TP 2 : Langages réguliers, expressions régulières

Environnement

Nous travaillerons en mode interprété pour ce TP. Nous vous laissons le choix de l'environnement, selon vos habitudes. Vous pouvez utiliser :

- L'interpréteur en ligne BetterOCaml : https://BetterOCaml.ml/
- Emacs, muni du mode Tuareg.

Rappel

Ce TP va beaucoup mobiliser le module `List` de OCaml, dont la documentation est disponible ici :
https://v2.ocaml.org/api/List.html/
Si vous choisissez de travailler en mode compilé, on rappelle comment compiler avec `ocamlc` depuis un terminal : `ocamlc -o nom_executable nom_fichier1.ml nom_fichier2.ml ....` Puis vous pouvez lancer votre exécutable avec `./nom_executable`

Mots et opération basiques

Les fonctions demandées peuvent, pour beaucoup, être écrites très simplement à l'aide des fonctions de la documentation. Vous avez le droit de réinventer la roue, mais nous vous encourageons à utiliser la documentation...

1. Déclarez le type `'a mot`, qui correspond simplement au type d'une liste d'éléments de type `'a`.
2. Comment s'appelle un type comportant une inconnue tel que celui-ci ? Quel intérêt d'en utiliser un ?

   Tout au long du TP, vous veillerez à créer des exemples de mots que vous utiliserez pour tester toutes les fonctions qui vont suivre. Nous vous encourageons à utiliser des mots de différents types, dont voici quelques exemples :

   - des `char mot`, comme `['a';'b';'b';'a';'b']`
   - des `bool mot`, comme `[false;true;false;false]`, qu'on peut utiliser pour se représenter un alphabet à deux lettres.
   - vous pouvez même définir vos propres types. Par exemple, `type sigma = A | B` pour se représenter un alphabet à deux lettres $\Sigma = \{a, b\}$. Vous pouvez ensuite écrire des mots comme `[A;B;B;A;B]`.
3. Ecrivez deux fonctions `longueur : 'a mot -> int` et `est_vide : 'a mot -> bool`, qui respectivement, donne la longueur d'un mot et précise par un booléen si un mot est vide ou non.
4. Ecrivez une fonction `miroir : 'a mot -> 'a mot`, pour un mot donné, renvoie le même mot lu dans l'autre sens. Par exemple, pour l'appel `miroir ['a';'b';'b';'a';'b']` doit renvoyer le mot `['b','a','b','b','a']`. Quel est le type de ce mot d'exemple ? Quelle est la complexité de cette opération ?
5. Ecrivez une fonction `concat : 'a mot -> 'a mot -> 'a mot`, de sorte que l'appel `concat u v` renvoie le mot constitué des lettres de `u`, suivies de celles de `v`, dans le même ordre. Quelle est la complexité de cette opération ?

Sous-mots

6. Ecrivez la fonction `est_prefixe : 'a mot -> 'a mot -> bool`, de sorte que l'appel `est_prefixe u v` renvoie `true` si `u` est un préfixe de `v`, et `false` sinon.
7. Même question pour la fonction `est_suffixe : 'a mot -> 'a mot -> bool`, dont nous vous laissons inférer le sens. (Indice : soyez intelligent, utilisez les questions précédentes...)

8. Ecrivez la fonction `est_facteur : 'a mot -> 'a mot -> bool`, de sorte que l'appel `est_facteur u v` renvoie `true` si `u` est un facteur de `v`, et `false` sinon.
9. Ecrivez la fonction `est_sous_mot : 'a mot -> 'a mot -> bool`, de sorte que l'appel `est_sous_mot u v` renvoie `true` si `u` est un sous-mot de `v`, et `false` sinon.
10. Ecrivez la fonction `coupe : 'a mot -> int -> ('a mot * 'a mot)`, telle que l'appel `coupe u i` renvoie un couple `(v,w)` où `v` est le préfixe de longueur `i` de `u`, et où `u` est la concaténation de `v` et `w`. Quelle est sa complexité ?
11. A partir de la fonction précédente, écrivez deux fonctions `prefixe : 'a mot -> int -> 'a mot` et `suffixe : 'a mot -> int -> 'a mot` de sorte que `prefixe u i` (resp. `suffixe u i`) renvoie le préfixe (resp. suffixe) de `u` de longueur `i`.

Expressions régulières

12. Déclarez le type récursif `'a exp_reg`, qui permet de construire des expressions régulières sur un alphabet dont les éléments sont de type `'a`. Vous utiliserez les constructeurs `Vide`, `Epsilon`, `Symbole`, `Union`, `Concat` et `Etoile`, dont vous déterminerez vous-même l'arité et les données si nécessaire.
13. En exemple, construisez l'expression régulière $e = ab^* \mid ba^*$, en considérant a et b comme des symboles de type `char`, et l'expression régulière $e' = 10^* \mid 0$, à partir des éléments de type `int` 1 et 0.
14. (*) Ecrivez une fonction récursive `test : 'a mot -> 'a exp_reg -> bool`, de sorte que l'appel `test u e` renvoie `true` si `u` peut être engendré par `e`, et `false` sinon.
    Aide : Procédez par pattern-matching sur `e`. Les cas de la concaténation et de l'étoile seront traités de manière naïve.
15. (*) Estimez la complexité de la fonction précédente. [1]

(*)Application : recherche de motifs dans une liste de mots[2]

On s'intéresse dans cette partie à la construction d'expression régulières qui filtrent certains motifs. On souhaite tester tous les mots d'une liste donnée avec ces expressions régulières.

16. Ecrire une fonction `string_vers_mot : string -> char mot`, qui prend une chaîne de caractères m et la transforme en un mot de `char`, contenant les mêmes caractères dans le même ordre.
17. Récupérez le fichier `dinosaures.ml` dans le répertoire partagé de la classe. Vous y trouverez :
    - l'objet `list_of_dinosaurs` de type `string list`.
    - l'objet `alphabet`, de type `char exp_reg`. Il s'agit d'une expression régulière que nous fournissons, qui représente le langage $\{\texttt{A}, \ldots, \texttt{Z}, \texttt{a}, \ldots, \texttt{z}\}$

    Si vous travaillez en mode interprété, recopiez-en le contenu dans votre interpréteur. Si vous travaillez en mode compilé, n'oubiez pas de l'intégrer à votre commande de compilation.
18. Ecrivez une expression régulière de type `char exp_reg` exprimant tous les noms finissant par "raptor" (sans les guillemets). Utilisez la pour trouver tous les noms de raptor dans la liste.
19. Ecrivez une expression régulière `char exp_reg` exprimant tous les noms finissant par "saurus", et commençant par un A majuscule. Quels mots de la liste respectent ces critères ?
20. Ecrivez une expression régulière `char exp_reg` exprimant tous les noms commençant par "Mega", et ceux finissant par "saurus". Quels mots de la liste respectent ces critères ?

[1] Vous trouverez une complexité peu intéressante, qui vous fera remettre en question le choix d'expressions rationnelles pour tester si un mot appartient à un langage... Plus tard dans l'année, nous verrons un autre modèle : les automates finis. Ce modèle est équivalent à celui des expressions régulières, dans le sens où chaque langage engendré par une expression régulière peut être reconnu par un automate fini, et vice-versa. Utiliser un automate est cependant plus efficace du point de vue de la machine, là où les expressions régulières ont l'avantage de fournir une description simple d'un langage pour sa compréhension par l'humain.

[2] Adaptation de l'exercice fourni ici : https://www.normalesup.org/ doulcier/teaching/pythonEx03_regexp_dino.html