Programmation Fonctionnelle
Projet

28 octobre 2023

1 Prérequis

Contrairement aux TPs, ce projet ne vit pas sur Learn-OCaml. Avant toute chose, il vous faut donc installer le nécessaire pour travailler (compilation, tests).

Pour ce faire, faites un **fork** du projet

https://gaufre.informatique.univ-paris-diderot.fr/Bernardi/project-pf5/

(pour quelques rappels sur `git`, reportez-vous à la section 5). Suivez les instructions données dans le fichier `README.md`.

Si vous rencontrez un problème, faites-nous signe au plus vite. Tant que cette étape d'installation n'est pas réalisée, vous ne pourrez pas faire le projet.

La date limite pour rendre le projet est le 22 décembre 2023.

2 Introduction

L'ADN est le code qui régule pratiquement toute activité biochimique des cellules des êtres vivants. Les séquences d'ADN sont formées à partir de quatre *nucléotides* : cytosine (C), guanine (G), adénine (A) et thymine (T). Ces nucléotides sont également appelés *bases nucléiques*. Par exemple :

`TATAAA`	contient	6 bases
`TAATACGACTCACTATAG`	contient	18 bases

Le rôle de l'ADN est d'encoder les *protéines* produites par les cellules, une protéine pouvant être vue comme l'une des briques de base du vivant. Chaque protéine est encodée par un *gène*, une sous-partie d'une séquence d'ADN (une protéine donnée pouvant par ailleurs être encodée par plusieurs gènes distincts).

Lorsqu'une certaine protéine est nécessaire à l'activité d'une cellule donnée, le gène associé est lu et traduit en une séquence d'ARN messager[1]. Les *ribosomes* décodent ensuite cette séquence pour effectuer la synthèse de la protéine cible.

1. Très similaire à l'ADN, mais où la thymine est remplacée par l'uracil.

Ce processus de traduction d'un gène en ARN suivi de la synthèse de la protéine encodée par ce gène est appelé *transcription*[2].

L'encodage d'information par l"ADN est un procédé si peu coûteux qu'il est parfois envisagé comme un avenir possible pour le problème général du stockage de données [2]. On estime que la biosphère contient $5.3(\pm 3.6) \times 10^{31}$ millions de bases [3]. Le génome humain est stocké dans 23 paires de chromosomes, pour un total de 3×10^9 paires of bases : déplié, il mesurerait presque 5 cm de long.

Du fait de sa taille, la manipulation d'une molécule d'ADN en laboratoire est un problème loin d'être simple. La plus ancienne technique de séquençage d'un brin d'ADN (*i.e.* déterminer sa séquence de nucléotides) est le *séquençage shotgun*, découvert en 1979. L'idée est de découper un ensemble de copies d'un même brin en fragments aléatoires (par exemple de longueurs $2, 10, 50$, ou 150 milliers de bases), puis de recomposer la séquence de ce brin par *alignement* de ces fragments. Malgré sa simplicité, le séquençage shotgun est à la base du projet de séquençage du génome humain lancé en 1990 et achevé en 2022. Un élément-clef de ce projet est la mise en œuvre d'algorithmes d'*alignements de chaînes* [1], *i.e.* de programmes cherchant à réaligner les fragments d'un brin obtenus par découpage aléatoire.

Sans surprise, compte tenu de l'essor des techniques de séquençage et de l'immense quantité d'ADN présente dans la biosphère, il existe aujourd'hui de nombreuses bases de données répertoriant des collections de gènes ainsi que leurs protéines associées. L'exemple le plus connu est sans doute celui-ci :

`https://www.ncbi.nlm.nih.gov/`

Outre des données brutes, ces bases proposent une masse considérable de métadonnées sur les séquences biologiques découvertes à ce jour. On pourra par exemple consulter la page suivante, qui donne le détail de chaque protéine encodée par le génome de la bactérie Escherichia coli :

`https://www.ncbi.nlm.nih.gov/nuccore/NZ_MT263755.1`

L'extraction rapide des informations d'une telle page relève d'une nécessité pratique : les *expressions régulières* se prêtent très efficacement à ce rôle. Vous les avez déjà rencontrées dans l'usage d'outils tels que `sed`, `awk`, and `grep`.

Objectif du projet. Ce projet est inspiré par les problématiques soulevées ci-dessus. La première partie porte sur les problèmes de l'extraction de gènes et de la reconstruction d'une séquence d'ADN (ou plus modestement, celui du calcul de ce qu'on appelle la séquence consensus d'un ensemble de fragments – nous ignorerons ici le problème de leur alignement). La seconde partie porte sur les expressions régulières.

2. *c.f.* par exemple https://www.khanacademy.org/science/biology/gene-expression-central-dogma/transcription-of-dna-into-rna/a/stages-of-transcription

Remarques. Le niveau de difficulté des exercices est indiqué par zéro (facile) une (⋆ difficulté moyenne) ou deux étoiles (⋆⋆ plus difficile). N'hésitez pas à poster des questions (et non des réponses !) sur le sujet en écrivant à la liste de diffusion de PF5 :

l3.pf5.info@listes.u-paris.fr

3 Analyse de brins d'ADN

Comme indiqué dans l'introduction, un brin d'ADN forme une séquence de bases nucléiques. Les quatre bases nucléiques, représentées par les lettres A (adénine), T (thymine), G (guanine), et C (cytosine), seront représentées par le type OCaml suivant :

```
type base = A | C | G | T | WC
```

Notons que ce type possède une cinquième valeur possible, `WC` (pour « Wild Card »), représentant une base inconnue (*e. g.* qui n'a pu être correctement identifiée lors du séquençage). Un brin d'ADN sera représenté par une liste de bases :

```
type dna = base list
```

Dans cette section, il vous est demandé d'implémenter quelques fonctions génériques de manipulation de listes qui pourront être utiles pour la suite. Chaque fonction peut bien sûr se servir des précédentes.

3.1 Échauffement

Exercice 3.0. Écrire une fonction

```
explode : string -> char list
```

qui convertit une chaîne de caractères en une liste de caractères. Par exemple :
`explode "Hello!" = ['H', 'e', 'l', 'l', 'o', '!']`

Exercice 3.1. Écrire une fonction

```
base_of_char : char -> base
```

qui convertit un caractère en base nucléique. Les caractères `'A'`, `'T'`, `'G'`, `'C'` seront respectivement convertis en `A`, `T`, `G`, `C`. Tout autre caractère sera converti en `WC`. Par exemple :
`base_of_char 'G' = G` et `base_of_char '.' = WC`.

Exercice 3.2. Écrire une fonction

```
dna_of_string : string -> dna
```

qui convertit une chaîne de caractères en un brin d'adn. Par exemple :
`dna_of_string "GTAA..CT" = [G; T; A; A; WC; WC; C; T]`

Exercice 3.3. Écrire une fonction

```
string_of_dna : dna -> string
```

qui inversement, convertit un brin d'adn en chaîne de caractères. Le nucléotide `WC` sera converti en un point simple (`.`). Par exemple :
`string_of_dna [G; T; A; A; WC; WC; C; T] = "GTAA..CT"`

3.2 Découpage d'un brin d'ADN en gènes

Les gènes encodés par un brin d'ADN (les sous-parties de ce brin encodant des protéines) sont encadrés par des délimiteurs appelés *codons*. Chaque gène est précédé d'un certain *codon de départ*, START, et est suivi par un *codon d'arrêt*, STOP.

L'objectif de cette section est, étant donné un brin d'ADN, d'extraire tous les gènes qui y sont encodés. Dans un premier temps, on écrira des fonctions polymorphes qui manipulent des listes de type `'a list`, et où les séquences START et STOP sont des listes quelconques non-vides. On pourra ensuite les instancier avec les valeurs qui nous intéressent, qui seront précisées plus loin.

Rappels. On rappelle qu'une *sous-liste* d'une liste `l` est une liste formée d'éléments consécutifs dans `l`. Par exemple `[3; 4]` est sous-liste de `[1; 2; 3; 4]`.

Rappelons également qu'une liste `pre` est *préfixe* d'une liste `l` s'il existe une liste `suf` telle que `l = pre @ suf`. Par exemple `[1; 2]` est préfixe de `[1; 2; 3; 4]`. La liste vide est par définition préfixe de toute liste.

Exercice 3.4. (⋆) Écrire une fonction

```
cut_prefix : 'a list -> 'a list -> 'a list option
```

qui prend en entrée une liste `pre` et une liste `l`, et renvoie :

- `None`, si la liste `pre` n'est pas un préfixe de `l`.
- `Some(suf)`, si `l = pre @ suf`.

Par exemple :

```
cut_prefix [A; G; T] [A; G; T; C; A] = Some ([C; A])
cut_prefix [1; 2; 3] [1; 2; 3] = Some ([])
cut_prefix [1; 2; 3] [1; 2] = None
cut_prefix ['b'; 'c'] ['a'; 'b'; 'c'; 'd'] = None
```

Exercice 3.5. ($\star$) Écrire une fonction

```
first_occ : 'a list -> 'a list -> ('a list * 'a list) option
```

qui prend en entrée une liste `slice` et une liste `l`, et renvoie :

- `None`, si `slice` n'est pas sous-liste de `l`.
- `Some(before, after)`, si `slice` est sous-liste de `l`, si `before` est la plus grande sous-liste de `l` qui précède la *première occurrence* de `slice` dans `l`, et si `after` est la plus grande sous-liste qui suit la cette *première occurrence*. On doit donc avoir `l = before @ slice @ after`.

Par exemple :

```
first_occ [A; G] [A; A; A; G; T; C] = Some ([A; A], [T; C])
first_occ [A; A] [A; A; A; G; T; C] = Some ([], [A; G; T; C])
first_occ [A; T] [A; A; A; G; T; C] = None
```

Exercice 3.6. ($\star$) Écrire une fonction

```
slices_between : 'a list -> 'a list -> 'a list -> 'a list list
```

qui prend en entrée trois listes `start`, `stop` et `l`, et qui renvoie la liste de toutes les sous-listes de `l` collectées de la manière suivante :

- on cherche la première occurrence de `start` dans la liste,
- si cette occurrence de `start` existe, on cherche la première occurrence de `stop` dans la sous-liste qui suit cette occurrence,
- si cette occurrence de `stop` existe, la sous-liste encadrée par les occurrences de `start` et `stop` fait partie de la liste résultat, et le traitement se poursuit récursivement sur la liste qui suit l'occurrence de `stop`.

Les listes `start` et `stop` seront supposées toutes les deux non vides. Par exemple :

Si `l = start @ l1 @ start @ l2 @ stop` où l'occurrence finale de `stop` est la seule occurrence de `stop` dans `l`, alors
`slices_between start stop l = [l1 @ start @ l2]`,
mais la sous-liste `l2` ne fait pas partie du résultat final.

```
slices_between [3; 3] [4; 4] [1; 1; 2; 3; 3; 1; 4; 1; 2]
 = []
```

```
slices_between [1; 2] [4; 1] [1; 1; 2; 3; 2; 1; 4; 1; 2]
 = [[3; 2 ;1]]
slices_between [A] [G] [A; C; T; G; G; A; C; T; A; T; G; A; G]
 = [[C; T]; [C; T; A; T]; []]
```

On souhaite à présent écrire une fonction qui, étant donné un brin d'ADN, renvoie la liste de ses gènes. Un codon contient trois bases. L'unique codon START est `[A; T; G]`. Trois codons STOP sont possibles, mais nous nous limiterons ici à un seul choix de codon, `[T; A; A]`.

Exercice 3.7. (⋆⋆) Écrire une fonction

```
cut_genes : dna -> dna list
```

qui prend en entrée un brin d'ADN, et renvoie la liste de ses gènes, c'est-à-dire les sous-listes de la liste représentant ce brin encadrées par les listes START et STOP extraites par la fonction précédente.

Par exemple, en exécutant le code suivant :

```
let strand = dna_of_string
   "ATGCCTGGGCATTGAGATCATTGGCACCCTGCATAAGATGTGTGAC
   TGTAGAGCTCTTCCTGACCATGCATAAAGAATG.CCAATGGCACAGC
   TGGTATC..TTTGCCATAAATGGCTCCTGGTGGAGCTGATAGTCACT
   TTCCATAATTAATGCATGGTGGTGGAGTTATTCTTGACTTTCCATAA"

let genes = List.map (string_of_dna) (cut_genes strand)
```

on obtient :

```
val genes : string list =
["CCTGGGCATTGAGATCATTGGCACCCTGCA";
 "TGTGACTGTAGAGCTCTTCCTGACCATGCA";
 ".CCAATGGCACAGCTGGTATC..TTTGCCA";
 "GCTCCTGGTGGAGCTGATAGTCACTTTCCA";
 "CATGGTGGTGGAGTTATTCTTGACTTTCCA"]
```

3.3 Calcul de séquences consensus

L'objectif de cette section est de calculer la *séquence consensus* d'une liste de gènes. Une liste de gènes sera représentée en OCaml par le type `dna list` : c'est donc une liste de listes de bases nucléiques. De plus, on supposera ici que dans une telle liste, les gènes ont été préalablement alignés, c'est-à-dire qu'ils sont tous de même longueur.

L'idée d'une séquence consensus est, pour chaque position, de chercher quelle base nucléique apparaît le plus souvent parmi les gènes considérés. Pour chaque position donnée, le consensus peut être de trois types :

- *Consensus total* : tous les gènes de la liste ont la même base à cette position.
- *Consensus partiel* : le nombre d'occurrences de l'une des bases est strictement supérieur à celui des autres bases.
- *Pas de consensus* : aucune base n'apparaît plus souvent que les autres à cette position.

Un exemple est représenté dans la figure ci-dessous. Cinq gènes sont alignés (un par ligne). La dernière ligne représente la séquence consensus de ces gènes. Pour chaque position, un consensus total est représenté par le symbole `!`, un consensus partiel est représenté par le symbole $\star$, et une absence de consensus est représentée par une espace. Lorsqu'il y a un consensus (total ou partiel), la base surlignée en bleu est la plus fréquente à cette position.

```
AQP1nuc.SEQ  CCTGGGCATTGAGATCATTGGCACCCTGCA   443
AQP2nuc.SEQ  TGTGACTGTAGAGCTCTTCCTGACCATGCA   419
AQP3nuc.SEQ  .CCAATGGCACAGCTGGTATC..TTTGCCA   424
AQP4nuc.SEQ  GCTCCTGGTGGAGCTAATAATCACTTTCCA   506
AQP5nuc.SEQ  CATGGTGGTGGAGTTAATCTTGACTTTCCA   422
consensus    **** **** *!!*! *! ** ******!!
```

On représentera le résultat d'un calcul de consensus à une position donnée à l'aide du type polymorphe OCaml suivant :

```
type 'a consensus =
  | Full of 'a
  | Partial of 'a * int
  | No_consensus
```

Le choix du polymorphisme permettra l'écriture de fonctions génériques qui ne seront spécialisées en choisissant le type `base` comme valeur de `'a` que dans les dernières questions. Le type `'a list` sera quant à lui spécialisé à `dna = base list`.

Exercice 3.8. ($\star$) Écrire une fonction

```
consensus : 'a list -> 'a consensus
```

qui, étant donnée une liste `l`, calcule le *consensus* de ses valeurs, défini comme :

- `Full b` si tous les éléments de `l` sont égaux à `b`,
- `Partial (b, n)` si `b` est l'unique valeur apparaissant le plus grand nombre de fois dans `l` et si `b` apparaît exactement `n` fois dans `l`,

– `No_consensus` dans tous les autres cas.

Par exemple,

```
consensus [A; A; A; A] = Full A
consensus [C; C; T; C] = Partial (C, 3)
consensus [A; A; G; G; T] = No_consensus
consensus [] = No_consensus
```

Exercice 3.9. (⋆⋆) Écrire une fonction

```
consensus_sequence : 'a list list -> 'a consensus list
```

qui prend en entrée une liste de listes supposées de même longueur et calcule leur *séquence consensus* : la liste contenant, à chaque position, le consensus des valeurs situées à cette position dans chacune des listes.
Par exemple,

```
consensus_sequence [[A; A; A; A];
                    [A; A; A; T];
                    [A; A; T; T];
                    [A; T; T; T]]
= [Full A; Partial (A, 3); No_consensus; Partial (T, 3)]
```

4 Expressions Régulières

Avant d'effectuer les manipulations précédentes, il faut préalablement avoir importé une séquence ADN à partir d'un fichier formaté. Pour en extraire l'information pertinente, on peut utiliser les *expressions régulières* que vous avez déjà rencontrées en utilisant des outils tels que `sed`, `awk`, and `grep`.

Nous nous intéresserons ici au problème de la *reconnaissance* par une expression régulière, c'est-à-dire le fait de répondre à la question : "est-ce que tel mot est reconnu par telle expression régulière ?" Ce problème étant non trivial, nous nous limiterons au cas où l'ensemble des mots reconnus par une expression est fini.

4.1 Préliminaires

Notons Σ un ensemble de lettres appelé *alphabet*. Un *mot sur* Σ est une liste finie d'éléments de Σ. En particulier, [] désigne le mot vide.

L'ensemble des *expressions régulières sur* Σ est défini inductivement par :

– ε est une expression régulière ;

– tout caractère c de Σ est une expression régulière ;

- le joker `?` est une expression régulière ;
- si e_1 et e_2 sont deux expressions régulières, la concaténation $e_1 \cdot e_2$ et l'alternative $e_1 + e_2$ sont des expressions régulières ;
- si e est une expression régulière, $e^\star$ est une expression régulière.

Formellement :

$$e ::= \varepsilon \mid c \mid \texttt{?} \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^\star$$

Pour tout ensemble de mots X et tout entier n, on note X^n l'ensemble des mots formés de la concaténation de n éléments de X (où @ désigne comme en OCaml la concaténation des listes).

$$\begin{aligned} X^0 &= \{[\,]\} \\ X^{n+1} &= \{w_1 @ w_2 \mid w_1 \in X \wedge w_2 \in X^n\} \end{aligned}$$

Étant donnée une expression régulière e, le *langage reconnu par* e, noté $\mathcal{L}(e)$, est défini par :

$$\begin{aligned} \mathcal{L}(\varepsilon) &= \{[\,]\} \\ \mathcal{L}(c) &= \{[c]\} \\ \mathcal{L}(\texttt{?}) &= \{[c] \mid c \in \Sigma\} \\ \mathcal{L}(e_1 \cdot e_2) &= \{w_1 @ w_2 \mid w_1 \in \mathcal{L}(e_1) \wedge w_2 \in \mathcal{L}(e_2)\} \\ \mathcal{L}(e_1 + e_2) &= \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \\ \mathcal{L}(e^\star) &= \bigcup_{0 \leqslant n} \mathcal{L}(e)^n \end{aligned}$$

Enfin, une expression régulière e *reconnaît* le mot w lorsque w appartient au langage reconnu par e. Il vient donc :

- ε reconnaît le mot vide $[\,]$;
- c reconnaît le singleton $[c]$;
- `?` reconnaît tous les singletons $[c]$ pour c une lettre de l'alphabet Σ ;
- $e_1 \cdot e_2$ reconnaît tout mot formé de la concaténation d'un mot reconnu par e_1 et d'un mot reconnu par e_2 ;
- $e_1 + e_2$ reconnaît tout mot reconnu par e_1 ou reconnu par e_2 ;
- $e^\star$ reconnaît tout mot formé de la concaténation d'une suite finie de mots reconnus par e.

On remarquera que l'ensemble $\mathcal{L}(e)$ n'est pas nécessairement fini. Par exemple, pour tout $c \in \Sigma$, l'ensemble $\mathcal{L}(c^\star)$ contient les mots $[\,], [c], [c; c], [c; c; c] \ldots$

4.2 Représentation en OCaml

Les expressions régulières seront représentées par le type algébrique suivant :

```
type 'a expr =
  | Eps
  | Base of 'a
  | Joker
  | Concat of 'a expr * 'a expr
  | Alt of 'a expr * 'a expr
  | Star of 'a expr
```

Le type `'a expr` est polymorphe sur le type des caractères : `'a` représente le type des lettres de l'alphabet. Nous utiliserons souvent l'instanciation `char expr`. On fournit une fonction :

```
expr_to_string : char expr -> string
```

convertissant une expression de type `char expr` en chaîne de caractères N'hésitez à l'utiliser pour déboguer dans votre programme ou directement dans le toplevel.

```
# print_endline (expr_to_string (Alt (Base 'a', Base 'b')));;
(a + b)
- : unit = ()
```

Exercice 4.1. Écrire une fonction

```
repeat : int -> 'a list -> 'a list
```

telle que `repeat n w` renvoie le mot `w` concaténé `n` fois avec lui-même.

```
# repeat 3 ['a'; 'b'] ;;
- : char list = ['a'; 'b'; 'a'; 'b'; 'a'; 'b']
```

Exercice 4.2. Écrire une fonction

```
expr_repeat : int -> 'a expr -> 'a expr
```

telle que `expr_repeat n e` renvoie une expression régulière qui reconnaît les mots formés de la concaténation de `n` mots reconnus par `e`.

```
# expr_repeat 3 (Base 'a') ;;
- : expr = Concat (Base 'a', Concat (Base 'a', Base 'a'))
```

Exercice 4.3. ($\star$) Écrire une fonction

```
is_empty : 'a expr -> bool
```

telle que `is_empty e` renvoie `true` si et seulement si le langage reconnu par `e` ne contient que le mot vide. À noter que `e` n'est pas nécessairement `Eps`.

```
# is_empty Eps ;;
- : bool = true
# is_empty (Concat (Eps, Eps)) ;;
- : bool = true
# is_empty (Star Eps) ;;
- : bool = true
# is_empty (Base 'a') ;;
- : bool = false
```

Exercice 4.4. ($\star$) Écrire une fonction

```
null : 'a expr -> bool
```

telle que `null e` renvoie `true` si et seulement si le mot vide est reconnu par `e`.

```
# null Eps ;;
- : bool = true
# null (Concat (Base 'a', Eps)) ;;
- : bool = false
# null (Alt (Base 'a', Eps)) ;;
- : bool = true
```

4.3 Reconnaissance de langages finis

La définition formelle du langage reconnu par une expression n'est pas facilement exploitable : comme nous l'avons déjà indiqué, l'ensemble $\mathcal{L}(e)$ peut être infini. Nous nous concentrons ici sur la reconnaissance des *langages finis*. Les fonctions de cette partie peuvent librement utiliser celles de la précédente.

Dans toute la suite, nous représenterons les ensembles finis (de mots, ou plus généralement les ensembles finis d'éléments de type `'a` dans les fonctions génériques) par des *listes triées sans duplicata*. Vous pouvez utiliser la fonction `union_sorted : 'a list -> 'a list -> 'a list` vue en TD, qui fait l'«union» de deux ensembles représentés par des listes triées. Vous pouvez aussi utiliser la fonction `sort_uniq : 'a list -> 'a list` qui renvoie son argument trié et sans duplicata. Vu le coût de cette fonction, attention à ne pas l'utiliser trop souvent.

Exercice 4.5. ($\star$) Écrire une fonction

```
is_finite : char expr -> bool
```

telle que `is_finite e` renvoie `true` si et seulement si le langage reconnu par `e` est fini.

```
# is_finite Eps ;;
- : bool = true
# is_finite (Concat (Base 'a', Joker)) ;;
- : bool = true
# is_finite (Star (Base 'a')) ;;
- : bool = false
```

Exercice 4.6. ($\star$) Écrire une fonction

```
product : 'a list list -> 'a list list -> 'a list list
```

telle que `product l1 l2` renvoie l'ensemble des mots formés de la concaténation d'un mot de `l1` et d'un mot de `l2`.

```
# product [[];['a']] [['b']] ;;
- : char list list = [['b']; ['a';'b']]
# product [['a'];['b']] [['c']; ['d']];;
- : char list list = [['a';'c']; ['a';'d']; ['b';'c']; ['b';'d
   ']]
```

Exercice 4.7. ($\star\star$) Écrire une fonction

```
enumerate : char list -> char expr -> char list list option
```

telle que si `e` est une expression sur l'ensemble fini de lettres `alphabet`, alors `enumerate alphabet e` renvoie : `Some l` où `l` est le langage reconnu par `e` si ce langage est fini ; `None` si ce langage est infini.

```
# enumerate ['a'; 'b'; 'c'] Eps
- : char list list option = Some [[]]
# enumerate ['a'; 'b'; 'c'] (Concat (Base 'a', Joker));;
- : char list list option = Some [['a'; 'a']; ['a'; 'b']; ['a';
    'c']]
# enumerate ['a'; 'b'; 'c'] (Star (Base 'a'));;
- : char list list option = None
```

Exercice 4.8. Écrire une fonction

```
alphabet_expr : 'a expr -> 'a list
```

telle que `alphabet_expr e` renvoie l'ensemble (la liste triée sans duplicata) des lettres apparaissant dans `e`.

```
# alphabet_expr Eps;;
- : char list = []
# alphabet_expr (Star (Concat (Base 'a', Joker))) ;;
- : char list = ['a']
# alphabet_expr (Alt (Base 'a', Base 'b')) ;;
- : char list = ['a'; 'b']
```

Exercice 4.9. Écrire une fonction

```
accept_partial : char expr -> char list -> answer
```

où `answer` est le type suivant

```
type answer = Infinite | Accept | Reject
```

et pour toute expressions `e` et tout mot `w`, `accept_partial e w` renvoie

- `Infinite` si le langage reconnu par `e` est infini,
- `Accept` si le langage reconnu par `e` est fini et contient le mot `w`,
- `Reject` si le langage reconnu par `e` est fini et ne contient pas `w`.

Dans cette fonction, ce qu'on considérera comme l'alphabet de `e` sera l'union des ensembles des lettres apparaissant dans l'expression `e` et dans le mot `w`.

```
# accept_partial (Star (Base 'a')) ['a'] ;;
- : answer = Infinite
# accept_partial (Concat (Base 'a', Joker)) ['a'; 'b'] ;;
- : answer = Accept
# accept_partial (Concat (Base 'a', Joker)) ['b'; 'a'] ;;
- : answer = Reject
```

5 Usage de GitLab

La création d'un depôt git sur le GitLab de l'UFR permettra à votre groupe de disposer d'un dépôt commun de fichiers sur ce serveur. Chaque membre du groupe pourra ensuite disposer d'une copie locale de ces fichiers sur sa machine, les faire évoluer, puis sauvegarder les changements jugés intéressants et les synchroniser sur

le serveur. Si vous ne vous êtes pas déjà servi de GitLab dans d'autres matières, cette section décrit son usage le plus élémentaire. Pour plus d'informations sur git et GitLab, il existe de multiples tutoriels en ligne. Nous contacter rapidement en cas de problèmes.

Accès au serveur et configuration personnelle. Se connecter via l'interface web : `https://gaufre.informatique.univ-paris-diderot.fr`. Utilisez pour cela les mêmes nom et mots de passe que sur les machines de l'UFR, et pas vos compte "ENT" de paris diderot ou u-paris. Cliquer ensuite sur l'icone en haut à droite, puis sur "Settings". A droite, aller ensuite dans la section "SSH Keys", et ajouter ici la partie public de votre clé ssh (ou de vos clés si vous en avez plusieurs). Cela facilitera grandement l'accès ultérieur à votre dépôt git, et vous évitera de taper votre mot de passe à chaque action.

Si vous n'avez pas encore de clé ssh, s'en générer une sur sa machine. L'usage de ssh n'est pas spécifique à git et GitLab, et permet des connections "shell" à des machines distantes. Si vous n'utilisez pas encore ssh et les clés publiques/privées ssh, il est temps de s'y mettre ! Pour plus d'information sur ssh, consulter :

`http://www.informatique.univ-paris-diderot.fr/wiki/doku.php/wiki/howto_connect`

Création du dépôt. Pour ce projet, nous vous fournissons quelques fichiers initiaux. Votre dépôt git sera donc un dérivé (ou "fork") du dépôt publique du cours. En pratique :

1. L'un des membres de votre groupe se rend sur la page du cours
   https://gaufre.informatique.univ-paris-diderot.fr/Bernardi/project-pf5/
   s'identifie si ce n'est pas déjà fait, et appuie sur le bouton "fork" (vers le haut, entre "Star" et "Clone"). Attention, un seul "fork" par groupe suffit.
2. Ensuite, aller dans la section "Settings" en bas à gauche, défiler un peu et cliquer sur "Visibility", et selectionner "Private" comme "projet visibility", puis "Save changes" un peu plus bas. Vérifier qu'un cadenas apparaît maintenant à côté de `pf5` quand vous cliquez sur "Projet" en haut à gauche.
3. Toujours dans "Settings" en bas à gauche, mais sous-section "Members" maintenant. "Invitez" votre collègue de projet, ainsi que les logins `nobrakal`, `allainc`, `bernardi`, `padovani`, `ledent`, `kesner`, en choissisant "Maintainer" comme rôle.
4. Voilà, votre dépôt sur le GitLab est prêt !

Création et synchronisation de vos copies locales de travail. Chaque membre du projet "clone" le dépôt du projet sur sa propre machine, c'est-à-dire en télécharge une copie locale : `git clone` suivi de l'addresse du projet tel qu'il apparaît dans l'onglet "Clone" sur la page du projet, champ en "SSH". Pour cela, il faut avoir installé `git` et `ssh` et configuré au moins une clé ssh dans GitLab.

Une fois le dépôt créé et cloné et en se plaçant dans le répertoire du dépôt, chaque membre pourra à tout moment :

- télécharger en local la version la plus récente du dépôt distant sur Gitlab : `git pull`.
- téleverser sa copie locale modifiée sur GitLab : `git push`.

Avant toute synchronisation, il est demandé d'avoir une copie locale "propre" (où toutes les modifications sont enregistrées dans des "commits").

Modifications du dépôt : les commits. Un dépôt Git est un répertoire dont on peut sauvegarder l'historique des modifications. Chaque action de sauvegarde est appelée une *révision* ou "commit". L'*index* du dépôt est l'ensemble des modifications qui seront sauvegardées à la prochaine révision. La commande

`git add` suivi du nom d'un ou plusieurs fichiers

permet d'ajouter à l'index toutes les modifications faites sur ces fichiers. Si l'un d'eux vient d'être crée, on ajoute dans ce cas à l'index l'opération d'ajout de ce fichier au dépôt. La même commande suivie d'un nom de répertoire ajoute à l'index l'opération d'ajout du répertoire et de son contenu au dépôt. La révision effective du dépôt se fait par la commande

`git commit -m` suivi d'un message entre guillemets doubles.

Invocable à tout instant, la commande

`git status`

permet d'afficher l'état courant du dépôt depuis sa dernière révision : quels fichiers ont été modifiés, renommés, effacés, crées, etc., et lesquelles de ces modifications sont dans l'index. Elle indique également comment rétablir l'état d'un fichier à celui de la dernière révision, ce qui est utile en cas de fausse manœuvre.

Les commandes `git mv` et `git rm` se comportent comme `mv` et `rm`, mais ajoutent immédiatement les modifications associées du répertoire à l'index.

Il est conseillé d'installer et d'utiliser les interfaces graphiques `gitk` (visualisation de l'arbre des commits) et `git gui` (aide à la création de commits).

Une dernière chose : git est là pour vous aider à organiser et archiver vos divers fichiers sources. Par contre il vaut mieux ne *pas* y enregistrer les fichiers issues de compilations (binaires, répertoire temporaire tels que `_build` pour `dune`, fichiers objets OCaml `*.cm{o,x,a}`, etc).

Les fusions (merge) et les conflits. Si vous êtes plusieurs à modifier vos dépôts locaux chacun de votre côté, celui qui se synchronisera en second avec votre dépôt GitLab commun aura une manoeuvre nommé "merge" à effectuer. Tant que vos modifications respectives concernent des fichiers ou des zones de code différentes, ce "merge" est aisé, il suffit d'accepter ce que git propose, en personnalisant éventuellement le message de merge. Si par contre les modifications se chevauchent et sont incompatibles, il y a alors un conflit, et git vous demande d'aller décider quelle version est à garder. Divers outils peuvent aider lors de cette opération, mais au plus basique il s'agit d'aller éditer les zones entre `<<<<<` et `>>>>>` puis faire `git add` et `git commit` de nouveau.

Intégrer les modifications venant du dépôt du cours. Si le dépôt du cours reçoit ultérieurement des correctifs ou des évolution des fichiers fournis pour le projet, ces modifications peuvent être intégrés à vos dépôts.

- La première fois, aller dans votre répertoire de travail sur votre machine, et taper :

  ```
  git remote add prof \
  https://gaufre.informatique.univ-paris-diderot.fr/
  Bernardi/project-pf5/.git
  ```

- Ensuite, à chaque fois que vous souhaiterez récupérer des commits du dépôt du cours :

  ```
  git pull prof master
  ```

- Selon les modifications récupérées et les votres entre-temps, cela peut occasionner une opération de "merge" comme décrite auparavant.

- Enfin, ces modifications sont maintenant intégrés à votre copie locale de travail, il ne reste plus qu'à les transmettre également à votre dépôt sur GitLab :

  ```
  git push
  ```

Les branches. Il est parfois pratique de pouvoir essayer différentes choses, même incompatibles. Pour cela, Git permet de travailler sur plusieurs exemplaires d'un même dépôt, des *branches*. Un dépôt contient toujours une branche principale, la branche "master", dont le rôle est en principe de contenir sa dernière version stable. Les autres branches peuvent servir à développer des variantes de la branche master, par exemple pour tenter de corriger un bug sans altérer cette version de référence. La création d'une nouvelle branche, copie conforme de la branche courante – initialement, master – dans son état courant, se fait par :

`git branch` suivi du nom choisi pour la branche.

Sans arguments, cette commande indique la liste des branches existantes, ainsi que celle dans laquelle se trouve l'utilisateur. Le passage à une branche se fait par

`git checkout` suivi du nom de la branche.

Pour ajouter au dépôt distant une branche qui n'est pas encore sur celui-ci, après s'être placé dans la branche :

`git push -set-upstream origin` suivi du nom de la branche

Un `push` depuis une branche déjà sur le serveur se fait de la manière habituelle. Enfin, on peut "réunifier" deux branches avec `git merge`, voir la documentation pour plus de détails.

Noter que GitLab propose également un mécanisme de "Merge Request" : il permet de proposer des modifications, soit à son propre projet, soit au projet qui a été "forké" à l'origine, les membres du projet en question pouvant alors accepter ou non ces suggestions après discussion.

Références

[1] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

[2] L. Ionkov and B. Settlemyer. Dna : The ultimate data-storage solution. https://www.scientificamerican.com/article/dna-the-ultimate-data-storage-solution/, 2021.

[3] Cockell C. S. Landenmark H. K. E., Forgan D. H. An estimate of the total dna in the biosphere. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4466264/, 2015.