# Projet final IPFL - Rapport technique

**Programmation Fonctionnelle**

**Lucas RODRIGUEZ**

17 Mai 2021

## Introduction

Ce rapport présente les résultats de mes recherches relatives à la réalisation du projet de Programmation Fonctionnelle. Les pistes de recherches ainsi que les codes abondamment commentés suivis de plusieurs exemples d'exécution sont présents ci-dessous. Vous pouvez trouver ci-joint le dossier `projet.ml` contenant le code dans son intégralité.

## 1 Séquences ADN/ARN

On commence le projet en déclarant les types suivants :

```
(* Déclaration des types *)
type nucleotide = A | C | G | T;;
type brin = nucleotide list;;
```

**Question 1** On souhaite écrire une fonction `contenu_gc` retournant le contenu GC d'un brin d'ADN passé en argument.
Pour cela, on va parcourir le brin à l'aide d'un `List.fold_left` en incrémentant un compteur à chaque fois que le nucléotide rencontré est un `C` ou un `G`. A chaque occurence, on incrémente le compteur de $\frac{1}{\text{longueur}(x)}$ où $x$ est le brin. Au final, la fonction retourne bien le contenu GC du brin $x$

```
(* Fonction retournant le contenu GC d'un brin d'ADN passé en argument *)
let contenu_gc (x : brin) : float = List.fold_left (
    fun compteur e -> if (fun e -> e = C || e = G) e
      then compteur +. (1./. float_of_int (List.length x))
      else compteur
  ) 0. x;;

(* TEST *)
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.);;
```

**Question 2** Afin de calculer le complémentaire d'un brin, on va parcourir la structure du brin via une clause `match` et on va traiter les différents cas pour chaque nucléotide : si c'est un `A`, `T`, `C` ou bien `G`.

```
(* Fonction calculant et retournant le brin complémentaire du brin donné en paramètre *)
let rec brin_complementaire (x : brin) : brin = match x with
  | [] -> []
  | x::r -> if x = A
    then T::brin_complementaire r
    else
    if x = T
```

```
    then A::brin_complementaire r
    else
    if x = C
    then G::brin_complementaire r
    else
    if x = G then C::brin_complementaire r else [C];;

(* TEST *)
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]);;
```

**Question 3** Nous souhaitons implémenter une fonction retournant la distance d'édition entre 2 brins $x$ et $y$ donnés en paramètres.

**Définition 1 (Distance d'édition)** Soient $x$ et $y$ 2 brins de taille $n \in \mathbb{N}^*$. On définit la distance d'édition $d_e$ entre $x$ et $y$ par

$$d_e(x, y) = \sum_{i=1}^{n} \delta(x_i, y_i)$$

où $\forall i \in [\![1, n]\!],\ \delta(x_i, y_i) = \begin{cases} 1 \text{ si } x_i = y_i \\ 0 \text{ si } x_i \neq y_i \end{cases}$ [1]

Afin de calculer cette fonction, il faut donc parcourir simultanément les 2 brins et les comparer nucléotide par nucléotide. Pour cela, nous allons utiliser la fonction `List.fold_left2` qui permet de parcourir les 2 listes avec un accumulater pour réaliser des opérations sur chaque nucléotide parcouru (ici : la fonction sera un opérateur de comparaison permettant d'incrémenter l'accumulateur servant de compteur pour la distance)

```
(* Fonction retournant la distance d'édition entre les 2 brins donnés en paramètres *)
let distance (x : brin) (y : brin) : int =
  List.fold_left2 (fun acc x y -> if x = y then acc else 1 + acc) 0 x y;;

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

**Question 4** Nous voulons implémenter une fonction calculant la similarité procentuelle entre 2 brins $x$ et $y$

**Définition 2 (Similitude procentuelle)** Soient $x$ et $y$ 2 brins de même taille $n \in \mathbb{N}^*$. On définit la similarité procentuelle $s_p$ entre $x$ et $y$ par :

$$s_p(x, y) = 1 - \frac{d_e(x, y)}{|x|}$$

où $|.|$ représente la longueur d'un brin.

Afin d'implémenter cette fonction, nous comparons tout d'abord si les 2 brins $x$ et $y$ possèdent la même longueur ; dans le cas échéant, nous levons une exception. S'ils sont tous les 2 de même longueur, nous pouvons ainsi calculer la similarité de manière analogue à la formule énoncée ci-dessus. Il faut faire attention à la conversion en nombres flottants pour respecter le typage demandé : `val similarite : brin -> brin -> float = <fun>`

```
(* Fonction calculant et retournant la similarité entre 2 brins donnés en paramètre *)
let similarite (x : brin) (y : brin) : float =
```

1. Il s'agit du delta de Kronecker appliqué aux brins $x$ et $y$

```
  if (List.length x) <> (List.length y)
  then raise (Failure "brin invalide")
  else
    1.0 -. (float_of_int (distance x y)) /. (float_of_int (List.length x));;

(* TEST *)
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 [A; G; C; T] [A; G; C] = raise (Failure "brin invalide"));;
```

Nous implémentons maintenant le type `acide`, ainsi que la fonction `codon_vers_acide`.

**Question 5** Tout d'abord, nous implémentons une fonction externe vérifiant si la taille du brin donné en paramètre est bien un multiple de 3 (pour une cohérence avec la taille des codons)

```
let verif_taille_brin (brin:brin): bool  = if List.length brin mod 3 = 0 then true else false;;
```

Nous pouvons ensuite implémenter la fonction `brin_vers_chaine` selon le protocole suivant :

(1) Vérification de la taille du brin en paramètre (si $|x| \equiv 0 \mod 3$, alors nous poursuivons)

(2) Parcours sur la structure du brin grâce à une clause `match`

(3) Filtrage selon les cas
- ▷ Si le nucléotide code un `STOP`, alors on renvoie `[]`
- ▷ Si le nucléotide code un `START`, alors on effectue un appel de la fonction sur la suite de la liste (après le 2ème nucléotide suivant celui inspecté)
- ▷ Si le nucléotide ne code ni un `START`, ni un `STOP`, alors on décode le codon pour trouver l'acide correspondant, que l'on concaténe avec un appel à la fonction sur la suite de la liste (après le 2ème nucléotide suivant celui inspecté actuellement)

```
(* Fonction qui détermine et renvoie une liste d'acide aminés de la première chaîne du brin *)
let rec brin_vers_chaine (x : brin) : acide list =
  if verif_taille_brin x = false
  then raise (Failure "brin invalide")
  else match x with
    | e::y::z::l ->
      if codon_vers_acide e y z = STOP
      then []
      else
      if codon_vers_acide e y z != START
      then [codon_vers_acide e y z] @ brin_vers_chaine l
      else brin_vers_chaine l
    | _ -> raise (Failure "brin invalide");;

(* TEST *)
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 [T; A; C; T; A; C] = raise (Failure "brin invalide"));;
let _ = assert (brin_vers_chaine [T; A; C; G; G; A; T; C] = raise (Failure "brin invalide"));;
```

**Question 6 (optionnel)** Nous avons également voulu coder la fonction `brins_vers_chaine`. Le fonctionnement de cette fonction semble assez similaire à celui de la **Question 5**. Néanmoins, quelques modifications semblent être nécessaire.
Après avoir vérifié la cohérence du brin en regardant sa taille, on effectue un parcours du brin, on regarde si les 3 premiers nucléotides codent un acide `START`.
Si c'est le cas, on appelle la fonction précédente sur la suite de la liste que l'on concatène avec un appel récursif de la fonction actuelle sur la même suite de la liste. sinon on effectue un simple appel récursif sur la suite de la liste.

```
(* Fonction retournant le décodage de toutes les chaînes du brin donné en paramètre *)
let rec brin_vers_chaines (x : brin) : acide list list  =
  if verif_taille_brin x = false
  then raise (Failure "brin invalide")
  else match x with
    | e::y::z::l -> if codon_vers_acide e y z = START then
        [brin_vers_chaine l]@ brin_vers_chaines l
      else brin_vers_chaines l
    | _ -> [];;

(* TEST *)
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 [T; A; C; T; A; C] = raise (Failure "brin invalide"));;
let _ = assert (brin_vers_chaines [T; A; C; G; G; A; T; C] = raise (Failure "brin invalide"));;
```

## 2 Arbres phylogénétiques

**Question 1** Nous commençons par implémenter une fonction externe retournant une représentation sous forme de `string` d'un brin donné en paramètre :

```
(* Fonction auxiliaire qui retourne une string formatée pour un brin en particulier *)
let rec affichage_brin (brin: brin) : string = match brin with
  | [] -> ""
  | x::s -> if x = A then "A"^(affichage_brin s)
    else if x = T then "T"^(affichage_brin s)
    else if x = C then "C"^(affichage_brin s)
    else "G"^(affichage_brin s);;
```

Nous pouvons ensuite parcourir les différents noeuds de l'arbre pour en ressortir une représentation en `string` unique de l'arbre :

```
(* Fonction retournant une string unique représentant de manière unique un arbre phylogénétique
let rec arbre_phylo_vers_string (a : arbre_phylo) : string = match a with
  | Lf(brin) ->  (affichage_brin brin)
  | Br(l, brin, malus, r) ->
    (arbre_phylo_vers_string l)^"\n"^(affichage_brin brin)^" ("
    ^(string_of_int malus)^")"^"\n"^(arbre_phylo_vers_string r);;
```

On peut enfin afficher si on le souhaite la représentation en faisant
`print_string (arbre_phylo_vers_string arbre);;`

**Question 2** Nous souhaitons créer une fonction qui avec un arbre phylogénétique de référence et une liste d'autres arbres phylogénétiques, souhaite retourner l'arbre de cette liste qui lui est le plus similaire, au sens de la mesure $s_p$.
Pour cela, nous allons découper le travail en plusieurs fonctions externes qui seront utilisées par la fonction principale `similaire : arbre_phylo -> arbre_phylo list -> arbre_phylo`

▷ **Fonction de parcours d'un arbre phylogénétique** Pour cette fonction, nous effectuons un simple parcours de l'arbre selon le modèle : racine - sous-arbre gauche - sous-arbre droit.

```
(* Fonction de parcours d'un arbre (parcours racine - sous-arbre gauche - sous-arbre droit) *)
let rec arbre_parcours (arbre:arbre_phylo) : brin list = match arbre with
  | Lf(x) -> [x]
  | Br(l, x, malus,  r) ->  [x] @ arbre_parcours l @ arbre_parcours r ;;
```

▷ **Fonction de calcul de similarité entre 2 arbres** Cette fonction est très importante. En effet, elle va calculer la similarité $S$ entre 2 arbres.

**Définition 3 (Similarité entre 2 arbres)** Soit $\mathcal{A}_1$ et $\mathcal{A}_2$ : 2 arbres phylogénétiques de même nombre de noeuds $n \in \mathbb{N}^*$. On définit la similarité entre $\mathcal{A}_1$ et $\mathcal{A}_2$ par la quantité suivante :

$$S(\mathcal{A}_1, \mathcal{A}_2) = \sum_{i=1}^{n} s_p(\mathcal{A}_1^i, \mathcal{A}_2^i)$$

où $\mathcal{A}_1^i$ est le i-ème noeud de $\mathcal{A}_1$.

Pour calculer cette quantité, nous allons utiliser la fonction `List.fold_left2` afin d'incrémenter un accumulateur initialement nul avec les similarités de chacun des noeuds appartenants aux 2 arbres.

```
(* Fonction calculant la similarité entre 2 arbres phylogénétiques *)
let arbre_similarite (a1:arbre_phylo) (a2:arbre_phylo) : float =
  List.fold_left2 (fun acc x y -> acc +. similarite x y) 0. (arbre_parcours a1)
  (arbre_parcours a2);;
```

▷ **Fonction de comparaison de similarité entre une référence et 2 arbres** Cette fonction prend en paramètre 1 arbre phylogénétique de référence puis 2 autres arbres $\mathcal{A}_1$ et $\mathcal{A}_2$, et retourne l'arbre parmi ces 2 qui est le plus similaire à l'arbre de référence :

```
(* Fonction comparant les similarités de 2 arbres par rapport à un arbre de référence *)
let comparaison_similarite (ref:arbre_phylo) (a1:arbre_phylo) (a2:arbre_phylo) : arbre_phylo =
  if abs_float (arbre_similarite ref a1) < abs_float (arbre_similarite ref a2)
  then a1 else a2;;
```

▷ **Fonction principale** Finalement, grâce aux fonctions ci-dessus, nous pouvons enfin implémenter la fonction principale `similaire` :

```
(* Fonction retournant l'arbre le plus similaire à l'arbre a parmi une liste l d'arbres phylo *
let rec similaire (a : arbre_phylo) (l : arbre_phylo list) : arbre_phylo = match l with
  | [] -> failwith "liste d'arbres invalide"
  | [arbre] -> arbre
  | x::suite -> comparaison_similarite a x (similaire a suite);;
```

**Question 3 (1)** Cette fonction `get_root` doit retourner le brin qui compose la racine d'un arbre phylogénétique donné en paramètre. Pour cela, nous effectuons un simple filtrage sur la structure de l'arbre passé en argument.

- ▷ Si l'arbre n'est composé que d'une feuille, c'est donc la racine et on la renvoie.
- ▷ Si l'arbre est plus important, on n'extraie que l'élément à la racine (ici, `x`)

```
(* Fonction d'extraction du brin de la racine de l'arbre *)
let get_root (a : arbre_phylo) : brin = match a with
  | Lf(x) -> x
  | Br(l, x, m, r) -> x;;
```

**Question 3 (2)** Cette fonction `get_malus` doit retourner le malus global de l'arbre phylogénétique passé en paramètre. Pour cela, nous effectuons également un filtrage sur la strucutre de l'arbre passé en argument. On prend comme hypothèse que le malus d'un arbre avec qu'un noeud (la racine) est égal à 0

```
(* Fonction d'extraction du malus de la racine de l'arbre *)
let get_malus (a : arbre_phylo) : int = match a with
  | Lf(x) -> 0
  | Br(l, x, m, r) -> m;;
```

**Question 3 (3)** La fonction `br` permet de construire un arbre phylogénétique à partir d' un brin et de 2 arbres phylogénétiques, en paramètre.
L'implémentation de cette fonction semble être simple à premier abord. Néamoins, il faut faire attention à bien implémenter le malus global de l'arbre créé. Pour cela, nous utilisons les fonctions `distance`, `get_root` et `get_malus` précédemment créées.

```
(* Fonction de construction d'un arbre phylo à partir de données en paramètre *)
let br (ag : arbre_phylo) (b : brin) (ad : arbre_phylo) : arbre_phylo =
  Br(
    ag,
    b,
    distance (get_root ag) b + distance (get_root ad) b + (get_malus ag) + (get_malus ad),
    ad);;
```

**Question 4** On cherche à créer une fonction `gen_phylo` qui, à partir de 3 brins de même taille, génère et renvoie la liste de tous les arbres phylogénétiques. Pour des besoins théoriques, nous allons noter $X$, $Y$ et $Z$ ces 3 brins
Nous effectuons un petit travail théorique préliminaire afin d'écrire une fonction propre et correcte :
Nous avons l'alphabet $\mathcal{A} = \{X, Y, Z\}$. Le problème de génértion est équivalent à la recherche de toutes les permutations de cet alphabet. Il y a donc $3! = 6$ possibilités d'arbres [2]
Voici les différentes permutations de $\mathcal{A}$ : elles constituent tous les arbres possibles que l'on recherche.

XYZ - XZY - YXZ - YZX - ZXY - ZYX

En vérifiant au préalable que les 3 brins sont de même taille, nouvs pouvons ensuite renvoyer la liste des arbres phylogénétiques possibles à l'aide de la fonction `br` récemment implémentée.

```
(*
  Fonction générant tous les arbres phylo possibles depuis 3 brins d'ADN
*)
let gen_phylo (x : brin) (y : brin) (z : brin) : arbre_phylo list =
  if List.length x = List.length y && List.length x = List.length z
  then
    [br (Lf x) y (Lf z); br (Lf x) z (Lf y); br (Lf y) x (Lf z);
```

2. en ayant fait l'hypothèse qu'on enlève les arbres triviaux (limités à 1 seul noeud)

```
    br (Lf y) z (Lf x); br (Lf z) x (Lf y); br (Lf z) y (Lf x)]
  else
    failwith "brins de tailles différentes";;
```

**Question 5** On souhaite écrire une fonction `min_malus` qui renvoie l'arbre avec le malus global minimal parmi une liste d'arbres donnée en paramètre. Il s'agit donc d'un problème de minimisation d'un critère : le malus parmi un ensemble d'éléments : les arbres de la liste. Procédons en 2 temps : on commence par écrire une fonction externe qui permettra de renvoyer l'arbre avec le malus global le plus petit parmi 2 arbres donnés en paramètre :

```
(* Fonction retournant l'arbre avec le plus petit malus global parmi 2 arbres *)
let min_malus_comparaison (a1:arbre_phylo) (a2:arbre_phylo) : arbre_phylo =
  if get_malus a1 < get_malus a2 then a1 else a2;;
```

Grâce à l'implémentation de la fonction précédente, nous pouvons dès à présent s'occuper de la fonction principale `min_malus` :

```
(* Fonction retournant l'arbre avec le plus petit malus global parmi une liste d'arbres *)
let rec min_malus (l : arbre_phylo list) : arbre_phylo = match l with
  | [] -> failwith "liste invalide d'arbres"
  | [x] -> x
  | x::suite -> min_malus_comparaison x (min_malus suite);;

(* TEST *)
let _ = assert (min_malus [arbre2; arbre] = arbre);;
```

**Question 6** On souhaite créer une fonction `gen_min_malus_phylo` qui renvoie l'arbre phylogénétique minimisant son malus global, réalisable depuis une liste de brins donnée initialement en paramètre.
Néanmoins, avant de commencer à détailler le protocole adopté pour réaliser ce projet, nous devons définir quelques hypothèses de base sur lesquelles reposeront une bonne partie de nos fonctions implémentées.
Nous savons dès le début de la section 2 du projet qu'un arbre phylogénétique est modélisé théoriquement par un arbre binaire où chaque noeud sauf les feuilles (noeuds du dernier niveau) possèdent 2 fils. Il s'agit donc d'un **arbre binaire parfait** [3].
Une caractéristique numérique de ces arbres est la suivante. On note $h$ la hauteur de l'arbre phylogénétique et $n$ son nombre de sommets (noeuds).

Les arbres binaires parfaits nous permettent d'énoncer que :

$$2^h - 1 = n \iff n + 1 = 2^h \iff \boxed{h = \log_2(n+1)}$$

On peut donc énoncer directement une vérification [4] préliminaire :

Si $h \notin \mathbb{N}$, la fonction retourne une erreur.
En effet, une liste de $n$ brins donnés en paramètres ne pourra, dans cette situation, en aucun cas créer un arbre binaire parfait.

3. https://fr.wikipedia.org/wiki/Arbre_binaire#Types_d'arbres_binaires
4. Après implémentation, cette vérification est plus que nécessaire car dans le cas contraire, la fonction `List.fold_left2` présente dans la fonction `distance| lévèra une exception qui se propagera sur l'intégralité de l'exécution du programme.`

Après ces clarifications nécessaires, passons à la description du protocole adopté :

(1) Vérification de la taille de la liste des brins
E : `brin list`
S : `bool`

(2) Génération de toutes les permutations possibles de la liste des brins
E : `brin list`
S : `brin list list`

(3) Génération de tous les arbres possibles (1 arbre/permutation)
E : `brin list list`
S : `arbre_phylo list`

(4) Détermination de l'arbre minimisant le malus global
E : `arbre_phylo list`
S : `arbre_phylo`

Afin d'implémenter ces différentes étapes, nous allons découper l'implémentation en plusieurs fonctions :

**Etape 1**

▷ **Fonction** $x \longmapsto \log_b(x)$

```
(*
  Fonction logarithme de base b appliqué à x
 *)
let logb x b = (log10 x)/.(log10 b);;
```

▷ **Fonction vérifiant si un nombre est dans** $\mathbb{N}$
- ▷ La fonction `modf` prend un flottant en entrée et renvoie un tuple avec la partie décimale en premier élément et la partie entière inférieure en deuxième élément.
- ▷ La fonction `fst` correspond à la projection canonique sur le premier élément d'un tuple.

```
(* Fonction de vérification si un nombre est un entier *)
let isInt x = (fst (modf x)) = 0.
```

▷ **Fonction de vérification de la cohérence de la taille de la liste d'entrée** On vérifie si $\log_2(n+1) \in \mathbb{N}$ ou pas. On renvoie le résultat de cette clause d'égalité.

```
(*
  Fonction vérifiant si la taille de la liste des brins est autorisée pour
  la construction des arbres phylogénétiques
*)
let verification_taille (l : brin list) : bool = isInt (logb (float_of_int(List.length l + 1))
```

**Etape 2**

▷ **Fonction d'insertion dans une liste** Cette fonction permet d'insérer un élément $x$ à tous les emplacements possibles d'une liste. La fonction renvoie ainsi la liste de toutes les listes réalisables par cette insertion.

```
(*
  Fonction renvoyant une liste de liste de brins où dans chaque nouvelle liste de cette liste,
  le brin elt est rajouté à chaque endroit de la liste d'origine

  Exemple : Si liste = [[A]; [T]] et elt = [C] alors
  insertion_liste [C] [[A]; [T]];; renverra [[C; A; T]; [A; C; T]; [A; T; C]]
 *)
let rec insertion_liste (elt : brin) (liste : brin list) : (brin list list) = match liste with
  | [] -> [[elt]]
  | x::l -> (elt::liste) :: (List.map (fun e -> x::e) (insertion_liste elt l));;
```

▷ **Fonction générant toutes les permutations possibles de la liste de brins**

```
(*
  Fonction générant la liste de toutes les permutations possibles depuis une liste de brins
  donnée en paramètre
*)
let rec generation_permutations (liste : brin list) : (brin list list) = match liste with
  | [] -> [liste]
  | x::l -> List.flatten (List.map (insertion_liste x) (generation_permutations l));;
```

**Etape 3**

▷ **Fonction permettant de découper une liste en 2 listes à partir du $n$-ème indice**

```
(*
  Fonction permettant de découper une liste en 2 listes
  découpées à partir du n-ième index de la liste d'origine
*)
let rec decoupe_liste (n : int) (l : brin list) =
  if n = 0 then ([], l) else
    match l with
    | [] -> ([], [])
    | h :: t -> let (l1, l2) = decoupe_liste (n-1) t in (h :: l1, l2);;
```

▷ **Fonction permettant de découper une liste en 2 listes découpées à partir du centre de la liste d'origine**

```
let decoupe_liste_moitie (l : brin list) = decoupe_liste (List.length l / 2) l;;
```

▷ **Fonction retournant la première liste découpée (première moitiée de la liste d'origine)**

```
let get_first_half (l : brin list) : (brin list) = fst (decoupe_liste_moitie l);;
```

▷ **Fonction retournant la deuxième liste découpée (deuxième moitiée de la liste d'origine)**

```
let get_second_half (l : brin list) : (brin list) = snd (decoupe_liste_moitie l);;
```

▷ **Fonction de construction d'un arbre à partir d'une liste de brins**

```
let rec generation_arbre (liste : brin list) : (arbre_phylo) = match liste with
  | [] -> Lf([])
  | [x] -> Lf(x)
  | x::l -> br (generation_arbre (get_first_half l)) x (generation_arbre (get_second_half l));;
```

Il faut maintenant étendre cette fonction à une liste de listes de brins, qui renvoie tous les arbres possiblement réalisables :

▷ **Fonction générant tous les arbres possibles depuis une liste de liste de brins**

```
let rec generation_liste_arbres (liste : brin list list) : (arbre_phylo list) = match liste
with
  | [] -> []
  | x::l -> (generation_arbre x)::(generation_liste_arbres l);;
```

**Etape 4**

Il suffit maintenant d'implémenter la fonction `gen_min_malus_phylo` qui assemble toutes les fonctions implémentées précédemment et qui retourne l'arbre réalisable à partir des brins donnés en paramètre **et** qui minimise le malus global.

```
let gen_min_malus_phylo (l : brin list) : arbre_phylo =
  if verification_taille l then
    min_malus (generation_liste_arbres (generation_permutations l))
  else
    raise (Failure "La liste ne permet pas de créer des arbres binaires parfaits.");;
```

### Etude de complexité pour la question (6)

On note $\mathcal{O}$, la notation de Landau désignant la complexité temporelle et $n$ le nombre de brins entrés en paramètre.
Analysons la complexité de chacune des étapes de construction de la question 6.

(1) Vérification de la taille des brins : Il faut effectuer un parcours linéaire de la liste : $\mathcal{O}(n)$

(2) Génération de toutes les permutations possibles de la liste des brins : $\mathcal{O}(n!)$

(3) Génération de tous les arbres possibles :
   ▷ Découpage de chaque liste en 2 : $\mathcal{O}(n)$
   ▷ Génération d'un arbre : $\mathcal{O}(2^n)$
   ▷ Génération de tous les arbres : $\mathcal{O}(2^n n!)$

Au final, la complexité est en $\boxed{\mathcal{O}(n!2^n n)}$. On peut prévoir une erreur de type StackOverflow vers des valeurs de $n$ assez peu élevées.

# Conclusion

Au final, j'ai réussi à traiter et finaliser l'intégralité des questions de ce projet. Même si au premier abord, ce thème lié à la biologie ne m'a pas particulièrement inspiré, la découverte d'OCaml et de la programmation fonctionnelle m'ont beaucoup plu et m'ont permis d'appréhender les différentes questions avec plus d'assurance.
N'ayant jamais fait de programmation fonctionnelle en MPSI/MP, j'ai découvert ce paradigme que récemment et j'ai particulièrement apprécié le fort typage des données offert par Caml, permettant de « contrôler » avec plus de rigueur les structures manipulées.
Vu la complexité moyenne en temps pour la méthode implémentée pour la dernière question, de nouvelles modifications majeures doivent sans doute être mises en oeuvre afin de la tester sur des valeurs élevées de $n$.