Projet IPFL

Noah Kwa Moutome

April 2023

Table des matières

1 Introduction

Ce projet consistait en la réalisation d'un programme, permettant de décoder et d'afficher des messages envoyés par des extraterrestes. Ces messages devaient être retranscrits sur un ruban infini, sur lequel nous pouvions déplacer un curseur pour écrire différents caractères. Le cryptage était effectué par différentes instructions que voici :

— déplacer le curseur vers la droite ou la gauche
— écrire un caractère "c" à l'emplacement du curseur
— repéter n fois une liste d'instuctions
— appliquer un encodage de César de pas n au message déjà décrypté
— effacer un caratere "c" dans le message déjà decrypté
— inverser le contenu actuel du ruban et replacer le curseur

Nous verrons dans les parties suivantes le choix des structures et des implémentations utiles à la réalisation de ce projet.

2 Les 2Aiens

Dans cette partie, nous allons voir l'implémentation des types et fonctions nécessaires à la réalisation de ce début de projet. Seuls les instructions de déplacement du curseur, d'écriture et de répétition de liste d'instructions sont traités dans cette partie.

2.1 Le ruban

Le ruban, supposé infini, devait représenté la structure sur laquelle nous devions écrire le message décodé. Plusieurs interprétations étaient possibles vis à vis du mot "infini". J'ai décidé de l'interpréter comme étant une structure de taille variable, non prédéfinie et potentiellement très élevée. Le caractère infini de ce ruban supposait l'utilisation de listes. Une première idée possible aurait été d'utiliser une structure, composée d'une liste et d'un entier correspondant à la place de notre curseur dans cette liste. Cependant, cette implémentation nécessitait la gestion de cet entier, ce qui ne me semblait pas être chose aisée. J'ai donc opté pour une structure à mon sens plus adaptée : un *zipper*. Ce *zipper*, composé d'une liste *left* (dont les éléments, correspondant au début du ruban, sont inversés : la list left est donc *l'inverse* du début du ruban) et d'une autre *right*, permettait de naviguer vers la gauche ou la droite du ruban assez librement. Dans le cas où l'une des deux listes était vide, il suffisait simplement d'écrire une caractère vide en cas de déplacement vers la gauche (ou la droite). Le curseur se situait alors sur la tête de la liste *right*, en cas d'écriture on écrasait donc cette tête.

2.2 La fonction execute_program

Cette fonction est la fonction principale, permettant de lire le programme passé en argument de cette dernière (et présent dans le fichier à décoder après quelques transformations préalables de fonctions annexes), et ensuite de renvoyer le message décodé. Pour ce faire, nous utilisons une fonction auxiliare récursive composée d'un simple *match* sur la structure du programme, pour déterminer l'action de celui-ci, puis nous mettons à jour le ruban en conséquence avant de rappeler cette même fonction auxiliaire sur la suite du programme et sur le ruban post-modification. Le seul petit problème ici était la gestion de l'instruction *Repeat*. En effet, cette instruction est composée de 2 arguments : un entier n et une liste d'instruction l. Je devais donc distinguer 2 cas : si n était nul ou non. Dans le cas $n = 0$, je ne modifiais pas le ruban. Dans le cas où $n>0$, j'appelais ma fonction récursive sur le ruban auquel j'avais appliqué une première fois la liste d'instruction l, avec la suite du programme auquel j'ai ajouté un *Repeat(n - 1, l)* en tête de ce dernier.

2.3 La fonction fold_ruban

Ici, l'idée était d'appliquer un *fold* de gauche à droite à mon ruban. Le seul problème résidait en le type ruban défini, qui était pour ma part un *zipper* composé de 2 *listes*. La liste *left* étant l'inverse de la première partie réelle du ruban (le dernier élément de *left* correspond au premier élément du ruban) il fallait faire attention. L'idée était donc de simplement appliquer un fold_right à la liste *left* et de stocker ce résultat dans l'accumulateur appelé dans le fold_left appliqué à *right*. En faisant ainsi, on appliqué un bien un *fold* de gauche à droite à l'ensemble du ruban.

3 Les 3Aiens

Nous allons ici implémenter les instructions d'effacement, d'encodage de Cesar et d'inversion.

3.1 Effacement

Pour cette partie j'ai simplement coder une fonction récursive *suppr_char_list* qui prend en entrée une liste et un caractère à supprimer et qui renvoie la liste privée de toutes les occurences du caractère passé en argument. Une fois cela fait, je rend simplement le filtrage sur le *program* passé en argument de la fonction *execute_program* plus exhaustif en y ajoutant l'instruction *Delete(a)*. Si on match sur ce motif, on fait alors 2 fois appel à *suppr_char_list* sur la partie gauche et droite de ruban, avant de refaire une appel récursif sur la suite du *program*.

3.2 Inversion

Ici, pas de fonction annexe nécessaire. On se contente de rendre le filtrage de la fonction *execute_program* plus exhaustif, en matchant le *program* sur *Invert*. Dans ce cas là, on échange les listes *right* et *left* du ruban, car au vu de l'implémentation du ruban, on sait que *left* correspond à l'inverse du début du message. Il reste simplement à replacer le curseur, pour cela on effectue un *go_left* sur le ruban. En faisant cela, on retourne bien sur l'ancien caractère courant, car celui-ci se trouvait en tête du nouveau *left*. On a donc inversé le ruban et replacé le curseur.

3.3 Encodage de César

Pour coder cette instruction, j'ai décider d'utiliser un *fold_right* sur chacune des deux listes du ruban, via la fonction nommée *caesar_list*. Ce *fold* est utilisé avec une fonction f décrite ci-dessous :

— prend en argument un accumulateur, un caractère et un entier n

— on vérifie le signe de n, dans le cas où n est négatif on effectuera simplement une addition de 26 pour ramener le code de notre caractères crypté dans les positifs
— on regarde si le caractère c est une majuscule, une minuscule ou autre chose. Si c'est autre chose on n'y touche pas.
— on "ramène le problème en 0" en fonction de la nature de c en soustrayant à l'entier codant c l'entier codant A ou a en fonction des cas. Ainsi, c est représenté par un entier compris entre 0 et 25.
— on additionne n et on effectue un modulo 26 pour retourner sur un entier compris entre 0 et 25
— puis on redécale en ajoutant l'entier codant A ou a selon les cas.

Une fois cela fait, on rend simplement le filtrage de *execute_program* plus exhaustif en y ajoutant *Caesar(n)* et en appelant *caesar_list n* sur les deux listes du ruban en cas de match sur ce motif.

4 Les diplômiiens

Nous devions ici faire le processus inverse, i.e crypter un message dans le language utilisé par les 3Aiens. Pour ce faire, j'ai implémenté la fonction *generate_program* qui sur l'entrée d'une chaine de caractère renvoie un *program* correspond à cette dernière. Le message correspond à une *char list*, on fait donc simplement un match sur cette liste et on ajoute chaque *instruction* correspondante. Évidemment, pas d'instructions de la partie 2 ici, seulement des *Write*, *Right* et des *Repeat*. La seule difficulté ici est donc de gérer les Repeat. Cependant, j'ai trouvé très difficile de gérer les motifs répétés (tel que "mama" par exemple). Je me suis donc simplement contenté de faire une fonction auxiliaire dans *generate_program* qui permet de gérer les répétitions de lettres (tel que "lll"). Le principe est assez simple : utiliser un compteur qu'on incrémente tant que 2 lettres identiques se suivent. Bien sûr, cette fonction est récursive. Une fois cela fait j'utilise une autre fonction auxilaire qui permet de lire le message en faisant appel à la première fonction auxiliaire.

5 Manuel d'utilisation

Pour utiliser le programme il faut d'abord le compiler avec *ocamlc projet.ml -o projet*. Puis il y a 3 modes d'utilisation, correspondant à chacune des trois parties. Il suffit d'entrer la commande *./projet i fichier.prog* avec $i \in \{1, 2\}$ pour les parties 1 et 2, et *./projet 3 fichier.txt* avec le message a crypté dans *fichier.txt*.