

# Projet – Évaluation d’expressions booléennes

Version du 21 novembre 2023

Pierre Coucheney – [pierre.coucheney@uvsq.fr](mailto:pierre.coucheney@uvsq.fr)  
Franck Quessette – [franck.quessette@uvsq.fr](mailto:franck.quessette@uvsq.fr)  
Yann Strozecki – [yann.strozecki@uvsq.fr](mailto: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 en C	3
7	Exemple en Python	3
8	Code à faire en C ou en Python	4
9	Travail en plus	4
10	Ce qui vous est fourni	4
11	Organisation, rendu et date	4

Le but de ce projet est de construire un automate reconnaissant un langage défini 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.  $\Sigma$  contient toutes les lettres minuscules de `a` à `z` ;
2.  $\varepsilon$  est noté `E` ;
3.  $\emptyset$  est noté `O` .

— 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

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

Par exemple :

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

**Question 2 :** Créez un fichier lex `regex.1` qui permettra de reconnaître les différents tokens.

**Question 3 :** Tester votre fichier lex sur différents exemples.

## 3 Analyse syntaxique

**Question 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 défini par l'expression régulière. Ce programme sera ensuite exécuté.

**Question 5 :** Créez un fichier yacc `regex.y` contenant les règles de votre grammaire.

**Question 6 :** Utiliser `regex.1` et `regex.y` conjointement en testant.

## 4 Génération de code

Chaque règle de la grammaire doit remonter une chaîne de caractère construite à partir des chaînes de caractère issu de son arbre de dérivation. Ce qui remonte à la racine de l'arbre est le code final (en C ou en Python). Pour ce faire un ensemble fonctions de base de manipulation des automates vous est fourni. Voir les sections Langage C et Python 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
```

**Question 7 :** Modifier votre fichier `regex.1` et `regex.y` pour reconnaître en plus de l'expression régulière, les mots à tester.

**Question 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 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)
```

Les fichiers `automate.h` et `automate.c` vous sont fournis.

## 7 Exemple en Python

À venir...

## 8 Code à faire en C ou en Python

Vous devez implémenter la fonction de reconnaissance d'un mot par un automate.

## 9 Travail en plus

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

- Suppression des  $\varepsilon$ -transitions ;
- Détermination ;
- 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  $\varepsilon$ -transitions, mais pas les autres

## 10 Ce qui vous est fourni

Un fichier `IN520_Projet.zip`, qui contient un dossier `IN520_Projet` contenant

- Le `Makefile`, il peut y avoir des adaptations à faire selon votre système.
- Les fichiers `automate.h` et `automate.c` pour le C
- Un fichier `main.1.c` permettant de tester l'exécution de `automate.h` et `automate.c` pour la création de l'automate. Il faut copier `main.1.c` en `main.c` et paer `make main` puis `./main`.
- Un fichier `test.1`, celui de l'exemple ci-dessus.

## 11 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.