Rapport projet IPFL

Data Saiyentist

26/04/2021

I/ Fonctionnement général du projet

A- Constitution du projet

Le projet contient plusieurs fichiers `.ml` et un `Makefile`.

- le fichier `squelette.ml` contient tous les types construits pour ce projet dont
  - `nucleotide` pour les nucléotides `A, C, G, T`
  - `brin` pour une liste de nucléotides
  - `acide` pour les acides aminés
  - `arbre_phylo` pour les arbres phylogénétiques
  - une fonction `codon_vers_acide` qui retourne l'acide aminé encodé par le triplet de nucléotides
- le fichier `source.ml` contient toutes les fonctions codées
- le fichier `test.ml` contient tous les tests réalisés pour chacune des fonctions
- le fichier `Makefile` permet de compiler l'ensemble du projet

```
type arbre_phylo = Lf of brin | Br of arbre_phylo * brin * int * arbre_phylo;;
```

> Remarque : Je ne prends pas en compte l'existence de l'ARN (constitué des nucléotides `A, C, G, U`). En réalité, les acides aminés sont encodés par des triplets de nucléotides de l'ARN et non de l'ADN. Mais ça ne pose pas de problème puisqu'il suffit de remplacer les `U` par les `T` lors de l'encodage.

B- Manuel d'utilisation

Le fichier `test.ml` permet d'afficher les résultats des tests sur l'écran.

Pour compiler le projet, vous aurez besoin d'installer OCaml sur votre machine et la commande `make` (si vous ne les avez pas, sous Linux, entrez `sudo apt install ocaml` et `sudo apt install make`).

Placez vous dans le répertoire où se trouvent les fichiers associés au projet.

Entrez `make prog` pour seulement compiler les fonctions ou `make test` pour compiler l'ensemble du projet y compris les tests.

Puis entrez `./prog` si vous aviez tapé la 1ère commande sinon `./test` si vous aviez tapé la 2nde commande.

> Certaines fonctions lèvent des exceptions `failwith` suite à des tests ne respectant pas certaines conditions. Néanmoins, lorsqu'une exception est levée, l'exécution s'arrête. C'est pourquoi, j'ai mis en commentaire les appels de fonctions levant des exceptions. Veuillez donc les décommenter pour vérifier si les exceptions sont bien levées.

II/ Démarche générale

Séquences ADN/ ARN

Question 1 `contenu_gc : brin -> float` retourne la proportion de `G` ou de `C` du brin passé en argument.

Pour cela, j'utilise simplement un `List.fold_left` qui permet d'appliquer une fonction à tous les éléments de la liste en partant de la gauche.

Pour éviter l'utilisation d'un `List.length`, je considère un couple comme accumulateur. Le premier terme du couple compte le nombre de `G` ou de `C` dans le brin. Le second terme compte le nombre de nucléotides dans le brin.

Ainsi, si le second terme de l'accumulateur est non nul, je retourne leur quotient (je pense à convertir les entiers en flottants, avec `float_of_int` avant de réaliser la division pour flottant). Sinon, je lève une exception, car le brin est vide.

Cette fonction est en $\mathcal{O}(n)$, avec n la taille du brin donné en argument.

Question 2 `brin_complementaire : brin -> brin` retourne le brin complémentaire du brin passé en argument.

Je fais comme précédemment en utilisant un folder. Cette fois, j'utilise un `List.fold_right` pour éviter l'utilisation de `List.rev`.

En partant de la droite, j'ajoute le complémentaire du nucléotide dans l'accumulateur pour ensuite le retourner.

Cette fonction est aussi en $\mathcal{O}(n)$, avec n la taille du brin donné en argument.

Question 3 `distance : brin -> brin -> int` retourne la distance d'édition, d_e, entre deux brins de même longueur donnés en arguments.

J'utilise une fonction récursive pour calculer d_e en comptant les nucléotides qui doivent être changés pour rendre les brins égaux . Cette fonction inclut deux arguments (pour éviter l'utilisation de `List.length`) :

- `bool` un booléen de type `int` qui vaut 1 s'il y a au moins un calcul de distance entre deux nucléotides, 0 sinon
- `acc` un accumulateur qui est retourné à la fin de l'appel de `distance` et qui correspond à d_e

Cette fonction est en $\mathcal{O}(n)$, avec $n = min\{n_1, n_2\}$ avec n_1 la taille du 1er brin et n_2 la taille du 2nd brin donnés en argument. En effet (les cas qui suivent assurent la terminaison):

- Si `bool=0` et `brin1=[]`, alors `brin1` était vide lors de l'appel de `distance`. Donc, je lève un `failwith`
- Si `bool=1`, `brin1=[]` et `brin2=[]`, alors `brin1` et `brin2` ont la même taille et `acc` est la distance d'édition
- Si `bool=1`, `brin1=[]` et `brin2!=[]`, ou `brin1!=[]` et `brin2=[]` alors `brin1` et `brin2` n'ont pas la même taille et je lève un `failwith`
- Sinon je calcule la distance entre les deux nucléotides des deux brins, je mets `bool` à 1 et j'appelle la fonction récursive

Question 4 `similarite : brin -> brin -> float` calcule la similarité procentuelle, s_p, entre deux brins de la même longueur donnés en arguments.

avec

- $s_p(h_1, h_2) = 1 - \frac{d_e(h_1,h_2)}{|h_1|}$
- h_1 et h_2 deux brins de même longueur
- $d_e(h_1, h_2)$ la distance d'édition entre h_1 et h_2
- $|h_1|$ la longueur du brin h_1

`similarite` doit utiliser la fonction précédente `distance`. Donc, je suis obligé de vérifier la taille de `brin1` et `brin2` avec `List.length` avant de faire appel à cette fonction dans `similarite`, ie que les deux sont non vides et qu'ils ont la même taille.

Cette fonction est en $\Theta(n_1+n_2)$ (en raison des appels de `List.length`) si les conditions ne sont pas respectées, avec n_1 la taille du 1er brin et n_2 la taille du 2nd, sinon en $\Theta(3n)$ (en raison des appels de `List.length` et de `distance`) avec n la taille commune des deux brins.

D'ailleurs, lors du calcul de s_p, je n'oublie pas de convertir $d_e(h_1, h_2)$ et $|h_1|$ en flottants, avec `float_of_int` avant de réaliser la division pour flottant.

Question 5 `brin_vers_chaine : brin -> acide list` prend en argument un brin bien formé, décode les codons à l'intérieur (entre les délimiteurs `START` et `STOP`) de sa première chaîne et les renvoie dans une liste d'acides aminés.

J'ai jugé mieux de considérer une fonction récursive comme lambda expression. En effet, le brin doit être parcouru un à un temps qu'un délimiteur `START` n'a pas été trouvé. Et dès que je trouve un `START`, alors je parcours le brin de trois en trois jusqu'à trouver un `STOP`.

D'où l'utilisation du couple `verif` :

- `fst verif=1` dès que je trouve un `START`
- `snd verif=1` dès que j'ai réussi à trouver une chaîne

Enfin, il ne reste plus qu'à utiliser `List.rev` pour renverser la liste obtenue.

Arbres phylogénétiques

Question 1 `arbre_phylo_vers_string : arbre_phylo -> string` prend en argument un arbre phylogénétique et retourne une représentation unique de cet arbre sous forme de chaîne de caractères.

Cette question est l'occasion de construire une fonction auxiliaire pour les questions suivantes : `is_complet : arbre_phylo -> int`.

Un arbre est dit complet si :

- celui-ci ne possède aucun brin vide
- les brins sont de même taille
- les malus sont corrects

Ainsi, lorsque l'une de ces conditions n'est pas respectée, je retourne un entier > 0, sinon je retourne 0.

`is_complet` est une fonction récursive qui traite 5 cas majeurs (assurant la terminaison):

- Si a est une feuille
- Si a est une branche de la forme a = Br(l, x, i, r) :
  - Si l est une feuille et r aussi
  - Si l est une feuille et r une branche
  - Si l est une branche et r une feuille
  - Si l est une branche et r aussi

Revenons à `arbre_phylo_vers_string`. Pour coder cette fonction, j'utilise une autre fonction auxiliaire `brin_vers_string : brin -> string` qui prend en argument un arbre phylogénétique et retourne une représentation unique de ce brin sous forme de chaîne de caractères. Ainsi, dans `arbre_phylo_vers_string`, il ne reste plus qu'à vérifier si l'arbre est complet ou non :

- Si l'arbre est complet, j'utilise l'opération de concaténation `^` et `brin_vers_string`
- Sinon, je lève un `failwith`

Question 2 `similaire: arbre_phylo -> arbre_phylo list -> arbre_phylo` calcule, pour un arbre phylogénétique donné et une liste d'arbres qui ont tous la même forme (donc même hauteur), l'arbre le plus similaire avec lui.

Il y a plusieurs conditions à vérifier :

- l'arbre doit être complet (se fait avec `is_complet`)
- les arbres de la liste doivent être complets (se fait avec `is_complet`)
- l'arbre doit avoir la même hauteur que tous les autres arbres de la liste
- les brins de l'arbre doivent avoir la même taille que ceux de tous les arbres de la liste

Pour vérifier la hauteur de l'arbre, j'utilise la fonction `tree_size: arbre_phylo -> int` qui retourne la taille de l'arbre. Cette fonction est construite à partir de la fonction auxiliaire `fold_tree: (brin -> 'a -> 'a) -> arbre_phylo -> 'a -> 'a` qui permet d'appliquer une fonction à tous les éléments de l'arbre en utilisant un parcours racine - sous-arbre gauche - sous-arbre droit.

Pour appliquer une fonction à des éléments deux à deux de deux arbres, j'utilise la fonction auxiliaire `fold_tree_compare: (brin -> brin -> 'a -> 'a) -> arbre_phylo -> arbre_phylo -> 'a -> 'a` qui utilise également un parcours racine - sous-arbre gauche - sous-arbre droit.

J'utilise une fonction récursive pour calculer $S(\mathcal{A}_1, \mathcal{A}_2)$ à l'aide de la fonction `similaire` et du couple `acc`:

- `fst acc` est l'arbre le plus similaire à a, noté a_s
- `snd acc` est un couple :
  - `fst(snd acc)` est la similarité entre a et a_s (ie $S(a, a_s)$)
  - `snd(snd acc)` est un booléen de type `int` qui vaut 1 si `liste_a` contient 1 élément, 0 sinon (en effet, si `liste_a=[]` et `snd(snd(acc))=0` alors `liste_a` était vide lors de l'appel de `similaire` et je lève un `failwith`)

Question 3

- `get_root : arbre_phylo -> brin` extrait le brin de la racine d'un arbre.
- `get_malus : arbre_phylo -> int` extrait le malus de la racine d'un arbre.
- `br : arbre_phylo * brin * arbre_phylo -> arbre_phylo` permet de construire un arbre à partir d'un sous-arbre gauche, d'un brin et d'un sous-arbre droit.

Pour `get_root` et `get_malus`, il suffit simplement de retourner le brin ou le malus de la racine. Si l'arbre n'est pas complet ou que c'est juste une feuille, je lève un `failwith`.

Pour `br`, je vérifie avant tout que les deux sous-arbres donnés sont complets et que le brin est non vide (auquel cas, je lève un `failwith`). Ensuite, il ne reste plus qu'à retourner la branche associée avec le nouveau malus qui en découle.

Question 4 `gen_phylo : brin -> brin -> brin -> arbre_phylo list` prend en paramètres trois brins de la même taille et renvoie la liste de tous les arbres phylogénétiques que je peux générer avec ces brins. J'exclus les cas triviaux des arbres formés juste d'une feuille (aucun fils).

Je vérifie que l'un des brins est non vide et que les brins ont la même taille (auquel cas je lève un `failwith`).

Ayant exclu les cas triviaux, il suffit de retourner la liste des trois branches que je peux former à partir des trois brins. Pour cela, j'utilise une fonction récursive qui inclut deux arguments :

- `acc` qui correspond à la liste des branches que je peux former avec les trois brins
- `n` la taille de la liste `acc`. Dès qu'elle atteint 3, j'arrête l'appel de la fonction récursive (d'où la terminaison de la fonction `gen_phylo`).

Question 5 `min_malus : arbre_phylo list -> arbre_phylo` prend en paramètre une liste d'arbres phylogénétiques et renvoie celui qui a la valeur minimale de malus global.

Je vérifie comme d'habitude pour toute liste d'arbres que celle-ci est non vide et que tous les arbres sont complets (dans le cas contraire, je lève un `failwith`).

J'utilise simplement un `List.fold_left` qui retourne à la fin le couple `acc` :

- `fst acc` est aussi un couple tel que
  - `fst (fst acc)` est un booléen de type `int` qui vaut 0 si un des arbres est incomplet
  - `snd (fst acc)` contient le malus de `snd acc`
- `snd acc` est la première occurence dans la liste de l'arbre ayant le plus petit malus

Question 6 `gen_min_malus_phylo : brin list -> arbre_phylo` prend en paramètre une liste de brins de la même taille et renvoie l'arbre phylogénétique qui a comme feuilles ces brins et qui a la plus petite valeur du malus global parmi tous les arbres générés.

N'ayant pas réussi cette question au brouillon, je ne l'ai pas codé.

III/ Conclusion

A- Remarques complémentaires

- J'ai choisi d'utiliser des `Printf.printf` au lieu d'`assert`, car je souhaitais que le code soit accessible de tous (d'où notamment l'utilisation d'un `Makefile` pour séparer le code source des tests).
- Pour toutes informations complémentaires, n'hésitez pas à jeter un coup d'oeil au fichier `source.ml` qui est commenté (et au fichier `test.ml`, si vous le souhaitez, car ce fichier dispose de certaines fonctions non mentionnées dans ce rapport).
- Toutes les fonctions sont commentées de la même manière
  - `@requires` cite ce que prend en argument une fonction
  - `@assigns` cite les objets qui sont modifiés après entrée dans une fonction
  - `@ensures` cite ce que fait la fonction en question

B- Difficultés rencontrées

Seules la question 5 de la partie 1 et la question 6 de la partie 2 m'ont posé des problèmes :

- Comme indiqué précedemmment, je n'ai pas réussi la question 6 au brouillon.
- Et en ce qui concerne la question 5, il m'a fallu plusieurs jours pour l'appréhender. Voici plus bas une version antérieure de la fonction `brin_vers_chaine`. Comme vous pouvez le constater, ce sont le changement de pas et la gestion de deux `START` consécutifs qui ont été difficile à gérer.

```
let brin_vers_chaine brin =
  let end_bvc snd_verif chain = if (snd_verif) = 0
                                then failwith "brin_vers_chaine : brin invalide (ie vide ou mal formé)"
                                else chain
  in let rec rec_brin_vers_chaine brin chain verif =
    match brin with
      | [] -> end_bvc ( snd verif ) chain
      | n1::ns1 -> match ns1 with
        | [] ->  end_bvc ( snd verif ) chain
        | n2::ns2 -> match ns2 with
          | [] ->  end_bvc ( snd verif ) chain
          | n3::ns3 -> if ns3=[] then end_bvc ( snd verif ) chain
                       else let a = codon_vers_acide n1 n2 n3 in
                        if ( snd verif ) = 1 then rec_brin_vers_chaine ns3 chain verif
                        else if ( fst verif ) = 0 then
                          ( if a= START then rec_brin_vers_chaine ns3 chain (1,0)
                            else rec_brin_vers_chaine ns1 chain verif )
                           else match a with
                            | STOP -> if chain = [] then rec_brin_vers_chaine ns3 chain (0,0)
                                      else chain
                            | START -> rec_brin_vers_chaine ns3 [] (1,0)
                            | _ -> let chain2 = a::chain in rec_brin_vers_chaine ns3 chain2 verif
  in List.rev(rec_brin_vers_chaine brin [] (0,0));;
```