INF203 - Travaux pratiques, séance 11
Développement d'un interpréteur de commandes (1/2)

Dans ce TP, nous allons initier un projet de développement d'un interpréteur de commandes réduit qui s'étalera sur deux semaines. Les fichiers de commandes traités cette semaine seront constitués de suites de commandes en séquence, avec variables et, à la fin du TP, avec utilisation des arguments. Nous ne passerons aux structures de contrôle que la semaine prochaine. Tout au long du TP, vous pouvez écrire vos propres fichiers `.sh` que vous utiliserez pour vos tests en plus de ceux qui vous seront fournis. Attention, comme dans l'exemple `test_1_lecture_ligne.sh`, ne commencez pas le fichier par un commentaire spécial comme `#!/bin/bash` (*votre* interpréteur ne reconnaîtra pas les commentaires).

1 Boucle de lecture

`[TP11]` Commencez par observer `interpreteur.c`. Le programme principal (`main`), après quelques initialisations, commence par ouvrir le fichier d'entrée : entrée standard ou fichier de nom donné, selon le nombre d'arguments passés. Ensuite, le programme passe à une boucle de lecture et interprétation qui ne se terminera qu'à la fin du fichier d'entrée. Chaque ligne lue est passée à la fonction `executer_ligne_commande`.
[a] Quelles étapes de traitement `executer_ligne_commande` applique-t-elle à chaque ligne lue ? ■

La lecture d'une ligne et les fonctions correspondant aux trois premières étapes d'`executer_ligne_commande` sont pour l'instant incomplètes ou incorrectes et l'interpréteur n'est pas encore fonctionnel. Seule la fonction `exectuer_ligne_decoupee` est déjà écrite car elle fait appel à des notions de système que nous n'avons pas vu. Dans la suite de ce TP, nous allons compléter au fur et à mesure les fonctions manquantes.
[b] Observez les fichiers `ligne.c` et `variables.c` et dites ce que font pour l'instant la lecture de ligne, l'expansion des variables, le découpage de la ligne et la reconnaissance d'une affectation. ■

2 Lecture d'une ligne

[c] Dans le fichier `lignes.c`, complétez la fonction `lire_ligne_fichier` selon la spécification indiquée en commentaire. ■
Pour tester, utilisez le `Makefile` fourni et exécutez `interpreteur` en lui donnant le nom d'un script de test en argument. Vous pouvez commencer par le fichier fourni, `test_1_lecture_ligne.sh` ou un fichier que vous aurez écrit. Seules les commandes sans arguments et sans variables fonctionnent pour l'instant. Vous pouvez constater que la lecture de chaque ligne du fichier est suivie d'un affichage à l'écran de la ligne lue et d'un affichage des détails de l'exécution (nom de commande, arguments et code de retour). Vous pouvez supprimer ces affichages détaillés en enlevant le `-DDEBUG` utilisé lors de la compilation. Vous pouvez même n'enlever le `-DDEBUG` que pour certains fichiers : les messages de debug ne seront supprimés que pour les fonctions contenues dans ces fichiers.
[d] Que constatez vous lors de l'exécution du fichier `test_1_lecture_ligne.sh` ? ■

3 Gestion d'un ensemble de variables

Nous allons maintenant nous attaquer à la gestion des variables : un ensemble de couples de chaînes de caractères où chaque élément correspond au couple `(nom, valeur)` associé à une variable. Cette partie ressemble très fortement à l'exercice sur les joueurs de billes du TP8, n'hésitez pas à vous en inspirer.
[e] Écrivez les fonctions décrites dans le fichier `variables_base.h` dont l'implémentation (incomplète) se trouve dans le fichier `variables.c`. Utilisez le programme `main_test_vars` pour tester votre solution. ■

Il s'agit ensuite de reconnaître les lignes de commandes qui correspondent à une affectation de variable : si la ligne de commande contient un caractère `'='` alors la partie à gauche de l'affectation est un nom de variable et celle à droite est sa valeur.
[f] Complétez la fonction `trouver_et_appliquer_affectation_variable` dans le fichier `variables.c`. Ajoutez des messages de *debug* pour vous assurer que l'affectation est faite le cas échéant. ■

Enfin, il nous faut procéder à l'expansion des variables dans les lignes de commande lues. Pour cela, chaque ligne lue doit être copiée dans une ligne expansée dans laquelle les apparitions de sous chaînes de la forme `$nom` sont remplacées par la valeur de la variable `nom`.
[g] Complétez la fonction `appliquer_expansion_variables` dans le fichier `variables.c`. Vos variables devraient maintenant fonctionner totalement, vous pouvez tester le résultat en exécutant `test_2_variables.sh` ou un script de votre composition. ■

4 Découpage de la ligne

Il s'agit maintenant de découper en morceaux la ligne de commande ce qui constituera une commande et la séquence de ses arguments. Pour cela il s'agit juste d'isoler les différents mots présents sur la ligne de commande. Cela peut se faire directement dans le tableau de caractères où est stocké la ligne expansée.
[h] Complétez la fonction `decouper_ligne` du fichier `lignes.c`. Pour tester, utilisez les scripts précédents, toutes les commandes devraient fonctionner normalement ! ■

5 Variables automatiques

Pour traiter les variables correspondant aux arguments de la ligne de commande d'un script shell, nous avons besoin de fonctions permettant de :

1. concaténer des chaînes de caractères, pour fabriquer la liste des arguments. La fonction `strcat` fait cela, consultez sa page de manuel.
2. affecter à une chaîne l'écriture décimale d'un nombre : pour pouvoir donner le nom "13" à la variable associée au treizième argument de la ligne de commande par exemple, ou pour donner la valeur "42" à la variable représentant le nombre d'arguments (s'ils sont très nombreux). La fonction `sprintf` que vous avez déjà rencontrée permet de faire cela.

Exercice complémentaire :

> Quel est le nom de la variable qui représente le nombre d'arguments d'un programme shell ? Pourquoi faudra-t-il lui donner comme valeur la chaîne "42" et non pas l'entier 42 ?

Les variables automatiques sont initialisées avec les arguments de la ligne de commande, leur nombre, et leur concaténation en une liste. Ces valeurs peuvent donc être connues grâce à `argc` et `argv`, dès le début du `main`.
Attention : pour simuler l'exécution de

```
./fichier.sh argument1 argument2 argument3 ...
```

notre interpréteur s'utilise avec la syntaxe suivante :

```
./interpreteur fichier.sh argument1 argument2 argument3 ...
```

donc dans la fonction `main` de l'interpréteur, `$0` est `argv[1]`, `$1` est `argv[2]` ...

Exercice complémentaire :

> Que vaut `$#` ? Par quel `argv[i]` commence la liste des arguments `$*` ?

Exercice complémentaire :

> Complétez la fonction `affecter_variables_automatiques` du fichier `variables.c`. Procédez par étapes : commencez par le nombre d'arguments, puis par chaque argument, et enfin terminez par la liste des arguments. Testez à chaque étape, une fois l'ensemble terminé, le fichier `test_3_automatique.sh` devrait entièrement fonctionner.