Langages de scripts (LS4)

TP 1 – Le *shell* `bash`

Recommandations. Il est fortement encouragé de conserver une trace de votre travail. Nous vous conseillons de créer en début de séance un répertoire `ls4` contenant un sous-répertoire pour chaque sujet de TP, et un ou plusieurs fichiers texte ou scripts bash pour chaque exercice.

1 Quelques rappels sur `bash`

1.1 Notions de base sur les variables

Une variable est repérée par un nom appelé *identificateur*, qui peut être n'importe quelle suite de caractères commençant par une lettre ou le caractère « souligné » ('_') et ne contenant que des lettres, des chiffres ou le caractère souligné[1].

Les variables `bash` ont un seul type possible, le type chaîne de caractères. On peut déclarer et affecter simultanément une variable nommée `var` avec l'instruction `var="`*`valeur`*`"` (sans espace avant ni après le signe '='). On accède à la valeur associée à une variable en faisant précéder son identificateur du symbole `$`, comme par exemple dans la commande `echo $var` qui affiche la valeur de la variable `var`.

D'autres usages plus sophistiqués des variables sont possibles (tableaux, manipulations de la valeur d'une variable, etc.). `bash` possède également un certain nombre de variables spéciales destinées à un usage bien particulier. Nous découvrirons ces concepts petit à petit en fonction de leur utilité.

1.2 Commandes et structures de contrôle

1.2.1 Commandes simples

Une commande simple se compose d'un *nom de commande* (le nom d'un fichier exécutable) suivi d'un ou plusieurs *arguments* (ou *paramètres*). Selon les commandes, certains paramètres peuvent être facultatifs ou obligatoires, et leur nombre et leur ordre peuvent avoir une importance.

Les paramètres de la forme `-x` où `x` est une lettre quelconque et les paramètres commençant par deux tirets (`--`) sont appelés *options*, ils sont en général facultatifs et servent à modifier le comportement par défaut de la commande utilisée.

Par exemple, la commande `ls -R -l ~ /usr/local/bin` possède deux options, `-R` et `-l`, et deux arguments `~` et `/usr/local/bin`. On peut en général regrouper les options à une lettre (ici on peut donc écrire `ls -Rl ~ /usr/local/bin`).

[1] À l'exception de certaines variables spéciales du shell.

On peut également composer plusieurs commandes entre elles. La syntaxe `commande1 ; commande2` exécute `commande1` puis `commande2` l'une après l'autre. D'autres méthodes pour former des commandes composées sont décrites dans le manuel de `bash`.

1.2.2 Code de retour

Il est d'usage qu'une commande informe son environnement du succès ou de l'échec de son exécution à l'aide d'un *code de retour* entier, stocké dans la variable « `?` ». On peut donc consulter le code de retour de la dernière commande ayant été exécutée en tapant `echo $?`. Le code de retour final d'une suite de commandes séparées par « ; » est celui de la dernière commande qu'elle contient.

Par convention, un code de retour égal à `0` signifie que la commande a « réussi », et un code strictement positif signifie qu'elle a échoué ou qu'une condition particulière a été rencontrée. La signification exacte du code de retour peut varier selon les commandes, elle est en générale documentée dans la page `man` de cette commande.

1.2.3 Conditionnelles

En utilisant le code de retour d'une commande, il est possible d'écrire des conditionnelles en `bash`. La syntaxe d'une conditionnelle est :
`if commande-test ; then commande1 ; else commande2 ; fi.`
Elle est exécutée de la façon suivante : premièrement, `commande-test` est exécutée. Si son code de retour est nul, alors `commande1` est exécutée, sinon c'est `commande2` qui est exécutée. On peut bien sûr imbriquer plusieurs conditionnelles.

Primitives de test. On utilise fréquemment comme tests dans des conditionnelles des commandes de la forme `[[ expression ]]`, où `expression` est formée à l'aide d'un ensemble de primitives de test. Le code de retour de ce type de commande vaut `0` si l'expression est vraie, et `1` sinon.

Les primitives existantes permettent entre autre de comparer des chaînes de caractères ou des nombres, de tester l'existence de fichiers, etc. On peut écrire par exemple

```
if [[ -e fic && ( ! -x fic ) ]] ; then echo Victoire ; fi
```

qui affiche « Victoire » si le fichier `fic` existe et n'est pas exécutable (`-e` teste l'existence, `&&` signifie « et », `-x` teste si `fic` est exécutable, et `!` exprime la négation).

D'autres primitives de test intéressantes sont les opérateurs `==` et `!=` qui permettent de comparer deux chaînes de caractères. Par exemple, `[[ "$reponse" != "jaune" ]]` teste si la valeur de la variable `reponse` est différente de la chaîne « `jaune` ». On peut utiliser dans la partie droite d'une telle expression les caractères spéciaux `*` qui représente une suite quelconque de caractères et `?` un caractère quelconque. Ainsi, la commande `[[ "$reponse" == jaun* ]]` teste si la valeur `$reponse` commence par le préfixe « `jaun` ».

La syntaxe complète de ces expressions est donnée dans la page de manuel de `bash`, dans le paragraphe sur les expressions conditionnelles (« Conditional Expressions »).

Tests triviaux. Les commandes `true` et `false` peuvent aussi servir comme commandes de test, leur code de retour vaut toujours 0 et 1 respectivement.

1.2.4 Boucles

Les boucles `bash` sont principalement de deux types (il existe des variantes moins courantes dont nous ne nous servirons pas).

Boucles « pour tout ». La syntaxe `for var in liste ; do commande ; done` affecte successivement à la variable `var` toutes les valeurs apparaissant dans `liste`. Pour chacune de ces valeurs, `commande` est exécutée.
Par exemple :

```
for carte in as 2 3 4 5 6 7 8 9 10 valet dame roi; do
    for couleur in trefle carreau coeur pique; do
        echo "$carte de $couleur"
    done
done
```

affiche la liste des cartes d'un jeu de 52 cartes.

Boucles « tant que ». La syntaxe `while commande-test ; do commande ; done` commence par exécuter `commande-test` et par évaluer son code de retour. S'il est différent de `0`, rien ne se passe. S'il vaut `0` en revanche, alors `commande` est exécutée, et l'opération est répétée depuis le début : `commande-test` est exécutée à nouveau, son code de retour évalué, etc.

Le fonctionnement des boucles `while` est un peu similaire à celui des conditionnelles. En particulier on utilise très souvent les primitives de test pour écrire la condition de boucle `commande-test`. Par exemple :

```
while [[ "$reponse" != "jaune" ]]; do
    echo "devinez à quelle couleur je pense"
    read reponse[2]
done
echo "bravo !"
```

demande à l'utilisateur de deviner une couleur jusqu'à ce que la réponse soit correcte.

1.3 Substitution de commandes

Il est fréquent de vouloir utiliser dans une commande les résultats affichés par une autre commande. Pour cela, on utilise l'une des syntaxes `$(commande)` ou `` `commande` ``[3], qui, lorsqu'elles apparaissent dans une autre commande, sont remplacées par le texte qu'afficherait `commande` si on l'exécutait dans un terminal.

Par exemple, si le répertoire courant est `/home/dugenou42` alors la commande `echo "Vous êtes dans le répertoire" $(pwd)` affichera « `Vous êtes dans le répertoire /home/dugenou42` »[4].

[2] La commande `read` suivie du nom d'une variable permet de saisir une valeur au clavier et de la stocker dans cette variable.

[3] Le caractère « ` » est appelé *backquote*, on l'obtient par `AltGr-7` sur un clavier français.

[4] La commande `pwd` affiche en effet le répertoire courant.

1.4 Structure d'un script

Une particularité des *shells* Unix comme `bash` est qu'ils permettent très facilement de combiner les outils de base de l'environnement pour créer ses propres commandes, appelées scripts. Ils obéïssent à quelques conventions :

- ils contiennent des séquences de commandes dans la syntaxe habituelle. Les retours à la ligne sont interprétés comme des points virgules ;
- ils commencent par une ligne permettant d'identifier le programme à utiliser pour les exécuter. Pour `bash`, la première ligne doit être : `#!/usr/local/bin/bash`[5]
- toute portion de ligne apparaissant à droite d'un caractère `#` est considérée comme un commentaire. Il est utile de commenter chaque partie importante d'un script ;
- pour pouvoir être exécuté, un script doit posséder les droits en **exécution** et en **lecture** pour l'utilisateur.

Gestion des paramètres. Les scripts que vous réalisez vous-mêmes peuvent accepter des paramètres, comme toute commande habituelle. Pour y accéder, on utilise des variables spéciales du shell, dont voici une courte liste :

- la variable `#` contient le nombre de paramètres passés au script ;
- la variable `0` contient le nom du script en cours d'exécution ;
- pour chaque entier `i` entre 1 et `$#`, la variable `i` contient le i-ème paramètre.
- la variable `@` contient la liste de tous les paramètres séparés par des espaces[6].

Par exemple, si l'on tape la commande `monscript machin bidule` dans un shell, `$#` vaudra 3, `$0` vaudra `monscript`, `$1` prendra la valeur `machin` et `$2` la valeur `bidule`.

Dans certains cas, on ne connaît pas à l'avance le nombre de paramètres qui seront reçus par le script. Dans ce cas, on peut utiliser la commande `shift`, qui permet de décaler tous les paramètres d'un « cran », en oubliant le premier paramètre. Ainsi, sur l'exemple précédent, si l'on exécute `shift`, la nouvelle valeur de `$1` est `bidule`, `$2` vaut `truc` et `$3` n'est plus affectée.

[5] Ceci dépend de la machine sur laquelle on travaille. Cette ligne est valable sur les machines du script, mais pas sur le système Juppix par exemple. Pour déterminer l'emplacement de l'exécutable `bash`, tapez `which bash` dans un terminal.

[6] Attention, il faut toujours utiliser cette variable **entre guillemets doubles** !

2 Exercices

Exercice 1 – *Documentation*

Toutes les commandes que vous serez amenés à utiliser disposent d'une documentation en ligne accessible grâce à la commande `man` *`nom de commande`*.

1. Consultez la documentation[7] de la commande *man*. Quelle option de `man` permet de rechercher un mot-clé dans l'ensemble des pages de manuel ?
2. L'affichage proprement dit des pages de manuel est effectué par la commande `less`. Consultez le manuel de `less`. Quels raccourcis clavier permettent de faire défiler le texte d'une ligne ? D'une page ? Comment peut-on effectuer une recherche vers l'avant ? Vers l'arrière ? Quelle touche permet de quitter le manuel ?

Exercice 2 – *Système de fichiers*

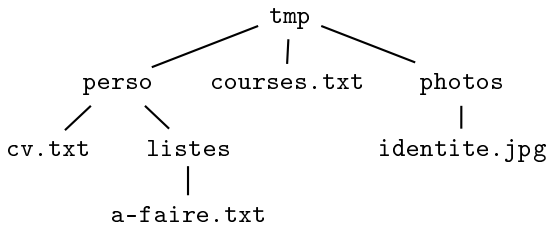
1. Parcourez rapidement les pages de manuel des commandes `ls`, `cd`, `pwd`, `cp`, `mv`, `rm`, `touch`, `mkdir` et `rmdir`. Décrivez en 3 mots leur fonction.
2. Quelle est la différence entre un nom de fichier absolu et un nom relatif ?
3. Quels répertoires désignent les raccourcis '~', '.' et '..' ? Ces raccourcis sont-ils des noms absolus ou relatifs ?
4. Donnez le nom absolu des fichiers ou répertoires suivants vus depuis votre répertoire « maison » (ou répertoire personnel) :

   `../usr/../bin/ls` `../glambert/mobylette.jpg` `./ls4/tp1`

5. Donnez le nom relatif des fichiers ou répertoires suivants par rapport au répertoire `/usr/lib` :

   `~/perso` `/usr/bin/xclock`

6. Reproduisez à l'aide de `touch` et `mkdir` mais *sans utiliser la commande* `cd` l'arborescence suivante dans le répertoire `/tmp` :



7. Déplacez le fichier `courses.txt` vers `listes`, puis recopiez le répertoire `photos` et tout ce qu'il contient dans `perso`.

[7] Des traductions françaises de certaines pages de manuel sont disponibles sur le web (liste à l'adresse `http://fr.wikipedia.org/wiki/Man_(commande_Unix)`). Attention cependant aux différences de version !

Exercice 3 – *Un script simple.*

1. Créez un script `bonjour.sh` qui, lorsqu'il est lancé, affiche le texte « Bonjour tout le monde ! ».
2. Ajoutez un commentaire indiquant l'auteur (vous !) et la date de création du script. Vous pouvez garder cette habitude pour tous les scripts que vous créez ou modifiez.
3. La commande `whoami` permet de connaître le nom de connexion de l'utilisateur qui la lance. Modifiez votre script pour qu'il affiche « Bonjour dugenou42 » si votre nom de connexion est `dugenou42`.
4. La commande `date` permet d'afficher la date et l'heure actuelles. Modifiez votre script pour qu'il demande à l'utilisateur `Souhaitez-vous afficher la date (o/n) ?` puis affiche la date si la réponse est `o`.

Exercice 4 – *Comptage des paramètres.*

On veut écrire un script `param.sh` qui compte le nombre de ses paramètres et affiche leur valeur. Par exemple, si l'on tape la commande `param.sh un deux trois machin "bidule truc"`, une sortie possible est :

```
Le paramètre 1 est un.
Le paramètre 2 est deux.
Le paramètre 3 est trois.
Le paramètre 4 est machin.
Le paramètre 5 est bidule truc.
Le script a reçu 5 paramètres.
```

1. Réalisez ce script à l'aide d'une boucle `for` et de la variable `$@`.
2. Réalisez un script `param-shift.sh` qui fait la même chose, cette fois à l'aide de la commande `shift`.
3. Créez un nouveau script `param-options.sh` qui compte et affiche séparément ses options et ses paramètres. Par exemple, `param-options.sh -a -b machin bidule` doit afficher :

```
L'option 1 est -a.
L'option 2 est -b.
Le paramètre 1 est machin.
Le paramètre 2 est bidule.
Le script a reçu 2 options et 2 paramètres.
```

Exercice 5 – *Compilation de programmes*

Exercice pouvant compter pour le contrôle continu.

1. Écrivez un script `run-java.sh` prenant en argument un nom de **classe** Java, qui lance une compilation du fichier `.java` correspondant et exécute le programme.
2. Ajoutez à votre script un test vérifiant qu'il a bien reçu un (et un seul) argument avant de tenter une compilation. Si ce n'est pas le cas, affichez un message indiquant l'usage correct du script.
3. Ajoutez à présent un test qui vérifie si le fichier Java à compiler existe et est lisible, et si le fichier `.class` à produire peut bien être créé. Affichez un message d'erreur si ce n'est pas le cas.
4. Ajoutez un test à votre script afin de faire en sorte que le programme ne soit pas recompilé si le fichier `.class` est plus récent que le fichier `.java` concerné.
5. Enfin, ajoutez la possibilité de passer en paramètre un nombre quelconque de noms de classes Java. Le script doit alors compiler tous les fichiers sources correspondants (si nécessaire) et lancer le programme principal correspondant au premier paramètre.
6. (*Facultatif*) En général, une classe java est définie dans un fichier portant son nom, par exemple la classe `Arbre` dans le fichier `Arbre.java`. Cependant, cette convention n'est pas toujours respectée. En utilisant l'outil `grep`, améliorez votre script pour qu'il détecte quel fichier source contient réellement le code de la ou des classe(s) passées en paramètre.

Exercice 6 – *Localiser un fichier*

Écrivez (**sans utiliser la commande** `find`) un script `trouve.sh` qui recherche dans le répertoire courant et ses sous répertoires le fichier dont le nom est passé en paramètre, et affiche son nom absolu si ce fichier est trouvé.

Conseils : On ne prendra en compte que les fichiers réguliers et les répertoires (et pas les liens logiques par exemple). On peut envisager au moins deux solutions à ce problème :

- Faire de `trouve.sh` un script récursif. On pourra retrouver le nom relatif du fichier `trouve.sh` par rapport au répertoire courant grâce à la variable `$0`, qui indique par quel nom le script a été appelé.
- Utiliser l'option de `ls` qui effectue une liste récursive des sous-répertoires. Dans ce cas, les commandes `grep` et/ou `basename` pourront être utiles.