TP n°7 – Programmation de scripts en Bash

1) Premiers essais

a) Au clavier

Tapez les commandes suivantes ligne par ligne sans les mettre dans un script et regardez à chaque fois les résultats. Bash peut à la fois être programmé par des scripts et au clavier.

```
echo Salut a toi
echo Salut $LOGNAME dont le compte est dans $HOME
echo Tu fais ce TP dans $PWD, quel beau dossier '!'
```

explications : $*mot* est compris par bash comme étant à remplacer par la valeur de la variable *mot*. Les deux variables sont prédéfinies et gérées automatiquement par bash. Attention, A ≠ a.

```
nom=Yakari
animal=Aigle
taille=Grand
echo le totem de $nom est $taille $animal
```

explications : une affectation se fait par variable=valeur sans aucun espace. Pour voir pourquoi il ne faut pas d'espaces, essayez d'en mettre avant ou après le =. Souvenez-vous bien de ça !

```
totem=$taille$animal                    # sans espace pour commencer
echo le totem de $nom est $totem
totem=$taille $animal                   # oups, c'est quoi qu'il fait ?
echo le totem de $nom est $totem
totem=$taille_$animal_noir              # grand_aigle_noir ? même pas
echo le totem de $nom est $totem
totem=${taille}_${animal}_noir          # ok ok…
echo le totem de $nom est $totem
totem='$taille $animal noir'
echo le totem de $nom est $totem        # il est bouché, bash ?
totem="$taille $animal noir"            # ah enfin !
echo le totem de $nom est $totem
echo 'le totem de $nom est $totem'
echo "le totem de $nom est $totem"
echo 'le totem de '$nom" est $totem"    # on peut juxtaposer des chaînes
```

explications : bash n'accepte plusieurs mots que s'ils sont encadrés par des délimiteurs de chaînes. Les délimiteurs de chaînes ' et " sont spécialisés – chacun son rôle. On peut juxtaposer des chaînes délimitées différemment. Si on colle un mot juste après le nom d'une variable, bash ne peut pas savoir où s'arrête la variable. On doit mettre des {} pour isoler le nom de la variable du reste :

```
totem=${taille}${animal}noir      ou   ${taille}_${animal}_noir
echo mon totem est $totem
```

Corrigez les lignes suivantes pour qu'elles affichent les textes sans faire d'erreurs avec hero et totem remplacés par leurs valeurs mais en laissant $USD tel quel, et avec les " ' et ! affichés à l'écran :

```
hero=Yakari
echo l'animal fetiche de $hero est $totem, il vaut 3000$US, c'est cher !
echo "vole, petit $totem !" lui-dit $hero
```

Voici maintenant d'autres exemples à essayer toujours au clavier :

```
nombre=`ls $HOME | wc -l`                # syntaxe obsolète
echo "il y a $nombre éléments (fichiers ou dossiers) dans $HOME"
nombre=$(ls $HOME | wc -l)               # syntaxe recommandée
echo il y a toujours $nombre éléments dans $HOME
```

explications : les délimiteurs `...` ou $(...) doivent encadrer une commande Unix ; ce qu'elle affiche est redirigé dans la variable. Il est recommandé d'employer la syntaxe $( ).

Complétez les lignes suivantes pour qu'elles affichent le nombre total de processus qui tournent actuellement sur votre machine :

```
nombre= ... à vous de mettre ce qu'il faut...
echo il y a $nombre processus sur $HOSTNAME
```

Pour finir sur les affectations de variables, voici comment on fait des calculs *entiers* en bash :

```
x=3
y=$(( 11 - (-x * -7) ))                  # pensez à vérifier les calculs
echo $x $y $(( y / x )) $((-y % x))      # vérifiez les calculs
```

Gare à ne pas confondre les syntaxes $(commande) et $((expression)).

b) Dans un script

On va maintenant passer à l'écriture de scripts bash. On rappelle que ce sont des fichiers texte non compilés mais exécutables, qui contiennent des commandes Unix et des instructions Bash telles que les affectations vues précédemment. On doit simplement placer `#!/bin/bash` dans la toute première ligne du fichier et se rajouter le droit d'exécution dessus.

Ouvrez l'éditeur de votre choix et tapez le texte suivant dedans. Enregistrez-le sous un nom tel que essai1.sh, ou essai1, mais surtout pas test ou script qui sont des noms de commandes existantes (vous n'auriez pas l'idée de l'appeler ls ou rm ?).

```
#!/bin/bash
echo mon premier script bash fonctionne parfaitement du premier coup
```

Ensuite rajoutez le droit d'exécution par `chmod u+x essai1.sh`. Puis lancez lescript en tapant juste son nom : `essai1.sh`

Essayez maintenant le script suivant (essai2.sh,avec toujours `chmod u+x essai2.sh`) :

```
#!/bin/bash
echo -n 'Quel est ton prénom ? '
read prenom
echo -n "Bonjour, cher(e) $prenom, comment vas-tu ? "
read etat
echo ah, tiens moi aussi, je vais $etat
```

Essayez avec différentes valeurs y compris des réponses vides ou très longues...

Maintenant justement on rajoute un test pour refuser les réponses vides :

```
#!/bin/bash
read -p 'quel est ton prénom : ' prenom
if test "$prenom" = ''
then
     echo "désolé, je ne parle pas aux inconnus."
     exit 1            # sortie immédiate du programme : pas besoin de else
fi
... comme précédemment
```

Remarquez l'emploi de l'option -p message de la commande read pour afficher la question et lire la réponse en une seule commande.

Remarquez les " " autour des variables et les '' qui créent une chaîne vide. Ils sont nécessaires puisque ces variables peuvent contenir aucun ou plusieurs mots. Grâce à ces délimiteurs, les mots restent ensembles.

Autre remarque importante : vous êtes prié(e) de respecter scrupuleusement l'indentation. Nous refuserons de vous aider si vous nous montrez un programme non indenté.

Modifiez le script pour qu'il vérifie aussi que l'état n'est pas vide.

Voici un autre type de test dans essai3.sh (`chmod u+x essai3.sh`). Lancez-ce script et testez tous les cas (fichier et dossier existant, absent...).

```
#!/bin/bash
read -p 'donnez un nom de fichier ou de répertoire : ' nomfich
if test -f "$nomfich"
then
     echo "$nomfich" est un fichier
else
     if test -d "$nomfich"
     then
           echo "$nomfich" est un dossier
     else
           echo "je ne vois ni fichier ni dossier qui s'appelle $nomfich"
     fi
fi
```

Voici un autre concept très important pour programmer en bash : les paramètres. Testez le script suivant en le lançant avec ou sans paramètres sur la ligne de commande :

```
#!/bin/bash
if test $# -eq 0
then
     echo il faudrait fournir un paramètre sur la ligne de commande
     echo par exemple, lancez : $0 bonjour mon beau script
     exit 1
fi
echo le premier paramètre est : $1
if test "$2" = ''
then
     echo "il n'y a pas d'autres paramètres"
else
     echo vous avez aussi fourni : $2
fi
echo vous avez fourni $# paramètres en tout, voici la liste : $@
```

Les paramètres sont le moyen normal de spécifier les objets (fichiers, utilisateurs...) traités par le script. NB: $0 est le nom du script lui-même, $# est le nombre de paramètres fournis.

Pour finir le tutoriel, voici deux exemples de boucles :

```
#!/bin/bash
echo "j'ai caché un nombre, veuillez le deviner"
while read -p "quel nombre proposez-vous ? " proposition
do
     if test "$proposition" -eq $$
     then
           echo "t'es trop fort, j'abandonne"
           break
     else
           echo "eh non, c'est pas ça"
     fi
done
echo "fin du jeu"
```

La comparaison entre la proposition et la bonne réponse, $$ est faite par -eq car ce sont des nombres. Quand ce sont des chaînes, on les compare par = ou !=. Il y a d'autres comparateurs, comme -lt, -ge à retrouver dans le cours.

Indice, le nombre caché est $$, c'est à dire le numéro de processus de ce script (qui change à chaque lancement). Utilisez ps pour le connaître (mettez le jeu en arrière-plan le temps de regarder son PID).

Voici un second type de boucle extrêmement utile, la boucle for :

```
#!/bin/bash
echo "Voici ce que je vois dans ce dossier :"
for nom in *
do
     if test -f "$nom"
     then
           if test ${nom#*.} = 'sh'
           then
                 echo "Il y a $nom et c'est probablement un script bash"
           else
                 echo "Il y a le fichier $nom"
           fi
     else
           echo "Il y a $nom mais ce n'est pas un fichier"
     fi
done
```

Explication : la boucle for parcourt tous les fichiers et dossiers du répertoire courant : ceux qui sont désignés par *. Pour chacun, il y a un premier test qui signifie : est-ce un fichier ? Et il y a un second test qui signifie : est-ce que la fin de son nom est « sh » ? La notation ${variable#motif} extrait le reste de la variable si son début correspond au motif. Vous irez voir dans le cours toutes les autres conditions possibles.

2) Mini-projet : commandes del et undel

On va mettre ces connaissances à profit pour programmer une nouvelle commande très utile pour les utilisateurs : pouvoir supprimer un fichier comme la commande rm mais avec la possibilité de le récupérer dans une corbeille. Le principe est de déplacer le fichier à supprimer dans un répertoire ~/corbeille au lieu de le supprimer[1]. Cette corbeille est dans le compte, donc facilement accessible de partout. Vider la corbeille = simplement supprimer ce dossier.

Voici l'algorithme du script **del** à programmer :

1 Ça fonctionne exactement comme ça sur Windows, le dossier s'appelle RECYCLER et il est caché.

```
vérifier qu'il y a un paramètre, erreur sinon
vérifier que ce paramètre est un fichier, erreur sinon
vérifier que le répertoire ~/corbeille existe, sinon le créer
déplacer le fichier dans ~/corbeille
```

Vérifiez qu'il marche correctement en le testant sur plusieurs fichiers.

Vous remarquerez qu'il ne sait pas traiter plusieurs fichiers, donc on ne peut pas le lancer avec des jokers dans le paramètre. (Pourquoi ?). Malheureusement les boucles ne sont pas au programme de ce TP donc on ne pourra pas améliorer cet aspect aujourd'hui.

Écrire maintenant un script appelé **undel** qui remet un fichier effacé avec del. Voici son algorithme. Nb : il est possible qu'il y ait une autre commande appelée pareil, faire `which undel` pour savoir, et dans ce cas, il faudra appeler le votre autrement.

```
vérifier qu'il y a un paramètre, erreur sinon
vérifier que ~/corbeille/ le paramètre est un fichier, sinon erreur : pas trouvé
déplacer le fichier de ~/corbeille vers le répertoire courant
```

Vous remarquerez que si on efface plusieurs fois un fichier ayant le même nom (parce qu'on le recrée à chaque fois), on ne retrouve que la dernière version dans la corbeille :

```
prompt% echo salut > toto
prompt% del toto
prompt% echo ciao > toto
prompt% del toto
prompt% ls ~/corbeille
titi       toto
prompt% more ~/corbeille/toto
ciao
prompt% undel toto
prompt% more toto
ciao
prompt%
```

On voudrait garder toutes les versions effacées d'un même fichier. L'idée est de leur rajouter un numéro quand elles sont dans la corbeille. Ainsi, dans la corbeille, on pourra trouver :

```
prompt% ls ~/corbeille
titi.1     titi.2      titi.3      toto.1      toto.2
prompt%
```

Voici la modification du script **del** :

```
... tests sur le fichier et la corbeille...
y-a-t-il au moins un fichier dans la corbeille qui s'appelle fichier.1 ?
      * si oui : compter les fichiers qui sont dans ~/corbeille/ et dont le nom commence
comme celui qu'on veut supprimer : ls $1.* | wc -l
            il faut récupérer ce nombre dans une variable puis lui ajouter 1
      * sinon le nombre est 1
déplacer le fichier dans ~/corbeille en le nommant fichier.nombre
```

Tester ce script avec plusieurs fichiers et plusieurs versions pour chacun.

Ensuite, on s'intéresse à la récupération d'une certaine version d'un fichier. L'utilisateur va fournir le nom du fichier à récupérer et le numéro de version qu'il souhaite. Il faut remettre ce fichier dans le répertoire courant en enlevant le numéro de version.

Si l'utilisateur ne fournit pas le numéro de version, on doit lui remettre la dernière version du fichier. Il suffit simplement de compter les versions présentes dans la corbeille pour savoir laquelle est la

dernière. Voici donc le nouvel algorithme de **undel** :

```
... tests sur le paramètre et la corbeille...
si le n° de version n'est pas fourni, compter les versions du fichier -> version à rendre
vérifier que ~/corbeille/ le fichier.la version est un fichier, sinon erreur : pas trouvé
déplacer le fichier de ~/corbeille vers le répertoire courant en retirant le n° de version
```

Quel est le gros problème que ça pose de déplacer une version qui n'est pas la dernière ? Que se passe-t-il quand on supprime à nouveau cette dernière, quel est le numéro de version qui est choisi ?

Modifiez le script recup afin qu'il laisse la version dans la corbeille quand ce n'est pas la dernière qui avait été effacée.

3) Mini-projet : le script lancer-progs

On voudrait programmer un script très utile dans un système d'exploitation. Ce script appelé run-parts dans le vrai système exécute chaque programme d'un certain dossier, en leur passant à chacun les mêmes paramètres.

D'abord, il faut créer un sous-dossier contenant quelques scripts ultra simples. Tapez ces commandes (ou copiez-collez, mais une ligne à la fois) :

```
mkdir dossier
for nom in  06-s2.sh  01-script1.sh  45-balablabla.sh  13-essai.sh
do
    echo '#!/bin/bash' > dossier/$nom
    echo -n echo je suis le script $nom >> dossier/$nom
    echo ' et mes paramètres sont $@' >> dossier/$nom
    echo 'sleep $((RANDOM % 5 + 4))' >> dossier/$nom
    echo echo je suis $nom et je me termine >> dossier/$nom
    chmod u+x dossier/$nom
done
```

Comme vous le voyez, ça crée 4 scripts. Chacun, quand on le lance, affiche son nom et les paramètres qu'il reçoit, ensuite il attend entre 4 et 8 secondes puis affiche un dernier message avant de se terminer. Exemple à essayer ensuite : `dossier/01-script1.sh titi toto`

On remarque que les noms sont structurés en NN-reste. Ça permet de classer ces scripts : leur nom donne l'ordre dans lequel on doit les exécuter. On trouve cette technique de nommage pour avoir un ordre de prise en compte dans le dossier /etc/rc*.d, dans /etc/php5/conf.d, dans /etc/fonts/conf.d, dans /etc/X11/Xsession.d (faites un ls sur ces dossiers).

On demande de faire un script appelé `lancer-progs` qui va lancer automatiquement tous ces scripts dans le bon ordre. NB : il peut y avoir autre chose que des scripts dans le dossier : des fichiers texte, des images... il ne faut lancer que ceux qui sont exécutables (script ou programme).

> `lancer-progs dossier titi toto` doit lancer 01-script1.sh, puis 06-s2.sh puis 13-essai.sh puis 45-... chacun avec les paramètres `titi toto`.

Éléments : il faut faire une boucle sur tous les fichiers du dossier, pour chacun, tester s'il est exécutable : test -x $nomfich, si c'est le cas, alors le lancer en lui passant les paramètres qu'on a passé à lancer-progs.

Amélioration : actuellement votre script lance successivement tous les programmes du dossier : il fonctionne de manière synchrone : le suivant ne démarre que lorsque le précédent s'est fini. On voudrait maintenant que si on fournit l'option `--jobs` à `lancer-progs`, il lance les programmes en arrière-plan. La commande interne `shift` vous servira sûrement.

`lancer-progs --jobs dossier riri loulou` doit lancer 01-script1.sh, puis 06-s2.sh puis 13-essai.sh puis 45-... chacun en arrière-plan, et tous avec les paramètres `riri loulou`.

Si vous avez fin ça, essayez de faire qu'on puisse soit mettre `--jobs` (option longue) soit `-j` (option courte). Si vous avez fini, alors allez voir la documentation de la commande `getopts` de bash : faites man bash puis cherchez *getopts* vers le début de ligne. Attention, ce n'est pas man getopt (au singulier) qu'il faut faire. `getopts` permet d'analyser les paramètres d'un script et d'extraire ceux qui sont des options, telles que `-j`. Ça vous donnera quelque chose comme :

```
while getopts 'j' option
do
    case option in
        j)  traiter l'option -j... (noter dans une variable qu'elle est là)
            ;;
        *)  echo "usage $0 [-j] dossier paramètres..."
            exit 1
            ;;
    esac
done
...suite normale du programme...
```