Evaluation d'expressions régulières

Devoir à rendre le 7 mai. Pénalité de 2 points par jour de retard.

17	18	19	20	21	(22)	23
24	25	26	27	28	29	30
1	2	3	4	5	6	(7)

Les devoirs sont en rendre par mail dans un fichier tarbal à ⟨lozes@lsv.ens-cachan.fr⟩.

Pour décider si un mot fini est reconnu par une expression régulière, l'approche la plus connue est de traduire l'expression régulière en un automate (déterministe ou non-déterministe), puis de tester si le mot est accepté par cet automate. Il existe de nombreuses façons de construire cet automate, la plus classique étant l'algorithme de Thomson-McNaughton-Yamada par induction sur l'expression régulière.

Le problème de l'approche de Thomason-McNaughton-Yamada est qu'elle peut supposer de générer complètement l'automate avant de commencer à tester si le mot est accepté. L'automate peut cependant être de taille exponentielle en la taille de l'expression régulière (considérer par exemple l'expression régulière $(a+b)^*a(a+b)^n a(a+b)^*$), alors que pour accepter un mot de longueur ℓ, on ne lira de toute façon qu'au plus ℓ états de cet automate.

D'autres approches permettent de construire un automate "à la volée", notamment les approches par expression régulières dérivées de Brzozowski et Antimirov, ou encore l'approche par expression régulières pointée de Glushkov. Ce devoir a pour but de voir (ou revoir) ces approches, de les implémenter, de les comparer en terme de performances, puis d'étendre la dernière à des expressions régulières pondérées.

L'archive fournie [1] contient les fichiers suivants

- Fichiers contenant des définitions générales (n'ont pas besoin d'être changés)
  - `Makefile` : le makefile pour toutes les parties
  - `regexp.ml` : expressions régulières
  - `wregexp.ml` : expressions régulières pondérées
- Fichiers à compléter

1. cf http://www.lsv.fr/~lozes/Enseignement/ProgAvancee/DM/src.tgz

— `dfa.ml` : automates déterministes (quelques lignes à rajouter, cf préliminaires)
— `test.ml` : les tests pour toutes les parties
— `part1.ml` : le fichier à compléter pour la partie 1
— `part2.ml` : le fichier à compléter pour la partie 2
— `part3.ml` : le fichier à compléter pour la partie 3

Préliminaires

On représente une expression régulière par le type de données suivant

```
type 'a regexp =
  | Eps                              (* le mot vide *)
  | Sym of 'a                        (* un mot-lettre *)
  | Alt of 'a regexp * 'a regexp     (* union *)
  | Seq of 'a regexp * 'a regexp     (* concatenation *)
  | Rep of 'a regexp                 (* iteration *)
```

où le type 'a est le type des lettres de l'alphabet considéré. On représente aussi un mot par la liste de ses lettres, et on va chercher à construire des accepteurs.

```
type 'a word = 'a list

type 'a acceptor = 'a regexp -> 'a word -> bool
```

Pour construire un accepteur, on va se baser sur des traductions d'expressions régulières en automates.

```
module type DFA = sig
  type 'a state
  val init : 'a regexp -> 'a state
  val next : 'a state -> 'a -> 'a state
  val final : 'a state -> bool
end
```

Complétez le fichier `dfa.ml` (votre code doit pouvoir tenir sur moins de cinq lignes).

Partie 1 : Expressions régulières dérivées

On rappelle que l'automate des résiduels associé au langage régulier L est l'automate déterministe dont les états sont les langages $w^{-1}L$ pour tous les $w \in \Sigma^*$.

Le principe de la construction par expressions régulières dérivées est de représenter les états de cet automate à l'aide d'expressions régulières et de calculer l'état qui succède à l'état/expression régulière e suite à la lecture de la lettre a par induction sur e. Par exemple, partant de l'expression régulière

$(aa+b)^*(a+\epsilon)b$, on pourrait obtenir une exécution de l'automate qui ressemble à ceci [2].

$$\begin{array}{ll} & (aa+b)^*(a+\epsilon)b \\ \xrightarrow{a} & a(aa+b)^*(a+\epsilon)b \ + \ b \\ \xrightarrow{a} & (aa+b)^*(a+\epsilon)b \\ \xrightarrow{b} & (aa+b)^*(a+\epsilon)b \ + \ \epsilon \end{array}$$

Il se peut que l'on rencontre un résiduel qui soit le langage vide. La définition choisie d'expression régulière ne permettant pas de représenter le langage vide, on utilisera le type suivant pour représenter un état de l'automate.

```
type 'a state = 'a regexp option
```

Complétez le module du fichier `part1.ml` puis utilisez ce module pour construire une fonction `accept` du type indiqué.

Ajoutez des tests dans le fichier `test.ml`, et proposez éventuellement des optimisations pour accélérer cette méthode.

Partie 2 : Expressions régulières pointées

Le principal défaut avec l'approche précédente, si l'on n'y prend pas garde, est qu'on peut générer des expressions régulières de tailles de plus en plus grandes. Pour éviter cela, on peut représenter un résiduel non pas par une expression régulière, mais par une expression régulière pointée. Une expression régulière pointée est une expression régulière dans laquelle on a "pointé" certaines lettres, intuitivement les lettres que l'on vient de lire et à partir desquelles il faut continuer de lire l'expression régulière. Si l'on reprend le même exemple que précédemment, cela donne :

$$\begin{array}{ll} & \underline{\cdot}(aa+b)^*(a+\epsilon)b \\ \xrightarrow{a} & (\underline{a}a+b)^*(\underline{a}+\epsilon)b \\ \xrightarrow{a} & (a\underline{a}+b)^*(a+\epsilon)b \\ \xrightarrow{b} & (aa+\underline{b})^*(a+\epsilon)\underline{b} \end{array}$$

On peut expliquer l'idée sous-jacente assez simplement en identifiant une expression régulière à un programme impératif jouet comme celui utilisé dans un cours de programmation : les lettres sont les instructions atomiques du programme, le + est un `if then else` non déterministe, la concaténation est la séquence de deux programmes, et l'étoile la boucle `while`.

Une expression régulière pointée peut être vue comme la représentation d'une continuation de ce programme : une lettre pointée correspond à un point de programme actif, et le non-déterminisme induit la possibilité d'avoir plusieurs points de programme actifs simultanément.

2. les expressions régulières effectivement calculées par induction peuvent être un peu plus verbeuses que celles données dans cet exemple

Notez que, dans l'exemple ci-dessus, pour la première expression régulière, on a souligné une "lettre fantôme" au début de l'expression régulière, ce qui correspond au point de programme initial. Une expression régulière pointée est donc donnée par le type

```
type 'a state =  bool * ('a * bool) regexp
```

Complétez le fichier `part2.ml` et testez votre code. Comparer l'efficacité de cette méthode à celle de la question précédente.

Partie 3 : Expressions régulières pondérées

Les expressions régulières pondérées sont une généralisation des expressions régulières où les expressions atomiques sont des fonctions qui associent un poids à chaque lettre de l'alphabet.

```
type ('sym,'weight) wregexp = ('sym -> 'weight) regexp
```

La sémantique d'une expression régulière pondérée e est un pondérateur $\widehat{e}$, qui associe à chaque mot u un poids $\widehat{e}(u)$.

```
type ('sym,'weight) weighter =
  ('sym,'weight) wregexp -> 'sym word -> 'weight
```

Pour définir le pondérateur $\widehat{e}$, il faut pouvoir combiner les poids, et pour cela équiper l'ensemble des poids d'une structure d'anneau.

```
module type SEMIRING = sig
  type t
  val sum : t list -> t
  val prod : t list -> t
end

module type WEIGHTER = functor (W:SEMIRING) -> sig
  val eval : ('a,W.t) weighter
end
```

On notera $\oplus, \otimes, 0, 1$ respectivement la somme, le produit, le neutre de la somme et le neutre du produit dans l'anneau W. Le pondérateur associé à une expression régulière pondérée sur l'anneau de poids W est défini par induction sur l'expression régulière comme suit.

- $\widehat{\epsilon}(u) = 1$ si $u = \epsilon$, sinon $\widehat{\epsilon}(u) = 0$.
- $\widehat{f}(u) = f(u)$ si u est un mot contenant un seul symbole, sinon $\widehat{f}(u) = 0$
- $\widehat{e_1 + e_2}(u) = \widehat{e_1}(u) \oplus \widehat{e_2}(u)$
- $\widehat{e_1.e_2}(u) = \bigoplus\{\widehat{e_1}(u_1) \otimes \widehat{e_2}(u_2) \mid u = u_1u_2\}$
- $\widehat{e^*}(u) = \bigoplus\{u_1 \otimes \cdots \otimes u_n \mid u = u_1 \dots u_n$ et $u_i \neq \epsilon$ pour tout i$\}$ si $u \neq \epsilon$, et $\widehat{e^*}(\epsilon) = 1$.

On retrouve alors la sémantique habituelle d'une expression régulière en la voyant comme une expression régulière pondérée dans l'anneau des Booléens où la somme est la disjonction et le produit la conjonction.

On peut aussi voir une expression régulière comme une expression régulière pondérée dans l'anneau des entiers, en interprétant l'expression régulière réduite à une lettre a comme le pondérateur $\widehat{a}(u) = 1$ si $u = a$, et $\widehat{a}(u) = 0$ sinon. Dans cette interprétation, si e est sans étoile, $\widehat{e}(u)$ correspond au nombre de façons de prouver que u est dans le langage de e. [3]

La définition par induction du pondérateur se traduit aisément en un algorithme permettant de le calculer. Cependant, cet algorithme n'est pas très efficace. Pour faire plus efficace, il est possible d'étendre l'approche de la partie 2.

Dans cette dernière partie, vous devrez

1. écrire un pondérateur de référence en suivant la définition inductive ci-dessus ;
2. écrire un autre pondérateur "efficace" qui généralise la construction étudiée dans la partie 2.
3. ajouter des tests dans `test.ml`, vérifier que les deux pondérateurs coïncident, et evaluer leur efficacité.

3. Si e contient des étoiles, on peut avoir en général une infinité de telles preuves, et $\widehat{e}(u)$ correspond alors au nombre de preuves qui n'utilisent que des découpages "sans ϵ inutiles", comme dans la fonction `parts` ci-dessus.