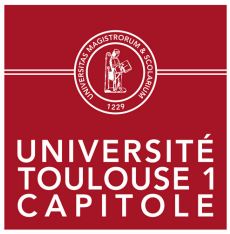


Introduction à la programmation en Bash

Version 0.2

Eric Sanchis

IUT, 50 avenue de Bordeaux, 12000 Rodez

Tél : 05.65.77.10.80 – Fax : 05.65.77.10.81 – Internet : www.iut-rodez.fr

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 et la subtilité des mécanismes de l'interpréteur afin d'insister sur quelques concepts synthétiques tels que la *substitution*, la *redirection* ou le *filtrage*.

Cette deuxième version a été revue et augmentée. Document destiné principalement aux étudiants de premier et deuxième cycle en informatique, le style adopté devrait toutefois permettre au lecteur ayant une première expérience de la « ligne de commande » de s'approprier les différentes notions présentées. Ce dernier pourra ensuite aborder des ouvrages plus extensifs ou plus spécialisés.

Cette publication étant loin d'être parfaite[1], j'encourage le lecteur à me faire parvenir ses remarques ou suggestions, ainsi qu'à me signaler toute inexactitude (eric.sanchis@iut-rodez.fr).

Pour en savoir plus :

[1] Mendel Cooper, **Advanced Bash Scripting Guide** (http://tldp.org/LDP/abs/html). Une traduction en français est disponible (http://abs.traduc.org).

[2] Cameron Newham, Bill Rosenblatt, **Le shell bash**, 3ème édition, Ed. O'Reilly, Paris, 2006.

[3] Carl Albing, JP Vossen, Cameron Newham, **Bash : le livre des recettes**, Ed. O'Reilly, Paris, 2007.

[4] Arnold Robbins, Nelson H. F. Beebe, **Introduction aux scripts shell**, Ed. O'Reilly, Paris, 2005.

Plate-forme logicielle utilisée :

Interpréteur **bash 4.2**, système **PC/Ubuntu 12.04**

Licence :

[1] Les exemples figurant dans ce document ont pour but d'illustrer les notions traitées. Il n'est donné aucune garantie quant à leur fonctionnement ou à leurs effets.

Table des matières

Chapitre 1 : Introduction à bash

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**, **dash**).

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[1]. 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*.

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 interne **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.

Ex : `sanchis@jade:/bin$`

[1] http://www.gnu.org

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*).

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

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

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 : |, &)

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.

Ex :
```
$ lpstat -a
HP-LaserJet-2420 acceptant des requêtes depuis jeu. 14 mars 2013 10:38:37 CET
HP-LaserJet-P3005 acceptant des requêtes depuis jeu. 14 mars 2013 10:39:54 CET
$
$ lpstat -a | wc -l
2
$
```

La commande unix **lpstat** permet de connaître les noms et autres informations relatives aux imprimantes accessibles. La commande unix **wc** munie de l'option **l** affiche le nombre de lignes qu'elle a été en mesure de lire.
En connectant avec un tube la sortie de **lpstat -a** à l'entrée de la commande **wc –l**, on obtient le nombre d'imprimantes accessibles sur le 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*.

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. Ils 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
- suivant le contexte, 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, seule la syntaxe la plus lisible sera présentée, parfois au détriment de la portabilité vers les autres shells ou de la compatibilité POSIX.

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 :

Ex :
```
$ ps
  PID TTY          TIME CMD
 6908 pts/4    00:00:00 bash
 6918 pts/4    00:00:00 ps
$
```
=> l'interpréteur utilisé est *bash*

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**.

Ex :
```
$ bash --version
GNU bash, version 4.2.24(1)-release (i686-pc-linux-gnu)
Copyright (C) 2011 Free Software Foundation, Inc.
Licence GPLv3+ : GNU GPL version 3 ou ultérieure <http://gnu.org/licen
ses/gpl.html>
      ...
$
```

La version de **bash** qui sera utilisée est la version **4.2**.

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* (ex : (, ||) et des *mots réservés* :

!	**case**	**do**	**done**	**elif**	**else**	**esac**
fi	**for**	**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.
Ex : **ls -l RepC RepD** : commande unix **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.

3. Commandes internes et externes

Le shell distingue deux sortes de commandes :
- les commandes internes
- les commandes externes.

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.

Ex :
```
$ 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** :

Ex :
```
$ 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**.

Ex :
```
$ 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.

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**, **cat**, **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**.

Ex :
```
$ cal
     Février 2014
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

$
```

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

Ex :
```
$ type -t sleep
file          => sleep est une commande externe
$ type -t echo
builtin       => echo est une commande interne du shell
$
```

4. Modes d'exécution d'une commande

Deux modes d'exécution peuvent être distingués :
- l'exécution séquentielle
- l'exécution en arrière-plan.

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 5 entrée date entrée
```

où **entrée** désigne la touche *entrée*, l'exécution de **date** débute après que le délai de 5 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 **;**

Ex :
```
$ 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 :

Ex :
```
$ stty -a
speed 38400 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?;
       . . .
$
```

Supposons que la commande unix **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.

Ex :
```
$ cat
^C
$
```

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** (également appelé *pid*) 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 unix **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*.

Ex :
```
$ 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) non fournie par le système d'exploitation mais par **bash**.
L'exécution en arrière-plan est souvent utilisée lorsqu'une commande est gourmande en temps CPU (ex : longue compilation d'un programme).

5. Commentaires

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

Ex :
```
$ 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.

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

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 programme (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 : jeudi 27 février 2014, 17:34:25 (UTC+0100)
$
```

4.) Lorsque l'on est un utilisateur sans privilège particulier (à la différence de l'utilisateur **root**), 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 févr. 26 16:46 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 févr. 26 16:46 premier

$
```

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

Ex :
```
$ premier
premier: commande introuvable                    => Problème !
$
```

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

Ex :
```
$ echo $PATH                    => affiche la valeur de la variable PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
$
$ pwd
/home/sanchis
$
```

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 : jeudi 27 février 2014, 17:39:27 (UTC+0100)
$
```

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:.                  => ajout du répertoire courant dans PATH
$
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:.
$
$ premier
la date du jour est : jeudi 27 février 2014, 17:41:21 (UTC+0100)
$
```

Remarque : on évitera d'appeler *test* un script shell. En effet, **test** est une commande interne de **bash** qui, exécutée sans argument, ne produit aucune sortie.
Lorsqu'une commande interne et un fichier shell portent le même nom, **bash** exécute la commande interne.

Ex :
```
$ cat test                      => contenu d'un script shell appelé test
#!/bin/bash
      #     @(#)  test

echo coucou !

$
```

```
$ ls -l test
-rwxr--r--  1 sanchis sanchis   40 févr. 26 18:46 test
$
$ test              => exécution de la commande interne test alors que
$                   =>      l'utilisateur pensait avoir lancé le shell script test !
$
$ ./test            => exécution du shell script
coucou !
$
```

Exercice 2 : En utilisant les commandes appropriées, écrire 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
$
```

Chapitre 2 : 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.

Ex : `$ echo $PATH` => affiche la valeur de la variable *PATH*

```
/usr/sbin:/usr/bin:/sbin:/bin:.:/home/sanchis/bin
$ echo $$                    => affiche la valeur du paramètre spécial $
17286
$
```

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**, **HOME**)
- 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**.

¤ 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.

Ex : `$ 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 **=**.

Ex :
```
$ b    =coucou
b : commande introuvable
$
```

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

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

Ex :
```
$ y=    coucou
coucou : commande introuvable
$
```

La chaîne *coucou* est interprétée comme la commande à exécuter.

Lorsque le shell rencontre la chaîne *$x*, il la remplace textuellement par la valeur de la variable *x*, c.-à-d. *coucou* ; en d'autres termes, la chaîne *$x* est remplacée par la chaîne *coucou*. Dans l'exemple ci-dessous, la commande finalement exécutée par **bash** est : *echo x est coucou*

Ex :
```
$ 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.

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

La nouvelle valeur de la variable *x* est constituée de la valeur précédente de *x* (chaîne *coucou*) à laquelle a été collée (concaténée) la valeur de la variable *y* (chaîne *bonjour*).

Lorsque l'on souhaite effectuer ce type de concaténation, il est possible d'utiliser l'opérateur **+=** de **bash**.

Ex :
```
$ x=coucou  y=bonjour
$ x+=$y
$ echo $x
coucoubonjour
$
```

¤ 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* **...]**

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

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

Ex :
```
$ 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* **…]**

Ex :
```
$ 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.

Ex :
```
$ 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*).

Ex :
```
$ read a b
un
$
$ echo $a
un
$
$ echo $b
                    => valeur null
$
```

Exercice 1 : Ecrire un programme shell *deuxfois* qui affiche le message "*Entrez un mot :* ", lit le mot saisi par l'utilisateur puis affiche ce mot deux fois sur la même ligne.

Ex :
```
$ 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.

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

¤ 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* …]

Ex :
```
$ 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.

Ex :
```
$ mess=salut
bash: mess : variable en lecture seule
$
```

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

Ex :
```
$ declare -r
declare -r
BASHOPTS="checkwinsize:cmdhist:expand_aliases:extglob:extquote:force_fign
ore:histappend:interactive_comments:progcomp:promptvars:sourcepath"
declare -ir BASHPID
declare -r BASH_COMPLETION="/etc/bash_completion"
declare -r BASH_COMPLETION_COMPAT_DIR="/etc/bash_completion.d"
declare -r BASH_COMPLETION_DIR="/etc/bash_completion.d"
declare -ar BASH_VERSINFO='([0]="4" [1]="2" [2]="24" [3]="1"
[4]="release" [5]="i686-pc-linux-gnu")'
declare -ir EUID="1000"
declare -ir PPID="1864"
declare -r
SHELLOPTS="braceexpand:emacs:hashall:histexpand:history:interactive-
comments:monitor"
declare -ir UID="1000"
declare -r mess="bonjour"
$
```

Plusieurs constantes sont prédéfinies dont le tableau **BASH_VERSINFO** (attribut **a**) et les entiers **EUID**, **PPID** et **UID** (attribut **i**).

¤ Variables d'environnement

Par défaut, lorsqu'un utilisateur définit une variable *var*, celle-ci reste locale à l'interpréteur **bash** qui l'a créée. Cela signifie par exemple que si l'utilisateur lance l'exécution d'un deuxième shell, celui-ci n'aura pas accès à la variable *var*.

Ex :
```
$ a=coucou          => définition de la variable locale a
$
$ echo $a
coucou              => la valeur de a est accessible
$
$ bash              => création d'un shell bash, fils de l'interpréteur précédent, qui
$                   =>      interprètera les futures commandes saisies
$ ps
  PID TTY          TIME CMD
 9918 pts/1    00:00:00 bash       => bash père
 9933 pts/1    00:00:00 bash       => bash fils
 9948 pts/1    00:00:00 ps
$
$
$ echo $a
                    => la variable a est inconnue pour le shell bash fils
$
$ exit              => terminaison du shell fils
exit
$
$ ps
  PID TTY          TIME CMD
 9918 pts/1    00:00:00 bash       => bash père
 9949 pts/1    00:00:00 ps
$
$ echo $a
coucou
$
```

En fait, l'ensemble des variables accessibles à un interpréteur (ou plus généralement à un processus) peut se décomposer en deux classes :

- les *variables locales* (comme la variable *a* de l'exemple précédent) : elles disparaissent lorsque l'interpréteur (ou le processus) se termine
- les *variables d'environnement* : celles-ci ont la particularité d'être automatiquement transmises à sa descendance, c.-à-d. copiées, à tous les shells fils (ou processus fils) qu'il créera dans le futur.

Des exemples de *variables d'environnement* usuelles sont **PATH** et **HOME**.

Pour créer une *variable d'environnement* ou faire devenir *variable d'environnement* une variable déjà existante, on utilise la commande interne **export**.

Ex :
```
$ a=coucou
$
$ b=bonjour             => définition de la variable locale b
$ export b              => la variable b devient une variable d'environnement
```

```
$
$ export c=bonsoir          => définition de la variable d'environnement c
$
$ bash                      => création d'un shell fils
$
$ echo $a
                            => la variable a est inconnue
$ echo $b
bonjour
$
$ echo $c
bonsoir                     => les variables b et c sont connues du shell fils
$
$ exit                      => fin du shell fils
exit
$
```

La commande interne **set** sans argument affiche (entre autres) toutes les variables associées à un processus alors que la commande unix **env** affiche uniquement les *variables d'environnement*.

Ex :
```
$ set
. . .
a=coucou
b=bonjour
c=bonsoir
. . .
$
$ env
. . .                       => la variable a est absente de l'environnement
b=bonjour
c=bonsoir
. . .
$
```

La transmission des *variables d'environnement* est unidirectionnelle : du processus père vers le(s) processus fils. Si un processus fils modifie la valeur d'une *variable d'environnement*, seule sa copie locale en sera affectée ; cette modification sera transparente pour son processus père.

Ex :
```
$ export a=coucou
$
$ bash
$ echo $a
coucou
$
$ a=bonsoir                 => modification de la variable d'environnement a dans le shell fils
$ echo $a
bonsoir
$
$ exit
exit
$
$ echo $a
coucou                      => la valeur n'a pas été modifiée dans le shell père
$
```

2. Paramètres de position et paramètres spéciaux

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.

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

Ex :
```
$ 23=bonjour
23=bonjour : commande introuvable
$
$ read 4
aa
bash: read: « 4 » : identifiant non valable
$
```

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* **...**

Ex :
```
$ 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.

Ex :
```
$ 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.

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

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

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

Variable prédéfinie **IFS** et commande interne **set**

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

En modifiant judicieusement la valeur de **IFS**, il est possible avec la commande interne **set** de décomposer une chaîne de caractères en plusieurs mots et de les associer à des paramètres de position.

Ex :
```
$ a=Jean:12:Rodez          => chaîne composée de 3 champs séparés par le caractère :
$ svIFS=$IFS               => sauvegarde de la valeur courante de IFS
$ IFS=:                    => le caractère : devient un séparateur
$ set $a                   => substitution, décomposition en mots puis numérotation
$ IFS=$svIFS               => restitution de la valeur originelle de IFS
$ echo "Prenom : $1, Age : $2, Ville : $3"
Prenom : Jean, Age : 12, Ville : Rodez
$
```

Remarques :

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

Ex :
```
$ set | more
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* ...**

Ex :
```
$ a=+qui
$ set $a
bash: set: +q : option non valable
set : utilisation : set [-abefhkmnptuvxBCHP] [-o option-name] [--]
[arg ...]
$ set -- $a
$
$ echo $1
+qui
$
```

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.

Ex :
```
$ 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 non vide de mots puis affiche le nombre de mots saisis.

Ex :
```
$ nbmots
Entrez une suite de mots : un deux trois quatre cinq
5              => 5 mots ont été saisis
$
```

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* ème paramètre de position en paramètre de position **1**, le *n+2* ème 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.

Ex :
```
$ set a b c d e
 =>    1 2 3 4 5
$ echo $1 $2 $#
a b 5
$
$ shift 2
=>       a b c d e          les mots a et b sont devenus inaccessibles
=>          1 2 3
$ echo $1 $3
c e
$ echo $#
3
$
```

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

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

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* :

```
#!/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 :

Ex :
```
$ 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.

Exemple : 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
----------------
```

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

2.5 Paramètres spéciaux * et @

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

Ex :
```
$ 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" ...

Ex :
```
$ set bonjour "deux coucou" salut   => trois paramètres de position sont initialisés
$
$ set "$*"                          => est équivalent à : set "bonjour deux coucou salut"
$
$ echo $#
1
$
$ echo $1
bonjour deux coucou salut
$
```

La syntaxe "$*" traite l'ensemble des paramètres de position initialisés comme une unique chaîne de caractères.

Inversement, la syntaxe "$@" produit autant de chaînes que de paramètres de positions initialisés.

Ex :
```
$ set bonjour "deux coucou" salut       => trois paramètres de position initialisés
$
$
$ set "$@"                              => est équivalent à : set "bonjour" "deux coucou" "salut"
$
$ echo $#
3
$ echo $2
deux coucou                             => l'intégrité de la chaîne a été préservée
$
```

Lorsque l'on souhaite mentionner dans une commande la liste des paramètres de position initialisés, il est conseillé d'utiliser la syntaxe "$@" car elle protège les éventuels caractères **espace** présents dans ces derniers.

Exercice 5 : Ecrire un programme shell *cx* prenant en arguments un nombre quelconque de noms d'entrées[1] et ajoute à leur propriétaire la permission **x**.

Ex :
```
$ ls -l
total 12
-rwxr--r-- 1 sanchis sanchis  15 nov.   2  2010 cx
-rw-r--r-- 1 sanchis sanchis  58 nov.   2  2010 script.bsh
-rw-r--r-- 1 sanchis sanchis 135 nov.   2  2010 une commande
$
$ cx "une commande"  script.bsh
$
$ ls -l
total 12
-rwxr--r-- 1 sanchis sanchis  15 nov.   2  2010 cx
-rwxr--r-- 1 sanchis sanchis  58 nov.   2  2010 script.bsh
-rwxr--r-- 1 sanchis sanchis 135 nov.   2  2010 une commande
$
```

3. Suppression des ambiguïtés

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

Ex :
```
$ 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
$
```

Ex :
```
$ 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
$
```

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.

[1] On appelle *entrée* tout objet pouvant être référencé dans un répertoire : fichier ordinaire, répertoire, fichier spécial, etc.

Ex :
```
$ set -o nounset                    => option nounset à l'état on
$
$ echo $e                           => variable utilisateur e non définie,
bash: e : variable sans liaison     => message d'erreur !
$
$ set --                            => paramètres de position sont réinitialisés
$ echo $1
bash: $1 : variable sans liaison
$
$ 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 deux 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*}**

Ex :
```
$ set -o nounset
$
$ ut1=root                          => ut1 définie et non vide
$ ut2=                              => ut2 définie et vide
$
$ echo $ut1
root
$
$ echo $ut2

$ echo $1
bash: $1 : variable sans liaison    => 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.

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

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

Ex :
```
$ echo $1
bash: $1 : variable sans liaison    => 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.

Ex :
```
$ 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

$
```

5. Suppression de variables

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

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

Ex :
```
$ 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 : variable sans liaison
$
```

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

Ex :
```
$ declare -r a=coucou     => définition de la constante a
$ echo $a
coucou
$ unset a
bash: unset: a : « unset » impossible : variable est en lecture seule
$
```

6. Indirection

Bash offre la possibilité d'obtenir la valeur d'une variable *v1* dont le nom est contenu (en tant que valeur *"v1"*) dans une autre variable *var*. Il suffit pour cela d'utiliser la syntaxe de substitution suivante : **${!***var***}**.

Ex :
```
$ var=v1
$ v1=un
$
$ echo ${!var}
un
$
```

Ce mécanisme, appelé *indirection*, permet d'accéder de manière indirecte et par conséquent de façon plus souple, à la valeur d'un deuxième objet.

Le fichier shell *indir* illustre cette souplesse en montrant comment il est possible de créer « à la volée » le nom de la variable qui contient l'information souhaitée.

```
#!/bin/bash
      #     @(#)  indir

agePierre=10
ageJean=20

read -p "Quel age (Pierre ou Jean) voulez-vous connaitre ? "  prenom

rep=age$prenom          # construction du nom de la variable
echo ${!rep}
```

Deux variables *agePierre* et *ageJean* contiennent respectivement l'âge des personnes Pierre et Jean. Le programme demande à l'utilisateur de saisir le prénom dont il souhaite connaître l'âge, construit le nom de la variable contenant l'âge demandé puis affiche ce dernier.

Ex :
```
$ indir
Quel age (Pierre ou Jean) voulez-vous connaitre ? Pierre
10
$
$ indir
Quel age (Pierre ou Jean) voulez-vous connaitre ? Jean
20
$
```

Ce mécanisme s'applique également aux deux autres types de paramètres : les *paramètres de position* et les *paramètres spéciaux*.

Ex :
```
$ a=un
$ b=deux
$ c=trois
$
$ set a b c          => $1, $2 et $3 ont respectivement pour valeur les chaînes
$                    =>  "a", "b", "c"
$ echo $2
b
$
$ echo ${!2}         => la valeur de $2 est la chaîne "b" qui est le nom d'une variable
deux                 =>  contenant la valeur "deux"
$
$ echo ${!3}
trois
$
$ echo ${!#}         => la valeur de $# est 3 et la valeur de $3 est la chaîne "c"
c
$
```

Remarque : il ne faut pas confondre deux syntaxes très proches fournies par **bash** mais qui correspondent à deux mécanismes totalement différents :

${!***param***}** et **${!*var****}**

La première syntaxe sera interprétée par le shell comme une indirection, tandis que la deuxième sera remplacée par les noms de variables qui commencent par *var*.

Ex :

```
$ a=alpha
$ x=beta
$ age=20
$
$ echo ${!a*}
a age
$
```

=> *${!a*}* sera remplacée par les noms de
=> variables qui commencent par *a*

Chapitre 3 : Substitution de commandes

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[1]. Ces résultats peuvent alors être affectés à une variable ou bien servir à initialiser des paramètres de position.

Ex :
```
$ pwd
/home/sanchis        => résultat écrit par pwd sur sa sortie standard
$ repert=$(pwd)      => la chaîne /home/sanchis  remplace la chaîne $(pwd)
$
$ echo mon repertoire est $repert
mon repertoire est /home/sanchis
$
```

Exercice 1 : En utilisant la substitution de commande, écrire un fichier shell *mach* affichant : "Ma machine courante est *nomdelamachine*"

Ex :
```
$ 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 : option non valable
set : utilisation : set [-abefhkmnptuvxBCHP] [-o option-name] [--] [arg
...]
$ ls -l .bashrc
-rw-r--r-- 1 sanchis sanchis 3486 mai   18  2013 .bashrc
$
```

Le premier mot issu de la substitution commence par un caractère **tiret** : la commande interne **set** l'interprète comme une suite d'options, ce qui provoque une erreur. Pour résoudre ce type de problème, on double le caractère **tiret** [cf. *Chapitre 2, §2.1*].

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

[1] 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.

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.

Ex :
```
$ ls -l .bashrc
-rw-r--r-- 1 sanchis sanchis 3486 mai   18  2013 .bashrc
$
$ taille .bashrc
3486
$
```

Plusieurs commandes peuvent être présentes entre les parenthèses.

Ex :
```
$ 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.

Ex :
```
$ date
lundi 24 mars 2014, 09:08:02 (UTC+0100)
$
$ jour
24
$
```

Exercice 4 : a) Ecrire un programme shell *heure1* qui affiche l'heure sous la forme : *heures:minutes:secondes*

Ex :
```
$ heure1
09:11:40
$
```

b) Ecrire un programme shell *heure* qui n'affiche que les heures et minutes. On pourra utiliser la variable prédéfinie **IFS** du shell [cf. *Chapitre 2, § 2.1*].

Ex :
```
$ heure
09:11
$
```

Exercice 5 : 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**.

Les substitutions de commandes peuvent être imbriquées.

Ex :
```
$ ls .gnome2
keyrings  nautilus-scripts
$
$ set $( ls $(pwd)/.gnome2 )
$ echo $#          => nombre d'entrées visibles[2] du répertoire .gnome2
2
$
```

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

Ex :
```
$ read rep
.gnome2
$ set  $( ls  $( pwd )/$rep )    => substitutions de deux commandes et d'une
$                                => variable
$ echo $2
nautilus-scripts
$
```

La substitution de commande est utile pour capter les résultats écrits par un fichier shell lors de son exécution.

Ex :
```
$ jour                  => [cf. Exercice 3]
24
$
$ resultat=$(jour)      => la variable resultat contient la chaîne 24
$
```

En utilisant ce mécanisme, un processus père peut récupérer les modifications locales effectuées par un processus fils sur son propre environnement [cf. *Chapitre 2, §1.3*].

Ex :
```
$ cat modifLOGNAME
#!/bin/bash

LOGNAME=Eric      # modification de la variable d'environnement
echo $LOGNAME
$
```

Le fichier shell *modifLOGNAME* modifie et affiche la nouvelle valeur de la variable d'environnement **LOGNAME** avant de se terminer.

Ex :
```
$ echo $LOGNAME
sanchis                          => valeur initiale de la variable d'environnement
$
$ LOGNAME=$(modifLOGNAME)        => capture de la nouvelle valeur
$
$ echo $LOGNAME
Eric                             => la variable d'environnement a été modifiée
$
```

Exécutée par un processus fils, la modification apportée par *modifLOGNAME* sur son propre environnement est transmise au processus père (l'interpréteur **bash**) via :

- l'affichage de la nouvelle valeur (côté fils)
- la substitution de commande (côté père).

[2] Une entrée est dite *visible* si son nom ne commence pas par un **point**.

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 informations régionales.

Ex :
```
$ locale
LANG=fr_FR.UTF-8
LANGUAGE=
LC_CTYPE="fr_FR.UTF-8"
LC_NUMERIC="fr_FR.UTF-8"
LC_TIME="fr_FR.UTF-8"
LC_COLLATE="fr_FR.UTF-8"
LC_MONETARY="fr_FR.UTF-8"
LC_MESSAGES="fr_FR.UTF-8"
LC_PAPER="fr_FR.UTF-8"
LC_NAME="fr_FR.UTF-8"
LC_ADDRESS="fr_FR.UTF-8"
LC_TELEPHONE="fr_FR.UTF-8"
LC_MEASUREMENT="fr_FR.UTF-8"
LC_IDENTIFICATION="fr_FR.UTF-8"
LC_ALL=
$
```

Des commandes telles que **date** utilisent ces valeurs pour afficher leurs résultats.

Ex :
```
$ date
lundi 24 mars 2014, 16:28:02 (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.

Toutefois, 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

Ex :
```
$ LC_ALL=C date
Mon Mar 24 16:29:10 CET 2014
$
```

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 :

Ex :

```
$ set $(LC_ALL=C date) ; echo $3
24
$
```

Chapitre 4 : 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-rw-r-- 1 sanchis sanchis 0 mars  27 15:58 1
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:58 a
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:58 _a
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:58 A
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:58 à
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:58 ami
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:58 an
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:59 Arbre
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:59 e
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:59 é
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:59 émirat
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:59 En
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:59 état
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:59 minuit
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:59 zaza
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:59 Zoulou
$
```

La valeur des paramètres régionaux influe grandement sur l'interprétation 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-rw-r-- 1 sanchis sanchis 0 Mar 27 15:58 1
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:58 A
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:59 Arbre
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:59 En
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:59 Zoulou
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:58 _a
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:58 a
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:58 ami
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:58 an
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:59 e
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:59 minuit
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:59 zaza
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:58 ??
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:59 ??
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:59 ??mirat
-rw-rw-r-- 1 sanchis sanchis 0 Mar 27 15:59 ??tat
$
```

On s'aperçoit que l'interprétation des noms d'entrées contenant des caractères accentués est altérée.

1. Caractères génériques

Les caractères génériques du shell sont : ***** **?** **[]**

Remarque : il ne faut pas confondre la syntaxe [] du méta-langage [cf. *Chapitre 1, § 2*] 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**.

Ex :
```
$ set -o
allexport         off
braceexpand       on
emacs             on
errexit           off
errtrace          off
functrace         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
pipefail          off
posix             off
privileged        off
verbose           off
vi                off
xtrace            off
$
```

1.1 Le caractère *

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

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

Tous les noms d'entrées sont affichés, triés.

Ex :
```
$ ls -l a*
```

```
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:58 a
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:58 ami
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:58 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 unix **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
$
```

Remarques :

- quand aucune entrée ne correspond au modèle fourni, les caractères et expressions génériques ne sont pas interprétés (cf. option **nullglob** à la *fin de ce chapitre*).

Ex :
```
$ echo Z*t
Z*t
$
```

- le caractère **.** qui débute le nom des fichiers cachés (ex: **.profile**) ou de répertoires (ex : **..**) ainsi que le caractère **/** doivent être explicitement mentionnés dans les modèles (cf. option **dotglob** à la *fin de ce chapitre*).

Ex :
```
$ pwd
/home/sanchis
$ echo .pr*  ../../bi*
.profile ../../bin
$
```

1.2 Le caractère ?

Le caractère générique **?** désigne un et un seul caractère.

Ex :
```
$ echo ?            => noms d'entrées composés d'un seul caractère
1 a A à e é
$
$ echo ?n
an En
$
```

Plusieurs caractères génériques différents peuvent être présents dans un modèle.

Ex :
```
$ echo ?mi*
ami émirat
$
```

Exercice 1 : Ecrire un modèle permettant d'afficher le nom des entrées dont le deuxième caractère est un *a*.

Exercice 2 : Ecrire un modèle permettant d'afficher le nom des entrées dont le premier caractère est un *a* suivi par deux caractères quelconques.

1.3 Les caractères []

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 [].

Ex :
```
$ ls -l [ame]
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:58 a
-rw-rw-r-- 1 sanchis sanchis 0 mars  27 15:59 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 **-** .

Ex :
```
$ 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).

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

Ex : `$ echo [[:lower:]]` => noms d'entrées constitués d'une seule minuscule

```
a à e é
$
```

Si l'on omet de placer le nom de la classe entre les caractères génériques **[]**, ce sont les caractères [] du nom de la classe qui seront interprétés par le shell comme caractères génériques.

Ex : `$ echo [:lower:]` => sera interprété comme `[:lower:]`

```
e
$
```

Pour afficher les noms d'entrées commençant par une majuscule :

Ex : `$ echo [[:upper:]]*`

```
A Arbre En Zoulou
$
```

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

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

```
a A ami an Arbre En Zoulou
$
```

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 **[**.

Ex : `$ echo ?[!n]*` => noms d'entrées ne comportant pas le caractère *n* en 2ème position

```
_a ami Arbre émirat état minuit zaza Zoulou
$
$ echo [![:lower:]]
1 A
$
$ echo [![:upper:]]*
1 a _a à ami an e é émirat état minuit zaza
$
$ echo [![:upper:]]*[!at]
ami an
$
```

2. Expressions génériques

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

Ex : `$ shopt`

```
autocd          off
cdable_vars     off
cdspell         off
checkhash       off
checkjobs       off
checkwinsize    on
cmdhist         on
compat31        off
compat32        off
```

```
compat40          off
compat41          off
dirspell          off
dotglob           off
execfail          off
expand_aliases    on
extdebug          off
extglob           on
extquote          on
failglob          off
force_fignore     on
globstar          off
gnu_errfmt        off
histappend        on
histreedit        off
histverify        off
hostcomplete      off
huponexit         off
interactive_comments    on
lastpipe          off
lithist           off
login_shell       off
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

Dans une expression générique, *liste_modèles* désigne une suite d'un ou plusieurs modèles séparés par un caractère |. Dans ce contexte, le caractère | signifie OU.

Ex :
```
$ 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
$
$ echo !(*at|a*)
```

```
1 _a A à Arbre e é En minuit zaza Zoulou
$
```

Exercice 3 : Ecrire un modèle correspondant aux noms d'entrées comportant plus de deux caractères.

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

3. Options relatives aux caractères et expressions génériques

(1) 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*

(2) 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***).

A l'état ***on*** :

dotglob signifie que les noms d'entrées commençant par **.** (caractère **point**) seront également traités

nocaseglob signifie que les majuscules et minuscules ne seront pas différenciées

nullglob signifie qu'un modèle ne correspondant à aucune entrée sera remplacé par une chaîne vide et non par le modèle lui-même

failglob signifie qu'un modèle ne correspondant à aucune entrée provoquera une erreur.

Ex :

```
$ pwd
/home/sanchis/glob
$
$ ls -la
total 8
drwxr-xr-x 2 sanchis sanchis 4096 nov.  24  2011 .
drwxr-xr-x 5 sanchis sanchis 4096 nov.  27  2012 ..
-rw-r--r-- 1 sanchis sanchis    0 nov.  24  2011 .sessions
-rw-r--r-- 1 sanchis sanchis    0 nov.  24  2011 video
-rw-r--r-- 1 sanchis sanchis    0 nov.  24  2011 viDEos
$
$ echo p*
p*                                  => comportement par défaut
$
$ shopt -s nullglob
$
```

```
$ echo p*
                                        => chaîne vide
$
$ shopt -s failglob
$
$ echo p*
bash: Pas de correspondance : p*        => message d'erreur
$
$ echo *s*
viDEos                                  => comportement par défaut
$
$ shopt -s dotglob
$
$ echo *s*
.session viDEos                         => fichier caché .session pris en compte
$
$ echo *d*
Video                                   => comportement par défaut
$
$ shopt -s nocaseglob
$
$ echo *d*
video viDEos
$
```

(3) A l'état ***on***, l'option **noglob** de la commande interne **set** supprime l'interprétation des caractères et expressions génériques.

Ex :
```
$ ls -la
total 8
drwxr-xr-x 2 sanchis sanchis 4096 mars  27 20:21 .
drwxr-xr-x 7 sanchis sanchis 4096 mai    4 16:37 ..
-rw-r--r-- 1 sanchis sanchis    0 nov.  24  2011 .sessions
-rw-r--r-- 1 sanchis sanchis    0 nov.  24  2011 video
-rw-r--r-- 1 sanchis sanchis    0 nov.  24  2011 viDEos
$
$ shopt -u dotglob      => le fichier caché .session ne sera pas pris en compte
$ shopt -s extglob      => interprétation des expressions génériques activée
$ set +o noglob         => interprétation des caractères génériques activée
$
$ echo +([[:lower:]]) *
video video videos
$
$ set -o noglob         => suppression de l'interprétation
$ set -o | grep noglob
noglob          on
$
$ echo +([[:lower:]]) *
+([[:lower:]]) *        => aucune interprétation
$
```

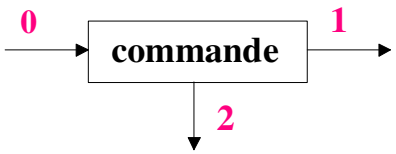
Chapitre 5 : Redirections élémentaires

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.



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.

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.

2.1 Redirection de la sortie standard

Syntaxe : **>** *fichier* ou **1>** *fichier*

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

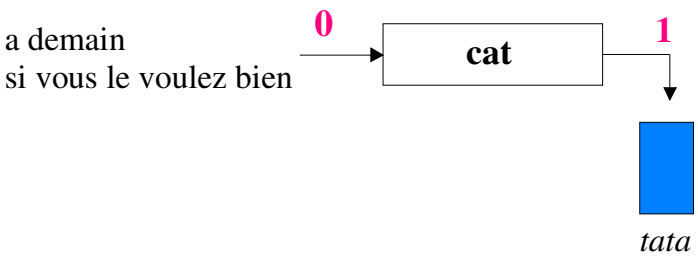
Ex :
```
$ >f            => crée ou remet à zéro le fichier f
$
$ ls -l f
-rw-rw-r-- 1 sanchis sanchis 0 mars  31 10:26 f
$
```

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.

Ex :
```
$  > f1  ;  >f2
$
```

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

Ex :
```
$ 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
$
```



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*

Ex :
```
$ pwd > t
$
$ date >> t
$
$ cat  t
/home/sanchis
lundi 31 mars 2014, 10:28:24 (UTC+0200)
$
```

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

2.2 Redirection de la sortie standard pour les messages d'erreur

Syntaxe : **2>** *fichier*

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

Ex :
```
$ pwd
/home/sanchis
$ ls  vi          => l'éditeur de texte vi se trouve dans le répertoire /usr/bin
ls: impossible d'accéder à vi: Aucun fichier ou dossier 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 [cf. *Chapitre 7*] de la commande plutôt que par les résultats ou messages d'erreur qu'elle engendre.

Pour écrire un message *mess* sur la sortie standard pour les messages d'erreur :
```
echo >&2  mess
```

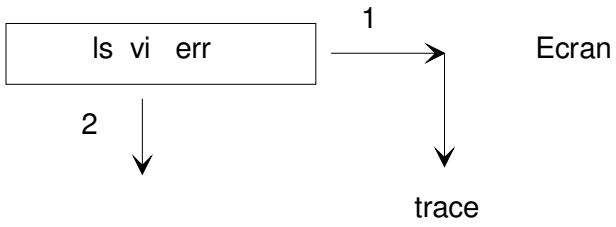
Comme pour la sortie standard, il est possible de concaténer la sortie standard pour les messages d'erreur d'une commande au contenu d'un fichier : **2>>** *fichier*

Ex :
```
$ ls  vi
ls: impossible d'accéder à vi: Aucun fichier ou dossier de ce type
$
$ ls vi 2>err
$
$ ls date 2>>err
$
$ cat err
ls: impossible d'accéder à vi: Aucun fichier ou dossier de ce type
ls: impossible d'accéder à date: Aucun fichier ou dossier de ce type
$
```

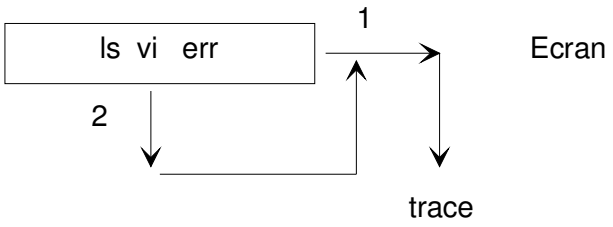
Pour rediriger la sortie standard pour les messages d'erreur vers la sortie standard (c.-à-d. vers le fichier de descripteur **1**), on utilisera la syntaxe : **2>&1**
Cela est souvent utilisé lorsqu'on désire conserver dans un même fichier toutes les sorties.

Ex :
```
$ ls vi err  >trace  2>&1
$
$ cat trace
ls: impossible d'accéder à vi: Aucun fichier ou dossier 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].



[a]



[b]

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

Ex :
```
$ ls vi err  &> trace
$
$ cat trace
ls: impossible d'accéder à vi: Aucun fichier ou dossier de ce type
err
$
```

L'ajout en fin de *fichier* s'effectue en utilisant la syntaxe **&>>** *fichier*

Ex :
```
$ ls /etc/passwd date &>> trace
$
$ cat trace
ls: impossible d'accéder à vi: Aucun fichier ou dossier de ce type
err
ls: impossible d'accéder à date: Aucun fichier ou dossier de ce type
/etc/passwd
$
```

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

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

Pour éviter que le contenu d'un fichier ne soit écrasé lors d'une redirection malheureuse, il est possible d'utiliser l'option **noclobber** de la commande interne **set**.

Ex :
```
$ cat t
bonjour
$
$ set -o noclobber        => activation de l'option noclobber
$
$ echo coucou > t
bash: t : impossible d'écraser le fichier existant
```

```
$
$ ls vi 2>t
bash: t : impossible d'écraser le fichier existant
$
$ echo ciao >> t
$                          => la concaténation reste possible
$ cat t
bonjour
ciao
$
$ set +o noclobber         => pour désactiver l'option
$
```

2.3 Redirection de l'entrée standard

Syntaxe : **<** *fichier*

Ex :
```
$  mail  sanchis  < lettre
$
```

La commande unix **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.

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

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 fic          =>  cat ouvre le fichier fic afin de lire son contenu
bonjour
et au revoir
$
$ cat  <fic        =>  cat lit son entrée standard (redirigée par le shell), sans savoir
bonjour            =>      qu'il s'agit du contenu du fichier fic
et au revoir
$
```

Il n'en est pas ainsi avec la commande unix **wc** : celle-ci n'écrit pas les résultats de la même manière suivant qu'un argument lui a été fourni ou bien qu'elle lise son entrée standard.

Ex :
```
$ wc -l  fic_noms
3 fic_noms
$
```

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

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

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

Exercice 2 : Ecrire un programme shell *nblignes* prenant un nom de fichier en argument et qui affiche le nombre de lignes de ce fichier.

Ex :
```
$ nblignes fich_noms
3
$
```

2.4 Redirections séparées des entrées / sorties standard

Les entrées / sorties peuvent être redirigées indépendamment les unes des autres.

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

La commande **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.

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

2.5 Texte joint

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

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* comprenant éventuellement plusieurs lignes et délimité par deux balises *mot*.
La deuxième balise *mot* doit impérativement se trouver en début de ligne. Par contre, un ou plusieurs caractères **espace** ou **tabulation** peuvent être présents entre **<<** et *mot*.

Ex :
```
$ a=3.5  b=1.2
$
$ bc <<  EOF
> $a + $b          => la chaîne "> " indique que la commande n'est syntaxiquement
> EOF              =>      pas terminée
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 puis affiche son résultat sur sa sortie standard.

Lorsque **bash** détecte qu'une commande n'est pas syntaxiquement terminée, il affiche une chaîne d'appel différente (contenue dans la variable prédéfinie **PS2** du shell) matérialisée par un caractère **>** suivi d'un caractère **espace**, invitant l'utilisateur à continuer la saisie de sa commande.

2.6 Chaîne jointe

Syntaxe : *cmd* **<<<** *chaîne*

C'est une syntaxe plus compacte que le *texte joint* : le contenu transmis à l'entrée standard de la commande *cmd* se présente sous la forme d'une chaîne de caractères *chaîne*.

Ex :
```
$ a=1.2  b=-5.3
$
$ bc <<< "$a + $b"
-4.1
$
```

Le choix d'utiliser un *texte joint* ou bien une *chaîne jointe* s'effectue suivant la structure et la longueur du texte à transmettre à la commande *cmd*.

Exercice 2 : Ecrire un programme shell *add* prenant deux nombres en arguments et qui affiche leur somme.

Ex :
```
$ add 1.2 -6
-4.8
$
```

2.7 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>&-**

Ex :
```
$ >&-   echo coucou
bash: echo: erreur d'écriture : Mauvais descripteur de fichier
$
```

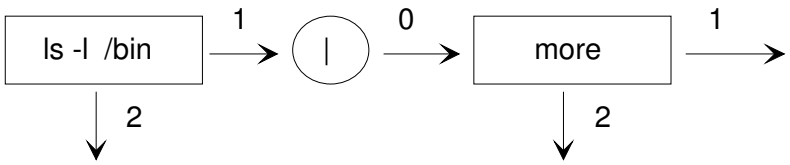
Il y a fermeture de la sortie standard puis une tentative d'écriture sur celle-ci : l'erreur est signalée par l'interpréteur de commandes.

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 unix **more**). En effet, les informations affichées par **ls -l** sont envoyées vers l'entrée de la commande unix **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.



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.

3.1 Pipelines

De manière simplifiée, on appelle *pipeline*, une suite non vide de commandes connectées par des tubes : cmd_1 | ... | cmd_n

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.

Ex :
```
$ date  |  tee trace1 trace2 | wc -l
1            => date n'écrit ses résultats que sur une seule ligne
$
$ cat trace1
lundi 31 mars 2014, 10:31:56 (UTC+0200)
$
$ cat trace2
lundi 31 mars 2014, 10:31:56 (UTC+0200)
$
```

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**.

3.2 Tubes et chaînes jointes

Le shell crée un processus différent pour chaque commande d'un tube cmd_1 | cmd_2. Cela provoque l'effet suivant : si cmd_2 modifie la valeur d'une variable, cette modification disparaitra avec la fin du processus qui exécute cette commande.

Ex :
```
$ a=bonjour
$ echo $a | read rep          # (a)
$ echo $rep
      => la variable rep ne contient pas la chaîne bonjour ; elle n'est pas initialisée
$
```

Si l'on fait afficher la valeur de la variable *rep*, ligne (*a*), par le même processus qui a lu la valeur, alors on s'aperçoit que l'affectation a bien été effectuée mais celle-ci disparait avec la fin de ce processus.
Pour voir ce comportement, on place les deux commandes à l'intérieur d'une paire de **parenthèses** (cf. *Chapitre 6*).

Ex :
```
$ echo $a | ( read rep ; echo $rep )
bonjour            => cet affichage montre que l'affectation à rep a bien été effectuée
$
$ echo $rep
            => variable rep non initialisée
$
```

Cela provient du fait qu'il y a deux variables *rep* différentes : une qui est créée et utilisée par le processus qui exécute la commande *read rep* et une qui est créée et utilisée par le processus qui exécute la commande *echo $rep*.

L'utilisation d'une *chaîne jointe* permet de résoudre de manière élégante ce problème de transmission de données entre processus.

Ex :
```
$ read rep <<< "$a"
$
$ echo $rep
bonjour
$
```

4. Substitution de processus

La substitution de processus généralise la notion de tube. La principale forme de substitution de processus est la suivante : *cmd* **<**(*suite_cmds*)

Le shell crée un fichier temporaire, connecte la sortie de *suite_cmds* à ce fichier temporaire puis lance l'exécution de cette suite de commandes. Cette suite de commandes écrit ses résultats dans le fichier temporaire. Ensuite, le shell remplace la syntaxe **<**(*suite_cmds*) par la référence à ce fichier temporaire et lance l'exécution de la commande *cmd*.
Du point de vue de cette commande *cmd*, tout se passe comme si les données produites par *suite_cmds* étaient présentes dans un fichier ordinaire dont l'utilisateur lui aurait passé le nom en argument.

Ex :
```
$ ls -l
total 8
-rw-rw-r-- 1 sanchis sanchis 140 mars  31 10:29 err
-rw-rw-r-- 1 sanchis sanchis   0 mars  31 10:26 f
-rw-r--r-- 1 sanchis sanchis 156 mars  31 10:30 trace
$
$ wc -l <(ls -l)
4 /dev/fd/63            => la commande ls –l affiche 4 lignes
$
```

Dans l'exemple ci-dessus, la sortie de la commande **ls –l** a été enregistrée dans un fichier temporaire */dev/fd/63*. Puis, la commande *wc –l /dev/fd/63* a été exécutée.

La substitution de processus est intéressante car elle permet de créer en parallèle plusieurs flots de données qui seront ensuite traités séquentiellement par la même commande.

La syntaxe de la commande se présente alors sous la forme :
cmd **<**(*suite_cmds*$_1$) **<**(*suite_cmds*$_2$) **…**
L'exécution des suites de commandes *suite_cmds*$_i$ est effectuée en parallèle, chaque flot de résultats produit par celles-ci étant enregistré dans un fichier temporaire différent. Puis, le shell substitue chaque syntaxe **<**(*suite_cmds*$_i$) par la référence au fichier temporaire correspondant. Enfin, la commande *cmd* est exécutée munie de ces différents arguments.

L'exemple ci-dessous illustre le cas où l'on souhaite afficher la première ligne de deux fichiers *fich1* et *fich2* non triés et pouvant contenir des doublons.

Ex :
```
$ cat fich1
Pierre
Anne
Pierre
Pierre
Jean
$
$ cat fich2
Paul
Paul
Pierre
Jean
$
$ head -q -n 1   <(sort fich1 | uniq)   <(sort fich2 | uniq)
Anne
Jean
$
```

Dans un premier temps, les fichiers *fich1* et *fich2* sont triés (commande unix **sort**) en parallèle et débarrassés de leurs doublons (commande unix **uniq**). Le résultat du traitement de ces deux fichiers est placé dans deux fichiers temporaires préalablement créés par **bash** (les contenus de *fich1* et *fich2* restent, eux, inchangés). Puis, la première ligne (option **–n 1** de la commande unix **head**) de chacun de ces deux fichiers est affichée, sans mentionner les noms de fichiers (option **-q**).

En résumé, la substitution de processus **<(***suite_cmds***)** peut être utilisée dans une commande *cmd* en lieu et place d'un nom de fichier dont le contenu sera lu par cette commande.

Chapitre 6 : 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.

Ex :
```
$ 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 :

Ex :
```
$ 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.

Ex :
```
$ { 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: Erreur de syntaxe près du symbole inattendu « } »
$
```

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

Ex :
```
$ 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.

Ex :
```
$ 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.

Ex :
```
$ 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.

Ex :
```
$ { 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 l'entrée standard originelle (**/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.

Ex :
```
$ 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 aussi être utilisé pour effectuer une redirection globale.

Ex :
```
$ ( pwd ; date ; echo FIN ) > fin
$
$ cat fin
/home/sanchis
lundi 31 mars 2014, 10:55:31 (UTC+0200)
FIN
$
```

Chapitre 7 : Code de retour [1]

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**.

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*).

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

Exercice 1 : 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.

Ex :
```
$ 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 égal 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 **?**.

---

[1] Ce texte est paru sous une forme légèrement différente dans la revue « Linux Magazine France », n°39, mai 2002

Ex :
```
$ pwd                          => mise à zéro du paramètre spécial ?
/home/sanchis
$
$ echo $?
0
$ ls -l vi &                   => commande exécutée en arrière-plan
[1] 5577
$ ls: impossible d'accéder à vi: Aucun fichier ou dossier de ce type

[1]+  Termine 2                  ls --color=auto -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.**

Ex :
```
$ :                    => commande deux-points
$ echo $?
0
$
```

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

Ex :
```
$ 1>&- echo coucou
bash: echo: erreur d'écriture : 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*.

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

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

Ex :
```
$  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$"*

Ex :
```
$  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.

Ex :
```
$ 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 tutu
grep: tutu: Aucun fichier ou dossier de ce type
$
$ echo $?
2                          => le fichier tutu n'existe pas !
$
```

Exercice 2 : 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.

Ex :
```
$ connu  bert
1
$ connu albert
0
$
```

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).

Ex :
```
$ lvi
ls: impossible d'accéder à vi: Aucun fichier ou dossier 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.

Ex :
```
$ lvi1
ls: impossible d'accéder à vi: Aucun fichier ou dossier 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**.

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
```

Ex :
```
$ lvi2
ls: impossible d'accéder à vi: Aucun fichier ou dossier 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*).

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*.

Ex :
```
$ pwd ; ls vi ; echo bonjour
/home/sanchis
ls: impossible d'accéder à vi: Aucun fichier ou dossier 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*.

Ex :
```
$ cat pass tutu | grep sanchis
cat: tutu: Aucun fichier ou dossier de ce type
sanchis:x:103:100::/home/sanchis:/bin/bash
$
$ echo $?
0
$
```
=> code de retour de *grep sanchis* (le code de retour de *cat pass tutu* est 1)

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**.

Ex :
```
$ ! ls pass
pass
```
=> le code de retour de *ls pass* est égal à 0

```
$
$ 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**.

Ex :
```
$ ! grep sanchis tutu
grep: tutu: Aucun fichier ou dossier de ce type
$
$ echo $?
0
$
```

5. Code de retour d'une commande lancée en arrière-plan

Il n'existe pas de mécanisme direct pour récupérer le code de retour d'une commande exécutée de manière asynchrone. Une méthode permettant de l'obtenir consiste à utiliser le *paramètre spécial* **!** (à ne pas confondre avec le *mot-clé* **!** présenté ci-dessus) et la commande interne **wait**.
Le *paramètre spécial* **!** contient le numéro d'identification (*pid*) de la dernière commande exécutée en arrière-plan.
La commande interne **wait** munie d'un pid ou d'un numéro de job attend la fin du processus correspondant et place le code de retour de ce dernier dans le *paramètre spécial* **?**.

Ex :
```
$ grep bert tutu & wait $!          => (a)
[1] 6427
grep: tutu: Aucun fichier ou dossier de ce type
[1]+  Termine 2                  grep --color=auto bert tutu
$
$ echo $?                                  => (b)
2
$
```

Le shell lance la commande *grep bert tutu* en arrière-plan et attend qu'elle se termine (a), puis affiche son code de retour (b).

6. 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.

Ex : `$ 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
```

7. 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.

Ex :
```
$ 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 *tutu* n'existe pas.

Ex :
```
$ ls pass tutu
ls: impossible d'accéder à tutu: Aucun fichier ou dossier de ce type
pass
$
$ rm tutu || echo tutu non efface
rm: impossible de supprimer «tutu»: Aucun fichier ou dossier de ce type
tutu non efface
$
```

Le fichier *tutu* n'existant pas, la commande *rm tutu* 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.

Ex :
```
$ ls pass || ls tutu || echo fini aussi
pass
$
```

Le code de retour de *ls pass* est égal à **0** car *pass* existe, la commande *ls tutu* ne sera donc pas exécutée. D'autre part, le code de retour de l'ensemble *ls pass || ls tutu* 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** :

Ex :
```
$ ls tutu || ls pass || echo fini
ls: impossible d'accéder à tutu: Aucun fichier ou dossier de ce type
pass
$
```

Le code de retour de *ls tutu* 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 || :

Ex :
```
$ ls pass || ls tutu || 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 tutu* n'est pas exécutée : le code de retour de l'ensemble *ls pass || ls tutu* est égal à **0**, la commande *echo suite et* n'est pas exécutée. Le code de retour de *ls pass || ls tutu || echo suite et* est égal à **0**, donc la commande *echo fin* est exécutée !

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

Ex :
```
$ ls pass && ls tutu && echo fini
pass
ls: impossible d'accéder à tutu: Aucun fichier ou dossier de ce type
$
$ ls tutu && ls pass && echo suite et && echo fin
ls: impossible d'accéder à tutu: Aucun fichier ou dossier de ce type
$
$ ls pass && ls tutu && echo suite et || echo fin
pass
ls: impossible d'accéder à tutu: Aucun fichier ou dossier de ce type
fin
$
```

Exercice 3 :

a) Ecrire un programme shell *present0* prenant en argument un nom d'utilisateur et affiche uniquement le code de retour **0** si l'utilisateur est connecté ou sinon affiche **1**.

b) Modifier ce programme (soit *present1*) pour qu'il affiche le message "Connecte" si l'utilisateur est connecté et n'affiche rien sinon.

c) Modifier ce programme (soit *present2*) pour qu'il affiche le message "Connecte" si c'est le cas et affiche le message "Pas connecte" sinon.

d) Modifier ce programme (soit *present*) pour qu'il vérifie préalablement si le nom de l'utilisateur passé en argument existe dans le fichier **/etc/passwd**. Si ce n'est pas le cas, le programme ne doit rien afficher et se terminer avec le code de retour 1. Sinon, il doit se comporter comme *present2*.

Chapitre 8 : Structures de contrôle *case* et *while*

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 ; cela 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. *Chapitre 4, 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 égal à **0** si aucun modèle n'a pu correspondre à la valeur de *mot*. Sinon, c'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 réponse : "  rep
case $rep in
o|O )    echo OUI ;;
*)       echo Indefini
esac
```

Rq : - une substitution (de paramètre, de commande ou autre) doit être présente après le mot-clé **case** (caractère $)

- il n'est pas obligatoire de terminer la dernière *suite_de_commandes* par **;;**

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. Aucune vérification n'est effectuée sur le nombre d'arguments passés au programme.

```
#!/bin/bash
#      @(#)   nombre

shopt  -s extglob
case $1 in
+([[:digit:]]) )  echo "$1 est une suite de chiffres" ;;
esac
```

Rq : pour que les *expressions génériques* puissent être traitées dans un fichier shell, il est nécessaire d'activer l'option correspondante (**shopt -s extglob**)

Exercice 1 : 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 2 : 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 3 : Tout système Linux n'est pas nativement configuré pour afficher la date en français. En utilisant la commande unix **date** et la locale standard [cf. *Chapitre 3, Substitution de commandes*], écrire un programme shell *datefr* qui affiche la date courante en français de la manière suivante :

mercredi 5 janvier 2005 10:00

Ignorer la casse :

Si l'on souhaite ne pas distinguer *majuscules* et *minuscules* lors de la comparaison avec les modèles, il suffit de positionner l'option **nocasematch** de la commande interne **shopt**.

Soit le fichier shell *OUI* :

```
#!/bin/bash
#      @(#)   OUI

read -p "Entrez un mot : "  mot

shopt -s nocasematch
case $mot in
 oui ) echo "Vous avez saisi le mot oui" ;;
 * )  echo "$mot n'est pas le mot oui" ;;
esac
```

Ce programme shell reconnait le mot *oui,* écrit avec des *majuscules* ou des *minuscules*.

Ex :
```
$ OUI
Entrez un mot : Oui
Vous avez saisi le mot oui
$
```

```
$ OUI
Entrez un mot : aoui
aoui n'est pas le mot oui
$
$ OUI
Entrez un mot : oUI
Vous avez saisi le mot oui
$
```

Remarque : il ne faut pas confondre les options **nocasematch** et **nocaseglob**. L'option **nocasematch** ne traite pas la casse lors de l'exécution des commandes internes **case** et **[[** [cf. *Chapitre 11*, *Entiers et expressions arithmétiques*] tandis que l'option **nocaseglob** ne traite pas la casse lors de la génération des noms d'entrées [cf. *Chapitre 4, Caractères et expressions génériques*].

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 réexécutée. Si son code de retour est différent de **0**, 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 **0**.

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 réexécutée. 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 sort 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            => pour arrêter l'exécution : kill  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 shell *wh* qui affiche les noms des utilisateurs connectés

```
#!/bin/bash
       #      @(#)   wh

who  > tmp
while  read nom reste
do
     echo $nom
done < tmp
rm tmp
```

Exercice 4 : 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 : programme shell *wh1*

```
#!/bin/bash
#       @(#)   wh1

who | while  read nom reste
      do
        echo $nom
      done
```

Par rapport au programme shell *wh*, il n'y a pas de fichier temporaire *tmp* à gérer.

Exercice 5 : Un utilisateur a la possibilité d'envoyer un message à un autre utilisateur connecté sur la même machine à l'aide de la commande **write**.
Ecrire un programme shell *acceptmess* qui affiche le nom de tous les utilisateurs connectés qui acceptent les messages.

La commande **who –T** indique pour chaque terminal ouvert si l'utilisateur accepte les messages (+) ou les refuse (-).

Ex :
```
$ who -T
bond         - pts/0        2013-12-02 15:53 (10.2.2.11:0.0)
sanchis      + pts/1        2013-12-02 15:53 (10.2.2.11:0.0)
sanchis      + pts/2        2013-12-02 15:53 (10.2.2.21:0.0)
$
$ acceptmess
sanchis
sanchis
$
```

Exercice 6 :

a) Ecrire un programme shell *etatmess1* prenant en argument un nom d'utilisateur et affiche pour chaque terminal qu'il a ouvert, s'il accepte (*oui*) ou refuse (*non*) les messages. On ne considèrera aucun cas d'erreur.

Ex :
```
$ etatmess1  bond
bond        pts/0    non
$
```

b) Modifier ce programme, soit *etatmess2*, pour qu'il affiche un message d'erreur lorsque le nom de l'utilisateur passé en argument n'existe pas.

c) Modifier le programme précédent, soit *etatmess*, pour qu'il vérifie également le nombre d'arguments passés lors de l'appel.

Remarques sur la lecture d'un fichier avec **while**

1) 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 avec l'option **–d** le caractère séparateur de champ (par défaut, il s'agit du caractère **tabulation**) ; les numéros de champs doivent alors être indiqués avec l'option **-f**.

Exemple : programme shell *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*).

Lorsque le traitement à effectuer sur le contenu d'un fichier est plus complexe, il est préférable d'utiliser des commandes spécialisées telles que **awk** et **sed**.

2) 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, simple redirection ou substitution de processus).

Prenons l'exemple où l'on souhaite mettre à jour 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 unix **ls** à l'entrée de la commande interne **while**. Après lecture de la première ligne, la valeur de la variable *var* est modifiée puis l'itération se termine (commande interne **break**). 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** [cf. *Chapitre 5, Redirections élémentaires, § 3*].

Exemple : programme shell *maj* :

```
#!/bin/bash
       #   @(#)    maj

var=0
ls | while read
         do
            var=1
            break
         done

echo $var
```

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

Pour résoudre ce problème, il suffit d'utiliser une substitution de processus [cf. *Chapitre 5, Redirections élémentaires, § 4*] à la place d'un tube.

Exemple : programme shell *maj1* :

```
#!/bin/bash
       #   @(#)    maj1

var=0
while read
do
    var=1
   break
```

```
done < <(ls)

echo $var
```

Ex :
```
$ maj1
1
$
```

En utilisant la substitution de processus **<(***ls***)**, le shell crée un fichier temporaire dans lequel sont écrites les données produites par l'exécution de la commande unix *ls*. Le contenu de ce fichier alimente ensuite l'entrée standard de la commande interne **while** à l'aide d'une redirection **<**.
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.

3) On ne doit pas utiliser le couple de commandes internes **while read** pour lire le contenu d'un *fichier binaire*.

Chapitre 9 : Chaînes de caractères

1. Protection de caractères

1.1 Mécanismes

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 :
```
$ ls
tata  toto
$ echo *
tata toto
$ echo \*     => le caractère * perd sa signification de caractère générique
*
$
$ echo \\  => le deuxième caractère \ perd sa signification de caractère de protection
\
$
```

Remarque : dans les exemples de cette section, les caractères ou touches **en vert** ont perdu leur signification particulière, les caractères ou touches **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, la signification de la touche **Entrée** a été inhibée par le caractère \ : **bash** détecte que la commande interne **echo** n'est pas terminée et affiche la chaîne d'appel contenue dans la variable prédéfinie **PS2** du shell.

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

¤ Protection partielle **"***chaîne***"**

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

Ex :
```
$ echo  "< * $PWD  ' >"
< * /home/sanchis ' >
$
$ echo  "\"$PS2\""
"> "                    => valeur de la variable prédéfinie PS2 entre guillemets
$
```

¤ Protection totale **'***chaîne***'**

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

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

L'utilisation du caractère **apostrophe** peut provoquer l'erreur suivante.

Ex :
```
$ echo c'est lundi
>                       => le shell interprète l'apostrophe comme un début de chaîne à
>'                      =>  protéger et attend par conséquent la deuxième apostrophe
cest lundi


$
```

Pour éviter cela, on peut supprimer la signification particulière de cette **apostrophe** ou bien utiliser une paire de **guillemets**.

Ex :
```
$ echo N\'oublie pas !
N'oublie pas !
$
$ echo "c'est lundi"
c'est lundi
$
```

1.2 Exemples d'utilisation

¤ Certaines commandes unix telles que **find** utilisent pour leur propre fonctionnement les mêmes caractères génériques que ceux du shell. Utilisés sans précaution, ces caractères provoquent le plus souvent une erreur d'exécution. Par exemple, la commande

*find . -name *.py –print*

recherche à partir du répertoire courant et affiche (ou plutôt devrait afficher) tous les noms d'entrées se terminant par la chaîne *.py*.

Ex :
```
$ ls -l
total 20
-rw-r--r-- 1 sanchis sanchis 1280 oct.   5  2010 cliTCP.py
```

```
drwxr-xr-x 2 sanchis sanchis 4096 oct.   5  2010 Divers
-rwxr--r-- 1 sanchis sanchis  117 oct.   5  2010 ou
-rwxr--r-- 1 sanchis sanchis  117 oct.   5  2010 ou0
-rw-r--r-- 1 sanchis sanchis 2397 oct.   5  2010 servTCP.py
$
$ find . -name *.py -print
find: les chemins doivent précéder l'expression : servTCP.py
Utilisation : find [-H] [-L] [-P] [-Olevel] [-D
help|tree|search|stat|rates|opt|exec] [chemin...] [expression]
$
```

L'origine du problème est que le caractère * aurait dû être interprété par la commande **find** alors qu'il a été interprété par **bash**, ce qui a conduit à l'exécution de la commande

find . -name cliTCP.py servTCP.py –print

La commande **find** signale une erreur de syntaxe car elle n'attend qu'une seule chaîne après son option *–name*.

Pour corriger le problème, il suffit de protéger le caractère * de l'interprétation de **bash**.

Ex :
```
$ find . -name "*.py" -print
./servTCP.py
./cliTCP.py
$
```

Dans ce cas de figure, le type de protection utilisé n'a pas d'importance : on aurait également pu écrire *'*.py'* ou **.py*.

¤ La présence de caractères **espace** (ou de caractères spéciaux de **bash**) dans la valeur d'un paramètre (variable ou paramètre de position) peut poser problème lorsque cette valeur participe à l'exécution d'une commande.

Soit le programme shell *nblig0* qui affiche le nombre de lignes (commande **wc –l**) d'un fichier dont le nom est saisi par l'utilisateur et contenu dans la variable *rep*.

```
#!/bin/bash
#     @(#)  nblig0

read -p "Saisissez un nom d'entree : " rep
wc -l $rep
```

Ex :
```
$ ls -l
total 12
-rw-r--r-- 1 sanchis sanchis 1728 nov.   1  2010 Meteo du jour.txt
-rwxr--r-- 1 sanchis sanchis   83 nov.   1  2010 nblig
-rwxr--r-- 1 sanchis sanchis   82 nov.   1  2010 nblig0
$
$ nblig0
Saisissez un nom d'entree : Meteo du jour.txt
wc: Meteo: Aucun fichier ou dossier de ce type
wc: du: Aucun fichier ou dossier de ce type
wc: jour.txt: Aucun fichier ou dossier de ce type
0 total
$
```

La variable *rep* contient bien la chaîne *Meteo du jour.txt* mais après substitution de *$rep*, la commande qui sera finalement exécutée est : *wc –l Meteo du jour.txt*

La commande unix **wc** n'est pas exécutée avec un seul argument mais avec trois. Chacun d'eux est considéré par **wc** comme un nom de fichier qui, absent, provoque une erreur d'exécution.

La solution à ce problème consiste à préserver l'intégrité de la valeur de la variable *rep* après que la substitution ait été effectuée par le shell. On utilise pour cela la syntaxe *"$rep"*.
Il est à noter que la syntaxe *'$rep'* aurait été inadéquate car elle interdit au shell d'effectuer la substitution *$rep*.

Ex :
```
$ cat nblig
#!/bin/bash
#     @(#)  nblig

read -p "Saisissez un nom d'entree : " rep
wc -l "$rep"

$
$ nblig
Saisissez un nom d'entree : Meteo du jour.txt
35 Meteo du jour.txt
$
```

De manière générale, il est fortement conseillé de préserver l'intégrité d'une chaîne de caractères obtenue après une substitution, surtout lorsque l'on ne connait pas à l'avance la composition de cette chaîne.

Cette politique est systématiquement suivie dans l'écriture des programmes shell qui accompagnent l'interpréteur **bash** (ex : **.bashrc**, **.profile**).

Ex :
```
$ cat ~/.profile
# ~/.profile: executed by the command interpreter for login shells.
#     . . .

# if running bash
if [ -n "$BASH_VERSION" ]; then
      # include .bashrc if it exists
      if [ -f "$HOME/.bashrc" ]; then
            . "$HOME/.bashrc"
      fi
fi

# set PATH so it includes user's private bin if it exists
#if [ -d "$HOME/bin" ] ; then
#     PATH="$HOME/bin:$PATH"
#fi
$
```

Les fichiers shell **.bashrc** et **.profile** permettent à l'utilisateur de configurer son environnement de travail [cf. *Chapitre 1, Introduction à Bash, § 1.1*]. Ces deux fichiers sont présents dans le répertoire de connexion de l'utilisateur dont le chemin peut être rapidement désigné par le caractère **~**.

¤ Avant qu'une syntaxe spécifique ne soit introduite dans **bash**, le mécanisme d'indirection [cf. *Chapitre 2, Substitution de paramètres, § 6*] était obtenu à l'aide de la commande interne **eval** et des caractères de protection du shell.

La syntaxe de cette commande interne est la suivante : **eval [*arg* ...]**

La commande interne **eval** interprète au sens du shell (substitution de paramètres, etc.) chacun de ses arguments *arg* puis concatène la chaîne résultante en une seule chaîne. Celle-ci est ensuite à nouveau interprétée et exécutée comme une commande par le shell.
En d'autres termes, **eval** permet d'effectuer une interprétation en deux passes.

Ex :
```
$ var=v1
$ v1=un
$ eval echo \$$var
un
$
```

Lors de la première passe, le première caractère *$* n'est pas interprété car il est protégé de l'interprétation du shell (*\$*) et la chaîne *$var* est remplacée par la valeur de la variable *var*. A l'issu de la première interprétation, on obtient la chaîne suivante : *echo $v1*
Cette chaîne est ensuite interprétée comme commande par **bash**.

Une autre manière de procéder est la suivante :

```
$ var=\$v1
$ v1=un
$
$ eval echo $var
un
$
```

Cette formulation permet de généraliser le mécanisme et d'obtenir une indirection multi niveau.

Ex :
```
$ var=\$v1
$ v1=\$v2
$ v2=\$v3
$ v3=trois
$
$ eval eval eval echo $var
trois
$
```

Les interprétations successives donnent les résultats suivants :

```
$ echo $var
$v1
$ eval echo $var
$v2
$ eval eval echo $var
$v3
$
$ eval eval eval echo $var
trois
$
```

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
$ echo ${#PWD}
13                          =>  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 de cette chaîne est 1
```

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 "82a34a"
$
$ echo ${1#*a}
34a                         => suppression de la sous-chaîne 82a
$
```

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 72a34ab
$
$ echo ${1##*a}
b
$
```

Suppression de la plus courte sous-chaîne à droite

Syntaxe : **${*paramètre*%*modèle*}**

Ex:
```
$ echo  $PWD
/home/sanchis
$ echo ${PWD%/*}          => suppression de la sous-chaîne /sanchis
/home
$
```

Suppression de la plus longue sous-chaîne à droite

Syntaxe : **${*paramètre*%%*modèle*}**

Dans l'exemple ci-dessous, 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).

Ex :
```
$ eleve="Pierre Dupont::12:10::15:9"
$
$ echo ${eleve%%:*}             => extraction du prénom et nom
Pierre Dupont
$
```

Exercice 1 : En utilisant les modificateurs de chaînes,

a) Ecrire un programme shell *touslesutil* qui lit le contenu du fichier **/etc/passwd** (utilisation de la syntaxe **while read**) et affiche le nom des utilisateurs enregistrés (1er champ du fichier).

b) Modifier ce programme (soit *touslesutiluid*) pour qu'il affiche le nom et l'uid de chaque utilisateur (1er et 3ème champs).

Exercice 2 : En utilisant les modificateurs de chaînes,

a) Ecrire 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.

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"
$ #     01234567..10
$
$ echo ${ch:3}
defghijk
$
```

${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
$
```

Ces deux syntaxes extraient des sous-chaînes uniquement à partir de leur position. La commande unix **expr**, plus générale, permet d'extraire une sous-chaîne suivant un modèle exprimé sous la forme d'une *expression régulière*.

Exercice 3 :

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).

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/}
${paramètre//mod/} : suivant la syntaxe utilisée, lorsque la chaîne ***ch*** est absente, la première ou toutes les occurrences satisfaisant le modèle ***mod*** 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***) satisfaisant le modèle. Le traitement à effectuer est la suppression.

Si on le souhaite, il est possible de préciser 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
$
```

6. Transformation en majuscules/minuscules

${paramètre^^} : transforme la valeur de **paramètre** en *majuscules*.
Les caractères autres que les *minuscules* ne sont pas modifiés.

Ex :
```
$ a=bonjour
$
$ echo ${a^^}
BONJOUR
$
$ a=aB,c:D
$
$ echo ${a^^}
AB,C:D
$
$ set un dEuX::  trois
$
$ echo ${2^^}
DEUX::
$
```

Pour que la valeur d'une variable *var* soit toujours en *majuscules*, on utilise la commande interne : **declare –u** *var*

Ex :
```
$ declare -u x
$
$ x=coucou
$
$ echo $x
COUCOU
$
$ x=aB,c:D
$
$ echo $x
AB,C:D
$
```

${paramètre,,} : transforme la valeur de **paramètre** en *minuscules*.
Les caractères autres que les *majuscules* ne sont pas modifiés.

Ex :
```
$ b="CouCOU :)"
$
$ echo ${b,,}
coucou :)
$
$ set Un,uN deux trois
$
$ echo ${1,,}
un,un
$
```

Pour que la valeur d'une variable *var* soit toujours en *minuscules*, on utilise la commande interne : **declare –l** *var*

Ex :
```
$ declare -l y
$
$ y=COUCoU
$
$ echo $y
coucou
$
$ y='COU  COU Cou !'
$
$ echo $y
cou cou cou !
$
```

7. Formatage de chaînes

Lorsque l'on souhaite afficher une chaîne de caractères sur la sortie standard, il est d'usage d'utiliser **echo**, commande interne historique des premiers shells. Pourtant, **echo** souffre de plusieurs handicaps :

- au cours du temps, différentes versions de cette commande interne ont vu le jour, ayant à la fois des syntaxes et des comportements différents
- le formatage des chaînes à afficher est malaisé.

L'intérêt de la commande interne **printf** est qu'elle ne présente pas ces inconvénients. Issue directement de la fonction ***printf*** de la bibliothèque standard du langage C, elle en partage les principales caractéristiques.

Sa syntaxe est la suivante : **printf** *chaîne_format* **[** *argument1* **...** **]**

Comme sa consœur du langage C, *chaîne_format* est constituée de caractères littéraux et de spécificateurs de format. Les caractères littéraux seront affichés tels quels sur la sortie standard tandis que les spécificateurs, introduits à l'aide du caractère **%**, indiqueront à la fois le type et le formatage à utiliser pour afficher les arguments qui suivent *chaîne_format*.
Pour provoquer un retour à la ligne, il faudra également utiliser la séquence d'échappement **\n**.

Pour que **printf** fonctionne correctement, il est fortement conseillé de fournir autant d'arguments qu'il y a de spécificateurs dans *chaîne_format*.

Dérivant de la fonction C ***printf***, les possibilités syntaxiques offertes sont trop riches pour être complètement détaillées[1]. C'est pourquoi seuls quelques spécificateurs seront présentés.

Les types pris en compte sont les entiers (**%d**), les réels (**%f**), les chaînes (**%s**) et les caractères (**%c**).

```
$ printf "%d kilo(s) de %s (categorie %c) a %f euros l'unite\n" 2 fruits A 3,45
2 kilo(s) de fruits (categorie A) a 3,450000 euros l'unite
$
```

Suivant la représentation souhaitée, il existe plusieurs spécificateurs pour un même type. Par exemple, pour le type des réels, on peut utiliser le spécificateur **%f** (notation usuelle) ou **%e** (notation scientifique). Le spécificateur **%g** évite l'affichage des **0** superflus.

```
$ printf "%f ou %e\n"  13,45  13,45
13,450000 ou 1,345000e+01
$
% printf "%g %g\n"  13,4500  3,450000e+01
13,45 34,5
$
```

Entre le caractère **%** et l'indicateur de conversion tel que **d**, **f**, etc. peuvent être mentionnés différents attributs qui affineront la représentation affichée. Par exemple,

%10d : affiche l'entier sur 10 caractères. Par défaut, la justification est *à droite*.

```
$ printf "'%10d'\n" 123
'       123'
$
```

%-20s : justifie la chaîne *à gauche* sur 20 caractères.

```
$ printf "'%20s'\n" coucou
'              coucou'
$ printf "'%-20s'\n" coucou
'coucou              '
$
```

La justification *à gauche* est également utilisable avec les autres types.

```
$ printf "'%-10d'\n" 123
```

[1] L'exécution des commandes **man 1 printf** et **man 3 printf** permet de se faire une idée plus précise des possibilités offertes par la commande interne **printf**.

```
'123       '
$
```

Si la longueur spécifiée est trop courte pour l'affichage, elle est ignorée. La valeur mentionnée correspond en fait à un nombre minimal de caractères à afficher.

```
$ printf "'%2s'\n" coucou
'coucou'
$
```

Si à la place d'une longueur numérique on utilise le caractère *, c'est l'argument qui précède l'argument à afficher qui sera interprété comme la longueur d'affichage.

```
$ read -p "longueur d'affichage : "  gabarit
longueur d'affichage : 10
$
$ printf "'%*d'\n"  ${gabarit}  123
'       123'       => affichage sur 10 caractères
$
$ read -p "longueur d'affichage : "  gabarit
longueur d'affichage : 5
$
$ printf "'%-*s'\n"  ${gabarit}  cou
'cou  '            => affichage sur 5 caractères
$
```

%.3f : un nombre situé après un caractère **point** indique la précision souhaitée. Pour un nombre réel, il s'agit du nombre de chiffres après la virgule à afficher.

```
$ printf "%.3f\n" 12,34589
12,346
$
$ printf "%.3f\n" 2,34
2,340
$
```

Si la précision est appliquée à une chaîne de caractères, elle sera interprétée comme un nombre maximal de caractères à afficher.

```
$ printf "%.3s\n" coucou
cou
$
```

Certains caractères non imprimables (ex : le caractère **interligne**) disposent d'une séquence d'échappement (ex : **\n**) qui facilite leur utilisation. Quelques exemples de séquences d'échappement reconnues sont : **\t** (tabulation horizontale), **\r** (retour chariot), **\a** (bip).

```
$ printf "\tcoucou\n"
      coucou
$
```

Enfin, l'option **–v** de **printf** permet d'affecter à une variable, une chaîne formatée par cette commande interne. Dans ce cas, la chaîne résultat n'est pas affichée.

Ex :
```
$ printf -v util "prenom:%-10s/taille:%.2f" Pierre 1,8345
$
$ echo "$util"
prenom:Pierre     /taille:1,83
$
```

8. Génération de chaînes de caractères

Bash offre la possibilité de créer automatiquement une liste de chaînes de caractères à l'aide de la syntaxe générale suivante :

[préfixe]{chaîne1,[chaîne2 …]}[suffixe]

Entre la paire d'**accolades**, les chaînes doivent être séparées par une **virgule** (pas de caractère **espace** par exemple). Au moins un caractère **virgule** doit être présent entre les **accolades**.

Ce mécanisme de substitution très particulier prend tout son sens lorsqu'il est utilisé avec un *préfixe* : *préfixe{chaîne1,chaîne2 …}*
La chaîne résultat est alors constituée de toutes les combinaisons <*préfixe*><*chaîne*$_i$> possibles. Cette syntaxe est utilisée, par exemple, pour créer en une seule commande une suite de répertoires en mentionnant une seule fois le chemin du répertoire destination.

Ex :
```
$ mkdir -p Projets/P1/{src,include,bin}
$
$ ls -l Projets/P1
total 12
drwxrwxr-x 2 sanchis sanchis 4096 avril  1 08:07 bin
drwxrwxr-x 2 sanchis sanchis 4096 avril  1 08:07 include
drwxrwxr-x 2 sanchis sanchis 4096 avril  1 08:07 src
$
```

L'option **–p** de la commande unix **mkdir** crée les répertoires *Projets* et *P1* s'ils n'existaient pas.

L'option **braceexpand** de la commande interne **set** permet d'activer ou d'inhiber la substitution d'accolades.

Ex :
```
$ set -o | grep braceexpand
braceexpand        on                  => le mécanisme est déjà activé
$
$ set +o braceexpand                   => désactivation
$
$ echo {coucou,}
{coucou,}
$
```

Chapitre 10 : Structures de contrôle *for* et *if*

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 shell ***for_arg***

```
#!/bin/bash

for i
do
  echo $i
  echo "Passage a l'argument suivant ..."
done
```

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

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

Exercice 1 : Ecrire un programme shell *uid_gene* prenant un nombre quelconque de noms d'utilisateurs et qui affiche pour chacun d'eux son *uid*. Ce dernier peut être obtenu directement à l'aide de la commande : **id -u** *nomutilisateur*

Ex : `$ uid_gene root sanchis`

```
root 0
sanchis 1002
$
```

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 shell ***for_set***

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

```
set $(date)

for i
do
  echo $i
done
------------------
```

Ex :
```
$ for_set
mardi
1
avril
2014,
09:10:06
(UTC+0200)
$
```

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 shell ***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 shell ***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 2 : 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.

2. Choix if

2.1 Fonctionnement

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 shell ***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
$
$ rm1  toto
toto a ete supprime
$
$ rm1  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 ...
  then ...
  elif ...
     then ...
fi
```

```
if ...
  then ...
  else  if ...
           then ...
        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
```

2.2 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 avril  1 09:14 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 shell ***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
```

Ex :
```
$ affic .                => traitement du répertoire courant
. : repertoire
affic  err  Exercices  for_liste  for_set  rm1 testval
$
```

. Renseignements divers sur une entrée :

-a *entrée*	vraie si	*entrée* existe
-s *entrée*	vraie si	*entrée* existe et sa taille est différente de **0** Rq : la taille d'un *répertoire vide* est toujours différente de **0**
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*

Ex :
```
$ > err                  => création d'un fichier err vide
$
$ ls -l err
-rw-rw-r-- 1 sanchis sanchis 0 avril  1 08:35 err
$
$ if [[ -a err ]]
> then echo err existe
> fi
err existe
$
$ if [[ -s err ]]
> then echo le contenu du fichier err est non vide
> else echo son contenu est vide
> fi
son contenu est vide
$
```

Exercice 3 : Ecrire un programme shell *vide* prenant un nom de répertoire en argument et qui indique s'il est vide ou non vide.
Indication : on pourra utiliser l'option **–A** de la commande **ls**.

. Longueur d'une chaîne de caractères :

-z *ch*	vraie si	la longueur de la chaîne *ch* est égale à zéro
ch (ou bien **-n** *ch*)	vraie si	la longueur de la chaîne *ch* est différente de zéro

Ex :
```
$ a=            => la variable a est définie et est vide
$
$ if [[ -z $a ]]
> then echo la longueur de a est egale a 0
> fi
la longueur de a est egale a 0
$
$ echo $#
0               => aucun paramètre de position n'est initialisé
$
$
$ if [[ -z $1 ]]
> then echo la longueur est egale a 0
> fi
la longueur est egale a 0
$
```

. 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.

Ex :
```
$ a=bonjour A=Bonjour
$
$ if [[ $a < $A ]]
> then echo $a precede $A
> elif [[ $a > $A ]]
> then echo $a suit $A
> else echo elles sont egales
> fi
bonjour precede Bonjour
$
```

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*. Ces derniers ne doivent pas être protégés de l'interprétation de **bash** lorsqu'ils sont placés entre les caractères [[ ]].

Ex :
```
$ a="au revoir"
$
$ [[ $a == 123 ]]          => faux
$
$ echo $?
1
$
$ [[ $a == a* ]]           => vrai : la valeur de a commence par le caractère a ; on
$                          =>     peut remarquer la non protection du caractère *
$ echo $?
0
$
```

```
$ b=123
$
$ if [[ $b == +([[:digit:]]) ]]
> then
>   echo "c'est une suite de chiffres"
> fi
c'est une suite de chiffres
$
$ c=BonJOuR
$
$ shopt -s nocasematch          => minuscules et majuscules ne sont pas
$                               =>   distinguées
$ [[ $c == bonjour ]]
$
$ echo $?
0           => la valeur de la variable c correspond au modèle bonjour
$
```

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

Exemple : programme shell ***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
$
$ a=bonjour testval bonjour    => la variable a est locale au fichier shell testval,
OUI                 => elle est initialisée lors de l'appel du fichier shell
$
```

Remarque : il existe un opérateur **=~** qui permet de mettre en correspondance une chaîne de caractère *ch* avec une ***expression régulière*** *expr_reg* : *ch* **=~** *expr_reg*
Contrairement aux *caractères* et *expressions génériques*, les *expressions régulières* ne sont pas spécifiques à **bash**.

Exercice 4 :

a) Ecrire un programme shell *minchaines* prenant au moins une chaîne de caractères en argument et qui affiche la plus petite (au sens lexicographique).
Aucun cas d'erreur ne doit être traité.

b) Modifier ce programme pour qu'il vérifie qu'il y a au moins un argument et qu'aucun argument n'est la chaîne vide ("").

. 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 oct.  25  2011 /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 **0** car l'expression conditionnelle globale est vraie.

Attention : il ne faut pas confondre les opérateurs **! && ||** utilisés par la commande interne **[[** pour construire les expressions conditionnelles et ces mêmes opérateurs **! && ||** portant sur les codes de retour [cf. *Chapitre 7, Code de retour*].

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

Ex :
```
$ ls -l toto
--w------- 1 sanchis sanchis 7 mai   28 14:26 toto
$
$ [[ -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: [: trop d'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
$
```

Chapitre 11 : Entiers et expressions arithmétiques

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 b=-6
$
$ b=7
bash: b : variable en lecture seule        => 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 BASHPID
declare -ir EUID="1000"
declare -i HISTCMD
declare -i LINENO
declare -i MAILCHECK="60"
declare -i OPTIND="1"
declare -ir PPID="20391"
declare -i RANDOM
declare -ir UID="1000"
declare -ir b="-6"
declare -i v="12"
declare -i w="34"
declare -i x="35"
$
```

2. Représentation d'une valeur de type entier

Comme en langage C, un littéral entier (valeur de type entier) qui ne commence pas par le chiffre **0** est considéré comme étant exprimé en *base décimale* (ex : 11, -234). Un littéral qui commence par

le chiffre **0** est interprété comme un *nombre octal* (ex : 071). S'il débute par **0x** ou **0X**, il est interprété comme un *nombre hexadécimal* (ex : 0xA, 0X0d) : les minuscules et majuscules ne sont pas distinguées.

Bash permet d'exprimer des nombres dans une base allant de **2** à **64**. On utilise pour cela la syntaxe : *base#a*
Toutefois, les résultats seront exprimés en *base décimale*.

Ex :
```
$ declare -i a="023 + 3#11"
$
$ echo $a
23              => le résultat est exprimé en base 10
$
$ echo 3#$a
3#23            => tentative infructueuse pour exprimer le résultat en base 3 !
$
```

La commande interne **printf** accepte en arguments des entiers exprimés en base **10**, **8** ou **16**.

Ex :
```
$ printf "%d %d %d\n" 23 023 0x23
23 19 35            => valeurs exprimées en décimal
$
```

Inversement, elle est capable d'afficher une valeur directement dans une de ces trois bases.
Dans la chaîne *format* de **printf**, on utilise la syntaxe :

%d pour *afficher en décimal* un *entier quelconque*
%u pour *afficher en décimal* un *entier non signé*
%o pour *afficher en octal* un *entier non signé*
%x ou **%X** pour *afficher en hexadécimal* un *entier non signé*

Ex :
```
$ printf "%o\n" 23
27              => valeur décimale 23 exprimée en octal
$
```

Exercice 1 : Ecrire un convertisseur ***h2d*** prenant en argument un entier exprimé en *hexadécimal* et qui affiche sa représentation en *décimal*. On supposera que l'argument est une valeur hexadécimale correcte.

Ex :
```
$ h2d 0x34e5F
216671
$
$ h2d 0X12
18
$
```

Exercice 2 : Ecrire un convertisseur ***d2h*** prenant en argument un entier en *base 10* positif ou nul et qui affiche sa représentation en *hexadécimal*. On supposera que l'argument est une valeur décimale correcte.

Ex :
```
$ d2h +10
a
$
```

```
$ d2h 156
9c
$
```

3. Intervalles d'entiers

Pour spécifier un intervalle d'entiers, on utilise la syntaxe : {n_1..n_2[..*pas*]}
Les bornes de l'intervalle sont n_1 et n_2. Le shell remplace cette syntaxe par la liste des nombres entiers compris entre n_1 et n_2 inclus. Par défaut, le pas d'incrémentation est **1** (ou **-1** si $n_1 > n_2$).

Ex :
```
$ echo {4..23}
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
$
$ echo {5..-7}
5 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7
$
```

Il est possible d'ajouter un ou plusieurs **0** devant les bornes pour que l'affichage de chaque valeur s'effectue avec le même nombre de caractères.

Ex :
```
$ echo {04..15}
04 05 06 07 08 09 10 11 12 13 14 15
$
$ echo {004..010}
004 005 006 007 008 009 010
$
$ echo {04..-7}
04 03 02 01 00 -1 -2 -3 -4 -5 -6 -7
$
$ echo {04..-07}
004 003 002 001 000 -01 -02 -03 -04 -05 -06 -07
$
```

De manière facultative, un troisième nombre *pas* peut fixer le pas d'incrémentation ou de décrémentation.

Ex :
```
$ echo {-4..7..3}
-4 -1 2 5
$
```

Combinés avec le mécanisme de génération de chaînes [cf. *Chapitre 9, Chaînes de caractères §8*], les intervalles d'entiers sont utilisés pour créer rapidement des noms de fichiers numérotés.

Ex :
```
$ touch fich{0..4}
$
$ ls
fich0  fich1  fich2  fich3  fich4
$
```

L'exemple ci-dessous utilise un intervalle d'entiers et une itération **for** pour afficher les dix chiffres.

Ex :
```
$ for i in {0..9}
> do
```

```
> echo $i
> done
0
1
2
3
4
5
6
7
8
9
$
```

Remarque : le paramétrage des bornes d'un intervalle d'entiers ne peut s'effectuer directement car les substitutions ne sont pas interprétées à l'intérieur des accolades.

Ex :
```
$ a=1 b=5
$
$ echo {$a..$b}
{1..5}
$
```

Pour que l'interprétation soit exécutée, on peut utiliser la commande interne **eval**. Cela permet, par exemple, de construire une itération paramétrée.

Ex :
```
$ eval echo {$a..$b}
1 2 3 4 5
$
$ for i in $( eval echo {$a..$b} )
> do
> echo $i
> done
1
2
3
4
5
$
```

4. 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 10 avril 2014, 17:39:55 (UTC+0200)
$
$ set $(date)
$
$ (( b = $2 +1 ))          => incrémentation du jour courant
$
$ echo $b
11
$
```

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 de *i* 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
$
```

5. 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*6789999999999999999999 ))
-848393034087410691      => nombre négatif !
$
```

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

Ex :
```
$ bc -q
897655*785409*56789*6789999999999999999999
2718562508883222424499599622605466388845
quit                    => pour sortir de bc
$
```

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
$
```

6. 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 du langage C.

Les opérateurs traités par la commande ((sont mentionnés ci-dessous, munis de leur associativité (g : de gauche à droite, d : de droite à gauche). 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 ~a	négation logique, négation binaire	(d)
(5)	a**b	exponentiation : a^b	(d)
(6)	a*b a/b a%b	multiplication, division entière, reste	(g)
(7)	a+b a-b	addition, soustraction	(g)
(8)	a<<n a>>n	décalage binaire à gauche, à droite	(g)
(9)	a<b a<=b a>b a>=b	comparaisons	(g)
(10)	a==b a!=b	égalité, différence	(g)
(11)	a&b	ET binaire	(g)
(12)	a^b	OU EXCLUSIF binaire	(g)
(13)	a|b	OU binaire	(g)
(14)	a&&b	ET logique	(g)
(15)	a||b	OU logique	(g)
(16)	a?b:c	opérateur conditionnel	(d)

(17) a=b a*=b a%=b opérateurs d'affectation (d)
a/=b a+=b a-=b
a&=b a^=b a|=b
a<<=n a>>=n

(18) a,b opérateur virgule (g)

Ces opérateurs se décomposent en :
- opérateurs arithmétiques
- opérateurs relationnels
- opérateurs logiques
- opérateurs binaires
- opérateurs d'affectation
- 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++ ))
$ echo "y=$y  x=$x"
y=9  x=10
$
```
=> *y* reçoit la valeur 9 ; *x* vaut 10

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

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

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

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

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

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

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

Plus unaire :	**+** *expr_arith*
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 :
```
$ declare -i  b=8  c=2
$ (( 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 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 ))
0
$
```
=> `echo $(( !12 ))` provoque une erreur

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 ))
0
$
$ echo $(( a > 3 &&  a <5 ))
1
$ ((a>3 && a<5 ))
$ echo $?
0
$
```
=> expression *fausse*

=> expression *vraie*, valeur égale à 1

=> expression *vraie*, code de retour égal à 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
$
```

Opérateurs binaires :

Négation binaire : ~ *expr_arith*
Décalage binaire à gauche : *expr_arith* **<<** *nb_bits*
Décalage binaire à droite : *expr_arith* **>>** *nb_bits*
Et binaire : *expr_arith* **&** *expr_arith*
Ou exclusif binaire (**xor**) : *expr_arith* **^** *expr_arith*
Ou binaire : *expr_arith* **|** *expr_arith*

Ces opérateurs sont utilisés pour manipuler la valeur d'entiers via leur représentation binaire.

Ex :
```
$ declare -i a=2#1000011110100100
$
$ printf "%d\n" $a
34724
$
$ printf "%0x\n" $a
87a4
$
$ declare -i b=$(( a & 0xf )) => les 4 bits de droite de a sont copiés dans b
$
$ printf "%d\n" $b
4                          => b : 2#0100
$
$ (( b = b | 2#1 ))              => le bit de droite de b est mis à 1
$ printf "%d\n" $b
5                          => b : 2#0101
$
```

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.

Dans l'exemple ci-dessous, la valeur *5* est préalablement affectée à la variable *c* : la valeur de l'expression *c=5* est par conséquent égale à *5*, puis cette valeur est affectée à la variable *b*, etc.

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*.

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 divers :

Exponentiation : *expr_arith1* ** *expr_arith2*

Contrairement aux autres opérateurs, l'opérateur d'exponentiation ** n'est pas issu du langage C. Son associativité est de droite à gauche (**d**).

Ex :
```
$ echo $(( 2**3**2 ))
512
$
```
=> interprétée comme `2**(3**2)` : 2^9

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
$
```

7. 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        # affiche les dix chiffres
for (( x=0 ; x<10 ; x++ ))
do
      echo $x
done
```

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

Exercice 3 : En utilisant **RANDOM**, écrire un programme ***tirage_flash*** qui affiche cinq entiers différents compris entre 1 et 49 et un nombre compris entre 1 et 10.

Exercice 4 : 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 5 : Ecrire une version récursive ***factrecur*** du programme précédent.

8. Exemple : les tours de Hanoi

Le problème des « tours de Hanoï » 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.

```
#!/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
$
```

Chapitre 12 : Tableaux

Moins utilisés que les *chaînes de caractères* ou les *entiers*, **bash** intègre les *tableaux classiques* et les *tableaux associatifs*. Les éléments des premiers sont indicés par des *entiers*, tandis que les éléments des seconds sont indexés par des *chaînes de caractères*.
Les tableaux *classiques* et *associatifs* sont monodimensionnels.

1. Tableaux classiques

1.1 Définition et initialisation

¤ Pour créer un ou plusieurs tableaux classiques vides, on utilise généralement l'option **-a** 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 lister les tableaux classiques 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]="4" [1]="2" [2]="24" [3]="1"
[4]="release" [5]="i686-pc-linux-gnu")'
declare -a DIRSTACK='()'
declare -a FUNCNAME='()'
declare -a GROUPS='()'
declare -a PIPESTATUS='([0]="0")'
$
```

¤ Pour désigner un élément d'un tableau classique, on utilise la syntaxe : *nomtab*[*indice*]

Ex :
```
$ declare  -a   tab      => définition du tableau tab
$
$ read  tab[1]  tab[3]
coucou  bonjour
$
$ tab[0]=hello
$
```

¤ Pour définir et initialiser un tableau classique : **declare -a** *nomtab*=(*val0 val1* **...**)

Comme en langage C, l'indice d'un tableau classique 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.

Il n'est pas obligatoire d'utiliser la commande interne **declare** pour créer un tableau classique, 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 classique :

- ***nomtab*=(*val0 val1* ...)**

- ***nomtab*=([*indice*]=*val* ...)**

Ex :
```
$ arr=([1]=coucou  [3]=hello)
$
```

- l'option **-a** de la commande interne **read** ou **readonly**. Chaque mot saisi devient un élément du tableau classique :

Ex :
```
$ read  -a  tabmess
bonjour tout  le  monde
$
```

¤ Pour créer un tableau classique en lecture seule, on utilise les options **–ra**.

Ex :
```
$ declare -ra tabconst=( bonjour coucou salut )
$
$ tabconst[1]=ciao
bash: tabconst : variable en lecture seule
$
```
=> tableau en lecture seule

¤ Pour afficher les valeurs et attributs d'un tableau classique, on utilise la syntaxe :
declare –p ***tab*** **…**

Ex :
```
$ declare -p tabconst
declare -ar tabconst='([0]="bonjour" [1]="coucou" [2]="salut")'
$
```

1.2 Opérations sur un élément de tableau classique

¤ *Valeur d'un élément*

On obtient la valeur d'un élément d'un tableau classique à l'aide de 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]}
monde
$
$ echo ${tabmess[1**2+1]}
le
$
```

=> ou bien : `${tabmess[$((RANDOM%4))]}`

- une variable contenant un entier

Ex :
```
$ a=3
$ echo ${tabmess[a]}
monde
$
```

Exercice 1 : 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
$
```

Lorsqu'un nom de tableau est présent sans indice dans une chaîne de caractères ou une expression, **bash** l'interprète comme élément d'indice **0**.

Ex :
```
$ echo $tabmess
bonjour
$
```

Réciproquement, une variable non préalablement définie comme tableau peut être interprétée comme un tableau classique.

Ex :
```
$ var=bonjour
$ echo ${var[0]}
bonjour
$ var=( coucou "${var[0]}" )
$ echo ${var[1]}
bonjour
$
```

=> *var* est interprétée comme un tableau à un seul élément
=> d'indice 0

=> *var* est devenu un véritable tableau

¤ *Longueur d'un élément*

Pour obtenir la longueur d'un élément d'un tableau classique : **${#*nomtab*[*indice*]}**

Ex :
```
$ echo ${tabmess[0]}
bonjour
$
$ echo ${#tabmess[0]}
7
$
```

=> longueur de la chaîne *bonjour*

¤ *Suppression d'un élément*

Suppression d'un ou plusieurs éléments d'un tableau classique : **unset** *nomtab*[*indice*] **...**

¤ *Autres opérations*

Les différents traitements sur les chaînes de caractères vus précédemment [cf. *Chapitre 9, Chaînes de caractères § 3, 4, 5, 6*] sont également applicables aux éléments d'un tableau classique.

Suppression d'une sous-chaîne :

${***nomtab*[*indice*]**#***modèle***}** **${*nomtab*[*indice*]**##***modèle***}**
${***nomtab*[*indice*]**%***modèle***}** **${*nomtab*[*indice*]**%%***modèle***}**

Extraction d'une sous-chaîne :

${***nomtab*[*indice*]**:ind}** **${*nomtab*[*indice*]**:ind:nb}**

Substitution de sous-chaînes :

${***nomtab*[*indice*]**/mod/ch}** **${*nomtab*[*indice*]**//mod/ch}**

Transformation majuscules/minuscules :

${***nomtab*[*indice*]**^^}** **${*nomtab*[*indice*]**,,}**

Ex :
```
$ tpers=("Jean Fontaine:20:1,72"  "Pierre Cascade:18:1,83")
$
$ echo ${tpers[1]#*:}
18:1,83
$
$ echo ${tpers[0]^^}
JEAN FONTAINE:20:1,72
$
```

1.3 Opérations sur un tableau classique

¤ *Nombre d'éléments d'un tableau classique* : **${#***nomtab***[@]}**

Parmi les différentes notations possibles pour connaître le nombre d'éléments d'un tableau classique, on utilisera celle-ci : **${#***nomtab***[@]}**

Il est à noter que seuls les éléments initialisés sont comptés.

Ex :
```
$ arr=([1]=coucou  [3]=hello)
$ echo ${#arr[@]}
2
$
```

¤ *Accès à tous les éléments d'un tableau classique* : **"${***nomtab***[@]}"**

Comme pour **"$@"** et **"$*"** [cf. *Chapitre 2, Substitution de paramètres § 2.5*], les syntaxes **"${***nomtab***[*]}"** et **"${***nomtab***[@]}"** n'ont pas la même signification pour le shell :

- **"${***nomtab***[*]}"** sera remplacée par l'ensemble des éléments du tableau classique *nomtab*, concaténés en une seule chaîne et séparés par le premier caractère de **IFS** :

"*val0 val1 ...* "

- "${*nomtab*[@]}" sera remplacée par autant de chaînes qu'il y a d'éléments :
"*val0*" "*val1*" ...

Ex :
```
$ tabPays=("Etats Unis"  France)
$
$ set "${tabPays[*]}"
$ echo $#
1                    => une seule chaîne
$
$ echo $1
Etats Unis France
$
$ set "${tabPays[@]}"
$ echo $#
2                    => deux chaînes
$
$ echo $1
Etats Unis
$ echo $2
France
$
```

Lorsque les éléments sont de simples chaînes constituées d'un seul mot, il n'y a aucune difficulté pour préserver l'intégrité de celles-ci. Cela est différent avec des chaînes composées.

Ex :
```
$ for p in ${tabPays[*]}
> do
>   echo $p
> done
Etats                    => l'intégrité du premier élément est rompue
Unis
France
$
```

Pour préserver l'intégrité de chaque élément, il est préférable d'utiliser la syntaxe :
"${*nomtab*[@]}".

Ex :
```
$ for p in "${tabPays[@]}"
> do
>   echo $p
> done
Etats Unis               => l'intégrité du premier élément est préservée
France
$
```

¤ *Liste de tous les indices définis* : "${!*nomtab*[@]}"

Il existe également plusieurs syntaxes pour obtenir la liste de tous les indices conduisant à un élément défini d'un tableau classique. Afin d'utiliser une syntaxe commune avec les tableaux associatifs, on utilisera la notation : "${!*nomtab*[@]}"

Ex :
```
$ arr=([1]=coucou  bonjour [5]=hello)
$
$ echo "${!arr[@]}"
1 2 5                    => pour affecter un indice à bonjour, bash a incrémenté l'indice
```

```
$                    =>    de l'élément précédent
```

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
$
```

¤ *Copie d'un tableau classique*

Pour copier un tableau classique *tab* : *tabcopie*=("${*tab*[@]}")

Ex :
```
$ tabcopy=( "${tabPays[@]}" )
$
$ echo "${tabcopy[@]}"
Etats Unis France
$
```

¤ *Ajout d'éléments à un tableau classique*

Pour ajouter un ou plusieurs éléments à un tableau classique *tab* :
- à la fin : *tab*+=(*val1 val2* **…**)
- en tête : *tab*=(*val0 val1* **…** "${*tab*[@]}")

Ex :
```
$ tab=("un 1"  "deux 2"  "trois 3")
$
$ tab+=("quatre 4" fin)          => ajout de plusieurs éléments en fin de tableau
$
$ echo "${tab[@]}"
un 1 deux 2 trois 3 quatre 4 fin
$
$ tab=(debut "zero 0" "${tab[@]}")          => ajout de plusieurs éléments en
$                                           =>    tête de tableau
$ echo ${#tab[@]}
7                          => nombre d'éléments du tableau
$ echo "${tab[@]}"
debut zero 0 un 1 deux 2 trois 3 quatre 4 fin
$
$ echo ${tab[5]}
quatre 4
$
```

¤ *Suppression d'un tableau classique*

Pour supprimer un ou plusieurs tableaux classiques : **unset** *nomtab* **…**

Ex :
```
$ unset tabcopy  tab
$
```

Exercice 2 : 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 3 : Ecrire un programme shell ***tabnoms*** qui place dans un tableau les noms de tous les utilisateurs enregistrés dans le fichier **/etc/passwd**, affiche le nombre total d'utilisateurs enregistrés, puis tire au hasard un nom d'utilisateur.

2. Tableaux associatifs

2.1 Définition et initialisation

Dans un *tableau associatif*, les index permettant d'accéder aux éléments du tableau ne sont plus des entiers positifs ou nuls mais des chaînes de caractères appelées *clés*.

¤ Pour créer un ou plusieurs tableaux associatifs vides, on utilise la syntaxe :

declare -A *nomtabA* **...**

¤ Pour définir et initialiser un tableau associatif, il est possible d'utiliser les syntaxes suivantes :

declare -A *nomtabA*=([*clé*]=*val* **...**)

ou bien

declare -A *nomtabA*
nomtabA=([*clé*]=*val* **...**)

Contrairement aux tableaux classiques, l'utilisation de la syntaxe **declare -A** est impérative pour créer un tableau associatif.

Ex :
```
$ declare -A tabCapA=( [fr]=Paris  [it]=Rome )
$
$ declare -A tabMonA
$ tabMonA=( [France]=euro  ["Etats Unis"]="dollar US" )
$
```

¤ Pour lister les tableaux associatifs définis : **declare -A**

Ex :
```
$ declare -A
declare -A BASH_ALIASES='()'
declare -A BASH_CMDS='()'
declare -A tabCapA='([fr]="Paris" [it]="Rome" )'
declare -A tabMonA='([France]="euro" ["Etats Unis"]="dollar US" )'
$
```

¤ Pour désigner un élément d'un tableau associatif : *nomtabA*[*clé*]

Ex :
```
$ declare -A tailleA
$ read tailleA[Pierre] tailleA[Jean]
1,83 1,72
$
```

¤ Pour créer un tableau associatif en lecture seule, on utilise les options **–rA** :

Ex :
```
$ declare -rA tconstA=( [Aline]=22  [Marie]=43 )   => tableau en lecture seule
$
$ tconstA[Aline]=23                 => modification d'une valeur
bash: tconstA : variable en lecture seule
$
$ tconstA[Julie]=34                 => ajout d'un élément
bash: tconstA : variable en lecture seule
$
```

¤ Pour afficher les valeurs et attributs d'un tableau associatif, on utilise la syntaxe usuelle :
declare –p *nomtabA* **…**

Ex :
```
$ declare -p tailleA
declare -A tailleA='([Jean]="1,72" [Pierre]="1,83" )'
$
```

2.2 Opérations sur un élément de tableau associatif

¤ *Valeur d'un élément*

Pour obtenir la valeur d'un élément d'un tableau associatif, on utilise la syntaxe :
${*nomtabA***[***clé***]}**

Ex :
```
$ echo ${tailleA[Jean]}
1,72
$
```

¤ *Longueur d'un élément*

Pour obtenir la longueur d'un élément d'un tableau associatif : **${#***nomtabA***[***clé***]}**

Ex :
```
$ echo ${#tailleA[Pierre]}
4                   => longueur de la chaîne 1,83
$
```

¤ *Suppression d'un élément*

Suppression d'un ou plusieurs éléments d'un tableau associatif : **unset** *nomtabA***[***clé***]** **…**

¤ *Autres opérations*

Les traitements sur les éléments d'un tableau classique sont également applicables aux éléments d'un tableau associatif.

Ex :
```
$ declare -A tPaysA=([France]="Paris:euro:fr" ["Etats Unis"]="Washington,
D.C.:dollar US:us")
$
$ echo ${tPaysA["Etats Unis"]/dollar US/USD}
Washington, D.C.:USD:us
$
```

2.3 Opérations sur un tableau associatf

¤ *Nombre d'éléments d'un tableau associatif* : **${#*nomtabA*[@]}**

Pour obtenir le *nombre d'éléments* du tableau associatif ***nomtabA***, on utilisera la même syntaxe que pour les tableaux classiques : **${#*nomtabA*[@]}**

Ex :
```
$ declare -A tabPersA=([Pierre]="Canson Bonnefoy:1,73:20" ["Jean Michel "]="Dupont:1,72:18")
$
$ echo ${#tabPersA[@]}
2
$
```

¤ *Accès à tous les éléments d'un tableau associatif* : **"${*nomtabA*[@]}"**

Pour lister *toutes les valeurs* d'un tableau associatif, on utilise la même syntaxe que pour les tableaux classiques : **"${*nomtabA*[@]}"**

Ex :
```
$ for p in "${tabPersA[@]}"
> do
>   echo $p
> done
Canson Bonnefoy:1,73:20          => l'intégrité de la valeur de l'élément est préservée
Dupont:1,72:18
$
```

¤ *Liste de toutes les clés définies* : **"${!*nomtab*[@]}"**

Il est déconseillé d'utiliser la syntaxe **${!nomtabA[@]}** pour lister *toutes les clés* d'un tableau associatif, car les éléments sont accédés via des clés qui sont elles même des chaînes de caractères pouvant être composées de plusieurs mots.

Ex :
```
$ for cl in ${!tabPersA[@]}
> do
>   echo "$cl => ${tabPersA[$cl]}"
> done
Pierre => Canson Bonnefoy:1,73:20
Jean =>                                  => l'intégrité de la clé n'a pas été préservée
Michel =>
$
```

Par conséquent, pour lister *toutes les clés* d'un tableau associatif, on préfèrera utiliser la syntaxe : **"${!*nomtab*[@]}"**

Ex :
```
$ for cl in "${!tabPersA[@]}"
> do
>   echo "$cl => ${tabPersA[$cl]}"
> done
Pierre => Canson Bonnefoy:1,73:20
Jean Michel => Dupont:1,72:18            => l'intégrité de la clé est préservée
$
```

Remarque : contrairement aux tableaux classiques, la syntaxe *${tabPersA[**cl**]}* ne peut être utilisée pour un tableau associatif car ***cl*** est une chaîne de caractères. Seule la

notation *${tabPersA[$cl]}* doit être employée.

¤ *Ajout d'éléments à un tableau associatif*

Les notions de début ou de fin d'un tableau associatif ne sont pas pertinentes. De même, il ne peut y avoir de « trous » dans un tableau associatif : aucune relation d'ordre n'est définie sur les éléments d'un tableau associatif.

Pour ajouter un ou plusieurs éléments à un tableau associatif *tabA*, on peut utiliser la syntaxe :
tabA+=([*clé*]=*val* **…**)

Ex :
```
$ declare -A tabCapA=( [fr]=Paris  [it]=Rome )
$
$ tabCapA+=(  [es]=Madrid  [de]=Berlin)
$
```

¤ *Copie d'un tableau associatif*

Pour copier un tableau associatif *tabA* dans un autre *cpA*, la méthode la plus simple consiste à copier itérativement élément par élément.

Ex :
```
$ declare -A tabA=( [jean louis]="dupond latour"  [yves andre michel]="de
la rosette")
$
$ declare -A cpA                => création du nouveau tableau associatif cpA
$ for cl in "${!tabA[@]}"
> do
>   cpA+=( [$cl]="${tabA[$cl]}" )          => ajout dans le nouveau tableau
> done
$
$ declare -p tabA cpA           => affichage
declare -A tabA='(["jean louis"]="dupond latour" ["yves andre
michel"]="de la rosette" )'
declare -A cpA='(["jean louis"]="dupond latour" ["yves andre michel"]="de
la rosette" )'
$
```

¤ *Suppression d'un tableau associatif*

Pour supprimer un ou plusieurs tableaux associatifs : **unset** *nomtabA* **…**

Exercice 4 : Ecrire un programme shell ***tabnomsA*** qui crée à partir du fichier **/etc/passwd** un tableau associatif de la manière suivante :
- le nom d'utilisateur est utilisé comme *clé* du tableau associatif
- le restant de la ligne constitue la *valeur* de l'élément. On omettra toutefois d'inclure dans cette valeur, le champ correspondant au mot de passe *:x:*.

Par exemple, la ligne du fichier **/etc/passwd** :
```
sanchis:x:1000:1000:eric sanchis,,,:/home/sanchis:/bin/bash
```
devient :
```
tabnomA[sanchis]=1000:1000:eric sanchis,,,:/home/sanchis:/bin/bash
```

Chapitre 13 : Alias

Un alias permet d'abréger 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. De manière plus générale, un alias est utilisé pour remplacer une longue chaîne de caractères par une chaîne plus courte.

Pour connaître l'ensemble des alias définis, on utilise la commande **alias** sans argument.

Ex :
```
$ alias
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo
terminal || echo error)" "$(history|tail -n1|sed -e '\''s/^\s*[0-
9]\+\s*//;s/[;&|]\s*alert$//'\'')"'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
$
```

Pour connaître la valeur d'un ou plusieurs alias : **alias** *nom* **…**

Ex :
```
$ alias ll
alias ll='ls -alF'
$
```

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 : supprimer fichier vide «err» ? o          => l'alias rm demande
$                                              =>  confirmation
$ ls err      => le fichier err a été supprimé
ls: impossible d'accéder à err: Aucun fichier ou dossier de ce type
$
```

Si l'on souhaite temporairement obtenir le fonctionnement originel de la commande unix **rm**, il suffit de placer le caractère **** devant l'alias.

Ex :
```
$ > err              => création du fichier err
$
$ \rm err            => fonctionnement standard de rm
$
$ ls err
ls: impossible d'accéder à err: Aucun fichier ou dossier de ce type
$
```

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
Commande « aff » non trouvée, vouliez-vous dire :
 La commande « apf » issue du paquet « apf-firewall » (universe)
 La commande « caff » issue du paquet « signing-party » (universe)
 La commande « aft » issue du paquet « aft » (universe)
aff : commande introuvable     => 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
wheezy/sid
$
$ 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
wheezy/sid
$
```

La valeur d'un alias peut inclure plusieurs commandes.

Ex :
```
$ pwd
/tmp                          => répertoire courant
$ alias scd='echo  salut ; cd '      => après cd il y a un caractère espace
$ 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
lundi 24 février 2014, 17:15:50 (UTC+0100)
$
$ alias h='set $(date) ; echo ${5%:*}'
$
$ h
17:19
$
$ sleep 60
      ...          => attente de 60 secondes
$ h
17:20              => 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: Erreur de syntaxe près du symbole inattendu « ( »
$
```

C'est la présence de la **parenthèse ouvrante** de la chaîne de caractères :

set lundi 24 février 2014, 17:15:50 (UTC+0100)

Cette chaîne est exécutée comme une commande par le shell, ce qui provoque l'erreur.

2. Suppression d'un alias

Pour rendre indéfinie la valeur d'un ou plusieurs alias, on utilise la commande interne **unalias**.

La syntaxe est : **unalias** *nom* **...**

Ex :
```
$ unalias g alert rep aff scd
$
$ alias
alias c='cat '
alias cx='chmod u+x'
alias d='/etc/debian_version'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias h='set $(date) ; echo ${5%:*}'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
alias rm='rm -i'
$
```

3. Utilisation des alias

Même s'il est possible d'utiliser des alias dans un fichier shell, il est conseillé de ne les utiliser qu'en session interactive. En effet, leur utilisation dans un programme shell présente peu d'intérêt :

- on ne peut leur fournir des arguments
- lorsqu'ils sont utilisés comme raccourcis de commandes ou de chaînes de caractères, ils obscurcissent la lisibilité du code.

Dans un fichier shell, il est hautement préférable d'utiliser des fonctions plutôt que des alias.

Chapitre 14 : Fonctions shell

1. Définition d'une fonction

Pour définir une fonction, le shell **bash** propose plusieurs syntaxes telles que :

```
nom_fct()                    ou bien    function nom_fct
{                                       {
 suite_de_commandes                      suite_de_commandes
}                                       }
```

Pour des raisons de lisibilité, nous utiliserons la deuxième forme.

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, s*uite_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 : commande introuvable
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
$
```

2. Suppression d'une fonction

Une fonction est rendue indéfinie par la commande : **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 !
        ...
$
```

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[*]}"    # (c)
}

function f1
{
echo " --- Dans f1 :"
echo " --- FUNCNAME : $FUNCNAME"
echo " --- tableau FUNCNAME[] : ${FUNCNAME[*]}"       # (b)
echo " --- - Appel a f2 "
echo

f2
}

echo "Debut :"
echo "FUNCNAME : $FUNCNAME"
echo "tableau FUNCNAME[] : ${FUNCNAME[*]}"            # (a)
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)
$
```

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"
}


echo "Avant f : \$0 : $0"
echo "Avant f : \$# : $#"
echo "Avant f : \$1 : $1"
f  pierre  paul jacques
echo "Apres f : \$1 : $1"
```

=> affichage du 1er argument de la fonction *f*

=> affichage du 1er argument du programme *args*

=> 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
Apres f : $1 : un
$
```

=> nom du programme

=> nombre d'arguments passés à *args*

=> 1er argument passé à *args*

=> 1er argument passé à la fonction *f*

=> les arguments passés à *args* ont été restitués

Utilisée dans le corps d'une fonction, la commande interne **shift** décale la numérotation des paramètres de position internes à la fonction.

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)
      echo " -- Dans f : Apres execution de set \$(date) : \$* : $*"
      }

      echo "Avant f : $*"
      echo Appel : f alpha beta
      f alpha beta
      echo "Apres f : $*"
-----------------------------------
```

Ex :
```
      $ argsSet un
      Avant f : un
      Appel : f alpha beta
      -- Dans f : Avant execution de set $(date) : $* : alpha beta
      -- Dans f : Apres execution de set $(date) : $* : vendredi 18 avril
      2014, 16:04:50 (UTC+0200)
      Apres f : un
      $
```
=> l'argument a été convenablement sauvegardé puis restitué

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(s)*] [*nom*[=*valeur*] **...**]

Les options utilisables avec **local** sont celles de la commande interne **declare**. Par conséquent, on définira une ou plusieurs variables locales de type entier avec la syntaxe **local –i** (mais aussi **local –a** pour un tableau classique, **local –A** pour un tableau associatif).

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 (modification)"
}


function f2
{
echo "f2 : $((x+10))"           => utilisation de x définie dans la fonction f1
f3                              => appel de f3
}


function f1
{
      local -i x                => définition de x

x=2                             => initialisation de x
echo "f1 : x=$x (initialisation)"
f2                              => appel de f2
echo "f1 : Valeur finale : x=$x"
}


f1                              => appel de f1
-----------------------------------
```

Ex :
```
$ appelsCascade
f1 : x=2 (initialisation)
f2 : 12
f3 : x=-2 (modification)
f1 : Valeur finale : x=-2
$
```

La fonction *f3* affecte la valeur *-2* à la variable *x* précédemment initialisée à la valeur *2* dans la fonction *f1*. A la fin de l'exécution de *f3*, la valeur de *x* reste égale à *-2* dans la fonction initiale *f1*.

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 : commande introuvable
$
```

Pour que la fonction *bonj* puisse être exécutée par *autreProgShell*, 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*. Après avoir utilisée la fonction *bonj* (*bonj Madame*), il exécute le programme *autreProg1*.
Le programme *autreProg1* utilise la fonction *bonj* (*bonj Monsieur*) puis exécute le programme *autreProg2*.
Le programme *autreProg2* utilise la fonction *bonj* (*bonj Mademoiselle*) puis se termine.

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
$
```

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 par la fonction.

Dans le programme shell *return0*, la fonction *f* retourne le code de retour *5* au corps du programme shell.

retour0

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

      function f
      {
      echo coucou
      return 5
      echo a demain    # jamais execute
      }

      f
      echo code de retour de f : $?
-----------------------------------
```

Ex :
```
$ return0
coucou
code de retour de f : 5
$
```

Remarque : il ne faut pas confondre **return** et **exit**. Cette dernière arrête l'exécution du programme shell qui la contient.

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
$
```

9. Fonctions récursives

Comme pour les programmes shell, **bash** permet l'écriture de fonctions récursives.

Le programme *fctfactr* 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.

fctfactr

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


shopt -s extglob

if (( $# != 1 )) || [[ $1 != +([0-9]) ]]
 then
   echo "syntaxe : fctfactr n" >&2
   exit 1
fi



function f
{
        local -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 euzte12uz
syntaxe : fctfactr n
$ fctfactr 1
1
$ fctfactr 4
24
$
```

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
$
```