Université de Versailles – Saint-Quentin-en-Yvelines
Licence 2 Informatique

2023-2024

Projet – Construction d'automate à partir d'une expression régulière

Version du 20 décembre 2023

Pierre Coucheney – pierre.coucheney@uvsq.fr
Franck Quessette – franck.quessette@uvsq.fr
Yann Strozecki – yann.strozecki@uvsq.fr

Table des matières

1 Le langage des expressions régulières **1**

2 Analyse lexicale **2**

3 Analyse syntaxique **2**

4 Génération de code **2**

5 Exécution **2**

6 Exemple et fichiers fournis **3**
6.1 Exemple en C 3
6.2 Fichiers fournis pour travailler en C 3
6.3 Exemple en Python 4
6.4 Fichiers fournis pour pour travailler en Python 4

7 Travail en plus **5**

8 Organisation, rendu et date **5**

Le but de ce projet est de construire un automate reconnaissant un langage définit par une expression régulière et de tester l'appartenance de mots en utilisant l'automate.

1 Le langage des expressions régulières

Les expressions régulières sont définies de la façon suivante :

— Les constantes sont :
  1. Σ contient toutes les lettres minuscules de `a` à `z` ;
  2. ε est noté `E` ;
  3. $\emptyset$ est noté `0`.

— 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 même si ça ne change rien.

— 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 C ou Python permettant la création d'un automate reconnaissant le langage définit 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 C ou en Python) sous forme d'une chaîne de caractères. Pour ce faire un ensemble fonctions de base de manipulation des automates vous est fourni en C et en Python, voir la section 6 ci dessous.

5 Exécution

Le fichier que vous donnerez entrée à votre compilateur devra s'appeler `test.1`, ce fichier contiendra sur la première ligne une expression régulière et sur les lignes suivantes des mots à tester, par exemple :

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

Étape 7 : Modifier votre fichier `regexp.l` et `regexp.y` pour reconnaitre en plus de l'expression régulière, les mots à tester.

Étape 8 : Vous générerez un fichier C `main.c` ou Python `main.py` qui sera en suite compilé (pour le C) et exécuté.

Le résultat de l'exécution devra être

— Affichage de l'automate sous forme texte, compréhensible.

— Pour chaque des mots dire s'il est reconnu ou pas avec un affichage texte compréhensible.

6 Exemple et fichiers fournis

6.1 Exemple en C

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

```
#include "automate.h"
#include <stdio.h>

int main () {
AUTOMATE A0, A1, A2, A3, A4, A5;
AUTOMATE A_final;
A0 = creer_automate_une_lettre('a');
A1 = creer_automate_une_lettre('b');
A2 = union_automate(A0,A1);
A3 = etoile_automate(A2);
A4 = creer_automate_une_lettre('c');
A5 = concat_automate(A3,A4);

A_final = supprime_epsilon_transitions (A5);
afficher(A_final);

reconnait(A_final,"ab");
reconnait(A_final,"aaaaac");
reconnait(A_final,"c");

return 0;
}
```

Après compilation l'exécution donne :

```
### (((a+b)*).c) : 7 états, 12 transitions
    États finaux : 6
    Transitions  : (0,a,2) (0,b,4) (0,c,6) (1,a,2) (2,a,2)
                   (2,b,4) (2,c,6) (3,b,4) (4,a,2) (4,b,4)
                   (4,c,6) (5,c,6)

ab N'est PAS reconnu par (((a+b)*).c)
aaaaac EST RECONNU PAR (((a+b)*).c)
c EST RECONNU PAR (((a+b)*).c)
```

6.2 Fichiers fournis pour travailler en C

Un fichier `IN520_Projet_C.zip` qui contient le dossier `IN520_Projet_C` 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.

— Les fichies `automate.c` et `automate.h`, contenant les fonctions de base de manipulation des automates. Il est à compléter, voir la section "Travail en plus".

— 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`.

6.3 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);

a_final = a5
print(a_final);
print(reconnait(a_final,"ab"))
print(reconnait(a_final,"aaaaac"))
print(reconnait(a_final,"c"))
```

L'exécution donne :

```
Automate (((a)+(b))*.(c))
Nombre d'états 9
Etats finals [8]
Transitions:
(2, 'a'): [3]
(4, 'b'): [5]
(1, 'E'): [2, 4]
(3, 'E'): [1, 7]
(5, 'E'): [1, 7]
(0, 'E'): [1, 6]
(7, 'c'): [8]
(6, 'E'): [7]
**********
False
True
True
```

6.4 Fichiers fournis pour travailler en Python

Un fichier `IN520_Projet_Python.zip` qui contient le dossier Le fichier `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 les fonctions de base de manipulation des automates. Il est à compléter, voir la section "Travail en plus".

— 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 Travail en plus

Vous pouvez enrichir le code C ou le code Python, en ajoutant les fonctions de :

— Suppression des ε-transitions ;

— Déterminisation ;

— Minimisation.

Ces fonctions modifient en simplifiant la fonction de reconnaissance.

Dans l'exemple en C ci-dessus, il y a la fonction de suppression des ε-transitions, mais pas les autres

8 Organisation, rendu et date

Le projet est à faire en binôme, vous pouvez également le faire seul.

Vous devez rendre sur ecampus dans le section projet un fichier qui s'appelle `NOM1_NOM2.zip` avec les noms des deux membres du binôme (un seul nom si vous êtes seul). 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.

La date de rendu est fixée au **mardi 09 janvier 23h59**. Chaque heure de retard retire 1 point.

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

Il y a 10 points pour le rendu de projet et 10 points pour la soutenance.