IPFL - Projet OCAML

TAN Victor - 1A - ENSIIE

20 Janvier 2023

Table des matières

1 Introduction

Dans le cadre de l'UE intitulé **IPFL11**, un projet en langage de programmation Ocaml a été demandé. Celui-ci doit interpréter un fichier d'entrée contenant soit un message crypté pour les parties 1 et 2, de la forme *W(a) ;R ;W(b) ;R ;W(c)* et sous extention *.prog* ; ou alors une chaîne de caractères affichables de l'ASCII non étendu pour la partie 3, de la forme *abc* et sous extension *.txt* (En réalité, l'extension n'a pas d'importance ici).

Toutes les fonctions utilisées ont été codées ou alors vues en cours.

2 Composition de l'archive

Dans l'archive tar gzippée **victor_tan.tgz**, vous trouverez :

- Le fichier **rapport.pdf** ci-présent ;
- Le fichier principal **victor.ml** ;
- L'exécutable **prog** préalablement compilé avec ocamlc ;
- Le dossier **Tests** qui contient :
  - Le fichier de test **tests.ml** qui contient plusieurs tests ;
  - L'exécutable **tests** préalablement compilé avec ocaml ;
  - Le fichier de test **msg.txt** qui est utilisable pour la partie 3 ;
  - Le fichier de test **msgc.prog** qui est utilisable pour la partie 1 ou 2, et qui reprend en partie le fichier msg.txt (et qui rajoute l'utilisation de Delete, Caesar).

3 Manuel d'utilisation

À l'exécution, le comportement de l'exécutable **prog** variera selon les arguments :

Nombre d'arguments	Arg 1	Arg 2	Remarque
0 ou 1	-	-	Erreur d'usage
2	Entier 1 ou 2	fichier.prog	OK
2	Entier 3	fichier.txt	OK
2	Autre chose	Peu importe	Erreur d'usage
2	Entier entre 1 et 3	Autre chose	Erreur d'usage
3 ou plus	Peu importe	Peu importe	Erreur d'usage

TABLE 1 – Comportement du programme selon le nombre d'arguments.

4 Explication des choix

4.1 Structure de données : Le ruban

Le sujet était assez libre quant à la création du type du ruban. En lisant l'intégralité du sujet, on pouvait à peu près discerner plusieurs types de ruban possibles, avec leurs avantages et inconvénients.

4.1.1 Ruban en 3 parties

Un ruban en trois parties était possible, avec le code suivant :

```
type ruban = {
  gauche: char list;
  curseur: char;
  droite: char list;
};;
```

Les avantages de cette méthode sont :

- Elle est relativement simple à visualiser pour comprendre le ruban, mais il faut "retourner la liste de gauche" ;
- Elle rend compte de la présence d'un curseur qui ne peut 'Write' que sur un caractère sans pouvoir bouger ;
- Aller à gauche ou à droite est une action assez simple à concevoir, où il suffirait par exemple de déplacer l'élément du curseur vers le début de la liste de droite et le premier élément de la liste de gauche vers le curseur (pour un déplacement à gauche) ;
- L'écriture d'un caractère se fait sans parcourir les listes ;
- L'inversion du ruban se fait très simplement en interchangeant les listes de gauche et de droite car le curseur ne se déplacera pas.

Les désavantages de cette méthode sont :

- La représentation du ruban vide se fait avec un char vide, donc il n'y a pas de distinction entre le ruban vide et le ruban avec un char vide ;
- La lecture finale du ruban doit se faire en déplaçant le curseur tout à gauche, ce qui implique un parcours de la liste pour s'y rendre et un autre pour la lecture ;
- On ne peut pas tout simplement s'imaginer le ruban si on ne "retourne" pas la liste de gauche ;

Par exemple, pour avoir le ruban : Hello et un curseur sur le premier 'l', il faut la structure :

```
{
  gauche: ['e';'H'];
  curseur: 'l';
  droite: ['l';'o'];
};;
```

4.1.2 Ruban en 2 parties

Un ruban en deux parties était possible (comme un zipper), avec le code suivant :

```
type ruban = {
  gauche: char list;
  droite: char list; (*Le curseur est sur le 1er élément de la liste de droite*)
};;
```

Les avantages de cette méthode sont :
- Elle est relativement simple à visualiser pour comprendre le ruban, mais il faut "retourner la liste de gauche" ;
- Aller à gauche ou à droite est une action assez simple à concevoir, où il suffirait par exemple de déplacer le premier élément de la liste de gauche vers le début de la liste de droite (pour un déplacement à gauche) ;
- L'écriture d'un caractère se fait sans parcourir les listes ;
- Le ruban vide est possible avec 2 listes vides.

Les désavantages de cette méthode sont :
- Elle rend moins compte d'un curseur, qui est un peu abstrait ;
- La lecture finale du ruban doit se faire en déplaçant le curseur tout à gauche, ce qui implique un parcours de la liste pour s'y rendre et un autre pour la lecture ;
- On ne peut pas tout simplement s'imaginer le ruban si on ne "retourne" pas la liste de gauche ;

Par exemple, pour avoir le ruban : Hello et un curseur sur le premier 'l', il faut la structure :

```
{
  gauche: ['e';'H'];
  droite: ['l';'l';'o'];
};;
```

Cette méthode est celle qui a été utilisé dans le projet ocaml. Il a pour avantage d'être plus compact par rapport à la précédente implémentation.

4.1.3 Ruban en 1 partie

Un ruban en 1 partie était possible, avec le code suivant :

```
type ruban = {
  liste: char list;
  curseur: int; (* Le numéro de l'élément vers lequel le curseur pointe*)
  (*Commence donc par 1, 0 sera pour le ruban vide*)
};;
```

Les avantages de cette méthode sont :
- Elle est plus simple à visualiser pour comprendre le ruban, il suffit de lire de gauche à droite ;
- Aller à gauche ou à droite est une action très simple, où il suffit d'incrémenter ou décrémenter le curseur (et rajouter des espaces si nécessaire) ;
- Le ruban vide est possible avec 1 liste vide et le curseur à 0.
- La lecture finale est plus simple, car il n'y a qu'un seul parcours de liste pour la lire (et pas de parcours pour former une liste).

Les désavantages de cette méthode sont :
- L'écriture d'un caractère est peu optimale, car il faudra parcourir la liste ;
- La lecture finale du ruban doit se faire en déplaçant le curseur tout à gauche, ce qui implique un parcours de la liste pour s'y rendre et un autre pour la lecture ;

Par exemple, pour avoir le ruban : Hello et un curseur sur le premier 'l', il faut la structure :

```
{
  liste: ['H';'e';'l';'l';'o'];
  curseur: 3;
};;
```

Mais étant donné que la fonction ‘write‘ pour écrire un caractère est une fonction qui est souvent sollicitée, nous risquons de parcourir l'entièreté de la liste un grand nombre de fois. La complexité serait donc beaucoup trop grande si l'on a un grand message. Mais elle semble meilleure pour des petits messages.

4.2 Choix personnels

Le sujet laissait de la place à interprétation plusieurs points :

Si on décide d'aller à droite alors que le ruban a une liste de droite vide (Ou à gauche si la liste de gauche de droite est vide) ? Alors un caractère ' ' (espace) est placé au début de la liste de gauche (ou au début de la liste de droite). Ceci restera vrai même si les deux listes du ruban sont vides.

Pour la fonction de suppression, seule la première occurence est supprimée ou bien toutes les occurences ? Arbitrairement, toutes les occurences seront supprimées.

Que se passe-t-il si on inverse un ruban où la liste de gauche est non-vide et la liste de droite vide ? Arbitrairement, il aurait été possible de simplement inverser le ruban de manière classique et de mettre un caractère ' ' (espace) au début de la liste de droite. Cependant, il est peu probable qu'un programme contienne un Invert en se positionnant intentionnellement tout à droite. Donc un simple message d'erreur sera affiché : *failwith "Impossible d'utiliser la fonction inverser avec une liste de droite vide"*.

4.3 Remarque

Une remarque peut être fait concernant la fonction **generate_program**, donc la fonction de la dernière partie du sujet. Dans cette implémentation, peu importe le fichier d'entrée, toute lecture d'un fichier non-vide et correct renvoie un 'R ;' à la fin, c'est-à-dire que le ruban final avant affichage possède une liste de droite vide. Une alternative est d'avoir une liste de droite avec le dernier caractère.

Pour avoir cette implémentation, on aurait pu faire un match avec un caractère simple 'cara' (après le match avec lettre : :restre à la ligne 357) et simplement écrire 'Write cara'. Mais pour plus de clarté dans le code, le match n'a pas été implémenté.

5 Tests personnels

Cette partie est destinées à l'ensemble des tests effectués avec les fichiers de test présents dans l'archive. Elle n'est pas nécessaire à la compréhension du projet.

5.1 Avec le fichier de test tests.ml

Des fonctions 'assert' ont été utilisé pour donner le ruban que je pense être bon. Les images suivantes permettent de voir l'ensemble de ces tests et des résultats que j'attendais.

5.2 Avec le fichier msgc.prog

Ce test utilise les instructions associées W, R, F, C et D. Le message est :

```
# Utilisation de W, R, F, C et D.
W(Z) ;R ;W(N) ;R ;W(l) ;... ;C(-2) ;
```

Et le résultat attendu est :

```
OCaml, anciennement ... l'Inria.
```

5.3 Avec le fichier msg.txt

Ce fichier est utilisé pour faire le test de la partie 3. Le message est :

```
OCaml, anciennement ... l'Inria.
```

Et le résultat attendu est :

```
W(O) ;R ;W(C) ;R ;... ;R ;W(.) ;R ;
```

Comme indiqué dans la partie sur la remarque concernant le 'R ;' à la fin, celui-ci aurait pu être retiré en modifiant le code.

6 Limites du programme

Pour la fonction **generate_program**, il est possible d'optimiser la fonction :

- Dans notre programme, la fonction détecte si une même lettre est répétée plusieurs fois de suite pour les concaténer avec 'Repeat'.

- Mais il est possible de chercher si un motif de plusieurs lettres est répété. Par exemple, on peut demander à l'utilisateur d'ajouter un argument 'm' au programme, où 'm' correspond à la profondeur de la recherche du motif. Cela peut se faire en créant regardant si 'm' lettres après la lettre courante, on est sur la même lettre ; et si oui, on regarde si pour les 'm - 1' prochaines lettres, on conserve l'égalité. La complexité reste donc en O(n).

- Une autre possibilité serait de ne pas avoir de 'm' fixé, et on pose 'm' la distance entre la lettre courante et la prochaine lettre qui est identique à la lettre courante. La complexité serait en $O(n^2)$.