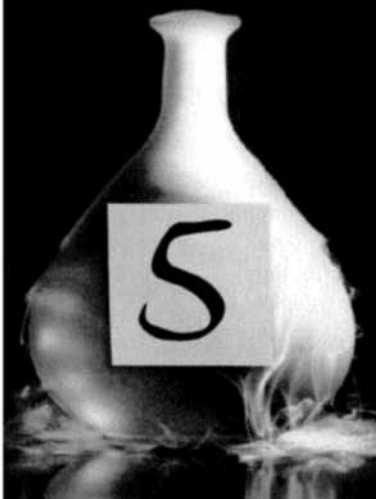


PREMIERS PAS EN SCRIPT SHELL

Tristan Colombo



Certaines commandes shell sont difficiles à concevoir et longues à écrire. Pour pouvoir s'en servir à nouveau sans avoir à perdre une nouvelle fois le temps de réflexion nécessaire à l'élaboration de la commande (ou de la suite de commandes), il pourrait être intéressant de stocker ces commandes dans un fichier pour les appeler par la suite. Il serait encore mieux de pouvoir modifier certaines valeurs sans récrire la commande. C'est ce que permet le shell...

Derrière toute application – bien conçue – ayant une interface graphique, se cachent des appels à des programmes en mode console. Prenons l'exemple de K3b, le logiciel de gravure sous KDE : toutes les actions font appel à des programmes en ligne de commandes (mkisofs, cdrecord, etc.). En connaissant les informations à fournir à ces programmes, vous pouvez donc les utiliser directement en ligne de commandes. Et si vous voulez enchaîner des actions, vous aurez le choix entre cliquer frénétiquement des dizaines de fois sur votre souris, ou écrire un script shell...

1 Qu'est-ce qu'un script shell ?

Tout d'abord, un shell est un environnement dans lequel on peut taper des commandes qui seront exécutées et renverront un résultat. C'est plus précisément un interpréteur de commandes, c'est-à-dire un programme capable de comprendre et d'exécuter un ensemble de commandes paramétrables. Il sert d'intermédiaire entre l'utilisateur et le système d'exploitation d'où son nom de « shell » : la coquille qui entoure le « noyau » du système.

Lorsque vous ouvrez un terminal ou lorsque vous ne vous connectez pas à Linux en mode graphique, vous aboutissez sur le shell de votre système. Un prompt apparaît alors vous signifiant

que l'interpréteur attend que vous saisissiez une commande. Ce prompt est configurable (voir article en page 08) mais ressemble plus ou moins à :

```
login@serveur:~$
```

En tapant une commande derrière ce prompt, si cette dernière est reconnue par le système, vous obtiendrez un résultat. Les commandes shell admettent généralement la même structure : le nom de la commande suivi parfois d'options (précédées du caractère - ou de --) et éventuellement d'arguments. Par exemple, dans la commande `ls -aild /home`, `ls` est la commande, `-aild` représentent les options (`-aild` est un raccourci pour `-a -i -l -d`), et `/home` est l'argument.

Un script shell est un ensemble de commandes contenues dans un fichier et exécutées séquentiellement. En plus des commandes, le shell offre la possibilité d'utiliser des structures de contrôle qui permettront de gérer de manière précise l'exécution des commandes.

2 Quels outils utiliser ?

Pour écrire des scripts shell qui sont donc des fichiers texte, vous allez avoir besoin, bien sûr, d'un éditeur de texte. Mais attention, si vous n'avez jamais « programmé », certains éditeurs de texte sont plus

pratiques que d'autres dans le cadre de l'écriture de lignes de code. Il est bien plus simple d'utiliser un éditeur possédant une coloration syntaxique des instructions du shell. La coloration syntaxique consiste à afficher les différents « mots » de votre programme de différentes couleurs en fonction de leur catégorie (mot-clé, valeur numérique, opérateur arithmétique, etc.).

Il existe de nombreux éditeurs mettant en place cette fonctionnalité. Ce qui les différencie, c'est le nombre de langages qu'ils sont capables de reconnaître, leur capacité de personnalisation et leur difficulté d'emploi. Nous pouvons ici citer `gedit`, `kedit` et `nedit` qui sont très simples à employer. A contrario, `vim` [1][2] ou `emacs` seront un peu plus difficiles à prendre en main (mais offrent des possibilités de configuration bien supérieures). Tous ces éditeurs sont disponibles dans les dépôts des diverses distributions Linux et vous n'aurez aucun mal à les installer (toutes les distributions contiennent au moins un éditeur qui sera installé par défaut).

Le fichier de configuration de votre shell est un fichier caché dont le nom dépend du shell que vous utilisez (voir l'article d'introduction en page 04 du présent magazine). Je considérerai dans cet article que vous utilisez le shell `bash`, installé par défaut sur de

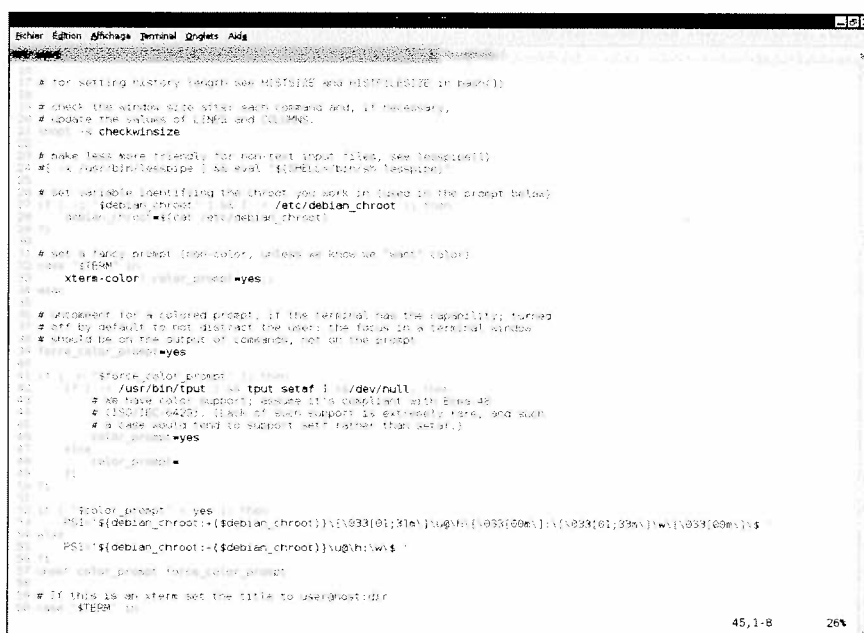


Fig. 1 : Coloration syntaxique du fichier .bashrc dans l'éditeur vim.

nombreuses distributions. Le fichier de configuration de ce shell, le fichier .bashrc, se trouve dans votre répertoire personnel et il est lui-même un fichier de script shell. Pour avoir un aperçu de coloration syntaxique, il vous suffit donc d'ouvrir ce fichier dans votre éditeur de texte. La figure 1 montre le contenu du fichier .bashrc dans Vim.

3 Premier script

Dans cette partie, nous allons voir comment réaliser notre premier script. Nous allons donc bien sûr devoir utiliser l'un des éditeurs vus précédemment pour insérer des commandes, mais nous devons également pouvoir exécuter le fichier.

3.1. Exécution de commandes shell

Pour exécuter une commande shell, il existe deux alternatives : soit lancer un shell en lui donnant en paramètre le nom d'un fichier contenant des commandes à exécuter, soit indiquer directement dans le fichier quelle est la commande (le shell) qui permettra de l'interpréter.

La première solution est la plus simple à mettre en œuvre. Une fois votre fichier tapé et enregistré (nous l'appellerons monScript.sh), vous n'avez plus qu'à lancer la commande :

```
bash monScript.sh
```

La deuxième solution est un tout petit peu plus longue à mettre en place, mais elle est bien plus pratique à l'usage. Dans un premier temps, il faut indiquer sur la première ligne du fichier de script quel est l'interpréteur qui permettra d'exécuter ces commandes. Cette ligne possède une syntaxe particulière : les caractères #! suivis du chemin absolu (depuis la racine) vers l'interpréteur. Dans le cas du bash, nous aurons comme première ligne du fichier monScript.sh :

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

Si vous utilisez un autre shell, vous pourrez obtenir le chemin absolu vers votre interpréteur en utilisant la commande which suivie du nom de votre shell.

Dans un deuxième temps, il faudra rendre ce script exécutable (attention à la gestion des droits utilisateur (u), groupe (g) et autres (o)). Ici j'utilise la commande chmod sur le fichier

monScript.sh pour donner les droits d'exécution à l'utilisateur (moi) :

```
chmod u+x monScript.sh
```

Ces opérations effectuées, il ne vous reste alors plus qu'à taper le nom de votre fichier dans un terminal pour que celui-ci soit exécuté...

3.2. Un petit « hello world »

Pour ce premier exemple, nous allons utiliser une des commandes les plus simples du shell : la commande d'affichage echo. Cette commande admet pour paramètre une chaîne de caractères qu'elle imprimera sur le périphérique de sortie standard (l'écran par défaut). Nous pouvons utiliser cette commande en shell interactif en tapant directement après le prompt :

```
echo "hello world"
```

Pour utiliser cette même commande dans un script, il faut ouvrir un fichier texte (que nous appellerons ici hello.sh) et y taper :

```
01: #!/usr/bin/bash
02:
03: # Nous allons afficher un message !
04: echo "hello world"
```

Il faut noter que les caractères suivant un caractère # ne sont pas interprétés (sauf cas spécial du #! de la première ligne). Ainsi la phrase de la troisième ligne constitue un commentaire permettant d'indiquer à un lecteur du code ce que l'on a souhaité réaliser. Ce lecteur peut être un développeur qui veut améliorer votre script ou bien tout simplement vous-même qui allez reprendre un script et le modifier quelques temps après l'avoir créé. Il est essentiel de bien commenter un code pour pouvoir le maintenir, c'est-à-dire le modifier pour corriger des bogues ou pour l'améliorer. Notez que les commentaires ne commencent pas forcément une ligne. En ligne 4, nous aurions pu écrire :

```
04 : echo "hello world" # ceci est un
commentaire
```

Les lignes vides ne sont pas interprétées non plus : utilisez-les pour aérer votre code et le rendre plus lisible

en regroupant les lignes de même « fonction ». Comme dans un texte, vous aurez ainsi des « paragraphes » permettant de faciliter la lecture.

Pour tester le script, n'oubliez pas de rendre votre fichier exécutable, puis lancez votre script simplement à l'aide de la commande :

```
hello.sh
```

Pour l'instant, rien de bien excitant ; nous avons simplement placé notre commande dans un fichier et celle-ci est exécutée lors de l'appel au fichier. Ce qui serait déjà plus intéressant, ce serait de pouvoir modifier une partie de la chaîne de caractères, la rendre variable...

4 Les variables

Les variables sont des éléments très utiles permettant de conserver et modifier des valeurs de différents types.

4.1. Qu'est-ce qu'une variable ?

Une variable pourrait être représentée par une boîte avec une étiquette. Lorsque je parle du nom inscrit sur l'étiquette, je fais référence au contenu de ma boîte. Si je place une olive dans ma boîte et que j'appelle la boîte « zimboum », quand je dirai « Tiens, peux-tu me donner un zimboum ? », tout le monde traduira instantanément par « Tiens, peux-tu me donner une olive ? ». Et si dans ma boîte je place deux olives, alors lorsque je dirai « zimboum » je ne ferai plus référence à une olive mais à deux olives.

Pour revenir dans un cadre un peu plus formel, une variable est définie par un identifiant (son nom) et une valeur (son contenu). Comme le shell est sensible à la casse (différence majuscules/minuscules), il est d'usage de n'utiliser que des caractères minuscules pour nommer ses variables. Les noms en majuscules sont réservés aux variables d'environnement. Mais tout d'abord, comment définir une variable ?

4.2. Définir et utiliser une variable

La définition se fait de manière très naturelle comme en mathématiques : identifiant=valeur (attention de ne surtout pas mettre d'espace entre la fin du nom de la variable et le signe égal). La valeur pourra bien sûr être un nombre, une lettre, une chaîne de caractères (encadrée par des guillemets «...» ou des quotes '...'), etc.

Pour utiliser la valeur contenue dans une variable, il faudra la faire précéder du caractère \$. Ainsi, si la variable a vaut 12, pour accéder à la valeur de a je devrai écrire \$a.

Reprenons notre exemple précédent où le texte à afficher sera cette fois placé dans une variable :

```
01: #!/usr/bin/bash
02:
03: msg = "hello world"
04: echo $msg
```

Vous obtiendrez exactement le même résultat que précédemment car, lors de l'exécution, \$msg sera remplacé par la valeur contenue dans la variable msg, c'est-à-dire « hello world ».

Il est ensuite possible d'exécuter des opérations sur les variables. Si les variables sont des entiers on pourra utiliser les opérateurs arithmétiques classiques, plus % pour le reste de la division entière et ** pour la puissance. La syntaxe est alors un peu particulière et il y a deux manières de faire :

- soit on encadre l'opération par \$((...)) ce qui donne par exemple : a=\$((3+4)) ;
- soit on utilise la commande let "...": let "a=3+4".

Dans le cas des chaînes de caractères, pour réaliser une concaténation (coller des chaînes bout à bout), il suffit de mettre les variables côte-à-côte. Par exemple, si a="Linux" et b=" Pratique", alors, on peut écrire msg=\$a\$b et la variable msg contiendra la valeur « Linux Pratique ».

Voici un tableau récapitulatif indiquant la valeur d'une variable après des opérations d'exemple :

Commande	Valeur de la variable var (résultat de : echo \$var)
var=3+4	3+4
let "var=3+4"	7
var=\$((3+4))	7
a="Linux" var=\$a Pratique"	Linux Pratique

Tableau 1

4.3. Les variables d'environnement

J'indiquais en introduction que les variables d'environnement étaient notées en majuscules. Un article de ce magazine en parlant déjà (voir en page 04), je ne m'attarderai pas sur ce type de variables. Sachez simplement que pour avoir accès à leur valeur, vous devrez préfixer également leur nom par le caractère \$.

4.4. Variables spéciales

Il existe un certain nombre de variables spéciales qui sont initialisées au lancement de chaque script. Voici un tableau les résumant :

Tableau 2.

Grâce à ces variables on peut déjà commencer à écrire des scripts plus intéressants puisque l'utilisateur pourra transmettre des données au script sans pour autant avoir à modifier le script. Voici un exemple simple d'utilisation de quelques-unes de ces variables :

```
01: #!/bin/bash
02:
03: echo "Le script s'appelle" $0
04: echo "Il y a" $# "arguments."
05: echo "La liste de ces arguments est :"
06: echo $*
07: echo "Le premier argument est :" $1
08: echo "Le deuxième argument est :" $2
```

Lors de l'exécution de ce script, si vous lui passez en paramètres quelques valeurs, vous obtiendrez l'affichage suivant :

Variable	Description
\$0	Le nom du script (dans notre exemple précédent, la valeur est ./hello.sh si le script est appelé depuis son répertoire de stockage).
\$1, \$2, ...	Les arguments passés au script : \$1 est le premier argument, etc.
\$*	La liste de tous les arguments passés au script, séparés par un espace.
\$#	Le nombre d'arguments passés au script.
\$?	Le code de retour de la dernière commande exécutée. Toutes les commandes shell renvoient une valeur : 0 lorsque la commande s'est exécutée correctement et une valeur d'erreur sinon. Par exemple, après un appel à <code>ls</code> , <code>\$?</code> contiendra 0. Par contre, après un appel à <code>ls /zimboum</code> , <code>\$?</code> contiendra la valeur 2 (on suppose que le répertoire /zimboum n'existe pas, bien sûr).
!	Le numéro de processus de la dernière commande lancée en tâche de fond.
\$\$	Le numéro de processus du script lui-même.

Tableau 2

```
login@serveur:~$ variables.sh arg1 arg2 arg3
Le script s'appelle ./variables.sh
Il y a 3 arguments.
La liste de ces arguments est :
arg1 arg2 arg3
Le premier argument est : arg1
Le deuxième argument est : arg2
```

Ici l'utilisateur transmet des données au lancement du script. Mais il est possible qu'une information soit requise par le script au milieu de son traitement. Il faut alors utiliser une commande pour permettre à l'utilisateur de « reprendre la main » et de pouvoir saisir des données.

5 Interaction avec l'utilisateur : la commande read

La commande `read` permet de lire des données au clavier et de les stocker dans une variable dont le nom est spécifié à la suite de la commande : `read nomVariable`. La variable contenant la saisie utilisateur sera alors utilisée comme n'importe quelle variable en la préfixant par le caractère `$` pour avoir accès à sa valeur. Voici un exemple de « hello world » amélioré utilisant la commande `read` :

```
01: #!/bin/bash
02:
03: echo "Comment vous appelez-vous ?"
04: read nom
05: echo "hello" $nom
```

En ligne 4, l'exécution du script sera suspendue jusqu'à ce que l'utilisateur ait saisi un texte validé par la touche [Entrée]. Le texte sera alors stocké dans la variable `nom` et l'exécution du script reprendra en ligne 5 pour afficher le message.

La commande `read` permet également de lire plusieurs variables : il suffit de spécifier plusieurs noms de variables à la suite de l'appel à la commande et, lors de la saisie de l'utilisateur, le caractère espace sera utilisé pour délimiter les valeurs. En modifiant l'exemple précédent, on obtiendrait :

```
01: #!/bin/bash
02:
03: echo "Comment vous appelez-vous (prenom nom)?"
04: read prenom nom
05: echo "Votre prénom : " $prenom
06: echo "Votre nom : " $nom
```

Lors de l'exécution, si vous saisissez deux mots, le premier sera stocké dans `prenom` et le second dans `nom`. Attention : si vous saisissez plus d'un mot, le premier sera bien stocké dans `prenom` mais tous les autres seront stockés dans `nom`. Exemple de déroulement :

```
login@serveur:~$ nom.sh
Comment vous appelez-vous (prenom nom)?
Linus Torvalds Encore_un_mot
Votre prénom : Linus
Votre nom : Torvalds Encore_un_mot
```

Enfin, la commande `read` peut être utilisée pour réaliser une pause dans un programme en demandant à l'utilisateur d'appuyer sur [Entrée] pour continuer. Il s'agit en fait de l'usage classique de la commande, mais comme on ne souhaite pas récupérer la saisie dans une variable, on ne précise pas de nom de variable :

```
echo "Appuyer sur <Return> pour continuer..."
read
```

6 Les structures conditionnelles

Nous avons pu voir précédemment comment construire des scripts rudimentaires et interagir avec le shell. Nous pouvons manipuler des variables mais nous ne pouvons pas encore décider que faire en fonction de la valeur d'une variable. Par exemple, si nous posons une simple question « Voulez-vous continuer (O/N) ? », comment faire comprendre au script qu'il doit effectuer une action si l'utilisateur a saisi « O » et une autre s'il a saisi « N » ? Nous avons

besoin des structures conditionnelles : si telle condition est vraie (ou vérifiée) alors exécute telle action. Cela est rendu possible grâce à l'instruction `if`.

6.1. L'instruction if

L'instruction `if` qui se traduit par « si » s'écrit de la manière suivante :

```
if condition; then
...
fi
```

Le terme `condition` doit être remplacé par la condition à vérifier pour exécuter la ou les lignes représentée(s) par les points de suspension. Pour marquer la fin du bloc de lignes conditionnées, on utilise l'instruction `fi` (if à l'envers). Notez également la présence du point-virgule après le terme `condition` : le point-virgule permet de séparer plusieurs commandes sur une même ligne. L'instruction `then` (qui signifie « alors ») étant une nouvelle instruction, il faut la séparer du début de la ligne par un point-virgule.

Il existe des variantes de cette structure permettant d'exécuter une autre commande au cas où la condition ne serait pas vérifiée. La structure est alors « si condition alors ... sinon ... » :

```
if condition; then
...
else
...
fi
```

Et enfin, on peut imbriquer les tests dans des structures de type « si condition_1 alors ... sinon, si condition_2 alors sinon, si ... etc. » :

```
if condition_1; then
...
elif condition_2; then
...
elif ...etc...
...
fi
```

Mais tant que nous ne savons pas comment construire une condition, nous ne pouvons pas utiliser ces structures...

6.2. La construction des conditions

La condition ou test peut s'écrire de deux manières différentes :

- soit en l'encadrant par des crochets : `[...]` ;
- soit en utilisant la commande `test` ...

Une fois la syntaxe choisie, en conjuguant cette écriture avec un opérateur, on obtient une condition. Nous allons voir les opérateurs les plus fréquemment utilisés dans la suite, mais voici un exemple de condition utilisant les deux syntaxes :

```
[ 2 -lt 3 ]
test 2 -lt 3
```

Ces deux lignes sont identiques et testent si 2 est inférieur ou égal à 3. Dans tout test une condition vérifiée renverra 0 et sinon, un entier quelconque différent de 0. En utilisant ce test simple nous pouvons écrire un script effectuant un test sur le nombre d'arguments fournis :

```
01: #!/bin/bash
02:
03: if [ $# -lt 2 ]; then
04:   echo "Erreur : Vous n'avez pas
    fourni assez d'arguments"
05: else
06:   echo "OK : " $# "arguments"
07: fi
```

Si le nombre d'arguments est inférieur à 2 (ligne 3), alors on affiche un message d'erreur (ligne 4) et sinon on affiche le nombre d'arguments (ligne 6).

Pour pouvoir exploiter pleinement les structures conditionnelles, il faut savoir quelles conditions on va pouvoir écrire. Cela passe par la connaissance des différents opérateurs.

6.2.1. Les opérateurs logiques

Les opérateurs logiques permettent de composer des conditions plus « complexes » à l'aide des opérateurs booléens ET, OU et NON.

L'opérateur NON est un opérateur unaire : il n'accepte qu'un seul argument dont il va inverser la valeur de retour. Si une condition est vraie alors elle sera fausse et inversement. Le symbole de cet opérateur est `!`.

Si vous voulez tester si 4 est supérieur ou égal à 3 en ne connaissant que l'opérateur `-lt` (inférieur à), il faudra prendre la négation de la condition : `[ ! 4 -lt 3 ]`.

Les opérateurs ET et OU sont des opérateurs binaires permettant de composer des conditions : « si condition_1 est vraie ET/OU si condition_2 est vraie, alors ... ». On utilise `&&` pour l'opérateur ET et `||` pour l'opérateur OU. Ces opérateurs peuvent être combinés à l'infini.

Pour tester si une variable `var` est inférieure à 4 et inférieure à une variable `var_2`, on utilisera le test : `[ $var -lt 4 ] && [ $var -lt $var_2 ]`.

6.2.2. Les opérateurs arithmétiques

Nous avons déjà vu un opérateur arithmétique avec `-lt` pour « inférieur à » (*lower than*). Les autres opérateurs sont construits sur le même modèle :

- `-gt` pour plus grand que (*greater than*)
- `-eq` pour égal à (*equal*)
- `-ne` pour différent de (*not equal*)
- `-ge` pour plus grand ou égal à (*greater or equal*)
- `-le` pour plus petit ou égal à (*lower or equal*).

Nous avons déjà étudié des exemples utilisant de tels opérateurs. Toutefois, sachez que si vous voulez effectuer des tests sur des chaînes de caractères vous devrez utiliser des opérateurs différents : `==` pour l'égalité et `!=` pour la différence. Par exemple pour savoir si la variable `var` contient la valeur « linux » (attention à bien encadrer le nom de la variable par des guillemets pour indiquer qu'il s'agit d'une chaîne de caractères) : `[ "$var" == "linux" ]`.

6.2.3. Les opérateurs de test de fichier

Pour finir, les opérateurs de test de fichier permettent d'obtenir très facilement des informations sur les fichiers. À part lors de la comparaison de fichiers, il s'agit d'opérateurs unaires portant sur le nom d'un fichier. Voici quelques-uns de ces opérateurs :

- `-e` pour vérifier l'existence d'un fichier (*exists*) ;
- `-f` pour vérifier l'existence d'un fichier et qu'il s'agisse bien d'un fichier et pas d'un répertoire par exemple (*file*) ;

- -d pour vérifier l'existence d'un répertoire (*directory*) ;
- -r pour vérifier si un fichier est accessible en lecture (*readable*) ;
- -w pour vérifier si un fichier est accessible en écriture (*writable*) ;
- -x pour vérifier si un fichier a les droits d'exécution (*executable*).

Pour une liste complète des opérateurs disponibles pour la création de conditions, dans un shell tapez la commande :

```
info coreutils 'test invocation'
```

7 Un exemple récapitulatif

Pour utiliser un peu tout ce que nous avons pu voir dans cet article, je vous propose un exemple récapitulatif. Nous allons écrire un script `menu_file.sh` affichant le menu suivant :

1. Afficher la liste des fichiers
2. Afficher les droits d'un fichier

Si l'utilisateur choisit l'option 1, on exécute une commande `ls` et s'il choisit l'option 2, on lui demande de saisir un nom de fichier et on affiche les droits du fichier. Le script pourra être utilisé en mode interactif si aucun argument ne lui est donné au lancement (affichage du menu) ou en mode automatique si l'utilisateur indique en argument l'option du menu choisie et éventuellement le nom du fichier cible (option 2). Voici le code :

```
01: #!/bin/bash
02:
03: # Contrôle des arguments
04: menu=0
05: file=""
06: if [ $# -eq 1 ] || [ $# -eq 2 ]; then
07:     menu=$1
08: fi
09: if [ $# -eq 2 ]; then
10:     file=$2
11: fi
12: if [ $# -gt 2 ]; then
13:     echo "Syntaxe: menu_file.sh [<menu> [<fichier>]]"
14:     exit 1
15: fi
16:
17: # Affichage du menu
18: if [ $menu -eq 0 ]; then
19:     echo "1. Afficher la liste des fichiers"
20:     echo "2. Afficher les droits d'un fichier"
21:     echo "Votre choix ?"
22:     read menu
23: fi
24:
25: # Gestion des actions
26: if [ $menu -eq 1 ]; then
27:     # Option 1 : Affichage des fichiers
28:     ls
29: elif [ $menu -eq 2 ]; then
30:     # Option 2 : Affichage des droits d'un fichier
31:     if [ "$file" == "" ]; then
```

```
32:         echo "Donnez le nom du fichier :"
33:         read file
34:     fi
35:     if [ ! -e $file ]; then
36:         echo "Le fichier '$file' n'existe pas !"
37:         exit 2
38:     fi
39:     if [ -r $file ]; then
40:         echo "Le fichier est accessible en lecture"
41:     fi
42:     if [ -w $file ]; then
43:         echo "Le fichier est accessible en écriture"
44:     fi
45:     if [ -x $file ]; then
46:         echo "Le fichier a les droits d'exécution"
47:     fi
48: else
49:     echo "L'option '$menu' n'est pas définie !"
50:     exit 3
51: fi
52:
53: # Fin du script
54: exit 0
```

On commence par récupérer et contrôler le nombre d'arguments dans les lignes 3 à 15. Le choix du menu et le nom du fichier sont stockés dans des variables `menu` et `file`. Ces variables sont initialisées dans les lignes 4 et 5 de manière à ce que si le script est appelé sans argument on puisse quand même tester leur valeur. Si plus de deux arguments sont donnés au script, on affiche un message d'erreur (ligne 13) et on quitte le script avec le code d'erreur 1 (ligne 14). Le code d'erreur que nous renvoyons est celui qui pourra être récupéré à l'aide de la variable spéciale `$?`.

Ensuite, dans les lignes 17 à 23, on affiche le menu si aucun choix n'a été fait à l'aide d'un argument lors de l'appel du script. En fonction du choix, on affiche la liste des fichiers (ligne 28), on demande le nom d'un fichier (lignes 31 à 34) et, s'il existe, on affiche ses droits pour l'utilisateur (lignes 39 à 47), ou on affiche un message d'erreur (lignes 49 à 50). Enfin, en ligne 54, on quitte le script en renvoyant un code 0 pour indiquer qu'il s'agit de la fin normale du script et qu'aucune erreur n'est apparue.

Conclusion

Cet article constitue une introduction au script shell avec les notions fondamentales permettant d'écrire de petits scripts. Il est important d'avoir bien compris et assimilé les notions présentées ici avant de lire la suite de cet article intitulée « Scripts shell : notions avancées », présent dans les pages suivantes. ■

Références

- [1] COLOMBO (Tristan), « Vim : votre nouvel Environnement de Développement Intégré », GNU/Linux Magazine n°110, p. 68 à 75, novembre 2008.
- [2] COLOMBO (Tristan), « Vim : votre nouvel Environnement de Développement Intégré (notions avancées) », GNU/Linux Magazine n°111, p. 77 à 85, décembre 2008.