Travaux dirigés Système n°1 : le Shell

Éric Gaudefroy

École Hexagone
Cursus Cyberdéfense - M1

1. Éléments de cours

1.1. Redirections

<Commande> >& <Fichier> : cette construction permet la redirection de la sortie standard (*stdout*) et de la sortie d'erreur (*stderr*) de la commande <Commande> en mode écrasement vers <Fichier>.

1.2. Recherche de fichiers exécutables

which <Commande> : la commande **which** permet de trouver l'emplacement du programme qui serait lancé si la commande Commande était invoquée. En cas de succès, l'emplacement est renvoyé sur la sortie standard (*stdout*) et **which** retourne la valeur 0. En cas d'échec, la valeur 1 est retournée.

1.3. Réussite et échec d'une commande

La notion de réussite ou d'échec d'une commande est basée sur son code de retour. Contrairement à ce qui pourrait être intuitif, notamment vis-à-vis des conventions du langage C, une commande *shell* est considérée avoir réussi lorsqu'elle renvoie 0, et est considérée avoir échoué lorsqu'elle renvoie une valeur différente de 0. Ainsi, la commande **true** renvoie toujours 0, et la commande **false** renvoie toujours 1. Par conséquent, sous *bash* :

- avec <C1> && <C2>, la commande <C2> est lancée si la commande <C1> a réussi, c'est-à-dire si son code de retour vaut 0 ;
- avec <C1> || <C2>, la commande <C2> est lancée si la commande <C1> a échoué, c'est-à-dire si son code de retour est différent de 0.

En revanche, en langage C :

- avec if (<Exp> && <F>()), la fonction <F>() est lancée si l'expression <Exp> est vraie, c'est-à-dire différente de 0 ;
- avec if (<Exp> || <F>()), la fonction <F>() est lancée si l'expression <Exp> est fausse, c'est-à-dire égale à 0.

Nota : nous pouvons cependant noter qu'en langage C, les appels système renvoient souvent 0 en cas de réussite et -1 (ou un code d'erreur plus explicite) en cas de problème.
Remarque : l'évaluation des propriétés logiques (aussi appelée évaluation booléenne), dans le cas du shell et du langage C, est paresseuse : dès que le résultat est connu, l'évaluation s'arrête. Ce n'est pas forcément le cas dans tous les langages de programmation.

1.4. Attribution d'une valeur à une variable

VARIABLE=valeur : l'attribution d'une valeur à une variable réussit toujours.

1.5. Substitution caractère par caractère

La substitution caractère par caractère peut se faire via deux commandes :

- **sed y/<Source>/<Destination>/**
- **tr <Source> <Destination>**

Ces commandes permettent de substituer chaque caractère de <Source> par le caractère correspondant de <Destination>. Les deux chaînes de caractères doivent contenir le même nombre de caractères.
Remarque : La commande **tr** accepte de plus un formalisme proche de celui des expressions rationnelles (« [a-z] », « [:lower :] »).

1.6. Recherche de fichiers dont le contenu correspond à un motif

grep -r -l <Motif> <Répertoire> : Utilisée avec ces paramètres, la commande **grep** permet de rechercher de façon récursive (**-r**) tous les fichiers dont le contenu correspond à un motif donné, et de n'afficher que leur nom, et non leur contenu (**-l**).

1.7. Création de fichier

touch <Fichier> : la commande **touch** permet de créer un fichier vide. Si le fichier existe déjà, sa date de modification est mise à jour.

2. Énoncés

On suppose que l'ensemble des fichiers traités dans les exercices suivants ne comportent que des caractères ne posant pas de problème : typiquement, on considère qu'ils ne contiennent que des caractères alphanumériques et des caractères soulignés (_).

Exercice n°1 :

Écrire une commande la plus concise possible qui copie un fichier « fichier_v1.html.tar.gz » en un fichier « fichier_v2.html.tar.gz ».

Exercice n°2 :

Écrire une commande permettant de renommer tous les fichiers du répertoire courant de la forme **<Nom>.Tab** en **<Nom>.Tab.tex**. Ainsi, **A.Tab** sera renommé en **A.Tab.tex**.

Exercice n°3 :

Écrire une commande permettant de placer dans une variable liste_fichiers les noms des fichiers situés dans l'arborescence courante et contenant la chaîne de caractères **toto**. *Nota* : c'est le fichier qui doit contenir **toto**, et non le nom du fichier.

Exercice n°4 :

Écrire une commande permettant de charger dans une unique session **vi** tous les fichiers contenus dans l'arborescence courante et contenant la chaîne de caractères **toto**.

Exercice n°5 :

Écrire un script **table.sh** qui envoie sur la sortie standard (*stdout*) la table de multiplication pour les nombres de 1 à 9. On attend une sortie de la forme suivante :

```
$ ./table.sh
    1   2   3   4   5   6   7   8   9
1   1   2   3   4   5   6   7   8   9
2   2   4   6   8   10  12  14  16  18
3   3   6   9   12  15  18  21  24  27
4   4   8   12  16  20  24  28  32  36
5   5   10  15  20  25  30  35  40  45
6   6   12  18  24  30  36  42  48  54
7   7   14  21  28  35  42  49  56  63
8   8   16  24  32  40  48  56  64  72
9   9   18  27  36  45  54  63  72  81
```

On pourra utiliser le caractère de tabulation \t et la commande **printf** pour séparer proprement les nombres.

Exercice n°6 :

Écrire un filtre **majuscule.sh** qui renvoie sur la sortie standard le flux de l'entrée standard (*stdin*) en remplaçant chaque lettre minuscule par la lettre majuscule correspondante. On attend par exemple le comportement suivant :

```
$ echo "message telegraphique" | ./majuscule.sh
MESSAGE TELEGRAPHIQUE
```

Exercice n°7 :

Écrire un script silencieux **ce.sh** (« commande existante ») qui permette d'attribuer à une variable **CE** la valeur Oui si la commande passée en argument existe, et la valeur Non si cette commande n'existe pas. Ainsi, en admettant que la commande **grep** existe et que la commande **gre** n'existe pas, le script **ce.sh** aura donc le comportement suivant :

```
$ source ce.sh grep; echo $CE
Oui
$ source ce.sh gre ; echo $CE
Non
```

Nota : pour que ce script ait un intérêt, il devra être lancé à l'aide de source pour que la modification de la variable soit effective dans l'interpréteur de commandes courant.

Exercice n°8 :

Écrire un script **pr.sh** (« parcours de répertoire ») qui balaie tous les répertoires de l'arborescence descendant du répertoire courant. Dans chacun de ces répertoires, si le fichier **Makefile** existe, le script doit lancer le **Makefile.local** à l'aide de la commande **make -f Makefile.local**, localement à ce répertoire. Si un argument est passé au script, ils sont retransmis à la commande **make**.

- **pr.sh** regarde dans chaque répertoire descendant du répertoire courant et y lance éventuellement la commande **make -f Makefile.local** ;
- **pr.sh clean** regarde dans chaque répertoire descendant du répertoire courant et y lance éventuellement la commande **make -f Makefile.local clean**.

Remarque : un tel script peut par exemple être utilisé :

- pour vérifier qu'une nouvelle bibliothèque n'empêche pas la compilation des programmes se trouvant sur l'ordinateur (**pr.sh**) ;
- pour nettoyer une arborescence en vue d'une sauvegarde (**pr.sh clean**).

Exercice n°8 :

Écrire un script prenant un nom de fichier en argument, et qui permette de supprimer toutes les lignes du texte du fichier qui comportent au moins un caractère « . » . On pourra utiliser la commande **grep** ou la commande **sed**.

Ce document est inspiré du « Système appliqué : exercices sur le shell » délivré par monsieur Jean-Yves Migeon dans les cours dispensés à l'Agence nationale de la sécurité des systèmes d'information (septembre 2016).