NOISETTE

Hugo GENEST

21 avril 2023

Table des matières

Introduction

Grâce à l'Enormous National Spatial Intelligent Interactive E, l'Institut National Purement Futile a pu intercepté des communications extraterrestres venant des 2Aiens et des 3Aiens, de la galaxie des estudiens. Nous avons dû, avec l'aide du Facilitateur Organisateur Réprimandeur Estimateur Surveillant Télépathe, concevoir une machine permettant de décoder ces messages. Mais afin de prouver ma valeur au grand FOREST, et ainsi m'élever au rang de diplômieen, il m'eût fallu construire un système permettant de traduire du terrien en 2Aiens et 3Aiens. C'est dans cet objectif que j'ai conçu le sujet de ce rapport : le **Nouvel Outil Informatique Super Efficace de Traduction Terrien Estudien.**

1 Présentation du Produit

1.1 Principes de base

Dans ce projet, nous avons été amené à manipuler un **ruban** à l'aide **d'instructions**. Un ruban est objet de longueur infinie sur lequel peuvent être inscrits des caractères, et que nous pouvons manipuler à l'aide d'instructions. Ici, nous représentons le ruban en ocaml par le type ruban, correspondant au type usuel du `zipper` :

```
type ruban = {left: char list; right: char list};;
```

Nous considérons, comme pour les `zippers`, un curseur situé en tête de la liste `right`, et à gauche de ce curseur, nous imaginons la liste `left`, avec sa tête situé juste avant le curseur :

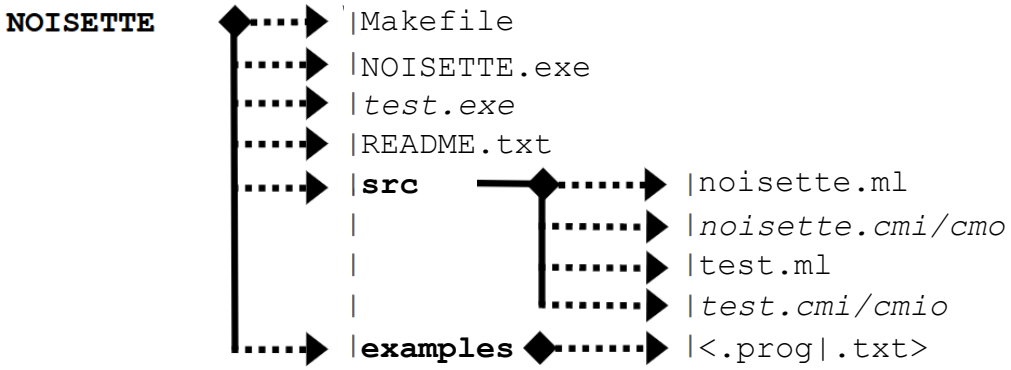
Les instructions sont des commandes qui servent à modifier le ruban, et qui seront rassemblées pour former un programme dans un fichier `.prog`. Il existe 7 instructions différentes (avec leur représentation dans les fichier `.prog`) : Left et Right (`L` et `R`) pour déplacer le curseur, Write (`W(c)`) pour écrire le caractère `c` au niveau du curseur, Repeat (`F(n,i)`) pour répéter `n` fois la liste d'instructions `i`, puis Delete (`D(c)`) pour supprimer toutes les instances du caractère `c` dans le ruban déjà formé, Invert (`I`) pour inverser celui-ci, et enfin Caesar (`C(n)`) pour appliquer une clef de César n au ruban déjà écrit (c'est-à-dire déplacer les lettres de n places dans l'alphabet).

Un programme a pour but de modifier un ruban vierge pour pouvoir finalement y écrire un message. Par exemple, le programme suivant permet d'écrire `CCB` :

```
[F(3,[W(A);R;]);C(2);L;W(B)]
```

Notre objectif est dans un premier temps d'écrire un algorithme capable d'appliquer un tel programme à notre ruban, puis dans un second temps, de faire l'inverse, c'est-à-dire de concevoir une machine capable, depuis un texte en entrée sous forme d'une liste de caractères, d'écrire un programme capable de l'écrire, avec le minimum d'instructions possible.

1.2 Présentation des Documents



L'application est composé d'un dossier principal `NOISETTE`. Dans celui-ci nous retrouvons un `README.txt`, l'exécutable `NOISETTE.exe` ainsi qu'un `Makefile`. Nous y trouvons également le dossier `src` contenant les fichiers `noisette.ml` et `test.ml`. C'est ici que seront générés les fichiers en `.cmi` et `.cmo` . Enfin, dans `NOISETTE`, il y a un dossier `examples` dans lequel se trouvent de nombreux fichiers `.prog` avec les fichiers `.txt` correspondants. Les fichiers `test.prog`, `test.txt`, `txt2prog.prog` et `prog2txt.txt` sont aussi stockés ici.

Le `Makefile` possède plusieurs fonctionnalité. Il permet de compiler les fichiers `noisette.ml` en `NOISETTE.exe` et `test.ml` en `test.exe` qui apparaîtra alors dans le dossier principal. Celui-ci peut être exécuté dès sa compilation avec la commande `make test`. Le `Makefile` permet également de supprimer les fichiers `.cmi` et `.cmo`. Enfin, avec la commande `make try`, `NOISETTE.exe` sera exécuté sur les fichiers `test.txt` et `test.prog`, et les résultats seront respectivement inscrits dans `txt2prog.prog` et `prog2txt.txt`.

1.3 Présentation des exécutables

L'utilisateur a donc accès à l'exécutable `NOISETTE.exe`, qui prend en arguments une valeur de phase entre 1 et 3, puis le nom d'un fichier à lire. Si la phase est 1 ou 2, alors le fichier d'entrée devra être un fichier `.prog`, ce dernier sera lu et le message résultat sera écrit sur la sortie standard. Attention, en phase 1, l'exécutable ne sera pas capable de gérer les instructions `delete`, `invert` et `caesar`, la phase 2 est nécessaire pour ces dernières.

Si la phase indiquée est la phase 3, alors le fichier d'entrée devra être un fichier en `.txt`, et l'exécutable écrira sur la sortie standard un programme capable de le rédiger. Ce programme doit contenir un faible nombre d'instructions.

L'exécutable `test.exe` permet de lancer une série de test afin de vérifier le fonctionnement de `noisette.ml` pour les deux premières phases. Malheureusement, il est complexe de connaître le nombre minimum d'instructions possibles pour écrire un texte, ce qui rend difficile la correction de la troisième phase. Pour pouvoir vérifier l'efficacité de notre programme, nous devons donc effectuer des essais à l'aide des fichiers test de `examples`, et en utilisant la commande `make try`.

Les fonctionnements du `Makefile` et des différents exécutables sont décrits dans le README.

2. Les 2Aiens et 3Aiens

Cette première moitié était la plus simple. C'est dans celle-ci qu'il nous aait été demandé d'implémenter le type `ruban`, les fonctions `execute_program` et `fold_ruban`, avec les instructions de la première phase : `Right`, `Left`, `Write` et `Repeat`. Puis nous avons ensuite dû incorporer les instructions de la deuxième phase : `Delete`, `Invert` et `Caesar`.

2.1 Implémentation des Instructions

Les instructions avaient déjà été conçues par le FOREST, en tant que type :

```
type instruction =
  | Left
  | Right
  | Write of char
  | Repeat of (int * instruction list)
  | Caesar of int
  | Delete of char
  | Invert
```

De même, le type `program` était déjà présent, en tant que liste d'instructions.

```
type program = instruction list
```

Il nous suffisait alors seulement d'implémenter les fonctionnalités des différentes instructions. Les fonctions que j'ai écrite sont très simples, voici quelques informations sur lesquelles il a fallu faire attention.

Pour les fonctions de déplacement du curseur, `go_right` et `go_left`, le ruban étant de taille infini, il a fallu décider du comportement à adopter lorsqu'une extrémité était atteinte. J'ai alors

décider d'ajouter un caractère par défaut dans le sens de déplacement. J'ai d'abord utilisé arbitrairement le caractère `'0'`, puis j'ai pris connaissance du caractère nul `'\000'`.

```
let go_left ruban =
  match ruban with
  | {left= [] ; right=rlist} -> {left=[]; right= '\000'::rlist};
  | {left=hd::tl; right=rlist} -> {left=tl; right=hd::rlist};;
```

J'ai utilisé dans la fonction `caesar` les fonctions Char.code et Char.chr, qui permettent respectivement d'obtenir le code ASCII d'un caractère et le caractère correspondant à un code ASCII, afin de manipuler les lettres de mon ruban avec l'entier n en argument.

```
let caesar_shift caesar_key c =
  let code = Char.code c in
  if code > 64 && code < 91 then
    Char.chr ( 65 + ( ( code - 65 + caesar_key ) mod 26 ) )
  else if code > 96 && code < 123 then
    Char.chr ( 97 + ( ( code - 97 + caesar_key ) mod 26 ) )
  else
    c;;
```

Les différentes valeurs que nous pouvons voir ici correspondent aux codes ASCII des extrémités de l'alphabet ('A' = 65, 'Z' = 90, 'a' = 97, 'z' = 122).

Pour la fonction `delete`, j'ai fait usage de la fonction `List.filter`, qui prend en argument une fonction `f` et une liste, et renvoie la liste, mais uniquement avec les éléments `x` tels que `f x = true`. Il suffisait donc d'applique ce filtre aux deux listes de notre ruban en prenant comme fonction une comparaison avec le caractère à supprimer.

```
let delete c ruban =
  let aux elt = (elt != c) in
  {left=List.filter aux ruban.left; right=List.filter aux ruban.right};;
```

La fonction `invert` correspond à une simple inversion des deux éléments du ruban, suivie d'un déplacement vers la gauche pour replacer le curseur au bon endroit.

```
let invert ruban =
  go_left {left = ruban.right ; right = ruban.left};;
```

Enfin, l'instruction `repeat` n'a pas eu le plaisir de se voir attribuer une fonction, car j'ai décidé d'incorporer sa fonctionnalité directement dans la fonction `execute_on_ruban`, se fondant bien dans le `match` de cette dernière.

2.2 `execute_program` puis `fold_ruban`

Maintenant que les différentes instructions des 2Aiens et 3Aiens ont été implémentées, il suffit de les rassembler pour créer la fonction `execute_program`. En effet, nous pouvons initialiser un ruban vide que nous modifions en parcourant l'ensemble des instructions du program

donné en argument. Ce parcours de liste se fait avec de la récurrence, qui devra également retenir l'état du ruban afin de ne pas perdre notre avancé. C'est pourquoi je suis passé par l'intermédiaire d'une fonction auxiliaire récurrente `execute_on_ruban`, prenant en argument un programme et un ruban, afin de ne pas modifier la forme de la fonction `execute_program`.

```
let rec execute_on_ruban p ruban  =
  match p with
  | [] -> ruban;
  | Left::tl -> (*recursive call*) ;
  | Right::tl -> (*recursive call*) ;
  | (Delete a)::tl -> (*recursive call*) ;
  | (Caesar n)::tl -> (*recursive call*) ;
  | Invert::tl -> (*recursive call*) ;;
```

Cette fonction identifie la première instruction du program `p` grâce à un `match`, puis l'applique au ruban. Nous rappelons ensuite la fonction sur le reste de `p` et sur notre ruban modifié. La fonction s'arrêtera donc une fois `p` achevé, et nous retournerons l'état final du ruban.

L'instruction `Repeat` s'inclut naturellement dans ce `match`. Il suffit de regarder le nombre de boucles demandées `n` par une instruction `Repeat`. Si `n` est strictement positif, alors nous exécutons les instructions du `Repeat` sur le ruban, puis nous refaisons un appelle récursif sur notre nouveau ruban avec le même programme mais en considérant un nouveau `n` valant `n-1`. Une fois que `n` vaut 0, nous pouvons continuer sur la suite du programme.

```
match p with
...
| (Repeat (n,instructions))::tl ->
  if n=0 then
    execute_on_ruban tl ruban
  else
    let new_ruban = execute_on_ruban instructions ruban in
    execute_on_ruban ((Repeat (n-1,instructions))::tl) new_ruban ;
```

Pour la fonction `fold_ruban`, en considérant un ruban :

$$\texttt{r = \{left=[}a_1,\dots,a_n\texttt{];right=[}b_1,\dots,b_n\texttt{]}\},$$

nous souhaitons appliquer la fonction `f` passée en argument du `fold` de même manière qu'on l'utiliserait avec un `fold_left` sur la liste $[a_n,\dots,a_1,b_1,\dots,b_n]$, c'est-à-dire, en notant `v0` l'argument extérieur fourni :

$$f(\dots f(f(\dots f(f(v0,a_n),a_1)\dots,b_1),b_2)\dots,b_n)$$

Nous devons donc appliquer un `fold_right` à la liste de gauche, en prenant garde à inverser les arguments de la fonction `f`, puis un `fold_left` à la liste de droite.

```
let fold_ruban f v0 r =
  List.fold_left f (List.fold_right (fun x y -> f y x) r.left v0) r.right;;
```

3. Diplômiien

Cette partie était la plus complexe. Nous devions concevoir un outil capable de traduire un message `txt` en programme `prog` contenant un nombre minimum d'instructions. Je suis content du travail que j'ai réalisé, mais il est à noter que mon algorithme ne donne pas toujours le nombre minimum d'instructions.

3.1 Première Implémentation

La version finale du programme est la seconde implémentation réalisée. J'avais réalisé une première version qui écrivait un programme correctement, mais qui n'était pas satisfaisante. L'algorithme était le suivant :

Considérons un message : `msg = ababc`

I. Nous réécrivons le message en programme en remplaçant les caractère par des `writes` et `right`;

`[W(a);R;W(b);R;W(a);R;W(b);R;W(a);R;W(b);R]`

II. Nous découpons le programme obtenu en groupe de même taille, en commençant par une taille 1 (par soucis de visibilité, considérons qu'il n'y a pas de `rights`, mais attention ces derniers peuvent avoir leur importance);

`[W(a)] ; [W(b)] ; [W(a)] ; [W(b)] ; [W(a)] ; [W(b)]`

III. Nous parcourons ces groupes de droite à gauche, en regardant si ils sont identiques à leur voisin de gauche (nous comparons également avec les instructions incluses dans des `Repeats`). Si une similitude est repérée entre un groupe et son voisin, nous les fusionnons dans un `Repeat`.

`Aucunes similitudes entre voisins dans notre exemple`

IV. Nous réitérons les deux dernières étapes, avec des groupes de même taille si le programme a été modifié à l'étape III, en augmentant la taille de 1 sinon.

```
II.  [W(a);W(b)] ; [W(a);W(b)] ; [W(a);W(b)]
III. [W(a);W(b)] ; [ Repeat (2, [W(a);W(b)])]
                   [ Repeat (3,[W(a);W(b)])]
```

V. L'algorithme s'arrête en fin de l'étape IV, quand la taille de groupe dépasse la moitié de celle du programme. En effet, à cet instant, nous n'aurions que 2 groupes sans similitudes.

Dans notre exemple, le résultat serait donc (en reconsidérant les `rights`) :

`[ Repeat (3,[W(a);R;W(b);R])]`

Ce résultat est celui que nous attendions.

Cependant, cet algorithme est rendu trop faible à cause du découpage en différents groupes. En effet des découpages pourraient ne pas se faire au bonne endroit. Regardons un exemple de ce défaut (oublions à nouveau les `rights` durant l'algorithme):

`msg = ababc`

`II. (group_size = 2)  [W(a)] ; [W(b);W(a)] ; [W(b);W(c)]`

(Le groupe de gauche n'a pas la bonne taille car celle -ci ne divise pas celle du programme)

Ici, ce découpage empêche la reconnaissance du paterne « ab », « ab ». En poursuivant l'algorithme, nous obtenons finalement :

`[W(a);R;W(b);R;W(a);R;W(b);R;W(c);R]`

Nous trouvons un nombre de 10 instructions, ce n'est pas satisfaisant du tout. Nous aurions voulu en obtenir un nombre de 6 avec :

`[Repeat (2,[W(a);R;W(b);R]);W(c)]`

C'est pourquoi j'ai voulu modifier mon algorithme pour finalement proposer la version décrite en dessous.

3.2 Version Finale

Cette deuxième version est très proche de la première, à ceci près qu'elle ne fait plus de prédécoupage du programme. À chaque tour de boucle, nous récupérons les deux premiers groupes de notre programme, que nous comparons. Si une similitude est repérée, nous les fusionnons, sinon nous décalons nos sélections de 1 place. Le travail se fait ici de la gauche du programme vers la droite, considérons donc le dernier `msg` inversé : `msg = cbaba`

La première étape reste la même : `[W(c);R;W(b);R;W(a);R;W(b);R;W(a);R]`

Lors des groupes de taille 1 à 3, nous n'allons pas remarquer de similitudes. Mais regardons le tour de boucle pour les groupes de taille 4 :

`partie analysée : []`

`parties en comparaison :[] ; []`

`reste du programme : [W(c);R;W(b);R;W(a);R;W(b);R;W(a);R]`

On extrait les deux premiers groupes du programme pour commencer les comparaisons

—

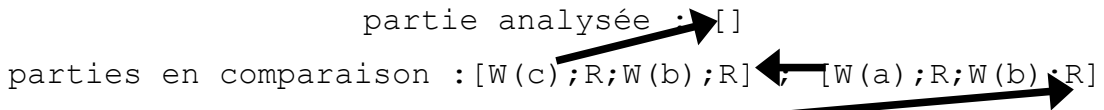
`partie analysée : []`

`parties en comparaison :[W(c);R;W(b);R] ; [W(a);R;W(b);R]`

`reste du programme : [W(a);R]`

—

Pas de similitudes observées pour l'instant, on déplace les sélections.



—

partie analysée : [W(c)]
parties en comparaison :[R;W(b);R;W(a)] ; [R;W(b);R;W(a)]
reste du programme : [R]

Une similitude ! Les deux parties comparées sont donc fusionnées.

—

partie analysée : [W(c)]
parties en comparaison :[Repeat (2,[R;W(b);R ;W(a)])]; [R]
reste du programme : []

—

La recherche des groupes de taille 4 s'arrête ici puisque les deux parties comparées sont différentes et qu'il ne reste plus de programme à analyser.
Après analyse des groupes de taille 4, nous obtenons donc le programme :

[W(c);Repeat (2,[R;W(b);R ;W(a)]);R]

De plus, ce programme est de taille 3, et la taille de groupe actuelle étant de 4 (> 3/2), le programme s'achève. Il s'agit donc du résultat final. Cela correspond au résultat recherché, avec néanmoins un `right` final de trop. Nous pourrions supprimer cet élément en trop, seulement cela impliquerai de devoir accéder au dernier élément d'une liste, ce qui devient lourd pour de trop grand message, et ça n'économiserai qu'une seule instruction. J'ai donc considéré que cela n'était pas pertinent.

Pour mettre en place cet algorithme, j'ai conçu plusieurs fonctions ; `match_pattern` qui va faire le parcours d'une liste à la recherche de paternes d'une taille particulière, `simplify` qui va s'occuper de lancer `match_pattern` pour les différentes tailles de groupe possibles, et enfin `extract_pattern`.

```
let rec extract_pattern_aux program current_pattern nb_elements =
  if nb_elements = 0 then
    (List.rev current_pattern,program)
  else
    match program with
    | [] -> (List.rev current_pattern,program);
    | hd::tl -> extract_pattern_aux tl (hd::current_pattern) (nb_elements-1);;
let extract_pattern program nb_elements =
  extract_pattern_aux program [] nb_elements;;
```

La fonction `extract_pattern` prend en argument un programme `program` et un entier `nb_elements` et permet d'extraire les `nb_element` premières instructions du programme. La fonction retourne un couple avec ces premiers éléments puis le reste du programme.

Enfin, j'ai pu remarquer au cours de mes tests que la présence des `rights` pouvait engendrer des défauts de détection de paterne. C'est pourquoi je ne les écrit plus lors de la première étape. Cela devient une ultime tâche, durant laquelle je rajoute un `right` après chaque `write`.

3.3 Limites du Programme

Nous avons donc un outil capable de traduire des fichiers `.prog` en texte, et des fichiers `.txt` en programme, et cela avec un faible nombre d'instructions. Cependant, mon implémentation ne permet pas de trouver le nombre minimum d'instructions (en plus du `right` de trop).

En effet, nous pouvons penser à certains cas où notre système faillit, comme l'exemple suivant : `msg = lllulu`.

Nous avons comme résultat pour ce message :

`F(3,[W(l);R;]);W(o);R;W(u);R;W(l);R;W(o);R;W(u);R;` *(13 instructions)*

Nous aurions voulu trouver :

`F(2,[W(l);R;]) ; F(2,[W(l);R;W(o);R;W(u);R;])` *(10 instructions)*

De plus, dans notre optimisation nous n'utilisons que les instructions `right`, `write`, et `repeat`. Or les autres instructions peuvent améliorer certaines solutions. Voici quelques exemples :

Message	Ma Solution	Autre Solution	Instruction Utilisée
`-----!-----`	`F(5,[W(-);R;]);W(!);R;F(5,[W(-);R;]` 8 instructions	`F(11,[W(-);R]);F(6,[L;]);W(!)` 6 instructions	`Left`
`abcde`	`W(a);R;W(b);R;W(c);R;W(d);R;W(e);R;` 10 instructions	`F(5,[W(f);R;C(25)]);` 4 instructions	`Caesar`

Conclusion

Finalement le programme, malgré qu'il ne soit pas optimal, reste efficace, et je suis satisfait de mon travail. Ce projet aura été intéressant car il m'aura permis de travailler à nouveau mon organisation, ma méthode de développer (effectuer des tests, commenter, couper son code en différentes parties,etc...). Mais il m'aura surtout permis de développer mes capacités en Ocaml.
Ce langage est nouveau pour moi, et très différent des autres langages de programmation que j'ai pu manipuler. Cela m'a poussé à réfléchir et à repenser ma manière de programmer.