# Projet : match d'expressions rationnelles

Version du 26 septembre 2016

Vous allez implémenter en Objective Caml un ensemble d'algorithmes permettant de tester un mot sur une expression rationnelle. Ceci vous permettra de maîtriser quelques uns des algorithmes vus en cours en les traduisant en code, et en testant leur comportement.

Pour ce faire, le parseur de l'expression rationnelle et la structure de données représentant les automates vous sont fournis. On se limitera à l'alphabet suivant : les trois lettres 'a', 'b' et 'c'.

**Évaluation** Le travail réalisé sera évalué par une procédure automatique en testant la validité des réponses données pour différentes expressions rationnelles par difficulté croissante. Nous ne vous fournissons aucun jeu de test, vous êtes fortement incités à en créer un à partir des exemples vus en cours, TDs et TPs. Un outil permettant de lancer votre jeu de tests est fourni.

**Présentation de l'environnement** Comme pour les TPs, nous avons installé les outils nécessaires à l'emplacement `/u/prof/linard_a/public/thlr` :

— un compilateur Objective Caml récent (3.10),
— les bibliothèques nécessaires (OCaml-PCRE, OCamlGraph, CamlTemplate).

Vous trouverez les sources du projet dans le répertoire `projet`, qui contient les répertoires et fichiers suivants :

— `sujet.pdf` : le sujet que vous devez implémenter,
— `install` : un script qui installe ce qu'il faut sur votre compte,

Avant tout, lancez la commande : `/u/prof/linard_a/public/thlr/projet/install` dans un terminal. Cette commande crée un répertoire `/thlr/projet/` dans votre répertoire personnel et y installe les fichiers du projet.

**Script "build"** Le script `build` permet de réaliser les actions suivantes :

— `compile` : compile le projet,
— `clean` : nettoie le projet des fichiers créés par la compilation,
— `trace` : lance les algorithmes sur une expression rationnelle et un mot et retourne des informations de débogage,
— `test` : lance les algorithmes sur une expression rationnelle et un ensemble de mots dans un fichier,
— `documentation` : génére une documentation en HTML des modules qui vous sont fournis, et place cette documentation dans `html.docdir/index.html`.

**"build trace"** Ce programme lit sur son entrée standard une expression rationnelle en notation polonaise inverse, puis lit un mot, et exporte le résultat de chacun de vos algorithmes dans un fichier XML lisible par Vaucanson. Il lance Vaucanson sur ces différents fichiers XML et le même mot et vous indique si vous obtenez le même résultat. Si les résultats sont différents, une indication sur l'algorithme à partir duquel vous divergez de Vaucanson est donnée.

Un exemple d'expression rationnelle : `eab+c.*+` qui se traduit par : $\varepsilon + ((a + b)c)^*$. Epsilon est noté par la lettre `e`. Aucune parenthèse n'est nécessaire, et il ne faut aucun espace dans l'expression rationnelle.

**"build test"** Ce programme lit sur son entrée standard une expression rationnelle et un nom de fichier, qui doit contenir un mot à lire par ligne. Il exécute ensuite votre chaîne de traitements sur chacun des mots et enregistre les résultats pour les comparer avec ceux donnés par Vaucanson.

**Travail à réaliser** Vous devez implémenter la chaîne complète des algorithmes ci-dessous, permettant de vérifier si un mot est reconnu ou non par une expression rationnelle. L'évaluation nécessitant l'implémentation de tous les algorithmes demandés, il vaut mieux fournir une implémentation de tous les algorithmes, même si elle contient des erreurs, plutôt qu'une incomplète.

Pour cela, vous complèterez les fichiers :

— `Thompson.ml` pour la génération d'un automate à partir d'une expression rationnelle,
— `Epsilon.ml` pour la suppression des transitions spontanées,
— `Determinization.ml` pour la déterminisation de l'automate obtenu,
— `Evaluation.ml` pour l'évaluation d'un mot par l'automate,

et ne toucherez à aucun autre, ni n'en créerez. Vous n'oublierez pas de remplir les noms et prénoms demandés en en-tête de chacun des fichiers que vous devez compléter (si vous êtes en trinôme, ajoutez une ligne).

Par défaut, leur comportement est de lever l'exception suivante, définie dans `AlgorithmBase.mli` :

```
41 exception Not_implemented
```

**Structures de données** Deux structures de données vous sont fournies. La première représente une expression rationnelle, la seconde un automate. Ces structures étant utilisées dans les signatures des fonctions que vous devez implémenter, vous aurez à les manipuler. A l'intérieur du code de chaque algorithme, vous êtes bien entendu libres d'utiliser toutes les structures annexes dont vous pourriez avoir besoin.

**Expression rationnelle** (`Re.mli`)

```
16 (** A letter in the automaton. *)
17 type letter =
18   Automaton.letter
19
20 (** A binary operation between two parts of a regular expression. *)
21 type binary_operator =
22   | Union
23   | Concatenation
24
25 (** A unary operation over a regular expression. *)
26 type unary_operator =
27   | Repetition
28
29 (** A regular expression. *)
30 type t =
31   | Epsilon
32     (** Epsilon *)
33   | Letter of letter
34     (** Only one letter *)
35   | BinaryOperation of (binary_operator * t * t)
36     (** A binary operation between two patterns. *)
37   | UnaryOperation of (unary_operator * t)
38     (** A unary operation over one pattern. *)
```

Ce module définit un arbre de syntaxe abstraite pour le langage d'entrée de la chaîne de traitements : une expression rationnelle, de type `t`.

**Automate** (`Automaton.mli`)

Le module `Automaton` permet de manipuler une représentation d'automate, paramétré par son alphabet, sous forme d'un graphe. Les opérations de manipulation du graphe sont purement fonctionnelles, le graphe d'origine n'est jamais modifié (comme les opérations sur les listes).

Un état de l'automate contient les informations du module `Automaton.StateKind` :

```
16 (** Data contained in each state. *)
17 module StateKind :
18   sig
19     type t =
20       {
21         identifier  : int ;  (** Unique identifier of the state *)
22         initial     : bool ; (** Is the state an initial state ? *)
23         final       : bool ; (** Is the state a final state ? *)
24       }
25   end
```

Chaque état a ainsi un identifiant codé sur un entier, et deux valeurs booléennes indiquant si l'état est initial et/ou final dans l'automate.

Les lettres sont définies par le type :

```
38 type letter = A | B | C
```

Une interface `Alphabet` est définie comme paramètre de la structure d'automate, ainsi que deux modules répondant à cette interface :

— `EpsilonAlphabet` définit un alphabet comprenant la lettre `Epsilon` pour les transitions spontanées,

— `LetterAlphabet` définit un alphabet n'utilisant que les lettres définies dans le type `letter`, l'automate n'ayant donc pas de transitions spontanées.

Un foncteur prend une interface d'alphabet et retourne une structure d'automate dont les transitions sont étiquetées par cet alphabet :

```
215 (** Functor to build an {!S} module using an {!Alphabet}. *)
216 module Make (A : Alphabet) : S with type Alphabet.t = A.t
```

La documentation de l'interface `S` n'est pas dans ce document car elle comprend un grand nombre de fonctions de manipulation. Vous devez lire la documentation générée automatiquement à partir des sources.

## Exercice 1 – Implémentation des algorithmes

Implémentez les algorithmes.

1. (**Codez l'algorithme de Thompson**) La signature de ce module est donnée par `Thompson.mli` :

```
(** @param  re       a Rationnal Expression (RE)
    @return          an Epsilon Non-deterministic Finite Automaton (e-NFA)
    @raise  AlgorithmBase.Not_implemented if the algorithm is not yet implemented
  *)
val apply :   re:Re.t
          ->  AlgorithmBase.EpsilonAutomaton.t
```

2. (**Élimination des transitions spontanées**) La signature de ce module est donnée par `Epsilon.mli` :

```
(** @param  automaton an Epsilon Non-deterministic Finite Automaton (e-NFA)
    @return           a Non-deterministic Finite Automaton (NFA)
    @raise  AlgorithmBase.Not_implemented if the algorithm is not yet implemented
  *)
val apply :   automaton:AlgorithmBase.EpsilonAutomaton.t
          ->  AlgorithmBase.LetterAutomaton.t
```

3. (**Déterminisation**) La signature de ce module est donnée par `Determinization.mli` :

```
(** @param  automaton a Non-deterministic Finite Automaton (NFA)
    @return           a Deterministic Finite Automaton (DFA)
    @raise  AlgorithmBase.Not_implemented if the algorithm is not yet implemented
  *)
val apply :   automaton:AlgorithmBase.LetterAutomaton.t
          ->  AlgorithmBase.LetterAutomaton.t
```

4. (**Évaluation**) La signature de ce module est donnée par `Evaluation.mli` :

```
(** @param  automaton a Deterministic Finite Automaton (DFA)
    @param  word      a list of letters
    @return           is [word] matched by [automaton] ?
    @raise  AlgorithmBase.Not_implemented if the algorithm is not yet implemented
  *)
val apply :   automaton:AlgorithmBase.LetterAutomaton.t
          ->  word:AlgorithmBase.word
          ->  bool
```