1 Transformation de la ligne de commandes et code de retour

Comme vu en cours, avant d'exécuter une commande le shell transforme la ligne qu'il reçoit. Il la développe en trois étapes successives (aussi bien en mode interactif que dans un script) :

1. **expansion des paramètres et variables** : les chaînes de la forme `${MOT}` ou `$MOT` sont remplacées par la valeur attribuée à `MOT` ;
2. **substitution des commandes** : les chaînes de la forme `$(MOT1 MOT2 ...)` sont remplacées par le résultat de (*i.e.* l'affichage produit par) l'exécution de la ligne composée des mots `MOT1 MOT2 ...` ;
3. **expansion des chemins** : les chaînes comprenant des caractères jokers (`*`, `?`, `[`, `]`, `-` et `^`) sont remplacées par les noms de fichiers qu'elles modélisent s'il en existe.

De nombreux caractères sont donc utilisés par le shell pour ces développements et le découpage en mots. Il est possible de protéger ces caractères (*i.e.* forcer le shell à les voir comme des caractères standards) de 3 manières possibles :
— en les préfixant chacun par un `\` ;
— en entourant des caractères par des quotes simples, `'` ;
— en entourant des caractères par des quotes doubles, `"` (dans ce cas les caractères `$`, `\` et `` ` `` continuent d'être utilisés par le shell).

Exercice 1.1 : Substitution de variables

Q1. Stockez votre identifiant dans une variable `LOGIN`.

Q2. Affichez le contenu de la variable `LOGIN`.

Q3. En utilisant la commande précédente comme commande d'affichage et un filtre, déterminez votre nom de famille.

Q4. Changez le contenu de la variable `LOGIN` pour l'identifiant d'un de vos camarades puis exécutez à nouveau la commande de la question précédente. Obtenez-vous bien son nom de famille ?

Exercice 1.2 : Substitution de commandes

Q1. Ouvrez `xeyes` en arrière-plan.

Q2. Trouvez une ligne de commande permettant de retourner uniquement le numéro de jobs du dernier `xeyes` lancé. Assurez-vous que votre ligne de commandes fonctionne même si vous avez d'autres programmes de lancés depuis votre terminal. (Si vous ne trouvez pas comment faire, retournez aux TP03 et TP04.)

Q3. En utilisant la commande précédente (mais sans utiliser son résultat, donc sans utiliser le numéro de jobs explicitement), terminez le dernier processus `xeyes` que vous avez lancé. Pour répondre à cette question, vous DEVEZ effectuer une substitution de commandes.

Q4. Ouvrez à nouveau `xeyes` en arrière-plan, puis en utilisant la même ligne de commandes que précédemment, vous devriez fermez cette instance de `xeyes`.

Q5. Faites de même en utilisant l'identificateur de processus plutôt que le numéro de jobs.

Exercice 1.3 : Jokers

Un **joker** est un caractère utilisé *à la place* d'un (ou de plusieurs) autre(s) caractère(s). Un **modèle** (*glob pattern*) est un mot qui contient un ou plusieurs jokers. Lorsque le shell trouve un modèle sur la ligne de commande :

1. il cherche la liste des fichiers dont le **chemin correspond** au modèle
2. il remplace le modèle par les **chemins des fichiers** correspondant en les séparant par **un** espace.
3. si aucun fichier ne correspond, le modèle est laissé tel quel

Il existe plusieurs jokers :
— `*` remplace n'importe quelle **suite de caractères** (y compris vide)
— `?` remplace n'importe quel **caractère**

— `[` ⟨liste⟩ `]` remplace n'importe quel caractère de ⟨liste⟩
  — on spécifie la liste des caractères que l'on veut représenter
  — `^` placé en début de liste signifie que l'on veut remplacer n'importe quel caractère **non présent** dans la liste
  — `-` utilisé dans la liste définie un intervalle plutôt qu'un ensemble de valeurs

Le but de cet exercice est de manipuler les jokers. On donnera pour exemple afin d'illustrer ce qu'il se passe que "`echo /*`" affiche l'ensemble des dossiers présent à la racine sous la forme `/nom_du_dossier`.

Chaque étudiant a son répertoire de travail dans un répertoire de `/home` dont le nom se termine par `etu`. Par exemple les étudiants du département informatique sont dans `/home/infoetu`, ceux du département génie biologique sont dans `/home/bioetu`, etc.

On rappelle que la séquence `\n` est considérée comme le caractère de passage à la ligne par la plupart des commandes.

Q1. Écrivez les lignes de commande, commençant par `echo`, qui permettent **d'afficher** les **chemins absolus** des répertoires principaux (un par ligne) de :

1. tous les étudiants du département informatique :
2. tous les étudiants :
3. tous les étudiants du département informatique dont le nom commence par le caractère `b` et le prénom commencent par le caractère `j` ;
4. tous les étudiants du département informatique dont le nom commence par le caractère `b` et le prénom commencent par le caractère `j` ou le caractère `m` ;
5. tous les étudiants du département informatique dont le prénom commence par une lettre entre `a` et `c` et le nom commencent par une lettre qui n'est pas entre `a` et `c` :
6. tous les étudiants dont le prénom fait exactement `5` caractères et dont le prénom commencent par le caractère `y` et le nom commence par la lettre `d` ;
7. tous les étudiants dont le login contient au moins une fois le caractère `b` ;
8. tous les étudiants dont le login contient au moins deux fois le caractère `b`.

Exercice 1.4 : Expension et Protections

Q1. Stockez le texte `echo machin` dans une variable `TOTO`.

Q2. En utilisant la variable `TOTO`, affichez sur la sortie standard le texte `machin`.

Q3. En utilisant `echo` et la variable `TOTO`, et sans écrire `machin`, affichez sur la sortie standard le texte :
`La variable TOTO contient "echo machin".`

Q4. En utilisant `echo` et la variable `TOTO`, et sans écrire `machin`, affichez sur la sortie standard le texte :
`$TOTO vaut "echo machin".`

Exercice 1.5 : Code de retour (*exit status*)

À la fin de l'exécution d'un processus, le shell conserve dans son contexte son code de retour. Il est accessible lors de l'expansion de la ligne suivante dans `$?`. Par convention on le considère comme le nombre d'erreurs rencontrées par le processus.

Une exécution **sans erreur** (un code d'erreur égal à 0) correspond à une **réussite**. Une exécution avec **une ou plusieurs erreurs** (un code d'erreur différent de 0) correspond à un **échec**.

Le code de retour d'un script est le code de retour de la dernière commande qu'il a exécutée.

En bash, la notion de booléen n'existe pas. Si vous devez faire des tests conditionnels, c'est le code de retour de la commande testée qui sera considérée. Un code `0` est une réussite et correspond à *vrai* et un code différent de `0` est un échec et correspond à *faux*.

Il est possible de chaîner les commandes les unes aux autres en les conditionnant à leur réussite ou à leur échec :
— la commande suivant `&&` n'est exécutée que si celle qui la précède réussit ;
— la commande suivant `||` n'est exécutée que si celle qui la précède échoue.

Q1. Dans votre terminal écrivez la ligne de commande :
`PS1='[Code de retour de la commande : $?]\n'$PS1`

Désormais, après l'exécution de vos commandes, votre prompt vous affiche en plus le code de retour de la dernière commande exécutée.

Q2. Écrivez et exécutez une commande dont le code de retour est `0`.

Q3. Écrivez et exécutez une commande dont le code de retour est une autre valeur que `0`.

Q4. Utilisez la commande `cat` sans argument puis testez deux approches différentes pour terminer son utilisation : En tuant le processus et en fermant l'entrée standard. Comment avez-vous procédé ? Que constatez-vous ?

Q5. Le fichier `/dev/null` fait disparaitre tout ce qui est écrit dedans. Sachant cela, qu'affichent et quels sont les codes de retours des commandes suivantes :

— `cat Shlaguevuk 2>/dev/null 1>&2 && echo Bleu`
— `cat Shlaguevuk 2>/dev/null 1>&2 || echo Rouge`

Q6. Dans quel cas la commande suivante affichera le mot `Jaune`

```
cat untruc || echo Jaune
```

2 Les scripts

Sous Unix il est possible d'écrire des programmes en shell. Pour cela il suffit de regrouper dans un fichier texte (*méthode* Unix) des lignes de commandes en s'assurant que :

1. la **première ligne** du fichier définit le *langage* à utiliser : `#!/bin/bash`,
2. le fichier doit être exécutable,
3. le shell puisse trouver le fichier.

Rappel : Pour rendre le fichier `toto` exécutable par l'utilisateur on doit appliquer la commande : `chmod u+x toto`

Rappel : Pour exécuter un fichier comme une commande, il faut donner le chemin (absolu ou relatif) vers ce fichier. Si vous utilisez un chemin relatif, il faut impérativement qu'il contienne au moins un caractère `/`. Donc si vous voulez exécuter le fichier `toto` et que vous êtes déjà dans le répertoire contenant `toto`, vous l'exécuterez en utilisant : `./toto`.

Lors de l'exécution du programme, certaines chaînes de caractères sont remplacées par les éventuels paramètres de la ligne ayant servi à démarrer la commande :

— `$0`, `$1`, `$2`, etc. par le premier, second, troisième, etc. mots présents sur la ligne de commandes ;
— `$#` par le **nombre** d'arguments présents sur la ligne de commandes ;
— `$*` ou `"$@"` par l'**ensemble des arguments** présents sur la ligne de commandes.

Dans ce contexte la commande `shift` permet d'oublier certains des arguments. Quand elle est appelée sans paramètre, elle permet d'oublier le premier paramètre en réadaptant ces chaînes de caractères (le contenu de `$1` devient celui de `$2`, le contenu de la `$2` devient celui de `$3`, etc.) Dans ce cas le contenu de `$*`, `$#` et `"$@"` sont également mis à jour.

Exercice 2.1 : Premiers scripts

Q1. Écrivez un script `bonjour` qui lorsqu'il est executé retourne : `Bonjour le monde !`

Q2. Écrivez un script `tout` qui, lorsqu'il est exécuté, retourne l'ensemble des arguments qui lui sont passé en paramètre.

Q3. Écrivez un script `premier` qui, lorsqu'il est exécuté, retourne uniquement le premier argument qui lui est donné en paramètre. (Attention on veut le premier argument et pas le nom du programme `premier`.)

Q4. Écrivez un script `nombre` qui lorsqu'il est exécuté, retourne le nombre d'arguments qui lui sont donné en paramètre.

Exercice 2.2 : Quelques scripts simples

Dans cet exercice vous allez travaillez sur la base des utilisateurs du système du département.
Les logins des utilisateurs étudiants sont de la forme `prenom.nom.etu` où

— `prenom` correspond au prénom de l'utilisateur
— `nom` correspond au nom de l'utilisateur éventuellement suivi d'un chiffre pour éviter d'affecter un même login à deux utilisateurs différents.

Q1. Déterminez une ligne de commande permettant d'afficher le prénom et le nom de l'utilisateur `toto.passoire.etu`. Cette commande devra commencer par `echo toto.passoire.etu`.

Q2. Déterminez une ligne de commande permettant d'afficher le prénom et le nom de l'utilisateur `toto.passoire5.etu`. Cette commande devra commencer par `echo toto.passoire5.etu`.

Q3. Écrivez un script `prenom` (dans votre dossier `TP05`) qui prend en paramètre un *login* et qui affiche le prénom de l'utilisateur concerné.

Q4. Écrivez un script `nom` (toujours dans votre dossier `TP05`) qui prend en paramètre un *login* et qui affiche le nom de l'utilisateur concerné.

Q5. Écrivez un script `identite` qui prend en paramètre un *login* et qui affiche le nom puis le prénom de l'utilisateur concerné. Appliqué à `toto.passoire5.etu`, votre script devra afficher : `passoire toto` dans cet ordre.

Exercice 2.3 : Scripts et expansions

La commande `getent group toto` affiche une ligne d'information sur le groupe `toto` avec la même structure que ce qui est décrit dans `group(5)`.

Les groupes pédagogiques du département sont identifiés sous la forme `info-butN-L` avec `N` le nombre 1 ou 2 (correspondant à la première ou deuxième année du BUT) et `L` une lettre minuscule entre a et j (correspondant au groupe).

Q1. Écrivez un script `logins` (toujours dans votre dossier `TP05`) qui prend un paramètre de la forme `NL` (avec `N` le nombre 1 ou 2 et `L` une lettre en capitale entre A et J) et qui affiche les logins de tous les étudiants du groupe désigné en en affichant un seul par ligne.

NOTE : Il pourra être judicieux de créer des variables intermédiaires.

Q2. Ecrivez un script `calculer` qui prend 3 paramètres correspondant à un calcul arithmétique (addition, soustraction, multiplication ou division) et affiche l'opération **et** son résultat correctement. Par exemple l'appel de `calculer 3 * 4` doit produire l'affichage `3 * 4 = 12`. La lecture de `expr(1)` vous sera utile.

Attention : La comprehension et la gestion des protections est toute la difficulté de l'exercice.

Exercice 2.4 : Scripts et codes de retour

Lorsque vous écrivez un script, il y a deux éléments qui peuvent vous intéresser lors de l'execution : Ce que retourne le script sur la sortie standard et son code de retour. Dans cet exercice, nous allons nous intéresser uniquement au code de retour, à tel point que l'on va explicitement demandé que les scripts ne produisent pas d'affichage. Afin de vérifier les codes de retour de vos scripts, assurez-vous que ce dernier est toujours affiché avec votre prompt, si ce n'est pas le cas, retournez à l'exercice 1.5.

> Les scripts que vous écrivez dans cet exercice ne doivent produire **aucun** affichage.

Comme dit précédemment, le fichier `/dev/null` fait disparaitre tout ce qui est écrit dedans. Vous pouvez donc rediriger vos sorties vers ce fichier.

Q1. Écrivez une commande, nommée `pareil`, qui ne réussit que si les deux paramètres qu'elle doit recevoir sont identiques.

Comprendre la valeur de sortie de `expr(1)` pourrait vous aider.

ATTENTION : Pour vérifier votre commande, il faut l'appeler, constater qu'il n'y a pas d'affichage **puis** utiliser la commande `echo $?` qui doit afficher 0 si les paramètres sont identiques et 1 sinon. Testez au moins avec `pareil toto toto`, `pareil toto tata`, `pareil toto -c`, `pareil -c toto`, `pareil -c -c`.

Q2. Écrivez une commande, nommée `est-vide`, qui ne réussit que si elle est appelée sans paramètre.

Q3. Écrivez une commande, nommée `est-repertoire`, qui ne réussit que si le paramètre qu'elle doit recevoir est un chemin valide vers un répertoire. La lecture de `stat(1)` vous aidera.

Q4. Écrivez une commande, nommée `est-executable`, qui ne réussit que si le paramètre qu'elle doit recevoir est un fichier (de n'importe quel type) dont le droit `x` est fixé pour son propriétaire.

Q5. Écrivez une commande, nommée `est-nombre-entier`, qui ne réussit que si le paramètre qu'elle doit recevoir est un nombre entier.

Q6. Écrivez une commande, nommée `existe`, qui ne réussit que si elle reçoit en paramètre un chemin vers un fichier qui existe.

3 Structures de Contrôle

Les structures de contrôle en `bash` ont un fonctionnement qui est différent du `ijava`. Il existe deux grosses différences :

- En `bash`, il n'y a pas de booléen, les tests conditionnels se basent sur le code de retour de la commande testée. Le code `0` correspond à *vrai* et les autres codes correspondent à *faux*.
- La boucle `for` en `bash` ne fonctionne pas du tout de la même manière qu'en `ijava`. En effet en `bash`, c'est une variable qui prend successivement la valeur des différents éléments listés par la boucle `for`.

Rupture de séquence

La séquence d'instructions peut être rompue grâce à la structure alternative qui conditionne l'exécution d'un bloc (ou d'un autre) à la réussite (ou l'échec) d'une commande préalable :

```
if <cmd-si> ; then ; <cmds-if> ; else <cmds-else> ; fi
```

```
if <cmd-si>
then
   <cmds-if>
else
   <cmds-else>
fi
```

Répétitions

La structure `for` permet de spécifier une liste de valeurs pour lesquelles un bloc de commandes doit être répété en affectant chaque valeur à une variable spécifiée. La liste de valeurs à utiliser est construite une seule fois par les expansions du shell avant les répétitions.

```
for var in <liste>
do
    <cmds>
done
```

La structure `while` répète une suite d'instructions tant qu'une commande spécifiée réussit.

```
while <cmd-tq>
do
    <cmds-while>
done
```

Opérateurs logiques

— la commande suivant `&&` n'est exécutée que si celle qui la précède réussit ;
— la commande suivant `||` n'est exécutée que si celle qui la précède échoue ;
Par ailleurs une commande précédée de `!` voit sa réussite transformée en échec et son échec transformé en réussite.

Exercice 3.1 : Comparaison des deux types de boucles

Q1. Écrivez deux commandes, `compter-for` et `compter-while`, permettant d'afficher les entiers de 1 à 10, un sur chaque ligne. `compter-for` devra utiliser le contrôle `for` et `compter-while` le contrôle `while`. Lire `expr(1)` pourra vous être utile dans un des deux cas.

La commande `time` permet d'évaluer le temps d'exécution d'une commande. Pour plus de détails sur cette commande référez-vous à la `time(1)`.

Q2. Comparez le temps d'exécution de `compter-for` et `compter-while`.

Q3. Écrivez deux commandes, `parametres-for` et `parametres-while`, qui affichent chacune les paramètres qu'elle reçoivent, **un par ligne**, en les préfixant par leur position sur la ligne de commande. `parametres-for` devra utiliser la structure `for` et `parametres-while` la structure `while`. La compréhension de l'instruction `shift` est indispensable pour réaliser `parametres-while`.

Exercice 3.2 : Entraînement

Q1. Écrivez une commande `options` qui n'affiche que les options qu'elle reçoit. On rappelle qu'une option est un paramètre qui commence par le caractère `-`.

Q2. Écrivez une commande `arguments` qui n'affiche que les arguments qu'elle reçoit. Elle ne doit afficher qu'une seule ligne contenant tous ses arguments. On considère ici que dès le premier paramètre qui n'est pas une option tous les autres (lui inclus) sont des arguments.

Exercice 3.3 : Lister des fichiers

On rappelle que la commande interne `exit` permet de quitter immédiatement le shell en cours qu'il soit interactif ou script.

Q1. Écrivez une commande `est-document` qui réussit si et seulement si elle reçoit un paramètre, que celui-ci correspond à un chemin de fichier existant et que ça **n'est pas** un dossier. Elle ne doit produire aucun affichage.

Q2. Écrivez une commande `est-script` qui réussit si et seulement si elle reçoit un paramètre, que celui-ci correspond à un chemin de fichier existant et que c'est un script shell. Pour mémoire un script shell est un fichier qui est exécutable et dont la première ligne est exactement `#!/bin/bash`. Elle ne doit produire aucun affichage.

Q3. Écrivez une commande `lister-fichiers` qui affiche les noms des fichiers d'un certain type contenu dans un dossier. Elle prend deux paramètres : le premier est un type, le second un chemin vers un dossier. Les types autorisés sont `repertoire`, `document` et `script`. La commande doit traiter correctement les fichiers dont les noms contiennent des espaces. Elle doit échouer et ne rien afficher si :
— elle ne reçoit pas assez ou trop d'arguments ;
— si le type reçu n'est pas autorisé ;
— si le chemin reçu est inaccessible ou pas un dossier.

Q4. Écrivez une commande `filtrer-fichiers` qui n'affiche que les fichiers d'un certain type parmi tous ceux dont elle reçoit les chemins en paramètre. Le premier paramètre est du même type que pour la commande `lister-fichiers`. Les suivants sont les chemins des fichiers à filtrer. La commande doit traiter correctement les fichiers dont les noms contiennent des espaces. Elle doit réussir et ne rien afficher si aucun chemin ne lui est donné. Elle doit échouer et ne rien afficher si :

— elle ne reçoit pas assez d'arguments ;
— si le type reçu n'est pas autorisé.

Affectation via l'entrée standard

La commande interne `read` lit sur l'entrée standard (et uniquement l'entrée standard) **une** ligne (une suite de caractères terminée par un retour-charriot) et affecte les mots lus (suite de caractères sans espace) aux *variables* dont les noms lui sont passés en paramètre.

Chaque variable est affectée avec un seul mot, sauf la dernière qui reçoit tous les mots restant après affectation des autres variables. Les variables ne sont visibles que dans l'environnement du processus dans lequel la commande `read` s'exécute.

Si l'entrée standard est vide, elle échoue, sinon elle réussit.

Vous pouvez étudier sa documentation (par exemple via `help read`) pour en comprendre le fonctionnement.

Exercice 3.4 : Entraînements

Utiliser les contrôles `for` ou `while` et la commande `read` pour écrire les scripts de cet exercice.

Q1. Écrivez une commande `prefixer` qui préfixe toutes les lignes qu'elle reçoit via l'entrée standard par un mot qui lui est donné en premier paramètre.

Q2. Écrivez une commande `bouts` qui affiche uniquement la dernière **puis** la première ligne qu'elle reçoit sur l'entrée standard.

Q3. Écrivez une commande `lignes` qui affiche uniquement les lignes qu'elle reçoit sur l'entrée standard et dont les numéros lui sont passés en paramètres. On considère, sans le vérifier, que les numéros sont donnés dans l'ordre croissant.