IPF - Rapport de projet.

Théo Lavigne

Table des matières

L'ADN pour acide désoxyribonucléique, est une macro molécule qui encode l'information génétique chez la majorité des êtres vivants. Semblable à programme informatique codé en binaire, l'ADN code l'information génétique à l'aide de 4 nucléotides:l'adénine (A), la thymine (T), la cytosine (C) et la guanine (G). On cherche alors à représenter ici des gênes, c'est à dire des séquences relativement longues de ces nucléotides de manière et ce de manière efficace.

Les propositions de réponse aux questions sont présentées ci-dessous. On associe, à chaque explication, un ou plusieurs tests des fonctions considérées et implémentées pour une question donnée. Chaque test présent dans ce rapport est *"fonctionnel"* et est aussi présent à la suite du code dans `main.ml`. Finalement, afin d'avoir des tests effectifs, un `string_builder , sb_test` est créé spécifiquement à cet effet.

1 Question 1

1.1 Question 1.1

On commence tout d'abord par déclarer le type `string_builder` qui sera utilisé tout au long du projet. C'est un arbre binaire, les longueurs des mots y sont stockées.

```
type string_builder =
  | Leaf of string * int
  | Node of string_builder * int * string_builder;;
```

1.2 Question 1.2

On utilise la méthode `String.length` pour avoir la longueur de la chaîne de caractères donnée en argument. De plus le constructeur `Leaf` permet de répondre à la question.

Test :

```
let () = assert (word "Camille" = Leaf ("Camille", 7))
```

1.3 Question 1.3

On commence par implémenter une fonction auxiliaire :

```
let word_length sb = match sb with
  |Leaf (_,i) -> i
  |Node (_,i,_) -> i;;
```

Cette fonction renvoie la longueur du mot représenté par un noeud ou bien une feuille.
Ce faisant, on peut écrire la fonction `concat` en utilisant le constructeur `Node` ainsi que la somme des longueurs des mots représentés par chaque `string_builder` donnés en argument.

Tests :

```
let () = assert( word_length sb_test = 18)
let () = assert( concat (Node(Leaf("I",1),3,Leaf("am",2)))
                        (Node(Leaf("a",1),15,Leaf("string_builder",14)))
              = sb_test)
```

2 Question 2

On implémente tout d'abord une fonction `mot:sting_builder -> string` qui renvoie le mot représenté par `string_builder` sous forme de chaîne de caractères (de manière récursive). On appelle ensuite cette

fonction en argument de la méthode `String.get` pour répondre à la question.

Tests :

```
let () = assert( mot sb_test = "Iamastring_builder")

let () = assert( char_at 2 sb_test = 'm')
```

3 Question 3

On veut récupérer le `string_builder` représentant les caractères i à $i + m - 1$, sous la forme d'un `string_builder` . Pour cela on *match* sur la structure du `string_builder` donnée en argument.

3.1 Cas de la feuille

On se sert de la méthode `String.substring:int->int->string` dans le cas de base pour récupérer la chaîne de caractères souhaitée.

3.2 Cas du noeud

On s'intéresse tout d'abord aux noeuds ayant comme fils une feuille à gauche et un noeud à droite. En considérant la longueur du mot contenu par la feuille (et en étant précis sur l'utilisation des indices) on renvoie le `string_builder` souhaité grâce aux constructeurs.

On s'intéresse ensuite au cas du noeud à gauche et on procède de manière similaire. La méthode `String.substring` est ici à nouveau utilisée.

4 Question 4

On implémente tout d'abord une fonction auxiliaire `aux1:string_builder -> int -> int` avec comme accumulateur `depht`. A chaque noeud, l'accumulateur `depth` est incrémenté de `1`.

```
let rec aux1 sb depht = match sb with
  |Leaf (_, i) -> i * depht
  |Node (left, _, right) -> aux1 left (depht + 1) + aux1 right (depht + 1);;
```

On applique alors cette fonction avec une profondeur nulle passée en paramètre pour répondre à la question.

Test :

```
let () = assert ( cost sb_test = 36)
```

5 Question 5

On implémente tout d'abord la fonction sans argument `random_letter :string` qui renvoie une lettre au hasard parmi les suivantes: *A, T, C, G*. Étant donné le contexte, on choisit uniquement de représenter les quatre bases nucléiques. Par ailleurs, la fonction `rdm_string:int -> string` renvoie une chaîne de caractères aléatoire composée des nucléotides et de longueur passée en argument. Finalement, en utilisant la fonction `word:string -> string_builder` et les deux fonctions précédentes, on crée récursivement un arbre binaire de profondeur `i`.

J'ai choisi arbitrairement 10 comme étant la longueur maximale des chaînes de caractères stockées au sein des feuilles. On peut concevoir que cette longueur puisse être en réalité bien plus grande. Quant à l'algorithme, pour un arbre de profondeur i il y a i appels récursifs. Dans le cas $i = 0$ on utilise la fonction

word pour créer une feuille, sinon on fait deux appels récursifs pour créer un noeud. La complexité est alors linéaire en la taille de l'arbre ce qui est acceptable.

```
let rec random_string i = if i = 0 then word (rdm_string 10)
                          else let sb1 = random_string (i-1) in
                            let sb2 = random_string (i-1) in
                              concat sb1 sb2;;
```

Tests :

```
print_tree (random_string 3)
Format.printf "%s" (random_letter())
Format.printf "%s" (rdm_string 5)
```

6 Question 6

Il s'agit ici d'implémenter tout d'abord une fonction auxiliaire ayant pour accumulateur une liste qui sera considérée vide par la suite (en argument au sein de la fonction principale). Pour le cas de la feuille, on *cons* l'élément stocké par la feuille à l'accumulateur. Quant au noeud, on appelle récursivement la fonction à droite puis à gauche avec la liste obtenue à droite.

```
let rec aux sb acc = match sb with
  |Leaf (a,_) -> a::acc
  |Node (left,_,right) -> let acc2 = aux right acc in
                            aux left acc2;;

let list_of_string sb = aux sb [];;
```

Tests :

```
let () = assert (list_of_string sb_test = ["I";"am";"a";"string_builder"])
```

7 Question 7

Diviser pour mieux régner

A l'aide d'un `map` appliqué à la liste renvoyée par la fonction `list_of_string:string_builder -> string list` et de la fonction `word` on écrit une fonction `leaf_list:string_builder -> string_builder list` qui pour un `string_builder` donné renvoie la liste de ses feuilles.

On implémente ensuite une fonction: `moy_cost:string_builder -> string_builder -> float` qui calcule le coût moyen de deux feuilles.

Il s'agit désormais de trouver la position du couple ayant la plus grande moyenne. On cherche la position du premier élément. Pour cela, on implémente la fonction `couple_max:string_builder list -> float -> int -> int -> int`. Le principe de la fonction repose sur un *match* de la `list` donnée en paramètre. L'accumulateur `acc` stocke la moyenne du couple considéré tandis que l'accumulateur `position` stocke la position du couple actuel, qui est attribuée à `res` si la moyenne du couple considérée est supérieure à l'ensemble des moyennes précédentes.

```
(*It is supposed that |list|>=2 *)
let rec couple_max list acc position res = match list with
  |x::(y::q as queue) -> let moy = moy_cost x y in
                         if moy > acc then couple_max queue moy (position+1) position
                         else couple_max queue acc (position+1) res
  |_ -> res
```

On récapitule. Nous avons actuellement à disposition une fonction qui nous renvoie la position du couple de feuilles ayant le coût le plus élevé. Par suite, on réalise une fonction `placement:string_builder list -> int -> string_builder list`. Cette fonction concatène et place, à une position donnée (`int`) le couple de feuille dont le coût est maximal. On utilise *de facto* la fonction précédente.

L'usage d'une dernière fonction auxiliaire, `aux2:string_builder list -> string_builder` qui utilise, a bon escient, les deux fonctions précédentes (et permet surtout de ne calculer `leaf_list sb` qu'une unique fois). Enfin, on appelle la fonction balance.

```
let rec aux2 list = match list with
  |[] -> failwith "aux2 : list shouldn't be empty"
  |[x] -> x
  |x::y::_ -> let pos = couple_max list (moy_cost x y) 0 0 in
                aux2 (placement list pos)

let balance sb = aux2 (leaf_list sb)
```

Tests :

```
let () = assert (leaf_list sb_test =
        [Leaf("I",1);Leaf("am",2);Leaf("a",1);Leaf("string_builder",14)] )
let () = assert (moy_cost (Leaf ("oui",3)) (Leaf ("ok", 2)) = 2.5)
let () = assert (couple_max (leaf_list sb_test) 0. 0 0= 2)
let _ = print_tree (balance sb_test)
```

8 Question 8

On implémente facilement une fonction `let gain sb = cost sb - (cost (balance sb))`. De même, une fonction `gain_list:int -> int list -> int list` qui renvoie une liste de gains correspondants à `n` arbres générés aléatoirement à l'aide de la question 5. La méthode `List.fold_left` permet d'implémenter le maximum (respectivement le minimum) d'une liste en utilisant la constante `min_int` (respectivement `max_int`).

```
let max list = List.fold_left max min_int list
let min list = List.fold_left min max_int list
```

On crée de même une fonction `average:int list -> int` qui calcule la moyenne d'une liste d'entier. On applique ces trois dernières fonctions à `stat_list = gain_list 10 []` et on obtient lors des tests des résultats très satisfaisant quant aux coût des arbres balancés.

Test :

```
let _ = Format.printf "max = %d; average = %f ; min = %d ;\n" max_gain average_gain
↪  min_gain
```