IFT2245 | Travail pratique #1

Mihai Ionescu - **1012378** André Langlais - **20044886**

February 5, 2019

1 Compréhension de la donnée

1.1 Survol & Cadeaux & Détails

Les sections **ci-haut** étaient très bien expliquées (et détaillées). Nous n'avons donc rencontré aucun problème.

1.2 CH: un shell

Nous avons dû relire quelques fois cette section, car nous ne sommes pas familier avec plusieurs de ces commandes. En particulier les points **2** et **4**.

Nous avons dû faire un peu de recherche (tutoriels et videos éducatives) pour en apprendre davantage sur le fonctionnement de certaines commandes.

2 Compréhension du code

Nous ne sommes pas des experts en **C** (loin de là, ...très loin), mais le code fourni était plutôt simple à comprendre.

Donc, rien à signaler pour cette section!

3 Complétion du code

Alors là! Nous avons pas mal de chose à dire.

Nous avons divisé cette section en plusieurs sous-sections pertinentes qui expliquent les différentes situations que nous avons rencontrées.

3.1 Exécution d'une commande

Pour exécuter une commande, nous avons choisi d'utiliser *exevp* afin de pouvoir passer un array contenant les options et paramètres tout en ayant accès au

PATH. Pour des raisons que nous ignorons toujours, il faut lui passé **le nom de la commande** comme premier argument et comme premier élément du second argument (tableau d'arguments). Il y a sûrement une raison d'implémentation mais nous n'avons pas chercher plus loin.

3.2 Énoncés conditionnels

Lors du **parsing** des commandes et des arguments, nous avons constaté un problème au niveau des énoncés conditionnels : *l'imbrication (illimité) de ceux-ci.*

Notre solution a été de traiter ces cas-là en premier. Donc en gros, nous évaluons (dans l'ordre) les expressions suivantes :

- Énoncés conditionnels.
- Les && et ||.
- Les & et >.
- Les commandes.

3.3 Procédure du découpage des expressions

Pour découper adéquatement les expressions à partir de la chaîne de caractères fournie à notre programme, nous utilisons une approche récursive. À chaque appel, nous effectuons un ou plusieurs forks (dans le cas des && ou ||) pour extraire les différentes expressions.

Nous utilisons **strtok** pour sectionner l'entrée faite par l'utilisateur en un tableau contenant chaque mots et nous procédons ensuite avec de petits algorithmes pour extraire chaque section de l'expression courante. Cette extraction consiste a créer un sous-tableaux dans lequel la sous-expressions est copié. La tableau principale ainsi qu'un pointeur vers une valeur servant à faire son itération sont envoyés à chaque fonction afin de conserver le point où l'analyse est rendue.

Nos algorithmes ont certainement des failles, mais nous avons mis en place quelques vérifications pour assurer que notre programme détecte les entrées invalides. Cela étant dit, il se peut que certains cas ne soient pas couverts (cela entraînera des comportements inconnus).

3.4 Cas limite et décisions d'implémentation

Afin d'implémenter la shell, nous avons dù prendre quelques décisions. En se fiant à la syntaxe des exemples fournis dans l'énoncé, nous avons pris la décision

de seulement gérer les cas où les mots-clés et les opérateurs sont séparés par des espaces.

Ce cas est valide

```
ls && ls
```

mais ce cas ne l'est pas.

```
ls&&ls
```

Aussi, nous avons décider de ne pas gérer le cas de l'utilisation du mot clé *if* comme paramètre dans une instruction. L'exemple suivant ne fonctionne pas:

```
echo if
```

Étant donné que les mots clés true et false sont présent dans l'énoncé, nous avons choisi de les implémenter. Ils fonctionnent comme des status de retour de processus. Nous avons aussi implémenter l'utilisation de ctrl-D pour quitter le shell grâcement.

3.5 Gestion de la mémoire

Comme il s'agit de notre premier programme en C, nous avons eu beaucoup de difficulté avec la gestion de la mémoire. Nous avons essayé de créer des tableaux dynamiques (de tailles différentes) pour sauvegarder les différentes chaînes de caractères. Par exemple, l'entrée suivante :

```
If true ; then ls -la ; done
```

. . . est découpée récursivement en plusieurs tableaux. Par exemple, nous aurions ici :

```
If true ; do ls -la ; done -> ["true", "ls -la"];
```

Puis, avec la recursion :

```
ls -la ->  ["ls", "-la"]
```

C'est bien beau tout ça, mais nous avons eu beaucoup de difficulté à libérer la mémoire récursivement. Nous avons tenté de gérer la mémoire allouée en sauvegardant l'adresse de chaque pointeur dans des tableaux que nous avons ensuite passé en paramètre récursivement.

Le plus difficile dans tout cela a été de déboguer le code ...et nous n'avons malheureusement pas réussi à intégrer correctement l'allocation dynamique de la mémoire.

Suite à quelques essais, nous avons finalement opté pour une solution simple : *allouer un tableau de taille fixe à chaque entrée faite par l'utilisateur.*

La taille du tableau est en fait le nombre d'espace contenu dans la commande. Cela équivaut donc au nombre de tokens entrés par l'utilisateur ou plus. Cette taille est utilisé pour tous les sous-tableaux. En effet, aucun des sous-tableaux ne peut dépasser la taille de l'entrée donc aucun problème de taille n'est possible. Par contre, les sous-tableaux sont souvent plus grand que nécessaire.

4 Conclusion

Notre plus grande surprise a été de constater le manque d'**IDE**s disponibles pour l'écriture de programme en **C** et la degré de difficulté relié au déboguage.

Outre cela, l'idée de ce **TP** était plutôt intéressante et au final, nous avons approfondi notre compréhension des processus.

THE END