



Introduction à la Programmation Fonctionnelle

Projet 2022-2023

Richard CHEAM

Table des matières

	Page
1 Un petit message	3
2 Désignation structurelle du programme	3
3 Synopsis du fonctionnement du programme	4
4 Clarification des choix effectués	4
5 Problèmes techniques et leur résolutions	9
6 Quelques tests	14
7 Limites du programme	15

1 Un petit message

Puisque mon **français** est encore limité, par conséquent, ce rapport contiendra inévitablement des erreurs en termes de grammaire et de l'orthographe. J'espère que vous savez que j'ai essayé de le faire de mon mieux et vous ne tiendrez pas en compte strictement de ces erreurs. Je voudrais vous remercier pour votre compréhension.

2 Désignation structurelle du programme

Types principaux de ce programme :

- **ruban** : un zipper avec 2 types enregistrés, **left**: `char list` et **right**: `char list`.
- **instruction** : le type énuméré contient 7 valeurs possibles et chaque valeur est simplement une indication pour le programme de faire une certaine chose comme indiqué ci-dessous :
 - **Left** : déplacement du curseur vers la gauche.
 - **Right** : déplacement du curseur vers la droite.
 - **Write of char** : écriture du caractère sur le ruban.
 - **Repeat of (int * instruction list)** : repeter `n(int)` fois la liste d'instructions.
 - **Caesar of int** : encodage de Caesar de pas `n` au message.
 - **Delete of char** : suppression du caractère à du message.
 - **Invert** : retournement du message.
- **program** : une liste d'instruction.

Fonctions utilisées dans ce programme (ma partie) :

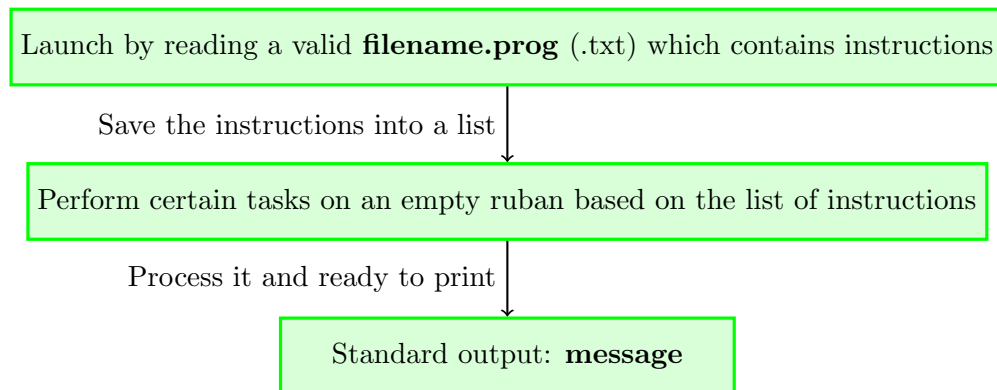
- **execute_program** : `program -> ruban` : prend un type **program** et fait les instructions sur un ruban vide.
- **fold_ruban** : `('a -> char -> 'a) -> 'a -> ruban -> 'a` : prend une fonction, un accumulateur initial, et un ruban. Il applique la fonction sur chaque élément de ruban, et à chaque fois il met à jour l'accumulateur.
- **move_left** : `ruban -> ruban` : fait un déplacement à gauche.
- **move_right** : `ruban -> ruban` : fait un déplacement à droite.
- **write** : `char -> ruban -> ruban` : écrit un caractère **char** sur le ruban.
- **repeat** : `int -> instruction list -> ruban -> ruban`: répète **int** fois la liste instruction d'entrée sur le ruban.
- **caesar** : `int -> ruban -> ruban` : applique l'encodage de Caesar de **int** décalage sur le ruban.
- **delete** : `char -> ruban -> ruban` : supprime le caractère **char** d'un ruban.
- **invert** : `ruban -> ruban` : fait l'inverse sur le ruban.
- **generate_program** : `char list -> program`: donne une liste des instructions qui permet de générer un message qui est identique à ceux qui se trouvent à l'intérieur de `char list` d'entrée.

3 Synopsis du fonctionnement du programme

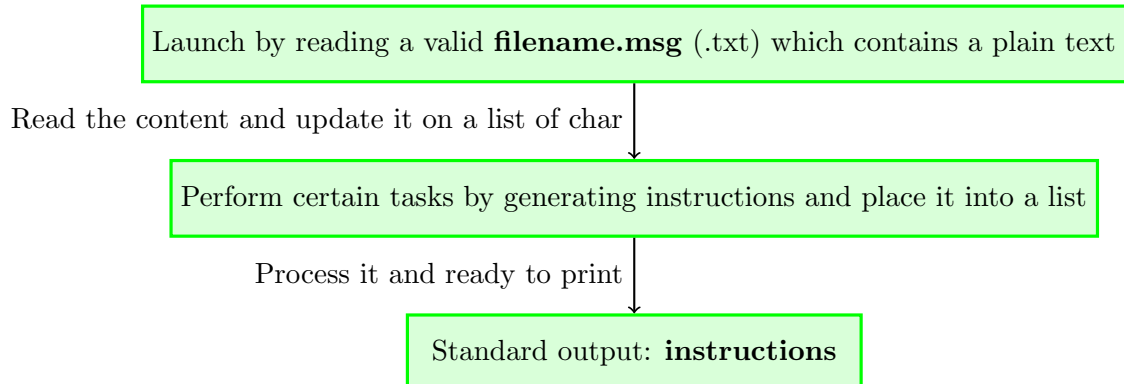
Ce programme prend 3 arguments : assembler output, numéro de phase, et fichier.

Il y a 3 phases dans ce programme; en effet, il pourrait faire 2 choses différents :

- **Phase 1 ou 2** : il lit un fichier qui contient les instructions à exécuter par exemple, `W(a)`; puis, il traduit ces raccourcis de l'instruction en forme `type instruction` et ça effectue les instructions sur un ruban. En fin, il imprime le message sur la sortie standard.



- **Phase 3** : il lit un fichier également, traduit le message en une liste de caractères. Ensuite, il génère la liste des instructions qui écrit un message comme le message d'entrée.



4 Clarification des choix effectués

- **type ruban**: ça permet de naviguer dans une liste de caractères, d'élément en élément. Pour ce programme là, on a besoin de ruban infini; de plus, comme éventuellement, on appliquera des modifications sur le ruban tel que déplacer à gauche, déplacer à droite, etc. Donc, un zipper est un type parfaitement adapté à cette procédure. Je n'ai pas besoin d'un type curseur dans le ruban comme il sera difficile à mettre à jour après chaque opération.

```

1 type ruban = {
2     left : char list;
3     right : char list;
4 }
5

```



Figure 1: zipper d'une liste (dans le cours)

D'après Figure 1, on peut voir que la ligne verticale sépare la liste à gauche et à droite, donc chaque fois on utilise la fonction `move_left` ou `move_right` cette ligne se déplace à gauche ou à droite de manière correspondante à ce que l'on appelle. Par exemple, si on appelle la fonction `move_right`, on obtiendra le zipper comme indiqué dans Figure 2.



Figure 2: zipper après un déplacement à droite (dans le cours)

- `move_right : ruban -> ruban` : j'ai fait le match pour la liste à droite et donc si elle est vide, le programme juste simplement renvoie `ruban r`, sinon le premier élément d'une liste à droite va devenir le premier élément d'une liste à gauche. (Figure 2)

```
1 let move_right = fun r ->
2   match r.right with
3   | [] -> r
4   | h :: t ->
5     {
6       left = (h :: r.left);
7       right = t;
8     }
9 ;;
```

- `move_left : ruban -> ruban` : de même manière que `move_right` juste au sens l'inverse.

```
1 let move_left = fun r ->
2   match r.left with
3   | [] -> r
4   | h :: t ->
5     {
6       left = t;
7       right = (h :: r.right);
8     }
9 ;;
```

- `write : char -> ruban -> ruban` : pour cela, j'ai fait la concaténation sur la liste à droite à chaque fois que la fonction `write` a été appelée. En faisant ça, il y aura un problème quand le curseur n'a pas encore bougé. On va voir plus dans la partie 4.

```
1 let write = fun c r -> { left = r.left ; right = c::r.right };;
2
```

- **fold_ruban** : ('a -> char -> 'a) -> 'a -> ruban -> 'a : comme c'est compliqué de faire la fonction fold pour un zipper, donc l'astuce ici c'est qu'on convertit d'abord le zipper comme une liste et après juste simplement appliquer l'algorithme pour fold une liste.

```

1 let rub_to_start = fun r -> {left = []; right = List.rev_append r.left r.right};;
2
3 let fold_ruban = fun f v0 r ->
4   let _r = rub_to_start r in
5   match _r.right with
6   | [] -> failwith "No message"
7   | l ->
8     let rec fold_ruban_aux = fun f v0 l ->
9       match l with
10      | [] -> v0
11      | h :: t -> fold_ruban_aux f (f v0 h) t
12    in fold_ruban_aux f v0 l
13  ;;
14

```

Ici on a besoin d'aide de la fonction **rub_to_start** pour convertir le zipper en une liste. L'implémentation de **rub_to_start** est juste mettre la liste à gauche à la liste à droite grâce à la fonction **List.rev_append** qui existe déjà dans OCaml. On fait l'inverse pour la liste à gauche puis l'ajouter à droite, donc comme ça on obtiendra une liste dans le même ordre que zipper sans un curseur.

- **execute_program** : program -> ruban : comme dans le sujet dans la partie **3.1.2 Travail à rendre**, il faut implementer la fonction permettant le décodage du message qui a le type de **program -> ruban**; par conséquent, la récursivité terminale est ce que j'ai fait; en effet, c'est une fonction dont la dernière instruction est un appel récursif. De plus, le match **Write** a été dû mettre en bas (explication est dans la partie 4).

```

1 let execute_program = fun p ->
2   match p with
3   | [] -> failwith "No instruction"
4   | h :: t ->
5     let rec execute_program_aux = fun p r ->
6       match p with
7       | [] -> r
8       | h :: t ->
9         match h with
10        | Left -> execute_program_aux t (move_left r)
11        | Right -> execute_program_aux t (move_right r)
12        | Caesar n -> execute_program_aux t (caesar n r)
13        | Delete(c) -> execute_program_aux t (delete c r)
14        | Invert -> execute_program_aux t (invert r)
15        | Repeat(n,li) ->
16          let rec repeat = fun n li r ->
17            if n = 0 then r
18            else repeat (n-1) li (execute_program_aux li r)
19          in execute_program_aux t (repeat n li r)
20        | Write c ->
21          match t with (* check if the next instruction is Write as well, meaning we
22            have not moved yet *)
23          | Write c' :: _ -> execute_program_aux (List.tl t) (write c' r) (*List.tl
24            sends back the input list without its head element*)
25          | _ -> execute_program_aux t (write c r)

```

```

24   in execute_program_aux p {left = []; right = []}
25 ;;
26

```

Sortie standard : `val execute_program : instruction list -> ruban = <fun>`

- `repeat : int -> instruction list -> ruban -> ruban` : pour ce programme, la fonction `repeat` a été implémenté directement dans la fonction `execute_program` parce qu'il a besoin d'appeler la fonction `execute_program_aux` si le nombre de fois `n` à répéter est plus grand que 0.
- `caesar : int -> ruban -> ruban` : ici j'ai implémenté la fonction `caesar` à l'aide `_caesar` et `fold_ruban`. `_caesar` a un rôle important à jouer, il applique l'encodage de Caesar de pas `n` à un character `c`.

```

1 let _caesar = fun n c ->
2   let i = Char.code c in
3   if (i >= 97 && i <= 122) then (* check if they are lowercase alphabet from a-z in
4     ASCII *)
5     let to_upper = Char.uppercase_ascii c in (* convert the input char c to upper
6       because the procedure of shifting only works with uppercase alphabet *)
7     let _i = Char.code to_upper in
8     let tmp1 = _i - 65 in
9     let tmp2 = (tmp1 + n) mod 26 in
10    let res_int = tmp2 + 65 in
11    let res_char = Char.chr res_int in
12    Char.lowercase_ascii res_char (* convert it back to lower *)
13  else
14    let tmp1 = i - 65 in
15    let tmp2 = (tmp1 + n) mod 26 in
16    let res = tmp2 + 65 in
17    Char.chr res
18 ;;
19
20 let caesar = fun n r ->
21   fold_ruban (fun acc h -> let c = _caesar n h in {left = c::acc.left; right = acc.
22     right}) {left = []; right = []} r
23 ;;
24

```

Donc, pour appliquer l'encodage de Caesar de pas `n` au message, il faut appeler `_caesar` à chaque élément d'un ruban car le message là est en forme de ruban. Grâce à la fonction `fold_ruban` je pourrais appliquer `_caesar` à chaque character dans le ruban et chaque fois je fais la concaténation à un nouvel ruban.

- `delete : char -> ruban -> ruban` : ici c'est simplement juste supprimer le caractere qui est identique à l'argument que l'on passe dans la fonction `delete`. Pour faire ça, j'ai fait l'appel `fold_ruban` en sorte que je pourrais parcourir dans le ruban en recherchant si le caractere je rencontre à chaque fois est le même que celui qui a été saisi. Si l'élément que l'on rencontre est même que caractère d'entrée ici c'est `a`, donc on ne modifie rien l'accumulateur (un ruban vide) parce que on veut un ruban sans caractère `a`. Sinon on fait la concatenation au ruban car il n'est pas identique que `a`, c'est-à-dire on remet dans le ruban.

```

1 let delete = fun a r ->

```

```

2  fold_ruban (fun acc h -> if a = h then {left = acc.left; right = acc.right} else {
    left = h::acc.left; right = acc.right}) {left = []; right = []} r
3  ;;
4

```

- **invert** : `ruban -> ruban` : pour cela, j'ai fait juste l'inverse de `ruban`; en effet, mettre la liste de gauche à droite et la liste de droite à gauche. Par exemple, après le `ruban` a été inversé, si je fais l'appel la fonction `move_left`, l'algorithme de `move_left` fonctionnera de même manière que précédente mais ce que a changé, c'est que le programme se déplace vers la droite au lieu de la gauche parce qu'après avoir inversé le `ruban`, c'est-à-dire, ayant le remis au sens de l'origine, on va constater que ce que l'on a déplacé se trouvait dans la direction opposée.

```

1  let invert = fun r -> {left = r.right; right = r.left};;
2

```

- **generate_program** : `char list -> program` : cette fonction prend une list de caractère et retourne un type `program`, c'est-à-dire instruction list; de plus, la fonction `read_file` dans `main`, lit un fichier qui contient un message par exemple,

hello

Donc ça va donner une liste de caractere

`['h' ; 'e' ; 'l' ; 'l' ; 'o']`

Donc j'ai fait le match juste simplement si la liste est vide donc renvoie l'accumulateur, sinon il y a 3 instructions à prendre en compte:

- **Write** : on veut des instructions qui permet de générer un message donc cette instruction est la plus importante.
- **Right** : on doit déplacer le curseur chaque fois.
- **Repeat** : ça marche aussi sans cette instruction, mais ici ça permet d'éviter la répétition de **Write** un caractère si le prochaine caractère est le même.

Ici, si `t` est vide, c'est-à-dire, il n'y a que un élément donc je l'écrirai en le mettant dans l'accumulateur. Et si, le premier élément est identique que la deuxième, c'est-à-dire, il y a des lettres consécutives, donc on va prendre en compte l'instruction **Repeat** à l'aide des fonctions ci-dessous :

- **count_consecutive** : `'a list -> int` : cette fonction ne compte pas le nombre d'occurrence d'une lettre dans la liste; en effet, elle compte le nombre d'alphabet consécutif dans une liste. Elle a été implémenté spécifiquement pour utiliser dans la fonction `generate_program`.

```

1  let rec count_consecutive = fun l ->
2    match l with
3    | [] -> 0
4    | [h] -> 1
5    | h1 :: h2 :: t ->
6      if h1 = h2 then 1 + count_consecutive (h2 :: t)
7      else 1
8  ;;
9

```


- `remove_element` : `int -> 'a list -> 'a list` : cette fonction enlever les `n` premiers éléments dans la liste d'entrée et renvoie une liste sans des `n` premiers éléments. Je l'ai besoin parce qu'il faut supprimer des lettre consecutives avant la prochaine appel de la fonction `generate_program_aux`, si j'utilise `t`, le numéro de la lettre consécutive apparaîtra plus qu'il ne l'est dans le message.

```

1 let rec remove_element n lst =
2   if n <= 0 then lst
3   else match lst with
4     | [] -> []
5     | _ :: tl -> remove_element (n-1) tl
6 ;;
7

```

```

1 let generate_program = fun l ->
2   let rec generate_program_aux = fun acc l ->
3     match l with
4     | [] -> acc
5     | h :: t ->
6       if t = [] then ([Right; Write h]@acc)
7       else if h = (List.hd t) then
8         let count = count_consecutive l in
9         generate_program_aux ((Repeat(count, [Write h; Right]))::acc) (remove_element
count l)
10      else generate_program_aux ([Right; Write h]@acc) t
11   in List.rev (generate_program_aux [] l)
12 ;;
13

```

J'ai créé une variable `count` pour savoir combien de fois je dois écrire la lettre et pour être plus lisible également. Et donc, j'ajoute l'instruction `Repeat` dans l'accumulateur et j'appelle récursivement la fonction `generate_program_aux`, mais ce fois, je l'applique sur la liste sans des `count` éléments. Par exemple,

`generate_program ['a' ; 'a' ; 'a' ; 'b' ; 'c']`

Dans ce cas là, `count` est égal à 3. Donc, j'ajoute `Repeat(3, [Write a; Right])` dans l'accumulateur. Chaque écriture doit se déplacer, sinon elle va écrire au même endroit et donnera éventuellement juste une lettre a. Et après, j'appelle la fonction aux sur la liste sans 3 premiers éléments :

`generate_program_aux ['b' ; 'c']`

Par conséquent, ça va donner des instructions :

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

Par ailleurs, le symbole `@` a été utilisé pour faire la concatenation entre deux liste. Et `List.rev` a été appelée pour faire l'inverse l'accumulateur. On va voir plus dans la partie 5.

5 Problèmes techniques et leur résolutions

- `execute_program` : lors de l'appel à `execute_program_aux t` à la ligne 7, `t` est encore une liste d'instructions et non une instruction, donc j'ai dû faire le match encore une fois.

```

1 let execute_program = fun p ->
2   match p with
3   | [] -> failwith "No instruction"
4   | h :: t ->
5     let rec execute_program_aux = fun h r ->
6       match h with
7       | Write c -> execute_program_aux t (write c r)
8       in execute_program_aux h {left = []; right = []}
9   ;;

```

```

1 let execute_program = fun p ->
2   match p with
3   | [] -> failwith "No instruction"
4   | h :: t ->
5     let rec execute_program_aux = fun p r ->
6       match p with
7       | [] -> r
8       | h :: t ->
9         match h with
10        | Write c -> execute_program_aux t (write c r)
11        in execute_program_aux p {left = []; right = []}
12   ;;

```

- **repeat** : au début, je voulais implémenter la fonction **repeat** à l'extérieur **execute_program** comme **move_left**, **move_right**, etc, mais je me suis rendu compte qu'il fallait appeler **execute_program_aux**. Donc, c'est inévitable de l'implémenter à l'intérieur. (Voyez la partie 3)
- **caesar** : pour cette fonction, l'encodage de Caesar a marché bien mais juste pour l'alphabet minuscule.

```

1 let _caesar = fun n c ->
2   let i = Char.code c in
3   let tmp1 = i - 65 in
4   let tmp2 = (tmp1 + n) mod 26 in
5   let res = tmp2 + 65 in
6   Char.chr res
7 ;;

```

Donc, j'ai dû utiliser built-in fonctions **Char.uppercase_ascii** et **Char.lowercase_ascii** que j'ai trouvé dans la documentation Ocaml. (Voyez ci-dessus)

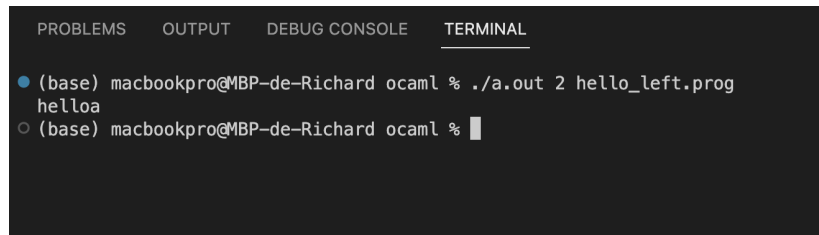
- **write** : comme j'ai mentionné dans la partie 3, j'ai fait la concatenation et donc lorsque le programme n'a pas encore bougé, il renvoyera le faux message.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
• (base) macbookpro@MBP-de-Richard ocaml % ocamlc last.ml
• (base) macbookpro@MBP-de-Richard ocaml % ./a.out 1 helloa
helloa
○ (base) macbookpro@MBP-de-Richard ocaml %

```

Figure 3: l'affichage de hello.prog



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
• (base) macbookpro@MBP-de-Richard ocaml % ./a.out 2 hello_left.prog
helloa
○ (base) macbookpro@MBP-de-Richard ocaml %

```

Figure 4: l’affichage de hello_left.prog

Pour fixer, j’ai dû faire la vérification pour voir si la prochaine instruction est encore **Write**, et si c’est le cas, le programme juste écrit l’instruction suivante à la place de dernière. Par instance, si j’ai le type **program** comme ça :

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

Donc, je vais ignorer **W(a)**, donc le résultat va être **b**.

Dans OCaml,

```

1 |Write c ->
2   match t with
3   |Write c' :: _ -> execute_program_aux (List.tl t) (write c' r)
4   |_ -> execute_program_aux t (write c r)

```

Mais,

```

1 let execute_program = fun p ->
2   match p with
3   |[] -> failwith "No instruction"
4   |h :: t ->
5     let rec execute_program_aux = fun p r ->
6       match p with
7       |[] -> r
8       |h :: t ->
9         match h with
10        |Left -> execute_program_aux t (move_left r)
11        |Right -> execute_program_aux t (move_right r)
12        |Write c ->
13          match t with
14          |Write c' :: _ -> execute_program_aux (List.tl t) (write c' r)
15          |_ -> execute_program_aux t (write c r)
16        |Repeat(n,li) ->
17          let rec repeat = fun n li r ->
18            if n = 0 then r
19            else repeat (n-1) li (execute_program_aux li r)
20          in execute_program_aux t (repeat n li r)
21        |Caesar n -> execute_program_aux t (caesar n r)
22        |Delete(c) -> execute_program_aux t (delete c r)
23        |Invert -> execute_program_aux t (invert r)
24    in execute_program_aux p {left = []; right = []}
25 ;;

```

```

Ⓢ (base) macbookpro@MBP-de-Richard ocaml % ocamlc last.ml
File "last.ml", line 255, characters 9-15:
255 |         |Repeat(n,li) ->
      |         ^^^^^
Error: This variant pattern is expected to have type instruction list
      There is no constructor Repeat within type list
Ⓢ (base) macbookpro@MBP-de-Richard ocaml %

```

Figure 5: message d'erreur

Parce que quand j'ai fait le match ci-dessus, à la ligne 14, il a cherché pour la prochaine instruction (`Write c' :: _ ->`), donc il faut que `execute_program` fasse le match de tous les autres instructions avant `Write`; par conséquent, j'ai mis `Write` en bas pour le program bien fait le match les autres instructions d'abord. (Voyiez la partie 3).

- `generate_program` : avant de réussir, j'ai commencé par :

```

1 let generate_program = fun l ->
2   let rec generate_program_aux = fun acc l ->
3     match l with
4     | [] -> acc
5     | h :: t -> generate_program_aux ([Write h; Right]@acc) t
6   in (generate_program_aux [] l)
7 ;;

```

Ça a marché, mais le résultat était au sens l'inverse; par exemple, si le contenu dans le fichier `.msg` est `abc`, elle va donner des instructions pour générer `cba` à la place, c'est-à-dire :

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

Donc, `List.rev` a été utilisé, j'ai fait l'inverse à la ligne 6 pour inverser les instructions. Mais, on peut voir qu'elle renvoie :

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

La première instruction `R` devrait être à la fin. Donc, l'astuce ici consiste à déplacer d'abord et après écrire. A la ligne 5, j'ai mis `([Right; Write h]@acc)`. Par conséquent ça donne les vraie instructions :

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

L'implémentation de `generate_program` marche bien mais ce n'est pas suffisant comme il y a l'instruction `Repeat`, donc j'ai fait deux fonctions : (c'est faux, la version corrigée est ci-dessus dans la partie 4)

```

1 let rec count_consecutive = fun l ->
2   match l with
3   | [] -> 0
4   | [h] -> 1
5   | h1 :: h2 :: t ->
6     if h1 = h2 then 1 + count_consecutive (h2 :: t)
7     else count_consecutive (h2 :: t)
8 ;;
9
10 let rec remove_element n lst =
11   if n <= 0 then lst
12   else match lst with
13     | [] -> []
14     | _ :: t1 -> remove_element (n-1) t1

```

```

15
16 let generate_program = fun l ->
17   let rec generate_program_aux = fun acc l ->
18     match l with
19     | [] -> acc
20     | h :: t ->
21       let count = 1 + count_consecutive l in
22       if count = 1 then generate_program_aux ([Right; Write h]@acc) t
23       else generate_program_aux ((Repeat(count,[Write h; Right]))::acc) (
24         remove_element count t)
25   in List.rev (generate_program_aux [] l)
26 ;;

```

Ici, la fonction `count_consecutive` compte le nombre d'occurrence d'une lettre dans la liste. Donc, ça ne marche pas dans ce cas parce qu'on veut compter des lettres consecutives pas l'occurrence. Donc, il faut supprimer le cas `else` et le remplacer par juste 1. En fait, la logique de cette fonction n'est pas vraiment vraie parce que si la liste ne commence pas par les lettres consecutives, autrement dit, la liste commence par deux éléments distincts, par exemple,

$$[1; 2; 3; 10; 10; 10] \rightarrow 1$$

La fonction va renvoyer 1 au lieu de 3 parce que le cas sinon a été modifié, c'est-à-dire, si le premier et la deuxième sont différents donc juste renvoie 1. Mais, dans la fonction `generate_program`, j'appelle `count_consecutive` juste seulement quand elle existe, autrement dit, il y aura des éléments consecutives au début de la liste.

De plus, à la ligne 23, `remove_element count t` était faux parce qu'il manquera un élément après des lettres consecutives. Le problème par exemple, si le contenu de `helloaaa.msg` est `helloaaa`, ça va donner les instructions sans `Write o` :

`generate_program helloaaa.msg`

`-> W(h);R;W(e);R;F(2,[W(1);R;]);F(3,[W(a);R;]);`

Parce que, après `he`, il y a 2 consecutives 1, donc

`remove_element 2 ['1'; 'o'; 'a'; 'a'; 'a'] -> ['a'; 'a'; 'a']`

Par conséquent, j'ai remplacé `t` par `l`, c'est-à-dire : `remove_element count l` (Voyez la partie 4), note que `l` ici n'est pas la liste originale, c'est la liste mise à jour après chaque écriture s'il n'y a pas des lettres consecutives. Donc dans ce cas :

`remove_element 2 ['1'; '1'; 'o'; 'a'; 'a'; 'a'] -> ['o'; 'a'; 'a'; 'a']`

Un autre problème était `exception hd` comme je l'ai utilisé pour comparer si le premier élément est même que la deuxième (à la ligne 7 dans partie 4). Donc, quand la liste devient vide ou il y a juste un élément dans la liste d'entrée, elle soulèvera une `exception hd`, parce qu'il n'y a pas des éléments pour comparer. Pour fixer, j'ai ajouté une condition à la ligne 6,

`if t = [] then ([Right ; Write h] @acc)`

S'il n'y a que un élément, juste l'écrire directement dans l'accumulateur.

Par ailleurs, dans Ocaml, par exemple,

`[1; 2; 3;]`

est même que

`[1; 2; 3]`

Donc avec ou sans `';` à la fin, il n'y a pas de différence comme le fichier `message_3aien.prog`, il n'y a pas de `';` à la fin `W(r)`. Donc, je n'ai besoin de prendre en compte ce cas là.

6 Quelques tests

Après avoir réussi pour les tests ont fourni, j'ai fait quelques tests supplémentaires pour s'assurer que le programme fonctionne bien.

- **caesar** : écrivons a b c, puis appliquon l'encodage de Caesar avec 1 déplacement 2 fois. Donc `abc` → `bcd` → **cde**

`W(a); R; W(b); R; W(c); C(1); C(1);` → *cde*

- **delete** : ajoutons `D(1)` à la fin de `hello.prog`, donc ça renvoie `hello` → **heo**

`W(a); W(h); R; W(e); R; F(2, [W(1); R]); W(o); D(1);` → *heo*

- **generate_program** : le programme a renvoyé les mêmes instructions que celles données dans `hello.prog`

`./a.out 3 hello.msg` →

`W(h); R; W(e); R; F(2, [W(1); R]); W(o); R;`

`./a.out 3 W(h); R; W(e); R; F(2, [W(1); R]); W(o); R;` → *hello*

Note que la liste dans `F` a un `';` supplémentaire à la fin à ce que dans `hello.prog`, c'est parce que la fonction `fprintf_program fmt l` donnée imprime `';` chaque fois. Donc, c'est pour ça il y a un `';` à la fin de liste instruction de `F`.

```
1 let fprintf_program fmt l =
2   List.iter (fun i -> Format.fprintf fmt "%a;" fprintf_instruction i) l
```

Mais, comme j'ai mentionné ci-dessus, avec ou sans `';` à la fin, c'est le même. De plus, il y a une instruction supplémentaire `R` à la fin, mais cela n'affectera pas le message.

`./a.out 3 message_3aien.msg` →

`W(T); R; W(e); R; W(r); R; W(r); R; W(i); R; W(e); R; W(n); R; W(s); R; W(); R; W(b); R; W(o); R; W(n); R; W(j); R; W(o); R; W(u); R; W(r); R; W(.); R; W(); R; W(V); R; W(o); R; W(u); R; W(s); R; W(); R; W(a); R; W(v); R; W(e); R; W(z); R; W(); R; W(e); R; W(n); R; W(f); R; W(i); R; W(n); R; W(); R; W(d); R; W(e); R; W(c); R; W(r); R; W(y); R; W(p); R; W(t); R; W(e); R; W(); R; W(1); R; W(e); R; W(); R; W(t); R; W(r); R; W(o); R; W(p); R; W(); R; W(s); R; W(i); R; W(m); R; W(p); R; W(1); R; W(e); R; W(); R; W(s); R; W(y); R; W(s); R; W(t); R; W(e); R; W(m); R; W(e); R; W(); R; W(2); R; W(A); R; W(i); R; W(e); R; W(n); R; W(.); R; W(); R; W(); R; W(V); R; W(o); R; W(i); R; W(c); R; W(i); R; W(); R; W(d); R; W(e); R; W(); R; W(n); R; W(o);`

```
R;W(u);R;W(v);R;W(e);R;W(l);R;W(l);R;W(e);R;W(s);R;W( );R;W(i);R;W(n);R;W(s);R;
W(t);R;W(r);R;W(u);R;W(c);R;W(t);R;W(i);R;W(o);R;W(n);R;W(s);R;W( );R;W(p);R;
W(o);R;W(u);R;W(r);R;W( );R;W(v);R;W(o);R;W(t);R;W(r);R;W(e);R;W( );R;W(p);R;
W(r);R;W(o);R;W(g);R;W(r);R;W(a);R;W(m);R;W(m);R;W(e);R;W( );R;W(:);R;W( );R;W(
);R;W( );R;W(-);R;W( );R;W(C);R;W( );R;W(n);R;W( );R;W( );R;W(:);R;W( );R;W(a);
R;W(p);R;W(p);R;W(l);R;W(i);R;W(q);R;W(u);R;W(e);R;W( );R;W(u);R;W(n);R;W( );R;
W(e);R;W(n);R;W(c);R;W(o);R;W(d);R;W(a);R;W(g);R;W(e);R;W( );R;W(d);R;W(e);R;
W( );R;W(C);R;W(e);R;W(s);R;W(a);R;W(r);R;W( );R;W(d);R;W(e);R;W( );R;W(p);R;
W(a);R;W(s);R;W( );R;W(n);R;W( );R;W(a);R;W(u);R;W( );R;W(m);R;W(e);R;W(s);R;
W(s);R;W(a);R;W(g);R;W(e);R;W( );R;W(d);R;W(e);R;W(j);R;W(a);R;W( );R;W(d);R;
W(e);R;W(c);R;W(r);R;W(y);R;W(p);R;W(t);R;W(e);R;W( );R;W(
);R;W( );R;W(-);R;W( );R;W(D);R;W( );R;W(a);R;W( );R;W( );R;W(:);R;W( );R;W(e);
R;W(f);R;W(f);R;W(a);R;W(c);R;W(e);R;W( );R;W(l);R;W(e);R;W( );R;W(c);R;W(a);
R;W(r);R;W(a);R;W(t);R;W(e);R;W(r);R;W(e);R;W( );R;W(a);R;W( );R;W(d);R;W(a);
R;W(n);R;W(s);R;W( );R;W(l);R;W(e);R;W( );R;W(m);R;W(e);R;W(s);R;W(s);R;W(a);
R;W(g);R;W(e);R;W( );R;W(d);R;W(e);R;W(j);R;W(a);R;W( );R;W(d);R;W(e);R;W(c);
R;W(r);R;W(y);R;W(p);R;W(t);R;W(e);R;W(
);R;W( );R;W(-);R;W( );R;W(I);R;W( );R;W(:);R;W( );R;W(i);R;W(n);R;W(v);R;
W(e);R;W(r);R;W(s);R;W(e);R;W( );R;W(l);R;W(e);R;W( );R;W(c);R;W(o);R;W(n);R;
W(t);R;W(e);R;W(n);R;W(u);R;W( );R;W(a);R;W(c);R;W(t);R;W(u);R;W(e);R;W(l);R;
W( );R;W(d);R;W(u);R;W( );R;W(r);R;W(u);R;W(b);R;W(a);R;W(n);R;W( );R;W(e);R;
W(t);R;W( );R;W(r);R;W(e);R;W(p);R;W(l);R;W(a);R;W(c);R;W(e);R;W( );R;W(l);R;
W(e);R;W( );R;W(c);R;W(u);R;W(r);R;W(s);R;W(e);R;W(u);R;W(r);R;
```

./a.out 3 repeat.msg →

```
W(h);R;W(e);R;F(4, [W(l);R;]);W(o);R;W( );R;W(b);R;W(o);R;W(n);R;W(j);R;
F(3, [W(o);R;]);W(u);R;F(2, [W(r);R;]);W( );R;W(c);R;W(');R;W(e);R;W(s);R;
W(t);R;W( );R;W(m);R;W(o);R;F(3, [W(i);R;]);W( );R;W(R);R;W(i);R;W(c);R;
W(h);R;W(a);R;W(r);R;W(d);R;
```

./a.out 2 ces_instructions → hellllo bonjooourr c'est moi iii Richard

7 Limites du programme

- **Repeat Write sans déplacement** : par exemple, `F(3, [W(a)])`, dans ce cas là, il faut imprimer juste un seul 'a' parce que j'ai écrit 3 fois sans déplacer le curseur. Mais, ce programme a donné 'aaa' au lieu car dans la fonction `exectue_program`, j'ai mis le match pour **Write** en bas, donc si je veux faire le match dans **Repeat** pour voir s'il n'y a qu'un seul W dans F, logiquement ça marche, mais le programme n'a pas encore sû ce qu'est W car il est en bas. Donc c'est un peu compliqué pour traiter ce cas. De plus, et cela n'a aucun sens pour faire ça parce que si on veut écrire un 'a', alors juste utiliser `W(a)` au lieu de `F(3, [W(a)])`.