Introduction aux systèmes d'exploitation (IS1)

TP 8 – Expansion et échappement

1 Expansion

Quand un processus shell reçoit une ligne de commande, il pré-traite cette chaîne de caractères avant de l 'exécuter. Tout d'abord, il découpe cette chaîne de caractères, selon le séparateur « espace ». Ainsi la ligne

```
cat fic1 fic2 fic3
```

est découpée en quatre chaînes de caractères

`cat` `fic1` `fic2` `fic3`

Ce qui permet au processus shell de différencier la commande des arguments dans la chaîne de caractères qu'on lui fournit. Mais ce n'est pas tout, il existe d'autres caractères spéciaux qui permettent de demander au shell de faire d'autres pré-traitements sur la chaîne de caractères, avant de l'exécuter (on dit que le shell procède à l'*expansion* de cette chaîne de caractères). Nous avons déjà vu au TP7 que lorsqu'on fournit la chaîne de caractères `$id` au shell, le shell va la remplacer par la valeur de la variable dont l'identifiant est `id`. Voici un tableau récapitulant les différents types d'expansions en `bash`.

Syntaxe	**Expression**	**Résultat**	**Explication**
`{ }`	`ba{ba,bu}`	`baba` `babu`	énumération
`$`	`$HOME`	`/home/roger`	valeur de variable
`${...}`	`${HOME}`	`/home/roger`	valeur de variable
`$(...)`	`$(which true)`	`/bin/true`	sortie d'une commande
`` `...` ``	`` `which true` ``	`/bin/true`	sortie d'une commande
`$((...))`	`$((12 + 4 * 2))`	`20`	valeur d'une expression

Lorsqu'une chaîne contient une expression `$`, `${...}`, `$(...)` ou `` `...` `` , celle-ci est expansée puis découpée selon les séparateurs contenus dans la variable `IFS` (Internal Field Separator), qui contient habituellement les caractères «espace», «tabulation» et «retour à la ligne».

Attention dans le tableau ci-dessus, `` ` `` est un accent grave et pas une apostrophe.

Exercice 1 – *Parlez-vous* `bash` *? Leçon 1*

Créez un fichier exécutable, que vous nommerez `discours1.sh` . Ce script devra afficher dans le terminal les lignes suivantes :

```
Le chemin absolu de mon répertoire de login est <valeur donnée par le terminal>
Le résultat de la commande whoami est <résultat>
Le produit de 33 par 17 est <résultat>
Lucie a agi anticonstitutionnellement anticonventionnellement antisocialement
```

Si ce n'est pas déjà fait, modifiez votre fichier `discours1.sh` en utilisant le mécanisme d'expansion afin d'éviter d'avoir à taper la dernière ligne en entier.

Correction.

```
echo Le chemin absolu de mon répertoire de login est $HOME
echo Le résultat de la commande whoami est $(whoami) (ou encore `whoami`)
echo Le produit de 33 par 17 est $((33*17))
echo Lucie a agi anti{con{stitu,ven}tionnel,socia}lement
```

D'autres expressions sont expansées selon le contenu du répertoire courant. Par exemple, dans un répertoire contenant des fichiers nommés `ab`, `abx`, `abd`, `abdx`, `adx`, `ax` et `bdx`, on aura :

Caractères	Expression	Résultat	Explication
*	ab*	ab abx abd abdx	joker chaîne de caractères
?	ab?	abx abd	joker un caractère exactement
[...]	a[abcd]x	abx adx	liste de possibilités (un caractère)

Exercice 2 – *Parlez-vous* `bash` *? Leçon 2*

Dans un répertoire Maritime, créez les fichiers suivants :

```
butin  gratin  main  malin  marin  marine  matin  mutin  patin  patio
```

Créez un fichier exécutable, que vous nommerez `discours2.sh` . Ce script devra afficher dans le terminal les lignes suivantes, en utilisant le mécanisme d'expansion :

```
marin marine
matin patin
gratin matin patin
matin patin patio
malin marin
butin matin mutin
main malin marin matin
```

Correction.

```
ls marin*
ls ?atin
ls *atin
ls ?ati?
ls ma[lr]in
ls [bm]?tin
ls ma*in
```

Exercice 3 – *Outils pour les chaînes de caractères*

Instanciez une variable que vous nommerez `var` à la valeur `unmottrestreslong` puis affichez les expansions des expressions suivantes. Déduisez en le sens de chacune d'entre elles.

1. `${#var}`
2. `${var:4}`
3. `${var:4:3}`
4. `${var#unmot}`
5. `${var%long}`

Correction.

1. `${#var}` compte le nombre de caractères (donc ici on obtient 17)
2. `${var:4}` enlève les 4 premières lettres (donc ici on obtient ttrestreslong)
3. `${var:4:3}` enlève les 4 premières lettres et garde juste les 3 suivantes (donc on obtient ttr)
4. `${var#unmot}` enlève le préfixe unmot s'il est effectivement un préfixe (donc on obtient trestreslong)
5. `${var%long}` enlève le suffixe long s'il est effectivement un suffixe (donc on obtient unmottrestres)

2 Échappement

Nous venons de voir que certains caractères indiquent au shell qu'il doit opérer une expansion. D'autres caractères tels que ~, &, <, >, | et ; jouent également un rôle spécial lors de l'interprétation des commandes, comme nous l'avons déjà vu au cours des TP prédédents.

Pour qu'on puisse afficher tels quels ces caractères spéciaux, il existe un mécanisme d'échappement. Voici un tableau récapitulant les différents types d'échappement en `bash`.

Syntaxe	Expression	Résultat	Explication
`\`	`\"\'\$\{\ \A B`	`"'${ A` `B`	échappement d'un caractère
`"..."`	`" a $HOME $((1+1)) ' "`	`a /home/roger 2 '`	échappe tout dans une chaîne sauf $, \, ! et "
`'...'`	`' a $HOME $((1+1)) " '`	`a $HOME $((1+1)) "`	échappe tout dans une chaîne sauf '

Attention dans le tableau ci-dessus, ' est une apostrophe et pas un accent grave.

Exercice 4 – *Parlez-vous* `bash` *? Leçon 3*

Créez un fichier exécutable, que vous nommerez `discours3.sh` . Ce script devra afficher dans le terminal les lignes suivantes :

```
A      B
L'ensemble {1, 2, 3} est inclus dans N
Elle demanda d'une voix *forte* : "Qui est-ce ?"
$HOME = <le chemin absolu vers votre répertoire de login donné par le shell>
${HOME} = <le chemin absolu vers votre répertoire de login donné par le shell>
19 * 216 = <la valeur du produit calculée par le shell>
Ajoutez un peu d'huile d'olive, d'humeur joyeuse, d'humus, et remuez bien !
```

Correction.

```
echo "A     B"
echo L\'ensemble \{1,2,3\} est inclus dans N
echo "Elle demanda d'une voix *forte* : \"Qui est-ce ?\""
echo '$HOME =' $HOME
echo $\{HOME\} = $HOME
echo 19 \* 216 = \$((19*216))
echo Ajoutez un peu d\'hu{"ile d'olive","meur joyeuse",mus}, et remuez bien !
```

Exercice 5 – *Parlez-vous* `bash` *? Leçon 4*

Créez un fichier exécutable, que vous nommerez `discours4.sh` . Définissez-y une variable `TOTO` égale à `whoami`. Chaque ligne à afficher devra utiliser cette variable (il est important de respecter cette consigne). Ce script devra afficher dans le terminal les lignes suivantes :

```
J'aime bien utiliser la commande "whoami".
Ma variable $TOTO contient 'whoami'.
Mon login est <votre login donné par le shell>.
whoamitruc, c'est un bidule imaginaire !
La commande d'aujourd'hui est \whoami\.
```

Correction.

```
TOTO=whoami
echo J\'aime bien utiliser la commande \"$TOTO\".
echo Ma variable \$TOTO contient \'$TOTO\'.
echo Mon login est $($TOTO) (ou encore `$TOTO`).
echo ${TOTO}truc, c\'est un bidule imaginaire !
echo La commande d\'aujourd\'hui est \\$TOTO\\.
```

Exercice 6 – *Mon script perso*

Note : vous aurez besoin des commandes `which`, `test` et `date`. À quoi servent-elles ?

Créez un fichier exécutable, que vous nommerez `mon_script_perso.sh` . Ce script devra afficher dans le terminal les lignes suivantes :

```
Qui suis-je? <votre login donné par le shell>
Quelle est ma maison ? <votre répertoire de login donné par le shell>
Quel est mon shell? <votre shell donné par lui-même (sans son chemin d'accès complet)>
Quel jour sommes-nous? <la date du jour donnée par le shell>
Emplacement de firefox: <le chemin d'accès à votre exécutable firefox>
Somme des nombres de la date = <la somme des numéros des jour, mois et année courants>
<Une ligne indiquant si votre exécutable firefox et votre exécutable kedit
sont dans le même répertoire>.
```

Correction.

```
echo Qui suis-je ? `whoami`
echo Quelle est ma maison ? $HOME
echo Quel est mon shell ? ${SHELL##*/}
echo Quel jour sommes-nous ? `date +%x`
pathfirefox=`which firefox`
pathkedit=`which kedit`
echo Emplacement de firefox : $pathfirefox
echo Somme des nombres de la date: $((`date +%-d` + `date +%-m` + `date +%-y`))
test ${pathfirefox%/*} = ${pathkedit%/*} && echo firefox et kedit sont voisins || echo firefox
et kedit ne sont pas voisins
```

Attention dans les lignes ci-dessus, ` est un accent grave et pas une apostrophe (contrairement à ').

3 Travailler sur une arborescence avec `find`

La syntaxe de la commande `find` est la suivante :

```
find [chemin...] [expression]
```

La commande `find` parcourt les arborescences de répertoires commençant en chacun des `chemins` mentionnés, en évaluant les `expressions` fournies pour chaque fichier rencontré. Le premier argument commencant par `-`, `(`, `)`, `,`, ou `!` est considéré comme le début de l'expression, tous les arguments précédents sont considérés comme des chemins à parcourir, tous les arguments ultérieurs sont considérés comme le reste de l'expression régulière.

Si aucun chemin n'est mentionné, le repertoire en cours sert de point de depart. Si aucune expression n'est fournie, `find` utilise l'expression `-print` par défaut.

L'expression est constituée d'options, de tests et d'actions, tous ces éléments étant séparés par des opérateurs. Quand un opérateur est manquant, l'opération par defaut `-and` est appliquée.

Regardez la page de manuel de `find` pour connaître les options et les opérateurs entre options.

Les actions peuvent être de plusieurs types. Regardez la page de manuel pour compléter cette introduction.

L'action `-print` (par défaut) affiche sur la sortie standard le résultat de la commande. L'option `-exec` permet d'exécuter une commande sur chaque fichier trouvés correspondant à la demande, le fichier est désigné par `\{\}` et la fin de la commande est signalée par `\;`. Par exemple, pour connaître le numéro d'i-nœud de tous les fichiers contenant le motif `toto` de son arborescence, on peut écrire :

```
find ~ -name '*toto*' -exec ls -i \{\} \;
```

Vous noterez que le motif désignant le nom des fichiers doit être mis entre guillemets pour éviter l'expansion immédiate par le shell. De la même manière, il est nécessaire d'échapper les caractères spéciaux {, } et ;.

- {} représente le fichier trouvé par `find` et sur lequel on exécute la commande introduite par `-exec`
- ; représente la fin de la commande introduite par `-exec`

Remarque : pour échapper ces caractères spéciaux, on aurait aussi pu écrire :

```
find ~ -name '*toto*' -exec ls -i '{}' ';'
```

Exercice 7 – *Recherche par nom*

1. Listez tous vos fichiers d'extension `.java`
2. Listez tous vos fichiers dont le nom est de longueur 6
3. Listez tous vos fichiers dont le nom possède au moins 2 points
4. Listez tous vos fichiers dont le chemin absolu possède au moins 5 points
5. On a trop utilisé `xemacs` et on a plein de fichiers inutiles se terminant par ~. Ecrivez une ligne de commande permettant de supprimer tous ces fichiers de son arborescence. (Il est recommandé pour éviter les mauvaises surprises de tester votre ligne de commande sur une sous-arborescence construite pour l'occasion et d'utiliser l'option `-i` de `rm`).

Correction.

```
find ~ -name '*.java'
find ~ -name '??????'
find ~ -name '*.*.*'
find ~ -path '*.*.*.*.*.*'
find ~ -name '*~' -exec rm -i \{\} \;
```

Exercice 8 – *Autres options de recherche*

Pour résoudre les questions qui suivent, vous pourrez feuilleter la page de manuel de `find`.

1. Listez tous vos liens symboliques.
2. Listez tous vos répertoires.
3. Listez tous vos répertoires vides.
4. Listez tous vos répertoires ayant 12 liens.
5. Listez tous vos fichiers réguliers ayant 2 ou 3 liens.
6. Listez tous vos fichiers sur lesquels vous avez tous les droits, et votre groupe et les autres n'ont aucun droit.
7. Listez tous les fichiers qui sont chez vous mais qui ne vous appartiennent pas. L'affichage se fera avec `ls -l`.

Correction.

```
find ~ -type l
find ~ -type d
find ~ -type d -empty
find ~ -type d -links 12
find ~ -type f -and \( -links 2 -or -links 3 \)
find ~ -perm 700
find ~ -not -user \$USER -exec ls -l \{\} \;
```

Exercice 9 – *Modification ciblée d'une sous-arborescence du SGF*

Écrire un script `modif-droits3.sh` qui prend en paramètre un chemin et :
- si l'utilisateur rentre 0 ou plusieurs paramètres, indique à l'utilisateur qu'il doit rentrer exactement un paramètre, et quitte en renvoyant une erreur de code 18
- si le chemin ne désigne aucun fichier dans le SGF, affiche "Attention : le fichier ... n'existe pas", et quitte avec le code 19
- si le chemin désigne un fichier régulier, ajoute les droits en lecture pour tous les utilisateurs sur ce fichier et affiche "Droit en lecture ajouté à tous"
- si le chemin désigne un répertoire, ajoute les droits en écriture pour tous sur les répertoires de la sous-arborescence et affiche "Droit en écriture ajouté à tous sur les répertoires de l'arborescence"
- si le chemin désigne un autre type de fichier, ne fait rien.

Testez votre script sur une arborescence qui ne craint rien.

Correction.

```
#!/bin/bash
test $# -eq 1 || { echo "Attention : vous devez rentrer exactement 1 paramètre."; exit 18; }
test -e $1 || { echo "Attention : le fichier $1 n'existe pas"; exit 19; }
test -f $1 && chmod a+r $1 && echo "Droit en lecture ajouté à tous"
test -d $1 && find $1 -type d -exec chmod a+w \{\} \; && echo "Droit en écriture
      ajouté à tous sur les répertoires de l'arborescence"
```