Projet IPF

BOUCHET ULYSSE

Table des matières

Introduction

On représente un brin d'ADN comme une liste de nucléotides. Les quatre nucléotides de l'ADN sont l'adénine, la cytosine, la guanine et la thymine. On va les noter comme A, C, G et, respectivement, T.

L'ADN est constituée de deux brins qui sont liés de manière complémentaires deux à deux, c'est à dire que : l'adénine ne peut se lier qu'à la thymine et la cytosine ne peut se lier qu'à la guanine.

Comme demandé, on a donc créé les types OCAML suivants :

```
type nucleotide = A | C | G | T;;
type brin = nucleotide list;;
```

Des triplets de nucléotides, aussi appelés codons, peuvent encoder des acides aminés.

Les types OCAML suivants ont donc été créés :

```
type acide =    Ala | Arg | Asn | Asp | Cys | Glu | Gln | Gly | His | Ile |
                Leu | Lys | Phe | Pro | Ser | Thr | Trp | Tyr | Val |
                START | STOP;;
```

```
type chaine = acide list;;
```

Un brin encode des chaînes consécutives d'acides aminés, chacune étant délimitée par un acide aminé START et un STOP.

On dit que le brin est bien formé s'il correspond bien à des chaînes consécutives, dont chacune commence par START et se termine par STOP.

Un arbre phylogénétique est un arbre binaire complet. Dans cet arbre, chaque nœud qui est une feuille contient un brin et chaque nœud qui n'est pas une feuille contient un couple formé par un brin, ainsi qu'une valeur, le malus, qui correspond à la somme entre les distances d'édition entre le nœud et chacun de ses deux enfants et celle de leurs malus respectifs.

Attention : tous les brins d'un arbre ont la même longueur.

Choix effectués

Types

nucleotide
Pour le type nucleotide, j'ai décidé de simplement reprendre celui du sujet.

brin
Il en va de même pour le type brin, j'ai simplement repris celui du sujet.

acide
Encore une fois, j'ai repris le type du sujet.

chaine
J'ai décidé de créer le type chaîne, qui représente une chaîne d'acide aminés, donc une liste :

```
type chaine = acide list;;
```

arbre_phylo
Une fois n'est pas coutume, j'ai repris le type du sujet.

Fonctions

contenu_gc
Pour la fonction contenu_gc (question 1), j'ai décidé de lancer une erreur lorsque le brin passé en paramètre était vide :

```
failwith "erreur : brin vide"
```

En effet, il n'a pas de sens selon moi à calculer le taux de nucléotides GC d'un brin qui ne possède aucun nucléotide.

nucleotide_complementaire
Pour la fonction brin_complementaire (question 2), j'ai décidé d'utiliser une fonction auxiliaire :

```
let nucleotide_complementaire (nuc: nucleotide): nucleotide =
    match nuc with
        | T -> A
        | A -> T
        | G -> C
        | C -> G
    ;;
```

Cette fonction permet d'obtenir facilement le nucléotide complémentaire d'un nucléotide donné.

distance

J'ai décidé de faire appel à une fonction auxiliaire récursive interne (distance_rec) pour calculer la distance :

```
let distance (b1: brin) (b2: brin): int =
    let rec distance_rec (b1: brin) (b2: brin) (cpt: int): int =
        match (b1, b2) with
        | [], []                -> cpt
        | [], _                 -> failwith "[distance_rec] erreur : brins de taille différente"
        | _, []                 -> failwith "[distance_rec] erreur : brins de taille différente"
        | (h1::t1), (h2::t2)    -> if h1 <> h2 then distance_rec t1 t2 (cpt + 1) else distance_rec t1 t2 cpt
    in
        distance_rec b1 b2 0
    ;;
```

similarite

Je n'ai pas effectué de choix particulier pour cette fonction.

codon_vers_acide

J'ai simplement repris le code fourni par le sujet.

brin_vers_chaine

J'ai choisi de faire appel à une fonction auxiliaire interne pour convertir un brin en chaîne d'acides aminés :

```
let brin_vers_chaine (b: brin): chaine =
    let rec brin_vers_chaine_rec (b: brin) (chaine: chaine) (start: bool): chaine =
        match b with
            | []            -> chaine
            | _::[]         -> failwith "[brin_vers_chaine] erreur : brin invalide"
            | _::_::[]      -> failwith "[brin_vers_chaine 188] erreur : brin invalide"
            | n1::n2::n3::q ->
                let acide =
                    codon_vers_acide n1 n2 n3
                in
                    match (q, start, acide) with
                    | (_, true, STOP)       -> chaine (* On arrive au bout du brin *)
                    | ([], _, _)            -> failwith "[brin_vers_chaine] erreur : brin invalide" (* On arrive au bout du brin sans STOP => brin invalide *)
                    | (_, false, START)     -> brin_vers_chaine_rec q [] true (* On commence la chaîne *)
                    | (_, true, START)      -> failwith "[brin_vers_chaine] erreur : brin invalide" (* On a déjà commencé la chaîne => brin invalide *)
                    | (_, false, _)         -> failwith "[brin_vers_chaine] erreur : brin invalide" (* On a un acide sans avoir commencé de chaîne => brin invalide *)
                    | (_, true, _)          -> brin_vers_chaine_rec q (chaine@[acide]) true (* On passe au codon suivant *)
    in
        brin_vers_chaine_rec b [] false
    ;;
```

brin_vers_chaines

De même, j'ai choisi de faire appel à une fonction auxiliaire interne :

```
let brin_vers_chaines (b: brin): chaine list =
    let rec brin_vers_chaines_rec (b: brin) (chaine: chaine) (chaines: chaine list) (start: bool): chaine list =
        match b with
            | []            -> chaines
            | _::[]         -> failwith "[brin_vers_chaines] erreur : brin invalide"
            | _::_::[]      -> failwith "[brin_vers_chaines] erreur : brin invalide"
            | n1::n2::n3::q ->
                let acide =
                    codon_vers_acide n1 n2 n3
                in
                    match (q, start, acide) with
                    | ([], true, STOP)      -> chaines@[chaine] (* On arrive au bout du brin *)
                    | ([], _, _)            -> failwith "[brin_vers_chaines] erreur : brin invalide" (* On arrive au bout du brin sans STOP => brin invalide *)
                    | (_, false, START)     -> brin_vers_chaines_rec q [] chaines true (* On commence une nouvelle chaîne *)
                    | (_, true, START)      -> failwith "[brin_vers_chaines] erreur : brin invalide" (* On a déjà commencé une chaîne => brin invalide *)
                    | (_, true, STOP)       -> brin_vers_chaines_rec q [] (chaines@[chaine]) false (* On passe à la chaîne suivante *)
                    | (_, false, _)         -> failwith "[brin_vers_chaines] erreur : brin invalide" (* On a un acide sans avoir commencé de chaîne => brin invalide *)
                    | (_, true, _)          -> brin_vers_chaines_rec q (chaine@[acide]) chaines true (* On passe au codon suivant *)
    in
        brin_vers_chaines_rec b [] [] false
    ;;
```

brin_vers_string

Pour la fonction suivante, j'ai décidé de créer une fonction brin_vers_string pour convertir un brin en chaîne de caractères, en utilisant un simple fold_left :

```
let brin_vers_string (brin: brin) =
    List.fold_left (fun str nucl -> str ^ match nucl with |A->"A"|T->"T"|C->"C"|G->"G") "" brin
    ;;
```

arbre_phylo_vers_string

Pour cette fonction, j'ai décidé de faire appel à la fonction auxiliaire ci-dessus.

```
let rec arbre_phylo_vers_string (arbre: arbre_phylo): string =
    match arbre with
    | Lf (brin) ->
        "(" ^ brin_vers_string brin ^ ")"
    | Br (arbre_gauche, brin, malus, arbre_droit) ->
        "{" ^ arbre_phylo_vers_string arbre_gauche ^ "}" ^
        " <-- (" ^ brin_vers_string brin ^ " " ^ string_of_int malus ^ ") --> " ^
        "{" ^ arbre_phylo_vers_string arbre_droit ^ "}"
    ;;
```

J'ai choisi de représenter une arbre phylogénétique sous la forme suivante :

```
{{(GCAT)} <-- (ACAT 3) --> {(TCGT)}} <-- (AAAA 8) --> {{(TAGA)} <-- (AAGA 2) --> {(GAGA)}}
```

similarite_arbre

Je n'ai pas fait de choix particulier pour cette fonction.

similaire

J'ai choisi de faire appel à une fonction auxiliaire récursive interne :

```
let similaire (arbre: arbre_phylo) (arbres: arbre_phylo list): arbre_phylo =
    let rec aux restants plus_grand plus_grand_simi =
        match restants with
        | []    -> plus_grand
        | t::q  -> let simi = similarite_arbre arbre t in if simi < plus_grand_simi then aux q plus_grand plus_grand_simi else aux q t simi
    in
        match arbres with
        | []    -> failwith "[similaire] erreur : liste vite"
        | t::q  -> aux q t (similarite_arbre arbre t)
    ;;
```

get_root

Je n'ai pas réalisé de choix particulier pour cette fonction.

get_malus

Pas de choix particulier.

br

Encore une fois, pas de choix effectué.

gen_phylo

J'ai décidé de coder « en dur » les différents arbres possibles, cela me semble être le choix le plus simple et efficace :

```
let gen_phylo (b1: brin) (b2: brin) (b3: brin): arbre_phylo list =
    [
        br (Lf (b1)) b2 (Lf (b3));
        br (Lf (b1)) b3 (Lf (b2));
        br (Lf (b2)) b1 (Lf (b3));
        br (Lf (b2)) b3 (Lf (b1));
        br (Lf (b3)) b1 (Lf (b2));
        br (Lf (b3)) b2 (Lf (b1))
    ]
    ;;
```

min_malus

J'ai décidé de faire, encore une fois, appel à une fonction récursive interne :

```
let min_malus (arbres: arbre_phylo list): arbre_phylo =
    let rec min_malus_rec (arbres: arbre_phylo list) (min_malus: int) (min_arbre: arbre_phylo): arbre_phylo =
        match arbres with
        | []    -> min_arbre
        | t::q  ->
            let cur_malus =
                get_malus t
            in
                min_malus_rec q (if cur_malus < min_malus then cur_malus else min_malus) (if cur_malus < min_malus then t else min_arbre)
    in
        match arbres with
        | []    -> failwith "[min_malus] erreur : liste vide"
        | t::q  -> min_malus_rec q (get_malus t) t
    ;;
```

gen_min_malus_phylo

Ne sachant pas comment traiter une liste de taille n, j'ai décidé de ne traiter que les cas où n=1 ou n=3, les autres renvoyant une exception :

```
let min_malus (arbres: arbre_phylo list): arbre_phylo =
    let rec min_malus_rec (arbres: arbre_phylo list) (min_malus: int) (min_arbre: arbre_phylo): arbre_phylo =
        match arbres with
        | []    -> min_arbre
        | t::q  ->
            let cur_malus =
                get_malus t
            in
                min_malus_rec q (if cur_malus < min_malus then cur_malus else min_malus) (if cur_malus < min_malus then t else min_arbre)
    in
        match arbres with
        | []    -> failwith "[min_malus] erreur : liste vide"
        | t::q  -> min_malus_rec q (get_malus t) t
    ;;
```

Tests et résultats

Assertions

contenu_gc

Pour cette fonction, j'ai décidé de reprendre les exemples du poly afin d'en faire des assertions (validées) :

```
let () = assert (contenu_gc [A;T;G;T;T;G;A;C] = 0.375);;
let () = assert (contenu_gc [C;T;T;A] = 0.25);;
let () = assert (contenu_gc [A;A;A;T;A] = 0.);;
```

Il est également bon de noter qu'appeler contenu_gc avec un brin vide provoquerait une erreur (testé).

nucleotide_complementaire

Cette fonction n'existant pas dans le poly, j'ai moi-même décidé des assertions que j'allais effectuer. J'ai donc simplement testé la valeur de retour pour chaque type de nucléotide existant (assertions validées) :

```
let () = assert (nucleotide_complementaire T = A);;
let () = assert (nucleotide_complementaire A = T);;
let () = assert (nucleotide_complementaire C = G);;
let () = assert (nucleotide_complementaire G = C);;
```

brin_complementaire

Cette fonction existant dans le sujet, j'ai donc repris les exemples du poly. J'ai également testé qu'un brin vide possédait un autre brin vide comme brin complémentaire. Assertions (validées) :

```
let () = assert (brin_complementaire [T] = [A]);;
let () = assert (brin_complementaire [C;T;T;C] = [G;A;A;G]);;
let () = assert (brin_complementaire [C;T;A;A;T;G;T] = [G;A;T;T;A;C;A]);;
let () = assert (brin_complementaire [] = []);;
```

distance

Encore une fois, cette fonction existait dans le sujet ainsi j'ai repris les exemples du poly. J'ai aussi testé avec deux brins vides.

Assertions (validées) :

```
let () = assert (distance [T] [T] = 0);;
let () = assert (distance [T] [C] = 1);;
let () = assert (distance [G;A;G] [A;G;G] = 2);;
let () = assert (distance [] [] = 0);;
```

Il faut aussi savoir que lorsque les brins sont de longueurs différentes, une exception est levée.

similarite

J'ai choisi de reprendre les exemples du sujet, en plus d'un test avec deux listes vides :

```
let () = assert (similarite [C;G;A;T] [T;A;G;T] = 0.25);;
let () = assert (similarite [A;G;C;T] [T;A;A;G] = 0.);;
let () = assert (similarite [A;G;C;T] [A;G;C;T] = 1.);;
let () = assert (similarite [] [] = 1.);;
```

Il faut noter qu'appeler la fonction avec des listes de taille différentes provoquerait une exception.

brin_vers_chaine

J'ai encore décidé de reprendre l'exemple du sujet en plus d'un test avec un brin vide.

```
let () = assert (brin_vers_chaine [T;A;C;G;G;C;T;A;G;A;T;T;T;A;C;G;C;T;A;A;T;A;T;C] = [Pro;Ile]);;
let () = assert (brin_vers_chaine [] = []);;
```

Les brins ne respectant pas les contraintes décrites dans le sujet lancent une exception.

brin_vers_chaines

Pareil que pour la fonction précédente :

```
let () = assert (brin_vers_chaines [T;A;C;G;G;C;T;A;G;A;T;T;T;A;C;G;C;T;A;A;T;A;T;C] = [[Pro;Ile]; [Arg;Leu]]);;
let () = assert (brin_vers_chaines [] = []);;
```

brin_vers_string

Je n'ai pas testé cette fonction directement, voir arbre_phylo_vers_string.

arbre_phylo_vers_string

J'ai d'abord défini quelques arbres, dont ceux du poly :

```
let arbre_1 = Br (Br (Lf ([G;C;A;T]), [A;C;A;T], 3, Lf ([T;C;G;T])), [A;A;A;A], 8, Br (Lf ([T;A;G;A]), [A;A;G;A], 2, Lf ([G;A;G;A])));;
let arbre_2 = Br (Br (Lf ([G;A;A;T]), [G;C;T;T], 5, Lf ([C;A;G;T])), [A;A;T;A], 12, Br (Lf ([T;A;G;A]), [A;A;G;A], 3, Lf ([G;T;G;A])));;
let arbre_3 = Lf ([]);;
let arbre_4 = Br (Br (Lf ([G;A;G;T]), [G;C;T;T], 5, Lf ([C;A;G;T])), [A;A;T;A], 12, Br (Lf ([T;A;G;A]), [A;A;G;A], 3, Lf ([G;T;G;A])));;
```

Puis j'ai procédé aux tests :

```
let () = assert (arbre_phylo_vers_string arbre_1 =
                 "{{(GCAT)} <-- (ACAT 3) --> {(TCGT)}} <-- (AAAA 8) --> {{(TAGA)} <-- (AAGA 2) --> {(GAGA)}}");;
let () = assert (arbre_phylo_vers_string arbre_2 =
                 "{{(GAAT)} <-- (GCTT 5) --> {(CAGT)}} <-- (AATA 12) --> {{(TAGA)} <-- (AAGA 3) --> {(GTGA)}}");;
let () = assert (arbre_phylo_vers_string arbre_3 =
                 "()");;
```

similarite_arbre

Je n'ai pas testé cette fonction directement, voir similaire.

similaire

J'ai testé avec quelques arbres du poly le comportement de similaire :

```
let () = assert (similaire arbre_1 [arbre_4; arbre_2] = arbre_2);;
let () = assert (similaire arbre_1 [arbre_1; arbre_2; arbre_4] = arbre_1);;
```

get_root, get_malus

J'ai testé ces fonctions avec les arbres du poly :

```
let () = assert (get_root arbre_1 = [A;A;A;A]);;
let () = assert (get_root arbre_2 = [A;A;T;A]);;
let () = assert (get_root arbre_3 = []);;
```

```
let () = assert (get_malus arbre_1 = 8);;
let () = assert (get_malus arbre_2 = 12);;
let () = assert (get_malus arbre_3 = 0);;
```

br

J'ai testé cette fonction avec un arbre créé par moi-même :

```
let br_res = Br (Br (Lf ([A;A;T;T]), [T;T;T;T], 4, Lf ([T;T;A;A])), [A;A;A;A], 14, Br (Lf ([A;A;T;T]), [T;A;A;T], 4, Lf ([T;T;A;A])));;
let br_brin = [A;A;A;A];;
let br_gauche = Br (Lf ([A;A;T;T]), [T;T;T;T], 4, Lf ([T;T;A;A]));;
let br_droit = Br (Lf ([A;A;T;T]), [T;A;A;T], 4, Lf ([T;T;A;A]));;

let () = assert (br br_gauche br_brin br_droit = br_res);;
```

gen_phylo

Pareil que pour la fonction précédente, j'ai créé mes propres exemples :

```
let b1 = [A;A;A;A];;
let b2 = [T;T;T;T];;
let b3 = [C;C;C;C];;
let a1 = Br (Lf (b1), b2, distance b2 b1 + distance b2 b3, Lf (b3));;
let a2 = Br (Lf (b1), b3, distance b3 b1 + distance b3 b2, Lf (b2));;
let a3 = Br (Lf (b2), b1, distance b1 b2 + distance b1 b3, Lf (b3));;
let a4 = Br (Lf (b2), b3, distance b3 b2 + distance b3 b1, Lf (b1));;
let a5 = Br (Lf (b3), b1, distance b1 b3 + distance b1 b2, Lf (b2));;
let a6 = Br (Lf (b3), b2, distance b2 b3 + distance b2 b1, Lf (b1));;
let a_res = [a1; a2; a3; a4; a5; a6];;

let () = assert (gen_phylo b1 b2 b3 = a_res);;
```

min_malus

J'ai procédé au test présenté dans le poly :

```
let () = assert (min_malus [arbre_1; arbre_2] = arbre_1);;
```

gen_min_malus_phylo

J'ai testé avec un exemple que j'ai créé :

```
let () = assert (gen_min_malus_phylo [b1;b2;b3] = Br (Lf [A; A; A; A], [T; T; T; T], 8, Lf [C; C; C; C]));;
```