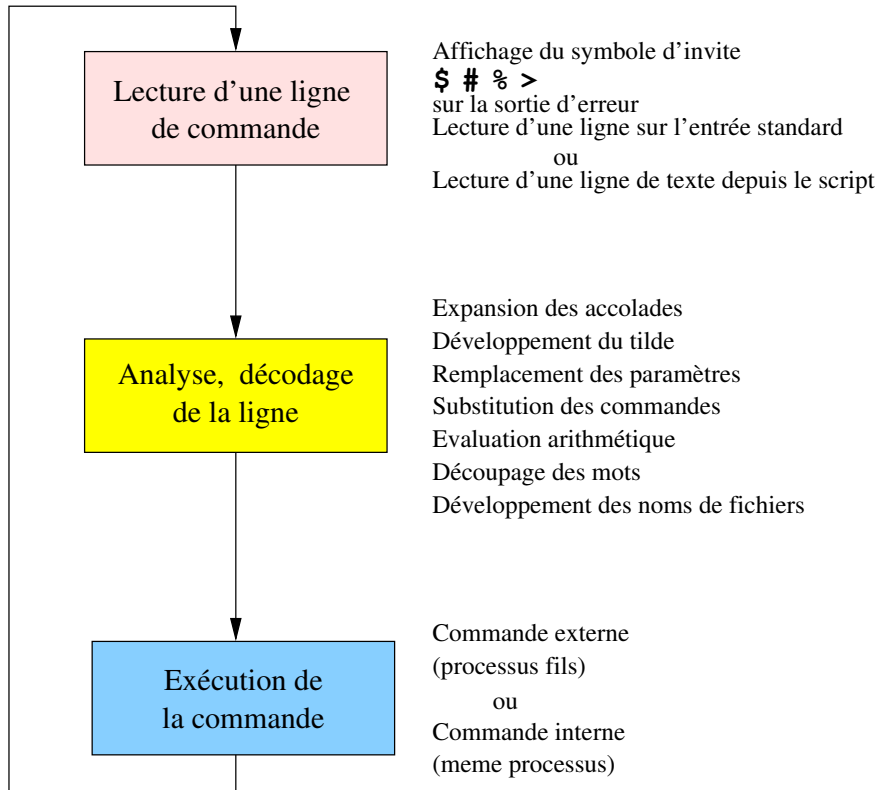
*Ecriture de scripts shell***Fonctionnement
du shell**

Christophe Blaess

– Interprétation d’une commande	3
– Boucle générale du shell	3
– Expansion des accolades	4
– Développement du tilde	6
– Remplacement des paramètres	8
– Substitution de commandes	15
– Evaluation arithmétique	17
– Protections et découpage en mots	19
– Développement des noms de fichiers	23
– Travaux pratiques	25
– Redirections des entrées-sorties	26
– Entrées-sorties standards d’un processus	26
– Redirections de la sortie standard	27
– Redirections de la sortie d’erreur	28
– Redirections des deux sorties	29
– Regroupement des deux sorties	30
– Redirections de l’entrée standard	31
– Tubes entre processus	32
– Document en ligne	33
– Variables et environnement	34
– Variables du shell	34
– Tableaux	35
– Paramètres non-modifiables	36
– Paramètres utilisables en lecture	37
– Paramètres de configuration	37
– Environnement d’un processus	37
– Travaux pratiques	39

Interprétation d'une commande

Boucle générale du shell



Expansion des accolades

A vous...

Que donnent les lignes suivantes :

```
$ echo image{05,06,07}.gif
```

```
$ echo image{01}.gif
```

```
debut_{mot1,mot2,mot3}_fin
est développé ainsi :
debut_mot1_fin debut_mot2_fin debut_mot3_fin
```

Si le shell ne trouve pas de virgule entre les accolades, il n'effectue pas de modification.

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 ~user02/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.

`/etc/passwd`

```

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:./:/sbin/nologin
...
user01:x:501:501:Utilisateur TP 01:/home/user01:/bin/bash
user02:x:502:502:Utilisateur TP 02:/home/user02:/bin/bash
...

```

Remplacement des paramètres

A vous...

Qu'affichent les lignes suivantes :

```
$ a = azert (avec des espaces autour du =)
```

```
$ 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}`.

Lors d'une affectation de variable ne jamais mettre d'espace autour du signe `=`.

A vous...

```
$ a=azert  
$ echo ${a-défaut}
```

```
$ echo ${inexistante-défaut}
```

```
$ vide=""  
$ echo ${vide-défaut}
```

```
$ echo ${vide:-défaut}
```

```
$ echo ${vide:=défaut}  
$ echo $vide
```

Le `$` est un véritable opérateur qui offre de multiples possibilités.

`${variable-défaut}`

est remplacé par `défaut` si la `variable` n'existe pas.

`${variable:-défaut}`

effectue aussi le remplacement par `défaut` si la `variable` contient une chaîne vide.

`${variable=défaut}`

remplit la `variable` avec la valeur par `défaut` si elle n'existait pas ; fournit le contenu de la `variable`.

`${variable:=défaut}`

remplit aussi la `variable` si elle contient une chaîne vide.

Il existe aussi des opérateurs `${var+val}`, `${var:+val}`, `${var?val}` et `${var:?val}` rarement utilisés.

A vous...

```
$ a=azertyuiop  
$ echo ${#a}
```

```
$ echo ${a#aze}
```

```
$ echo ${a%iop}
```

```
$ echo ${a:3:4}
```

```
$ echo ${a:4}
```

`${#variable}`

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

`${variable#prefixe}`

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

`${variable%suffixe}`

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

`${variable:debut:longueur}`

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

Les shells Korn antérieurs à *Ksh 93* n'offrent pas l'extraction de sous-chaîne.

Utilisation avancée

Le préfixe ou le suffixe éliminés peuvent contenir des caractères génériques du shell, comme ? ou *.

`${variable#motif}`

supprime le **préfixe** le plus **court** correspondant au **motif**.

`${variable##motif}`

supprime le **préfixe** le plus **long** correspondant au **motif**.

`${variable%motif}`

supprime le **suffixe** le plus **court** correspondant au **motif**.

`${variable%%motif}`

supprime le **suffixe** le plus **long** correspondant au **motif**.

Substitution de commandes

Qu'affichent les lignes suivantes :

```
$ a=$(date)
$ echo $a
```

```
$ a=$(date +%Y)
$ echo $a
```

```
$ a=$(ls)
$ echo $a
```

```
$ a=$(ls -l)
$ echo $a
$ echo "$a"
```

Ancienne construction :
 'commande'
Nouvelle construction :
 \$(commande)

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

L'éventuel saut de ligne final de la sortie standard est éliminé lors de la substitution de commande

L'ancienne construction est plutôt déconseillée pour deux raisons :

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

Le shell Bourne original n'accepte que la construction 'commande'.

Evaluation arithmétique

Ancienne construction (obsolète) : `$(expression)`

Nouvelle construction : `=$((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.

A vous. . .

```
$ echo=$((12 + 3*(5+3)*5))
```

```
$ a=10  
$ a=$((a+1))  
$ echo $a
```

```
$ echo=$((3.5))  
$ echo=$((010))  
$ echo=$((08))
```

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

Dans les expressions arithmétiques, les variables peuvent être préfixées de l'opérateur `$`, mais ce n'est pas obligatoire.

Les parenthèses ajoutent des priorités dans les expressions.

Les constantes sont :

- décimales par défaut : 9, 45, 13600...
- octales si elles commencent par zéro : 0644, 0755...
- hexadécimales si elles commencent par 0x : 0x4C...
- en base n si elles commencent par $n\#$: 2#10101010, 10#014...

NB : Des affectations de variables sont possibles au sein de l'expression arithmétique, mais on perd de la lisibilité : `i=$((j=5))`.

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

Opérateurs arithmétiques

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

- Le shell Korn 93 accepte les valeurs réelles (extension par rapport au standard SUSv3).
- Le shell Korn 88 arrondit les valeurs réelles à l'entier inférieur.
- Les shells Korn (88 et 93) refusent les valeurs hexadécimales, et ne reconnaissent pas les constantes octales. C'est un écart par rapport à SUSv3.

Protections et découpage en mots**A vous...**

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

```
$ echo "$(ls)"  
$ echo '$(ls)'
```

```
$ echo "$((1+2))"  
$ echo '$((1+2))'
```

```
$ 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
- évaluation arithmétique
- 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.

A vous...

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

```
$ a=un fichier
```

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

Les mots de la ligne sont séparés en utilisant comme délimiteurs les caractères contenus dans la variable d'environnement `IFS`.

Par défaut, `IFS` contient les caractères :

- espace,
- tabulation,
- saut de ligne.

Les caractères séparateurs précédés par un *backslash* `\` ne sont pas pris en compte, mais le *backslash* est supprimé.

Les portions de ligne encadrés par des guillemets ou des apostrophes ne sont pas découpées.

En raison de diverses problèmes de sécurité, la variable `IFS` est réinitialisée à chaque fois qu'un shell démarre.

```
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
```

Développement des noms de fichiers

A vous...

```
$ echo /dev/tty?
```

```
$ echo /dev/tty?0
```

```
$ echo /dev/tty*
```

```
$ echo "/dev/tty*"
```

```
$ echo /dev/ttyZZ*
```

```
$ ls /dev/ttyZZ*
```

```
$ ls "/dev/tty*"
```

Les mots contenant des caractères génériques *, ? [] sont considérés comme des motifs, et remplacés par la liste des fichiers correspondants.

- ? : n'importe quel caractère (sauf le *slash* / et le point initial) ;
- * : n'importe quelle chaîne – éventuellement vide – de caractères (sauf / et . initial) ;
- [] : liste de caractères.

Si aucun nom de fichier ne correspond au motif, le mot n'est pas modifié.

Le développement des noms de fichiers n'a pas lieu dans les chaînes entre guillemets ou apostrophes.

Travaux pratiques**Exercice numéro 1**

Manipulation des variables

Exercice numéro 2

Syntaxe d'affectation de variable (1)

Exercice numéro 3

Syntaxe d'affectation de variable (2)

Exercice numéro 4

Protection des expressions

Exercice numéro 5

Protection du contenu d'une variable

Exercice numéro 6

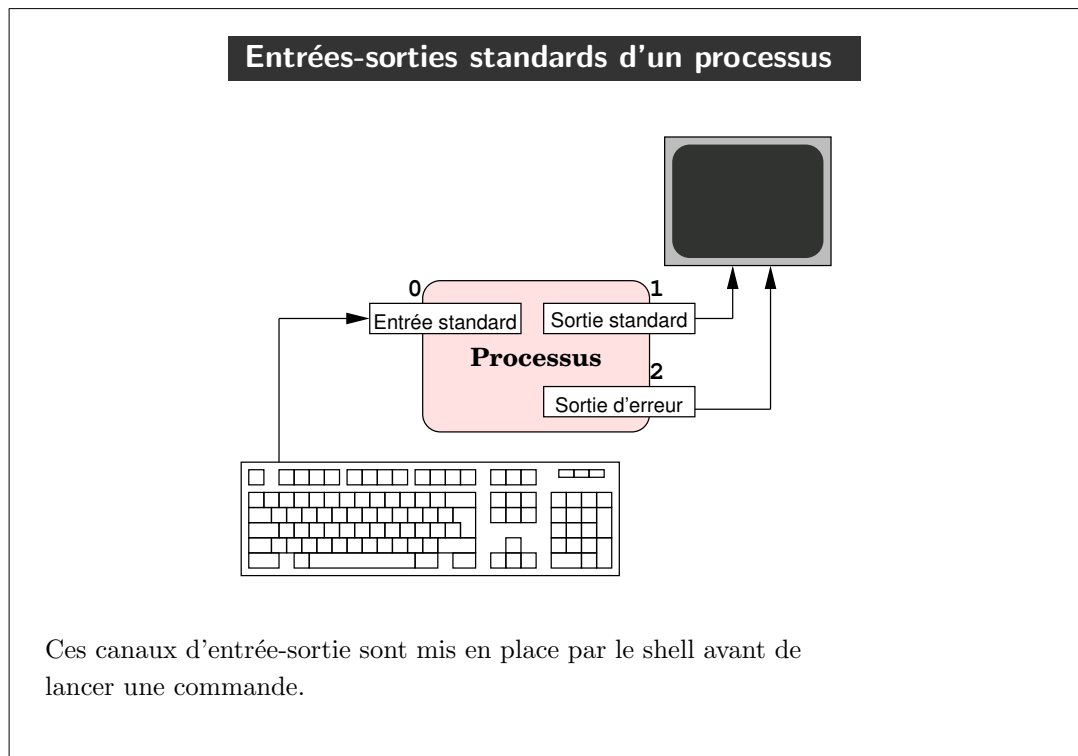
Développement des caractères génériques

Exercice numéro 7

Invocation de commande et évaluation arithmétique

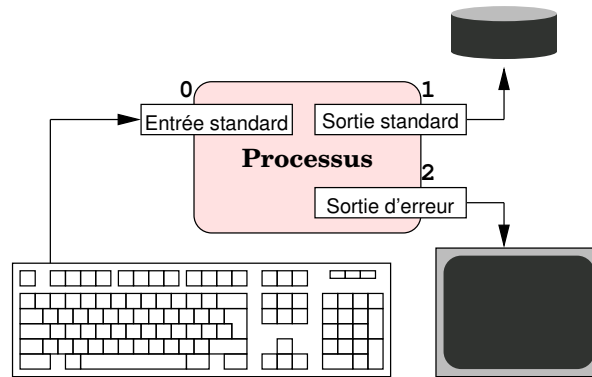
Exercice numéro 8

Manipulation du contenu d'une variable



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.

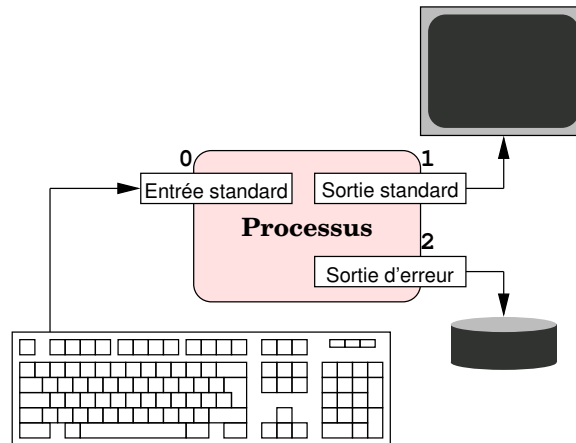


La redirection `commande >> fichier` fait un **ajout** en fin de fichier sans écrasement des données précédentes.

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

Redirections de la sortie d'erreur

La notation `commande 2> fichier` redirige la sortie d'erreur du processus. Et `commande 2>> fichier` ajoute en fin de fichier.

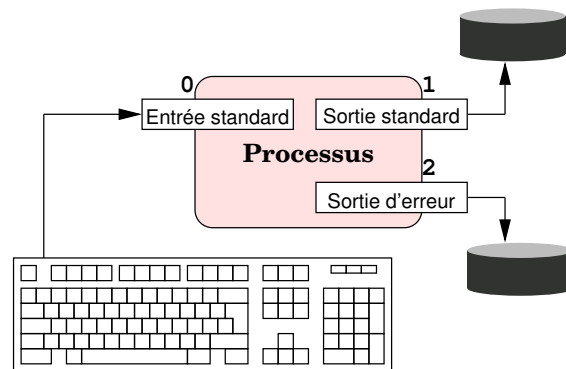


On utilise souvent commande `2>/dev/null`

```
$ grep root /etc/*  
$ grep root /etc/* 2> erreur  
$ grep root /etc/* 2> /dev/null
```

Redirections des deux sorties

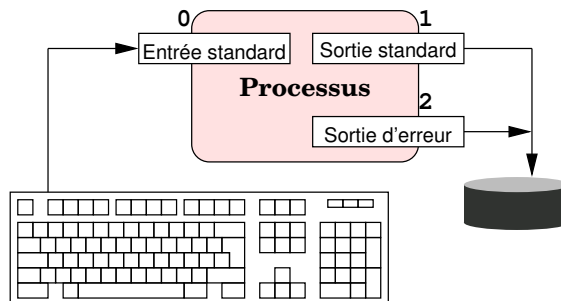
Les sorties standard et d'erreur peuvent être redirigées simultanément avec `commande > fichier 2> erreur`.



```
$ grep root /etc/* > standard 2> erreur
```

Regroupement des deux sorties

La notation `commande > fichier 2>&1` redirige la sortie standard puis recopie sa configuration sur celle de la sortie d'erreur.



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

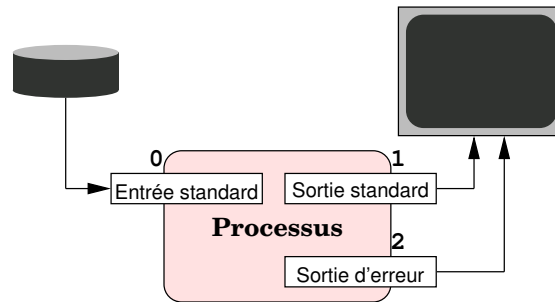
On peut réaliser le regroupement inverse `1>&2` comme dans

```
echo "Message d'erreur" >&2
```

```
$ grep root /etc/* > sorties 2>&1
```

Redirections de l'entrée standard

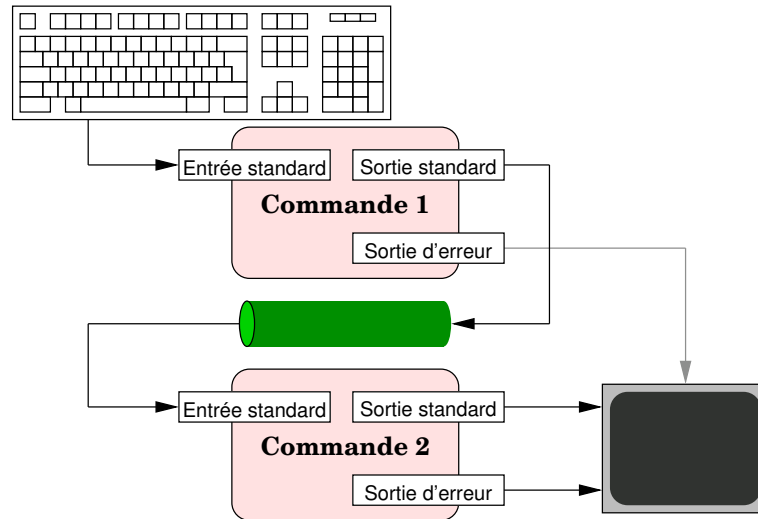
Avec la notation `commande < fichier` l'entrée standard est lue dans le fichier.



La redirection est transparente pour le processus, il ne voit pas de différence par rapport au terminal.

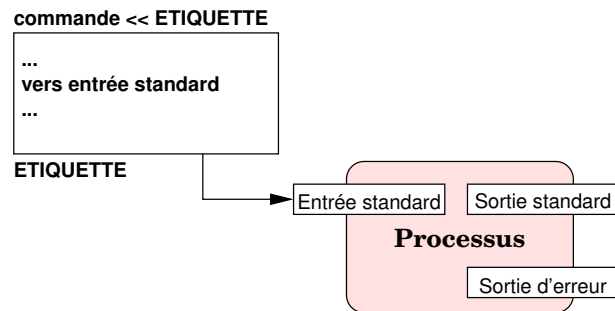
Tubes entre processus

Le symbole *pipe* dans `commande_1 | commande_2` crée un tube entre la sortie standard de la première commande et l'entrée standard de la seconde.



Document en ligne

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



Avec la redirection `commande <<- ETIQUETTE` les tabulations en début de ligne sont ignorées dans le document en ligne.

```
/tp/04_commandes/creation_base_mysql.sh
```

```
#!/bin/sh
mysql <<-FIN
  DROP DATABASE IF EXISTS geo;
  CREATE DATABASE geo;
  USE geo;
  CREATE TABLE depts (
    num   varchar(2),
    nom   varchar(50),
    pref  varchar(50),
    region varchar(50)
  );
  GRANT SELECT ON geo.depts TO geo;
FIN
[...]
```

Variables du shell

Affectation d'une variable avec la notation
`variable=valeur`

Pas d'espace autour du signe égal!

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
`tableau[$i]=valeur`
et consultées avec l'expression
`${tableau[$i]}`

Le rang *zéro* correspond la variable sans indice :
`$tableau` est équivalent à `${tableau[0]}`.

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

\$* et \$@ correspondent tout deux à \$1 \$2 ... \$n

"\$*" correspond à "\$1 \$2 ... \$n"

"\$@" correspond à "\$1" "\$2" ... "\$n"

En conclusion, il faut toujours utiliser "\$@".

Paramètres utilisables en lecture

PWD	Répertoire en cours
REPLY	Chaîne de réponse de la commande <code>read</code>
RANDOM	Valeur aléatoire
SECONDS	Durée d'exécution du shell
OPTARG	Dernier argument traité par <code>getopts</code>
OPTIND	Prochain argument à traiter par <code>getopts</code>
HOSTNAME	Nom de la machine hôte ...

Paramètres de configuration

IFS	Séparateurs de mots
PATH	Chemin de recherche des commandes
HOME	Répertoire personnel de l'utilisateur
PS1	Symbole d'invite principal
PS2, PS3, PS4	Symboles d'invite supplémentaires
LANG	Langue de l'utilisateur
LC_ALL	Localisation générale ...

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

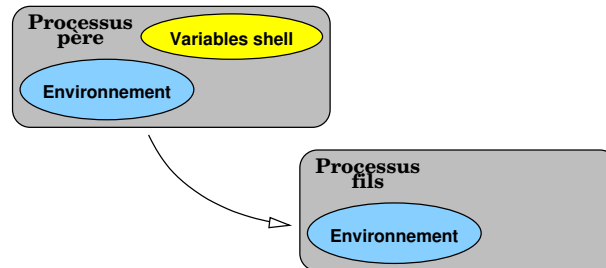
```
$ variable=abc
$ /bin/sh
[sh]$ echo $variable
```

```
[sh]$ exit
$ echo $variable
```

```
$ export variable
$ /bin/sh
[sh]$ echo $variable
```

```
[sh]$ variable=def
[sh]$ echo $variable
[sh]$ exit
$ echo $variable
```

Pour qu'une variable du shell soit visible par ses futurs processus fils, il faut *exporter* la variable dans l'*environnement* du père.



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.

Travaux pratiques**Exercice numéro 9**

Redirection des sorties standard et d'erreur

Exercice numéro 10

Aspects avancés des redirections (1)

Exercice numéro 11

Aspects avancés des redirections (2)

Exercice numéro 12

Aspects avancés des redirections (3)

Exercice numéro 13

Aspects avancés des redirections (4)

Exercice numéro 14

Paramètres de la ligne de commande

Exercice numéro 15

Extraction de préfixes et suffixes

Exercice numéro 16

Développement des arguments en ligne de commande

Index

\$! (paramètre)	36	env(1)	38
\$# (paramètre)	36	environnement	37, 38
\$\$ (paramètre)	36	export (commande)	37, 38
\$ ()	15, 16	expr(1)	18
\$(())	17, 18	guillemets	20, 21, 22
\$* (paramètre)	36	HOME (variable)	7, 36
\$0 (paramètre)	36	HOSTNAME (paramètre)	36
\$1 (paramètre)	36	IFS (variable)	22, 36, 36
\$? (paramètre)	36	LANG (variable)	36
\$@ (paramètre)	36	LC_ALL (variable)	36
\$[]	17	OPTARG (paramètre)	36
\${ }	8, 9, 10, 11, 12, 13, 14	OPTIND (paramètre)	36
* (caractère générique)	23, 24	PATH (variable)	36
/dev/null (fichier)	28	préfixe (extraction)	13, 14
/etc/passwd (fichier)	7	PS1 (variable)	36
2>	28	PS2 (variable)	36
2>&1	30	PS3 (variable)	36
2>>	28	PWD (paramètre)	36
<	31	RANDOM (paramètre)	36
<<	33	REPLY (paramètre)	36
>	27	SECONDS (paramètre)	36
>>	27	sortie d'erreur	26, 28, 29, 30
? (caractère générique)	23, 24	sortie standard	16, 26, 27, 29, 30, 32
‘ ’	16	suffixe (extraction)	13, 14
accolades	4, 5	tableaux (variables)	35
apostrophes	20, 21, 22	tilde	6, 7
arithmétique (évaluation)	17, 18	tube	32
Bourne (shell)	5, 16, 18	variable	8, 9, 10, 11, 12, 13, 14, 34
commande (substitution)	15, 16	{ }	4, 5
document en-ligne	33		32
entrée standard	26, 31, 32, 33		