

SCRIPTS SHELL : NOTIONS AVANCÉES

Tristan Colombo

Le shell permet de créer des scripts relativement complexes en utilisant des boucles ou même en définissant des fonctions. Comment les écrire ?

Cet article fait suite à l'article « Premiers pas en script shell ». Après avoir vu les bases de l'écriture des scripts shell, nous allons nous pencher sur des notions moins triviales mais offrant beaucoup plus de possibilités...

1 Une structure particulière : le case

Nous avons vu dans l'article précédent, les structures conditionnelles utilisant l'instruction `if`. Pour les structures à choix multiples, il est possible d'utiliser l'architecture « si ... sinon si ... sinon si ... etc. ». Si ces tests ont tous lieu sur une même variable `var`, on obtiendra une structure du type :

```
if [ $var -eq cond_1 ]; then
...
elif [ $var -eq cond_2 ]; then
...
elif ...; then
...
else
...
fi
```

Dans ce cas particulier, il existe une structure spéciale permettant de ne pas répéter les séquences `if/elif` : le `case`. Cette structure va permettre d'identifier une variable pour laquelle on va proposer plusieurs valeurs qui serviront d'aiguillage pour l'exécution d'instructions. Au niveau syntaxique, la valeur de la variable pour chaque cas sera suivie d'une parenthèse

fermante et le bloc d'instructions à exécuter pour ce cas se situera entre la parenthèse fermante et les caractères `;;` (une instruction vide signifiant la fin du bloc).

Appliquons cette définition à un exemple : la variable `choix` peut valoir 1, 2, 3, ou 4 et l'action à effectuer sera simplement d'afficher le chiffre correspondant en toutes lettres pour les valeurs 1 à 3 et d'afficher le message « coucou » pour la valeur 4. Dans tous les autres cas, nous afficherons un message d'erreur. Voici le code correspondant à cet exemple :

```
01: #!/bin/bash
02:
03: echo "Donnez un chiffre entre 1 et
04: 4 :"
```

```
04: read choix
05:
06: case $choix in
07:   1) echo "un";;
08:   2) echo "deux";;
09:   3) echo "trois";;
10:   4) echo "coucou";;
11:   *) echo "ERREUR !"
```

```
12: esac
```

Dans les lignes 3 et 4, on commence par interroger l'utilisateur pour qu'il saisisse une valeur. Ensuite, les lignes 6 à 12 correspondent à la structure de choix. Le dernier cas, en ligne 11, correspond au choix par défaut : si l'utilisateur saisit autre chose qu'un entier entre 1 et 4, c'est cette action qui sera effectuée. Enfin, en ligne 12, on referme le bloc `case` en inscrivant le nom de l'instruction à l'envers : `esac`.

L'utilisation du caractère `*` pour indiquer tous les cas différents de ceux traités (on dira « filtrés ») auparavant vient du fait qu'il s'agit en fait d'expressions régulières. Je ne détaillerai pas ici l'usage des expressions régulières, mais sachez que le caractère `*` remplace n'importe quelle suite de caractères (y compris aucun caractère). Ainsi, le mot (on parlera de « motif ») « `b*r` » pourra correspondre à « `bonjour` », « `bar` », « `br` », etc.

Imaginons maintenant que plusieurs cas déclenchent la même action. Il serait idiot de récrire à chaque fois les mêmes lignes de code... Vous pouvez grouper plusieurs cas à l'aide du caractère `|` qui signifie « ou ». Supposons que nous voulions modifier l'exemple précédent pour indiquer la saisie d'un chiffre pair (2 ou 4) ou la saisie d'un chiffre impair (1 ou 2). Le code des lignes 6 à 12 deviendrait alors :

```
06: case $choix in
07:   1 | 3) echo "Le chiffre est :"
```

```
08:           echo "impair";;
09:   2 | 4) echo "pair";;
10:   *) echo "ERREUR !"
```

```
11: esac
```

La seule chose notable dans ce code se situe dans les lignes 7 et 8 : comme il n'y a pas le symbole de fin de bloc à la fin de la ligne 7, les lignes 7 et 8 seront exécutées (la fin de bloc est bien signifiée par `;;` à la fin de la ligne 8).

Nous avons fait le tour des structures conditionnelles. Il nous reste maintenant à aborder les structures de boucle.

SCRIPTS SHELL : NOTIONS AVANCÉES

Tristan Colombo

Le shell permet de créer des scripts relativement complexes en utilisant des boucles ou même en définissant des fonctions. Comment les écrire ?

Cet article fait suite à l'article « Premiers pas en script shell ». Après avoir vu les bases de l'écriture des scripts shell, nous allons nous pencher sur des notions moins triviales mais offrant beaucoup plus de possibilités...

1 Une structure particulière : le case

Nous avons vu dans l'article précédent, les structures conditionnelles utilisant l'instruction `if`. Pour les structures à choix multiples, il est possible d'utiliser l'architecture « si ... sinon si ... sinon si ... etc. ». Si ces tests ont tous lieu sur une même variable `var`, on obtiendra une structure du type :

```
if [ $var -eq cond_1 ]; then
...
elif [ $var -eq cond_2 ]; then
...
elif ...; then
...
else
...
fi
```

Dans ce cas particulier, il existe une structure spéciale permettant de ne pas répéter les séquences `if/elif` : le `case`. Cette structure va permettre d'identifier une variable pour laquelle on va proposer plusieurs valeurs qui serviront d'aiguillage pour l'exécution d'instructions. Au niveau syntaxique, la valeur de la variable pour chaque cas sera suivie d'une parenthèse

fermante et le bloc d'instructions à exécuter pour ce cas se situera entre la parenthèse fermante et les caractères `;;` (une instruction vide signifiant la fin du bloc).

Appliquons cette définition à un exemple : la variable `choix` peut valoir 1, 2, 3, ou 4 et l'action à effectuer sera simplement d'afficher le chiffre correspondant en toutes lettres pour les valeurs 1 à 3 et d'afficher le message « coucou » pour la valeur 4. Dans tous les autres cas, nous afficherons un message d'erreur. Voici le code correspondant à cet exemple :

```
01: #!/bin/bash
02:
03: echo "Donnez un chiffre entre 1 et 4 : "
04: read choix
05:
06: case $choix in
07:   1) echo "un";;
08:   2) echo "deux";;
09:   3) echo "trois";;
10:   4) echo "coucou";;
11:   *) echo "ERREUR !";;
12: esac
```

Dans les lignes 3 et 4, on commence par interroger l'utilisateur pour qu'il saisisse une valeur. Ensuite, les lignes 6 à 12 correspondent à la structure de choix. Le dernier cas, en ligne 11, correspond au choix par défaut : si l'utilisateur saisit autre chose qu'un entier entre 1 et 4, c'est cette action qui sera effectuée. Enfin, en ligne 12, on referme le bloc `case` en inscrivant le nom de l'instruction à l'envers : `esac`.

L'utilisation du caractère `*` pour indiquer tous les cas différents de ceux traités (on dira « filtrés ») auparavant vient du fait qu'il s'agit en fait d'expressions régulières. Je ne détaillerai pas ici l'usage des expressions régulières, mais sachez que le caractère `*` remplace n'importe quelle suite de caractères (y compris aucun caractère). Ainsi, le mot (on parlera de « motif ») « `b*r` » pourra correspondre à « bonjour », « bar », « br », etc.

Imaginons maintenant que plusieurs cas déclenchent la même action. Il serait idiot de récrire à chaque fois les mêmes lignes de code... Vous pouvez grouper plusieurs cas à l'aide du caractère `|` qui signifie « ou ». Supposons que nous voulions modifier l'exemple précédent pour indiquer la saisie d'un chiffre pair (2 ou 4) ou la saisie d'un chiffre impair (1 ou 3). Le code des lignes 6 à 12 deviendrait alors :

```
06: case $choix in
07:   1 | 3) echo "Le chiffre est :";;
08:   *) echo "impair";;
09:   2 | 4) echo "pair";;
10:   *) echo "ERREUR !";;
11: esac
```

La seule chose notable dans ce code se situe dans les lignes 7 et 8 : comme il n'y a pas le symbole de fin de bloc à la fin de la ligne 7, les lignes 7 et 8 seront exécutées (la fin de bloc est bien signifiée par `;;` à la fin de la ligne 8).

Nous avons fait le tour des structures conditionnelles. Il nous reste maintenant à aborder les structures de boucle.

2 Les structures de boucle

Une boucle permet de répéter une action (ou un bloc de commandes) un certain nombre de fois. Pour prendre un exemple simple, supposons que vous vouliez afficher les 100 premiers résultats de la table de multiplication par 7. Vous n'allez certainement pas écrire 100 lignes de code et calculer « à la main » les résultats... d'où l'intérêt des boucles ! Le shell propose quatre types de boucles.

2.1. L'instruction while

La première des structures de boucle est composée à l'aide de l'instruction `while` et signifie : « tant que la condition est vérifiée, faire ... ». Dans le cas proposé en exemple, nous allons avoir besoin d'une variable `i` qui sera incrémentée et sera multipliée par 7 à chaque étape. La condition sera donc : « tant que `i` est inférieur ou égal à 100, faire... ». La traduction en script shell donne :

```
01: #!/bin/bash
02:
03: i=1
04:
05: while [ $i -le 100 ]; do
06:   echo "$i*7=$((i*7))"
07:   i=$((i+1))
08: done
```

La variable de boucle, nommée `i`, est initialisée en ligne 3. En ligne 5, nous définissons la condition d'arrêt de la boucle : « tant que `i` est inférieur ou égal à 100 ». Nous pouvons ensuite réaliser l'affichage en ligne 6 et, surtout, il ne faut pas oublier d'incrémenter la variable de boucle en ligne 7.

2.2. L'instruction until

La deuxième structure de boucle est la structure `until` qui se rapproche énormément de la boucle `while`... à la différence près que la condition d'arrêt de la boucle est une « véritable » condition d'arrêt. En effet, `until` signifie « jusqu'à ce que... ». La boucle est donc effectuée jusqu'à ce que la condition soit vérifiée (et non pas comme dans la boucle `while`,

tant que la condition est vérifiée). Pour illustrer ce type de boucle, on peut reprendre l'exemple utilisé précédemment avec le `while` : une seule ligne sera différente (voir ligne en jaune).

```
01: #!/bin/bash
02:
03: i=1
04:
05: until [ $i -gt 100 ]; do
06:   echo "$i*7=$((i*7))"
07:   i=$((i+1))
08: done
```

2.3. L'instruction for

Troisième type de structure de boucle : le `for`. Cette structure s'éloigne un peu des deux types de boucles vus précédemment, car ici il va s'agir de parcourir une liste d'éléments (chaînes de caractères, entiers). Une liste est spécifiée par une suite d'éléments séparés par un espace. Voici un exemple de liste :

```
"chaîne" 1 2 "autre chaîne" 3 4 5
```

Comme nous nous trouvons dans le shell, en utilisant le caractère `*` nous obtiendrons la liste des fichiers du répertoire courant.

Pour revenir à la boucle, nous allons utiliser une variable qui va prendre consécutivement pour valeur les éléments de la liste. La syntaxe sera la suivante : `for variable in liste; do`. Lorsque nous utiliserons la variable, il ne faudra bien sûr pas oublier de préfixer son nom par le caractère `$`. En guise d'exemple, voici comment afficher et compter les fichiers du répertoire courant :

```
01: #!/bin/bash
02:
03: i=1
04:
05: for file in *; do
06:   echo "$i". "$file"
07:   i=$((i+1))
08: done
```

En ligne 5, le caractère `*` va être remplacé par la liste des fichiers. Supposons que le répertoire courant contienne les deux fichiers `test.sh` et `fichier.py`. À l'exécution, la ligne 5 sera équivalente à :

```
for file in "test.sh" "fichier.py"; do
```

À chaque itération, la variable `file` va changer de valeur. Donc à la première itération `$file` vaudra `test.sh`, puis à la seconde itération nous aurons `fichier.py`.

Cette instruction permet de parcourir simplement une liste d'éléments (on en retrouve un équivalent en Python, PHP, Perl, ...).

2.4. Une « boucle » particulière : le select

L'instruction `select` permet de proposer à l'utilisateur de saisir un choix dans une liste... alors pourquoi classer cette instruction avec les instructions de boucle ? Tout simplement parce que le `select` va suivre la même construction que les boucles vues précédemment et qu'une boucle va effectivement être effectuée en demandant indéfiniment à l'utilisateur de faire un choix. Voici un exemple simple :

```
01: #!/bin/bash
02:
03: select choix in "Entrée 1" "Entrée 2" ;
04: do
05:   echo "Vous avez choisi" $choix
06: done
```

Lors de son exécution, ce script va réaliser l'affichage suivant :

```
1) Entrée 1
2) Entrée 2
#?
```

Vous pourrez saisir votre choix après le point d'interrogation et la variable `choix` prendra pour valeur votre choix (à condition que ce dernier fasse partie des choix possibles). Attention : si vous tapez « 1 », la variable `choix` ne vaudra pas « 1 » mais « Entrée 1 ».

Le problème de ce script, tel que nous l'avons écrit... c'est qu'il boucle à l'infini et demandera toujours à l'utilisateur de saisir un choix. Pour arrêter la boucle, nous allons utiliser l'instruction `break`. Le menu ressemblera alors à :

```

01: #!/bin/bash
02:
03: select choix in "Entrée 1" "Entrée 2" "q"; do
04:   if [ "$choix" = "Entrée 1" ]; then
05:     # traitement ...
06:   elif [ "$choix" = "Entrée 2" ]; then
07:     # traitement ...
08:   elif [ "$choix" = "q" ]; then
09:     break
10:   fi
11: done

```

Ainsi, avec ce genre de structure, on assure que la boucle sera exécutée tant que l'utilisateur ne saisit pas un choix valide et que le menu sera ré-affiché jusqu'à ce que l'utilisateur sélectionne le choix 3 (test conditionnel de la ligne 8 et sortie de la boucle select par le break de la ligne 9). Bien sûr, si vous souhaitez interrompre la boucle dès qu'un choix valide a été sélectionné, vous pouvez utiliser une structure de type case et faire des appels à break.

3 Les fonctions

Maintenant que nous avons pu voir comment assembler des lignes de code grâce à des structures de contrôle, nous allons structurer notre code à l'aide de fonctions. Avant de nous lancer dans le vif du sujet et donc d'étudier la syntaxe à employer, je vous propose un petit rappel de ce qu'est une « fonction ».

Une fonction est un ensemble d'instructions paramétrables. Cela signifie que vous allez pouvoir exécuter de nombreuses lignes de code à partir d'une seule instruction et que vous allez pouvoir modifier le comportement de ces lignes à l'aide de variables intégrées dans les lignes de code de la fonction.

Le fait d'utiliser des fonctions permet de synthétiser le code, de le rendre réutilisable, mieux structuré et plus lisible.

3.1. Définition d'une fonction

La définition d'une fonction s'effectue de façon très simple : après le mot-clé `function`, on donne un nom de fonction (un identifiant qui permettra de rappeler la fonction et qui doit être différent d'un nom de commande existant déjà) et on le fait suivre de parenthèses. L'ensemble des instructions intégrées dans la fonction sont incluses dans un bloc délimité par des accolades :

```

function nom_de_ma_fonction()
{
  # instructions
}

```

On ne définit pas les paramètres à l'avance mais on peut les récupérer dans le corps de la fonction (entre les accolades) à l'aide des variables `$1` à `$n` où `$1` correspond au premier paramètre, `$2` correspond au second paramètre, etc.

La variable `$$` indiquera le nombre d'arguments et la variable `*$` contiendra la liste de tous les paramètres. Voici comment déclarer une simple fonction d'affichage qui prend deux paramètres en arguments :

```

01: #!/bin/bash
02:
03: ##
04: # Fonction affichant un message de bienvenue
05: #
06: # $1 : Prénom de la personne à saluer
07: # $2 : Nom de la personne à saluer
08: function hello()
09: {
10:   echo "Salut \"$1\" \"$2"
11: }

```

Les lignes 3 à 7 sont très importantes : ce sont les lignes de commentaires qui expliquent ce que fait la fonction et quels sont les paramètres à lui passer. En ligne 8, nous définissons une fonction qui portera le nom « hello ». Le corps de la fonction, délimité par les accolades, se trouve dans les lignes 9 à 11. En ligne 10, nous affichons un message en utilisant les deux arguments passés en paramètre.

Nous aurions pu (ou plutôt dû), effectuer un test dans la fonction pour s'assurer que cette dernière a bien été appelée avec deux arguments. Pour effectuer un tel test, nous aurions ajouté après la ligne 9 :

```

10:   if [ $# -ne 2 ]; then
11:     echo "Erreur : La fonction hello() attend 2 arguments !"
12:     exit 1
13:   fi

```

3.2. Appel d'une fonction

L'appel à la fonction s'effectuera en spécifiant son nom et en le faisant suivre de la liste des paramètres séparés par un espace (comme un appel à une commande shell). Ainsi, pour appeler la fonction `hello()` définie précédemment, nous invoquerons :

```
hello "Linux" "Pratique"
```

3.3. Récupérer une valeur issue d'une fonction

En général, une fonction renvoie une valeur déterminée en utilisant les paramètres qui lui sont passés en argument. Par exemple, la fonction `f(x)=2x` accepte un argument `x` et renvoie son double. En shell aussi il est possible de récupérer une valeur « renvoyée » par une fonction. J'utilise ici des guillemets pour employer le terme « renvoyer » car, dans un premier temps, nous n'utiliserons pas d'instruction indiquant quelle est la valeur à retourner... En fait, on va capturer l'affichage de la fonction dans une variable. En reprenant l'exemple de la fonction `f(x)`, voici comment écrire un script shell qui va récupérer une valeur issue d'une fonction :

```

01: #!/bin/bash
02:
03: function double()
04: {
05:     echo $((1 * 2))
06: }
07:
08: resultat=$(double 4)
09:
10: echo '2 * 4 =' $resultat

```

La fonction est définie dans les lignes 3 à 6 et on stocke son résultat dans la variable `resultat` en ligne 8. Il n'y a plus qu'à utiliser cette variable en ligne 10 pour utiliser le résultat renvoyé par la fonction.

Ce qu'il faut savoir sur ce mécanisme :

- si vous utilisez plusieurs affichages (emploi de `echo`) à l'intérieur de la fonction, seul le dernier affichage sera capturé ;
- vous pouvez renvoyer n'importe quel type de valeur (nombre ou chaîne de caractères) puisque l'on récupère l'affichage.

Il existe une alternative à cette méthode pour récupérer une valeur issue d'une fonction à l'aide d'une instruction. Il s'agit de l'instruction `return` à laquelle on indique la valeur de retour de la fonction... Toutefois, cette instruction est moins souple que le mécanisme exposé précédemment. En effet, il ne sera possible de renvoyer que des valeurs entières comprises entre 0 et 255 et il faudra utiliser la variable `$?` immédiatement après l'appel de la fonction. Appliquons ce mécanisme sur l'exemple précédent (le code en jaune indique une modification par rapport à la version originale) :

```

01: #!/bin/bash
02:
03: function double()
04: {
05:     return $((1 * 2))
06: }
07:
08: double 4
09: resultat=$?
10:
11: echo '2 * 4 =' $resultat

```

En ligne 5, le résultat n'est plus affiché mais « retourné » à l'aide de l'instruction `return`. En ligne 8, nous appelons la fonction en lui transmettant un paramètre et le résultat, renvoyé dans `$?`, est enregistré dans la variable `resultat` de la ligne 9.

Bien sûr, dans cette solution, à partir de `double 128`, les résultats seront erronés... d'où l'intérêt de la première méthode qui fonctionne tout le temps.

4 Valeur de retour d'un script shell

Nous avons vu dans l'article sur les bases des scripts shell que chaque script renvoyait une valeur permettant de savoir s'il s'était déroulé correctement. Cette valeur peut être interceptée depuis le shell dans la variable spéciale `$?` (attention, il s'agit ici d'une variable récupérée à l'extérieur du script et qui aura pour valeur un entier compris entre 0 et 255). Par convention, un script s'étant correctement exécuté renvoie la valeur 0. La commande permettant de spécifier la valeur de retour est `exit`. Cette instruction va provoquer la sortie du script et les instructions suivantes ne seront pas exécutées. Voici un exemple :

```

01: #!/bin/bash
02:
03: echo "Test de valeur de retour"
04: exit 12
05: echo "Ne s'exécute pas ..."

```

Il s'agit d'un script très simple : la ligne 3 permet d'effectuer un affichage à l'écran et la ligne 4 sort du script en renvoyant la valeur 12 (la ligne 5 ne sera jamais exécutée). L'appel à ce script et la récupération de la valeur de retour depuis une console se font de la manière suivante :

```

login@serveur:~$ bash monScript.sh
Test de valeur de retour
login@serveur:~$ echo $?
12

```

5 Un exemple récapitulatif

Pour achever cette présentation des scripts shell, je vous propose un exemple récapitulatif permettant de mettre en pratique les différentes instructions que nous avons pu voir tout au long de ces deux articles.

Nous allons écrire un script qui va analyser tout le répertoire `/home` à la recherche de fichiers `core` et afficher ensuite un menu proposant leur suppression individuelle. Voici le code du script (qui sera forcément plus long que tout ce que nous avons pu voir jusqu'à présent) :

```

01: #!/bin/bash
02:
03: ##
04: # Affichage du menu
05: #
06: # $1 Liste des fichiers à afficher dans le menu
07: #
08: # Retourne le fichier sélectionné
09: ##
10: function menu()
11: {
12:     select choice in $1 "Quitter"; do
13:         echo $choice

```

```

14:     break
15: done
16: }
17:
18:
19: ##
20: # Recherche la liste des fichiers à supprimer
21: #
22: # $1 Chemin de départ pour la recherche
23: # $2 Nom du fichier à rechercher
24: #
25: # Retourne la liste de fichiers trouvés
26: ##
27: function get_files()
28: {
29:     echo $(find $1 -name "$2")
30: }
31:
32: ##
33: # Demande de confirmation à une action utilisateur
34: #
35: # Retourne 1 si la réponse est o ou O et 0 sinon
36: ##
37: function confirm()
38: {
39:     echo "Êtes-vous sûr de vouloir effacer" $1 "(O/n) ?"
40:     read response
41:     if [ "$response" == "o" ] || [ "$response" == "O" ]; then
42:         return 1
43:     else
44:         return 0
45:     fi
46: }
47:
48: ##
49: # Effectue une pause et attend que l'utilisateur appuie sur
50: # la
51: # touche <Return>
52: ##
53: function pause()
54: {
55:     echo "Appuyez sur <Return> pour continuer..."
56:     read
57: }
58: ##
59: # Boucle principale
60: ##
61: core_list=$(get_files "$1" "$2")
62: while true; do
63:     clear
64:     echo "Liste des fichiers core :"
65:     echo "-----"
66:     choice=$(menu "$core_list")
67:     if [ "$choice" == "Quitter" ]; then
68:         exit 0
69:     else
70:         confirm $choice
71:         if [ $? -eq 1 ]; then
72:             echo "Suppression du fichier..."
73:             rm $choice
74:             if [ $? -ne 0 ]; then
75:                 echo "La suppression n'a pas pu être effectuée"
76:             else
77:                 echo "Le fichier $choice a été supprimé !"
78:                 core_list=$(get_files "$1" "$2")
79:             fi
80:         else
81:             echo "Annulation de la suppression"
82:         fi
83:         pause
84:     fi
85: done

```

Le code du script a été décomposé en cinq parties : quatre fonctions et le code de la boucle principale. Pour suivre l'interprétation du script par le shell, nous allons commencer par analyser les lignes à partir de la ligne 61 où la variable `core_list` récupère les valeurs de la fonction `get_files` appelée avec les deux arguments fournis au script (il n'y a pas de mécanisme de vérification des arguments pour ne pas alourdir l'exemple).

La fonction `get_file` des lignes 19 à 30 effectue un appel à la commande `find` pour rechercher les fichiers et transmettre leur liste.

Retour au script principal : nous entrons ensuite dans une boucle sans fin qui commence en ligne 62 et s'achève en ligne 85. Les lignes 63 à 65 permettent d'obtenir un affichage propre avec effacement du terminal et affichage d'un titre. En ligne 66, nous appelons la fonction `menu` avec pour paramètre la liste des fichiers `core_list` (notez les guillemets de "`$core_list`"... si vous les omettez, vous ne transmettez plus un mais n paramètres et la fonction ne donne pas le résultat attendu).

La fonction `menu`, définie dans les lignes 3 à 16, affiche un menu à partir de la liste de fichiers fournie et renvoie le choix de l'utilisateur.

L'exécution se poursuit ensuite dans le script principal en testant la valeur sélectionnée par l'utilisateur (ligne 67). Si le choix est de quitter le script alors on quitte le script proprement en renvoyant la valeur 0 (ligne 68). Sinon, on demande confirmation du choix (ligne 70) par appel à la fonction `confirm` des lignes 32 à 46.

La confirmation utilisant une instruction `return` (lignes 42 et 44), nous testons la valeur de retour par `?` en ligne 71. Si la suppression est confirmée, nous la rendons effective par un appel à la commande `rm` en ligne 73. Pour savoir si la commande a pu accomplir correctement sa tâche, nous testons ensuite sa valeur de retour en ligne 74.

Si tout s'est déroulé correctement, il faut recalculer la variable `core_list` car un fichier a disparu (méthode naïve car la commande `find` est longue... il serait préférable d'exécuter une commande telle que `sed` permettant de supprimer le nom du fichier directement dans `core_list`).

Finalement, ce script propose son menu et permet de supprimer les fichiers `core` jusqu'à ce que l'utilisateur choisisse l'entrée « Quitter ».

Conclusion

Dans ces deux articles sur les scripts shell, nous avons pu voir comment réaliser simplement des actions relativement complexes grâce au shell. Les scripts permettant d'appeler et d'interagir avec toutes les commandes du shell, les possibilités sont infinies ! Seul le syndrome du bûcheron canadien pourra désormais vous empêcher de gagner du temps !

Note sur le syndrome du bûcheron canadien : un bûcheron coupe du bois avec sa hache lorsqu'un indien l'interpelle : « Pourquoi n'affûtes-tu pas ta hache ? ». Et le bûcheron de répondre « Mais voyons, avec tout le bois que j'ai à couper, vous pensez que j'ai le temps de m'arrêter ? ». À méditer... ■