Sommaire

[modifier]

Programmation de scripts : Une introduction

Programmation de Script: Une introduction
Comment écrire de petits scripts permettant d'automatiser la réalisation de taches répétitives.

[modifier]

Introduction

Vous aurez envie d'écrire un script (petit programme écrit avec un langage simple : *shell*, *perl* ou autre) dès que vous aurez tapé dans un terminal quatre fois la même série de commandes et que vous vous apercevrez que vous êtes amené à le refaire de nombreuses fois.

Un script est une suite d'instructions élémentaires qui sont éxécutées de façon séquencielle (les unes après les autres) par le langage de script. Dans cet article nous nous limiterons à

l'utilisation du *shell* comme langage, et en particulier à du *shell* bash. En guise de première introduction, vous pouvez lire ce qui concerne les commandes du shell dans l'article Le Shell et les Commandes. Attention, n'espérez pas que le présent document constitue un manuel complet de programmation ! C'est une courte introduction qui nous l'espérons, vous permettra d'écrire de petits scripts qui vous rendront de précieux services.

[modifier]

Notions de base

[modifier]

Mon premier script

Pour commencer, il faut savoir qu'un script est un fichier texte standard pouvant être créé par n'importe quel éditeur : vi, emacs, kedit, gnotepad, ou autre. D'autre part, conventionnellement, un script commence par une ligne de commentaire contenant le nom du langage à utiliser pour interpréter ce script, soit dans notre cas : /bin/sh (on parle alors de "script shell"). Donc un script shell élémentaire pourrait être :

```
#!/bin/sh
```

> **Note :** "#!" se prononce « *she bang* », soit « chi-bang ».

Évidemment un tel script ne fait rien ! Changeons cela. La commande qui affiche quelque chose à l'écran est echo. Donc pour créer le script bonjour_monde nous pouvons écrire :

```
#!/bin/sh
echo "Bonjour, Monde !"
echo "Un premier script est né."
```

Comment on l'exécute ? C'est simple il suffit de taper :

```
[user@becane user]$ sh bonjour_monde
Bonjour, Monde !
Un premier script est né.
[user@becane user]$ _
```

C'est pas cool, vous auriez préféré taper quelque chose comme :

```
[user@becane user]$ ./bonjour_monde
Bonjour, Monde !
```

```
Un premier script est né.
[user@becane user]$ _
```

C'est possible si vous avez au préalable rendu votre script exécutable par la commande :

```
[user@becane user]$ chmod +x bonjour_monde
[user@becane user]$ ./bonjour_monde
Bonjour, Monde !
Un premier script est né.
[user@becane user]$ _
```

Résumons : un script *shell* commence par : #!/bin/sh, il contient des commandes du shell et est rendu exécutable par chmod +x.

[modifier]

Quelques conseils concernant les commentaires

Dans un shell-script, est considéré comme un commentaire tout ce qui suit le caractère # et ce, jusqu'à la fin de la ligne. Usez et abusez des commentaires : lorsque vous relirez un script 6 mois après l'avoir écrit, vous serez bien content de l'avoir documenté. Un programme n'est jamais trop documenté, en revanche il peut être mal documenté ! Un commentaire est bon lorsqu'il décrit pourquoi on fait quelque chose, pas quand il décrit ce que l'on fait.

Exemple :

```
#!/bin/sh
# pour i parcourant tous les fichiers,
for i in ./* ; do
# copier le fichier vers .bak
 cp "$i" "$i.bak"
# fin pour
done
```

Que fait le script ? Les commentaires ne l'expliquent pas ! Ce sont de mauvais commentaires.

En revanche :

```
#!/bin/sh
# on veut faire un copie de tous les fichiers
for i in ./* ; do
# sous le nom *.bak
 cp "$i" "$i.bak"
```

```
done
```

Là au moins, on sait ce qu'il se passe (il n'est pas encore important de connaître les commandes de ces deux fichiers).

[modifier]

Le passage de paramètres

Un script ne sera, en général, que d'une utilisation marginale si vous ne pouvez pas modifier son comportement d'une manière ou d'une autre. On obtient cet effet en « passant » un (ou plusieurs) paramètre(s) au script *via* la ligne de commande. Voyons comment faire cela. Soit le script essai01 :

```
#!/bin/sh
echo le paramètre \$1 est \"$1\"
echo le paramètre \$2 est \"$2\"
echo le paramètre \$3 est \"$3\"
```

Que fait-il ? Il affiche les uns après les autres les trois premiers paramètres du script, donc si l'on tape :

```
$ ./essai01 paramètre un
le paramètre $1 est "paramètre"
le paramètre $2 est "un"
le paramètre $3 est ""
$ _
```

Donc, les variables $1, $2… $9 contiennent les « mots » numéro 1, 2… 9 de la ligne de commande. Attention : par « mot » on entend ensemble de caractères ne contenant pas de caractères de séparations. Les caractères de séparation sont l'espace, la tabulation, le retour chariot quand c'est possible et le point virgule.

Vous avez sans doute remarqué que j'ai utilisé les caractères : \$ à la place de $ ainsi que \" à la place de " dans le script. Pour quelle raison ? C'est simple : si l'on tape echo "essai" on obtient : essai, si l'on veut obtenir "essai" il faut dire à echo (en réalité, au shell qui va lancer echo) que le caractère " n'indique pas le début d'une chaîne de caractère (comme c'est le comportement par défaut) mais que ce caractère fait partie de la chaîne : on dit que l'on « échappe » ou « protège » le caractère " en tapant \". En « échappant » le caractère \ (par \\) on obtient le caractère \ sans signification particulière. On peut dire que le caractère \ devant un autre lui fait perdre sa signification particulière s'il en a une, ne fait rien si le caractère qui suit \ n'en a pas.

Maintenant, essayons de taper :

```
$ ./essai01 *
le paramètre $1 est "Mail"
le paramètre $2 est "essai01"
le paramètre $3 est "nsmail"
```

```
$ _
```

Le résultat doit être sensiblement différent sur votre machine : il dépend du contenu de votre répertoire courant. Que s'est-il passé ? Le *shell* a remplacé le caractère * par la liste de tous les fichiers non cachés présents dans le répertoire actif. En fait, toutes les substitutions du shell sont possibles ! Le *shell* qui substitue aux paramètres des valeurs étendues par les caractères spéciaux : * (toute suite de caractères) ? (un caractère quelconque), [dze] (l'un des caractères d, z ou e), [d-z] (les caractères de 'd' à 'z')…

Autre exemple :

```
$ ./essai01 \*
le paramètre $1 est "*"
le paramètre $2 est ""
le paramètre $3 est ""
$ _
```

Hé oui, on a « échappé » le caractère * donc il a perdu sa signification particulière : il est redevenu un simple *.

C'est bien, me direz vous, mais si je veux utiliser plus de dix paramètres ? Il faut utiliser la commande shift ; à titre d'exemple voici le script essai02 :

```
#!/bin/sh
echo le paramètre 1 est \"$1\"
shift
echo le paramètre 2 est \"$1\"
shift
echo le paramètre 3 est \"$1\"
shift
echo le paramètre 4 est \"$1\"
shift
echo le paramètre 5 est \"$1\"
shift
echo le paramètre 6 est \"$1\"
shift
echo le paramètre 7 est \"$1\"
shift
echo le paramètre 8 est \"$1\"
shift
echo le paramètre 9 est \"$1\"
shift
echo le paramètre 10 est \"$1\"
shift
```

```
echo le paramètre 11 est \"$1\"
```

Si vous tapez :

```
$ ./essai02 1 2 3 4 5 6 7 8 9 10 11 12 13
le paramètre 1 est "1"
le paramètre 2 est "2"
le paramètre 3 est "3"
le paramètre 4 est "4"
le paramètre 5 est "5"
le paramètre 6 est "6"
le paramètre 7 est "7"
le paramètre 8 est "8"
le paramètre 9 est "9"
le paramètre 10 est "10"
le paramètre 11 est "11"
$ _
```

À chaque appel de shift les paramètres sont décalés d'un numéro : le paramètre 2 devient le paramètre 1, 3 devient 2, etc. Évidemment le paramètre 1 est perdu par l'appel de shift : vous devez donc vous en servir avant d'appeler shift (ou le sauvegarder dans une variable).

[modifier]

Les variables

Le passage des paramètres nous a montré l'utilisation de "noms" particuliers : $1, $2 etc. Ce sont les substitutions des variables 1, 2, etc. par leur valeurs. Mais vous pouvez définir et utiliser n'importe quel nom. Attention toutefois à ne pas confondre le nom d'une variable (notée par exemple machin) et son contenu (noté dans notre cas $machin). Vous connaissez peut-être la variable PATH (attention, le shell différencie les majuscules des minuscules) qui contient la liste des répertoires (séparés par des ":") dans lesquels il doit rechercher les programmes. Si dans un script vous tapez :

```
1:#!/bin/sh
2:PATH=/bin # PATH contient /bin
3:PATH=PATH:/usr/bin # PATH contient PATH:/usr/bin
4:PATH=/bin # PATH contient /bin
5:PATH="$PATH:/usr/bin" # PATH contient /bin:/usr/bin
```

Les numéros ne sont là que pour repérer les lignes, il ne faut pas les taper. La ligne 3 est très certainement une erreur, à gauche du signe = il faut une variable (donc un nom sans $) mais à droite de ce même signe il faut une valeur, et la valeur que l'on a mise est PATH:/usr/bin : il n'y a aucune substitution à faire. Par contre la ligne 5 est certainement correcte : à droite du = on a mis "$PATH:/usr/bin", la valeur de $PATH étant /bin, la valeur après substitution par le shell de "$PATH:/usr/bin" est /bin:/usr/bin. Donc, à la fin de la ligne 5, la valeur de la variable PATH est "/bin:/usr/bin".

Attention : les caractères spéciaux (espaces, *, ...) ne peuvent pas apparaitre à gauche du signe =, et doivent être précédés d'un \ ou mis entre guillemets à droite du signe =.

Résumons : MACHIN est un nom de variable que l'on utilise lorsque l'on a besoin d'un nom de variable (mais pas de son contenu), et $MACHIN est le contenu de la variable MACHIN que l'on utilise lorsque l'on a besoin du contenu de cette variable.

[modifier]

Variables particulières

Il y a un certain nombre de variables particulières, en voici quelques unes :

- la variable @ (dont le contenu est $@) contient l'ensemble de tous les "mots" qui on été passé au script (c'est à dire toute la ligne de commande, sans le nom du script). On l'utilise en général avec des guillemets : "$@".
- la variable # contient le nombre de paramètres ($#) qui ont été passés au programme.
- la variable 0 (zéro) contient le nom du script (ou du lien si le script a été appelé depuis un lien).

Il y en a d'autres, moins utilisées : allez voir la man page de bash.

[modifier]

Saisir la valeur d'une variable

Les paramètres permettent à l'utilisateur d'agir sur le déroulement du script avant son exécution. Mais il est aussi souvent intéressant de pouvoir agir sur le déroulement du script lors de son exécution, c'est ce que permet la commande : read nom_variable. Dans cette commande vous pouvez bien sûr remplacer nom_variable par le nom de variable qui vous convient le mieux. Voici un exemple simple.

```
#!/bin/sh
echo -n "Entrez votre prénom : "
read prenom
echo -n "Entrez votre nom de login : "
read nomlogin
echo "Le nom de login de $prenom est $nomlogin."
```

Ce script se déroule ainsi :

```
$ ./essai02bis
Entrez votre prénom : Marc
Entrez votre nom de login : spoutnik
Le nom de login de Marc est spoutnik.
```

Lors du déroulement du script vous devez valider vos entrées en appuyant sur la touche "Entrée".

L'option -s de read permet de masquer la saisie. Par exemple read -s pass.

[modifier]

Arithmétique

Vous vous doutez bien qu'il est possible de faire des calculs avec le shell. En fait, le shell ne "sait" faire que des calculs sur les nombres entiers (ceux qui n'ont pas de virgules ;-). Pour faire un calcul il faut encadrer celui-ci de : $((un calcul)). Exemple, le script essai03 :

```
#!/bin/sh
echo 2+3*5 = $((2+3*5))
MACHIN=12
echo MACHIN*4 = $(($MACHIN*4))
```

Affichera :

```
$ ./essai03
2+3*5 = 17
MACHIN*4 = 48
```

Vous remarquerez que le shell respecte les priorités mathématiques habituelles (il fait les multiplications avant les additions !). L'opérateur puissance est "**" (ie : 2 puissance 5 s'écrit : 2**5). On peut utiliser des parenthèses pour modifier l'ordre des calculs.

[modifier]

Guillemets, échappement, ...

Nous avons vu plus haut qu'écrire par exemple * ou * étaient différents. En fait, la plupart des caractères autres que des chiffres ou des lettres ont une signification particulière pour le shell.

Pour exécuter une commande, le shell commence par regarder tous ces caractères particuliers. Il fait en particulier les choses suivantes (dans cet ordre) :

- Expansion des variables (par exemple, dans echo $var, $var sera remplacé par sa valeur),
- Découpage de la ligne de commande selon les blancs (par exemple, la commande echo un deux trois sera découpée en une commande echo avec les trois arguments un, deux et trois, alors que echo "un deux trois", avec des guillemets en plus, sera découpée en une commande echo avec un seul argument un deux trois).
- Expansion des jokers (* remplacé par la liste des fichiers non cachés dans le répertoire courrant, ?, ...)

Pour éviter ce traitement spécial pour les caractères spéciaux, on a plusieurs solutions :

- Précéder le caractère par un anti-slash :

```
echo \*        # affiche une étoile
echo \$        # affiche un dollar
cat fichier\ avec\ espaces  # affiche le contenu du fichier "fichier avec espaces"
```

- Mettre toute une chaine entre guillemets simples (apostrophes). Dans ce cas, seul le guillemet reste un caractère spécial (pour terminer la chaine) :

```
echo '*'       # affiche une étoile
echo '$'       # affiche un dollar
cat 'fichier avec espaces'  # affiche le contenu du fichier "fichier avec espaces"
```

- Mettre toute une chaine entre guillemets doubles. Dans ce cas, les caractères $ et \ restent actifs :

```
echo "*"       # affiche une étoile
var=toto
echo "$var"       # affiche "toto"
f=fichier
e=espaces
cat "$f avec $e"  # affiche le contenu du fichier "fichier avec espaces"
echo "\$f avec $e" # affiche "$f avec espaces"
```

Il reste un autre type de guillemets, le guillemet inversé (backquote en anglais) : `. Quand le shell rencontre une chaine du type `commande`, il exécute commande, et remplace la chaine par le résultat de la commande. Par exemple :

```
ancienne_date=`date`
sleep 3
echo "La date est maintenant :"
date
echo "Mais tout à l'heure, il était : $ancienne_date"
```

Note : à la place de `commande`, on peut aussi écrire $(commande), qui est plutôt plus lisible, et plus portable.

[modifier]

Les instructions de contrôle de scripts

Les instructions de contrôle du shell permettent de modifier l'exécution purement séquencielle d'un script. Jusqu'à maintenant, les scripts que nous avons créés n'étaient pas très complexes. Ils ne pouvaient de toute façon pas l'être car nous ne pouvions pas modifier l'ordre des instructions, ni en répéter.

[modifier]

L'exécution conditionnelle

Lorsque vous programmerez des scripts, vous voudrez que vos scripts fassent une chose si une certaine condition est remplie et autre chose si elle ne l'est pas. La construction de bash qui permet cela est le fameux test : if then else fi. Sa syntaxe est la suivante (la partie else... est optionnelle) :

```
if <test> ;
then
   <instruction 1>
   <instruction 2>
   ...
   <instruction n>
else
   <instruction n+1>
   ...
   <instruction n+p>
fi
```

Il faut savoir que tous les programmes renvoient une valeur. Cette valeur est stockée dans la variable ? dont la valeur est, rappelons le : "$?". Pour le shell une valeur nulle est synonyme de VRAI et une valeur non nulle est synonyme de FAUX. Ceci parce que, en général les programmes renvoie zéro quand tout c'est bien passé et un code d'erreur (nombre non nul) quand il s'en est produit une.

Il existe deux programmes particuliers : false et true. true renvoie toujours 0 et false renvoie toujours 1. Sachant cela, voyons ce que fait le programme suivant :

```
#!/bin/sh
if true ;
then
   echo Le premier test est VRAI($?)
else
   echo Le premier test est FAUX($?)
fi

if false ;
then
   echo Le second test est VRAI($?)
else
   echo Le second test est FAUX($?)
fi
```

Affichera :

```
$ ./test
Le premier test est VRAI(0)
Le second test est FAUX(1)
$ _
```

On peut donc conclure que l'instruction if ... then ... else ... fi, fonctionne de la manière suivante : si (**if** en anglais) le test est VRAI(0) alors (**then** en anglais) le bloc d'instructions compris entre le then et le else (ou le fi en l'absence de else) est exécuté, sinon (**else** en anglais) le test est FAUX(différent de 0)) et on exécute le bloc d'instructions compris entre le else et le fi si ce bloc existe.

Bon, évidemment, des tests de cet ordre ne paraissent pas très utiles. Voyons maintenant de vrais tests.

[modifier]

Les tests

Un test, nous l'avons vu, n'est rien de plus qu'une commande standard. Une des commandes standard est test, sa syntaxe est un peu complexe, nous allons la décrire avec des exemples.

- si l'on veut tester l'existence d'un répertoire <machin>, on tapera : test -d <machin> ('d' comme ***d****irectory*)
- si l'on veut tester l'existence d'un fichier <machin>, on tapera : test -f <machin> ('f' comme ***f****ile*)
- si l'on veut tester l'existence d'un fichier ou répertoire <machin>, on tapera : test -e <machin> ('e' comme ***e****xist*)

Pour plus d'information faites : man test.

On peut aussi combiner deux tests par des opérations logiques : 'ou' correspond à -o ('o' comme ***o****r*), 'et' correspond à -a ('a' comme ***a****nd*) (à nouveau allez voir la man page), exemple :

```
test -x /bin/sh -a -d /etc
```

Cette instruction teste l'existence de l'éxécutable /bin/sh (-x /bin/sh) et (-a) la présence d'un répertoire /etc (-d /etc).

On peut remplacer la commande test <un test> par [<un test>] qui est plus lisible, exemple :

```
if [ -x /bin/sh ] ; then
# ('x' comme "e_x_ecutable")
  echo "/bin/sh est exécutable. C'est bien."
else
  echo "/bin/sh n'est pas exécutable."
  echo "Votre système n'est pas normal."
fi
```

Toujours avec les crochets de test, si vous n'avez qu'une seule chose à faire en fonction du résultat d'un test, alors vous pouvez utiliser la syntaxe suivante :

[-x /bin/sh] && echo /bin/sh est exécutable.
ou encore :
[-x /bin/sh] || echo "/bin/sh n'est pas exécutable."

L'affichage du message est effectué, dans le premier cas que si le test est vrai et dans le second cas, que si le test est faux. Dans l'exemple on teste si /bin/sh est un fichier exécutable. Cela allège le script sans pour autant le rendre illisible, si cette syntaxe est utilisée à bon escient.

Mais il n'y a pas que la commande test qui peut être employée. Par exemple, la commande grep renvoie 0 quand la recherche a réussi et 1 quand la recherche a échoué.
Par exemple :

```
if grep -E "^frederic:" /etc/passwd > /dev/null ; then
   echo L'utilisateur frederic existe.
else
   echo L'utilisateur frederic n'existe pas.
fi
```

Cette série d'instruction teste la présence de l'utilisateur frederic dans le fichier /etc/passwd. Vous remarquerez que l'on a fait suivre la commande grep d'une redirection vers /dev/null pour que le résultat de cette commande ne soit pas affiché : c'est une utilisation classique. Ceci explique aussi l'expression : "Ils sont tellement intéressants, tes mails, que je les envoie vers /dev/null" ;-).

[modifier]

Faire quelque chose de différent suivant la valeur d'une variable

L'instruction case ... esac permet de modifier le déroulement du script selon la valeur d'un paramètre ou d'une variable. On l'utilise le plus souvent quand les valeurs possibles sont en nombre restreint et peuvent être prévues. Les imprévus peuvent alors être représentés par le signe *. Demandons par exemple à l'utilisateur s'il souhaite afficher ou non les fichiers cachés du répertoire en cours.

```
#!/bin/sh
# pose la question et récupère la réponse
echo "Le contenu du répertoire courant va être affiché."
echo -n "Souhaitez-vous afficher aussi les fichiers cachés (oui/non) : "
read reponse
# agit selon la réponse
case $reponse in
   oui)
     clear
     ls -a;;
   non)
```

```
     ls;;
   *) echo "Erreur, vous deviez répondre par oui ou par non.";;
esac
```

Seules les réponses "oui" et "non" sont réellement attendues dans ce script, toute autre réponse engendrera le message d'erreur. On notera qu'ici l'écran est effacé avant l'affichage dans le cas d'une réponse positive, mais pas dans celui d'une réponse négative. Lorsque vous utilisez l'instruction case ... esac, faites bien attention de ne pas oublier les doubles points-virgules terminant les instructions de chacun des cas envisagés.

[modifier]

Faire la même chose pour tous les éléments d'une liste

Lorsqu'on programme, on est souvent amené à faire la même chose **pour tous** les élément d'une liste. Dans un shell script, il est bien évidemment possible de ne pas réécrire dix fois la même chose. On dira que l'on fait une boucle. L'instruction qui réalise une boucle est

```
for <variable> in <liste de valeurs pour la variable> ; do
   <instruction 1>
   ...
   <instruction n>
done
```

Voyons comment ça fonctionne. Supposons que nous souhaitions renommer tous nos fichiers *.tar.gz en *.tar.gz.old, nous taperons le script suivant :

```
#!/bin/sh
# x prend chacune des valeurs possibles correspondant
# au motif : *.tar.gz
for x in ./*.tar.gz ; do
   # tous les fichiers $x sont renommés $x.old
   echo "$x -> $x.old"
   mv "$x" "$x.old"
   # on finit notre boucle
done
```

Simple, non ? Un exemple plus complexe ? Supposons que nous voulions parcourir tous les sous-répertoires du répertoire courant pour faire cette même manipulation. Nous pourrons taper :

```
 1:#!/bin/sh
2:for REP in `find -type d` ; do
3:   for FICH in "$REP/*.tar.gz" ; do
```

```
4:      if [ -f "$FICH" ] ; then
5:         mv "$FICH" "$FICH.old"
6:      else
7:         echo "On ne renomme pas $FICH car ce n'est pas un fichier"
8:      fi
9:   done
10:done
```

Explications : dans le premier 'for', on a précisé comme liste : `find -type d` (attention au sens des apostrophes, sur un clavier azerty français, on obtient ce symbole en appuyant sur ALTGR+é, ce ne sont pas des simples quotes ').
Lorsque l'on tape une commande entre apostrophes inverses, le shell exécute d'abord cette commande, et remplace l'expression entre apostrophes inverses par la sortie standard de cette commande (ce qu'elle affiche à l'écran). Donc, dans le cas qui nous intéresse, la liste est le résultat de la commande find -type d, c'est à dire la liste de tous les sous-répertoires du répertoire courant.
Ainsi, en ligne 2, on fait prendre à la variable REP le nom de chacun des sous-répertoires du répertoire courant, puis (en ligne 3) on fait prendre à la variable FICH le nom de chacun des fichiers .tar.gz de $REP (un des sous-répertoires), puis si $FICH est un fichier, on le renomme, sinon on affiche un avertissement.

note : ce script ne marchera pas si des noms de répertoires contiennent des espaces : dans ce cas, `find -type d` sera exécuté, remplacé par son résultat, et ensuite découpé selon les blancs.

Remarque : ce n'est pas le même fonctionnement que la boucle for d'autres langage (le pascal, le C ou le basic par exemple).

[modifier]

Faire une même chose tant qu'un certaine condition est remplie

Pour faire une certaine chose **tant qu'**une condition est remplie, on utilise un autre type de boucle :

```
while <un test> ; do
   <instruction 1>
   ...
   <instruction n>
done
```

Supposons, par exemple que vous souhaitiez afficher les 100 premiers nombres (pour une obscure raison), alors vous taperez :

```
i=0
while [ $i -lt 100 ] ; do
   echo $i
   i=$[$i+1]
done
```

Remarque : -lt signifie "**l**esser **t**han" ou "plus petit que" (et -gt signifie "plus grand", ou "**g**reater **t**han").

Ici, on va afficher le contenu de i et lui ajouter 1 tant que i sera (-lt) plus petit que 100. Remarquez que 100 ne s'affiche pas, car -lt est "plus petit", mais pas "plus petit ou égal" (dans ce cas, utilisez -le et -ge pour "plus grand ou égal").

[modifier]

Refaire à un autre endroit la même chose

Souvent, vous voudrez refaire ce que vous venez de taper autre part dans votre script. Dans ce cas il est inutile de retaper la même chose, préférez utiliser des fonctions qui permettent de réutiliser une portion de script. Voyons un exemple :

```
#!/bin/sh
addpath ()
{
   if echo $PATH | grep -v $1 >/dev/null; then
      PATH=$PATH:$1;
   fi;
   PATH=`echo $PATH|sed s/::/:/g`
}

addpath /opt/apps/bin
addpath /opt/office52/program
addpath /opt/gnome/bin
export PATH
```

Au début, nous avons défini une fonction nommée addpath dont le but est d'ajouter le premier argument ($1) de la fonction addpath à la variable PATH si ce premier argument n'est pas déjà présent (grep -v $1) dans la variable PATH, ainsi que supprimer les chemins vides (sed s/::/:/g) de PATH.
Ensuite, nous exécutons cette fonction pour trois arguments : /opt/apps/bin, /opt/office52/bin et /opt/gnome/bin.

En fait, une fonction est seulement un script écrit à l'intérieur d'un script. Les fonctions permettent surtout de ne pas multiplier les petits scripts, ainsi que de partager des variables sans se préoccuper de la clause export mais cela constitue une utilisation avancée du shell, nous n'irons pas plus loin dans cet article.

Remarque : on peut aussi utiliser le mot clé function, mais cette syntaxe n'est pas supportée par tous les shells, et n'est donc pas portable :

```
function addpath ()
{
# ...
}
```

[modifier]

Autres types de répétitions.

Il existe d'autres types de répétitions, mais nous ne nous en occuperons pas dans cet article, je vous conseille la lecture, forcément profitable, de la "man page" de bash (man bash).

À vous de jouer !

Cette page est issue de la documentation 'pré-wiki' de Léa a été convertie avec HTML::WikiConverter. Elle fut créée par Frédéric Bonnaud le 29/08/2000.
[modifier]

Copyright

Autres ressources

- Pour aller plus loin... (des trucs et astuces pour bash)
- Aller à 100% (Guide avancé d'écriture des scripts Bash)
- Introduction à Unix et à la programmation Shell (cours sur Unix de l'école d'ingénieur de l'ESME SUDRIA, publié sur [1])
- Tout ce que vous avez toujours voulu savoir sur Unix (cours de l'Énise)