École Nationale Supérieure
d'Informatique pour l'Industrie et
l'Entreprise

IPFL
Projet 2023
Rapport

Décodeur de messages extraterrestres de nos chers iiens

Élèves :
Colin Coërchon

Enseignants :
Julien Forest

27 janvier 2023

Table des matières

1 Introduction

***Scoop* ! Des extraterrestres venues de la planète `Second A` sont rentrés en communication avec nous !**

Je suis nouveau à l'`Institut National Purement Futile`, mais je compte bien donner tout mon possible pour arriver à décoder leur langage farfelu !

Ainsi, mon objectif va être dans un premier temps de décoder leurs messages envoyés à l'aide du protocole de communication qu'ils nous ont fourni, puis l'objectif sera d'encoder un message pour leur renvoyer !

2 Les 2Aiens

2.1 Définition du type ruban et premières fonctions

Dans le protocole de communication des 2Aiens, il était explicitement dit :

"Afin de décoder cette communication, il est du ressort de la terre de fournir un ruban infini et de mettre au point une machine permettant d'exécuter les instructions en fonction de leur sémantique afin d'écrire le message sur le ruban."

Pour coder ce *ruban infini*, j'ai alors choisi le type *zipper* vu en cours dans l'`Institut National Purement Futile`. Le type *zipper* est simplement un type constitué de 2 listes : une **liste gauche** et une **liste droite**.

Nous allons devoir utiliser ce ruban pour exécuter les instructions des messages codés des 2Aiens une après les autres. Il est alors important de noter qu'il va être nécessaire d'avoir une sorte de curseur pour ces deux listes pour savoir où écrire un caractère quand l'instruction est `W(a)` (c'est-à-dire "écrire `a` à l'emplacement du curseur").

Par choix arbitraire, on considérera que **le curseur se situe en tête de la liste droite du ruban**.

On a donc un type ruban ainsi défini qui se présente sous cette forme :

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

On peut alors naviguer librement de gauche à droite en déplaçant seulement le curseur et en ajoutant des éléments en tête de liste si nécessaire. Nous avons donc notre **ruban infini qui est opérationnel**.

Maintenant, il va falloir implémenter les fonctions qui réaliseront les instructions sur notre ruban :

- `go_left r` : déplacement de la tête du ruban `r` d'un caractère vers la gauche (pour implémenter l'instruction `L`).
- `go_right r` : déplacement de la tête du ruban `r` d'un caractère vers la droite (pour implémenter l'instruction `R`).

- `write c r` : écriture du caractère `c` sur le ruban `r` sans déplacer la tête de lecture (le curseur). A noter que plusieurs écriture sans déplacement la tête auront pour effet de changer le caractère courant. (Pour implémenter l'instruction `W(c)`).
- `repeat f_execute n l r` : répéter `n` fois la liste d'instructions `l` sur le ruban `r`. (Pour implémenter l'instruction `F(n,l)`).

Commençons alors par les fonctions `go_left` et `go_right` :

```
1  let go_left r =
2      match r.left with
3      | [] -> {left = []; right = '\000'::r.right}
4      | t::q -> {left = q; right = t::r.right};;
5
6  let go_right r =
7      match r.right with
8      | [] -> {left = '\000'::r.left; right = []}
9      | t::q -> {left = t::r.left; right = q};;
```

Les deux fonctions ne sont pas bien complexes, et très similaires. Expliquons alors pour `go_left` (puisque le raisonnement est symétrique pour `go_right`) :

Pour déplacer la tête de lecture vers la gauche, il faut donc vérifier que la liste gauche de notre ruban n'est pas vide [ligne 3].

- Si elle n'est pas vide, on prend le premier élément de la liste de gauche et on l'insère à la tête de la liste de droite [ligne 4].
- Si elle est vide, on ne peut donc pas retirer le premier élément de la liste de gauche. J'ai donc décider de placer un *caractère nul* (\000' en OCaml) en tête de la liste de droite pour tout de même faire un sorte que la tête de lecture se déplace vers la gauche.

  Une autre idée aurait été de renvoyer une erreur avec un `failwith "impossible de se déplacer à droite / à gauche"`, mais je trouvais ça plus « général » de renvoyer aucune erreur sans pour autant changer le résultat final.

En ce sens, on aura alors *déplacé* la tête de lecture à gauche (ou à droite si `go_right` est appelé).

Il est aussi nécessaire d'écrire la fonction `write` pour écrire un caractère à l'emplacement exact de la tête de lecture :

```
1  let write c r =
2      match r.right with
3      | [] -> {left = r.left; right = [c]}
4      | t::q -> {left = r.left; right = c::q};;
```

Si la liste droite est vide, comme le curseur se situe **toujours** en tête de la liste droite, on "force" simplement l'insertion du caractère `c` en tête de lecture en écrivant `r.right = [c]`.

Si elle n'est pas vide, on écrit par dessus le caractère situé en tête de lecture, on *l'écrase* en quelque sorte.

2.2 La fonction `repeat`

Il est maintenant tant d'écrire la fonction `Repeat` pour clôturer le processus de décodage des messages 2Aiens.

Cependant, cette fonction est plus complexe à écrire que les 3 précédentes car elle a, en quelque sorte, besoin d'une part de la fonction `execute_program` pour fonctionner. En effet, l'instruction `F(n,l)` doit répéter `n` fois la liste d'instructions `l` sur notre ruban. Et cette liste `l` contient alors d'autres instructions comme `Left`, `Right`, `Write` ou même d'autres `Repeat` !

Bref, nous allons donc supposer qu'il existe une fonction `execute_instruction r i` qui execute l'instruction `i` sur le ruban `r`. Et alors, dans ce cas, on peut donc écrire la fonction `repeat` :

```
1  let rec repeat f_execute n l r =
2    let liste_execution l r =
3      let f_aux r instru = f_execute r instru in
4      List.fold_left f_aux r l
5    in
6    if n>0 then
7      let r = liste_execution l r in
8      repeat f_execute (n-1) l r
9    else r ;;
```

La fonction `repeat` prend 4 arguments :

- `f_execute` qui joue le rôle d'une fonction de type `ruban -> instruction -> ruban` qui sera en pratique la fonction `execute_instruction`.
- `n` le nombre de fois que la liste d'instructions doit être appliquée
- `l` la liste d'instructions
- `r` notre ruban

Initialement, la fonction définit une sous fonction `liste_execution l r` de type `program -> ruban -> ruban` qui renvoie un nouveau ruban où la liste d'instructions l a été appliquée à notre ruban `r`. Pour cela, la liste fait appel à un `List.fold_left` [ligne 4] qui a pour fonction associée une fonction qui fait, elle, appel à `f_execute`.

La fonction `repeat` est récursive, donc elle fait `n` appel récursif où à chaque itération, le ruban `r` est modifié par la fonction auxiliaire `liste_execution`.

Si la fonction `execute_instruction` existe bel et bien, nos 4 fonctions `go_left`, `go_right`, `write` et `repeat` ont bien été créées et sont fonctionnelles.

2.3 La fonction `execute_program`

Il est ainsi temps de créer notre fonction `execute_program` pour décoder un programme 2Aien dans notre ruban final.

```
let rec execute_instruction r = function
  | Left -> go_left r
  | Right -> go_right r
  | Write(c) -> write c r
  | Repeat(n,l) -> repeat execute_instruction n l r
  | Caesar(n) -> r  (* Pour l'instant... *)
  | Delete(c) -> r  (* Pour l'instant... *)
  | Invert -> r ;;  (* Pour l'instant... *)

let execute_program p =
  let r = {left = []; right = []} in
  let rec execute_aux r = function
    | [] -> r
    | t::q -> let r = execute_instruction r t in execute_aux r q
  in execute_aux r p ;;
```

La fonction `execute_program` est décomposée en deux fonctions auxiliaires.

On commence avec un ruban vide où les deux listes sont des listes vides. On crée alors une sous-fonction auxiliaire récursive `execute_aux` qui prend en paramètre un ruban et un programme. Tant qu'il reste des instructions à appliquer, la fonction auxiliaire fait appel à la fonction `execute_instruction` qui exécute l'instruction et renvoie un nouveau ruban correspondant à l'aide des fonctions `go_left`, `go_right`, `write` et `repeat` précedemment créées.

Quand la liste d'instruction est vide, on renvoie alors notre **ruban final**.

2.4 La fonction `fold_ruban`

```
let fold_ruban f v0 r =
    let acc2 = List.fold_right (fun e acc -> f acc e) r.left v0 in
    List.fold_left f acc2 r.right ;;
```

La fonction a été donnée dans le cours. On applique successivement la fonction `f` dans la liste de gauche avec un `List.fold_right` avec comme accumulateur initial la variable `v_0`, puis on applique de la même manière cette fonction `f` dans la liste de droite avec un `List.fold_left` avec maintenant comme accumulateur initial le résultat obtenu par le `List.fold_right`.

Ainsi, notre fonction `fold_ruban` parcourt le ruban de gauche à droite en appliquant la fonction f.

2.5 Premiers résultats

3 Les 3Aiens

3.1 La fonction `caesar`

```
let map_ruban f r = {left = List.map f r.left; right = List.map f r.right};;

let caesar n r =
  let aux decalage c =
    let code = Char.code c in
    let i = (code - decalage + n) mod 26 in
    let new_code = i + decalage in
    if i >= 0 then Char.chr new_code
    else Char.chr (new_code + 26)
  in
  let applique_caractere c =
    if ('A'<= c && c <= 'Z') then aux 65 c
    else if ('a' <= c && c <= 'z') then aux 97 c
    else c
  in
  map_ruban applique_caractere r ;;
```

3.2 La fonction `delete`

Première idée : utiliser la fonction `fold_ruban`

```
let delete_caractere r =
    match r.right with
    | [] -> r (* failwith "la liste droite est vide" *)
    | _::q -> {left = r.left; right = q};;

let rec delete c r =
    let fold_ruban f v0 r =
        let acc2 = List.fold_right (fun e acc -> f acc e) r.left v0 in
        List.fold_left f acc2 r.right
    in
    let f r caractere =
        if caractere = c then delete_caractere r
        else r
    in
    fold_ruban f r r;;
```

Idée plus rapide et plus intuitive :

```
let delete c r =
  let f_supp = List.filter (fun x -> x <> c) in
  {left = f_supp r.left; right = f_supp r.right} ;;
```

3.3 La fonction `invert`

```
let invert r = {left = r.right; right = r.left};;
```

3.4 Nouvelle fonction `execute_program`

```
let rec execute_instruction r = function
  | Left -> go_left r
  | Right -> go_right r
  | Write(c) -> write c r
  | Repeat(n,l) -> repeat execute_instruction n l r
  | Caesar(n) -> caesar n r
  | Delete(c) -> delete c r
  | Invert -> invert r ;;

let execute_program p =
  let r = {left = []; right = []} in
  let rec execute_aux r = function
    | [] -> r
    | t::q -> let r = execute_instruction r t in execute_aux r q
  in execute_aux r p ;;
```

3.5 Différents tests et exemples

4 Les diplômiiens

4.1 Objectif de cette partie

Après avoir enfin fini la gestion du système de codage 3Aiens. Je suis retourné voir le `Facilitateur - Organisateur - Réprimandeur - Estimateur - Surveillant - Télépathe` en pensant mériter l'honneur d'être chargé du prochain `Traditionnel Défilé` de la compréhension universelle.

Mais, à mon plus grand désarrois, celui-ci m'explique qu'il n'est pas l'un des nôtres mais une entité dont le seul but est de me permettre d'atteindre un jour un plan d'existence supérieur : le plan des des mythiques **diplômiiens**.

Ainsi, le but de cette partie du programme est d'inverser le processus précédent. C'est-à-dire de prendre un message et de l'encoder en 2Aien de telle sorte que le nouveau message comporte aussi peu d'instructions que possible parce que :

— "Le temps c'est de l'argent.

— L'énergie c'est de l'argent.

— L'argent c'est du café.

— Le café c'est notre programme." [1]

Par conséquent, on doit prendre un message (`msg`) de type `char list`, et renvoyer le message encodé en langage 2Aiens, c'est-à-dire de renvoyer une liste d'instructions, donc une liste de type `program = instruction list`.

4.2 Première approche naïve de `generate_program`

Commençons par l'idée la plus naïve (mais aussi la plus naturelle), c'est-à-dire de créer une liste d'instructions remplie de `Write` et de `Right`. En effet, pour cela, il suffit simplement d'encoder le message en écrivant un caractère, puis se décaler à droite, écrire le prochain caractère, et cetera...

Cette fonction s'écrit alors logiquement à l'aide d'une sous-fonction auxiliaire récursive qui parcourt l'entièreté de notre message, et place dans la liste résultante les instructions `Write(t)` et `Right`.

```
1 let generate_program_naif msg =
2     let rec aux = function
3         | [] -> []
4         | t::q -> Write(t)::Right:: aux q
5     in aux msg;;
```

Avec cette méthode, le message est donc bien encodé dans le langage des 2Aiens, et cela de la manière **la plus optimale en terme de complexité temporelle**.

En effet, on parcourt une seule et unique fois notre liste `msg` donc on se retrouve avec `generate_program_naif` de complexité temporelle linéaire en la taille du message. En notant C_{naif} la complexité de l'algorithme naïf, on a :

$$\boxed{C_{naif} = \mathcal{O}(n)}$$

Seulement, il était demandé de construire une fonction `generate_program` de telle sorte que le nouveau message comporte aussi peu d'instructions que possible. Ce qui n'est pas le cas ici, car il y a potentiellement plein de motifs à la suite, ce qui peut être parfaitement géré par l'instruction `Repeat`, non exploité dans `generate_program_naif`.

Le nouvel objectif est donc de construire une fonction `generate_program` qui essaye d'écrire autant que possible l'instruction `Repeat` dans le nouveau message encodé.

4.3 Premier prototype de la fonction avec des `Repeat`

Dans la quête de mon nouveau objectif, ma première idée a été de, en quelque sorte, suivre la démarche de la méthode naïve. Mais il fallait quand même trouver une méthode pour insérer des `Repeat`.

Pour commencer, on crée une fonction récursive `meme_mot_au_debut mot msg` qui vérifie si le `mot` se situe au début du message (`msg`). Elle retourne donc un booléen.

```
let rec meme_mot_au_debut mot msg =
  match (mot, msg) with
  | [], _ -> true
  | _, [] -> false  (* Le mot est trop long *)
  | a::q1, b::q2 -> if a = b then meme_mot_au_debut q1 q2 else false ;;
```

Explications ici...

Ensuite, on crée une petite fonction récursive `enleve_premiers_termes n l` qui supprime les `n` premiers termes d'une liste `l`.

```
let rec enleve_premiers_termes n l =
  if n = 0 then l
  else
    match l with
    | [] -> failwith "la liste est trop petite, ce cas n'est pas sensé arriver"
    | t::q -> enleve_premiers_termes (n-1) q ;;
```

Et pour finir, une petite fonction récursive `ecrire_mot` qui, à la manière de `generate_program_naif`, écrit simplement le mot à l'aide d'instructions `Write` et `Right`.

```
let rec ecrire_mot = function
  | [] -> []
  | t::q -> Write(t)::Right:: ecrire_mot q ;;
```

Voici alors le premier prototype de la fonction `generate_program` qui utilise quelques instructions `Repeat` dans le message sortant.

```
1  let generate_program_1 message =
2    let rec aux prog count mot msg =
3      match msg with
4      | [] ->
5          if count >= 2 then List.rev (Repeat(count, ecrire_mot mot)::
6          Repeat(List.length mot,[Left]) :: prog)
7          else List.rev prog
8      | t :: q ->
9        if count = 0 then aux (Right::Write(t)::prog) 1 (mot @ [t]) q
10       else if meme_mot_au_debut mot msg then
11         let new_msg = enleve_premiers_termes (List.length mot) msg in
12         aux prog (count + 1) mot new_msg
13         else
14           if count >= 2 then
15             aux (Right::Write(t)::Repeat(count, ecrire_mot mot)::
16             Repeat(List.length mot,[Left]) :: prog) 1 [t] q
17           else aux (Right::Write(t)::prog) 1 (mot @ [t]) q
18   in
19   aux [] 0 [] message ;;
```

Voici le résumé du fonctionnement de cette fonction `generate_program_1` en quelques points :

- On définit une liste auxiliaire récursive qui prend en paramètre un programme (`prog`), un accumulateur (`count`), une liste de caractères (`mot`) et un message à encoder (`msg`). [ligne 2]
  - `prog` est une variable de type `program` qui stocke à chaque itération le programme que va renvoyer la fonction à la fin en le modifiant à chaque appel de `aux`.
  - `count` une varible de type `int`. C'est un accumulateur qui compte le nombre de répétitions d'un mot dans le programme.
  - `mot` est une variable de type `char list`. C'est donc une liste qui stocke les lettres qu'on rencontre dans l'optique de vérifier si le mot se répète immédiatement après dans la suite du message.
  - `msg` est simplement les message qu'on doit encoder, donc une variable de type `char list`.
- Cette fonction va faire un match de `msg`. Si le message est vide, on vérifie si le compteur est ≥ 2. Si c'est le cas, on ajoute en plus une instruction `Repeat` correspondant (expliqué plus bas) et autant de `Left` que nécessaire, et dans tous les cas, on renvoie alors l'**inverse** du programme qu'on a stocké en argument. [ligne 5,6,7] (Pourquoi l'inverse ? *J'explique la raison juste après...*)

- Si la liste n'est pas vide (de la forme `t::q`). Il survient **un cas qui n'arrive qu'au début** : `count = 0`. On écrit donc le premier caractère dans notre programme avec `Right::Write(t)::prog`, et on continue le parcours du message. [ligne 9]
- Sinon, on regarde si le motif `mot` est présent au début du message à l'aide de la fonction `meme_mot_au_debut` [ligne 7]. Si c'est le cas, c'est parfait ! On supprime les n premiers caractères du message (avec n la taille du mot) à l'aide de la fonction `enleve_premiers_termes`, on incrémente l'accumulateur, et en continue le parcours avec notre nouveau message. [ligne 11 et 12]
- Et dans le cas où le mot n'est pas le même au debut du message, on ne peut pas incrémenter `count`. Ainsi, on vérifie si `count` est supérieur ou égal à 2, car dans ce cas, on peut **insérer une instruction** `Repeat` ! [ligne 14]
  Seulement, le temps de trouver cette répétition, on a déjà écrit les caractères du mots dans le programme. Voici l'un des grands désavantages de cette fonction car il est donc nécessaire de se décaler à gauche avec un certains nombre d'instruction `Left` avant de mettre notre `Repeat`. [ligne 15 et 16]
- Pour écrire dans l'instruction `Repeat`, on utilise la fonction `ecrire_mot` : `Repeat(count, ecrire_mot mot)`. Le nouveau mot est alors réinitialisé avec `mot = [t]`.
- Et pour finir, s'il n'y a pas le même mot au début du message ET qu'il n'est pas possible d'écrire un `Repeat`, on écrit simplement le caractère `t` et on ajoute `t` à la fin du mot pour de potentiels nouveaux `Repeat`. [ligne 17]
- la fonction auxiliaire commence alors logiquement avec les paramètre `prog = []`, `count = 0` et `mot = []`. On parcourt alors notre `message` avec cette fonction auxiliaire qui crée notre programme souhaité. [ligne 19]

Remarque : Pourquoi donc renvoyer `List.rev prog` à la fin de l'algorithme ? Voici l'explication :

Pour pouvoir insérer nos potentiels `Repeat`, il est nécessaire de construire le programme *à l'envers* pour y avoir accès en permanence dans l'argument de la fonction `aux`. Une idée pour contrer cela aurait été d'écrire `Write(t)::Left::prog` à la place de `Right::Write(t)::prog` aux lignes 6, 12 et 14. Mais je trouve que cela aide encore moins à la compréhension, et ne change pas l'ordre de la complexité totale de l'algorithme (seulement la constante devant).

4.4 Tests et problèmes de cette implémentation

C'est un premier prototype, il est donc loin d'être parfait. Il présente de nombreux défauts, mais aussi quelques avantages que je vais détailler ci-dessous.

4.5 Prototype final de la fonction

Nous venons de voir les problèmes évidents de ce premier prototype de la fonction `generate_program_1`. J'ai donc passé une bonne soirée (qui a *légèrement* entaché la nuit) à réfléchir à une nouvelle manière de voir le problème.

Après quelques idées très vite abandonnée, j'ai finalement choisi de commencer par la recherche de motifs qui se répètent exactement **2 fois d'affilé** dans notre message à partir d'un caractère précis.

4.5.1 Extraction de préfixes de listes

Pour commencer, on regarde donc les motifs de taille n qui se répètent exactement 2 fois d'affilé au début dans notre message avec $n \in [\![1, N_{msg}]\!]$, et on les stocke dans une liste. (N_{msg} étant la taille du message).

Ainsi, j'ai d'abord commencé à définir 2 fonctions : `prefixe` et `double_prefixe`.

- `prefixe n l` retourne la liste comprenant les `n` premiers termes de la liste `l`.

```
1  let rec prefixe n l =
2    if n = 0 then []
3    else
4      match l with
5      | [] -> []
6      | t::q -> t :: prefixe (n-1) q ;;
```

Son fonctionnement n'est pas bien compliqué, c'est une boucle récursive sur `n` qui construit à force des appels récursifs à préfixe la liste souhaitée. [ligne 6]

- `double_prefixe n l` suit la même logique que la fonction `prefixe` en retournant un couple de liste. La première liste retournée correspond à la liste comprenant les `n` premiers termes de la liste `l`, et la deuxième est la liste de taille `n` de `l` qui suit la première. Elle fait un appel pour cela à la fonction `prefixe`.

```
1  let double_prefixe n l =
2    let rec aux m l =
3        if m = 0 then ([], prefixe n l)
4        else
5          match l with
6          | [] -> ([],[])
7          | t::q -> let (l1,l2) = aux (m-1) q in (t::l1, l2)
8    in
9    if List.length l < 2*n then ([],[])
10   else aux n l;;
```

Son fonctionnement n'est, lui-aussi, pas bien compliqué. La fonction fait appel à une sous-fonction récursive `aux` sur la variable `n` qui construit par l'exacte même méthode que `prefixe` la première liste du couple [ligne 7]. Et une fois que celle-ci est construire, il construit la deuxième liste du couple en faisant appel à `prefixe` sur la liste `l` retirée de ses `n` premiers éléments [ligne 3].

À noter que si la liste `l` ne permet pas d'extraire 2 sous-listes de taille `n`, elle renvoie un couple de listes vides, ce qui doit être considéré pour l'implémentation de la fonction `reviens_2_fois` qui suit... [ligne 9]

4.5.2 Reconnaissance de motifs identiques

Maintenant que ces 2 premières fonctions `prefixe` et `double_prefixe` sont créées, l'idée est de s'en servir pour créer une liste de potentiels préfixes qui se suivent et qui sont **identiques** !

Voici donc comment est naît la fonction `reviens_2_fois`.

```
1  let reviens_2_fois n l =
2    let rec aux n =
3      if n = 0 then []
4      else (double_prefixe n l) :: aux (n-1)
5    in
6    let f (x, y) acc =
7      if x <> [] && y <> [] && x = y then x :: acc
8      else acc
9    in List.fold_right f (aux n) [];;
```

Comme introduite au-dessus, la fonction `reviens_2_fois` prend 2 paramètres : un entier `n` et une liste `l`.

Elle définit une sous-fonction auxiliaire récursive `aux` qui construit une liste de couples de listes obtenus par la fonction `double_prefixe k l` en faisant varier `k` de `n` à 1 de manière décroissante [ligne 4]. En pratique, `n` vaudra la partie entière de la taille du message divisée par 2 pour être sûr de récupérer tous les motifs répétés possibles.

Ensuite, elle récupère de cette liste de couples, les premiers éléments de chacun des couples lorsque cela sont **identiques** et différents de `([],[])` (cf. ligne 9 de la fonction `double_prefixe`). Cela s'opère à l'aide de l'opérateur `List.fold_left` avec comme accumulateur initial une liste vide, et la fonction f créée juste au-dessus [ligne 9].

Remarque : Si cela peut aider à la compréhension, il faut savoir que l'appel `List.fold_right f (aux n) []` produit le même résultat que ces lignes de codes-ci :

```
1  let res = List.filter (fun (l1,l2) -> l1 <> [] && l2 <> [] && l1 = l2) (aux n) in
2    let (l1, l2) = List.split res in
3    l1;;
```

Et maintenant, supposons que nous avons trouvé un motif de taille k qui se répète 2 fois à l'aide de la fonction `reviens_2_fois`. C'est déjà une belle avancée, mais il faut quand même vérifier si ce motif ne se répéterait pas par hasard encore 1, 2 ou même 10 fois !

C'est pour cette raison là que la fonction `nombre_repetitions` a été créée. La fonction retourne ainsi un entier correspondant au nombre de répétitions du motif `mot` dans le message actuel `msg`.

```
1  let nombre_repetitions mot n_mot msg =
2    let rec aux msg =
3      let pref = prefixe n_mot msg in
4      if pref <> mot then 0
5      else 1 + aux (enleve_premiers_termes n_mot msg)
6    in aux msg;;
```

La fonction définit une fonction récursive auxiliaire `msg` qui compte le nombre de fois que le mot `pref` de taille `n_mot` apparaît à la suite dans le message. S'il apparaît,

on incrémente le compteur et on enlève les `n_mots` premières lettres du message pour continuer à chercher les potentielles occurrences du mot [ligne 5].

Et il est important de noter que la fonction `enleve_premiers_termes` définie dans le premier prototype de `generate_program` est **aussi utilisée dans le prototype final** (cf. ligne 5 de `nombre_repetitions` et ligne 13 de `generate_program`.

En effet, lorsque qu'un `Repeat(n,l)` est inséré dans notre programme de sortie, il est nécessaire de supprimer toutes les `n` occurrences du mot encodés par `l` dans la suite de notre message. Cela permet de continuer à travailler sur la suite de notre message, qui n'est pas une répétition de ce mot-ci.

4.5.3 Nouvelle implémentation de `generate_program`

Avec ces 5 fonctions auxiliaires définies juste avant, il est maintenant temps d'implémenter la fonction tant recherchée `generate_program`.

On rappelle son objectif : prendre un `message` et l'encoder en 2Aien de telle sorte que le nouveau message encodé comporte aussi peu d'instructions que possible. C'est-à-dire, minimiser le nombre d'instructions en utilisant le plus intelligemment possible des instructions `Repeat`.

Voici donc son implémentation :

```
1  let generate_program_2 message =
2    let n_max = (List.length message) /2 in
3    let rec aux prog msg =
4      match msg with
5      | [] -> List.rev prog
6      | c::q ->
7        let mots_repeat = reviens_2_fois n_max msg in
8        match mots_repeat with
9        | [] -> aux (Right::Write(c)::prog) q
10       | mot::_ ->
11         let n_mot = List.length mot in
12         let nb_rep = nombre_repetitions mot n_mot msg in
13         let new_msg = enleve_premiers_termes (nb_rep*n_mot) msg in
14         aux (Repeat(nb_rep, aux [] mot) :: prog) new_msg
15   in
16   aux [] message;;
```

Quelques explications sur cette fonction :

- La fonction prend un paramètre un `message` à encoder. Elle définit directement un entier `n_max` valant la taille totale du message divisée par 2 (utile dans la suite pour l'appel à `reviens_2_fois`) [ligne 2].

- On définit alors une fonction auxiliaire `aux` prenant 2 arguments : un programme `prog` et un message `msg`.
  - `prog` est une variable de type `program` qui stocke à chaque itération le programme que va renvoyer la fonction à la fin en le modifiant à chaque appel de `aux`.
  - `msg` est simplement les message qu'on doit encoder, donc une variable de type `char list`.
- On *match* alors notre message. Si le message est vide, ça y est, nous avons fini d'encoder le message ! On renvoie alors l'inverse de `prog` qui est le message encodé. (Pouquoi l'inverse ? *Voir la remarque juste en dessous...*) [ligne 5].
- Si le message n'est pas vide (c'est-à-dire de la forme `c::q`), on définit la liste `mots_repeat` à l'aide d'un appel à `reviens_2_fois n_max msg`. Cette liste de mots correspond donc aux préfixes de tailles différentes qui sont répétés dans la suite direct de notre message [ligne 7].
-
  - **Si la liste est vide** [ligne 9] : il n'y a aucune répétition dans la suite directe du message.

    On écrit donc le caractère `c` dans notre programme `prog`, et on réitère notre travail sur la suite du message `q`.
  - **Si la liste n'est pas vide** [ligne 10] : il y a au moins un mot qui se répète dans la suite directe du message.

    On prend alors le mot de taille maximal sans s'intéresser aux autres, on regarde le nombre de fois exact qu'il se repète à l'aide d'un appel à la fonction `nombre_repetitions` [ligne 12]. Et on supprime alors les itérations de ce mot dont nous n'aurons plus besoin par la suite à l'aide d'un appel à `enleve_premiers_termes` [ligne 13].

    Ensuite, on insère un `Repeat` dans notre programme, mais là, cerise sur le gâteau (!), on applique notre fonction auxiliaire récursive sur le reste du message (`new_msg`), **mais aussi dans la liste d'instructions du `Repeat` !!!** [ligne 14]
- Ensuite, comme initialisation de notre fonction auxiliaire, on commence logiquement avec le message entier, et un programme encore vide (`[]`) [ligne 16].

  L'appel à cette fonction auxiliaire renverra alors le programme ainsi souhaité.

Remarque : Pourquoi donc renvoyer `List.rev prog` à la fin de l'algorithme ? Voici l'explication :

Comme expliqué dans le premier prototype de la fonction `generate_program`, pour pouvoir insérer nos potentiels `Repeat`, il est nécessaire de construire le programme *à l'envers* pour y avoir accès en permanence dans l'argument de la fonction `aux`. Une idée pour contrer cela aurait été d'écrire `Write(t)::Left::prog` à la place de `Right::Write(t)::prog` à la ligne 9. Mais je trouve que cela aide encore moins à la compréhension, et **ne change pas l'ordre de la complexité totale de l'algorithme** (seulement la constante devant).

4.5.4 Implémentation finale de la fonction

Notre nouvelle fonction `generate_program` se rapproche bien davantage de notre objectif initial : minimiser le nombre d'instructions dans le programme renvoyé.

Or, comme nous pouvons le remarquer dans son implémentation, ou tout simplement dans le fichier `tests.ml`, **il y a un sérieux problème quand le motif se répète plus de 4 fois**.

En effet, l'appel à `generate_program ['a';'a';'a';'a']);;` retourne `[Repeat (2, [Repeat (2, [Write 'a'; Right])])]`. Ce qui est problématique puisque un simple `Repeat` de "a" de taille 4 suffirait.

Pour palier à cela, j'ai alors écrit la fonction `tous_egaux` :

```
1  let tous_egaux ll =
2    let l = List.concat ll in
3    match l with
4      |[] -> true
5      |t::q -> List.fold_left (fun b x -> b && x = t) true q ;;
```

La fonction prend en entrée une liste de listes de caractères (qui sera créée au préalable par la fonction `reviens_2_fois`) et renvoie un booléen.

En concaténant les listes [ligne 2], elle vérifie si on se trouve dans le cas où tous les motifs de toutes les tailles ne sont que des répétitions d'une seule et même lettre. Cela s'effectue à l'aide d'un appel à `List.fold_left` qui renvoie alors un booléen qui vaut `true` si et seulement si toutes les lettres de la liste concaténée sont les mêmes [ligne 5].

La fonction aurait ainsi aussi pu s'écrire sous cette forme, avec un `List.for_all` qui réalise le même travail :

```
1  let tous_egaux ll =
2    let l = List.concat ll in
3    match l with
4      |[] -> true
5      |t::q -> List.for_all (fun x -> x = t) q ;;
```

Ainsi, voici ma fonction `generate_program` finale :

```
1  let generate_program message =
2    let n_max = (List.length message) /2 in
3    let rec aux prog msg =
4      match msg with
5      | [] -> List.rev prog
6      | c::q ->
7        let mot_repeat = reviens_2_fois n_max msg in
8        match mot_repeat with
9        | [] -> aux (Right::Write(c)::prog) q
```

```
10      | mot::q2 ->
11        if tous_egaux mot_repeat then
12          let lettre = List.hd mot in
13          let nb_rep = nombre_repetitions [lettre] 1 msg in
14          let new_msg = enleve_premiers_termes nb_rep msg in
15          aux (Repeat(nb_rep, aux [] [lettre]) :: prog) new_msg
16        else
17          let n_mot = List.length mot in
18          let nb_rep = nombre_repetitions mot n_mot msg in
19          let new_msg = enleve_premiers_termes (nb_rep*n_mot) msg in
20          aux (Repeat(nb_rep, aux [] mot) :: prog) new_msg
21  in
22  aux [] message;;
```

La seule chose qui change par rapport à l'ancienne version de `generate_program` se situe entre les lignes 11 et 15.

- Avant de faire la procédure habituelle de l'ancienne version de la fonction, on vérifie à l'aide de la fonction `tous_egaux` si la liste de listes `mot_repeat` ne contient pas finalement qu'une seule et même lettre répétée un nombre de fois différent [ligne 11].
- Si c'est le cas, on récupère alors cette lettre, on regarde combien de fois elle se répète à l'aide d'un appel à la fonction `nombre_repetitions` [ligne 13], on enlève le nombre de lettres correspondants à notre message, et il est l'heure d'insérer notre `Repeat` dans notre programme.
- On insère donc notre `Repeat` avec, ici, le mot constitué d'une seule et unique lettre : `[lettre]` [ligne 15].

On se retrouve donc avec une très bonne fonction `generate_program`, qui minimise correctement le nombre d'instructions finales, mais malheureusement pas encore de manière optimale... (Cela sera détaillé dans la partie *tests*).

4.5.5 Complexité de la fonction `generate_program`

Notons T le coût de la complexité temporelle dans le pire des cas. L'algorithme `generate_program` utilise une fonction récursive dans son implémentation. On peut donc écrire que :

$$T(n) = T_{aux}(n) + f(n)$$

Avec :

- T_{aux} le coût de la complexité temporelle dans le pire des cas de la fonction auxiliaire `aux`

- n : la taille du message
- f : une fonction de coût intervenant à l'appel de `generate_program` (ligne 2)

En analysant le fonctionnement le l'algorithme, la relation de récurrence sur le coût T_{aux} est alors donnée par :

$$T_{aux}(n) = T_{aux}(n-1) + T_1(n) + T_2(n) + T_3(n) + T_4(n) + g(n)$$

Avec :

- n : la taille du message
- T_1 : la fonction de coût de `reviens_2_fois` (ligne 7)
- T_2 : la fonction de coût de `tous_egaux` (ligne 11)
- T_3 : la fonction de coût de `nombre_repetitions` (ligne 13 et ligne 18 (*dépend du résultat de* ***`tous_egaux`***))
- T_4 : la fonction de coût de `enleve_premiers_termes` (ligne 14 et ligne 19)
- g est une fonction de coût intervenant à chaque appel de la fonction (en particulier ligne 5, 12 et 17)
- T_{aux} apparaît ligne 9 avec l'appel potentiel de `aux` sur une liste q de taille $n-1$. Si elle n'est pas appelée là, alors la fonction `aux` est appelée ligne 15 ou 20. Dans les deux cas, le raisonnement est quasi le même, donc raisonnons avec l'appel ligne 20 :
  La fonction `aux` est appelée sur la liste `new_msg` de taille **au plus** $(n - 2 \times n_{mot})$, et sur la liste `mot` de taille n_{mot} dans le `Repeat`.
  Donc, on a : $T_{aux}(n - 2 \times n_{mot}) + T_{aux}(n_{mot}) \leq T_{aux}(n-1)$

Il est maintenant temps d'estimer chacune de ses fonction de coût.

— Les fonctions `List.rev` et `List.length` sont des fonctions linéaires en la taille de la liste en argument. Donc ici, f et g sont en $\mathcal{O}(n)$.

— Pour T_1 :

- ▶ La fonction `prefixe` est en $\mathcal{O}(n)$ avec n la taille de la liste en argument.
- ▶ De la même manière, `double_prefixe` est en $\mathcal{O}(n/2) + \mathcal{O}(n/2) = \mathcal{O}(n)$ avec n la taille de la liste en argument.
- ▶ la fonction `reviens_2_fois n l` est composée d'une fonction auxiliaire qui, à chaque appel, fait appel à la fonction `double_prefixe` de complexité $\mathcal{O}(n)$. Donc la fonction auxiliaire est en $\mathcal{O}(n^2)$.
- ▶ `List.fold_right` est en $\mathcal{O}(m)$ avec m la taille de la liste qu'il doit parcourir, c'est-à-dire la taille de (`aux n`) ici.
  Or, `aux n` produit une liste de taille au plus n, donc `reviens_2_fois` est en $\mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2)$.
- ▶ Pour être encore plus précis, ici, le n vaut en réalité au plus $n_{max} = n/2$ avec n la taille du message, donc la complexité est en :

$$\mathcal{O}\left(\left(\frac{n}{2}\right)^2\right) = \mathcal{O}(n^2)$$

(Cela ne change effectivement rien, mais la précision, c'est important parfois !)

- Pour T_2 : La fonction n'est pas récursive, et `List.concat` et `List.fold_left` sont linéaires en la longueur de la liste donc `tous_egaux` est en $\mathcal{O}(n)$.
- Pour T_3 : On remarque assez rapidement que `nombre_repetitions` est en $\mathcal{O}(n)$.
- Pour T_4 : De la même manière, on remarque assez vite que `enleve_premiers_termes` est en $\mathcal{O}(n)$.

Ainsi, d'après la relation de récurrence sur T_{aux}, on en déduit que :

$$\begin{aligned} T_{aux}(n) &= \sum_{k=1}^{n} [T_1(k) + T_2(k) + T_3(k) + T_4(k) + g(k)] \\ &= \sum_{k=1}^{n} \left[\mathcal{O}(k^2) + \mathcal{O}(k) + \mathcal{O}(k) + \mathcal{O}(k) + \mathcal{O}(k)\right] \\ &= \sum_{k=1}^{n} \mathcal{O}(k^2) \\ &= \mathcal{O}\left(\sum_{k=1}^{n} k^2\right) \\ & \text{(par sommation des relations de comparaison dans le cas des séries divergentes)} \\ &= \mathcal{O}\left(\frac{n(n+1)(2n+1)}{6}\right) \\ \implies \quad T_{aux}(n) &= \mathcal{O}(n^3) \end{aligned}$$

Comme on a $T(n) = T_{aux}(n) + \mathcal{O}(n)$, on en déduit donc que :

$$\boxed{T(n) = \mathcal{O}(n^3)}$$

La complexité temporelle de l'algorithme `generate_program` dans le pire des cas est donc en $\mathcal{O}(n^3)$.

4.6 Différents exemples

Dans l'optique de mieux illustrer le fonctionnement des différentes fonctions `generate_program`, rien de mieux que quelques exemples.

Pour chacune de ces implémentations, nous allons faire 4 tests :

```
generate_program (string_to_char_list "Bonjour");;

generate_program (string_to_char_list "lalalala");;

generate_program (string_to_char_list "lala ou blablabla");;

generate_program (string_to_char_list "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
↪ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
);;
```

Avec l'aide de la petite fonction `string_to_char_list` qui n'est pas bien méchante. Elle renvoie simplement une liste de caractères correspondant à la chaîne de caractères de type `string` en entrée. Elle a pour seule utilité de m'aider à la tâche durant les tests.

```
let string_to_char_list str =
  let len = String.length str in
  let rec aux i acc =
    if i < 0 then acc
    else aux (i-1) (str.[i] :: acc)
  in aux (len-1) [];;
```

4.6.1 Implémentation naïve

```
# generate_program_naif (string_to_char_list "Bonjour");;
- : instruction list =
[Write 'B'; Right; Write 'o'; Right; Write 'n'; Right; Write 'j'; Right;
 Write 'o'; Right; Write 'u'; Right; Write 'r'; Right]

# generate_program_naif (string_to_char_list "lalalala");;
- : instruction list =
[Write 'l'; Right; Write 'a'; Right; Write 'l'; Right; Write 'a'; Right;
 Write 'l'; Right; Write 'a'; Right; Write 'l'; Right; Write 'a'; Right]

# generate_program_naif (string_to_char_list "lala ou blablabla");;
- : instruction list =
[Write 'l'; Right; Write 'a'; Right; Write 'l'; Right; Write 'a'; Right;
 Write ' '; Right; Write 'o'; Right; Write 'u'; Right; Write ' '; Right;
 Write 'b'; Right; Write 'l'; Right; Write 'a'; Right; Write 'b'; Right;
 Write 'l'; Right; Write 'a'; Right; Write 'b'; Right; Write 'l'; Right;
 Write 'a'; Right]

# generate_program_naif (string_to_char_list "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
↪   aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
- : instruction list =
[Write 'a'; Right; Write 'a'; Right; Write 'a'; Right; Write 'a'; Right;
 Write 'a'; Right; Write 'a'; Right; Write 'a'; Right; Write 'a'; Right;
 Write 'a'; Right; Write 'a'; Right; Write 'a'; Right; Write 'a'; Right;
 Write 'a'; Right; Write 'a'; Right; Write 'a'; Right; Write 'a'; Right;
 (*Et cetera... *)
```

Ici, rien de bien surprenant, la fonction retourne seulement un succession de `Write` et de `Right` pour encoder un message. **La fonction marche, mais elle est loin de l'optimalité recherchée** en terme de minimisation de nombre d'instructions. (Cela se remarque bien sur le 4ème exemple qui est une succession de caractère `a`.)

4.6.2 Premier prototype

```
# generate_program_1 (string_to_char_list "Bonjour");;
- : instruction list =
[Write 'B'; Right; Write 'o'; Right; Write 'n'; Right; Write 'j'; Right;
 Write 'o'; Right; Write 'u'; Right; Write 'r'; Right]

# generate_program_1 (string_to_char_list "lalalala");;
- : instruction list =
[Write 'l'; Right; Write 'a'; Right; Repeat (2, [Left]);
 Repeat (4, [Write 'l'; Right; Write 'a'; Right])]

# generate_program_1 (string_to_char_list "lala ou blablabla");;
- : instruction list =
[Write 'l'; Right; Write 'a'; Right; Repeat (2, [Left]);
 Repeat (2, [Write 'l'; Right; Write 'a'; Right]); Write ' '; Right;
 Write 'o'; Right; Write 'u'; Right; Write ' '; Right; Write 'b'; Right;
 Write 'l'; Right; Write 'a'; Right; Write 'b'; Right; Write 'l'; Right;
 Write 'a'; Right; Write 'b'; Right; Write 'l'; Right; Write 'a'; Right]

# generate_program_1 (string_to_char_list "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
↪  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
- : instruction list =
[Write 'a'; Right; Repeat (1, [Left]); Repeat (100, [Write 'a'; Right])]
```

Pour ce premier prototype, on ressent que la fonction a un fonctionnement, disons, assez *spécial*...

- Lorsque nous faisons face à une répétition, comme dans le message `"lalalala"` par exemple, elle écrit dans le programme les caractères `l` et `a` avant de se rendre compte qu'il y a effectivement une répétition du mot `"la"`. Elle insère alors le nombre de `Left` nécessaire pour écraser ce qu'elle vient d'écrire, et insère finalement un beau `Repeat` du mot `"la"` écrit 4 fois.
- Seulement, lorsqu'il y a une "coupure" dans le motif qui se répète (comme dans l'exemple `"lala ou blablabla"`), elle détecte la répétition du mot `"la"` mais pas la répétition du mot `"blablabla"` car ils sont séparés d'un motif qui ne se répète pas : `"␣ou␣"`.
- Par contre, elle détecte parfaitement la répétition de la lettre `"a"` dans le dernier test, ce qui est une très bonne chose !

La fonction marche, mais on est encore très loin de l'optimalité recherchée. Cette implémentation présente finalement une multitude de défauts pour peu de réelles qualités.

Il reste à voir ce que cela donne pour les deux autres implémentations...

4.6.3 Deuxième prototype

```
# generate_program_2 (string_to_char_list "Bonjour");;
- : instruction list =
[Write 'B'; Right; Write 'o'; Right; Write 'n'; Right; Write 'j'; Right;
 Write 'o'; Right; Write 'u'; Right; Write 'r'; Right]

# generate_program_2 (string_to_char_list "lalalala");;
- : instruction list =
[Repeat (2, [Repeat (2, [Write 'l'; Right; Write 'a'; Right])])]

# generate_program_2 (string_to_char_list "lala ou blablabla");;
- : instruction list =
[Repeat (2, [Write 'l'; Right; Write 'a'; Right]); Write ' '; Right;
 Write 'o'; Right; Write 'u'; Right; Write ' '; Right;
 Repeat (3, [Write 'b'; Right; Write 'l'; Right; Write 'a'; Right])]

# generate_program_2 (string_to_char_list "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
↪  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
- : instruction list =
[Repeat
  (2,
   [Repeat
     (2,
      [Repeat
        (2, [Repeat (2, [Repeat (2, [Repeat (3, [Write 'a'; Right])])])]);
       Write 'a'; Right])])]
```

On remarque premièrement que la solution donnée pour l'exemple 3 (le `"lala ou blablabla"`) est **optimale**. On a bien un `Repeat` de taille 2 pour le "la" et un de taille 3 pour le "bla".

De plus, il y a bien une diminution du nombre d'instructions dans les exemples 2 et 4 puisque on se retrouve avec des `Repeat` de `Repeat` (etc...) et non plus seulement des `Write` et des `Right`. Mais, un seul `Repeat` de taille 4 pour l'exemple 2 et un seul de taille 100 pour l'exemple 4 suffiraient.

Cette implémentation de `generate_program` fonctionne, et **est bien plus aboutie** et présente de nombreuses qualités dans la reconnaissance de motifs. Mais une répétition trop importante d'un même motif provoque une succession de `Repeat` qui pourrait être remplacé par un seul...

Ainsi, on se rapproche grandement de l'optimalité tant recherchée, mais il reste encore quelques défauts.

4.6.4 Implémentation final

```
# generate_program (string_to_char_list "Bonjour");;
- : instruction list =
```

```
[Write 'B'; Right; Write 'o'; Right; Write 'n'; Right; Write 'j'; Right;
 Write 'o'; Right; Write 'u'; Right; Write 'r'; Right]

# generate_program (string_to_char_list "lalalala");;
- : instruction list =
[Repeat (2, [Repeat (2, [Write 'l'; Right; Write 'a'; Right])])]

# generate_program (string_to_char_list "lala ou blablabla");;
- : instruction list =
[Repeat (2, [Write 'l'; Right; Write 'a'; Right]); Write ' '; Right;
 Write 'o'; Right; Write 'u'; Right; Write ' '; Right;
 Repeat (3, [Write 'b'; Right; Write 'l'; Right; Write 'a'; Right])]

# generate_program (string_to_char_list "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
↪  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
- : instruction list = [Repeat (100, [Write 'a'; Right])]
```

Cette implémentation finale de `generate_program` est une simple amélioration du deuxième prototype. Ainsi, les résultats sont similaires pour les 3 premiers exemples, mais l'ajout de la fonction `tous_egaux` permet d'améliorer parfaitement les retours pour des exemples similaires à l'exemple 4.

En effet, on se retrouve avec un unique `Repeat` de taille 100 pour l'exemple 4, ce qui est **optimal**.

Par conséquent, sur ces 4 exemples, la fonction `generate_program` fournit **3 programmes optimaux** en terme de minimisation du nombre d'instructions (les exemples 1,3 et 4). Seulement, l'exemple 2 a été amélioré par rapport à l'implémentation naïve, mais il n'est pas totalement optimal.

On a bien un `Repeat` de `"lala"`, mais l'algorithme a ensuite appliqué récursivement un `Repeat` sur ce même motif, et a trouvé un nouveau motif pour faire un `Repeat` : le mot `"la"`. Ce n'était pas nécessaire, mais c'est comme cela qu'est construit la fonction `generate_program`.

Ainsi, **la fonction marche, et renvoie un résultat quasiment optimal dans tous les cas**.

Références

[1] Julien Forest. Citations du projet IPFL. http://web4.ensiie.fr/~forest/, 2023.