



www.dirasats.com



هذا الغلاف لا يعبر عن حقوق الملكية او فحوى الكتاب, فهو مجرد واجهة للموقع المحمل منه

شكرا لك على ثقتك بنا وعلى اختيار موقعنا



www.dirasats.com



من اجل تواصل معنا المرجو زيارة الموقع ستجد جميع المعلومات

www.dirasats.com

TP : SCRIPTS

Un script est une suite d'instruction élémentaire qui sont exécutées de façon séquentielle (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 bash.

Notions de base

Pour commencer, il faut savoir qu'un script est un fichier texte standard pouvant être créé par n'importe quel éditeur : vi, emacs 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, pour le cas qui nous intéresse : /bin/sh. Donc un script élémentaire pourrait être :

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

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 faire :

```
[user AT became user]$ sh bonjour_monde
Bonjour, Monde !
un premier script est né.
[user AT became user]$ _
```

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

```
[user AT became user]$ ./bonjour_monde
Bonjour, Monde !
un premier script est né.
[user AT became user]$ _
```

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

```
[user AT became user]$ chmod +x bonjour_monde
[user AT became user]$ ./bonjour_monde
Bonjour, Monde !
un premier script est né.
[user AT became user]$ _
```

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

- 1) 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. Utilisez et abusez des commentaires : lorsque vous relirez un script 6 mois après l'avoir écrit, vous serez bien content de l'avoir documenté.

Exemple :

```
/bin/sh
# on veut faire une copie de tous les fichiers
for i in * ; do
# sous le nom $i.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)

2) 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. 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 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 ? La raison est simple, si l'on tape : `echo "essai"` on obtient : `essai`, si l'on veut obtenir "essai" il faut dire à `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" 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). Que c'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 possible ! C'est le shell qui "substitue" aux paramètres des valeurs étendues par les caractères `*` (toute suite de caractères) et `[ab]` (l'un des caractères a ou b). Autre exemple :

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

Et 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 2 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 2 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"
$ _
```

A chaque appel de shift les paramètres sont déplacés d'un numéro : le paramètre un devient le paramètre deux et c. Évidemment le paramètre un est perdu par l'appel de shift : vous devez donc vous en servir avant d'appeler shift.

3) Les variables

Le passage des paramètres nous à montrer l'utilisation de "nom" particuliers : \$1, \$2 etc. ce sont les substitutions des variables 1, 2 et c. par leur valeurs. Mais vous pouvez définir et utiliser n'importe quelle nom. Attention toute fois, à ne pas confondre le nom d'une variable (notée par exemple machin) et son contenu (notée dans cas \$machin). Vous connaissez 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:/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 mis 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 : il ne doit y avoir aucun espace de part et d'autre du signe "=".

Résumons : MACHIN est un nom de variable que l'on utilise lorsque l'on a besoin d'un nom de 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. Variables particulières Il y a un certain nombre de variables particulières, voici leur signification :

- la variable * (dont le contenu est \$*) contient l'ensemble de tous les "mots" qui on été passé au script.
- 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.

4) 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 : `$((un calcul))` ou `$(un calcul)`. Exemple, le script `essai03` :

```
#!/bin/sh
echo 2+3*5 = $((2+3*5))
MACHIN=12
echo MACHIN*4 = [$MACHIN*4] Affichera :
$ sh 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.

Les instructions de controles de scripts

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

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 (ce qui est en italique est optionnel) :

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

Il faut savoir que tout les programmes renvoie une valeur. Cette valeur est stockée dans la variable `?` dont la valeur est, rapelons 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 numéro non nul d'erreur quand il s'en est produit une).

Il existe deux programmes particulier : `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 bloque d'instructions comprises 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 bloque d'instructions comprises entre le else et le fi si ce bloque existe.

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

5) 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, je vais la décrire avec des exemples.

- si l'on veut tester l'existence d'un répertoire , on tapera : test -d
- si l'on veut tester l'existence d'un fichier , on tapera : test -f
- si l'on veut tester l'existence d'un fichier ou répertoire , on tapera : test -e

Pour plus d'information faites : man test.

On peut aussi combiner deux tests par des opérations logiques : ou correspond à -o, et correspond à -a (à nouveau allez voir la man page), exemple : test -x /bin/sh -a -d /etc Cette instruction teste l'existence de l'exécutable /bin/sh (-x /bin/sh) et (-a) la présence d'un répertoire /etc (-d /etc).

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

```
if [ -x /bin/sh ] ; then
    echo /bin/sh est executable. C'est bien.
else
    echo /bin/sh n'est pas executable.
```

```
    echo Votre système n'est pas normal.
fi
```

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, 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 affichée : c'est une utilisation classique. Ceci explique aussi l'expression : "Ils sont tellement intéressants tes mails que je les envoie à /dev/null" ;-).

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

Faire la même chose pour tous les éléments d'une liste. Lorsque l'on programme, on est souvent amené à faire la même chose pour divers éléments 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
# I prend chacune des valeurs possibles correspondant
# au motif : *.tar.gz
for I in *.tar.gz ; do
    # tous les fichiers $I sont renommés $I.old
    echo "$I -> $I.old"
    mv $I $I.old
# on fini notre boucle
done
```

Simple, non ? Un exemple plus complexe ? Supposons que nous voulions parcourir tous les répertoires du répertoire courant pour faire cette même manipulation. Nous pourrions 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 répertoire
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). Lorsque l'on tape une commande entre apostrophes inverses, le shell exécute d'abord cette commande, et remplace l'expression entre apostrophe inverse par la sortie standard de cette commande. 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. Donc 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.

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

Faire une même chose tant qu'une 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 "lesser than" ou "plus petit que".

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.

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 l'instruction function qui permet de réutiliser une portion de script. Voyons un exemple :

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

Au debut, 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 vide (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. Elles 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 ne nous en occuperons pas dans cet article.

Remarque : le mot function peut être omis.