Projet – Égalité d'expressions régulières

Version du 14 novembre 2025

Pierre Coucheney – pierre.coucheney@uvsq.fr
Quentin Japhet – quentin.japhet@uvsq.fr
Franck Quessette – franck.quessette@uvsq.fr

Table des matières

Le but de ce projet est de construire les automates minimums complets de deux expressions régulières et de tester leur égalité.

1 Le langage des expressions régulières

Les expressions régulières sont définies de la façon suivante :

— Les constantes sont :
  1. $\Sigma = \{a, b, c\}$;
  2. ε est noté `E` ;
  3. $\emptyset$ n'est pas utilisé dans ce projet.

— Les opérateurs sont :
  — les opérateurs usuels des expressions régulières sont `+`, `.` et `*` ;
  — `*` est plus prioritaire que `.` et `.` est plus prioritaire que `+` ;
  — `+` et `.` sont associatifs de gauche à droite.
  — `.` est facultatif.

— Les parenthèses `(` et `)` permettent de forcer la priorité.

2 Analyse lexicale

Étape 1 : Définissez les différents token en leur donnant un numéro.

Par exemple :

```
#define PAR_O 101
#define PAR_F 102
```

Ces définitions seront inutiles par la suite, car ce seront les tokens définis dans le fichier yacc qui seront utilisés.

Étape 2 : Créez un fichier lex `regexp.l` qui permettra de reconnaître les différents tokens.

Étape 3 : Tester votre fichier lex sur différents exemples.

3 Analyse syntaxique

Étape 4 : Définissez la grammaire des expressions régulières respectant les indications de la section 1.

Le but est de générer (en utilisant lex et yacc) un fichier contenant un programme Python permettant la création d'un automate reconnaissant le langage défini par l'expression régulière. Ce programme sera ensuite exécuté.

Étape 5 : Créez un fichier yacc `regexp.y` contenant les règles de votre grammaire.

Étape 6 : Utiliser `regexp.l` et `regexp.y` conjointement en testant.

4 Génération de code

Chaque règle de la grammaire doit remonter une chaîne de caractères construite à partir des chaînes de caractères issu de son arbre de dérivation. Ce qui remonte à la racine de l'arbre est le code final en python sous forme d'une chaîne de caractères. Pour cela, un ensemble de fonctions de base de manipulation des automates vous est fourni en Python, voir la section 6 ci-dessous.

5 Exécution

Le fichier que vous donnerez en entrée à votre compilateur devra s'appeler `test.1`, ce fichier contiendra deux lignes avec une expression régulière sur chaque des lignes.

```
(a+b)*.c
a*.c + b*.c
```

Étape 7 : Vous générerez un fichier Python `main.py` qui sera en suite exécuté.

Le résultat de l'exécution devra être `EGAL` si les deux expressions régulières définissent le même langage et `NON EGAL` sinon.

6 Exemple et fichiers fournis

6.1 Exemple en Python

Le fichier `main.py` généré pourra être de la forme :

```
from automate import *

a0 = automate("a")
a1 = automate("b")
a2 = union(a0, a1)
a3 = etoile(a2)
a4 = automate("c")
a5 = concatenation(a3, a4)
a6 = tout_faire (a5)

a7 = automate("a")
a8 = etoile(a7)
a9 = automate("c")
a10 = concatenation(a8,a9)
a11 = automate("b")
a12 = etoile(a11)
a13 = automate("c")
a14 = concatenation(a12,a13)
a15 = union(a10,a14)
a16 = tout_faire(a15)

if egal(a6,a16):
    print("EGAL")
else:
    printf("NON EGAL")
```

6.2 Fichiers fournis pour travailler en Python

Un fichier `IN520_Projet_Python.zip` qui contient le dossier `IN520_Projet_Python` qui contient :

— Un fichier `Makefile`, à modifier pour adapter les commandes à votre OS et éventuellement faire une cible pour créer le zip à rendre.

— Un fichier `automate.py`, contenant certaines fonctions de base de manipulation des automates. Vous devrez programmer les fonctions manquantes nécessaires.

— Un fichier `test.1`, fichier contenant l'exemple décrit dans cet énoncé.

— Un fichier `main.1.py`, fichier du code python que vous devez générer pour l'exemple `test.1`.

7 Programmation à faire sur les automates

Les différentes fonctions sont :

— `automate` qui crée un automate à partir d'un caractère ou de epsilon. Cette fonction est déjà écrite.

— `union`, fonction à écrire.

— `etoile`, fonction à écrire.

— `concatenation`, fonction à écrire.

— `tout_faire`, le code de cette fonction est

```
def tout_faire (a):
    a1 = supression_epsilon_transitions(a)
    a2 = determinisation(a1)
    a3 = completion(a2)
    a4 = minimisation(a3)
    return a4
```

— `supression_epsilon_transitions`, déjà écrite.

— `minimisation`, déjà écrite.

— `determinisation`, fonction à écrire.

— `completion`, fonction à écrire.

— `egal`, fonction à écrire.

8 Tests de vos fonctions

Pour chaque fonction que vous écrivez, vous devez programmer un code permettant de la tester sur deux ou trois exemples pertinents. De plus vous devez fournir un document pdf donnant les dessins des automates résultants.

9 Organisation, rendu et date

Le projet est à faire en binôme.

Vous devez rendre sur ecampus dans la section projet un fichier qui s'appelle `NOM1_NOM2.zip` avec les noms des deux membres du binôme. Un seul des deux membres du binôme dépose le projet.

Ce fichier zip doit contenir un dossier `NOM1_NOM2` contenant tous les fichiers nécessaires à votre projet ainsi que le document pdf des tests.

La date de rendu est fixée au **xx janvier 2026** à 23h59. Chaque heure de retard retire 1 point.

Une soutenance oral **obligatoire** sera organisée courant janvier.

Lors de la soutenance vous devrez être capable d'expliquer précisément toutes les parties de votre code.