Scripts sous Linux

Shell Bash, Sed, Awk, Perl, Tcl, Tk, Python, Ruby...

Christophe Blaess

2e édition

ISBN : 2-212-11405-2

EYROLLES

2

Évaluation d'expressions avec Bash

Une part importante de la programmation de scripts repose sur une bonne compréhension des mécanismes qui permettent au shell d'évaluer correctement les expressions reçues. Nous allons donc commencer par étudier l'utilisation des variables, avant d'observer en détail les évaluations d'expressions. Une fois que cela aura été effectué, l'étude des structures de contrôle et des fonctions internes du shell sera plus facile, et nous serons à même de réaliser de véritables scripts.

Variables

La programmation sous shell nécessite naturellement des variables, pour stocker des informations temporaires, accéder à des paramètres, etc. Par défaut, les variables utilisées dans les scripts shell ne sont pas typées. Le contenu d'une variable est considéré comme une chaîne de caractères, sauf si on indique explicitement qu'elle doit être traitée comme une variable entière, qui peut être utilisée dans des calculs arithmétiques. De plus, le shell ne permet pas de manipuler directement de données en virgules flottantes.

> Nous verrons plus avant comment il convient d'employer l'utilitaire `bc` au sein de scripts shell pour réaliser des opérations sur des nombres réels.

À la différence des langages compilés habituels, une variable n'a pas à être déclarée explicitement. Dès qu'on lui affecte une valeur, elle commence à exister. Cette affectation

prend la forme `variable=valeur` sans espaces autour du signe égal. Le message d'erreur '*i: command not found*' est peut-être le plus connu des utilisateurs du shell :

```
$ i = 1
bash: i: command not found
$
```

En raison des espaces autour du signe égal, Bash a cru que l'on essayait d'invoquer la commande '`i`' en lui transmettant les arguments '`=`' et '`1`'. La bonne syntaxe est la suivante :

```
$ i=1
$
```

Pour accéder au contenu d'une variable, il suffit de préfixer son nom avec le caractère `$`. Il ne faut pas confondre ce préfixe des variables avec le symbole d'invite du shell, qui est généralement le même caractère `$`. La commande `echo` affiche simplement le contenu de sa ligne de commande :

```
$ echo $i
1
$
```

Le nom attribué à une variable peut contenir des lettres, des chiffres, ou le caractère souligné '_'. Il ne doit toutefois pas commencer par un chiffre. Voyons quelques exemples.

```
$ variable=12
$ echo $variable
12
$
```

Il faut comprendre que le shell a remplacé la chaîne `$variable` par sa valeur, 12, avant d'appeler la commande `echo`. Cette dernière a donc été invoquée par la ligne de commande `echo 12`.

```
$ variable=abc def
bash: def: command not found
$
```

Ici, le shell n'a pas pu interpréter correctement cette ligne, car il a cru qu'elle se composait d'une affectation `variable=abc`, suivie d'une commande nommée `def` (syntaxe rarement utilisée, mais autorisée). Il faut lui indiquer que les mots à droite du signe égal forment une seule chaîne de caractères. On emploie pour cela les guillemets droits :

```
$ variable="abc def"
$ echo $variable
abc def
$
```

Nous pouvons vérifier qu'une variable qui n'a jamais été affectée est considérée comme une chaîne vide :

```
$ echo $inexistante

$
```

Une variable à laquelle on affecte une chaîne vide existe quand même. La différence entre une variable inexistante et une variable vide peut être mise en évidence à l'aide de certaines options des constructions de test, ou par l'intermédiaire d'une configuration particulière de Bash qui déclenchera une erreur si on essaie de lire le contenu d'une variable inexistante. Cette configuration s'obtient au moyen de la commande interne `set` (que nous détaillerons dans le chapitre 4) et de son option `-u`.

```
$ set -u
$ echo $inexistante
bash: inexistante: unbound variable
$ vide=
$ echo $vide

$
```

Précisions sur l'opérateur $

On consulte le contenu d'une variable à l'aide de l'opérateur `$`. Toutefois, la forme `$variable`, même si elle est la plus courante, est loin d'être la seule, et l'opérateur `$` propose des fonctionnalités d'une richesse surprenante.

Délimitation du nom de variable

Tout d'abord, la construction `${variable}` est une généralisation de `$variable`, qui permet de délimiter précisément le nom de la variable, dans le cas où on souhaiterait le coller à un autre élément. Par exemple, supposons que nous souhaitions classer les fichiers source d'une bibliothèque de fonctions en leur ajoutant un préfixe qui corresponde au projet auquel ils appartiennent. Nous pourrions obtenir une séquence du type :

```
$ PREFIXE=projet1
$ FICHIER=source1.c
$ NOUVEAU_FICHIER=$PREFIXE_$FICHIER
```

On espère obtenir « `projet1_source1.c` » dans la variable `NOUVEAU_FICHIER`, mais malheureusement ce n'est pas le cas :

```
$ echo $NOUVEAU_FICHIER
source1.c
$
```

Le fait d'avoir accolé directement `$PREFIXE`, le caractère souligné et `$FICHIER` ne fonctionne pas comme nous l'attendions : le shell a bien remarqué qu'il y a deux opérateurs `$`

agissant chacun sur un nom de variable, mais seul le second a été correctement remplacé. En effet, le caractère souligné que nous avons ajouté comme séparateur entre le préfixe et le nom du fichier a été considéré comme appartenant au nom de la première variable. Ainsi le shell a-t-il recherché une variable nommé `PREFIXE_` et n'en a évidemment pas trouvé.

La raison en est que le caractère souligné n'est pas un caractère séparateur pour le shell, et qu'on peut le rencontrer dans les noms de variables. Si nous avions utilisé un caractère interdit dans ces noms, nous n'aurions pas eu le même problème car le premier nom de variable aurait été clairement délimité :

```
$ echo $PREFIXE.$FICHIER
projet1.source1.c
$ echo $PREFIXE@$FICHIER
projet1@source1.c
$ echo $PREFIXE-$FICHIER
projet1-source1.c
$
```

On pourrait même utiliser les guillemets droits pour encadrer le caractère souligné afin de le séparer du nom de la variable. Ce mécanisme sera détaillé plus avant, lorsque nous étudierons les méthodes de protection des caractères spéciaux.

```
$ echo $PREFIXE"_"$FICHIER
projet1_source1.c
$
```

Quoi qu'il en soit, il arrive que l'on doive ajouter des caractères à la fin d'un nom de variable, et l'opérateur `${ }` est alors utilisé à la place du simple `$` pour marquer les limites. Ainsi, on peut utiliser :

```
$ echo ${PREFIXE}_$FICHIER
projet1_source1.c
$
```

ou des constructions comme :

```
$ singulier=mot
$ pluriel=${singulier}s
$ echo $pluriel
mots
$
```

Extraction de sous-chaîne et recherche de motifs

Le shell offre une possibilité d'extraction automatique de sous-chaîne de caractères au sein d'une variable. La version la plus simple est apparue dans Bash 2, et s'appuie sur l'option ':' de l'opérateur `${}`. Ainsi l'expression `${`*variable*`:`*debut*`:`*longueur*`}` est-elle automatiquement remplacée par la sous-chaîne qui commence à l'emplacement indiqué en seconde position, et qui contient le nombre de caractères indiqué en dernière position.

La numérotation des caractères commence à zéro. Si la longueur n'est pas mentionnée, on extrait la sous-chaîne qui s'étend jusqu'à la fin de la variable. En voici quelques exemples :

```
$ variable=ABCDEFGHIJKLMNOPQRSTUVWXYZ
$ echo ${variable:5:2}
FG
$ echo ${variable:20}
UVWXYZ
$
```

> Attention, cette extraction de sous-chaîne existe également sous shell Korn, mais n'est pas décrite dans les spécifications *Single Unix version 3*, et n'est pas implémentée par exemple dans le shell Zsh. On lui accordera donc une confiance limitée en ce qui concerne sa portabilité.

Ce mécanisme fonctionne sans surprise, mais n'est pas aussi utile, dans le cadre de la programmation shell, que l'on pourrait le croire au premier abord. Dans nos scripts, nous manipulons en effet souvent des noms de fichiers ou des adresses réseau, et une autre possibilité d'extraction de sous-chaîne se révèle en général plus adaptée : elle repose sur l'emploi de **motifs** qui sont construits de la même manière que lors des recherches de noms de fichiers en ligne de commande. Ces motifs peuvent contenir des caractères génériques particuliers :

- Le caractère `*` correspond à n'importe quelle chaîne de caractères (éventuellement vide).
- Le caractère `?` correspond à n'importe quel caractère.
- Le caractère `\` permet de désactiver l'interprétation particulière du caractère suivant. Ainsi, la séquence `\*` correspond à l'astérisque, `\?` au point d'interrogation et `\\` au caractère *backslash* (barre oblique inverse).
- Les crochets `[` et `]` encadrant une liste de caractères représentent n'importe quel caractère contenu dans cette liste. La liste peut contenir un intervalle indiqué par un tiret comme `A-Z`. Si l'on veut inclure les caractères `-` ou `]` dans la liste, il faut les placer en première ou dernière position. Les caractères `^` ou `!` en tête de liste indiquent que la correspondance se fait avec n'importe quel caractère qui n'appartient pas à l'ensemble.

L'extraction de motifs peut se faire en début, en fin, ou au sein d'une variable. Il s'agit notamment d'une technique très précieuse pour manipuler les préfixes ou les suffixes des noms de fichiers.

L'expression `${variable#motif}` est remplacée par la valeur de la variable, de laquelle on ôte la chaîne initiale la plus courte qui corresponde au motif :

```
$ variable=AZERTYUIOPAZERTYUIOP
$ echo ${variable#AZE}
RTYUIOPAZERTYUIOP
```

La portion initiale AZE a été supprimée.

```
$ echo ${variable#*T}
YUIOPAZERTYUIOP
```

Tout a été supprimé jusqu'au premier T.

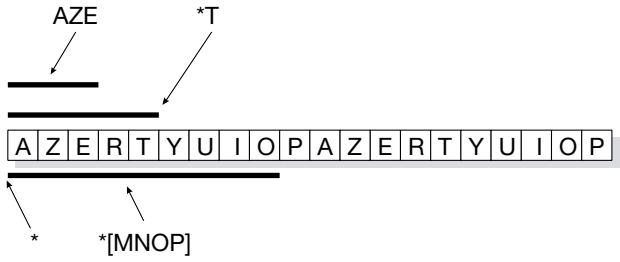
```
$ echo ${variable#*[MNOP]}
PAZERTYUIOP
```

Élimination du préfixe jusqu'à la première lettre contenue dans l'intervalle.

```
$ echo ${variable#*}
AZERTYUIOPAZERTYUIOP
```

Suppression de la plus petite chaîne quelconque, en l'occurrence la chaîne vide, et il n'y a donc pas d'élimination de préfixe.

Figure 2–1
Actions de l'opérateur ${…#…}



L'expression `${variable##motif}` sert à éliminer le plus long préfixe correspondant au motif transmis. Ainsi, les exemples précédents nous donnent :

```
$ echo ${variable##AZE}
RTYUIOPAZERTYUIOP
```

La chaîne AZE ne peut correspondre qu'aux trois première lettres ; pas de changement par rapport à l'opérateur précédent.

```
$ echo ${variable##*T}
YUIOP
```

Cette fois-ci, le motif absorbe le plus long préfixe qui se termine par un T.

```
$ echo ${variable##*[MNOP]}

```

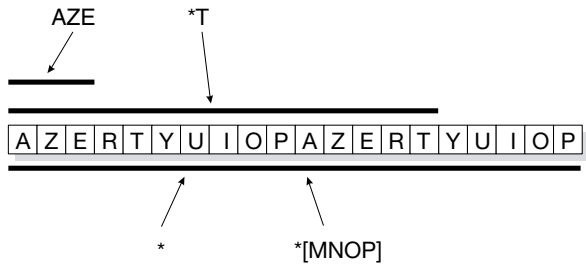
Le plus long préfixe se terminant par M, N, O ou P correspond à la chaîne elle-même, puisqu'elle se finit par P. L'expression renvoie donc une chaîne vide.

```
$ echo ${variable##*}

```

De même, la suppression de la plus longue chaîne initiale, composée de caractères quelconques, renvoie une chaîne vide.

Figure 2–2
Actions de l'opérateur ${…##…}



Symétriquement, les expressions `${variable%motif}` et `${variable%%motif}` correspondent au contenu de la variable indiquée, qui est débarrassé, respectivement, du plus court et du plus long suffixe correspondant au motif transmis. En voici quelques exemples :

```
$ variable=AZERTYUIOPAZERTYUIOP
$ echo ${variable%IOP*}
AZERTYUIOPAZERTYU
$ echo ${variable%%IOP*}
AZERTYU
$ echo ${variable%[X-Z]*}
AZERTYUIOPAZERT
$ echo ${variable%%[X-Z]*}
A
$ echo ${variable%*}
AZERTYUIOPAZERTYUIOP
$ echo ${variable%%*}

$
```

Figure 2–3
Actions de l'opérateur ${…%…}

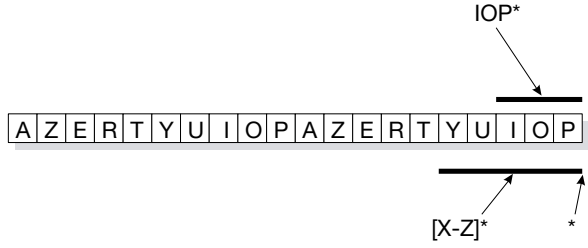
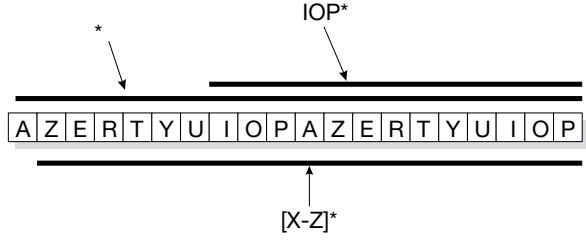


Figure 2–4
Actions de l'opérateur ${…%%…}



Un opérateur est apparu dans Bash 2, qui permet de remplacer une portion de chaîne grâce aux expressions

- `${variable/motif/remplacement}`, qui permet de remplacer la première occurrence du motif par la chaîne fournie en troisième position ;
- `${variable//motif/remplacement}`qui remplace toutes les occurrences du motif.

Cet opérateur remplace la plus longue sous-chaîne possible correspondant au motif.

> Cet opérateur est présent dans les shells Bash, Korn 93 et Zsh, mais n'est pas normalisé par *Single Unix version 3*. De légères divergences sont donc susceptibles d'apparaître entre les différentes implémentations.

Voyons quelques utilisations classiques de ces opérateurs :

Afficher le répertoire de travail en cours en remplaçant le répertoire personnel par le symbole ~ : `${PWD/$HOME/~}`

```
$ cd /etc/X11
$ echo ${PWD/$HOME/~}
/etc/X11
$ cd
$ echo ${PWD/$HOME/~}
~
$ cd Doc/ScriptLinux/
$ echo ${PWD/$HOME/~}
~/Doc/ScriptLinux
$
```

Retrouver le nom de login dans une adresse e-mail : `${adresse%%@*}`

```
$ adresse=utilisateur@machine.org
$ echo ${adresse%%@*}
utilisateur
$ adresse=utilisateur
$ echo ${adresse%%@*}
utilisateur
$
```

Récupérer le nom d'hôte dans une adresse complète de machine : `${adresse%%.*}`

```
$ adresse=machine.entreprise.com
$ echo ${adresse%%.*}
machine
$ adresse=machine
$ echo ${adresse%%.*}
machine
$
```

Obtenir le nom d'un fichier débarrassé de son chemin d'accès : `${fichier##*/}`

```
$ fichier=/usr/src/linux/kernel/sys.c
$ echo ${fichier##*/}
```

```
sys.c
$ fichier=programme.sh
$ echo ${fichier##*/}
programme.sh
$
```

Éliminer l'extension éventuelle d'un nom de fichier : `${fichier%.*}`

```
$ fichier=module1.c
$ echo ${fichier%.*}
module1
$ fichier=projet.module.h
$ echo ${fichier%.*}
projet.module
$ fichier=module
$ echo ${fichier%.*}
module
$
```

Renommer tous les fichiers dont l'extension est `.TGZ` qui se trouvent dans le répertoire courant en fichiers `.tar.gz` :

```
$ ls *.TGZ
a.TGZ b.TGZ
$ for i in *.TGZ ; do mv $i ${i%TGZ}tar.gz ; done
$ ls *.tar.gz
a.tar.gz b.tar.gz
$
```

Il est également possible d'enchaîner les opérateurs `${…#…}`, `${…##…}`, `${…%…}` et `${…%%…}` pour accéder à divers constituants d'une chaîne de caractères. Par exemple, le script suivant va recevoir sur son entrée standard un article Usenet, et extraire la liste des serveurs NNTP par lesquels il est passé. Pour ce faire, il recherche une ligne qui soit constituée ainsi :

```
Path: serveur_final!precedent!antepenultieme!deuxieme!premier!not-for-mail
```

Chaque serveur ajoute son identité en tête de cette chaîne.

> Par convention, la chaîne `Path` se termine par un pseudo-serveur « `not-for-mail` ». Nous ne nous préoccupons pas de ce détail, et le considérerons comme un serveur normal.

Voici le script qui extrait automatiquement la liste des serveurs.

extraction_serveurs.sh :

```
1    #! /bin/sh
2
3    ligne_path=$(grep "Path: ")
4    liste_serveurs=${ligne_path##Path: }
5    while [ -n "$liste_serveurs" ] ; do
```

```
 6      serveur=${liste_serveurs%%!*}
 7      liste_serveurs=${liste_serveurs#$serveur}
 8      liste_serveurs=${liste_serveurs#!}
 9      echo $serveur
10    done
```

La ligne 3 remplit la variable avec le résultat de la commande `grep`, qui recherche la ligne « `Path:` » sur son entrée standard ; nous verrons plus en détail cette syntaxe très prochainement. Cette commande donne le résultat suivant lorsqu'on l'applique à un message quelconque :

```
$ grep "Path: " message.news
Path: club-internet!grolier!brainstorm.fr!frmug.org!oleane!
news-raspail.gip.net!news-stkh.gip.net!news.gsl.net!gip.net
!masternews.telia.net!news.algonet.se!newsfeed1.telenordia.
se!algonet!newsfeed1.funet.fi!news.helsinki.fi!not-for-mail
$
```

La ligne 4 élimine le préfixe « `Path:` » dans cette ligne. La boucle `while` qui s'étend de la ligne 5 à la ligne 10 se déroule tant que la variable `liste_serveurs` n'est pas une chaîne de longueur nulle.

Sur la ligne 6, nous extrayons le premier serveur, c'est-à-dire que nous supprimons le plus long suffixe commençant par un point d'exclamation. La ligne 7 nous permet de retirer ce serveur de la liste, puis la suivante retire l'éventuel point d'exclamation en tête de liste (qui sera absent lors de la dernière itération).

L'exécution de ce script donne le résultat suivant :

```
$ ./extraction_serveurs.sh < message.news
club-internet
grolier
brainstorm.fr
frmug.org
oleane
news-raspail.gip.net
news-stkh.gip.net
news.gsl.net
gip.net
masternews.telia.net
news.algonet.se
newsfeed1.telenordia.se
algonet
newsfeed1.funet.fi
news.helsinki.fi
not-for-mail
$
```

Les modificateurs `#`, `##`, `%` et `%%` de l'opérateur `$` sont, comme nous pouvons le constater, très puissants ; ils donnent accès à de sérieuses possibilités de manipulation des chaînes de caractères. Cela concerne l'extraction des champs contenus dans les lignes d'un fichier (`/etc/passwd` par exemple), l'automatisation d'analyse de courriers électroniques

ou de news Usenet, ou le dépouillement de journaux de statistiques. Nous verrons que d'autres langages comme Awk ou Perl sont plus adaptés pour certaines de ces tâches, mais il convient quand même de ne pas négliger les possibilités des scripts Bash.

Longueur de chaîne

L'opérateur `$` offre aussi la possibilité de calculer automatiquement la longueur de la chaîne de caractères représentant le contenu d'une variable, grâce à sa forme `${#variable}`.

```
$ variable=azertyuiop
$ echo ${#variable}
10
$
```

La commande `grep` utilisée précédemment renvoyait une longue chaîne contenant tous les serveurs NNTP :

```
$ variable=$(grep Path message.news )
$ echo ${#variable}
236
$
```

Si la variable est numérique, sa longueur correspond à celle de la chaîne de caractères qui la représente. Cela est également vrai pour les variables qui sont explicitement déclarées arithmétiques. Par exemple, la variable `EUID` est une variable arithmétique définie automatiquement par le shell.

```
$ echo $EUID
500
$ echo ${#EUID}
3
$
```

Les variables vides ou non définies ont des longueurs nulles.

```
$ variable=
$ echo ${#variable}
0
$ echo ${#inexistante}
0
$
```

Actions par défaut

Lorsqu'un script fonde son comportement sur des variables qui sont fournies par l'utilisateur, il est important de pouvoir configurer une action par défaut, dans le cas où la variable n'aurait pas été remplie. L'opérateur `${ }` vient encore à notre aide avec l'appui de quatre modificateurs.

L'expression `${variable:-valeur}` prend la valeur indiquée à droite du modificateur `:-`, si la variable n'existe pas, ou si elle est vide. Cela nous permet de fournir une valeur par

défaut. Naturellement, si la variable existe et est non vide, l'expression renvoie son contenu :

```
$ variable=existante
$ echo ${variable:-defaut}
existante
$ variable=
$ echo ${variable:-defaut}
defaut
$ echo ${inexistante:-defaut}
defaut
$
```

Dans le script `rm_secure` du chapitre 1, nous avons fixé systématiquement en début de programme la variable `sauvegarde_rm=~/.rm_saved/`. Si le script se trouve dans un répertoire système (`/usr/local/bin`), l'utilisateur ne peut pas le modifier. Il peut toutefois préférer employer un autre répertoire pour la sauvegarde. Cela peut être réalisé en l'autorisant à fixer le contenu d'une variable d'environnement – disons `SAUVEGARDE_RM` - avant d'invoquer `rm_secure`. On utilisera alors le contenu de cette variable, sauf si elle est vide, auquel cas on emploiera la chaîne `~/.rm_saved/` par défaut. Ainsi la première ligne du script deviendrait-elle :

```
1           sauvegarde_rm=${SAUVEGARDE_RM:-~/.rm_saved/}
```

L'emploi systématique de valeurs par défaut en début de script permet d'améliorer la robustesse du programme, et de le rendre plus souple, plus facile à configurer par l'utilisateur.

Le contenu même de la variable n'est pas modifié. Toutefois, il est des cas où cela serait préférable. La construction `${variable:=valeur}` peut être utilisée à cet effet. Si la variable n'existe pas ou si elle est vide, elle est alors remplie avec la valeur. Ensuite, dans tous les cas, le contenu de la variable est renvoyé.

```
$ echo $vide

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

Lorsqu'il s'agit simplement de remplir la variable avec une valeur par défaut, le retour de l'expression ne nous intéresse pas. On ne peut toutefois pas l'ignorer purement et simplement car le shell le considérerait comme une commande et déclencherait une erreur :

```
$ ${vide:=contenu}
bash: contenu: command not found
$
```

L'utilisation de la commande `echo`, même redirigée vers `/dev/null`, manque franchement d'élégance. Il est toutefois possible de demander au shell d'ignorer le résultat de l'évaluation

de l'expression grâce à la commande interne deux-points ':' qui signifie « aucune opération ». En reprenant l'exemple `rm_secure`, on peut décider que l'utilisateur peut fixer directement le contenu de la variable `sauvegarde_rm` s'il le souhaite avant d'appeler le script. La première ligne deviendra donc :

```
1          : ${sauvegarde_rm:=~/.rm_saved/}
```

Dans certaines situations, le script peut vraiment avoir besoin d'une valeur qu'il ne peut pas deviner. Si la variable en question n'existe pas, ou si elle est vide, le seul comportement possible est d'abandonner le travail en cours. La construction `${variable:?message}` réalise cette opération d'une manière simple et concise. Si la variable est définie et non vide, sa valeur est renvoyée. Sinon, le shell affiche le message fourni après le point d'interrogation, et abandonne le script ou la fonction en cours. Le message est préfixé du nom de la variable.

Comme précédemment, pour simplement vérifier si la variable est définie ou non, sans utiliser la valeur renvoyée, on peut utiliser la commande ':'. Nous allons insérer le test dans une petite fonction (définie directement sur la ligne de commande) pour vérifier que son exécution s'arrête bien quand le test échoue.

```
$ function ma_fonction
> {
>   : ${ma_variable:?"n est pas définie"}
>   echo ma_variable = $ma_variable
> }
$ ma_fonction
bash: ma_variable: n est pas définie
$ ma_variable=son_contenu
$ ma_fonction
ma_variable = son_contenu
$
```

Si le message n'est pas précisé, Bash en affiche un par défaut :

```
$ echo ${inexistante:?}
bash: inexistante: parameter null or not set
$
```

On dispose d'une dernière possibilité pour vérifier si une variable est définie ou non. Utilisée plus rarement, la construction `${variable:+valeur}` renvoie la valeur fournie à droite du symbole :+ si la variable est définie et non vide, sinon elle renvoie une chaîne vide.

```
$ existante=4
$ echo ${existante:+1}
1
$ echo ${inexistante:+1}

$
```

Cette opération est surtout utile en la couplant avec le modificateur `:=` pour obtenir une valeur précise si une variable est définie, et une autre valeur si elle ne l'est pas. Ici, la variable `definie` prendra pour valeur `oui` si `var_testee` est définie et non vide, et `non` dans le cas contraire :

```
$ var_testee=
$ definie=${var_testee:+oui}
$ : ${definie:=non}
$ echo $definie
non
$ var_testee=1
$ definie=${var_testee:+oui}
$ : ${definie:=non}
$ echo $definie
oui
$
```

Les quatre modificateurs précédents considèrent au même titre les variables indéfinies et les variables contenant une chaîne vide. Il existe quatre modificateurs similaires qui n'agissent que si la variable est vraiment indéfinie ; il s'agit de `${ - }`, `${ = }`, `${ ? }` et `${ + }` :

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

$
```

Calcul arithmétique

Le shell permet de réaliser des calculs arithmétiques simples – uniquement avec des valeurs entières –, ce qui est parfois très précieux. Nous verrons plus avant que l'utilitaire système `bc` nous permettra de réaliser des opérations en virgule flottante.

Les calculs arithmétiques sont représentés par l'opérateur `$((`*opérations*`))`. Il est déconseillé d'utiliser sa forme ancienne, `$[`*opérations*`]`, car elle est considérée comme obsolète.

Les opérateurs arithmétiques disponibles sont les mêmes qu'en langage C : `+`, `-`, `*`, et `/` pour les quatre opérations de base, `%` pour le modulo, `<<` et `>>` pour les décalages binaires à gauche et à droite, `&`, `|`, et `^` pour les opérateurs binaires ET, OU et OU EXCLUSIF, et finalement `~` pour la négation binaire. Les opérateurs peuvent être regroupés entre parenthèses pour des questions de priorité. On peut placer des blancs (espaces) à volonté pour améliorer la lisibilité du code.

```
$ echo $((2 * (4 + (10/2)) - 1))
17
$ echo $((7 % 3))
1
```

Une constante numérique est considérée par défaut en base 10. Si elle commence par `0` elle est considérée comme étant octale, et par `0x` comme hexadécimale. On peut aussi employer une représentation particulière *`base#nombre`* où l'on précise explicitement la base employée (jusqu'à 36 au maximum). Cela peut servir surtout pour manipuler des données binaires :

```
$ masque=2#000110
$ capteur=2#001010
$ echo $(($masque & $capteur))
2
$ echo $(($masque | $capteur))
14
$ echo $(($masque ^ $capteur))
12
$ echo $(($masque << 2))
24
$
```

> Il arrive que les données numériques d'un calcul proviennent d'un programme externe ou d'une saisie de l'utilisateur, et que rien ne garantisse qu'elles ne commencent pas par un zéro, même si elles sont en décimal. Supposons que l'utilisateur remplisse la variable `x=010`, le calcul `$(($x+1))` renverrait le résultat (décimal) 9 car le shell interpréterait le contenu de `x` comme une valeur octale. Pour éviter cela, on peut forcer l'interprétation en décimal avec `$((10#$x+1))`. Ceci est utile lorsqu'on récupère des nombres provenant de fichiers formatés avec des colonnes de chiffres complétés à gauche par des zéros.

Les variables qui se trouvent à l'intérieur de la structure de calcul `$(( ))` ne sont pas tenues d'être précédées par le caractère `$`, mais cela améliore généralement la lisibilité du programme. Si une variable n'est pas définie, est vide, ou contient une chaîne non numérique, elle est interprétée comme une valeur nulle.

Il est aussi possible de réaliser, au sein de la structure `$(( ))`, une ou plusieurs affectations de variables à l'aide de l'opérateur `=` .

```
$ echo $((x = 5 - 2))
3
$ echo $x
3
$ echo $((y = x * x + x + 1))
13
$ echo $y
13
$
```

Les affectations peuvent aussi se faire avec les raccourcis `+=`, `-=`, `*=`, `/=`, `<<=`, `>>=`, `&=`, `|=`, `^=`. Par exemple, `$((a+=4))` est équivalent à `$((a=a+4))`, `$((a*=2))` équivaut à `$((a=a*2))`, ou encore `$((masque&=0x01))` à `$((masque=masque&0x01))`.

Une affectation renvoie la valeur calculée, ce qui permet de la réutiliser dans une autre opération : `$((a*=b+=2))` correspond à `$((b=b+2))` suivie de `$((a=a*b))`. Bien entendu, ce genre d'écriture est franchement illisible et totalement déconseillé.

La structure `$(( ))` peut également servir à vérifier des conditions arithmétiques. Les opérateurs de comparaison renvoient la valeur 1 pour indiquer qu'une condition est vérifiée, et 0 sinon. Il s'agit des comparaisons classiques `<`, `<=`, `>=`, `>` ainsi que `==` pour l'égalité et `!=` pour la différence. Les conditions peuvent être associées par un ET LOGIQUE `&&` ou un OU LOGIQUE `||`, ou encore être niées avec `!`.

```
$ echo $(((25 + 2) < 28))
1
$ echo $(((12 + 4) == 17))
0
$ echo $(((1 == 1) && (2 < 3)))
1
$
```

Nous verrons plus avant comment employer ces conditions dans des constructions `if-then`, `while-do`, etc.

Invocation de commande

Une dernière construction à base de l'opérateur `$` est appelée « substitution de commande ». Elle a la forme suivante : `$(`*commande*`)`. La forme `` `commande` ``, probablement héritée du shell C, est quant à elle peu recommandée car moins lisible et difficile à imbriquer. La commande qui se trouve entre les parenthèses est exécutée, et la valeur renvoyée est une chaîne de caractères contenant tout ce qu'elle a écrit sur sa sortie standard. Par exemple, l'invocation

```
$ variable=$(ls -l)
```

placera le résultat de la commande `ls` dans la variable. Les caractères de saut de ligne qui se trouvent à la fin de la chaîne sont éliminés, mais ceux qui sont rencontrés dans le cours de la chaîne sont conservés. Lorsqu'on désire examiner le contenu d'une telle variable, il ne suffit donc pas d'appeler

```
$ echo $variable
```

En effet, tout le contenu de la chaîne de caractères est ainsi placé sur la ligne de commande de `echo`, y compris les éventuelles successions d'espaces multiples, les tabulations et les retours à la ligne. Lors de l'interprétation de cette ligne de commande, le shell va remplacer tous ces éléments qu'il considère comme des séparateurs d'arguments par des espaces uniques. En voici un exemple :

```
$ cd /var/spool/lpd/
$ ls -l
total 3
drwxr-xr-x   2 root     lp           1024 Sep 21 13:59 lp0
-rw-r--r--   1 root     root            4 Oct  3 23:47 lpd.lock
```

```
drwxr-xr-x   2 root     lp           1024 Oct  2 18:35 photo
$ variable=$(ls -l)
$ echo $variable
total 3 drwxr-xr-x 2 root lp 1024 Sep 21 13:59 lp0 -rw-r--r-- 1 root
root 4 Oct 3 23:47 lpd.lock drwxr-xr-x 2 root lp 1024 Oct 2 18:35 ph
oto
$
```

Pour afficher correctement le contenu d'une variable qui renferme des espaces multiples, des tabulations et des retours à la ligne, il faut que ce contenu soit protégé de l'interprétation du shell, ce qui s'obtient en l'encadrant par des guillemets. Nous reviendrons sur ce mécanisme.

```
$ echo "$variable"
total 3
drwxr-xr-x   2 root     lp           1024 Sep 21 13:59 lp0
-rw-r--r--   1 root     root            4 Oct  3 23:47 lpd.lock
drwxr-xr-x   2 root     lp           1024 Oct  2 18:35 photo
$
```

Pour mettre en pratique ces dernières acquisitions, nous allons étudier un petit script, assez compliqué à première vue. Il s'agit de calculer automatiquement la date de la fête de Pâques pour une année donnée. Pour cela, nous utiliserons un algorithme décrit dans la FAQ `sci.astro` (section C.07) à laquelle il est utile de se reporter pour obtenir plus de renseignements sur la signification des calculs. Toutes les variables sont entières.

A étant l'année, on calcule successivement :

$G = (A \bmod 19) + 1$

$$H = \frac{A}{100}$$

$$C = -H + \text{int}\left(\frac{H}{4}\right) + \text{int}\left(\frac{8 \times (H + 11)}{25}\right)$$

$J = 50 - (11 \times G \times C) \bmod 30$

Si *J* vaut 50 alors il faut le remplacer par 49.

Si *J* vaut 49 et si *G* est supérieur à 12, alors *J* doit être remplacé par 48.

J représente le jour du mois de mars durant lequel tombe la pleine lune pascale. Ce numéro de jour peut aller jusqu'à 49 qui représente le 18 avril. Pâques se fête le dimanche qui suit immédiatement cette pleine lune. Voici le script correspondant.

calcule_paques.sh :

```
1    #! /bin/sh
2
3    annee=${1:-$(date +%Y)}
4    echo "Calcul de la date de Pâques pour" $annee
5    : $((G = $annee % 19 + 1))
6    : $((H = $annee / 100))
```

```
 7    : $((C = -$H + $H/4 + (8 * ($H + 11)) / 25))
 8    : $((J = 50 - (11 * $G + $C) % 30))
 9    if [ $J -eq 50 ] ; then
10        J=49
11    fi
12    if [ $J -eq 49 ] && [ $G -ge 12 ] ; then
13        J=48
14    fi
15    # Pâques tombe le dimanche immédiatement après le
16    # J Mars (J pouvant aller jusqu'à 49, c'est à dire
17    # le 18 Avril).
18    Jour_Semaine=$(date -d "03/$J/$annee" +%w)
19    : $((Jour_Paques = $J + (7 - $Jour_Semaine) % 7))
20
21    date -d "03/$Jour_Paques/$annee" +%x
```

En premier lieu, ce script peut être invoqué avec un argument qui représente l'année du calcul. Si aucun argument n'est fourni, on utilise par défaut l'année en cours. On réalise cela sur la ligne 3 : nous verrons un peu plus bas que le premier argument de la ligne de commande peut être consulté avec l'invocation `$1`. Ici, la construction est un peu plus complexe, puisque, si `$1` n'est pas définie, nous employons une valeur par défaut renvoyée par la commande `date`, et son option `+Y%` qui fournit l'année en cours.

Sur les lignes 5 à 14, nous pouvons reconnaître les éléments de l'algorithme de calcul que nous avons présenté plus haut. Comme les valeurs de retour des constructions arithmétiques ne sont pas utilisées, nous employons la commande interne deux-points pour les ignorer proprement. Les tests des lignes 9 et 12 vérifient des conditions sur les valeurs arithmétiques des variables `J` et `G`, comme nous le verrons dans le prochain chapitre.

La ligne 18 donne le numéro du jour de la semaine de la pleine lune pascale, entre 0 et 6, le 0 représentant le dimanche. La ligne 19 permet d'obtenir le dimanche suivant cette pleine lune, compté à partir du 1er mars. La ligne 21 affiche finalement ce résultat dans le format préféré pour la localisation courante. Voici quelques exemples d'exécution :

```
$ ./calcule_paques.sh
Calcul de la date de Pâques pour 2003
20.04.2003
$ ./calcule_paques.sh 2004
Calcul de la date de Pâques pour 2004
11.04.2004
$ ./calcule_paques.sh 2005
Calcul de la date de Pâques pour 2005
27.03.2005
$ ./calcule_paques.sh 2000
Calcul de la date de Pâques pour 2000
23.04.2000
$ ./calcule_paques.sh 1970
Calcul de la date de Pâques pour 1970
22.03.1970
$
```

Le shell Bash nous permet donc de réaliser un calcul déjà relativement complexe. Nous sommes pour le moment limités aux valeurs entières, mais nous verrons qu'au moyen d'un utilitaire comme `bc`, il est tout à fait possible de traiter également des valeurs réelles.

Portées et attributs des variables

Les variables employées par le shell sont caractérisées par leur portée, c'est-à-dire par une certaine visibilité dans les sous-processus, fonctions, etc.

Une variable qui est simplement définie avec une commande `variable=valeur`, sans autres précisions, est accessible dans l'ensemble du processus en cours. Cela comprend les fonctions que l'on peut invoquer :

```
$ var=123
$ function affiche_var ()
> {
>   echo $var
> }
$ affiche_var
123
$ var=456
$ affiche_var
456
$
```

La fonction a accès au contenu de la variable en lecture, mais aussi en écriture :

```
$ function modifie_var ()
> {
>     var=123
> }
$ var=456
$ echo $var
456
$ modifie_var
$ echo $var
123
$
```

Une fonction peut aussi définir une variable qui n'existait pas jusqu'alors, et qui sera désormais accessible par le reste du processus, même après la fin de la fonction. Pour observer ce phénomène, il faut configurer le shell avec la commande `set -u` pour qu'il fournisse un message d'erreur si on tente d'accéder à une variable indéfinie.

```
$ set -u
$ function definit_var ()
> {
>   nouvelle_var=123
> }
$ echo $nouvelle_var
```

```
bash: nouvelle_var: unbound variable
$ definit_var
$ echo $nouvelle_var
123
$
```

Une variable qui est définie sans autres précisions est partagée avec les scripts que l'on exécute avec la commande `source` *`fichier`* (ou le point . *`fichier`*). Nous avons utilisé cette propriété dans le script `rm_secure` du chapitre 1, et la variable `sauvegarde_rm`. En revanche, la variable n'est accessible ni par un script invoqué par l'intermédiaire d'un sous-processus, ni par le processus père qui a lancé le shell en cours.

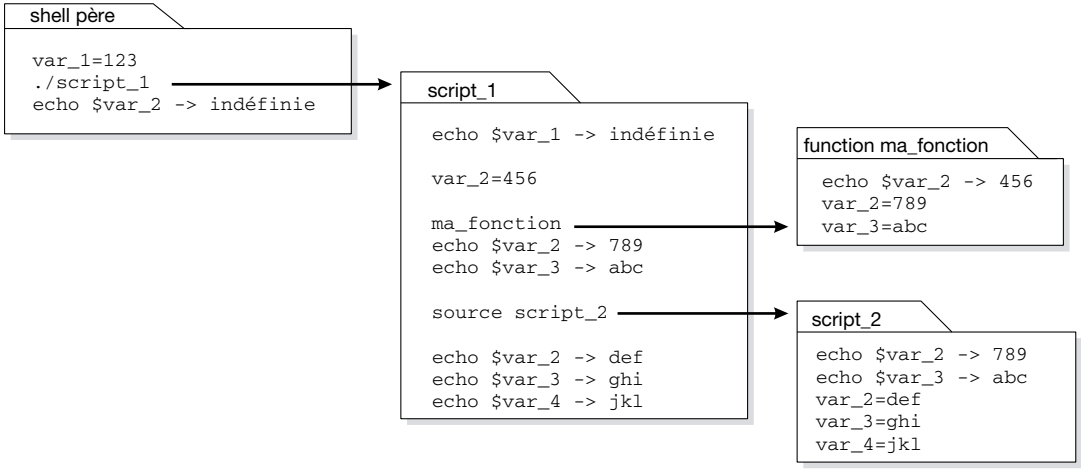


Figure 2–5
Visibilité des variables normales

La figure 2-5 résume la visibilité des variables classiques. Le shell père (ou un script original) définit une variable `var_1` et invoque `script_1` dans un sous-processus (par l'intermédiaire de l'appel `./script_1`). Lors du retour de ce script, la variable `var_2` qui y est définie reste invisible pour le shell père.

Le processus `script_1` ne peut pas voir la variable `var_1` qui est définie par le shell père. En revanche, la variable `var_2` qu'il définit est visible dans la fonction et dans le `script_2` qu'il invoque par la commande `source`. De même, lorsqu'une variable est définie ou modifiée dans la fonction ou dans le `script_2`, les résultats sont visibles dans `script_1`.

Restriction de portée

La seule restriction possible de visibilité concerne les fonctions. Il est possible d'y déclarer des variables locales qui ne seront pas accessibles dans le processus supérieur. Elles seront en revanche parfaitement disponibles pour les sous-fonctions éventuellement invoquées.

Lorsqu'on déclare une variable avec le mot-clé `local`, et qu'elle existait déjà au niveau supérieur du script, la nouvelle instance masque la précédente jusqu'au retour de la fonction. Le script suivant va nous permettre de mettre en évidence ce comportement.

var_locales.sh :

```
#! /bin/sh

function ma_fonction ()
{
    local var="dans fonction"
    echo "  entrée dans ma_fonction"
    echo "  var = " $var
    echo "  appel de sous_fonction"
    sous_fonction
    echo "  var = " $var
    echo "  sortie de ma_fonction"
}

function sous_fonction ()
{
    echo "    entrée dans sous_fonction"
    echo "    var = " $var
    echo "    modification de var"
    var="dans sous_fonction"
    echo "    var = " $var
    echo "    sortie de sous_fonction"
}

    echo "entrée dans le script"
    var="dans le script"
    echo "var = " $var
    echo "appel de ma_fonction"
    ma_fonction
    echo "var = " $var
```

L'invocation résume bien l'accès uniquement local aux variables :

```
$ ./var_locales.sh
entrée dans le script
var =  dans le script
appel de ma_fonction
  entrée dans ma_fonction
  var =  dans fonction
  appel de sous_fonction
    entrée dans sous_fonction
    var =  dans fonction
    modification de var
    var =  dans sous_fonction
    sortie de sous_fonction
```

```
  var =  dans sous_fonction
  sortie de ma_fonction
var =  dans le script

$
```

On notera, toutefois, que le comportement des variables locales n'est pas tout à fait le même que dans d'autres langages de programmation. On peut constater que, dans les scripts shell, le masquage d'une variable globale par une variable locale concerne aussi les sous-fonctions qui peuvent être appelées à partir de la routine en question.

En C, par exemple, si une variable locale d'une routine a le même nom qu'une variable globale, la superposition ne recouvre que la portée de la routine en question, et pas les fonctions qu'elle peut appeler. Nous verrons qu'en Perl les deux comportements sont possibles suivant l'emploi du mot-clé `local` ou de `my`.

Extension de portée – Environnement

Nous avons remarqué que, lorsqu'un processus lance un script par l'intermédiaire d'un appel `./script`, il ne lui transmet pas les variables définies d'une manière courante. Ces variables en effet sont placées en mémoire dans une zone de travail du shell, qui n'est pas copiée lorsqu'un nouveau processus démarre pour exécuter le script. Il est parfois indispensable de transmettre des données à ce nouveau script, et il faut donc trouver un moyen de placer les variables dans une zone de mémoire qui sera copiée dans l'espace du nouveau processus. Cette zone existe : c'est l'*environnement* du processus.

L'environnement fait partie des éléments qui sont hérités lorsqu'un processus crée – grâce à l'appel système `fork()` – un nouveau processus fils. Pour qu'une variable y soit disponible, il faut qu'elle soit marquée comme exportable. On obtient cela tout simplement grâce à la commande interne `export`. Pour observer le comportement de ce mécanisme, nous définissons des variables, en exportons une partie, invoquons un nouveau shell (donc un sous-processus) et examinons celles qui sont visibles :

```
$ var1="variable non-exportée"
$ var2="première variable exportée"
$ export var2
$ export var3="seconde variable exportée"
$ /bin/sh
$ echo $var1

$ echo $var2
première variable exportée
$ echo $var3
seconde variable exportée
$
```

Une variable qui se trouve dans l'environnement lors du démarrage d'un processus est automatiquement exportée pour ses futurs processus fils. On peut le vérifier en continuant la manipulation et en invoquant un nouveau shell.

```
$ var2="variable modifiée dans le second shell"
$ /bin/sh
$ echo $var2
variable modifiée dans le second shell
$ echo $var3
seconde variable exportée
$ exit
exit
$
```

La variable `var2` a été modifiée dans l'environnement du sous-shell (et de son fils), mais elle ne le sera pas dans l'environnement du shell original. Vérifions cela en remontant d'un niveau :

```
$ exit
exit
$ echo $var2
première variable exportée
$
```

La commande `export` marque les variables transmises en argument comme exportables (par exemple, `export var1 var2`), mais peut aussi regrouper une ou plusieurs affectation de variables sur la même ligne, par exemple :

```
$ export v1=1 v2=2
$ echo $v1 $v2
1 2
$
```

Bash dispose d'une autre commande interne, nommée `declare`, dont l'option `-x` permet de marquer une variable pour qu'elle soit exportée dans l'environnement, exactement comme la fonction `export` :

```
$ declare -x a1=123 a2=456
$ /bin/sh
$ echo $a1 $a2
123 456
$ exit
exit
$
```

La fonction `export` offre plusieurs options avec lesquelles il est possible de manipuler l'environnement.

« `export -p` » affiche la liste des éléments marqués comme exportables :

```
$ export -p
declare -x BASH_ENV="/home/ccb/.bashrc"
declare -x HISTFILESIZE="1000"
```

```
declare -x HISTSIZE="1000"
declare -x HOME="/home/ccb"
declare -x HOSTNAME="venux.ccb"
declare -x HOSTTYPE="i386"
declare -x INPUTRC="/etc/inputrc"
declare -x KDEDIR="/usr"
declare -x LANG="fr_FR"
declare -x LC_ALL="fr_FR"
declare -x LINGUAS="fr"
declare -x LOGNAME="ccb"
declare -x MAIL="/var/spool/mail/ccb"
declare -x OSTYPE="Linux"
declare -x PATH="/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin"
declare -x PS1="\$ "
declare -x QTDIR="/usr/lib/qt-2.0.1"
declare -x SHELL="/bin/bash"
declare -x SHLVL="1"
declare -x TERM="ansi"
declare -x USER="ccb"
declare -x USERNAME=""
$
```

On remarque d'ailleurs que les variables sont précédées d'un `declare -x` qui indique leur attribut « exportable ». On peut noter aussi que par convention le nom des variables d'environnement est écrit en majuscules. Ce n'est toutefois qu'une simple convention.

« `export -n` » enlève l'attribut « exportable » d'une variable. Elle n'est en revanche pas supprimée pour autant ; elle sera seulement indisponible dans les processus fils comme en témoigne l'exemple suivant :

```
$ export var1="exportée"
$ /bin/sh
$ echo $var1
exportée
$ exit
exit
$ export -n var1
$ echo $var1
exportée
$ /bin/sh
$ echo $var1

$ exit
exit
$
```

« `export -f` » permet d'exporter une fonction. Par défaut, les fonctions ne sont pas copiées dans l'environnement du processus fils. La commande interne « `declare -f` » affiche la liste de celles qui sont définies dans la zone de travail en cours (mais pas

nécessairement exportées). Nous voyons par exemple que, sur notre machine, la fonction `rm` définie dans le chapitre précédent n'est pas exportée dans l'environnement :

```
$ declare -f
declare -f rm ()
{
    local opt_force=0;
    local opt_interactive=0;
 ...
        mv -f "$1" "${sauvegarde_rm}/";
        shift;
    done
}
$ export -f
$ sh
$ declare -f
$ exit
exit
$
```

Pour qu'elle soit disponible dans un sous-shell, nous devons ajouter, dans le script `rm_secure`, la ligne suivante après la définition de la fonction :

```
106         export -f rm
```

De même, nous devons ajouter la commande `export` sur la ligne numéro 1 afin que la variable `sauvegarde_rm` soit également disponible dans les sous-shells.

Ainsi, lors de la connexion suivante, la fonction est bien disponible dans l'environnement des sous-shells :

```
$ export -f
declare -f rm ()
{
    local opt_force=0;
...
    done
}
$ sh
$ export -f
declare -f rm ()
{
    local opt_force=0;
...
    done
}
$ exit
exit
$
```

Les options de la commande `export` sont cumulables (par exemple, `-f -n`, pour annuler l'exportation d'une fonction). Le tableau suivant en rappelle la liste, ainsi que les options

équivalentes de la commande `declare` (qui en compte d'autres que nous verrons ci-après). Pour cette dernière, une option précédée d'un + a l'effet inverse de l'option précédée d'un –.

Option `export`	Option `declare`	Signification
`export var`	`declare -x var`	Marque la variable pour qu'elle soit exportée dans l'environnement.
`export -p`	`declare`	Affiche la liste des éléments exportés.
`export -n var`	`declare +x var`	Retire l'attribut exportable de la variable.
`export -f fonct`	`declare -f fonct`	Exporte une fonction et non pas une variable.

Le contenu actuel de l'environnement peut aussi être consulté avec un utilitaire système nommé `/usr/bin/printenv`.

> Il est important de bien comprendre qu'une variable exportée est simplement copiée depuis l'environnement de travail du shell père vers celui du shell fils. Elle n'est absolument pas partagée entre les deux processus, et aucune modification apportée dans le shell fils ne sera visible dans l'environnement de celui du père.

Attributs des variables

Nous avons déjà vu qu'une variable pouvait être dotée d'un attribut « exportable » qui permet de la placer dans une zone mémoire qui sera copiée dans l'environnement de travail du processus. Deux autres attributs sont également disponibles, que l'on peut activer à l'aide de la commande `declare`.

Variable arithmétique

On peut indiquer avec l'option `declare -i` que le contenu d'une variable sera strictement arithmétique. Lors de l'affectation d'une valeur à cette variable, le shell procédera systématiquement à une phase d'évaluation arithmétique exactement semblable à ce qu'on obtient avec l'opérateur `$(( ))`. En fait, on peut considérer que, si une variable *A* est déclarée arithmétique, toute affectation du type `A=`*xxx* sera évaluée sous la forme `A=$((`*xxx*`))`.

Si la valeur que l'on essaie d'inscrire dans une variable arithmétique est une chaîne de caractères, ou est indéfinie, la variable est remplie avec un zéro. Voici quelques exemples pour se fixer les idées : `A` est arithmétique, `B` et `C` ne le sont pas. On constate tout d'abord qu'une chaîne de caractères, « `1+1` » en l'occurrence, est évaluée différemment suivant le contexte.

```
$ declare -i A
$ A=1+1
$ echo $A
2
$ B=1+1
```

```
$ echo $B
1+1
$
```

Si on veut insérer des espaces dans la formule, il faut la regrouper en un seul bloc au moyen de guillemets ; nous reviendrons sur ce point ultérieurement.

```
$ A=4*7 + 67
sh: +: command not found
$ A="4*7 + 67"
$ echo $A
95
$
```

Lors de l'affectation d'une variable arithmétique, le contenu des variables éventuellement contenues dans l'expression est évalué sous forme arithmétique :

```
$ C=5*7
$ echo $C
5*7
$ A=C
$ echo $A
35
$
```

Nous pouvons d'ailleurs remarquer qu'à l'instar de la construction `$(( ))`, il n'était pas indispensable de mettre un `$` devant le `C` pour en prendre la valeur (cela améliorerait quand même la lisibilité). Vérifions finalement qu'une valeur non arithmétique est bien considérée comme un zéro :

```
$ A="ABC"
$ echo $A
0
$
```

La commande « `declare -i` » seule affiche la liste des variables arithmétiques. La commande « `declare +i` *var* » supprime l'attribut arithmétique d'une variable.

```
$ declare -i A
$ declare -i
declare -i A="0"
declare -ri EUID="500"
declare -ri PPID="12393"
declare -ri UID="500"
$ A=1+1
$ echo $A
2
$ declare +i A
$ A=1+1
$ echo $A
1+1
$
```

Les variables arithmétiques améliorent la lisibilité d'un script qui réalise de multiples calculs et en facilite l'écriture. Par exemple, on pourrait modifier le calcul de la date de la fête de Pâques de cette façon :

```
5    declare -i G H C J
6    G="$annee % 19 + 1"
7    H="$annee / 100"
8    C="-$H + $H/4 + (8 * ($H + 11)) / 25"
9    J="50 - (11 * $G + $C) % 30"
```

Variable en lecture seule

L'option `-r` de la commande `declare` permet de figer une variable afin qu'elle ne soit plus accessible qu'en lecture (ce qui en fait finalement une constante et non plus une variable !). Cela ne concerne toutefois que le processus en cours ; si la variable est exportée vers un shell fils, ce dernier pourra la modifier dans son propre environnement. Voyons quelques exemples :

```
$ A=Immuable
$ echo $A
Immuable
$ declare -r A
$ A=Modifiée
bash: A: read-only variable
$ echo $A
Immuable
$ unset A
unset: A: cannot unset: readonly variable
$ export A
$ sh
$ echo $A
Immuable
$ A=Changée
$ echo $A
Changée
$ exit
exit
$ echo $A
Immuable
$
```

Bash définit automatiquement quelques constantes non modifiables, concernant l'identification du processus en cours.

```
$ declare -r
declare -ri EUID="500"
declare -ri PPID="13881"
declare -ri UID="500"
$
```

On y recourt principalement pour définir des constantes en début de script, par exemple des limites ou des représentations symboliques de valeurs numériques. On peut alors les compléter avec l'option –i, qui indique qu'il s'agit de valeurs arithmétiques, par exemple :

```
declare -r  MSG_ACCUEIL="Entrez votre commande :"

declare -ri NB_PROCESSUS_MAX=256

declare -ri BIT_LECTURE=0x0200
declare -ri BIT_ECRITURE=0x0100
```

Paramètres

Dans la terminologie de la programmation shell, le concept de variable est inclus dans une notion plus large, celle de paramètres. Il s'agit pour l'essentiel de données en lecture seule, que le shell met à disposition pour que l'on puisse obtenir des informations sur l'exécution du script. Il en existe deux catégories : les paramètres positionnels et les paramètres spéciaux.

Paramètres positionnels

Les paramètres positionnels sont utilisés pour accéder aux informations qui sont fournies sur la ligne de commande lors de l'invocation d'un script, mais aussi aux arguments transmis à une fonction. Les arguments sont placés dans des paramètres qui peuvent être consultés avec la syntaxe `$1`, `$2`, `$3`, etc. Pour les paramètres positionnels, comme pour les paramètres spéciaux qui seront vus ci-après, l'affectation est *ipso facto* impossible car on ne peut pas écrire `1=2` !

Le premier argument transmis est accessible en lisant `$1`, le deuxième dans `$2`, et ainsi de suite. Pour consulter le contenu d'un paramètre qui comporte plus d'un chiffre, il faut l'encadrer par des accolades ; ainsi, `${10}` correspond bien au contenu du dixième argument, tandis que `$10` est une chaîne constituée de la valeur du premier argument, suivie d'un zéro.

Par convention, l'argument numéro zéro contient toujours le nom du script tel qu'il a été invoqué. Le script suivant affiche les arguments existants, jusqu'au dixième, en vérifiant chaque fois si la chaîne correspondante n'est pas vide :

affiche_arguments.sh :

```
1    #! /bin/sh
2
3    echo 0 : $0
4    if [ -n "$1"  ] ; then echo 1  : $1  ; fi
5    if [ -n "$2"  ] ; then echo 2  : $2  ; fi
6    if [ -n "$3"  ] ; then echo 3  : $3  ; fi
7    if [ -n "$4"  ] ; then echo 4  : $4  ; fi
8    if [ -n "$5"  ] ; then echo 5  : $5  ; fi
```

```
 9    if [ -n "$6"  ] ; then echo 6  : $6  ; fi
10    if [ -n "$7"  ] ; then echo 7  : $7  ; fi
11    if [ -n "$8"  ] ; then echo 8  : $8  ; fi
12    if [ -n "$9"  ] ; then echo 9  : $9  ; fi
13    if [ -n "${10}" ] ; then echo 10 : ${10} ; fi
```

Ce script n'est ni élégant ni très efficace, mais nous en verrons une version améliorée plus bas. Il permet quand même de récupérer le contenu des arguments :

```
$ ./affiche_arguments.sh
0 : ./affiche_arguments.sh
$ ./affiche_arguments.sh premier deuxième troisième
0 : ./affiche_arguments.sh
1 : premier
2 : deuxième
3 : troisième
$ ./affiche_arguments.sh a b c d e f g h et dix
0 : ./affiche_arguments.sh
1 : a
2 : b
3 : c
4 : d
5 : e
6 : f
7 : g
8 : h
9 : et
10 : dix
$
```

Lors de l'invocation d'une fonction, les paramètres positionnels sont temporairement remplacés par les arguments fournis lors de l'appel. Le script suivant affiche le contenu de `$0`, `$1`, `$2` dans et en dehors d'une fonction :

affiche_arguments_2.sh :

```
 1    #! /bin/sh
 2
 3    function fonct ()
 4    {
 5      echo "  Dans la fonction : "
 6      echo "  0 : $0"
 7      echo "  1 : $1"
 8      echo "  2 : $2"
 9    }
10
11      echo "Dans le script : "
12      echo 0 : $0
13      echo 1 : $1
14      echo 2 : $2
15      echo "Appel de : fonct un deux"
16      fonct un deux
```

Nous pouvons remarquer qu'il n'y a pas de différences sur le paramètre `$0` entre l'intérieur et l'extérieur de la fonction :

```
$ ./affiche_arguments_2.sh premier deuxième
Dans le script :
0 : ./affiche_arguments_2.sh
1 : premier
2 : deuxième
Appel de : fonct un deux
  Dans la fonction :
  0 : ./affiche_arguments_2.sh
  1 : un
  2 : deux
$
```

Nous avons indiqué plus haut qu'il n'était pas possible d'affecter une valeur à un paramètre positionnel par une simple phrase `1=xxx` ; malgré tout, il est quand même possible de modifier leur contenu. Toutefois, la modification porte sur l'ensemble des arguments.

La fonction `set` permet entre autres choses de fixer la valeur des paramètres `$1`, `$2`, etc. Les valeurs transmises sur la ligne de commande de `set` et qui ne font pas partie des (nombreuses) options de cette dernière sont utilisées pour remplir les paramètres positionnels.

```
$ set a b c
$ echo $1 $2 $3
a b c
$ set d e
$ echo $1 $2 $3
d e
$
```

On remarquera que c'est l'ensemble des paramètres positionnels qui est modifié ; la valeur de `$3` dans notre exemple est supprimée, même si on ne fournit que deux arguments à `set`.

On peut aussi modifier les paramètres positionnels dans un script en invoquant la commande `shift`. Celle-ci va décaler l'ensemble des paramètres : `$1` est remplacé par `$2`, celui-ci par `$3`, et ainsi de suite. Le paramètre `$0` n'est pas concerné par cette commande.

```
$ set a b c d e
$ echo $0 $1 $2 $3
-bash a b c
$ shift
$ echo $0 $1 $2 $3
-bash b c d
$ shift
$ echo $0 $1 $2 $3
-bash c d e
$ shift
$ echo $0 $1 $2 $3
-bash c d
$
```

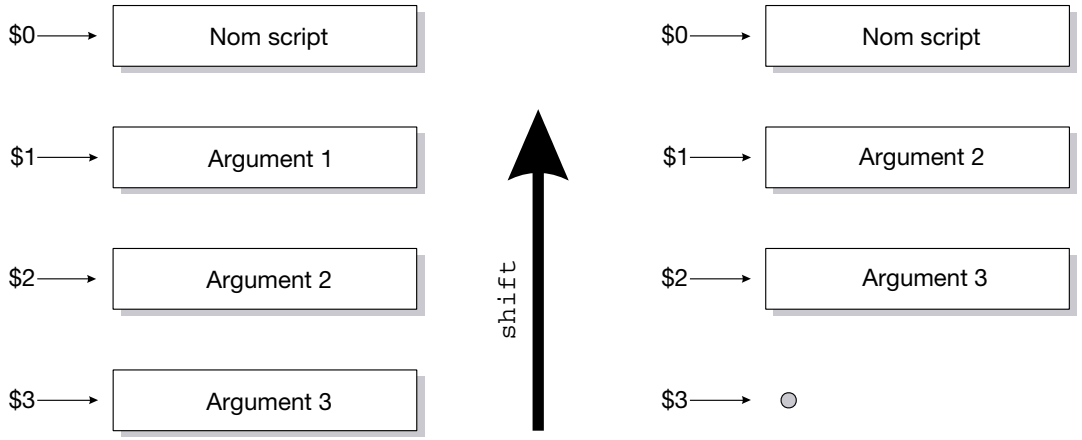


Figure 2–6
Décalage des paramètres positionnels avec `shift`

Si l'instruction `shift` est suivie d'un nombre, ce dernier représente le nombre de décalages désirés. Par défaut, il s'agit d'un seul décalage mais, par exemple, `shift 3` permet de remplacer `$1` par `$3`, `$2` par `$4`, etc. Il est possible de récrire une version un peu plus élégante du programme d'affichage des arguments, dans laquelle on boucle sur `echo` et `shift` tant que `$1` n'est pas vide.

affiche_arguments_3.sh :

```
1    #! /bin/sh
2
3    while [ -n "$1" ] ; do
4      echo $1
5      shift
6    done
```

Le résultat correspond à nos attentes :

```
$ ./affiche_arguments_3.sh un deux trois quatre
un
deux
trois
quatre
$
```

Dans un vrai script, l'analyse des arguments de la ligne de commande doit être menée avec soin. Nous avons déjà rencontré une commande nommée `getopts` qui permet de rechercher les options dans la liste des arguments ; nous reviendrons sur cette fonction dans le chapitre 4.

Paramètres spéciaux

Nos programmes d'affichage des arguments ne sont pas suffisants, car ils ne sont pas capables de faire la différence entre un argument qui représente une chaîne vide (saisi avec `""` sur la ligne de commande) et la fin de la liste des arguments. En voici la preuve :

```
$ ./affiche_arguments.sh un "" deux trois
0 : ./affiche_arguments.sh
1 : un
3 : deux
4 : trois
$ ./affiche_arguments_3.sh un deux "" trois quatre
un
deux
$
```

Le premier script n'a pas affiché `$2` car il l'a vu vide, quant au second il s'est tout simplement arrêté après avoir rencontré un `$3` vide, pensant être arrivé à la fin des arguments.

Pour corriger ce problème, il nous faudrait connaître le nombre de paramètres positionnels. Par chance, le shell nous y donne accès à travers l'un de ses *paramètres spéciaux*. Il s'agit simplement de paramètres en lecture seule qui nous donnent des indications sur l'environnement d'exécution du processus.

> Comme pour les paramètres positionnels, les noms des divers paramètres spéciaux (`$$`, `$*`, `$#`, etc.) font qu'il n'est pas possible de leur affecter une valeur avec la construction =. Ils ne sont donc, de fait, accessibles qu'en lecture seule.

Le nombre d'arguments, sans compter le paramètre `$0`, est contenu dans le paramètre `$#`. Cette valeur est naturellement modifiée par `shift`. Nous pouvons améliorer le script précédent de façon qu'il attende que `$#` soit égal à zéro avant de s'arrêter.

affiche_arguments_4.sh :

```
1    #! /bin/sh
2
3    while [ $# -ne 0 ]; do
4      echo $1
5      shift
6    done
```

Cette fois-ci, les arguments vides sont traités correctement :

```
$ ./affiche_arguments_4.sh un deux "" trois "" quatre cinq
un
deux

trois
```

```
quatre
cinq
$
```

Il est souvent utile de pouvoir manipuler en une seule fois l'ensemble des paramètres positionnels, afin de les transmettre à une fonction, à un autre script, à une commande, etc. Pour ce faire, on dispose de deux paramètres spéciaux, `$*` et `$@`, et il est important de bien comprendre ce qui les différencie.

L'un comme l'autre vont se développer pour représenter l'ensemble des paramètres positionnels. Avec `$*`, tous les paramètres vont être écrits les uns à la suite des autres. Le script suivant va invoquer `affiche_arguments_5.sh`, en lui transmettant l'ensemble de ses propres arguments :

affiche_arguments_6.sh :

```
1   #! /bin/sh
2
3   . :affiche_arguments_5.sh $*
```

Voici un exemple d'exécution :

```
$ ./affiche_arguments_6.sh un deux trois
0 :./affiche_arguments_5.sh
1 :un
2 :deux
3 :trois
$
```

Apparemment, tout fonctionne bien. Pourtant, un problème se pose lorsqu'un des arguments est une chaîne de caractères qui contient déjà une espace :

```
$ ./affiche_arguments_6.sh un deux "trois et quatre"
0 :./affiche_arguments_5.sh
1 :un
2 :deux
3 :trois
4 :et
5 :quatre
$
```

Toutefois, si nous appelons directement `affiche_arguments_5.sh`, le résultat est différent :

```
$ ./affiche_arguments_5.sh un deux "trois et quatre"
0 :./affiche_arguments_5.sh
1 :un
2 :deux
3 :trois et quatre
$
```

En fait, lors de l'évaluation de `$*`, tous les guillemets sont supprimés, et les arguments sont mis bout à bout. Si l'un d'eux contient une espace, il est scindé en plusieurs arguments. Cela peut paraître anodin, mais c'est en réalité très gênant. Supposons par exemple que

nous souhaitions créer, comme c'est souvent l'usage, un script nommé `ll` qui serve à invoquer `ls` avec l'option `-l` pour avoir un affichage long du contenu d'un répertoire. Notre première idée pourrait être celle-ci :

ll_1.sh :

```
1    #! /bin/sh
2    ls -l $*
```

À première vue, ce script fonctionne correctement :

```
$ ./ll_1.sh c*
-rwxr-xr-x  1 ccb   ccb      560 Oct  4 13:23 calcule_paques.sh
-rwxr-xr-x  1 ccb   ccb      551 Oct 17 15:54 calcule_paques_2.sh
$ ./ll_1.sh /dev/cdrom
lrwxrwxrwx  1 root  root       3 Aug 12  1999 /dev/cdrom -> hdc
$
```

Pourtant, si nous rencontrons un fichier dont le nom contient un blanc typographique (comme c'est souvent le cas sur les partitions Windows), le résultat est différent :

```
$ ls -d /mnt/dos/P*
/mnt/dos/Program Files
$ ./ll_1.sh -d /mnt/dos/P*
ls: /mnt/dos/Program: Aucun fichier ou répertoire de ce type
ls: Files: Aucun fichier ou répertoire de ce type
$
```

L'option `-d` que nous avons ajoutée demande à `ls` d'afficher le nom du répertoire, `Program Files` en l'occurrence, et non pas son contenu. Le problème est que notre script a invoqué `ls` en lui passant sur la ligne de commande le nom du fichier séparé en deux morceaux. En fait, `ls` a reçu en argument `$1` l'option `-d`, en `$2` le mot `/mnt/dos/Program`, et en `$3` le mot `Files`.

On peut alors s'interroger sur le comportement de `$*` lorsqu'il est encadré par des guillemets. Le résultat du développement sera bien conservé en un seul morceau, sans séparer les deux moitiés du nom de fichier. Hélas, le problème n'est toujours pas réglé, car lorsque `"$*"` est évalué, il est remplacé par la liste de tous les arguments regroupés au sein d'une même expression, encadrée par des guillemets. Si nous essayons de l'utiliser pour notre script, le fonctionnement en sera pire encore :

ll_2.sh :

```
1    #! /bin/sh
2    ls -l "$*"
```

Ce script pourrait en effet fonctionner avec un nom de fichier contenant un blanc. Toutefois, tous les arguments étant regroupés en un seul, il ne pourrait accepter qu'un seul nom de fichier à la fois :

```
$ ./ll_2.sh /mnt/dos/Mes documents/Etiquettes CD.cdr
-rw-rw-r--  1 ccb   ccb   12610 Nov 27  1999 /mnt/dos/Mes documents/Et
iquettes CD.cdr
$ ./ll_2.sh c*
```

```
ls: calcule_paques.sh calcule_paques_2.sh: Aucun fichier ou répertoire
de ce type
$
```

Lors de la seconde invocation, `ls` a reçu un unique argument constitué de la chaîne `"calcule_paques.sh calcule_paques_2.sh"`.

Il nous faudrait donc un paramètre spécial, qui, lorsqu'il se développe entre guillemets, fournisse autant d'arguments qu'il y en avait à l'origine, mais en protégeant chacun d'eux par des guillemets. Ce paramètre existe, et est noté `$@`. Lorsqu'il n'est pas encadré de guillemets, il se comporte exactement comme `$*`. Sinon, il se développe comme nous le souhaitons :

ll.sh :

```
1    #! /bin/sh
2    ls -l "$@"
```

Cette fois-ci, le script agit comme nous le voulons :

```
$ ./ll.sh c*
-rwxr-xr-x  1 ccb   ccb        560 Oct  4 13:23 calcule_paques.sh
-rwxr-xr-x  1 ccb   ccb        551 Oct 17 15:54 calcule_paques_2.sh
$ ./ll.sh -d /mnt/dos/P*
dr-xr-xr-x 34 ccb   ccb       8192 Aug  6  1999 /mnt/dos/Program Files
$
```

Ainsi, dans la plupart des cas, lorsqu'un programme devra manipuler l'ensemble de ces arguments, il sera préférable d'employer le paramètre `"$@"` plutôt que `$*`.

D'autres paramètres spéciaux sont disponibles, comme `$$`, `$!` ou `$?`, que nous étudierons ultérieurement.

Protection des expressions

Il peut arriver que certaines expressions possèdent des caractères qui ont une signification particulière pour le shell, et que nous ne souhaitions pas qu'ils soient interprétés par ce dernier. Par exemple, afficher le prix américain $5 n'est pas si facile :

```
$ echo $5

$ echo "$5"

$
```

En effet, le shell peut en déduire que nous voulons afficher le contenu du cinquième paramètre positionnel, vide en l'occurrence. De même, le symbole # sert à introduire un commentaire qui s'étend jusqu'à la fin de la ligne, et le sens d'un message peut en être modifié :

```
$ echo en Fa # ou en Si ?
en Fa
$
```

Une autre surprise attend le programmeur qui veut réaliser un joli cadre autour du titre de son script :

```
$ echo **********************
affiche_arguments.sh affiche_arguments_2.sh affiche_arguments_3.sh
affiche_arguments_4.sh affiche_arguments_5.sh affiche_arguments_6.
sh calcule_paques.sh calcule_paques_2.sh extraction_serveurs.sh ll
.sh ll_1.sh ll_2.sh message.news var_locales.sh
$
```

Le caractère `*` remplace n'importe quelle chaîne dans un nom de fichier. La commande `echo` reçoit donc en argument la liste des fichiers du répertoire courant.

> Comme `echo` est généralement une commande interne du shell ayant précédence sur l'utilitaire `/bin/echo`, cette fonctionnalité permet de consulter le contenu d'un répertoire sans accéder à `/bin/ls`. Bien des administrateurs l'ont utilisée pour accéder à un système sur lequel ils venaient par erreur de détruire `/bin`, alors qu'un shell était toujours connecté.

La solution adéquate consiste à protéger le caractère spécial de l'interprétation du shell. Cela peut s'effectuer de plusieurs manières. On peut ainsi utiliser le caractère backslash (barre oblique inverse) '`\`', les apostrophes `' '`, ou encore les guillemets `" "`.

Protection par le caractère backslash

Ce caractère sert à désactiver l'interprétation du caractère qu'il précède. Ainsi, on peut écrire :

```
$ echo \$5
$5
$ echo Fa \# ou Do \#
Fa # ou Do #
$ echo \*\*\*
***
$
```

Le caractère backslash peut servir à préfixer n'importe quel caractère, y compris lui-même :

```
$ echo un \\ précède \$5
un \ précède $5
$
```

Lorsqu'il précède un retour chariot, le backslash a pour effet de l'éliminer et la saisie continue sur la ligne suivante.

```
$ echo début \
> et fin.
début et fin.
$
```

Protection par apostrophes

La protection des expressions par un backslash qui précède chaque caractère spécial est parfois un peu fastidieuse, et on lui préfère souvent le mécanisme de protection par des apostrophes, qui permet de manipuler toute l'expression en une seule fois. Entre des apostrophes, tous les caractères rencontrés perdent leur signification spéciale. Cela signifie que le backslash est un caractère comme les autres, et que l'on ne peut pas l'employer pour protéger une apostrophe. Le seul caractère qui ne puisse pas être inclus dans une chaîne protégée par des apostrophes est donc l'apostrophe elle-même.

```
$ echo '#\$"&>|'
#\$"&>|
$
```

Lorsqu'un retour chariot est présent dans une chaîne entre apostrophes, la représentation du code est inchangée :

```
$ echo 'début
> milieu
> et fin'
début
milieu
et fin
$
```

Protection par guillemets

La protection par des apostrophes est totale, chaque caractère gardant sa signification littérale. Il arrive pourtant que l'on préfère quelque chose d'un peu moins strict. La protection par les guillemets est plus adaptée dans ce cas puisqu'elle permet de conserver l'unité d'une chaîne de caractères sans fragmentation en différents mots.

Entre les guillemets, les caractères `$`, apostrophe et backslash retrouvent leurs significations spéciales, alors que les autres sont réduits à leurs acceptions littérales. En fait, le backslash et le `$` n'ont un sens particulier que s'ils sont suivis d'un caractère pour lequel l'interprétation spéciale a un sens. Voyons quelques exemples :

```
$ A="ABC DEF"
$ B="$A $ \$A \ \" "
$ echo $B
ABC DEF $ $A \ "
$
```

Dans l'affectation de la variable `B`, le premier `$` est suivi de `A` ; l'évaluation fournit le contenu de la variable `A`. Le deuxième `$` est suivi par un blanc, et une interprétation autre que littérale n'aurait pas de sens. Le troisième `$` est précédé d'un backslash, il n'a donc pas de sens particulier, et l'on affiche les deux caractères `$A`. Le backslash suivant précède une espace, la seule interprétation est donc littérale. Comme le premier guillemet est précédé d'un backslash, il est littéral et ne sert pas à fermer la chaîne ; il est donc affiché. Le second guillemet en revanche clôt l'expression.

Les guillemets sont très utiles pour préserver les espaces, tabulations et retours chariot dans une expression. Nous en avions vu un exemple plus haut lorsque nous affections le résultat de la commande `ls` à une variable. Si on avait voulu afficher le contenu de cette variable en conservant les retours à la ligne, il aurait fallu empêcher le shell de procéder à la séparation des arguments et à leur remise en forme, séparés par des espaces. On aurait donc encadré la variable par des guillemets :

```
$ var=$(ls /dev/hda1*)
$ echo $var
/dev/hda1 /dev/hda10 /dev/hda11 /dev/hda12 /dev/hda13 /dev/hda14
/dev/hda15 /dev/hda16
$ echo "$var"
/dev/hda1
/dev/hda10
/dev/hda11
/dev/hda12
/dev/hda13
/dev/hda14
/dev/hda15
/dev/hda16
$
```

Nous avons vu que l'interprétation du paramètre spécial `$@` lorsqu'il est encadré par des guillemets est particulière. À ce propos, nous avons indiqué qu'il vaut généralement mieux utiliser `"$@"` que `$*`, mais cela est également vrai pour des paramètres isolés. Lorsqu'un script emploie une variable dont la valeur est configurée par l'utilisateur (paramètres positionnels, variable d'environnement, etc.), la prudence envers son contenu est de rigueur, car des caractères blancs peuvent s'y trouver, perturbant son interprétation par le shell. Ainsi, on préférera encadrer systématiquement ces paramètres par des guillemets, comme cela a été fait dans le script `rm_secure.sh` du précédent chapitre, et plus particulièrement pour sa variable `$sauvegarde_rm`.

En conclusion, la protection par des guillemets est la plus utilisée, peut-être parce qu'elle rappelle la manipulation des chaînes de caractères dans les langages comme le C. Elle est très commode, car elle conserve l'évaluation des variables, tout en préservant la mise en forme de l'expression. On emploie la protection par apostrophes lorsque le respect strict du contenu d'une chaîne est nécessaire. Il faut faire attention de ne pas confondre cette protection avec l'encadrement par des apostrophes inverses ` ` que l'on rencontre dans une forme obsolète d'invocation de commande, qu'il faut remplacer de préférence par `$( )`.

Nous encourageons vivement le lecteur à expérimenter les diverses formes de protection, en incluant dans ses chaînes des caractères spéciaux (`$`, `&`, `<`, `"`, etc.). Un exemple classique consiste à essayer de créer un programme capable d'écrire son propre code source sur sa sortie standard. Il en existe des versions dans de nombreux langages, et la création d'un tel script pour le shell est un défi amusant, essentiellement en raison de l'impossibilité d'inclure une apostrophe, même protégée par un backslash dans une expression encadrée par des apostrophes.

Le script suivant résout le problème en utilisant l'option `-e` de la commande `echo` qui permet de fournir le code ASCII d'un caractère en octal. L'option `-n` de cette commande est également employée pour éviter le retour à la ligne systématique. L'algorithme utilisé ici est décrit dans [HOFSTADTER 1985] *Gödel, Escher, Bach – Les Brins d'une guirlande éternelle*. On se reportera à ce livre fameux de Douglas Hofstadter pour comprendre en détail l'intérêt d'un tel mécanisme.

auto.sh :

```
 1   #! /bin/sh
 2   function quine () {
 3     echo -n "$@"
 4     echo -ne "\047"
 5     echo -n "$@"
 6     echo -e "\047"
 7   }
 8   quine '#! /bin/sh
 9   function quine () {
10     echo -n "$@"
11     echo -ne "\047"
12     echo -n "$@"
13     echo -e "\047"
14   }
15   quine '
```

Pour vérifier que l'exécution donne bien le résultat attendu, nous redirigerons la sortie du programme vers un fichier que nous comparerons avec le script original.

```
$ ./auto.sh
#! /bin/sh
function quine () {
  echo -n "$@"
  echo -ne "\047"
  echo -n "$@"
  echo -e "\047"
}
quine '#! /bin/sh
function quine () {
  echo -n "$@"
  echo -ne "\047"
  echo -n "$@"
  echo -e "\047"
}
quine '
$ ./auto.sh > sortie_auto.txt
$ diff auto.sh sortie_auto.txt
$
```

Il est sûrement possible de créer des scripts qui sont capables d'écrire leur propre code source d'une manière plus concise et plus élégante (je ne considère pas `cat $0` comme

plus élégant !), et le lecteur pourra s'amuser à rechercher les moyens de contourner les limitations d'emploi des apostrophes.

Tableaux

Nous n'avons évoqué, jusqu'à présent, que des variables scalaires. Le shell Bash 2 permet toutefois de manipuler des tableaux qui contiennent autant de données qu'on le souhaite (la seule limitation étant celle de la mémoire du système). Nous avons repoussé la présentation des tableaux à la fin de ce chapitre car ils ne présentent aucune difficulté d'assimilation, et nous allons rapidement voir leurs particularités par rapport aux constructions étudiées jusqu'ici. La notation

```
tableau[i]=valeur
```

a pour effet de définir un emplacement mémoire pour la i^{ème} case du tableau, et de la remplir avec la valeur voulue. La consultation se fera ainsi :

```
${tableau[i]}
```

Les accolades sont obligatoires pour éviter les ambiguïtés avec `${tableau}[i]`.

Les index des tableaux sont numérotés à partir de zéro. En fait, `${tableau[0]}` est identique à `$tableau`. Cela signifie donc que n'importe quelle variable peut être considérée comme le rang zéro d'un tableau qui ne contient qu'une case. On peut aussi ajouter autant de membres qu'on le désire à une variable existante :

```
$ var="valeur originale"
$ echo $var
valeur originale
$ echo ${var[0]}
valeur originale
$ var[1]="nouveau membre"
$ echo ${var[0]}
valeur originale
$ echo ${var[1]}
nouveau membre
$
```

Les notations `${table[*]}` et `${table[@]}` fournissent une liste de tous les membres du tableau. Placées entre guillemets, ces deux notations présentent les mêmes différences que `$*` et `$@` pour les paramètres positionnels. `${#table[i]}` fournit la longueur du i^{ème} membre du tableau, alors que `${#table[*]}`, comme `${#table[@]}`, fournissent le nombre de membres :

```
$ echo ${#var[@]}
2
$ echo ${#var[*]}
2
$ echo ${#var[0]}
16
```

```
$ echo ${#var[1]}
14
$
```

Il n'est pas obligatoire de déclarer préalablement un tableau, puisque la première affectation d'un membre le définira, mais on peut être amené à le faire pour des raisons de lisibilité. Dans ce cas, l'option `-a` de la fonction interne `declare` permet de préparer l'interpréteur. Il n'est pas non plus nécessaire de définir les membres dans l'ordre, une case non remplie n'existant pas :

```
$ declare -a table
$ echo ${#table[*]}
0
$ table[1]="abc"
$ echo ${#table[*]}
1
$ table[0]="abc"
$ echo ${#table[*]}
2
$
```

Évaluation explicite d'une expression

L'une des grosses différences entre les langages compilés et les langages interprétés est qu'il est généralement possible avec ces derniers de construire de toutes pièces une expression, puis de l'évaluer comme s'il s'agissait d'une partie du programme en cours. Ce mécanisme est très utile dans les langages d'intelligence artificielle comme Lisp.

Le shell propose une fonction nommée `eval` qui permet d'obtenir ce comportement. Elle sert à réclamer au shell une seconde évaluation des arguments qui lui sont transmis. Pour bien comprendre ce que cela signifie, il nous faut procéder par étapes successives. Tout d'abord, on définit une variable `A`, et une variable B contenant une chaîne de caractères dans laquelle on trouve l'expression `$A`.

```
$ A="Contenu de A"
$ B=" A = \$A"
$ echo $B
A = $A
$
```

Lors de l'affectation de `B`, nous avons protégé le caractère `$` pour qu'il ne soit pas vu comme une consultation du contenu de `A`. Examinons à présent ce qui se passe lorsqu'on invoque `eval` avant `echo` :

```
$ eval echo $B
A = Contenu de A
$
```

En fait, lorsque le shell interprète la ligne « `eval echo $B` », il remplace `$B` par le contenu de la variable, puis invoque `eval` en lui passant en arguments « `echo` », « `A` », « `=` » et

« $A ». Le travail d'eval consiste alors à regrouper ses arguments en une seule ligne « echo A = $A » qu'il fait réévaluer par le shell. Celui-ci remplace alors $A par son contenu et affiche le résultat attendu.

Il faut bien comprendre que l'évaluation de $A étant retardée, le contenu de cette variable n'est pas inscrit dans la variable B. Si on modifie A et que l'on interroge à nouveau $B au travers d'un eval, on obtient :

```
$ eval echo $B
A = Contenu de A
$ A="Nouveau contenu"
$ eval echo $B
A = Nouveau contenu
$
```

Les implications sont très importantes car nous sommes à même de construire dynamiquement, au sein d'un script, des commandes dont nous demandons l'évaluation au shell :

```
$ commande="echo \$B"
$ echo $commande
echo $B
$ eval $commande
A = $A
$
```

On peut même imbriquer un appel à eval au sein de la chaîne évaluée :

```
$ commande="eval echo \$B"
$ echo $commande
eval echo $B
$ eval $commande
A = Nouveau contenu
$
```

Le petit script suivant est une représentation très simplifiée de la boucle d'interprétation du shell. Il lit (avec la fonction read que nous verrons dans le prochain chapitre) la saisie de l'utilisateur dans sa variable ligne, et en demande l'évaluation au shell. Si la saisie est vide, le script se termine.

mini_shell.sh :

```
 1   #! /bin/sh
 2
 3   while true ; do
 4     echo -n "? "
 5     read ligne
 6     if [ -z "$ligne" ] ; then
 7       break;
 8     fi
 9     eval $ligne
10   done
```

Son fonctionnement est naturellement très simple, mais néanmoins intéressant :

```
$ ./mini_shell.sh
? ls
affiche_arguments.sh    affiche_arguments_6.sh  ll_1.sh
affiche_arguments_2.sh  calcule_paques.sh       ll_2.sh
affiche_arguments_3.sh  calcule_paques_2.sh     message.news
affiche_arguments_4.sh  extraction_serveurs.sh  mini_shell.sh
affiche_arguments_5.sh  ll.sh                   var_locales.sh
? A=123456
? echo $A
123456
? B='A = $A'
? echo $B
A = $A
? eval echo $B
A = 123456
?   (Entrée)
$
```

Ce genre de boucle, *read-eval-print*, a été rendu célèbre principalement grâce au langage Lisp (qui est quand même l'aîné de Bash de 30 ans !).

Conclusion

Ce chapitre nous a servi à mettre en place les principes de l'évaluation des expressions par le shell. Nous pouvons à présent étudier les constructions propres à la programmation, ainsi que les commodités offertes par Bash pour écrire des scripts puissants.