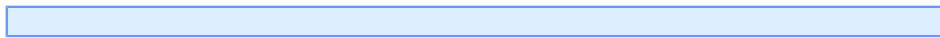


Introduction à la programmation en Bash

par [Eric Sanchis](#)

Date de publication : 01/02/2008

Dernière mise à jour :



- 0 - Préface
 - 0-1 - Pour en savoir plus :
- I - Introduction à bash
 - I-1 - Les shells
 - I-1-1 - Un environnement de travail
 - I-1-2 - Un langage de programmation
 - I-1-3 - Atouts et inconvénients des shells
 - I-1-4 - Shell utilisé
 - I-2 - Syntaxe d'une commande
 - I-3 - Commandes internes et externes
 - I-3-1 - Commandes internes
 - Commandes simples
 - Commandes composées
 - I-3-2 - Commandes externes
 - I-4 - Modes d'exécution d'une commande
 - I-4-1 - Exécution séquentielle
 - I-4-2 - Exécution en arrière-plan
 - I-5 - Commentaires
 - I-6 - Fichiers shell
- II - Substitution de paramètres
 - II-1 - Variables
 - II-1-1 - Affectation directe
 - II-1-2 - Affectation par lecture
 - II-1-3 - Variable en « lecture seule »
 - II-2 - Paramètres de position et paramètres spéciaux
 - II-2-1 - Paramètres de position
 - II-2-2 - Paramètres spéciaux
 - II-2-3 - Commande interne shift
 - II-2-4 - Paramètres de position et fichiers shell
 - II-2-5 - Paramètres spéciaux * et @
 - II-3 - Suppression des ambiguïtés
 - II-4 - Paramètres non définis
 - II-5 - Suppression de variables
- III - Substitution de commandes
 - III-1 - Présentation
 - III-2 - Substitutions de commandes et paramètres régionaux
- IV - Caractères et expressions génériques
 - IV-1 - Caractères génériques
 - IV-2 - Expressions génériques
- V - Redirections élémentaires
 - V-1 - Descripteurs de fichiers
 - V-2 - Redirections élémentaires
 - V-3 - Tubes
- VI - Groupement de commandes
- VII - Code de retour
 - VII-1 - Paramètre spécial ?
 - VII-2 - Code de retour d'un programme shell
 - VII-3 - Commande interne exit
 - VII-4 - Code de retour d'une suite de commandes
 - VII-5 - Résultats et code de retour
 - VII-6 - Opérateurs && et || sur les codes de retour
- VIII - Structures de contrôle case et while
 - VIII-1 - Choix multiple case
 - VIII-2 - Itération while

IX - Chaînes de caractères

- IX-1 - Protection de caractères
- IX-2 - Longueur d'une chaîne de caractères
- IX-3 - Modificateurs de chaînes
- IX-4 - Extraction de sous-chaînes
- IX-5 - Remplacement de sous-chaînes

X - Structures de contrôle for et if

- X-1 - Itération for
- X-2 - Choix if

XI - Entiers et expressions arithmétiques

- XI-1 - Variables de type entier
- XI-2 - Commande interne ((
- XI-3 - Valeur d'une expression arithmétique
- XI-4 - Opérateurs
- XI-5 - Structure for pour les expressions arithmétiques
- XI-6 - Exemple : les tours de Hanoi

XII - Tableaux

- XII-1 - Définition et initialisation d'un tableau
- XII-2 - Valeur d'un élément d'un tableau
- XII-3 - Caractéristiques d'un tableau
- XII-4 - Suppression d'un tableau

XIII - Alias

- XIII-1 - Création d'un alias
- XIII-2 - Suppression d'un alias

XIV - Fonctions shell

- XIV-1 - Définition d'une fonction
- XIV-2 - Suppression d'une fonction
- XIV-3 - Trace des appels aux fonctions
- XIV-4 - Arguments d'une fonction
- XIV-5 - Variables locales à une fonction
- XIV-6 - Exporter une fonction
- XIV-7 - Commande interne return
- XIV-8 - Substitution de fonction
- XIV-9 - Fonctions récursives
- XIV-10 - Appels de fonctions dispersées dans plusieurs fichiers

0 - Préface

Interpréteur de commandes par défaut des systèmes GNU/Linux, **bash** est devenu pour les administrateurs système, un outil incontournable. Ce document présente les principales constructions syntaxiques de bash utilisées dans l'écriture des programmes shell (scripts shell). L'objectif premier a été de laisser de côté les redondances syntaxiques de ce langage de programmation, la subtilité des mécanismes de l'interpréteur, afin d'insister sur quelques concepts synthétiques tels que la *substitution*, la *redirection* ou le *filtrage*.

Cet écrit étant destiné principalement aux étudiants de premier et deuxième cycle en informatique, j'ai choisi de le rédiger en adoptant un style 2Ex (1 Explication, 1 Exemple) qui devrait permettre au lecteur de mieux s'approprier chaque notion présentée. Ce dernier pourra ensuite aborder des publications plus extensives ou plus spécialisées.

Cette publication étant loin d'être parfaite¹, j'encourage le lecteur à me faire parvenir ses remarques ou suggestions, ainsi qu'à me signaler toute inexactitude (sanchis@iut-rodez.fr).

0-1 - Pour en savoir plus :

- Mendel Cooper, Advanced Bash Scripting Guide (<http://tldp.org/LDP/abs/html>). Une traduction en français est disponible (<http://abs.traduc.org>).
- Arnold Robbins, Nelson H. F. Beebe, Introduction aux scripts shell, Ed. O'Reilly, Paris, 2005.
- Cameron Newham, Bill Rosenblatt, Le shell bash, 3ème édition, Ed. O'Reilly, Paris, 2006.

I - Introduction à bash

I-1 - Les shells

Sous Unix, on appelle shell l'interpréteur de commandes qui fait office d'interface entre l'utilisateur et le système d'exploitation. Les shells sont des interpréteurs : cela signifie que chaque commande saisie par l'utilisateur (ou lue à partir d'un fichier) est syntaxiquement vérifiée puis exécutée.

Il existe de nombreux shells qui se classent en deux grandes familles :

- la famille C shell (ex : csh, tcsh)
- la famille Bourne shell (ex : sh, bash, ksh).

zsh est un shell qui contient les caractéristiques des deux familles précédentes. Néanmoins, le choix d'utiliser un shell plutôt qu'un autre est essentiellement une affaire de préférence personnelle ou de circonstance. En connaître un, permet d'accéder aisément aux autres. Lorsque l'on utilise le système GNU/Linux (un des nombreux systèmes de la galaxie Unix), le shell par défaut est bash (Bourne Again SHell). Ce dernier a été conçu en 1988 par Brian Fox dans le cadre du projet GNU. Aujourd'hui, les développements de bash sont menés par Chet Ramey.

Un shell possède un double aspect :

- un aspect environnement de travail
- un aspect langage de programmation.

I-1-1 - Un environnement de travail

En premier lieu, un shell doit fournir un environnement de travail agréable et puissant. Par exemple, bash permet (entre autres) :

- le rappel des commandes précédentes (gestion de l'historique) ; cela évite de taper plusieurs fois la même commande
- la modification en ligne du texte de la commande courante (ajout, retrait, remplacement de caractères) en utilisant les commandes d'édition de l'éditeur de texte vi ou emacs
- la gestion des travaux lancés en arrière-plan (appelés jobs) ; ceux-ci peuvent être démarrés, stoppés ou repris suivant les besoins
- l'initialisation adéquate de variables de configuration (chaîne d'appel de l'interpréteur, chemins de recherche par défaut) ou la création de raccourcis de commandes (commande alias).

Illustrons cet ajustement de configuration par un exemple. Le shell permet d'exécuter une commande en mode interactif ou bien par l'intermédiaire de fichiers de commandes (scripts). En mode interactif, bash affiche à l'écran une chaîne d'appel (appelée également prompt ou invite), qui se termine par défaut par le caractère # suivi d'un caractère espace pour l'administrateur système (utilisateur root) et par le caractère \$ suivi d'un caractère espace pour les autres utilisateurs. Cette chaîne d'appel peut être relativement longue.

```
sanchis@jade:/bin$
```

Celle-ci est constituée du nom de connexion de l'utilisateur (sanchis), du nom de la machine sur laquelle l'utilisateur travaille (jade) et du chemin absolu du répertoire courant de l'utilisateur (/bin). Elle indique que le shell attend que

l'utilisateur saisisse une commande et la valide en appuyant sur la touche entrée. Bash exécute alors la commande puis réaffiche la chaîne d'appel. Si l'on souhaite raccourcir cette chaîne d'appel, il suffit de modifier la valeur de la variable prédéfinie du shell PS1 (Prompt Shell 1).

```
sanchis@jade:/bin$ PS1='$ '
$ pwd
/home/sanchis => pwd affiche le chemin absolu du répertoire courant
$
```

La nouvelle chaîne d'appel est constituée par le caractère \$ suivi d'un caractère espace.

I-1-2 - Un langage de programmation

Les shells ne sont pas seulement des interpréteurs de commandes mais également de véritables langages de programmation. Un shell comme bash intègre :

- les notions de variable, d'opérateur arithmétique, de structure de contrôle, de fonction, présentes dans tout langage de programmation classique, mais aussi
- des opérateurs spécifiques (ex : |, ;)

```
$ a=5    => affectation de la valeur 5 à la variable a
$
$ echo $((a + 3 )) => affiche la valeur de l'expression a+3
8
$
```

L'opérateur |, appelé tube, est un opérateur caractéristique des shells et connecte la sortie d'une commande à l'entrée de la commande suivante.

```
$ ruptime
iutbis      up
jade        up
mobile1     up
mobile2     up
quartz      up
sigma       up
$
$ ruptime | wc -l
6
$
```

La commande unix ruptime affiche les noms et autres informations relatives aux machines visibles sur le réseau. La commande unix wc munie de l'option l affiche le nombre de lignes qu'elle a été en mesure de lire. (La commande ruptime est contenue dans le paquetage rwho. Ce paquetage n'est pas systématiquement installé par les distributions Linux.)

En connectant avec un tube la sortie de ruptime à l'entrée de la commande wc -l, on obtient le nombre de machines visibles du réseau.

Même si au fil du temps de nouveaux types de données comme les entiers ou les tableaux ont été introduits dans certains shells, ces derniers manipulent essentiellement des chaînes de caractères :

ce sont des langages de programmation orientés chaînes de caractères. C'est ce qui rend les shells à la fois si puissants et si délicats à utiliser.

L'objet de ce document est de présenter de manière progressive les caractéristiques de bash comme langage de programmation.

I-1-3 - Atouts et inconvénients des shells

L'étude d'un shell tel que bash en tant que langage de programmation possède plusieurs avantages :

- c'est un langage interprété : les erreurs peuvent être facilement localisées et traitées ; d'autre part, des modifications de fonctionnalités sont facilement apportées à l'application sans qu'il soit nécessaire de recompiler et faire l'édition de liens de l'ensemble
- le shell manipule essentiellement des chaînes de caractères: on ne peut donc construire des structures de données complexes à l'aide de pointeurs, ces derniers n'existant pas en shell. Ceci a pour avantage d'éviter des erreurs de typage et de pointeurs mal gérés. Le développeur raisonne de manière uniforme en termes de chaînes de caractères
- le langage est adapté au prototypage rapide d'applications : les tubes, les substitutions de commandes et de variables favorisent la construction d'une application par assemblage de commandes préexistantes dans l'environnement Unix
- c'est un langage « glu » : il permet de connecter des composants écrits dans des langages différents. Ces derniers doivent uniquement respecter quelques règles particulièrement simples. Le composant doit être capable :
 - de lire sur l'entrée standard,
 - d'accepter des arguments et options éventuels,
 - d'écrire ses résultats sur la sortie standard,
 - d'écrire les messages d'erreur sur la sortie standard dédiée aux messages d'erreur.

Cependant, bash et les autres shells en général ne possèdent pas que des avantages :

- issus d'Unix, système d'exploitation écrit à l'origine par des développeurs pour des développeurs, les shells utilisent une syntaxe « ésotérique » d'accès difficile pour le débutant
- l'oubli ou l'ajout d'un caractère espace provoque facilement une erreur de syntaxe
- bash possède plusieurs syntaxes pour implanter la même fonctionnalité, comme la substitution de commande ou l'écriture d'une chaîne à l'écran. Cela est principalement dû à la volonté de fournir une compatibilité ascendante avec le Bourne shell, shell historique des systèmes Unix
- certains caractères spéciaux, comme les parenthèses, ont des significations différentes suivant le contexte ; en effet, les parenthèses peuvent introduire une liste de commandes, une définition de fonction ou bien imposer un ordre d'évaluation d'une expression arithmétique. Toutefois, afin de rendre l'étude de bash plus aisée, nous n'aborderons pas sa syntaxe de manière exhaustive ; en particulier, lorsqu'il existera plusieurs syntaxes pour mettre en oeuvre la même fonctionnalité, une seule d'entre elles sera présentée.

I-1-4 - Shell utilisé

La manière la plus simple de connaître le shell que l'on utilise est d'exécuter la commande unix ps qui liste les processus de l'utilisateur :

```
$ ps
PID   TTY    TIME    CMD
6908 pts/4 00:00:00 bash    => l'interpréteur utilisé est bash
```

```
6918 pts/4 00:00:00 ps
$
```

Comme il existe plusieurs versions de bash présentant des caractéristiques différentes, il est important de connaître la version utilisée. Pour cela, on utilise l'option `--version` de bash.

```
$ bash --version
GNU bash, version 3.1.14(1)-release (i486-pc-linux-gnu)
Copyright (C) 2005 Free Software Foundation, Inc.
$
```

La dernière version majeure de bash est la version 3. C'est celle qui sera étudiée.

I-2 - Syntaxe d'une commande

La syntaxe générale d'une commande (unix ou de bash) est la suivante :

```
[ chemin/ ] nom_cmd [ option ... ] [ argument ... ]
```

C'est une suite de mots séparés par un ou plusieurs blancs. On appelle blanc un caractère tab (tabulation horizontale) ou un caractère espace.

Un mot est une suite de caractères non blancs. Cependant, plusieurs caractères ont une signification spéciale pour le shell et provoquent la fin d'un mot : ils sont appelés méta-caractères (ex : `|`, `<`).

Bash utilise également des opérateurs de contrôle (ex : `(`, `||`) et des mots réservés pour son propre fonctionnement :

```
!      case do          done elif else  esac
fi     fo      function if    in      select then
time  until while {      }      [[      ]]
```

Bash distingue les caractères majuscules des caractères minuscules.

Le nom de la commande est le plus souvent le premier mot.

Une option est généralement introduite par le caractère tiret (ex : `-a`) ou dans la syntaxe GNU par deux caractères tiret consécutifs (ex : `--version`). Elle précise un fonctionnement particulier de la commande.

La syntaxe `[elt ...]` signifie que l'élément `elt` est facultatif (introduit par la syntaxe `[elt]`) ou bien présent une ou plusieurs fois (syntaxe `elt ...`). Cette syntaxe ne fait pas partie du shell ; elle est uniquement descriptive (méta-syntaxe).

Les arguments désignent les objets sur lesquels doit s'exécuter la commande.

```
ls -l RepC RepD : commande ls avec l'option l et les arguments RepC et RepD
```

Lorsque l'on souhaite connaître la syntaxe ou les fonctionnalités d'une commande `cmd` (ex : `ls` ou `bash`) il suffit d'exécuter la commande `man cmd` (ex : `man bash`). L'aide en ligne de la commande `cmd` devient alors disponible.

I-3 - Commandes internes et externes

Le shell distingue deux sortes de commandes :

- les commandes internes
- les commandes externes.

I-3-1 - Commandes internes

Une commande interne est une commande dont le code est implanté au sein de l'interpréteur de commande. Cela signifie que, lorsqu'on change de shell courant ou de connexion, par exemple en passant de bash au C-shell, on ne dispose plus des mêmes commandes internes. Exemples de commandes internes : cd , echo , for , pwd Sauf dans quelques cas particuliers, l'interpréteur ne crée pas de processus pour exécuter une commande interne. Les commandes internes de bash peuvent se décomposer en deux groupes :

- les commandes simples (ex : cd, echo)
- les commandes composées (ex : for, ((, {).

Commandes simples

Parmi l'ensemble des commandes internes, echo est l'une des plus utilisée :

echo : Cette commande interne affiche ses arguments sur la sortie standard en les séparant par un espace et va à la ligne.

```
$ echo bonjour      tout le      monde
bonjour tout le monde
$
```

Dans cet exemple, echo est invoquée avec quatre arguments : bonjour, tout, le et monde. On remarquera que les espacements entre les arguments ne sont pas conservés lors de l'affichage : un seul caractère espace sépare les mots affichés. En effet, le shell prétraite la commande, éliminant les blancs superflus.

Pour conserver les espacements, il suffit d'entourer la chaîne de caractères par une paire de guillemets :

```
$ echo "bonjour      tout le      monde"
bonjour      tout le      monde
$
```

On dit que les blancs ont été protégés de l'interprétation du shell.

Pour afficher les arguments sans retour à la ligne, on utilise l'option -n de echo.

```
$ echo -n bonjour
bonjour$
```

La chaîne d'appel \$ est écrite sur la même ligne que l'argument bonjour.

Commandes composées

Les commandes composées de bash sont :

```
case ... esac      if ... fi      for ... done  select ... done
until ... done     while ... done  [[ ... ]]    ( ... )
{ ... }
```

Seuls le premier et le dernier mot de la commande composée sont indiqués. Les commandes composées sont principalement des structures de contrôle.

I-3-2 - Commandes externes

Une commande externe est une commande dont le code se trouve dans un fichier ordinaire. Le shell crée un processus pour exécuter une commande externe. Parmi l'ensemble des commandes externes que l'on peut trouver dans un système, nous utiliserons principalement les commandes unix (ex : ls, mkdir, vi, sleep) et les fichiers shell.

La localisation du code d'une commande externe doit être connue du shell pour qu'il puisse exécuter cette commande. A cette fin, bash utilise la valeur de sa variable prédéfinie PATH. Celle-ci contient une liste de chemins séparés par le caractère : (ex : /bin:/usr/bin). Par exemple, lorsque l'utilisateur lance la commande unix cal, le shell est en mesure de l'exécuter et d'afficher le calendrier du mois courant car le code de cal est situé dans le répertoire /usr/bin présent dans PATH.

```
$ cal
  septembre 2006
di lu ma me je ve sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
$
```



pour connaître le statut d'une commande, on utilise la commande interne type.

```
$ type sleep
sleep is /bin/sleep      => sleep est une commande externe
$ type echo
echo is a shell builtin
$
```

I-4 - Modes d'exécution d'une commande

Deux modes d'exécution peuvent être distingués :

- l'exécution séquentielle
- l'exécution en arrière-plan.

I-4-1 - Exécution séquentielle

Le mode d'exécution par défaut d'une commande est l'exécution séquentielle : le shell lit la commande entrée par l'utilisateur, l'analyse, la prétraite et si elle est syntaxiquement correcte, l'exécute.

Une fois l'exécution terminée, le shell effectue le même travail avec la commande suivante. L'utilisateur doit donc attendre la fin de l'exécution de la commande précédente pour que la commande suivante puisse être exécutée : on dit que l'exécution est synchrone.

Si on tape la suite de commandes :

```
sleep 3 entrée date entrée
```

où entrée désigne la touche entrée, l'exécution de date débute après que le délai de 3 secondes se soit écoulé.

Pour lancer l'exécution séquentielle de plusieurs commandes sur la même ligne de commande, il suffit de les séparer par un caractère ;

```
$ cd /tmp ; pwd; echo bonjour; cd ; pwd
/tmp                => affichée par l'exécution de pwd
bonjour             => affichée par l'exécution de echo bonjour
/home/sanchis       => affichée par l'exécution de pwd
$
```

Pour terminer l'exécution d'une commande lancée en mode synchrone, on appuie simultanément sur les touches CTRL et C (notées control-C ou ^C).

En fait, la combinaison de touches appropriée pour arrêter l'exécution d'une commande en mode synchrone est indiquée par la valeur du champ intr lorsque la commande unix stty est lancée :

```
$ stty -a
speed 38400 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
...
$
```

Supposons que la commande cat soit lancée sans argument, son exécution semblera figée à un utilisateur qui débute dans l'apprentissage d'un système Unix. Il pourra utiliser la combinaison de touches mentionnée précédemment pour terminer l'exécution de cette commande.

```
$ cat
^C
$
```

I-4-2 - Exécution en arrière-plan

L'exécution en arrière-plan permet à un utilisateur de lancer une commande et de récupérer immédiatement la main pour lancer « en parallèle » la commande suivante (parallélisme logique). On utilise le caractère & pour lancer une commande en arrière-plan.

Dans l'exemple ci-dessous, la commande sleep 5 (suspendre l'exécution pendant 5 secondes) est lancée en arrière-plan. Le système a affecté le numéro d'identification 696 au processus correspondant tandis que bash a

affecté un numéro de travail (ou numéro de job) égal à 1 et affiché [1]. L'utilisateur peut, en parallèle, exécuter d'autres commandes (dans cet exemple, il s'agit de la commande ps). Lorsque la commande en arrière-plan se termine, le shell le signale à l'utilisateur après que ce dernier ait appuyé sur la touche entrée.

```
$ sleep 5 &
[1] 696
$ ps
PID TTY          TIME CMD
683 pts/0        00:00:00 bash
696 pts/0        00:00:00 sleep
697 pts/0        00:00:00 ps
$
=> l'utilisateur a appuyé sur la touche entrée mais sleep n'était pas terminée
$
=> l'utilisateur a appuyé sur la touche entrée et sleep était terminée
[1]+  Done                  sleep 5
$
```

Ainsi, outre la gestion des processus spécifique à Unix, bash introduit un niveau supplémentaire de contrôle de processus. En effet, bash permet de stopper, reprendre, mettre en arrière-plan un processus, ce qui nécessite une identification supplémentaire (numéro de job).

L'exécution en arrière-plan est souvent utilisée lorsqu'une commande est gourmande en temps CPU (ex : longue compilation d'un programme).

I-5 - Commentaires

Un commentaire débute avec le caractère # et se termine avec la fin de la ligne. Un commentaire est ignoré par le shell.

```
$ echo bonjour      # l'ami
bonjour
$ echo coucou       #l'   ami
coucou
$ # echo coucou
$
```

Pour que le caractère # soit reconnu en tant que début de commentaire, il ne doit pas être inséré à l'intérieur d'un mot ou terminer un mot.

```
$ echo il est#10
il est#10 heures
$
$ echo bon# jour
bon# jour
$
```

I-6 - Fichiers shell

Lorsqu'un traitement nécessite l'exécution de plusieurs commandes, il est préférable de les sauvegarder dans un fichier plutôt que de les retaper au clavier chaque fois que le traitement doit être lancé. Ce type de fichier est appelé fichier de commandes ou fichier shell ou encore script shell.

Exercice 1 :

1.) A l'aide d'un éditeur de texte, créer un fichier premier contenant les lignes suivantes :

```
#!/bin/bash
# premier
echo -n "La date du jour est: "
date
```

La notation #! en première ligne d'un fichier interprété précise au shell courant quel interpréteur doit être utilisé pour exécuter le script shell (dans cet exemple, il s'agit de /bin/bash).

La deuxième ligne est un commentaire.

2.) Vérifier le contenu de ce fichier.

```
Ex : $ cat premier
#!/bin/bash
# premier
echo -n "La date du jour est: "
date
$
```

3.) Pour lancer l'exécution d'un fichier shell fich, on peut utiliser la commande : bash fich

```
Ex : $ bash premier
la date du jour est : dimanche 3 septembre 2006, 15:41:26 (UTC+0200)
$
```

4.) Il est plus simple de lancer l'exécution d'un programme shell en tapant directement son nom, comme on le ferait pour une commande unix ou une commande interne. Pour que cela soit réalisable, deux conditions doivent être remplies :

- l'utilisateur doit posséder les permissions r (lecture) et x (exécution) sur le fichier shell
- le répertoire dans lequel se trouve le fichier shell doit être présent dans la liste des chemins contenue dans PATH.

```
Ex : $ ls -l premier
-rw-r--r-- 1 sanchis sanchis 63 sep 3 15:39 premier
$
```

Seule la permission r est possédée par l'utilisateur.

Pour ajouter la permission x au propriétaire du fichier fich, on utilise la commande :

```
chmod u+x fich
```

```
Ex : $ chmod u+x premier
$
$ ls -l premier
-rwxr--r-- 1 sanchis sanchis 63 sep 3 15:39 premier
$
```

Si on exécute premier en l'état, une erreur d'exécution se produit.

```
Ex : $ premier
-bash: premier: command not found
$
```

=> Problème !

Cette erreur se produit car bash ne sait pas où se trouve le fichier premier.

```
Ex : $ echo $PATH
/bin:/usr/bin:/usr/bin/X11
$
$ pwd
/home/sanchis
$
```

=> affiche la valeur de la variable PATH

Le répertoire dans lequel se trouve le fichier premier (répertoire /home/sanchis) n'est pas présent dans la liste de PATH. Pour que le shell puisse trouver le fichier premier, il suffit de mentionner le chemin permettant d'accéder à celui-ci.

```
Ex : $ ./premier
la date du jour est : dimanche 3 septembre 2006, 15:45:28 (UTC+0200)
$
```

Pour éviter d'avoir à saisir systématiquement ce chemin, il suffit de modifier la valeur de PATH en y incluant, dans notre cas, le répertoire courant (.).

```
Ex : $ PATH=$PATH:.
$
$ echo $PATH
/bin:/usr/bin:/usr/bin/X11:.
$
$ premier
la date du jour est : dimanche 3 septembre 2006, 15:47:38 (UTC+0200)
$
```

=> ajout du répertoire courant dans PATH

Exercice 2 :

Ecrire un programme shell repcour qui affiche le nom de connexion de l'utilisateur et le chemin absolu de son répertoire courant de la manière suivante :

```
Ex : $ repcour
mon nom de connexion est : sanchis
mon repertoire courant est : /home/sanchis
$
```

II - Substitution de paramètres

Dans la terminologie du shell, un paramètre désigne toute entité pouvant contenir une valeur.

Le shell distingue trois classes de paramètres :

- les variables, identifiées par un nom (Ex : a , PATH)
- les paramètres de position, identifiés par un numéro (Ex : 0 , 1 , 12)
- les paramètres spéciaux, identifiés par un caractère spécial (Ex : # , ? , \$)

Le mode d'affectation d'une valeur est spécifique à chaque classe de paramètres. Suivant celle-ci, l'affectation sera effectuée par l'utilisateur, par bash ou bien par le système. Par contre, pour obtenir la valeur d'un paramètre, on placera toujours le caractère \$ devant sa référence, et cela quelle que soit sa classe.

```
$ echo $PATH          => affiche la valeur de la variable PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games:./home/sanchis/bin
$ echo $$             => affiche la valeur du paramètre spécial $
17286
$
```

II-1 - Variables

Une variable est identifiée par un nom, c'est-à-dire une suite de lettres, de chiffres ou de caractères espace souligné ne commençant pas par un chiffre. Les lettres majuscules et minuscules sont différenciées.

Les variables peuvent être classées en trois groupes :

- les variables utilisateur (ex : a, valeur)
- les variables prédéfinies du shell (ex : PS1, PATH, REPLY, IFS)
- les variables prédéfinies de commandes unix (ex : TERM).

En général, les noms des variables utilisateur sont en lettres minuscules tandis que les noms des variables prédéfinies (du shell ou de commandes unix) sont en majuscules.

L'utilisateur peut affecter une valeur à une variable en utilisant

- l'opérateur d'affectation =
- la commande interne read.

II-1-1 - Affectation directe

Syntaxe : **nom**=[valeur] [**nom**=[valeur] ...]

Il est impératif que le nom de la variable, le symbole = et la valeur à affecter ne forment qu'une seule chaîne de caractères.

Plusieurs affectations peuvent être présentes dans la même ligne de commande.

```
$ x=coucou      y=bonjour      => la variable x contient la chaîne de caractères coucou
                                     => la variable y contient la chaîne bonjour
$
```

ATTENTION : les syntaxes d'affectation erronées les plus fréquentes contiennent

- un ou plusieurs caractères espace entre le nom de la variable et le symbole =.

Le shell interprète b comme une commande à exécuter ; ne la trouvant pas, il signale une erreur.

```
$ b      =coucou
-bash: b: command not found
$
```

- un ou plusieurs caractères espace entre le symbole = et la valeur de la variable.

La chaîne coucou est interprétée comme la commande à exécuter.

```
$ y=      coucou
-bash: coucou: command not found
$
```

Lorsque le shell rencontre la chaîne \$x, il la remplace textuellement par la valeur de la variable x, c.-à-d. coucou ; la commande exécutée par bash est : echo x est coucou

```
$ echo x est $x
x est coucou
$
```

Il est important de remarquer que le nom d'un paramètre et la valeur de ce paramètre ne se désignent pas de la même manière.

```
$ x=$x$y      => x : contenant, $x : contenu
$ echo $x
coucoubonjour
$
```

II-1-2 - Affectation par lecture

Elle s'effectue à l'aide de la commande interne read. Celle-ci lit une ligne entière sur l'entrée standard.

Syntaxe : read [var1 ...]

```
$ read a b
bonjour Monsieur      => chaînes saisies par l'utilisateur et enregistrées
                                     => respectivement dans les variables a et b
```



```
$ echo $b
Monsieur
$
```

Lorsque la commande interne `read` est utilisée sans argument, la ligne lue est enregistrée dans la variable prédéfinie du shell `REPLY`.

```
$ read
bonjour tout le monde
$
$ echo $REPLY
bonjour tout le monde
$
```

L'option `-p` de `read` affiche une chaîne d'appel avant d'effectuer la lecture ; la syntaxe à utiliser est :

```
read -p chaîne_d_appel [ var ... ]
```

```
$ read -p "Entrez votre prenom : " prenom
Entrez votre prenom : Eric
$ echo $prenom
Eric
$
```

Remarques sur la commande interne `read`

- S'il y a moins de variables que de mots dans la ligne lue, le shell affecte le premier mot à la première variable, le deuxième mot à la deuxième variable, etc., la dernière variable reçoit tous les mots restants.

```
$ read a b c
un bon jour coucou
$ echo $a
un
$ echo $c
jour coucou
$
```

- S'il y a davantage de variables que de mots dans la ligne lue, chaque variable reçoit un mot et après épuisement de ces derniers, les variables excédentaires sont vides (c.-à-d. initialisées à la valeur null).

```
$ read a b
Un
$ echo $a
Un
$ echo $b
=> valeur null
$
```

Exercice 1 :

Ecrire un programme shell deux fois qui affiche le message "Entrez un mot : ", lit le mot saisi par l'utilisateur puis affiche ce mot deux fois sur la même ligne.

```
$ deuxfois
Entrez un mot : toto
toto toto
$
```

Exercice 2 :

Ecrire un programme shell untrois qui demande à l'utilisateur de saisir une suite de mots constituée d'au moins trois mots et qui affiche sur la même ligne le premier et le troisième mot saisis.

```
$ untrois
Entrez une suite de mots : un petit coucou de Rodez
un coucou
$
```

II-1-3 - Variable en « lecture seule »

Pour définir une variable dont la valeur ne doit pas être modifiée (appelée constante dans d'autres langages de programmation), on utilise la syntaxe :

```
declare -r nom=valeur [ nom=valeur ... ]
```

```
$ declare -r mess=bonjour
$
```

On dit que la variable `mess` possède l'attribut `r`. Une tentative de modification de la valeur d'une constante provoque une erreur.

```
$ mess=salut
-bash: mess: readonly variable
$
```

Pour connaître la liste des constantes définies : **declare -r**

```
$ declare -r
declare -ar BASH_VERSION='([0]="3" [1]="1" [2]="17" [3]="1"
[4]="release" [5]="i486-pc-linux-gnu")'
declare -ir EUID="1002"
declare -ir PPID="25165"
declare -r SHELLOPTS="braceexpand:emacs:hashall:histexpand:history:
interactive-comments:monitor"
declare -ir UID="1002"
declare -r mess="bonjour"
$
```


Plusieurs constantes sont prédéfinies : le tableau `BASH_VERSION` (attribut `a`), les entiers `EUID`, `PPID` et `UID` (attribut `i`) et la chaîne `SHELLOPTS`.

II-2 - Paramètres de position et paramètres spéciaux

II-2-1 - Paramètres de position

Un paramètre de position est référencé par un ou plusieurs chiffres : 8 , 0 , 23 L'assignation de valeurs à des paramètres de position s'effectue :

- soit à l'aide de la commande interne set
- soit lors de l'appel d'un fichier shell ou d'une fonction shell.

 *on ne peut utiliser ni le symbole =, ni la commande interne read pour affecter directement une valeur à un paramètre de position.*

```
$ 23=bonjour
-bash: 23=bonjour: command not found
$ read 4
aa
-bash: read: `4': not a valid identifier
$
```

Commande interne set : La commande interne set affecte une valeur à un ou plusieurs paramètres de position en numérotant ses arguments suivant leur position. La numérotation commence à 1.

Syntaxe : set arg1 ...

```
$ set alpha beta gamma => alpha est la valeur du paramètre de position 1, beta la
                        => valeur du deuxième paramètre de position et gamma
                        => la valeur du paramètre de position 3
$
```

Pour obtenir la valeur d'un paramètre de position, il suffit de placer le caractère \$ devant son numéro ; ainsi, \$1 permet d'obtenir la valeur du premier paramètre de position, \$2 la valeur du deuxième et ainsi de suite.

```
$ set ab be ga            => numérotation des mots ab, be et ga
$ echo $3 $2
ga be
$ echo $4
$
```

Tous les paramètres de position sont réinitialisés dès que la commande interne set est utilisée avec au moins un argument.

```
$ set coucou
$ echo $1
coucou
$ echo $2
=> les valeurs be et ga précédentes ont disparues
$
```

La commande **set --** rend indéfinie la valeur des paramètres de position préalablement initialisés.

```
$ set alpha beta
```

```
$ echo $1
alpha
$
$ set --
$ echo $1
=> les valeurs alpha et beta sont perdues
$
```

Remarques :

- Utilisée sans argument, set a un comportement différent : elle affiche la (longue) liste des noms et valeurs des variables définies.

```
$ set
BASH=/bin/bash
...
$
```

- Si la valeur du premier argument de set commence par un caractère - ou +, une erreur se produit. En effet, les options de cette commande interne commencent par un de ces deux caractères. Pour éviter que ce type d'erreur ne se produise, on utilise la syntaxe : **set -- arg ...**

```
$ a+=qui
$ set $a
-bash: set: +q: invalid option
set: usage: set [--abefhkmnptuvxBCHP] [-o option] [arg ...]
$ set -- $a
$ echo $1
+qui
$
```

II-2-2 - Paramètres spéciaux

Un paramètre spécial est référencé par un caractère spécial. L'affectation d'une valeur à un paramètre spécial est effectuée par le shell. Pour obtenir la valeur d'un paramètre spécial, il suffit de placer le caractère \$ devant le caractère spécial qui le représente.

Un paramètre spécial très utilisé est le paramètre # (à ne pas confondre avec le début d'un commentaire). Celui-ci contient le nombre de paramètres de position ayant une valeur.

```
$ set a b c

$ echo $#           => affichage de la valeur du paramètre spécial #
3                  => il y a 3 paramètres de position ayant une valeur
$
```

Exercice 3 :

Ecrire un programme shell nbmots qui demande à l'utilisateur de saisir une suite quelconque de mots puis affiche le nombre de mots saisis.

```
$ nbmots
Entrez une suite de mots : un deux trois quatre cinq
```

```
5      => 5 mots ont été saisis
$
```

II-2-3 - Commande interne shift

La commande interne shift décale la numérotation des paramètres de position ayant une valeur.

Syntaxe : `shift [n]`

Elle renomme le n+1 ième paramètre de position en paramètre de position 1, le n+2 ième paramètre de position en paramètre de position 2, etc. : (n+1) => 1 , (n+2) => 2 , etc. Une fois le décalage effectué, le paramètre spécial # est mis à jour.

```
$ set a b c d e      =>      12345
$ echo $1 $2 $#
ab5
$ shift 2      =>      abcde      les mots a et b sont devenus inaccessibles
      =>      123
$ echo $1 $3
ce
$ echo $#
3
$
```

La commande interne shift sans argument est équivalente à shift 1.



shift ne modifie pas la valeur du paramètre de position 0 qui possède une signification particulière.

II-2-4 - Paramètres de position et fichiers shell

Dans un fichier shell, les paramètres de position sont utilisés pour accéder aux valeurs des arguments qui ont été passés lors de son appel : cela signifie qu'au sein du fichier shell, les occurrences de \$1 sont remplacées par la valeur du premier argument, celles de \$2 par la valeur du deuxième argument, etc. Le paramètre spécial \$# contient le nombre d'arguments passés lors de l'appel.

Le paramètre de position 0 contient le nom complet du programme shell qui s'exécute.

Soit le programme shell copie :

copie

```
#!/bin/bash
#      @(#)      copie
echo "Nom du programme : $0"
echo "Nb d'arguments : $#"
```

```
echo "Source : $1"
echo "Destination : $2"
cp $1 $2
```

Pour exécuter ce fichier shell :

```
$ chmod u+x copie
$ copie /etc/passwd X
Nom du programme : ./copie
Nb d'arguments : 2
Source : /etc/passwd
Destination : X
$
```

Dans le fichier copie, chaque occurrence de \$1 a été remplacée par la chaîne de caractères /etc/passwd, celles de \$2 par X.

Exercice 4 :

Ecrire un programme shell cp2fois prenant trois arguments : le premier désigne le nom du fichier dont on veut copier le contenu dans deux fichiers dont les noms sont passés comme deuxième et troisième arguments. Aucun cas d'erreur ne doit être considéré.

Lorsque la commande interne set est utilisée à l'intérieur d'un programme shell, la syntaxe \$1 possède deux significations différentes : \$1 comme premier argument du programme shell, et \$1 comme premier paramètre de position initialisé par set au sein du fichier shell.

programme shell ecrase_arg

```
# !/bin/bash
echo '$1' est $1      => la chaîne $1 est remplacée par le premier argument
set hello             => set écrase la valeur précédente de $1
echo '$1' est $1
```

```
$ ecrase_arg bonjour coucou salut
$1 est bonjour
$1 est hello
$
```

II-2-5 - Paramètres spéciaux * et @

Les paramètres spéciaux @ et * contiennent tous deux la liste des valeurs des paramètres de position initialisés.

```
$ set un deux trois quatre
$ echo $*
un deux trois quatre
$ echo $@
un deux trois quatre
$
```

Ces deux paramètres spéciaux ont des valeurs distinctes lorsque leur référence est placée entre des guillemets ("\$*" et "\$@") :

```
"$*" est remplacée par "$1 $2 ... "
"$@" est remplacée par "$1" "$2" ...
```

```
$ set un deux trois      => trois paramètres de position sont initialisés
$ set "$*"               => est équivalent à : set "un deux trois"
$ echo $#
```

```
1
$ echo $1
un deux trois
$
```

L'intérêt de la syntaxe "\$*" est de pouvoir considérer l'ensemble des paramètres de position initialisés comme une unique chaîne de caractères. La syntaxe "\$@" produit autant de chaînes que de paramètres de positions initialisés.

```
$ set un deux trois          => trois paramètres de position initialisés
$ set "$@"                  => est équivalent à : set "un" "deux" "trois"
$ echo $#
3
$ echo $1
un
$
```

Variable prédéfinie IFS et paramètre spécial "\$*" :

La variable prédéfinie du shell IFS contient les caractères séparateurs de mots dans une commande. Par défaut, ce sont les caractères espace, tabulation et interligne. L'initialisation de IFS est effectuée par bash.

Le shell utilise le premier caractère mémorisé dans IFS (par défaut, le caractère espace) pour séparer les différents paramètres de position lorsque la syntaxe "\$*" est utilisée. En modifiant la valeur de IFS, l'utilisateur peut changer ce caractère séparateur.

```
$ IFS=:                      => le caractère deux-points est maintenant le premier caractère de IFS
$ set un deux trois quatre
$ echo $*
un deux trois quatre
$ echo "$@"
un deux trois quatre
$ echo "$*"
un:deux:trois:quatre        => seule la forme "$*" provoque un changement
$
```

II-3 - Suppression des ambiguïtés

Pour éviter les ambiguïtés dans l'interprétation des références de paramètres, on utilise la syntaxe \${paramètre}.

```
$ x=bon
$ x1=jour
$ echo $x1                    => valeur de la variable x1
jour
$ echo ${x}1                  => pour concaténer la valeur de la variable x à la chaîne
##1##
bon1
$
```

```
$ set un deux trois quatre cinq six sept huit neuf dix onze douze
$ echo $11
un1
$ echo ${11}                  => pour obtenir la valeur du onzième paramètre de position
onze
$
```

II-4 - Paramètres non définis

Trois cas peuvent se présenter lorsque le shell doit évaluer un paramètre :

- le paramètre n'est pas défini,
- le paramètre est défini mais ne possède aucune valeur (valeur vide),
- le paramètre possède une valeur non vide.

Lorsque l'option `nounset` de la commande interne **set** est positionnée à l'état *on* (commande **set -o nounset**), bash affiche un message d'erreur quand il doit évaluer un paramètre non défini.

```
$ set -o nounset                => option nounset à l'état on
$ echo $c                      => variable utilisateur c non définie,
-bash: c: unbound variable      => message d'erreur !
$ echo $1
-bash: $1: unbound variable
$ d=                            => d : variable définie mais vide
$ echo $d                      => valeur null, pas d'erreur
$
```

La présence de paramètres non définis ou définis mais vides dans une commande peut mener son exécution à l'échec. Afin de traiter ce type de problème, bash fournit plusieurs mécanismes supplémentaires de substitution de paramètres. L'un d'eux consiste à associer au paramètre une valeur ponctuelle lorsqu'il est non défini ou bien défini vide. La syntaxe à utiliser est la suivante : `${paramètre:-chaîne}`

```
$ set -o nounset
$ ut1=root                    => ut1 définie et non vide
$ ut2=                        => ut2 définie et vide
$ echo $ut1
root
$ echo $ut2
$ echo $1
-bash: $1: unbound variable    => paramètre de position 1 non défini
$
```

Avec la syntaxe mentionnée ci-dessus, il est possible d'associer temporairement une valeur par défaut aux trois paramètres.

```
$ echo ${ut1:-sanchis}
root                        => ut1 étant définie non vide, elle conserve sa valeur
$ echo ${ut2:-sanchis}
sanchis                    => ut2 est définie et vide, la valeur de remplacement est
                           utilisée
$ echo ${1:-sanchis}
sanchis                    => le premier paramètre de position est non défini
$
```

Cette association ne dure que le temps d'exécution de la commande.

```
$ echo $1
-bash: $1: unbound variable    => le paramètre est redevenu indéfini
$
```


La commande `set +o nounset` positionne l'option `nounset` à l'état off : le shell traite les paramètres non définis comme des paramètres vides.

```
$ set +o nounset      => option nounset à l'état off
$ echo $e             => variable e non définie,
                      => aucune erreur signalée

$ echo $2             => aucune erreur signalée

$ f=                  => f : variable définie vide
$ echo $f
$
```

II-5 - Suppression de variables

Pour rendre indéfinies une ou plusieurs variables, on utilise la commande interne `unset`.

Syntaxe : `unset var [var1 . . .]`

```
$ a=coucou            => la variable a est définie
$ echo $a
coucou
$ unset a             => la variable a est supprimée
$ set -o nounset      => pour afficher le message d'erreur
$ echo $a
-bash: a: unbound variable
$
```

Une variable en « lecture seule » ne peut être supprimée par `unset`.

```
$ declare -r a=coucou    => définition de la constante a
$ echo $a
coucou
$ unset a
-bash: unset: a: cannot unset: readonly variable
$
```

III - Substitution de commandes

III-1 - Présentation

Syntaxe : `$(cmd)`

Une commande `cmd` entourée par une paire de parenthèses `()` précédées d'un caractère `$` est exécutée par le shell puis la chaîne `$(cmd)` est remplacée par les résultats de la commande `cmd` écrits sur la sortie standard, c'est à dire l'écran. Ces résultats peuvent alors être affectés à une variable ou bien servir à initialiser des paramètres de position.

```
$ pwd
/home/sanchis
$ repert=$(pwd)
$ echo mon repertoire est $repert
mon repertoire est /home/sanchis
$
```

=> résultat écrit par `pwd` sur sa sortie standard
=> la chaîne `/home/sanchis` remplace la chaîne `$(pwd)`

Exercice 1 :

En utilisant la substitution de commande, écrire un fichier shell mach affichant : "Ma machine courante est nomdelamachine"

```
$ mach
Ma machine courante est jade
$
```

Lorsque la substitution de commande est utilisée avec la commande interne `set`, l'erreur suivante peut se produire :

```
$ set $( ls -l .bashrc )
bash: set: -w: invalid option
set: usage: set [--abefhkmnptuvxBCHP] [-o option] [arg ...]
$ ls -l .bashrc
-rw-r--r-- 1 sanchis users 1342 nov 29 2004 .bashrc
$
```

Le premier mot issu de la substitution commence par un caractère tiret : la commande interne `set` l'interprète comme une option, ce qui provoque une erreur. Pour résoudre ce type de problème, on double le caractère tiret.

```
$ set -- $(ls -l .bashrc)
$ echo $1
-rw-r--r--
$
```

Exercice 2 :

Ecrire un programme shell `taille` qui prend un nom de fichier en argument et affiche sa taille. On ne considère aucun cas d'erreur.

Plusieurs commandes peuvent être présentes entre les parenthèses.

```
$ pwd ; whoami
/home/sanchis
sanchis
$ set $(pwd ; whoami)
$ echo $2: $1
sanchis: /home/sanchis
$
```

Exercice 3 : A l'aide de la commande unix date, écrire un programme shell jour qui affiche le jour courant du mois

```
$ date
dimanche 22 octobre 2006, 18:33:38 (UTC+0200)
$ jour
22
$
```

Exercice 4 :

- a) Ecrire un programme shell heure1 qui affiche l'heure sous la forme heures:minutes:secondes

```
Ex : $ heure1
18:33:38
$
```

- b) Ecrire un programme shell heure qui n'affiche que les heures et minutes

```
Ex : $ heure
18:33
$
```

Exercice 5 :

Ecrire un programme shell nbconnect qui affiche le nombre d'utilisateurs connectés sur la machine locale



plusieurs solutions sont possibles :

solution 1, avec who | wc -l

solution 2, avec uptime

solution 3, avec users

solution 4, avec who -q

Les substitutions de commandes peuvent être imbriquées.

```
$ ls .gnome
application-info gnome-vfs mime-info
$ set $( ls $(pwd)/.gnome )
$ echo $#                                => nombre d'entrées du répertoire .gnome
3
```

```
$
```

Plusieurs sortes de substitutions (de commandes, de variables) peuvent être présentes dans la même commande.

```
$ read rep
gnome
$ set $( ls $( pwd )/$rep )  => substitutions de deux commandes et d'une variable
$
```

La substitution de commande permet également de capter les résultats écrits par un fichier shell.

```
$ jour
22
$ resultat=$( jour )
$
```

La variable resultat contient la chaîne 22.

Exercice 6 :

Ecrire un programme shell uid qui affiche l'uid de l'utilisateur. On utilisera la commande unix id, la commande interne set et la variable prédéfinie IFS.



Il existe pour la substitution de commande une syntaxe plus ancienne ``cmd`` (deux caractères accent grave entourent la commande cmd), à utiliser lorsque se posent des problèmes de portabilité.

III-2 - Substitutions de commandes et paramètres régionaux

Créé par des informaticiens américains, le système Unix était originellement destiné à des utilisateurs anglophones. La diffusion des systèmes de la famille Unix (dont GNU/Linux) vers des publics de langues et de cultures différentes a conduit leurs développeurs à introduire des paramètres régionaux (locale). Ces derniers permettent par exemple de fixer la langue d'affichage des messages, le format des dates ou des nombres. Les paramètres régionaux se présentent à l'utilisateur sous la forme de variables prédéfinies dont le nom commence par LC_.

La commande unix locale utilisée sans argument affiche la valeur courante de ces variables d'environnement.

```
$ locale
LANG=fr_FR@euro
LC_CTYPE="fr_FR@euro"
LC_NUMERIC="fr_FR@euro"
LC_TIME="fr_FR@euro"
LC_COLLATE="fr_FR@euro"
LC_MONETARY="fr_FR@euro"
LC_MESSAGES="fr_FR@euro"
LC_PAPER="fr_FR@euro"
LC_NAME="fr_FR@euro"
LC_ADDRESS="fr_FR@euro"
LC_TELEPHONE="fr_FR@euro"
LC_MEASUREMENT="fr_FR@euro"
LC_IDENTIFICATION="fr_FR@euro"
LC_ALL=
$
```

Des commandes telles que `date` utilisent la valeur de ces variables prédéfinies pour afficher leurs résultats.

```
$ date
vendredi 3 novembre 2006, 18:47:02 (UTC+0100)
$
```

On remarque que le format du résultat correspond bien à une date « à la française » (jour de la semaine, jour du mois, mois, année). Si cette internationalisation procure un confort indéniable lors d'une utilisation interactive du système, elle pose problème lorsque l'on doit écrire un programme shell se basant sur la sortie d'une telle commande, sortie qui dépend étroitement de la valeur des paramètres régionaux. La portabilité du programme shell en est fortement fragilisée.

Cependant, le fonctionnement standard d'un système unix traditionnel peut être obtenu en choisissant la « locale standard » appelée C. Pour que cette locale n'affecte temporairement que les résultats d'une seule commande, par exemple `date`, il suffit d'exécuter la commande `LC_ALL=C date`

```
$ LC_ALL=C date
Fri Nov 3 18:47:41 CET 2006
$
```

On s'aperçoit qu'avec la locale standard, le jour du mois est en troisième position alors qu'avec la locale précédente, il était en deuxième position. Pour obtenir de manière portable le jour courant du mois, on pourra exécuter les commandes suivantes :

```
$ set $(LC_ALL=C date) ; echo $3
3
$
```

IV - Caractères et expressions génériques

Les caractères et expressions génériques sont issus d'un mécanisme plus général appelé expressions rationnelles (regular expressions) ou expressions régulières. Ces caractères et expressions sont utilisés pour spécifier un modèle de noms d'entrées. Ce modèle est ensuite interprété par le shell pour créer une liste triée de noms d'entrées. Par défaut, le shell traite uniquement les entrées non cachées du répertoire courant ; cela signifie que les entrées dont le nom commence par le caractère `.` sont ignorées.

Pour illustrer l'utilisation des caractères et expressions génériques, on utilisera un répertoire appelé `generique` contenant les 16 entrées suivantes :

```
$ pwd
/home/sanchis/generique
$ ls -l
total 0
-rw-r--r-- 1 sanchis users 0 nov 10 2004 1
-rw-r--r-- 1 sanchis users 0 nov 10 2004 a
-rw-r--r-- 1 sanchis users 0 nov 10 2004 _a
-rw-r--r-- 1 sanchis users 0 nov 10 2004 A
-rw-r--r-- 1 sanchis users 0 nov 10 2004 à
-rw-r--r-- 1 sanchis users 0 nov 10 2004 ami
-rw-r--r-- 1 sanchis users 0 nov 10 2004 an
-rw-r--r-- 1 sanchis users 0 nov 10 2004 Arbre
-rw-r--r-- 1 sanchis users 0 nov 10 2004 e
-rw-r--r-- 1 sanchis users 0 nov 10 2004 é
-rw-r--r-- 1 sanchis users 0 nov 10 2004 émirat
-rw-r--r-- 1 sanchis users 0 nov 10 2004 En
-rw-r--r-- 1 sanchis users 0 nov 10 2004 état
-rw-r--r-- 1 sanchis users 0 nov 10 2004 minuit
-rw-r--r-- 1 sanchis users 0 nov 10 2004 zaza
-rw-r--r-- 1 sanchis users 0 nov 10 2004 Zoulou
$
```



La valeur des paramètres régionaux influe grandement sur la présentation et la méthode de tri des noms d'entrées. Si on utilise la locale standard, on obtient l'affichage suivant :

```
$ LC_ALL=C ls -l
total 0
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 1
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 A
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 Arbre
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 En
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 Zoulou
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 _a
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 a
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 ami
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 an
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 e
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 minuit
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 zaza
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 ?
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 ?
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 ?mirat
-rw-r--r-- 1 sanchis users 0 Nov 10 2004 ?tat
$
```



On s'aperçoit que l'affichage des noms d'entrées contenant des caractères accentués est altéré. La raison de cela est que seuls les 7 premiers bits codant les caractères sont

interprétés en locale standard. La locale précédente utilise un codage des caractères sur 8 bits.

IV-1 - Caractères génériques

Les caractères génériques du shell sont : * ? []



*Il ne faut pas confondre la syntaxe [] du méta-langage et les caractères génériques [] du shell. Pour que bash interprète les caractères génériques, il est nécessaire que l'option **noglob** de la commande interne **set** soit à l'état **off**. Pour connaître l'état des différentes options de la commande interne **set**, on utilise la commande **set -o**.*

```
$ set -o
allexport          off
braceexpand       on
emacs             on
errexit           off
hashall           on
histexpand        on
history           on
ignoreeof         off
interactive-comments on
keyword           off
monitor           on
noclobber         off
noexec            off
noglob            off
nolog             off
notify            off
nounset           off
onecmd            off
physical          off
posix             off
privileged        off
verbose           off
vi                off
xtrace            off
$
```

- Le caractère générique * désigne n'importe quelle suite de caractères, même la chaîne vide

```
$ echo *
a _a A à ami an Arbre e é émirat En état minuit zaza Zoulou
$
```


Tous les noms d'entrées sont affichés, triés.

```
$ ls -l a*
-rw-r--r-- 1 sanchis users 0 nov 10 11:41 a
-rw-r--r-- 1 sanchis users 0 nov 10 11:41 ami
-rw-r--r-- 1 sanchis users 0 nov 10 11:41 an
$
```

La notation **a*** désigne toutes les entrées du répertoire courant dont le nom commence par la lettre **a**. Par conséquent, dans la commande **ls -l a*** le shell remplace la chaîne **a*** par la chaîne : **a ami an**

En d'autres termes, la commande ls "ne voit pas" le caractère * puisqu'il est prétraité par le shell.

```
$ echo *a*           => noms des entrées contenant un caractère a
a _a ami an émirat état zaza
$
```

 *quand aucune entrée ne correspond au modèle fourni, les caractères et expressions génériques ne sont pas interprétés.*

```
$ echo Z*t
Z*t
$
```

- Le caractère générique ? désigne un et un seul caractère

```
$ echo ?           => noms d'entrées composés d'un seul caractère
laAàèé
$ echo ?n
an En
$
```

Plusieurs caractères génériques différents peuvent être présents dans un modèle.

```
$ echo ?mi*
ami émirat
$
```

Exercice :

Ecrire un modèle permettant d'afficher le nom des entrées dont le 2ème caractère est un a.

Exercice :

Ecrire un modèle permettant d'afficher le nom des entrées dont le 1er caractère est un a suivi par 2 caractères quelconques.

- Les caractères génériques [] désignent un seul caractère parmi un groupe de caractères

Ce groupe est précisé entre les caractères []. Il peut prendre deux formes : la forme ensemble et la forme intervalle.

Forme ensemble : tous les caractères souhaités sont explicitement mentionnés entre []

```
$ ls -l [ame]
-rw-r--r--      1 sanchis   users      0 nov 10 11:41 a
-rw-r--r--      1 sanchis   users      0 nov 10 11:41 e
$ echo ?[ame]*
_a ami émirat zaza
$
```

Le modèle [ame] désigne tous les noms d'entrées constitués d'un seul caractère a, m ou e.

Forme intervalle : seules les bornes de l'intervalle sont précisées et séparées par un - .


```
$ echo *
1 a _a A à ami an Arbre e é émirat En état minuit zaza Zoulou
$ echo [a-z]*           => le caractère Z est ignoré
a A à ami an Arbre e é émirat En état minuit zaza
$ echo [A-Z]*           => le caractère a est ignoré
A à Arbre e é émirat En état minuit zaza Zoulou
$
```

L'interprétation du modèle [a-z] est conditionnée par la valeur des paramètres régionaux : cette syntaxe est donc non portable et il est déconseillé de l'employer car elle peut avoir des effets néfastes lorsqu'elle est utilisée avec la commande unix rm.

Si l'on souhaite désigner une classe de caractères tels que les minuscules ou les majuscules, il est préférable d'utiliser la syntaxe POSIX 1003.2 correspondante. Ce standard définit des classes de caractères sous la forme *[:nom_classe:]*

Des exemples de classes de caractères définies dans ce standard sont :

- [:upper:] (majuscules)
- [:lower:] (minuscules)
- [:digit:] (chiffres, 0 à 9)
- [:alnum:] (caractères alphanumériques).

 *pour désigner n'importe quel caractère d'une classe, par exemple les minuscules, on place la classe entre les caractères génériques [].*

```
$ echo [[:lower:]]           => noms d'entrées constitués d'une seule minuscule
a àéé
$ echo [[:upper:]]*
A Arbre En Zoulou
$
```

Pour afficher les noms d'entrées commençant par une majuscule ou le caractère a :

```
echo [[:upper:]a]*
```

Il est possible de mentionner un caractère ne devant pas figurer dans un groupe de caractères. Il suffit de placer le caractère ! juste après le caractère [.

```
$ echo ?[!n]* => noms d'entrées ne comportant pas le caractère n en 2ème position
_a ami      Arbre émirat état minuit zaza Zoulou
$ echo [![:lower:]]
1A
$ echo [![:upper:]]*
1 a _a      à ami an e é émirat état minuit zaza
$ echo [![:upper:]]*[^at]
ami an
$
```

IV-2 - Expressions génériques

Pour que bash, interprète les expressions génériques, il est nécessaire que l'option *extglob* de la commande interne **shopt** soit activée.

```
$ shopt
cdable_vars off
cdspell off
checkhash off
checkwinsize on
cmdhist on
dotglob off
execfail off
expand_aliases on
extdebug off
extglob off
extquote on
failglob off
force_fignore on
gnu_errfmt off
histappend off
histreedit off
histverify off
hostcomplete on
huponexit off
interactive_comments on
lithist off
login_shell on
mailwarn off
no_empty_cmd_completion off
nocaseglob off
nocasematch off
nullglob off
progcomp on
promptvars on
restricted_shell off
shift_verbose off
sourcepath on
xpg_echo off
$
```

Pour activer une option de la commande interne shopt on utilise la commande : **shopt -s opt**

Pour activer le traitement des expressions génériques par le shell : **shopt -s extglob**

Les expressions génériques de bash sont :

- ? (liste_modèles) : correspond à 0 ou 1 occurrence de chaque modèle
- * (liste_modèles) : correspond à 0,1 ou plusieurs occurrences de chaque modèle
- + (liste_modèles) : correspond à au moins 1 occurrence de chaque modèle
- @ (liste_modèles) : correspond exactement à 1 occurrence de chaque modèle
- ! (liste_modèles) : correspond à tout sauf aux modèles mentionnés


```
$ echo +([[:lower:]])
bash: syntax error near unexpected token `('
$ shopt -s extglob
$ echo +([[:lower:]])          => noms constitués que de minuscules
a à ami an e é émirat état minuit zaza
$ echo !([[:lower:]])
1 _a A Arbre En Zoulou
```


```
$
```

Exercice :

Ecrire un modèle correspondant aux noms d'entrées ne comportant pas de caractère a

Si l'on souhaite utiliser les expressions génériques dans un fichier shell, on y inclura préalablement la commande **shopt -s extglob**

 Outre **extglob**, d'autres options de la commande interne **shopt** permettent de modifier la création de la liste des noms d'entrées. Ces options sont : **dotglob**, **nocaseglob**, **nullglob**, **failglob**. Par défaut elles sont inactives (off).

 Pour gérer ses propres options, bash intègre deux commandes, **set** et **shopt**. D'un point de vue pratique,

*pour connaître l'état des options de la commande interne set : **set -o***

*pour activer une option de la commande interne set : **set -o option***

*pour désactiver une option de la commande interne set : **set +o option***

*pour connaître l'état des options de la commande interne shopt : **shopt***

*pour activer une option de la commande interne shopt : **shopt -s option***

*pour désactiver une option de la commande interne shopt : **shopt -u option***

V - Redirections élémentaires

V-1 - Descripteurs de fichiers

Un processus Unix possède par défaut trois voies d'interaction avec l'extérieur appelées entrées / sorties standard identifiées par un entier positif ou nul appelé descripteur de fichier.

Ces entrées / sorties standard sont :

- une entrée standard , de descripteur 0
- une sortie standard , de descripteur 1
- une sortie standard pour les messages d'erreurs, de descripteur 2.

Toute commande étant exécutée par un processus, nous dirons également qu'une commande possède trois entrées / sorties standard.

```
0  --> commande  --> 1
    |
    v
    2
```

De manière générale, une commande de type filtre (ex : cat) prend ses données sur son entrée standard qui correspond par défaut au clavier, affiche ses résultats sur sa sortie standard, par défaut l'écran, et affiche les erreurs éventuelles sur sa sortie standard pour les messages d'erreurs, par défaut l'écran également.

V-2 - Redirections élémentaires

On peut rediriger séparément chacune des trois entrées/sorties standard d'une commande. Cela signifie qu'une commande pourra :

- lire les données à traiter à partir d'un fichier et non du clavier de l'utilisateur
- écrire les résultats ou erreurs dans un fichier et non à l'écran.

Redirection de la sortie standard : **> fichier** ou **1> fichier**

```
$ pwd
/home/sanchis
$ pwd > fich
=> aucun résultat affiché à l'écran !
=> le résultat a été enregistré dans le fichier fich
$ cat fich
/home/sanchis
$
```

Cette première forme de redirection de la sortie standard écrase l'ancien contenu du fichier de sortie (fichier fich) si celui-ci existait déjà, sinon le fichier est créé. Le shell prétraite une commande avant de l'exécuter : dans l'exemple ci-dessous, le shell crée un fichier vide f devant recevoir les résultats de la commande (ici inexistante) ; l'unique effet de >f sera donc la création du fichier f ou sa remise à zéro.

```
$ >f          => crée ou remet à zéro le fichier f
$ ls -l f
-rw-r--r--    1 sanchis    users          0 oct 22 11:34 f
$
```

Une commande ne possède qu'une seule sortie standard; par conséquent, si on désire effacer le contenu de plusieurs fichiers, il est nécessaire d'utiliser autant de commandes que de fichiers à effacer. Le caractère ; permet d'exécuter séquentiellement plusieurs commandes écrites sur la même ligne.

```
$ > f1 ; >f2
$
```

La redirection de la sortie standard de cat est souvent utilisée pour affecter un contenu succinct à un fichier.

```
$ cat >tata          => l'entrée standard est directement recopiée dans tata
a demain
si vous le voulez bien
^D
$ cat tata
a demain
si vous le voulez bien
$
```

```
0          1
a demain          -----> cat -----> tata
si vous le voulez bien
```

Pour concaténer (c'est à dire ajouter à la fin) la sortie standard d'une commande au contenu d'un fichier, une nouvelle forme de redirection doit être utilisée : **>> fichier**

```
$ pwd > t
$ date >> t
$ cat t
/home/sanchis
lundi 20 novembre 2006, 15:27:41 (UTC+0100)
$
```

Comme pour la redirection précédente, l'exécution de >> fich crée le fichier fich s'il n'existait pas.

Redirection de la sortie standard pour les messages d'erreur : **2> fichier**

On ne doit laisser aucun caractère espace entre le chiffre 2 et le symbole >.

```
$ pwd
/home/sanchis
$ ls vi          => l'éditeur de texte vi se trouve dans le répertoire /usr/bin
ls: vi: Aucun fichier ou répertoire de ce type
$ ls vi 2> /dev/null
$
```

Le fichier spécial **/dev/null** est appelé « poubelle » ou « puits » car toute sortie qui y est redirigée, est perdue. En général, il est utilisé lorsqu'on est davantage intéressé par le code de retour de la commande plutôt que par les

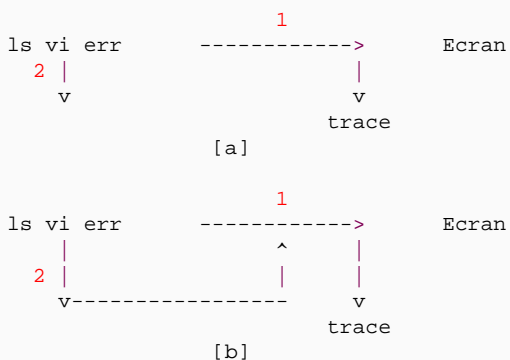
résultats ou messages d'erreur qu'elle engendre. Comme pour la sortie standard, il est possible de concaténer la sortie standard pour les messages d'erreur d'une commande au contenu d'un fichier : **2>> fichier**

```
$ ls vi
ls: vi: Aucun fichier ou répertoire de ce type
$ ls vi 2>err
$ ls date 2>>err
$ cat err
ls: vi: Aucun fichier ou répertoire de ce type
ls: date: Aucun fichier ou répertoire de ce type
$
```

Pour rediriger la sortie standard pour les messages d'erreur vers la sortie standard (c.a.d vers le fichier de descripteur 1), on utilisera la syntaxe : **2>&1** Cela est souvent utilisé lorsqu'on désire conserver dans un même fichier toutes les sorties.


```
$ ls
Mail                                err                                public_html                        tmp
_.forward                          fic_noms                          t
$ ls vi err >trace 2>&1
$ cat trace
ls: vi: Aucun fichier ou répertoire de ce type
err
$
```

La sortie standard est redirigée vers le fichier trace [a] puis la sortie standard pour les messages d'erreur est redirigée vers la sortie standard, c.-à-d. également vers le fichier trace [b].



La syntaxe **>> fichier** est équivalente à la syntaxe **> fichier 2>&1**

```
$ ls vi err >> trace
$ cat trace
ls: vi: Aucun fichier ou répertoire de ce type
err
$
```

 **Les redirections étant traitées de gauche à droite, l'ordre des redirections est important.**

Exercice : Que fait la commande : `ls vi fic_noms 2>&1 >trace`

Redirection de l'entrée standard : **< fichier**

```
$ mail sanchis < lettre
$
```

La commande **mail** envoie à l'utilisateur sanchis le contenu du fichier lettre. Une substitution de commande associée à une redirection de l'entrée standard permet d'affecter à une variable le contenu d'un fichier. En effet, la substitution de commande `$(cat fichier)` peut être avantageusement remplacée par la syntaxe `$(< fichier)`. Cette deuxième forme est plus rapide.

```
$ cat fic_noms
pierre betaille
anne debazac
julie donet
$ liste=$(cat fic_noms)
$ echo $liste                                => liste est une variable et non un fichier !
pierre betaille anne debazac julie donet    => elle contient le contenu du fichier fic_noms
$
```

Redirections séparées des entrées / sorties standard : Les entrées / sorties peuvent être redirigées indépendamment les unes des autres.

```
$ wc -l <fic_noms >fic_nblignes 2>err
$ cat fic_nblignes
3
$ cat err
$
```

La commande unix **wc -l** affiche le nombre de lignes d'un ou plusieurs fichiers texte. Ici, il s'agit du nombre de lignes du fichier fic_noms. Le fichier err est créé mais est vide car aucune erreur ne s'est produite. Dans cet exemple, la disposition des redirections dans la ligne de commande n'a pas d'importance, car les deux sorties ne sont pas redirigées vers le même fichier. Il est possible de placer les redirections où l'on souhaite car le shell traite les redirections avant d'exécuter la commande :

```
$ < fic_noms > fic_nblignes wc 2>err -l
$ cat fic_nblignes
3
$ cat err
$
```

La plupart des commandes affichent leurs résultats sous la même forme, suivant que l'on passe en argument le nom d'un fichier ou que l'on redirige son entrée standard avec ce fichier.

```
Ex :                                => cat ouvre le fichier fic afin de lire son contenu
$ cat fic
bonjour
et au revoir
$
$ cat <fic                          => cat lit son entrée standard (redirigée par le shell)
bonjour
et au revoir
$
```

Il n'en est pas ainsi avec la commande `wc` : celle-ci n'écrit pas les résultats de la même manière.

```
$ wc -l fic_noms
3 fic_noms
$ wc -l < fic_noms
3
$
```

Par conséquent, lorsque l'on désirera traiter la sortie d'une commande `wc`, il faudra prendre garde à la forme utilisée. La deuxième forme est préférable lorsque on ne souhaite que le nombre de lignes.

```
$ nb lignes=$( wc -l < fic_noms )
$ echo $nb lignes
3          => nombre de lignes
$
```

Texte joint :

Il existe plusieurs syntaxes légèrement différentes pour passer un texte joint comme entrée à une commande. Néanmoins, la syntaxe présentée ci-dessous est la plus simple.

```
Syntaxe :      cmd <<mot
                texte
                mot
```

L'utilisation de cette syntaxe permet d'alimenter l'entrée standard de la commande `cmd` à l'aide d'un contenu texte délimité par deux balises `mot`.

Aucun caractère espace ne doit être présent entre `<<` et `mot`.

```
$ a=3.5 b=1.2
$ bc <<EOF
> $a + $b
> EOF
4.7
$
```

La commande unix **bc** est une calculatrice utilisable en ligne de commande. Dans l'exemple ci-dessus, deux variables `a` et `b` sont initialisées respectivement avec les chaînes de caractères 3.5 et 1.2. Le shell effectue les deux substitutions de variables présentes entre les mots `EOF` puis alimente l'entrée standard de `bc` avec le texte obtenu. Cette commande calcule la valeur de l'expression fournie puis affiche son résultat sur sa sortie standard.

Fermeture des entrées / sorties standard :

Fermeture de l'entrée standard : `<&-`

Fermeture de la sortie standard : `>&-`

Fermeture de la sortie standard pour les messages d'erreur : `2>&-`

```
$ >&-      echo coucou
```



```
-bash: echo: write error: Mauvais descripteur de fichier
$
```

Il y a fermeture de la sortie standard puis une tentative d'écriture sur celle-ci : l'erreur est signalée par l'interpréteur de commandes.

V-3 - Tubes

Le mécanisme de tube (symbolisé par le caractère `|`) permet d'enchaîner l'exécution de commandes successives en connectant la sortie standard d'une commande à l'entrée standard de la commande suivante :

```
Ex : ls -l /bin | more
```

Il y a affichage du contenu du répertoire `/bin` (commande `ls -l /bin`) écran par écran (commande `more`). En effet, les informations affichées par `ls -l` sont envoyées vers l'entrée de la commande `more` qui les affiche écran par écran. La sortie standard de la dernière commande n'étant pas redirigée, l'affichage final s'effectue à l'écran.

```
ls -l /bin 1-----> | 0-----> more 1----->
2|
v
```

Un résultat équivalent aurait pu être obtenu d'une manière moins élégante en exécutant la suite de commandes :

```
ls -l /bin >temporaire ; more < temporaire ; rm temporaire
```

La commande `ls -l /bin` écrit le résultat dans le fichier temporaire ; ensuite, on affiche écran par écran le contenu de ce fichier ; enfin, on efface ce fichier devenu inutile. Le mécanisme de tube évite à l'utilisateur de gérer ce fichier intermédiaire.

Pipelines :

On appelle pipeline, une suite non vide de commandes connectées par des tubes :

```
cmd1 | ... | cmdn
```

Chaque commande est exécutée par un processus distinct, la sortie standard de la commande `cmd_i-1` étant connectée à l'entrée standard de la commande `cmd_i`.

```
$ date | tee trace1 trace2 | wc -l
1      => date n'écrit ses résultats que sur une seule ligne
$ cat trace1
lundi 20 novembre 2006, 15:37:49 (UTC+0100)
$ cat trace2
lundi 20 novembre 2006, 15:37:49 (UTC+0100)
$
```

La commande unix **tee** écrit le contenu de son entrée standard sur sa sortie standard tout en gardant une copie dans le ou les fichiers dont on a passé le nom en argument.

Dans l'exemple ci-dessus, tee écrit le résultat de date dans les fichiers trace1 et trace2 ainsi que sur sa sortie standard, résultat passé à la commande wc -l.

VI - Groupement de commandes

Le shell fournit deux mécanismes pour grouper l'exécution d'un ensemble de commandes sans pour cela affecter un nom à ce groupement :

- l'insertion de la suite de commandes entre une paire d'accolades
- l'insertion de cette suite de commandes entre une paire de parenthèses.

```
{ suite_cmds ; }
```

Insérée entre une paire d'accolades, la suite de commandes est exécutée dans le shell courant. Cela a pour effet de préserver les modifications apportées par la suite de commandes sur l'environnement du processus courant.

```
$ pwd
/home/sanchis      => répertoire initial
$ { cd /bin ; pwd ; }      => changement du répertoire courant
/bin
$ pwd
/bin               => répertoire final (le changement a été préservé !)
```

S'exécutant dans le shell courant, les modifications apportées aux variables sont également conservées :

```
$ a=bonjour
$ { a=coucou ; echo $a      ; }
coucou
$ echo $a
coucou
$
```

Les accolades { et } sont deux mots-clé de bash ; chacune d'elles doit donc être le premier mot d'une commande. Pour cela, chaque accolade doit être le premier mot de la ligne de commande ou bien être précédée d'un caractère ;. De plus, suite_cmds ne doit pas « coller » l'accolade ouvrante {. Par contre, la présence ou absence d'un caractère espace entre le caractère ; et le mot-clé } n'a aucune importance.

```
$ { cd /bin }      => l'accolade } n'est pas le premier mot d'une commande
>                => le shell ne détecte pas la fin du groupement
> ^C             => control-C
$ {cd /bin ;}      => il manque un espace ou une tabulation entre { et cd
-bash: syntax error near unexpected token `}'
$
```

Toutefois, lancée en arrière-plan, la suite de commandes n'est plus exécutée par le shell courant mais par un sous-shell.

```
$ pwd
/home/sanchis      => répertoire initial
$ { echo processus : ; cd / ; ps ; } &
processus :
[1]                27963
$
```

PID	TTY	TIME	CMD
-----	-----	------	-----

```
27942 pts/14      00:00:00 bash
27947 pts/14      00:00:00 bash
27963 pts/14      00:00:00 ps
[1] + Done          { echo processus ;; cd /; ps; }
$
$ pwd
/home/sanchis      => répertoire final
$
```

Bien qu'une commande `cd /` ait été exécutée, le répertoire courant n'a pas été modifié (`/home/sanchis`). Une utilisation fréquente du regroupement de commandes est la redirection globale de leur entrée ou sortie standard.

```
$ cat fich
premiere ligne
deuxieme ligne
troisieme ligne
quatrieme ligne
$ { read ligne1 ; read ligne2
> } < fich          => le caractère > est affiché par le shell, indiquant
                   => que l'accolade } est manquante
$ echo $ligne1
premiere ligne
$ echo $ligne2
deuxieme ligne
$
```

L'entrée standard des deux commandes `read ligne1` et `read ligne2` est globalement redirigée : elles liront séquentiellement le contenu du fichier `fich`. Si les commandes de lecture n'avaient pas été groupées, seule la première ligne de `fich` aurait été lue.

```
$ read ligne1 < fich ; read ligne2 < fich
$ echo $ligne1
premiere ligne
$ echo $ligne2
premiere ligne
$
```

Comme pour toutes commandes composées, lorsque plusieurs redirections de même type sont appliquées à la même suite de commandes, seule la redirection la plus proche est effective.

```
$ { read lig1 ; read -p "Entrez votre ligne : " lig2 </dev/tty
> read lig3
> } < fich
Entrez votre ligne : bonjour tout le monde
$ echo $lig1
premiere ligne
$ echo $lig2
bonjour tout le monde
$ echo $lig3
deuxieme ligne
$
```

Les commandes `read lig1` et `read lig3` lisent le contenu du fichier `fich` mais la commande `read -p "Entrez votre ligne : " lig2` lit son entrée standard (`/dev/tty`).

(suite_cmds)

Insérée entre une paire de parenthèses, la suite de commandes est exécutée dans un sous-shell. L'environnement du processus n'est pas modifié après exécution de la suite de commandes. Les parenthèses sont des opérateurs (et non des mots clé) : suite_cmds peut par conséquent coller une ou deux parenthèses sans provoquer une erreur de syntaxe.

```
$ pwd
/home/sanchis                => répertoire initial
$ ( cd /bin ; pwd )
/bin
$ pwd
/home/sanchis                => le répertoire initial n'a pas été modifié
$ b=bonjour
$ ( b=coucou ; echo $b )
coucou
$ echo $b
bonjour                      => la valeur de la variable b n'a pas été modifiée
$
```

Ce type de groupement de commandes peut être aussi utilisé pour effectuer une redirection globale.

```
$ ( pwd ; date ; echo FIN ) > fin
$ cat fin
/home/sanchis
lundi 20 novembre 2006, 15:47:45 (UTC+0100)
FIN
$
```

VII - Code de retour

Un code de retour (exit status) est fourni par le shell après exécution d'une commande. Le code de retour est un entier positif ou nul, compris entre 0 et 255, indiquant si l'exécution de la commande s'est bien déroulée ou s'il y a eu un problème quelconque.

Par convention, un code de retour égal à 0 signifie que la commande s'est exécutée correctement. Un code différent de 0 signifie une erreur syntaxique ou d'exécution.

L'évaluation du code de retour est essentielle à l'exécution de structures de contrôle du shell telles que if et while.

VII-1 - Paramètre spécial ?

Le paramètre spécial ? (à ne pas confondre avec le caractère générique ?) contient le code de retour de la dernière commande exécutée de manière séquentielle (exécution synchrone).

```
$ pwd
/home/sanchis
$ echo $?
0                                => la commande pwd s'est exécutée correctement
$ ls -l vi
ls: vi: Aucun fichier ou répertoire de ce type
$ echo $?
2                                => une erreur s'est produite !
$
```

Exercice :

En utilisant la commande unix ls et le mécanisme de redirection, écrire un programme shell dansbin prenant un nom de commande en argument et qui affiche 0 si cette commande est présente dans /bin, une valeur différente de 0 sinon.

```
$ dansbin    ls
0
$ dansbin    who
2
$
```

Chaque commande positionne « à sa manière » les codes de retour différents de 0. Ainsi, un code de retour égal à 1 positionné par la commande unix ls n'a pas la même signification qu'un code de retour à 1 positionné par la commande unix grep. Les valeurs et significations du code de retour d'une commande unix ou du shell sont documentées dans les pages correspondantes du manuel (ex : man grep).

Lorsque une commande est exécutée en arrière-plan (exécution asynchrone), son code de retour n'est pas mémorisé dans le paramètre spécial ?.

```
$ pwd          => mise à zéro du paramètre spécial ?
/home/sanchis
$ echo $?
0
$ ls -l vi &   => commande exécutée en arrière-plan
[1] 19331
$ ls: vi: Aucun fichier ou répertoire de ce type
[1]+ Exit 2      ls --color=tty -l vi
```

```
$ echo $?  
0  
$
```

On remarque que la commande s'est terminée avec la valeur 2 (Exit 2) mais que ce code de retour n'a pas été enregistré dans le paramètre ?. La commande interne deux-points (:) sans argument retourne toujours un code de retour égal à 0.

```
$:  
$ echo $?  
0  
$
```

=> commande deux-points

Il en est de même avec la commande interne echo : elle retourne toujours un code de retour égal à 0, sauf cas particuliers.

```
$ 1>&- echo coucou  
-bash: echo: write error: Mauvais descripteur de fichier  
$ echo $?  
1  
$
```

Enfin, certaines commandes utilisent plusieurs valeurs pour indiquer des significations différentes, comme la commande unix grep.

Commande unix grep :

Cette commande affiche sur sa sortie standard l'ensemble des lignes contenant une chaîne de caractères spécifiée en argument, lignes appartenant à un ou plusieurs fichiers texte (ou par défaut, son entrée standard).


La syntaxe de cette commande est : **grep [option(s)] chaîne_cherchée [fich_texte1 ...]**

Soit le fichier pass contenant les cinq lignes suivantes :

```
root:x:0:0:root:/root:/bin/bash  
daemon:x:1:1:daemon:/usr/sbin:/bin/sh  
bertrand:x:101:100::/home/bertrand:/bin/bash  
albert:x:102:100::/home/albert:/bin/bash  
sanchis:x:103:100::/home/sanchis:/bin/bash
```

La commande ci-dessous affiche toutes les lignes du fichier pass contenant la chaîne sanchis.

```
$ grep sanchis pass  
sanchis:x:103:100::/home/sanchis:/bin/bash  
$
```

 **grep recherche une chaîne de caractères et non un mot.**

```
$ grep bert pass  
bertrand:x:101:100::/home/bertrand:/bin/bash  
albert:x:102:100::/home/albert:/bin/bash
```

```
$
```

La commande affiche toutes les lignes contenant la chaîne bert (et non le mot bert). Si l'on souhaite la chaîne cherchée en début de ligne, on utilisera la syntaxe `##^chaîne_cherchée##`. Si on la veut en fin de ligne : `##chaîne_cherchée###`

```
$ grep ##^bert## pass
bertrand:x:101:100::/home/bertrand:/bin/bash
$
```

La commande unix **grep** positionne un code de retour

- égal à 0 pour indiquer qu'une ou plusieurs lignes ont été trouvées
- égal à 1 pour indiquer qu'aucune ligne n'a été trouvée
- égal à 2 pour indiquer la présence d'une erreur de syntaxe ou qu'un fichier mentionné en argument est inaccessible.

```
$ grep sanchis pass
sanchis:x:103:100::/home/sanchis:/bin/bash
$ echo $?
0
$ grep toto pass
$ echo $?
1      => la chaîne toto n'est pas présente dans pass
$ grep sanchis turlututu
grep: turlututu: Aucun fichier ou répertoire de ce type
$ echo $?
2      => le fichier turlututu n'existe pas !
$
```

Exercice :

Ecrire un programme shell connu prenant en argument un nom d'utilisateur qui affiche 0 s'il est enregistré dans le fichier pass, 1 sinon.

```
$ connu bert
1
$ connu albert
0
$
```

VII-2 - Code de retour d'un programme shell

Le code de retour d'un programme shell est le code de retour de la dernière commande qu'il a exécutée.

Soit le programme shell lvi contenant l'unique commande ls vi.

```
#!/bin/bash
#      @(#)      lvi
ls vi
```

Cette commande produira une erreur car vi ne se trouve pas dans le répertoire courant ; après exécution, le code de retour de lvi sera de celui de la commande ls vi (dernière commande exécutée).


```
$ lvi
ls: vi: Aucun fichier ou répertoire de ce type
$ echo $?
2                => code de retour de la dernière commande exécutée par lvi
                  => c.-à-d. ls vi
$
```

Autre exemple avec le programme shell lvi1 de contenu :

```
#!/bin/bash
#      @(#)      lvi1
ls vi
echo Fin
```

La dernière commande exécutée par lvi1 sera la commande interne echo qui retourne un code de retour égal à 0.

```
$ lvi1
ls: vi: Aucun fichier ou répertoire de ce type
Fin
$ echo $?
0  => code de retour de la dernière commande exécutée par lvi (echo Fin)
$
```

Il est parfois nécessaire de positionner explicitement le code de retour d'un programme shell avant qu'il ne se termine : on utilise alors la commande interne **exit**.

VII-3 - Commande interne exit

Syntaxe : `exit [n]`

Elle provoque l'arrêt du programme shell avec un code de retour égal à n. Si n n'est pas précisé, le code de retour fourni est celui de la dernière commande exécutée. Soit le programme shell lvi2 :

```
#!/bin/bash
#      @(#)      lvi2
ls vi
exit 23
```

```
$ lvi2
ls: vi: Aucun fichier ou répertoire de ce type
$ echo $?
23                => code de retour de exit 23
$
```

Le code de retour renvoyé par ls vi est 2 ; en utilisant la commande interne exit, le programme shell lvi2 positionne un code de retour différent (23).

VII-4 - Code de retour d'une suite de commandes

Le code de retour d'une suite de commandes est le code de retour de la dernière commande exécutée.

Le code de retour de la suite de commandes **cmd1 ; cmd2 ; cmd3** est le code de retour de la commande cmd3.

```
$ pwd ; ls vi ; echo bonjour
/home/sanchis
ls: vi: Aucun fichier ou répertoire de ce type
bonjour
$ echo $?
0                                => code de retour de echo bonjour
$
```

Il en est de même pour le pipeline **cmd1 | cmd2 | cmd3**. Le code de retour sera celui de cmd3.

```
$ cat pass | grep sanchis
sanchis:x:103:100::/home/sanchis:/bin/bash
$ echo $?
0                                => code de retour de grep sanchis
$
```

Il est possible d'obtenir la négation d'un code de retour d'un pipeline en plaçant le mot-clé **!** devant celui-ci. Cela signifie que si le code de retour de pipeline est égal à 0, alors le code de retour de **!** pipeline est égal à 1.

```
$ ! ls pass => le code de retour de ls passest égal à 0
pass
$ echo $?
1
$ ! cat pass | grep daemon
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
$ echo $?
1
$
```

Inversement, si le code de retour de pipeline est différent de 0, alors celui de **!** pipeline est égal à 0.

```
$ ! grep sanchis turlututu
grep: turlututu: Aucun fichier ou répertoire de ce type
$ echo $?
0
$
```

VII-5 - Résultats et code de retour

On ne doit pas confondre le résultat d'une commande et son code de retour : le résultat correspond à ce qui est écrit sur sa sortie standard ; le code de retour indique uniquement si l'exécution de la commande s'est bien effectuée ou non.

Parfois, on est intéressé uniquement par le code de retour d'une commande et non par les résultats qu'elle produit sur la sortie standard ou la sortie standard pour les messages d'erreurs.

```
$ grep toto pass > /dev/null 2>&1 => ou bien : grep toto pass &>/dev/null
$ echo $?
1                                => on en déduit que la chaîne toto n'est pas présente dans pass
$
```

VII-6 - Opérateurs && et || sur les codes de retour

Les opérateurs && et || autorisent l'exécution conditionnelle d'une commande cmd suivant la valeur du code de retour de la dernière commande précédemment exécutée.

Opérateur : &&

Syntaxe : `cmd1 && cmd2`

Le fonctionnement est le suivant : cmd1 est exécutée et si son code de retour est égal à 0, alors cmd2 est également exécutée.

```
$ grep daemon pass && echo daemon existe
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
daemon existe
$
```

La chaîne de caractères daemon est présente dans le fichier pass, le code de retour renvoyé par l'exécution de grep est 0 ; par conséquent, la commande echo daemon existe est exécutée.

Opérateur : ||

Syntaxe : `cmd1 || cmd2`

cmd1 est exécutée et si son code de retour est différent de 0, alors cmd2 est également exécutée. Pour illustrer cela, supposons que le fichier toto n'existe pas.

```
$ ls pass toto
ls: toto: Aucun fichier ou répertoire de ce type
pass
$
$ rm toto || echo toto non efface
rm: ne peut enlever `toto': Aucun fichier ou répertoire de ce type
toto non efface
$
```

Le fichier toto n'existant pas, la commande rm toto affiche un message d'erreur et produit un code de retour différent de 0 : la commande interne echo qui suit est donc exécutée.

Combinaisons d'opérateurs && et ||

Les deux règles mentionnées ci-dessus sont appliquées par le shell lorsqu'une suite de commandes contient plusieurs opérateurs && et ||. Ces deux opérateurs ont même priorité et leur évaluation s'effectue de gauche à droite.

```
$ ls pass || ls toto || echo fini aussi
pass
$
```

Le code de retour de `ls pass` est égal à 0 car `pass` existe, la commande `ls toto` ne sera donc pas exécutée. D'autre part, le code de retour de l'ensemble `ls pass || ls toto` est le code de retour de la dernière commande exécutée, c'est à dire est égal à 0 (car c'est le code de retour de `ls pass`), donc `echo fini` aussi n'est pas exécutée.

Intervertissons maintenant les deux commandes `ls` :

```
$ ls toto || ls pass || echo fini
ls: toto: Aucun fichier ou répertoire de ce type
pass
$
```

Le code de retour de `ls toto` est différent de 0, donc `ls pass` s'exécute. Cette commande renvoie un code de retour égal à 0, par conséquent `echo fini` n'est pas exécutée.

Combinons maintenant opérateurs `&&` et `||` :

```
$ ls pass || ls toto || echo suite et && echo fin
pass
fin
$
```

La commande `ls pass` est exécutée avec un code de retour égal à 0, donc la commande `ls toto` n'est pas exécutée : le code de retour de l'ensemble `ls pass || ls toto` est égal à 0, la commande `echo suite et` n'est pas exécutée. Le code de retour de `ls pass || ls toto || echo suite et` est égal à 0, donc la commande `echo fin` est exécutée !

Bien sûr, un raisonnement analogue s'applique avec l'opérateur `&&` :

```
$ ls pass && ls toto && echo fini
pass
ls: toto: Aucun fichier ou répertoire de ce type
$ ls toto && ls pass && echo suite et && echo fin
ls: toto: Aucun fichier ou répertoire de ce type
$ ls pass && ls toto && echo suite et || echo fin
pass
ls: toto: Aucun fichier ou répertoire de ce type
fin
$
```

VIII - Structures de contrôle case et while

VIII-1 - Choix multiple case

Syntaxe :

```
case mot in
[ modèle [ | modèle ] ... ) suite_de_commandes ;; ] ...
esac
```

Le shell évalue la valeur de *mot* puis compare séquentiellement cette valeur à chaque modèle. Dès qu'un modèle correspond à la valeur de *mot*, la suite de commandes associée est exécutée, terminant l'exécution de la commande interne composée *case*.

Les mots *case* et *esac* sont des mots-clé ce qui signifie que chacun d'eux doit être le premier mot d'une commande.

suite_de_commandes doit se terminer par deux caractères point-virgule collés, de manière à ce qu'il n'y ait pas d'ambiguïté avec l'enchaînement séquentiel de commandes *cmd1 ; cmd2*.

Un *modèle* peut être construit à l'aide des caractères et expressions génériques de **bash** [cf. § *Caractères et expressions génériques*]. Dans ce contexte, le symbole *|* signifie OU.

Pour indiquer le cas par défaut, on utilise le modèle ***. Ce modèle doit être placé à la fin de la structure de contrôle *case*.

Le code de retour de la commande composée *case* est celui de la dernière commande exécutée de *suite_de_commandes*.

Exemple 1 : programme shell *oui* affichant *OUI* si l'utilisateur a saisi le caractère *o* ou *O*

```
#!/bin/bash
# @(#) oui
read .p ##Entrez votre reponse : ## rep
case $rep in
o|O ) echo OUI ;;
*) echo Indefini
esac
```

Rq : il n'est pas obligatoire de terminer par *::* la dernière *suite_de_commandes*

Exemple 2 : programme shell *nombre* prenant une chaîne de caractères en argument et qui affiche cette chaîne si elle est constituée d'une suite de chiffres

```
#!/bin/bash
# @(#) nombre
shop s case$1in
t-extglob
[:digit:])) ) echo ##$1 est une suite de chiffres## ;;
+([esac
```

Exercice : Ecrire un programme shell *4arg* qui vérifie que le nombre d'arguments passés lors de l'appel du programme est égal à 4 et écrit suivant le cas le message *#Correct#* ou le message *#Erreur#*.

Exercice : Ecrire un programme shell **reconnaitre** qui demande à l'utilisateur d'entrer un mot, puis suivant le première caractère de ce mot, indique s'il commence par un chiffre, une minuscule, une majuscule ou une autre sorte de caractère.

Exercice : En utilisant la commande unix **date** et la locale standard [cf. *Substitution de commandes*], écrire un programme datefr qui affiche la date courante de la manière suivante : *mercredi 5 janvier 2005 10:00*

VIII-2 - Itération while

La commande interne **while** correspond à l'itération *tant que* présente dans de nombreux langages de programmation.

Syntaxe :

```
while suite_cmd1
do
suite_cmd2
done
```

La suite de commandes *suite_cmd1* est exécutée; **si son code de retour est égal à 0**, alors la suite de commandes *suite_cmd2* est exécutée, puis *suite_cmd1* est re-exécutée. Si son code de retour est différent de 0, alors l'itération se termine.

En d'autres termes, *suite_cmd2* est exécutée autant de fois que le code de retour de *suite_cmd1* est égal à zéro.

L'originalité de cette structure de contrôle est que le test ne porte pas sur une condition booléenne (vraie ou fausse) mais sur le code de retour issu de l'exécution d'une suite de commandes.

En tant que mots-clé, **while**, **do** et **done** doivent être les premiers mots d'une commande.

Une commande **while**, comme toute commande interne, peut être écrite directement sur la ligne de commande.

Ex :

```
$ while who | grep sanchis >/dev/null
> do
> echo ##l#utilisateur sanchis est encore connecte##
> sleep 5
> done
l#utilisateur sanchis est encore connecte
^C
$
```

Le fonctionnement est le suivant : la suite de commandes *who | grep sanchis* est exécutée. Son code de retour sera égal à 0 si le mot *sanchis* est présent dans les résultats engendrés par l'exécution de la commande unix **who**, c'est à dire si l'utilisateur *sanchis* est connecté. Dans ce cas, la ou les lignes correspondant à cet utilisateur seront affichées sur la sortie standard de la commande **grep**. Comme seul le code de retour est intéressant et non le résultat, la sortie standard de **grep** est redirigée vers le puits (*/dev/null*). Enfin, le message est affiché et le programme « s'endort » pendant 5 secondes. Ensuite, la suite de commandes *who | grep sanchis* est ré exécutée. Si l'utilisateur s'est totalement déconnecté, la commande **grep** ne trouve aucune ligne contenant la chaîne *sanchis*, son code de retour sera égal à 1 et le programme sortira de l'itération.

Commandes internes **while** et **deux-points**

La commande interne **deux-points** associée à une itération **while** compose rapidement un serveur (démon) rudimentaire.

```
Ex : $ while : => boucle infinie
> do
> who | cut -d' ' -f1 >>fic => traitement à effectuer
> sleep 300 => temporisation
> done &
[1] 12568
$
```

Lecture du contenu d'un fichier texte

La commande interne **while** est parfois utilisée pour lire le contenu d'un fichier texte. La lecture s'effectue alors ligne par ligne. Il suffit pour cela :

- de placer une commande interne **read** dans *suite_cmd1*
- de placer les commandes de traitement de la ligne courante dans *suite_cmd2*
- de rediriger l'entrée standard de la commande **while** avec le fichier à lire.

```
Syntaxe : while read [ var1 ... ]
do
commande(s) de traitement de la ligne courante
done < fichier_à_lire
```

Exemple : programme **wh** qui affiche les noms des utilisateurs connectés

```
#!/bin/bash
# @(#) wh
who > tmp
while read nom reste
do
edone < tmp
cho $nom
rm tmp
```

Exercice : On dispose d'un fichier *personnes* dont chaque ligne est constituée du prénom et du genre (*m* pour masculin, *f* pour féminin) d'un individu.

```
Ex : $ cat personnes
arthur m
pierre m
dominique f
paule f
sylvie f
jean m
$
```

Ecrire un programme shell **tripersonnes** qui crée à partir de ce fichier, un fichier *garcons* contenant uniquement les prénoms des garçons et un fichier *filles* contenant les prénoms des filles.

```
Ex : $ tripersonnes
```

```
$  
$ cat filles  
dominique  
paule  
sylvie  
$  
$ cat garcons  
arthur  
pierre  
jean  
$
```

Lorsque le fichier à lire est créé par une commande **cmd**, comme dans le programme **wh**, on peut utiliser la syntaxe :

```
cmd | while read [ var1 ... ]  
do  
commande(s) de traitement de la ligne courante  
done
```

Exemple : **wh1**

```
#!/bin/bash  
# @(#) wh1  
who | while read nom reste  
do  
echo $nom  
done
```

Par rapport au programme shell **wh**, il est inutile de gérer le fichier temporaire **tmp**.

Exercice : En utilisant la commande **ruptime** écrire un programme shell **up** qui affiche le nom de toutes les machines qui sont dans l'état **up**

Exercice :

- Écrire un programme shell **etat1** prenant un nom de machine en argument et qui affiche son état (**up** ou **down**). On ne considère aucun cas d'erreur.
- Modifier ce programme, soit **etat2**, pour qu'il affiche un message d'erreur lorsque le nom de la machine passé en argument n'est pas visible à l'aide de la commande **ruptime**.
- Modifier le programme précédent, soit **etat**, pour qu'il vérifie également le nombre d'arguments passés lors de l'appel.

Commande interne **while** et performances

La lecture ligne par ligne d'un fichier à l'aide d'une commande interne **while** est lente et peu élégante. Il est préférable d'utiliser une suite de filtres permettant d'aboutir au résultat voulu.

Par exemple, en utilisant un filtre supplémentaire (la commande unix **cut**), on peut s'affranchir de l'itération **while** dans le programme **wh1**. Il suffit d'extraire le premier champ de chaque ligne.

La commande unix **cut** permet de sélectionner un ou plusieurs champs de chaque ligne d'un fichier texte. Un champ peut être spécifié en précisant le caractère séparateur de champ (par défaut, il s'agit du caractère **tabulation**) avec l'option **#d** ; les numéros de champs doivent alors être indiqués avec l'option **-f**.

Programme **wh1.cut**

```
-----  
#!/bin/bash  
# @(#) wh1.cut  
who | cut -d ' ' -f1  
-----
```

```
Ex : $ wh1.cut  
sanchis  
root  
$
```

Dans le programme *wh1.cut*, on précise que la commande **cut** doit prendre comme séparateur le caractère **espace** (`-d ' '`) et que seul le premier champ de chaque ligne doit être extrait (`-f1`).

Exercice : Ecrire un programme shell *up.cut* qui, sans utiliser d'itération, affiche le nom de toutes les machines qui sont dans l'état *up*.

Une deuxième raison d'éviter la lecture ligne à ligne d'un fichier avec **while** est qu'elle peut conduire à des résultats différents suivant le mode de lecture choisi (tube ou simple redirection).

Prenons l'exemple où l'on souhaite modifier la valeur d'une variable *var* après lecture de la première ligne d'un fichier.

Le programme shell *maj* ci-dessous connecte à l'aide d'un tube la sortie standard de la commande *date* à l'entrée de la commande interne *while*. Après lecture de la première (et unique) ligne, la valeur de la variable *var* est modifiée. Pourtant, cette nouvelle valeur ne sera pas affichée. En effet, l'utilisation du tube a pour effet de faire exécuter l'itération *while* par un processus différent : il y a alors deux instances différentes de la variable *var* : celle qui a été initialisée à 0 au début de l'exécution de *maj* et celle interne au nouveau processus qui initialise à 1 sa propre instance de *var*. Après terminaison de l'itération *while*, le processus qui l'exécutait disparaît ainsi que sa variable *var* initialisée à 1.

Programme **maj**

```
-----  
#!/bin/bash  
# @(#) maj  
var=0  
date | while read  
do  
  var=1  
done  
echo $var  
-----
```

```
Ex : $ maj  
0  
$
```

Lorsque l'on exécute le programme *maj1*, **bash** ne crée pas un nouveau processus pour exécuter l'itération **while** : il n'y a qu'une instance de la variable *var* qui est convenablement mise à jour.

Programme **maj1**

```
- - - - -
#!/bin/bash
# @(#) majl
var=0
date > tmp
while read
do
var=1
done < tmp
rm tmp
echo $var
- - - - -
```

```
Ex : $ majl
1
$
```

IX - Chaînes de caractères

IX-1 - Protection de caractères

Le shell utilise différents caractères particuliers pour effectuer ses propres traitements (\$) pour la substitution, > pour la redirection de la sortie standard, * comme caractère générique, etc.). Pour utiliser ces caractères particuliers comme de simples caractères, il est nécessaire de les protéger de l'interprétation du shell. Trois mécanismes sont utilisables :

Protection d'un caractère à l'aide du caractère \

Ce caractère protège le caractère qui suit immédiatement le caractère \.

```
Ex : $ echo \* => le caractère * perd sa signification de caractère générique
*
$ echo *
tata toto
$
$ echo \\ => le deuxième caractère \ perd sa signification de caractère de protection
\
$ echo N\'oublie pas !
N'oublie pas !
$
```

Rq : dans les exemples de cette section, les caractères **en vert** ont perdu leur signification particulière, les caractères **en rouge** sont interprétés.

Le caractère \ permet également d'ôter la signification de la touche **Entrée**. Cela a pour effet d'aller à la ligne sans qu'il y ait exécution de la commande. En effet, après saisie d'une commande, l'utilisateur demande au shell l'exécution de celle-ci en appuyant sur cette touche. Annuler l'interprétation de la touche **Entrée** autorise l'utilisateur à écrire une longue commande sur plusieurs lignes.

Dans l'exemple ci-dessous, le shell détecte que la commande interne **echo** n'est pas terminée ; par conséquent, **bash** affiche une chaîne d'appel différente matérialisée par un caractère > suivi d'un caractère **espace** invitant l'utilisateur à continuer la saisie de sa commande.

```
Ex : $ echo coucou \Entrée
> salut Entrée => terminaison de la commande : le shell l'exécute !
coucou salut
$
```

Protection de caractères à l'aide d'une paire de guillemets "chaîne"

Tous les caractères de *chaîne* sauf \$ \ ` " sont protégés de l'interprétation du shell. Cela signifie, par exemple, qu'à l'intérieur d'une paire de guillemets le caractère \$ sera quand même interprété comme une substitution, etc.

```
Ex : $ echo "< * $PWD * >"
< * /home/sanchis * >
$
$ echo "< * \"$PWD\" * > "
< * "/home/sanchis" * >
$
```

Protection totale 'chaîne'

Entre une paire d'**apostrophes** ('), aucun caractère de chaîne (sauf le caractère ') n'est interprété.

```
Ex : $ echo '< * $PWD * >'  
< * $PWD * >  
$
```

IX-2 - Longueur d'une chaîne de caractères

Syntaxe : `${#paramètre}`

Cette syntaxe est remplacée par la longueur de la chaîne de caractères contenue dans *paramètre*. Ce dernier peut être une variable, un paramètre spécial ou un paramètre de position.

```
Ex : $ echo $PWD  
/home/sanchis  
ome/sanchis $ echo ${#PWD}  
14 => longueur de la chaîne /home/sanchis  
$  
$ set "au revoir"  
$ echo ${#1}  
9 => la valeur de $1 étant au revoir, sa longueur est 9  
$  
$ ls >/dev/null  
$  
$ echo ${#?}  
1 => contenue dans $?, la valeur du code de retour de ls  
$ => est 0, par conséquent la longueur est 1
```

IX-3 - Modificateurs de chaînes

Les modificateurs de chaînes permettent la suppression d'une sous-chaîne de caractères correspondant à un modèle exprimé à l'aide de caractères ou d'expressions génériques.

Suppression de la plus courte sous-chaîne à gauche

Syntaxe : `${paramètre#modèle}`

```
Ex : $ echo $PWD  
/home/sanchis  
$  
$ echo ${PWD#/}  
home/sanchis => le premier caractère / a été supprimé  
$  
$ set "12a34a"  
$  
$ echo ${1#a}  
34a => suppression de la sous-chaîne 12a  
$
```

Suppression de la plus longue sous-chaîne à gauche

Syntaxe : `${paramètre##modèle}`

```
Ex : $ echo $PWD
/home/sanchis
$
$ echo ${PWD##*/}
sanchis => suppression de la plus longue sous-chaîne à gauche se
$ => terminant par le caractère /
$ set 12a34ab
$
$ echo ${1##*a}
b
$
```

Suppression de la plus courte sous-chaîne à droite

Syntaxe : `${paramètre%modèle}`

```
Ex: $ echo $PWD
/home/sanchis
$ echo ${PWD%/*}
/home
$
```

Suppression de la plus longue sous-chaîne à droite

```
Syntaxe : ${paramètre%%modèle}
Ex : $ eleve="Pierre Dupont::12:10::15:9"
$
$ echo ${eleve%%:*}
Pierre Dupont
$
```

La variable *eleve* contient les prénom, nom et diverses notes d'un élève. Les différents champs sont séparés par un caractère **deux-points**. Il peut manquer des notes à un élève (cela se caractérise par un champ vide).

Exercices :

1. En utilisant les modificateurs de chaînes,

- écrire un programme shell *touslesutil* qui lit le contenu du fichier **/etc/passwd** et affiche le nom des utilisateurs enregistrés.
- modifier ce programme (soit *touslesutiluid*) pour qu'il affiche le nom et l'uid de chaque utilisateur.

2. En utilisant les modificateurs de chaînes,

écrire un programme shell *basenom* ayant un fonctionnement similaire à la commande unix **basename**. Cette commande affiche le dernier élément d'un chemin passé en argument. Il n'est pas nécessaire que ce chemin existe réellement.

```
Ex : $ basename /toto/tata/tutu
tutu
$
```

b) si un suffixe est mentionné comme deuxième argument, celui ci est également ôté de l'élément par la commande **basename**.

```
Ex : $ basename /toto/tata/tutu/prog.c .c
prog
$
```

Ecrire un programme *basenom1* qui se comporte de la même manière.

3. Ecrire une commande *dirnom*, similaire à la commande unix **dirname** ; la commande unix **dirname** peut être vue comme la commande complémentaire à **basename** car **dirname** supprime le dernier élément d'un chemin; si cela donne la chaîne vide, alors le caractère **.** est affiché.

```
Ex : $ dirname /home/sanchis/bin
/home/sanchis
$
$ dirname toto
$
```

IX-4 - Extraction de sous-chaînes

`${paramètre:ind}` :

extrait de la valeur de *paramètre* la sous - chaîne débutant à l'indice *ind*. La valeur de *paramètre* n'est pas modifiée.

Attention : l'indice du premier caractère d'une chaîne est **0**

```
Ex : ch="abcdefghijk" echo ${ch:3} affiche defghijk
01234567..10
${paramètre:ind:nb} : extrait nb caractères à partir de l'indice ind
Ex : $ echo ${ch:8:2}
ij
$ set ABCDEFGH
$
$ echo ${1:4:3}
EFG
$
```

Exercice : a) Ecrire un programme *calibre* prenant deux arguments, une chaîne de caractères *ch* et un nombre *nb*, qui affiche les *nb* premiers caractères de *ch*. Aucune erreur ne doit être traitée.

b) Modifier le programme précédent, soit *calibre1*, pour qu'il vérifie que :

- le nombre d'arguments est correct
- le deuxième argument est un nombre (suite non vide de chiffres).

IX-5 - Remplacement de sous-chaînes

`${paramètre/mod/ch}` :

bash recherche dans la valeur de *paramètre* la plus longue sous-chaîne satisfaisant le modèle *mod* puis remplace cette sous-chaîne par la chaîne *ch*. Seule la première sous-chaîne trouvée est remplacée. La valeur de *paramètre* n'est pas modifiée. Caractères et expressions génériques peuvent être présents dans *mod*.

Ce mécanisme de remplacement comprend plusieurs aspects :

(1) Remplacement de la première occurrence

```
Ex : $ v=totito
$ echo ${v/to/lo}
lotito
$
```

La valeur de la variable *v* (*totito*) contient deux occurrences du modèle *to*. Seule la première occurrence est remplacée par la chaîne *lo*.

```
Ex : $ set topo
$ echo $1
topo
$
$ echo ${1/o/i}
tipo
$
```

(2) Remplacement de la plus longue sous-chaîne

```
Ex : $ v=abcfefg
$ v1=${v/b*f/toto} => utilisation du caractère générique *
$ echo $v1
atotog
$
```

Deux sous-chaînes de *v* satisfont le modèle *b*f* : *bcf* et *bcfef*

C'est la plus longue qui est remplacée par *toto*.

```
${paramètre//mod/ch} :
```

contrairement à la syntaxe précédente, toutes les occurrences (et non seulement la première) satisfaisant le modèle *mod* sont remplacées par la chaîne *ch*

```
Ex : $ var=tobatoba
$ echo ${var//to/tou}
toubatouba
$
$ set topo
$ echo $1
topo
$
$ echo ${1//o/i}
tipi
$
```

```
${paramètre//mod/} :
```

lorsque la chaîne **ch** est absente, la première ou toutes les occurrences (suivant la syntaxe utilisée) sont supprimées

```
Ex : $ v=123azer45ty
$ shopt #s extglob
$ echo ${v//+([[:lower:]))/}
12345
$
```

L'expression générique **+([[:lower:]])** désigne la plus longue suite non vide de minuscules. La syntaxe utilisée signifie que toutes les occurrences doivent être traitées : la variable **v** contient deux occurrences (**azer** et **ty**). Le traitement à effectuer est la suppression.

Il est possible de préciser si l'on souhaite l'occurrence cherchée en début de chaîne de **paramètre** (syntaxe à utiliser : **#mod**) ou bien en fin de chaîne (**%mod**).

```
Ex : $ v=automoto
$ echo ${v/#aut/vel}
velomoto
$
$ v=automoto
$ echo ${v/%to/teur}
automoteur
$
```


X - Structures de contrôle for et if

X-1 - Itération for

L'itération **for** possède plusieurs syntaxes dont les deux plus générales sont :

Première forme :

```
for var
do
suite_de_commandes
done
```

Lorsque cette syntaxe est utilisée, la variable **var** prend successivement la valeur de chaque paramètre de position initialisé.

Exemple : programme **for_arg**

```
- - - - -
#!/bin/bash
for i
do
echo $i
echdone
o "Passage a l'argument suivant ..."
- - - - -
```

Ex : \$ **for_arg** **un** **deux** => deux paramètres de position initialisés

```
un
Passage a l'argument suivant ...
deux
Passage a l'argument suivant ...
$
```

Exercice : En utilisant la commande **ruptime**, écrire un programme shell **etatgene** prenant en arguments un ou plusieurs noms de machines et affiche pour chacune d'elles leur état (*up* ou *down*).

La commande composée **for** traite les paramètres de position sans tenir compte de la manière dont ils ont été initialisés (lors de l'appel d'un programme shell ou bien par la commande interne set).

Exemple : programme **for_set**

```
- - - - -
#!/bin/bash
set $(date)
for i
do
echo $i
done
- - - - -
```

```
Ex : $ for_set
dimanche
17
décembre
2006,
11:22:10
(UTC+0100)
$
```

Deuxième forme :

```
for var in liste_mots
do
suites_de_commandes
done
```

La variable **var** prend successivement la valeur de chaque mot de **liste_mots**.

Exemple : programme **for_liste**

```
- - - - -
#!/bin/bash
for a in toto tata
do
echo $a
done
- - - - -
```

```
Ex : $ for_liste
toto
tata
$
```

Si **liste_mots** contient des substitutions, elles sont préalablement traitées par **bash**.

Exemple : programme **affich.ls**

```
- - - - -
#!/bin/bash
for i in tmp $(pwd)
do
echo # --- $i ---#
ls $i
done
- - - - -
```

```
Ex : $ affich.ls
--- tmp ---
gamma
--- /home/sanchis/Rep ---
affich.ls alpha beta tmp
$
```

Exercice : Ecrire un programme shell **lsrep** ne prenant aucun argument, qui demande à l'utilisateur de saisir une suite de noms de répertoires et qui affiche leur contenu respectif.

X-2 - Choix if

La commande interne **if** implante le choix alternatif.

Syntaxe :

```
if suite_de_commandes1
then
suite_de_commandes2
[ elif suite_de_commandes ; then suite_de_commandes ] ...
[ else suite_de_commandes ]
fi
```

Le fonctionnement est le suivant : *suite_de_commandes1* est exécutée ; si son code de retour est égal à **0**, alors *suite_de_commandes2* est exécutée sinon c'est la branche **elif** ou la branche **else** qui est exécutée, si elle existe.

Exemple : programme **rm1**

```
- - - - -
#!/bin/bash
if rm $1 2>/dev/null
then echo $1 a ete supprime
else echo $1 n'a pas ete supprime
fi
- - - - -
```

```
Ex : $ >toto => création du fichier toto
$
$ rml toto
toto a ete supprime
$
$ rml toto
toto n'a pas ete supprime
$
```

Lorsque la commande **rm1 toto** est exécutée, si le fichier *toto* est effaçable, le fichier est effectivement supprimé, la commande unix **rm** renvoie un code de retour égal à 0 et c'est la suite de commandes qui suit le mot-clé **then** qui est exécutée ; le message *toto a ete supprime* est affiché sur la sortie standard.

Si *toto* n'est pas effaçable, l'exécution de la commande **rm** échoue ; celle-ci affiche un message sur la sortie standard pour les messages d'erreur que l'utilisateur ne voit pas car celle-ci a été redirigée vers le puits, puis renvoie un code de retour différent de 0. C'est la suite de commandes qui suit **else** qui est exécutée : le message *toto n'a pas ete supprime* s'affiche sur la sortie standard.

Les mots **if**, **then**, **else**, **elif** et **fi** sont des mots-clé. Par conséquent, pour indenter une structure **if** suivant le « style langage C », on pourra l'écrire de la manière suivante :

```
if suite_de_commandes1 ; then
suite_de_commandes2
else
```

```
suite_de_commandes ]  
fi
```

La structure de contrôle doit comporter autant de mots-clés **fi** que de **if** (une branche **elif** ne doit pas se terminer par un **fi**).

```
Ex : if ... if ...  
then ... then ...  
elif ... else if ...  
then ... then ...  
fi fi  
fi
```

Dans une succession de **if** imbriqués où le nombre de **else** est inférieur au nombre de **then**, le mot-clé **fi** précise l'association entre les **else** et les **if**.

```
Ex : if ...  
then ...  
if ...  
then ...  
fi  
else ...  
fi
```

Commande composée **[[**

La commande interne composée **[[** est souvent utilisée avec la commande interne composée **if**. Elle permet l'évaluation d'expressions conditionnelles portant sur des objets aussi différents que les permissions sur une entrée, la valeur d'une chaîne de caractères ou encore l'état d'une option de la commande interne **set**.

```
Syntaxe : [[ expr_cond ]]
```

Les deux caractères **crochets** doivent être collés et un caractère séparateur doit être présent de part et d'autre de **expr_cond**. Les mots **[[** et **]]** sont des mots-clé.

Le fonctionnement de cette commande interne est le suivant : l'expression conditionnelle **expr_cond** est évaluée et si sa valeur est *Vrai*, alors le code de retour de la commande interne **[[** est égal à **0**. Si sa valeur est *Faux*, le code de retour est égal à **1**. Si **expr_cond** est mal formée ou si les caractères **crochets** ne sont pas collés, une valeur différente est retournée.

La commande interne **[[** offre de nombreuses expressions conditionnelles ; c'est pourquoi, seules les principales formes de **expr_cond** seront présentées, regroupées par catégories.

. Permissions :

-r *entrée* vraie si *entrée* existe et est accessible en lecture par le processus courant.

-w *entrée* vraie si *entrée* existe et est accessible en écriture par le processus courant.

-x *entrée* vraie si le fichier *entrée* existe et est exécutable par le processus courant ou si le répertoire *entrée* existe et le processus courant possède la permission de passage.

```
Ex : $ echo coucou > toto
$ chmod 200 toto
$ ls -l toto
--w----- 1 sanchis sanchis 7 déc 17 17:21 toto
$
$ if [[ -r toto ]]
> then cat toto
> fi
$ => aucun affichage donc toto n'existe pas ou n'est pas accessible en lecture,
$ => dans le cas présent, il est non lisible
$
$ echo $?
0 => code de retour de la commande interne if
$
```

Mais,

```
$ [[ -r toto ]]
$
$ echo $?
1 => code de retour de la commande interne [[
$
```

. Types d'une entrée :

-f *entrée* vraie si *entrée* existe et est un fichier ordinaire

-d *entrée* vraie si *entrée* existe et est un répertoire

Exemple : programme **affic**

```
- - - - -
#!/bin/bash
if [[ -f $1 ]]
then
echo $1 : fichier ordinaire
cat $1
elif [[ -d $1 ]]
then
echo $1 : repertoire
ls $1
else
echo $1 : type non traite
fi
- - - - -
```

. Renseignements divers sur une entrée :

-a *entrée* vraie si *entrée* existe

-s *entrée* vraie si *entrée* existe et sa taille est différente de zéro

entrée1 **-nt** *entrée2* vraie si *entrée1* existe et sa date de modification est plus récente que celle de *entrée2*

entrée1 **-ot** *entrée2* vraie si *entrée1* existe et est plus ancienne que celle de *entrée2*

. Longueur d'une chaîne de caractères :

-z *ch* vraie si la longueur de la chaîne *ch* est égale à zéro

-n *ch* vraie si la longueur de la chaîne *ch* est différente de zéro

. Comparaisons de chaînes de caractères :

ch1 < *ch2* vraie si *ch1* précède *ch2*

ch1 > *ch2* vraie si *ch1* suit *ch2*

L'ordre des chaînes *ch1* et *ch2* est commandé par la valeur des paramètres régionaux.

ch == *mod* vraie si la chaîne *ch* correspond au modèle *mod*

ch != *mod* vraie si la chaîne *ch* ne correspond pas au modèle *mod*

mod est un modèle de chaînes pouvant contenir caractères et expressions génériques.

```
Ex : $ a="au revoir"
$
$ [[ $a == 123 ]] => faux
$
$ echo $?
1
$
$ [[ $a == a* ]] => vrai, la valeur de a commence par le caractère a
$
$ echo $?
0
$
```

Si par mégarde *ch* ou *mod* ne sont pas définies, la commande interne **[[** ne provoque pas d'erreur de syntaxe.

Exemple : programme **testval**

```
- - - - -
#!/bin/bash
if [[ $1 == $a ]]
then echo OUI
else echo >&2 NON
fi
- - - - -
```

```
Ex : $ testval coucou
NON => dans testval, $1 est remplacé par coucou, la variable a n'est pas
$ => définie
$ testval
OUI => aucun des deux membres de l'égalité n'est défini
$
```

. Etat d'une option :

-o *opt* vraie si l'état de l'option *opt* de la commande interne set est à on

```
Ex : $ set -o | grep noglob
noglob off
$
$ if [[ -o noglob ]]; then echo ON
> else echo OFF
> fi
OFF
$
```

. Composition d'expressions conditionnelles :

(*expr_cond*) vraie si *expr_cond* est vraie. Permet le regroupement d'expressions conditionnelles

! *expr_cond* vraie si *expr_cond* est fausse

expr_cond1 && *expr_cond2* vraie si les deux *expr_cond* sont vraies. L'expression *expr_cond2* n'est pas évaluée si *expr_cond1* est fausse.

expr_cond1 || *expr_cond2* vraie si une des deux *expr_cond* est vraie. L'expression *expr_cond2* n'est pas évaluée si *expr_cond1* est vraie.

Les quatre opérateurs ci-dessus ont été listés par ordre de priorité décroissante.

```
Ex : $ ls -l /etc/at.deny
-rw-r----- 1 root daemon 144 jan 3 2006 /etc/at.deny
$
$ if [[ ! ( -w /etc/at.deny || -r /etc/at.deny ) ]]
> then
> echo OUI
> else
> echo NON
> fi
OUI
$
```

Le fichier **/etc/at.deny** n'est accessible ni en lecture ni en écriture pour l'utilisateur *sanchis* ; le code de retour de la commande interne **[[** sera égal à zéro car l'expression conditionnelle est vraie.

Attention : on prendra soin de séparer les différents opérateurs et symboles par des **espaces**

```
Ex : $ if [[ !( -w /etc/at.deny || -r /etc/at.deny ) ]]
-bash: !: event : not found
$
```

Dans l'exemple ci-dessus, il n'y a aucun **blanc** entre **!** et **(**, ce qui provoque une erreur.

En combinant commande interne **[[**, opérateurs sur les codes de retour et regroupements de commandes, l'utilisation d'une structure **if** devient inutile.

```
Ex : $ [[ -r toto ]] || {
> echo >&2 "Probleme de lecture sur toto"
```

```
> }  
Probleme de lecture sur toto  
$
```

Remarque : par souci de portabilité, **bash** intègre également l'ancienne commande interne `[`. Celle-ci possède des fonctionnalités similaires à celles de `[[` mais est plus délicate à utiliser.

```
Ex : $ a="au revoir"  
$  
$ [ $a = coucou ] => l'opérateur égalité de [ est le symbole =  
-bash: [: too many arguments  
$
```

Le caractère **espace** présent dans la valeur de la variable `a` provoque une erreur de syntaxe. Il est nécessaire de prendre davantage de précaution quand on utilise cette commande interne.

```
Ex : $ [ "$a" = coucou ]  
$  
$ echo $?  
1  
$
```


XI - Entiers et expressions arithmétiques

XI-1 - Variables de type entier

Pour définir et initialiser une ou plusieurs variables de type entier, on utilise la syntaxe suivante :

```
declare -i nom[=expr_arith] [ nom[=expr_arith] ... ]
```

```
Ex : $ declare -i x=35 => définition et initialisation de la variable entière x
$
$ declare -i v w => définition des variables entières v et w
$
$ v=12 => initialisation de v par affectation
$
$ read w
34 => initialisation de w par lecture
$
```

Rappel : Il n'est pas nécessaire de définir une variable avant de l'utiliser !

Pour que la valeur d'une variable entière ne soit pas accidentellement modifiée après qu'elle ait été initialisée, il suffit d'ajouter l'attribut **r**.

```
Ex : $ declare -ir a=-6
$
$ a=7
-bash: a: readonly variable => seule la consultation est autorisée !
$
```

Enfin, pour connaître toutes les variables entières définies, il suffit d'utiliser la commande **declare #i**.

```
Ex : $ declare -i
declare -ir EUID="1007"
declare -ir PPID="19620"
declare -ir UID="1007"
declare -ir a="-6"
declare -i v="12"
declare -i w="14"
declare -i x="35"
$
```

XI-2 - Commande interne ((

Cette commande interne est utilisée pour effectuer des opérations arithmétiques.

```
Syntaxe : (( expr_arith ))
```

Son fonctionnement est le suivant : **expr_arith** est évaluée ; si cette évaluation donne une valeur différente de **0**, alors le code de retour de la commande interne **((** est égal à **0** sinon le code de retour est égal à **1**.

Il est donc important de distinguer deux aspects de la commande interne `((expr_arith))` :

- la valeur de `expr_arith` issue de son évaluation et
- le code de retour de `((expr_arith))`.

Attention : la valeur de `expr_arith` n'est pas affichée sur la sortie standard.

```
Ex : $ (( -5 )) => la valeur de l'expression arithmétique est égale à -5 (c.-à-d.  
$ => différente de 0), donc le code de retour de (( est égal à 0  
$ echo $?  
0  
$  
$ (( 0 )) => la valeur de l'expression arithmétique est 0, donc le code de retour  
$ => de (( est égal à 1  
$ echo $?  
1  
$
```

Les opérateurs permettant de construire des expressions arithmétiques évaluables par **bash** sont issus du langage C (`ex : + - > <=`).

```
Ex : $ declare -i a=2 b  
$ (( b = a + 7 ))  
$
```

Le format d'écriture est libre à l'intérieur de la commande `((`. En particulier, plusieurs caractères **espace** ou **tabulation** peuvent séparer les deux membres d'une affectation.

Il est inutile d'utiliser le caractère de substitution `$` devant le nom d'une variable car il n'y a pas d'ambiguïté dans l'interprétation ; par contre, lorsqu'une expression contient des paramètres de position, le caractère `$` doit être utilisé.

```
Ex : $ date  
jeudi 21 décembre 2006, 19:41:42 (UTC+0100)  
$  
$ set $(date)  
$  
$ (( b = $2 +1 )) => incrémentation du jour courant  
$  
$ echo $b  
22  
$
```

Code de retour de `((` et structures de contrôle

Le code de retour d'une commande interne `((` est souvent exploité dans une structure de contrôle **if** ou **while**.

Le programme shell **dix** ci-dessous affiche les dix chiffres :

```
- - - - -  
#!/bin/bash  
declare -i i=0  
while (( i < 10 ))  
do  
echo $i
```

```
(( i = i + 1 )) # ou (( i++ ))
done
```

Son fonctionnement est le suivant : l'expression $i < 10$ est évaluée, sa valeur est vraie, le code de retour de $((i < 10))$ est égal à 0, le corps de l'itération est exécuté. Après affichage de la valeur 9, la valeur devient égale à 10. L'expression $i < 10$ est évaluée, sa valeur est maintenant fausse, le code de retour de $((i < 10))$ est égal à 1, l'itération se termine.

Ex : \$ dix

0

1

2

3

4

5

6

7

8

9

\$

XI-3 - Valeur d'une expression arithmétique

La commande interne ((*expr_arith*)) n'affiche pas sur la sortie standard la valeur de l'expression arithmétique *expr_arith*.

Pour obtenir la valeur de l'expression arithmétique, on utilise la syntaxe : `$((expr_arith))`

```
Ex : echo $( 7 * 2 ) echo $( a= 12*8 ) echo $( 7 < 10 )
```

Substitutions de commandes et substitutions de paramètres peuvent être présentes dans une commande interne ((ou bien dans une substitution d'expression arithmétique \$(())\$ si le résultat aboutit à un nombre entier.

Par exemple, dans l'expression `$(($(ls -l | wc -l) - 1))` le résultat du pipeline `ls -l | wc -l` peut être interprété comme un nombre.

Attention : lors de l'évaluation d'une expression arithmétique, les débordements éventuels ne sont pas détectés.

```
Ex : $ echo $((897655*785409*56789*67899999999999999999 ))
-848393034087410691 => nombre négatif !
$
```

Si l'on souhaite utiliser de grands entiers (ou des nombres réels), il est préférable d'utiliser la commande unix **bc**.

[illegible]

L'option **#q** de **bc** évite l'affichage du message de bienvenue lorsqu'on utilise cette commande de manière interactive.

Basée sur le contexte, l'interprétation des expressions arithmétiques est particulièrement souple en **bash**.

```
Ex : $ declare -i x=35
$
$ z=x+5 => affectation de la chaîne x+5 à la variable z
$ echo $z
x+5 => non évaluée car z de type chaîne de caractères par défaut
$
$ echo ${z}) => par le contexte, z est évaluée comme une variable de type entier
40
$ (( z = z+1 ))
$ echo $z
41
$
```

XI-4 - Opérateurs

Une expression arithmétique contenue dans une commande interne **((** peut être une valeur, un paramètre ou bien être construite avec un ou plusieurs opérateurs. Ces derniers proviennent du langage C. La syntaxe des opérateurs, leur signification, leur ordre de priorité et les règles d'associativité s'inspirent également du langage C.

Les opérateurs traités par la commande **((** sont nombreux. C'est pourquoi, seuls les principaux opérateurs sont mentionnés dans le tableau ci-dessous, munis de leur associativité (**g** : de gauche à droite, **d** : de droite à gauche). Ne sont pas présentés dans ce tableau les opérateurs portant sur les représentations binaires (ex : **et bit à bit**, **ou bit à bit**, etc.).

Les opérateurs ayant même priorité sont regroupés dans une même classe et les différentes classes sont citées par ordre de priorité décroissante.

(1) **a++ a--** post-incrémentation, post-décrémentation (**g**)

(2) **++a --a** pré-incrémentation, pré-décrémentation (**d**)

(3) **-a +a** moins unaire, plus unaire (**d**)

(4) **! a** négation logique (**d**)

(5) **a**b** exponentiation : ab (**d**)

(6) **a*b a/b a%b** multiplication, division entière, reste (**g**)

(7) **a+b a-b** addition, soustraction (**g**)

(8) **a<b a<=b a>b a>=b** comparaisons (**g**)

(9) **a==b a!=b** égalité, différence (**g**)

(10) **a&&b** ET logique (**g**)

(11) **a||b** OU logique (**g**)

(12) **a?b:c** opérateur conditionnel (**d**)

(13) `a=b a*=b a%=b` opérateurs d'affectation (d)

`a+=b a-=b`

(14) `a,b` opérateur virgule (g)

Ces opérateurs se décomposent en :

- opérateurs arithmétiques
- opérateurs d'affectations
- opérateurs relationnels
- opérateurs logiques
- opérateurs divers.

Opérateurs arithmétiques :

Les quatre opérateurs ci-dessous s'appliquent à une *variable*.

Post-incrémentation : `var++` la valeur de `var` est d'abord utilisée, puis est incrémentée

```
Ex : $ declare -i x=9 y
$ (( y = x++ )) => y reçoit la valeur 9 ; x vaut 10
$ echo "y=$y x=$x"
y=9 x=10
$
```

Post-décrémentation : `var--` la valeur de `var` est d'abord utilisée, puis est décrémentée

```
Ex : $ declare -i x=9 y
$ (( y = x-- )) => y reçoit la valeur 9 ; x vaut 8
$ echo "y=$y x=$x"
y=9 x=8
$
```

Pré-incrémentation : `++var` la valeur de `var` est d'abord incrémentée, puis est utilisée

```
Ex : $ declare -i x=9 y
$ (( y=++x )) => x vaut 10 ; y reçoit la valeur 10
$ echo "y=$y x=$x"
y=10 x=10
$
```

Pré-décrémentation : `--var` la valeur de `var` est d'abord décrémentée, puis est utilisée

```
Ex : $ declare -i x=9 y
$ (( y=--x )) => x vaut 8 ; y reçoit la valeur 8
$ echo "y=$y x=$x"
y=8 x=8
$
```

Les autres opérateurs s'appliquent à des *expressions arithmétiques*.

Moins unaire : **- expr_arith**

Addition : **expr_arith + expr_arith**

Soustraction : **expr_arith - expr_arith**

Multiplication : **expr_arith * expr_arith**

Division entière : **expr_arith / expr_arith**

Reste de division entière : **expr_arith % expr_arith**

```
Ex : $ (( a = b/3 +c )) => division entière et addition
$
$ echo $(( RANDOM%49 +1 )) => reste et addition
23
$
```

La variable prédéfinie du shell **RANDOM** renvoie une valeur pseudo-aléatoire dès qu'elle est utilisée.

L'utilisation de **parenthèses** permet de modifier l'ordre d'évaluation des composantes d'une expression arithmétique.

```
Ex : $ (( a = ( b+c ) *2 ))
$
```

Opérateurs d'affectations :

Affectation simple : **nom = expr_arith**

Affectation composée : **nom opérateur= expr_arith**

Comme en langage C, l'affectation n'est pas une instruction (on dirait *commande* dans la terminologie du shell) mais une expression, et comme toute expression elle possède une valeur. Celle-ci est la valeur de son membre gauche après évaluation de son membre droit.

```
Ex : $ ((a=b=c=5)) => la valeur de l'expression a=b=c=5 est égale à 5
$ echo $a $b $c
5 5 5
$
```

Dans l'exemple ci-dessous, l'expression **a = b + 5** est évaluée de la manière suivante : **b** est évaluée (sa valeur est 2), puis c'est l'expression **b+5** qui est évaluée (sa valeur vaut 7), enfin cette valeur est affectée à la variable **a**. La valeur de l'expression **a = b + 5** est égale à celle de **a**, c'est à dire 7. Ceci permet d'écrire :

```
Ex : $ declare #i b=2
$
$ echo $(( a = b+5 ))
7
$
```

```
$ echo $a
7
$
```

La syntaxe nom *opérateur* = *expr_arith* est un raccourci d'écriture provenant du langage C et signifiant :

```
nom = nom opérateur expr_arith
```

opérateur pourra être : * / + -

```
Ex : $ (( a += 1 )) => ceci est équivalent à (( a = a + 1 ))
$
```

Opérateurs relationnels :

Lorsqu'une expression relationnelle (ex : *a < 3*) est *vraie*, la valeur de l'expression est égale à **1**.

Lorsqu'elle est *fausse*, sa valeur est égale à **0**.

Egalité : *expr_arith == expr_arith* (Attention aux deux caractères **égal**)

Différence : *expr_arith != expr_arith*

Inférieur ou égal : *expr_arith <= expr_arith*

Supérieur ou égal : *expr_arith >= expr_arith*

Strictement inférieur : *expr_arith < expr_arith*

Strictement supérieur : *expr_arith > expr_arith*

```
Ex : $ declare #i a=4
$
$ (( a < 3 )) => expression fausse, valeur égale à 0, code de retour égal à 1
$ echo $?
1
$
```

```
Ex : $ declare #i a=3 b=2
$
$ if (( a == 7 ))
> then (( a = 2*b ))
> else (( a = 7*b ))
> fi
$
$ echo $a
14
$
```

L'expression relationnelle *a == 7* est *fausse*, sa valeur est 0 et le code de retour de *((a == 7))* est égal à 1: c'est donc la partie else qui est exécutée.

Opérateurs logiques :

Comme pour une expression relationnelle, lorsqu'une expression logique (ex : `a > 3 && a < 5`) est *vraie*, la valeur de l'expression est égale à 1.

Lorsqu'elle est *fausse*, sa valeur est égale à 0.

Négation logique : `! expr_arith`

Si la valeur de `expr_arith` est différente de 0, la valeur de `! expr_arith` est égale à 0.

Si la valeur de `expr_arith` est égale à 0, la valeur de `! expr_arith` est égale à 1.

Au moins un caractère **espace** doit être présent entre `!` et `expr_arith`.

```
Ex : $ echo $(( ! 12 )) => echo $(( !12 )) provoque une erreur
0
$
```

Et logique : `expr_arith1 && expr_arith2`

Si la valeur de `expr_arith1` est égale à 0 (*fausse*), alors `expr_arith2` n'est pas évaluée et la valeur de `expr_arith1 && expr_arith2` est 0.

Si la valeur de `expr_arith1` est différente de 0 (*vraie*), alors `expr_arith2` est évaluée : si sa valeur est égale à 0, alors la valeur de `expr_arith1 && expr_arith2` est 0, sinon elle vaut 1.

```
Ex: $ declare #i a=4
$
$ echo $(( a<3 )) => expression fausse
0
$
$ echo $(( a > 3 && a < 5 )) => expression vraie, valeur égale à 1
1
$ ((a>3 && a<5 )) => expression vraie, code de retour égal à 0
$ echo $?
0
$
```

Ou logique : `expr_arith1 || expr_arith2`

Si la valeur de `expr_arith1` est différente de 0 (*vraie*), alors `expr_arith2` n'est pas évaluée et la valeur de `expr_arith1 || expr_arith2` est égale à 1.

Si la valeur de `expr_arith1` est égale à 0 (*fausse*), alors `expr_arith2` est évaluée : si sa valeur est différente de 0, alors la valeur de `expr_arith1 || expr_arith2` est 1, sinon elle vaut 0.

```
Ex : $ declare #i x=4
$
$ (( x > 4 || x < 4 ))
$ echo $?
1
$
```



```
$ echo $(( x > 4 || x<4 ))  
0  
$
```

Exponentiation : **expr_arith1** ** **expr_arith2**

Contrairement aux autres opérateurs, l'opérateur d'exponentiation ****** n'est pas issu du langage C. A partir de la version 3.1 de **bash**, son associativité a changé : elle est de droite à gauche (**d**).

```
Ex : $ echo $(( 2**3**2 )) => interprétée comme 2**(3**2) : 29  
512  
$
```

Opérateur conditionnel : **expr_arith1** ? **expr_arith2** : **expr_arith3**

Le fonctionnement de cet opérateur ternaire est le suivant : **expr_arith1** est évaluée, si sa valeur est *vraie* la valeur de l'expression arithmétique est celle de **expr_arith2**, sinon c'est celle de **expr_arith3**.

```
Ex : $ declare -i a=4 b=0  
$ echo $(( a < 3 ? (b=5) : (b=54) ))  
54  
$  
$ echo $b  
54  
$
```

Il aurait également été possible d'écrire : **((b = a < 3 ? 5 : 54))**

Opérateur virgule : **expr_arith1** , **expr_arith2**

expr_arith1 est évaluée, puis **expr_arith2** est évaluée ; la valeur de l'expression est celle de **expr_arith2**.

```
Ex : $ declare -i a b  
$ echo $(( a=4,b=9 ))  
9  
$ echo "a=$a b=$b"  
a=4 b=9  
$
```

XI-5 - Structure for pour les expressions arithmétiques

Bash a introduit une nouvelle structure **for** adaptée aux traitements des expressions arithmétiques, itération issue du langage C. Elle fonctionne comme cette dernière.

```
Syntaxe : for (( expr_arith1 ; expr_arith2 ; expr_arith3 ))  
do  
suite_cmd  
done
```

expr_arith1 est l'expression arithmétique d'initialisation.

expr_arith2 est la condition d'arrêt de l'itération.

expr_arith3 est l'expression arithmétique qui fixe le pas d'incrémentation ou de décrémentation.

Exemples :

```
declare -i x
for (( x=0 ; x<5 ; x++ ))
do
echo $(( x*2 ))
done
declare -i x y
for (( x=1,y=10 ; x<4 ; x++,y-- ))
do
echo $(( x*y ))
done
```

Exercice : En utilisant **RANDOM**, écrire un programme **tirage_flash** qui affiche six entiers différents compris entre 1 et 49.

Exercice : Ecrire un programme **factiter** qui prend un entier *N* positif ou nul en argument et affiche sa factorielle *N!*

Attention aux débordements : *N* ne doit pas être trop grand.

```
Ex : $ factiter 7
5040
$
$ factiter 0
1
$
```

Exercice : Ecrire une version récursive **factrecur** du programme précédent.

XI-6 - Exemple : les tours de Hanoi

Le problème des « tours de Hanoi » peut s'énoncer de la manière suivante :

- conditions de départ : plusieurs disques sont placés sur une table A, les uns sur les autres, rangés par taille décroissante, le plus petit étant au dessus de la pile
- résultat attendu : déplacer cette pile de disques de la table A vers une table B
- règles de fonctionnement :

on dispose d'une troisième table C,

. on ne peut déplacer qu'un disque à la fois, celui qui est placé en haut d'une pile et le déposer uniquement sur un disque plus grand que lui ou bien sur une table vide.

Le programme shell récursif *hanoi* traite ce problème ; il utilise quatre arguments : le nombre de disques et le nom de trois tables.

```
hanoi
- - - - -
#!/bin/bash
if (( $1 > 0 ))
```

```
then
hanoi $(( $1 - 1)) $2 $4 $3
echo Deplacer le disque de la table $2 a la table $3
hanoi $(( $1 - 1)) $4 $3 $2
fi
- - - - -
```

Pour exécuter ce programme, on indique le nombre total *N* de disques présents sur la première table, le nom de la table où sont placés ces *N* disques et le nom des deux autres tables.

Dans l'exemple mentionné, la table *A* contient au départ *trois* disques et les deux autres tables *B* et *C* sont vides. Le programme affiche chaque déplacement effectué.

```
Ex : $ hanoi 3 A B C
Deplacer le disque de la table A a la table B
Deplacer le disque de la table A a la table C
Deplacer le disque de la table B a la table C
Deplacer le disque de la table A a la table B
Deplacer le disque de la table C a la table A
Deplacer le disque de la table C a la table B
Deplacer le disque de la table A a la table B
$
```

XII - Tableaux

Moins utilisés que les *chaînes de caractères* ou les *entiers*, **bash** intègre également les tableaux monodimensionnels.

XII-1 - Définition et initialisation d'un tableau

Pour créer un tableau, on utilise généralement l'option **-a** (comme *array*) de la commande interne **declare** :

```
declare -a nomtab ...
```

Le tableau *nomtab* est simplement créé mais ne contient aucune valeur : le tableau est défini mais n'est pas initialisé.

Pour connaître les tableaux définis : **declare -a**

```
Ex : $ declare -a
declare -a BASH_ARGC='()'
declare -a BASH_ARGV='()'
declare -a BASH_LINENO='()'
declare -a BASH_SOURCE='()'
declare -ar BASH_VERSINFO='([0]="3" [1]="1" [2]="17" [3]="1" [4]="release"
[5]="i486-pc-linux-gnu")'
declare -a DIRSTACK='()'
declare -a FUNCNAME='()'
declare -a GROUPS='()'
declare -a PIPESTATUS='([0]="0")'
$
```

Pour définir et initialiser un tableau :

```
declare -a nomtab=( val0 val1 ... )
```

Comme en langage C, l'indice d'un tableau débute toujours à **0** et sa valeur maximale est celle du plus grand entier positif représentable dans ce langage (**bash** a été écrit en C). L'indice peut être une expression arithmétique.

Pour désigner un élément d'un tableau, on utilise la syntaxe :

```
nomtab[indice]
```

```
Ex : $ declare -a tab => définition du tableau tab
$
$ read tab[1] tab[3]
coucou bonjour
$
$ tab[0]=hello
$
```

Il n'est pas obligatoire d'utiliser la commande interne **declare** pour créer un tableau, il suffit d'initialiser un de ses éléments :

```
Ex : $ array[3]=bonsoir => création du tableau array avec  
$ => initialisation de l'élément d'indice 3
```

Trois autres syntaxes sont également utilisables pour initialiser globalement un tableau :

- **nomtab**=(val0 val1 ...)

- **nomtab**=([indice]=val ...)

```
Ex : $ arr=([1]=coucou [3]=hello)  
$
```

- l'option **-a** de la commande interne **read** ou **readonly** :

```
Ex : $ read -a tabmess  
bonjour tout le monde  
$
```

XII-2 - Valeur d'un élément d'un tableau

On obtient la valeur d'un élément d'un tableau à l'aide la syntaxe :

```
${nomtab[indice]}
```

bash calcule d'abord la valeur de l'indice puis l'élément du tableau est remplacé par sa valeur.

Il est possible d'utiliser toute expression arithmétique valide de la commande interne **((** pour calculer l'indice d'un élément.

```
Ex : $ echo ${tabmess[1]}  
tout  
$  
$ echo ${tabmess[RANDOM%4]} # ou bien ${tabmess[$((RANDOM%4))]}  
monde  
$  
$ echo ${tabmess[1**2+1]}  
le  
$
```

Pour obtenir la longueur d'un élément d'un tableau :

```
${#nomtab[indice]}
```

```
Ex : $ echo ${tabmess[0]}  
bonjour  
$  
$ echo ${#tabmess[0]}  
7 => longueur de la chaîne bonjour  
$
```

Lorsqu'un tableau sans indice est présent dans une chaîne de caractères ou une expression, **bash** utilise l'élément d'indice **0**.

```
Ex : $ echo $tabmess
bonjour
$
```

Réciproquement, une variable non préalablement définie comme tableau peut être interprétée comme un tableau.

```
Ex : $ var=bonjour
$ echo ${var[0]} => var est interprétée comme un tableau à un seul élément
bonjour => d'indice 0
$ var=( coucou ${var[0]} )
$ echo ${var[1]} => var est devenu un véritable tableau
bonjour
$
```

Exercice : Ecrire un programme shell **carte** qui affiche le nom d'une carte tirée au hasard d'un jeu de 32 cartes. On utilisera deux tableaux : un tableau *couleur* et un tableau *valeur*.

```
Ex : $ carte
huit de carreau
$ carte
as de pique
$
```

XII-3 - Caractéristiques d'un tableau

Le nombre d'éléments d'un tableau est désigné par :

Seuls les éléments initialisés sont comptés.

```
${#nomtab[*]}
```

```
Ex : $ echo ${#arr[*]}
2
$
```

Tous les éléments d'un tableau sont accessibles à l'aide de la notation :

```
${nomtab[*]}
```

Seuls les éléments initialisés sont affichés

```
Ex : $ echo ${arr[*]}
coucou hello
$
```

Pour obtenir la liste de tous les indices conduisant à un élément défini d'un tableau, on utilise la syntaxe :

```
${ !nomtab[*]}
```

```
Ex : $ arr=( [1]=coucou bonjour [5]=hello )
$
$ echo ${!arr[*]}
1 2 5
$
```

L'intérêt d'utiliser cette syntaxe est qu'elle permet de ne traiter que les éléments définis d'un tableau « à trous ».

```
Ex : $ for i in ${!arr[*]}
> do
> echo "$i => ${arr[i]}" => dans l'expression ${arr[i]}, bash interprète
> done => directement i comme un entier
1 => coucou
2 => bonjour
5 => hello
$
```

Pour ajouter un élément *val* à un tableau *tab* :

- à la fin : `tab[${#tab[*]}]=val`

- en tête : `tab=(val ${tab[*]})`

```
Ex : $ tab=( un deux trois )
$ echo ${#tab[*]}
3
$ tab[${#tab[*]}]=fin => ajout en fin de tableau
$ echo ${tab[*]}
un deux trois fin
$ tab=( debut ${tab[*]} ) => ajout en tête de tableau
$ echo ${tab[*]}
debut un deux trois fin
$
```

Exercice : Ecrire un programme shell **distrib** qui crée un paquet de 5 cartes différentes tirées au hasard parmi un jeu de 32 cartes et affiche le contenu du paquet.

Exercice : Ecrire un programme shell **tabnoms** qui place dans un tableau les noms de tous les utilisateurs enregistrés dans le système, affiche le nombre total d'utilisateurs enregistrés, puis tire au hasard un nom d'utilisateur.

Si l'on souhaite créer un tableau d'entiers on utilise la commande **declare #ai**. L'ordre des options n'a aucune importance.

```
Ex : $ declare #ai tabent=( 2 45 -2 )
$
```

Pour créer un tableau en lecture seule, on utilise les options **#ra** :

```
Ex : $ declare -ra tabconst=( bonjour coucou salut ) => tableau en lecture seule
$
```

```
$ tabconst[1]=ciao
-bash: tabconst: readonly variable
$
$ declare -air tabInt=( 34 56 ) => tableau (d'entiers) en lecture seule
$ echo $(( tabInt[1] +10 ))
66
$ (( tabInt[1] += 10 ))
-bash: tabInt: readonly variable
$
```

XII-4 - Suppression d'un tableau

Pour supprimer un tableau ou élément d'un tableau, on utilise la commande interne **unset**.

Suppression d'un tableau : **unset** *nomtab* ...

Suppression d'un élément d'un tableau : **unset** *nomtab*[*indice*] ...

XIII - Alias

Un alias permet d'abrégier une longue commande, de remplacer le nom d'une commande existante par un autre nom ou bien de modifier le comportement d'une commande existante.

XIII-1 - Création d'un alias

Pour créer un ou plusieurs alias, on utilise la syntaxe :

```
alias nom=valeur ...
```

```
Ex : $ alias cx='chmod u+x'  
$
```

Le nom de l'alias peut être présent dans sa propre définition. La commande *alias rm='rm -i'* redéfinit le comportement par défaut de la commande unix **rm** en demandant à l'utilisateur de confirmer la suppression (on force l'utilisateur à utiliser l'option **-i**).

```
Ex : $ alias rm='rm -i'  
$  
$ > err => création du fichier err  
$  
$ rm err  
rm: détruire fichier régulier vide `err'? o => l'alias rm demande  
$ => confirmation  
$ ls err  
ls: err: Aucun fichier ou répertoire de ce type => le fichier err a été  
$ => supprimé
```

Attention :

on ne doit pas définir un alias et utiliser cet alias dans la même ligne mais sur deux lignes différentes.

```
Ex : $ alias aff='echo bonjour' ; aff tout le monde  
-bash: aff: command not found => aff tout le monde n'a pu s'exécuter !  
$  
$ aff la compagnie  
bonjour la compagnie => l'alias est connu  
$
```

Si l'on désire utiliser plusieurs alias dans la même commande, il est nécessaire de laisser un caractère **espace** ou **tabulation** comme dernier caractère de *valeur*.

```
Ex : $ cat /etc/debian_version  
4.0  
$  
$ alias c=cat d=/etc/debian_version  
$  
$ c d  
cat: d: Aucun fichier ou répertoire de ce type  
$
```

Dans l'exemple ci-dessus, l'alias `d` n'a pas été interprété car le dernier caractère de la valeur de `c` n'est ni un **espace**, ni une **tabulation**. En ajoutant un caractère **espace**, on obtient le résultat voulu.

```
Ex : $ alias c='cat '  
$  
$ c d  
4.0  
$
```

La valeur d'un alias peut inclure plusieurs commandes.

```
Ex : $ pwd  
/tmp => répertoire courant  
$ alias scd='echo salut ; cd '  
$ alias rep=/home/sanchis/bin  
$  
$ scd rep  
salut  
$ pwd  
/home/sanchis/bin => nouveau répertoire courant  
$
```

La protection de la valeur d'un alias doit être choisie judicieusement. Définissons un alias affichant l'heure courante. La suite de commandes à exécuter est la suivante :

```
set $(date) ; echo ${5%:*}
```

Cette suite de commandes peut être entourée avec des caractères **quote** ou bien avec des caractères **guillemet**.

```
Ex : $ date  
vendredi 22 décembre 2006, 18:45:38 (UTC+0100)  
$  
$ alias h='set $(date) ; echo ${5%:*}'  
$  
$ h  
18:46  
$  
$ sleep 60  
... => attente de 60 secondes  
$ h  
18:47 => mise à jour de l'heure  
$
```

Attention :

en entourant la suite de commandes avec des caractères **guillemet**, le shell exécute les substitutions avant d'affecter la valeur à l'alias, puis interprète cet alias.

```
Ex : $ alias g="set $(date) ; echo ${5%:*}"  
$  
$ g  
-bash: syntax error near unexpected token `('
```

C'est la présence de la parenthèse ouvrante de la chaîne de caractères

```
set vendredi 22 décembre 2006, 19:08:21 (UTC+0100)
```

Cette chaîne est exécutée comme une commande par le shell, ce qui provoque l'erreur.

Liste des alias définis :

Pour connaître l'ensemble des alias définis, on utilise la commande **alias** sans argument :

```
Ex : $ alias
alias aff='echo bonjour'
alias c='cat '
alias cx='chmod u+x'
alias d='/etc/debian_version'
alias g='set vendredi 22 décembre 2006, 19:08:21 (UTC+0100) ; echo '
alias h='set $(date) ; echo ${5%:*}'
alias ll='ls -l'
alias ls='ls --color=auto'
alias rep='/home/sanchis/bin'
alias rm='rm -i'
alias scd='echo salut ; cd '
$
```

Pour connaître la valeur d'un ou plusieurs alias : **alias nom #**

```
Ex : $ alias ll
alias ll='ls -l'
$
```

XIII-2 - Suppression d'un alias

Pour rendre indéfinie la valeur d'un ou plusieurs alias, on utilise la commande **unalias**.

La syntaxe est :

```
unalias nom ...
```

```
Ex : $ unalias g rep aff scd
$
$ alias
alias c='cat '
alias cx='chmod u+x'
alias d='/etc/debian_version'
alias h='set $(date) ; echo ${5%:*}'
alias ll='ls -l'
alias ls='ls --color=auto'
alias rm='rm -i'
$
```

XIV - Fonctions shell

XIV-1 - Définition d'une fonction

Le shell **bash** propose plusieurs syntaxes pour définir une fonction. Nous utiliserons celle-ci :

```
function nom_fct
{
    suite_de_commandes
}
```

nom_fct spécifie le nom de la fonction. Le corps de celle-ci est *suite_de_commandes*.

Pour appeler une fonction, il suffit de mentionner son nom.

Comme pour les autres commandes composées de **bash**, une fonction peut être définie directement à partir d'un shell interactif.

```
Ex : $ function f0
> {
> echo Bonjour tout le monde !
> }
$
$ f0 => appel de la fonction f0
Bonjour tout le monde !
$
```

Les mots réservés **function** et **}** doivent être les premiers mots d'une commande pour qu'ils soient reconnus. Sinon, il suffit de placer un caractère **point-virgule** avant le mot-clé :

```
function nom
{ suite_de_commandes ;}
```

La définition d'une fonction « à la C » est également possible :

```
function nom {
    suite_de_commandes
}
```

L'exécution d'une fonction s'effectue dans l'environnement courant, autorisant ainsi le partage de variables.

```
Ex : $ c=Coucou
$
$ function f1
> {
> echo $c => utilisation dans la fonction d'une variable externe c
> }
$
$ f1
Coucou
$
```

Les noms de toutes les fonctions définies peuvent être listés à l'aide de la commande : **declare -F**

```
Ex : $ declare -F
declare -f f0
declare -f f1
$
```

Les noms et corps de toutes les fonctions définies sont affichés à l'aide de la commande : **declare -f**

```
Ex : $ declare -f
f0 ()
{
echo Bonjour tout le monde !
}
f1 ()
{
echo $c
}
$
```

Pour afficher le nom et corps d'une ou plusieurs fonctions : **declare -f nomfct ...**

```
Ex : $ declare -f f0
f0 ()
{
echo Bonjour tout le monde !
}
$
```

Une définition de fonction peut se trouver en tout point d'un programme shell ; il n'est pas obligatoire de définir toutes les fonctions en début de programme. Il est uniquement nécessaire que la définition d'une fonction soit faite avant son appel effectif, c'est-à-dire *avant* son exécution :

```
function f1
{ ... ;}
suite_commandes1
function f2
{ ... ;}
suite_commandes2
```

Dans le code ci-dessus, *suite_commandes1* ne peut exécuter la fonction *f2* (contrairement à *suite_commandes2*). Cela est illustré par le programme shell *appelAvantDef* :

appelAvantDef

```
- - - - -
# !/bin/bash
echo Appel Avant definition de fct
fct # fct non definie
function fct
{
echo Execution de : fct
sleep 2
echo Fin Execution de : fct
}
```

```
echo Appel Apres definition de fct
fct # fct definie
- - - - -
```

Son exécution se déroule de la manière suivante :

```
Ex : $ appelAvantDef
Appel Avant definition de fct
./appelAvantDef: line 4: fct: command not found
Appel Apres definition de fct
Execution de : fct
Fin Execution de : fct
$
```

Lors du premier appel à la fonction *fct*, celle-ci n'est pas définie : une erreur d'exécution se produit. Puis, le shell lit la définition de la fonction *fct* : le deuxième appel s'effectue correctement.

Contrairement au programme précédent, dans le programme shell *pingpong*, les deux fonctions *ping* et *pong* sont définies avant leur appel effectif :

pingpong

```
- - - - -
#!/bin/bash
function ping
{
echo ping
if (( i > 0 ))
then
((i--))
pong
fi
}
function pong
{
echo pong
if (( i > 0 ))
then
((i--))
ping
fi
}
declare -i i=4
ping # (1) ping et pong sont definies
- - - - -
```

Au point **(1)**, les corps des fonctions *ping* et *pong* ont été lus par l'interpréteur de commandes **bash** : *ping* et *pong* sont définies.

```
Ex : $ pingpong
ping
pong
ping
pong
ping
$
```

XIV-2 - Suppression d'une fonction

Une fonction est rendue indéfinie par la commande interne : `unset -f nomfct ...`

```
Ex : $ declare -F
declare -f f0
declare -f f1
$
$ unset -f f1
$
$ declare -F
declare -f f0 => la fonction f1 n'existe plus !
$
```

XIV-3 - Trace des appels aux fonctions

Le tableau prédéfini `FUNCNAME` contient le nom des fonctions en cours d'exécution, matérialisant la pile des appels.

Le programme shell `traceAppels` affiche le contenu de ce tableau au début de son exécution, c'est-à-dire hors de toute fonction : le contenu du tableau `FUNCNAME` est vide (a). Puis la fonction `f1` est appelée, `FUNCNAME` contient les noms `f1` et `main` (b). La fonction `f2` est appelée par `f1` : les valeurs du tableau sont `f2`, `f1` et `main` (c).

Lorsque l'on est à l'intérieur d'une fonction, la syntaxe `$FUNCNAME` (ou `${FUNCNAME[0]}`) renvoie le nom de la fonction courante.

traceAppels

```
- - - - -
#!/bin/bash
function f2
{
echo " ----- Dans f2 : "
echo " ----- FUNCNAME : $FUNCNAME"
echo " ----- tableau FUNCNAME[] : ${FUNCNAME[*]}"
}function f1
{
echo " --- Dans f1 : "
echo " --- FUNCNAME : $FUNCNAME"
echo " --- tableau FUNCNAME[] : ${FUNCNAME[*]}"
echo " --- - Appel a f2 "
echo
f2
}
echo "Debut : "
echo "FUNCNAME : $FUNCNAME"
echo "tableau FUNCNAME[] : ${FUNCNAME[*]}"
echo
f1
- - - - -
```

```
Ex : $ traceAppels
Debut :
FUNCNAME :
tableau FUNCNAME[] : (a)
--- Dans f1 :
--- FUNCNAME : f1
```

```
--- tableau FUNCNAME[] : f1 main (b)
--- - Appel a f2
----- Dans f2 :
----- FUNCNAME : f2
----- tableau FUNCNAME[] : f2 f1 main (c)
$
```

XIV-4 - Arguments d'une fonction

Les arguments d'une fonction sont référencés dans son corps de la même manière que les arguments d'un programme shell le sont : **\$1** référence le premier argument, **\$2** le deuxième, etc., **\$#** le nombre d'arguments passés lors de l'appel de la fonction.

Le paramètre spécial **\$0** n'est pas modifié : il contient le nom du programme shell.

Pour éviter toute confusion avec les paramètres de position qui seraient éventuellement initialisés dans le code appelant la fonction, la valeur de ces derniers est sauvegardée avant l'appel à la fonction puis restituée après exécution de la fonction.

Le programme shell *args* illustre ce mécanisme de sauvegarde/restitution :

args

```
- - - - -
#!/bin/bash
function f
{
echo " --- Dans f : \"$0 : $0"
echo " --- Dans f : \"$# : $#\"
echo " --- Dans f : \"$1 : $1\" => affichage du 1er argument de la fonction f
}
echo "Avant f : \"$0 : $0"
echo "Avant f : \"$# : $#\"
echo "Avant f : \"$1 : $1\" => affichage du 1er argument du programme args
f pierre paul jacques
echo "Après f : \"$1 : $1\" => affichage du 1er argument du programme args
- - - - -
```

```
Ex : $ args un deux trois quatre
Avant f : $0 : ./args
Avant f : $# : 4
Avant f : $1 : un
--- Dans f : $0 : ./args
--- Dans f : $# : 3
--- Dans f : $1 : pierre
Après f : $1 : un
$
```

Utilisée dans le corps d'une fonction, la commande interne **shift** décale la numérotation des paramètres de position internes à la fonction.

argsShift

```
argsShift
- - - - -
```



```
#!/bin/bash
function f
{
echo " --- Dans f : Avant 'shift 2' : \${*} : \${*}"
shift 2
echo " --- Dans f : Apres 'shift 2' : \${*} : \${*}"
}
echo Appel : f un deux trois quatre
f un deux trois quatre
- - - - -
```

```
Ex : $ argsShift
Appel : f un deux trois quatre
--- Dans f : Avant 'shift 2' : \${*} : un deux trois quatre
--- Dans f : Apres 'shift 2' : \${*} : trois quatre
$
```

Qu'elle soit utilisée dans une fonction ou à l'extérieur de celle-ci, la commande interne `set` modifie toujours la valeur des paramètres de position.

argsSet

```
- - - - -
#!/bin/bash
function f
{
echo " -- Dans f : Avant execution de set \$(date) : \${*} : \${*}"
set \$(date)
cho " -- Dans f : Apres execution de set \$(date) : \${*} : \${*}"
}
echo Appel : f alpha beta
f alpha beta
- - - - -
```

```
Ex : $ argsSet
Appel : f alpha beta
-- Dans f : Avant execution de set \$(date) : \${*} : alpha beta
-- Dans f : Apres execution de set \$(date) : \${*} : mar mar 7 18:41:39 CET 2006
$
```

XIV-5 - Variables locales à une fonction

Par défaut, une variable définie à l'intérieur d'une fonction est globale ; cela signifie qu'elle est directement modifiable par les autres fonctions du programme shell.

Dans le programme shell *glob*, la fonction *fUn* est appelée en premier. Celle-ci crée et initialise la variable *var*. Puis la fonction *fDeux* est appelée et modifie la variable (globale) *var*. Enfin, cette variable est à nouveau modifiée puis sa valeur est affichée.

glob

```
- - - - -
#!/bin/bash
function fUn
{

```

```
var=Un # creation de la variable var
}
function fDeux
var=${var}Deux # premiere modification de var
}
fUn
fDeux
var=${var}Princ # deuxieme modification de var
echo $var
- - - - -
```

```
Ex : $ glob
UnDeuxPrinc => trace des modifications successives de la variable globale var
$
```

Pour définir une variable locale à une fonction, on utilise la commande interne **local**. Sa syntaxe est :

```
local [option] [nom[=valeur] ...]
```

Les options utilisables avec **local** sont celles de la commande interne **declare**. Par conséquent, on définira une ou plusieurs variables de type entier avec la syntaxe **local #i** (**local #a** pour un tableau local).

Le programme shell *loc* définit une variable entière *a* locale à la fonction *f1*. Cette variable n'est pas accessible à l'extérieur de cette fonction.

```
loc
- - - - -
#!/bin/bash
function f1
{
local -i a=12 => a est une variable locale à f1
(( a++ ))
echo "-- Dans f1 : a => $a"
}
f1
echo "Dans main : a => $a" => tentative d'accès à la valeur de a
- - - - -
```

```
Ex : $ loc
-- Dans f1 : a => 13
Dans main : a => => a n'est pas visible dans le corps du programme
$
```

La portée d'une variable locale inclut la fonction qui l'a définie ainsi que les fonctions qu'elle appelle (directement ou indirectement).

Dans le programme shell *appelsCascade*, la variable locale *x* est vue :

- dans la fonction *f1* qui définit cette variable
- dans la fonction *f2* qui est appelée par la fonction *f1*
- dans la fonction *f3* qui est appelée par la fonction *f2*.

```
appelsCascade
```

```
- - - - -
#!/bin/bash
function f3
{
  (( x = -x )) => modification de x définie dans la fonction f1
  echo "f3 : x=$x"
}
function f2
{
  echo "f2 : $((x+10))" => utilisation de x définie dans la fonction f1
  f3 => appel de f3
}
function f1
{
  local -i x
  x=2 => initialisation de x
  f2 => appel de f2
  echo "f1 : x=$x"
}
f1 => appel de f1
- - - - -
```

```
Ex : $ appelsCascade
2 : 12
f3 : x=-2
f1 : x=-2
$
```

XIV-6 - Exporter une fonction

Pour qu'une fonction puisse être exécutée par un programme shell différent de celui où elle a été définie, il est nécessaire d'exporter cette fonction. On utilise la commande interne **export**.

Syntaxe : `export -f nomfct ...`

Pour que l'export fonctionne, le sous-shell qui exécute la fonction doit avoir une relation de descendance avec le programme shell qui exporte la fonction.

Le programme shell *progBonj* définit et utilise une fonction *bonj*. Ce script lance l'exécution d'un programme shell *autreProgShell* qui utilise également la fonction *bonj* (mais qui ne la définit pas) ; *autreProgShell* étant exécuté dans un environnement différent de *progBonj*, il ne pourra trouver la définition de la fonction *bonj* : une erreur d'exécution se produit.

```
progBonj
- - - - -
#!/bin/bash
function bonj
{
  echo bonj : Bonjour $1
}
bonj Madame
autreProgShell
- - - - -
```

```
autreProgShell
- - - - -
```

```
#!/bin/bash
echo appel a la fonction externe : bonj
bonj Monsieur
- - - - -
```

```
Ex: $ progBonj
bonj : Bonjour Madame
appel a la fonction externe : bonj
./autreProgShell: line 4: bonj: command not found
$
```

Pour que la fonction *bonj* puisse être exécutée par , il suffit que le programme shell qui la contient exporte sa définition.

```
progBonjExport
- - - - -
#!/bin/bash
function bonj
{
echo bonj : Bonjour $1
}
export -f bonj => la fonction bonj est exportée
bonj Madame
autreProgShell
- - - - -
```

Après son export, la fonction *bonj* sera connue dans les sous-shells créés lors de l'exécution de *progBonjExport*.

```
Ex : $ progBonjExport
bonj : Bonjour Madame
appel a la fonction externe : bonj
bonj : Bonjour Monsieur => affiché lors de l'exécution de autreProgShell
$
```

La visibilité d'une fonction exportée est similaire à celle d'une variable locale, c'est-à-dire une visibilité *arborescente* dont la racine est le point d'export de la fonction.

Le programme shell *progBonjExport2Niv* définit et exporte la fonction *bonj*. Celle-ci est utilisée dans le programme shell *autreProg1* exécuté par *progBonjExport2Niv* et est utilisée par le programme shell *autreProg2* exécuté par *autreProg1*.

```
progBonjExport2Niv
- - - - -
#!/bin/bash
function bonj
{
echo bonj : Bonjour $1
}
export -f bonj
bonj Madame
autreProg1
- - - - -
```

```
autreProg1
- - - - -
#!/bin/bash
echo "$0 : appel a la fonction externe : bonj"
```

```
bonj Monsieur
autreProg2
- - - - -
```

```
autreProg2
- - - - -
#!/bin/bash
echo "$0 : appel a la fonction externe : bonj"
bonj Mademoiselle
- - - - -
```

```
Ex : $ progBonjExport2Niv
bonj : Bonjour Madame
./autreProg1 : appel a la fonction externe : bonj
bonj : Bonjour Monsieur
./autreProg2 : appel a la fonction externe : bonj
bonj : Bonjour Mademoiselle
$
```

XIV-7 - Commande interne return

Syntaxe : **return** [n]

La commande interne **return** permet de sortir d'une fonction avec comme code de retour la valeur **n** (**0 à 255**). Celle-ci est mémorisée dans le paramètre spécial **?**.

Si **n** n'est pas précisé, le code de retour fourni est celui de la dernière commande exécutée.

Dans le programme shell *return0*, la fonction *f* retourne le code de retour **1** au corps du programme shell.

```
retour0
- - - - -
#!/bin/bash
function f
{
echo coucou
return 1
echo a demain # jamais execute
}
f
echo code de retour de f : $?
- - - - -
```

```
Ex : $ return0
coucou
code de retour de f : 1
$
```

Remarque : il ne faut confondre **return** et **exit**. Cette dernière arrête l'exécution du programme shell qui la contient.

XIV-8 - Substitution de fonction

La commande interne **return** ne peut retourner qu'un code de retour. Pour récupérer la valeur modifiée par une fonction, on peut :

- enregistrer la nouvelle valeur dans une variable globale, ou
- faire écrire la valeur modifiée sur la sortie standard, ce qui permet à la fonction ou programme appelant de capter cette valeur grâce à une substitution de fonction : `$(fct [arg #])`.

```
recupres
-----
#!/bin/bash
function ajouteCoucou
{
echo $1 coucou
}
echo la chaine est : $( ajouteCoucou bonjour )
-----
```

La fonction *ajouteCoucou* prend un argument et lui concatène la chaîne *coucou*. La chaîne résultante est écrite sur la sortie standard afin d'être récupérée par l'appelant.

```
Ex : $ recupres
La chaine est: bonjour coucou
$
```

XIV-9 - Fonctions récursives

Comme pour les programmes shell, **bash** permet l'écriture de fonctions récursives.

Le programme *factr* implante le calcul d'une factorielle sous la forme d'une fonction shell *f* récursive. Outre la récursivité, ce programme illustre

- la définition d'une fonction « au milieu » d'un programme shell
- la substitution de fonction.

```
factr
-----
#!/bin/bash
shopt -s extglob
if (( $# != 1 )) || [[ $1 != +([0-9]) ]]
then
echo "syntaxe : factr n" >&2
exit 1
fi
function f
{
declare -i n
if (( $1 == 0 ))
then echo 1
else
(( n=$1-1 ))
n=$( f $n ) => appel récursif
echo $(( $1 * $n ))
fi
}
f $1
```

```
Ex : $ fctfactr
syntaxe : fctfactr n
$ fctfactr euztel2uz
syntaxe : fctfactr n
$ fctfactr 1
1
$ fctfactr 4
24
$
```

XIV-10 - Appels de fonctions dispersées dans plusieurs fichiers

Lorsque les fonctions d'un programme shell sont placées dans différents fichiers, on exécute ces derniers dans l'environnement du fichier shell « principal ». Cela revient à exécuter plusieurs fichiers shell dans un même environnement.

Dans l'exemple ci-dessous, pour que la fonction *f* définie dans le fichier *def_f* puisse être accessible depuis le fichier shell *appel*, on exécute *def_f* dans l'environnement de *appel* (en utilisant la commande interne **source** ou **.**). Seule la permission lecture est nécessaire pour *def_f*.

```
appel
- - - - -
#!/bin/bash
source def_f # ou plus court : . def_f
# Permissions de def_f : r--r--r#
x=2
f # appel de la fonction f contenue dans def_f
- - - - -
```

```
def_f
- - - - -
#!/bin/bash
function f()
{
echo $((x+2))
}
- - - - -
```

```
Ex : $ appel
4
$
```

