Dans l'ensemble du TP on manipule des expressions régulières avec le type :

```
type 'a regex =
     | Vide
     | Epsilon
     | Lettre of 'a
     | Concat of 'a regex * 'a regex
     | Union of 'a regex * 'a regex
     | Etoile of 'a regex;;
```

Le fichier `regex.ml` (trouvable sur le site `https://github.com/nathaniel-carre/MPI-LLG`) contient la définition de ce type, ainsi qu'une fonction `parse : string -> char regex` qui permet de transformer une chaîne de caractère en une expression régulière, avec la convention que le symbole $\varnothing$ est représenté par `'#'` et que le symbole ε est représenté par `'&'`. Voici des exemples d'utilisation de la fonction :

```
# parse "a*|bc**(b|&)";;
- : char regex = Union (Etoile (Lettre 'a'),
                        Concat (Lettre 'b', Concat (Etoile (Etoile (Lettre 'c')),
                                                    Union (Lettre 'b', Epsilon))))
# parse "a|b|c|#";;
- : char regex = Union (Lettre 'a', Union (Lettre 'b', Union (Lettre 'c', Vide)))
# parse "(a|b)*(ab|ba)*";;
- : char regex = Concat (Etoile (Union (Lettre 'a', Lettre 'b')),
                         Etoile (Union (Concat (Lettre 'a', Lettre 'b'),
                                        Concat (Lettre 'b', Lettre 'a'))))
```

On rappelle qu'on peut interpréter le fichier par la commande (en remplaçant le chemin du fichier) :

```
#use "Chemin/du/fichier/regex.ml";;
```

Il est également possible d'inclure le fichier à la compilation, via la commande (avec des options éventuelles) `ocamlc regex.ml TP6.ml`. Dans ce cas, il faudra écrire `Regex.parse` pour appeler la fonction `parse`.

Toutes les fonctions seront testées au moins sur les expressions régulières $e_0 = (a|ba^\star)^\star|ab^\star a$ et $e_1 = a(a|b)^\star b$ (mais vous êtes invités à faire d'autres tests).

1 Manipulation des expressions régulières

Exercice 1

1. Montrer que toute expression régulière est équivalente à une expression régulière qui est soit $\varnothing$, soit une expression régulière ne contenant pas le symbole $\varnothing$.
2. Écrire une fonction `normaliser : 'a regex -> 'a regex` qui transforme une expression régulière en une expression régulière de l'une des deux formes précédentes.
3. En déduire une fonction `est_vide : 'a regex -> bool` qui détermine si l'interprétation d'une expression régulière est le langage vide ou non.

2 Automate de Glushkov

Exercice 2

1. Écrire une fonction `nombre_lettres : 'a regex -> int` qui compte le nombre de lettres (distinctes ou non) qu'une expression régulière contient.
2. Écrire une fonction `lineariser : char regex -> int regex * char array` qui prend en argument une expression régulière e sur des caractères et renvoie un couple formé d'une expression régulière f et d'un tableau `t` tel que :
   - f est l'expression e linéarisée : ses « lettres » sont des entiers tous distincts, consécutifs en partant de zéro ;
   - le tableau `t` indique la numérotation des lettres : la lettre numérotée i dans f correspond à la lettre `t.(i)` dans e.

Pour e une expression régulière, on note $V(e) = \mathcal{L}(e) \cap \{\varepsilon\}$.

3. Rappeler les formules inductives permettant de calculer $V(e)$, $P(e)$, $S(e)$ et $F(e)$ pour e une expression régulière.

   Pour la suite, on choisit de représenter ces ensembles dans un alphabet $\Sigma = [\![0, |\Sigma|-1]\!]$ de la manière suivante :

- $V(e)$ par un booléen qui vaut `true` si $V(e) = \{\varepsilon\}$ et `false` si $V(e) = \emptyset$;
- $P(e)$ par un tableau de booléens `pe` de taille $|\Sigma|$ tel que `pe.(a)` vaut `true` si et seulement si $a \in P(e)$ (et $S(e)$ est représenté de manière similaire) ;
- $F(e)$ par une matrice de booléens `fe` de taille $|\Sigma| \times |\Sigma|$ telle que `fe.(a).(b)` vaut `true` si et seulement si $ab \in F(e)$.

4. Écrire une fonction `mot_vide : int regex -> bool` qui calcule $V(e)$.
5. Écrire une fonction `union : bool array -> bool array -> bool array` qui prend en argument deux tableaux de booléens de même taille représentant des ensembles et renvoie leur union.
6. Écrire une fonction `prefixes : int -> int regex -> bool array` qui prend en argument un entier n et une expression e et calcule $P(e)$, en supposant que la taille de l'alphabet est n. Écrire de même une fonction `suffixes : int -> int regex -> bool array`.
7. Écrire une fonction `facteurs : int -> int regex -> bool array array` qui calcule $F(e)$.

Pour la suite de cette partie, on représente un automate fini non déterministe par le type suivant :

```ocaml
type 'a afnd = { finaux : bool array;
                 delta : ('a * int) list array };;
```

tel que si `aut` est un objet de type `'a afnd` représentant un AFND $A = (Q, \Sigma, \Delta, I, F)$, alors :

- $Q = [\![0, |Q|-1]\!]$;
- $I = \{0\}$;
- `aut.finaux` est un tableau de booléens de taille $|Q|$ tel que `aut.finaux.(q)` vaut `true` si et seulement si $q \in F$;
- `aut.delta` est un tableau de taille $|Q|$ tel que `aut.delta.(q)` est une liste contenant les couples (a, p) tels que $p \in \Delta(q, a)$.

8. Écrire une fonction `glushkov : 'a regex -> 'a afnd` qui applique l'algorithme de Berry-Sethi et construit l'automate de Glushkov associé à une expression régulière.

Exercice 3

1. Écrire une fonction `delta_ens : 'a afnd -> bool array -> 'a -> bool array` qui prend en argument un AFND, un tableau de booléens représentant une partie $X \subseteq Q$ et une lettre a de Σ et renvoie $\Delta(X, a)$ sous forme de tableau de booléens.
2. En déduire une fonction `delta_etoile : char afnd -> bool array -> string -> bool array` qui calcule $\Delta^*(X, u)$, pour u une chaîne de caractères.
3. En déduire une fonction `reconnu : char afnd -> string -> bool` qui teste l'appartenance d'un mot au langage d'un automate non déterministe.
4. Tester les automates de Glushkov des expressions régulières e_0 et e_1 sur des mots de votre choix.

3 Automate de Thompson

Soit A un AFND avec ε-transitions. On dit que A est **normalisé** si et seulement si :
- A a un unique état initial et un unique état final qui sont distincts ;
- A est standard ; aucune transition ne sort de l'état final ;
- chaque état est soit l'origine d'au plus une transition étiquetée par une lettre de Σ, soit l'origine d'au plus deux transitions étiquetées par ε.

Exercice 4

1. Déterminer des automates normalisés reconnaissant les langages $\emptyset$, ε et $\{a\}$, pour $a \in \Sigma$.
2. Soit A et B deux automates normalisés. Donner la forme d'automates normalisés reconnaissant :
   - $L(A) \cup L(B)$;
   - $L(A)L(B)$;
   - $L(A)^*$.

   On représente un automate normalisé avec le type suivant :

```
type 'a etat = { lettre : 'a option;
                 mutable sortie1 : 'a etat option;
                 mutable sortie2 : 'a etat option };;

type 'a afndn = { initial : 'a etat;
                  final : 'a etat};;
```

   L'idée est la suivante : chaque état garde en mémoire les transitions sortantes. Un automate fini non déterministe normalisé possède un unique état initial et un unique état final. Pour chaque état `q`, on distingue :
   - si `q` n'a pas de transition sortante, alors `q.lettre`, `q.sortie1` et `q.sortie2` valent tous les trois `None` ;
   - si `q` a une unique transition sortante étiquetée par la lettre `a`, vers un état `p`, alors `q.lettre` vaut `Some a`, `q.sortie1` vaut `Some p` et `q.sortie2` vaut `None` ;
   - si `q` a une unique transition sortante étiquetée par ε, vers un état `p`, alors `q.lettre` vaut `None`, `q.sortie1` vaut `Some p` et `q.sortie2` vaut `None` ;
   - si `q` a deux transitions sortantes étiquetée par ε, vers des états `p1` et `p2`, alors `q.lettre` vaut `None`, `q.sortie1` vaut `Some p1` et `q.sortie2` vaut `Some p2`.

   Les champs `sortie1` et `sortie2` sont mutables pour faciliter la construction.
3. Écrire une fonction `thompson : 'a regex -> 'a afndn` qui construit l'automate de Thompson associé à une expression régulière.

Pour tester si un mot est reconnu, on propose une méthode différente de celle de l'exercice 3. Étant donné qu'il y a au plus deux transitions sortantes par état, on propose, pour un mot $u \in \Sigma^*$, de calculer par backtracking l'existence d'un chemin étiqueté par u de l'état initial vers l'état final.

4. Écrire une fonction `reconnu_thompson : char afndn -> string -> bool` qui teste si un mot est reconnu par un automate normalisé. On testera l'égalité à l'état final avec l'opérateur `==` qui est le test d'identité (même adresse mémoire) et non l'opérateur `=` qui est le test d'égalité (même valeur, non défini pour un type enregistrement ici).
5. Quel problème peut se poser avec la fonction précédente, par exemple avec un automate de Thompson associé à $(a^\star)^\star$?
6. Proposer une modification de l'algorithme de Thompson pour éviter ce problème.