# Théorie des langages- IN406

# Evaluation d'expressions booléennes

Zeyneb Bouabdallaoui, Nicolas Fond

N° : 21904931 / 21908626 May 23, 2021

## Introduction

Ce projet consiste à évaluer des expressions booléennes en C à l'aide d'automates à piles. Nous expliquerons ici les automates construits ainsi qu'une bref description du code.

## 1 Reconnaissance par automate

Le but de cette partie est de construire un automate à pile permettant de reconnaître une expression booléenne. Chaque élément de l'expression devient ici un *token* et l'expression booléenne entière un mot. Pour construire l'automate, nous pouvons tout d'abord nous intéresser aux propriétés de chaque type de l'expression. Nous allons tout d'abord regarder quels sont les éléments qui peuvent venir avant et après un élément. Par exemple un 1, soit une constante, ne peut avoir une parenthèse ouvrante après lui. On peut alors émettre une liste de conditions qui indiquent quels sont les prédécesseurs et les successeurs possibles d'un élément donné. De ce fait, on crée une vérification locale de chaque élément:

| | Constante $0$, $1$ | Opérateur unaire $+, \bullet, \Rightarrow, \Longleftrightarrow, +$ | Parenthèse ouvrante $($ | parenthèse fermante $)$ | Opérateur unaire $NON$ |
|---|---|---|---|---|---|
| Constante | | X | X | | X |
| Opérateur binaire | X | | | X | |
| Parenthèse ouvrante | | X | X | | X |
| Parenthèse fermante | X | | | X | |
| Opérateur unaire | | X | X | | |

Chaque croix sur ce tableau signifie que l'élément $a$ est prédécesseur de l'élément $b$ et donc que $b$ est successeur de $a$. Le tableau se lit de la façon suivante : si on veut connaître les prédécesseurs, il suffit de regarder sur la colonne en abscisse l'élément voulu puis à chaque croix rencontrée dans la colonne correspond un prédécesseur. De la même manière, si on veut connaître les successeurs d'un élément, on regarde en ordonnée où se situe l'élément souhaité puis à chaque croix rencontrée dans la ligne correspond un successeur. Ce tableau nous sera très utile pour construire l'automate.

Maintenant que nous connaissons les contraintes pour chaque élément, nous pouvons construire les automates qui nous permettront de construire l'automate final comprenant les expressions booléennes.

Tous les automates seront des automates à reconnaissance par pile vide. On notera $\delta$ la pile vide. Pour clarifier les automates, chaque transition sera marquée sans parenthèses, un saut de ligne indiquera qu'on se trouve sur une autre transition.

On notera $a$ l'ensemble qui comprend les constantes $1$ et $0$ et $b$ l'ensemble qui comprend les opérateurs binaires $\Rightarrow$, $\bullet$, $\Longleftrightarrow$ et $+$. L'opérateur unaire sera noté $n$ et les parenthèses ( et ). On a donc :

$$a = \{1, 0\} \ ; \ b = \{\Rightarrow, \Longleftrightarrow, +, \bullet\}$$

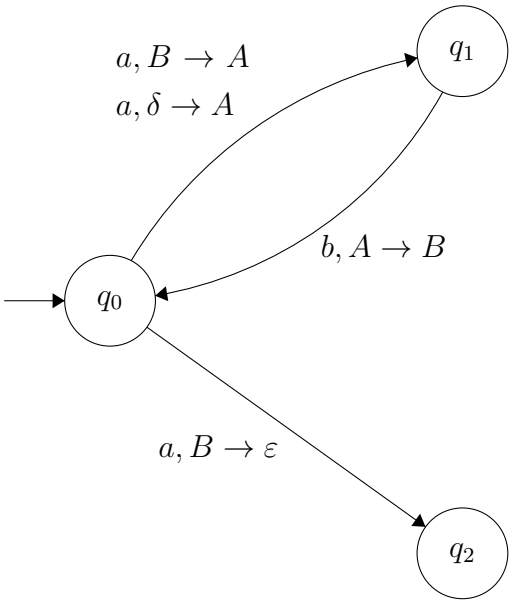
Chaque transition sera représentée sous cette forme :

$$x, Y \to X$$

Cela signifie que si la tête de lecture du mot lit un $x$ et que la pile contient un $Y$, alors $Y$ devient $X$. Les automates suivants sont non déterministes.

## 1.1 Automate 1

Commençons simplement par un automate qui reconnaît les expressions booléennes contenant uniquement des constantes et des opérateurs binaires. Soit en version simplifier une alternance de $a$ et de $b$ où $a$ doit commencer et finir l'expression. Par exemple $ababa$ est valide. Les mots $a$, $abb$, $abab$ ou $aaba$ ne ne sont pas valides et ne seront donc pas reconnu par l'automate. L'idée est de créer une pile permettant de stocker le prédécesseur de l'élément actuel. Ainsi, en connaissant le prédécesseur (en se servant du tableau), on peut créer des transitions possibles après ce token. La pile nous servira uniquement à regarder quel était le prédécesseur du token actuel. On crée donc une boucle qui permettra de prendre autant de tokens qu'il y a dans le mot.



**Construction de l'automate**

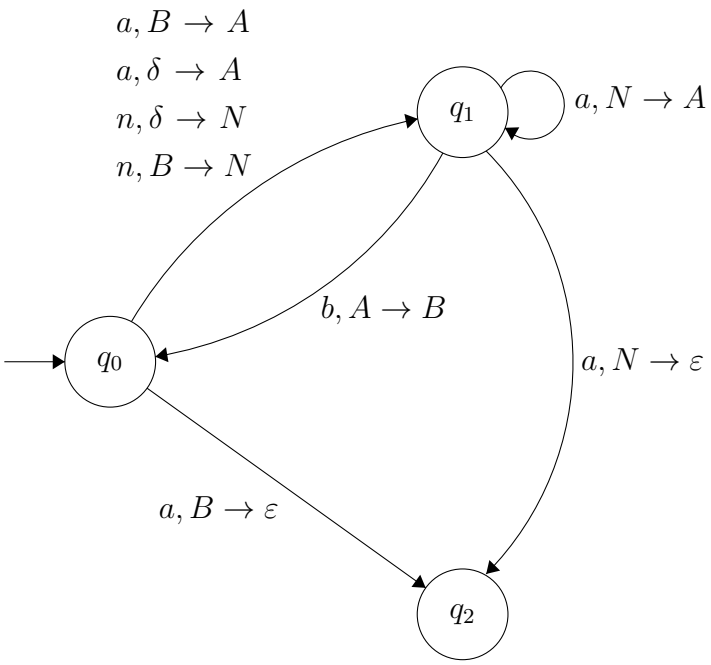
Si l'automate lit un $a$, on empilera $A$ pour le mémoriser. Pour passer à l'état suivant, le token d'après est obligatoirement un $b$, on vérifie alors dans cette transition si la valeur précédente est bien $a$ et on remplace le $A$ de la pile par un $B$. Une fois $B$ dans la pile, la transition suivante est obligatoirement un $a$. On a alors deux choix :

- La constante $a$ est le dernier token du mot : on ira donc dans un état qui videra la pile et le mot sera accepté.
- La constante $a$ n'est pas le denier token du mot : on refait un tour de boucle comme vu dans le paragraphe précédent.

## 1.2 Automate 2

À partir de l'automate précédent, on peut créer l'automate qui reconnaît les expressions booléennes sans les parenthèses. On a donc l'opérateur unaire qui vient s'ajouter à notre automate. Cet opérateur n'est pas très compliqué à implémenter dans l'automate puisqu'il n'a qu'un prédécesseur et un successeur (sans les parenthèses). Il faudra donc prêter attention aux opérateurs unaires quand on implémentera les parenthèses à l'automate.

L'idée est sensiblement la même que l'automate 1 à la différence que l'opérateur unaire peut se positionner avant un $a$ (soit une constante) et après un $b$ (soit un opérateur binaire). Il y aura donc une nouvelle transition de $q_0$ à $q_1$ pour gérer les sous mots $... + NON...$ et une nouvelle transition de $q_1$ à $q_1$ qui viendra ajouter à la pile le $a$ qui vient obligatoirement après un opérateur unaire. Enfin, une transition de $q_1$ à $q_2$ a été ajoutée pour prendre en compte les sous mots du type $NON0$.

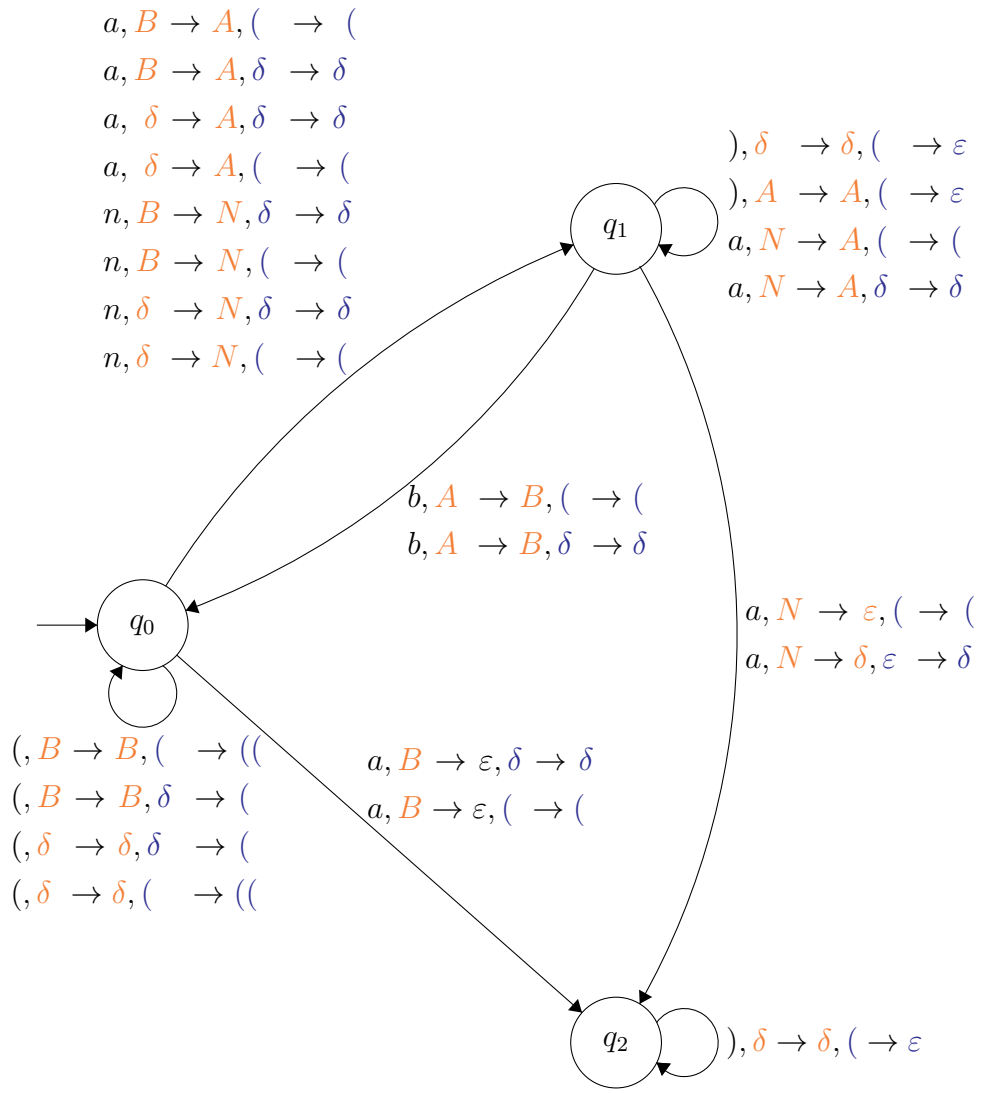


## 1.3 Automate final

Les parenthèses sont ici gérées avec une autre pile qui empilera les parenthèses ouvrantes. À chaque parenthèse fermante rencontrée, on dépilera la parenthèse ouvrante mis au préalable. De ce fait, on vérifie que chaque parenthèse ouvrante se ferme. Si toutes sont fermées, alors la pile sera vide en fin de parcours. Si ce n'est pas le cas, alors la pile ne sera pas vide et le mot ne sera pas reconnu par l'automate.

La gestion des parenthèses nécessite d'ajouter deux boucles. Une à l'état $q_0$ et une à l'état $q_1$. Celle à l'état $q_0$ empile les parenthèses ouvrantes à chaque parenthèse ouvrante rencontrée. Celle à l'état $q_1$ dépile les parenthèses ouvrantes à chaque parenthèse fermante rencontrée.

La transition de $q_1$ à $q_1$ permet aussi de gérer les parenthèses après les constantes. La pile contenant les opérateurs unaires et binaires ainsi que les constantes est représentée en orange et la pile contenant les parenthèses est représentée en bleu.




### Description formelle de l'automate

$\mathcal{A} = (, Q, q_0, \Gamma, \delta, F, T)$
$\Sigma = \{a, b, n, (, )\}$

$Q = \{q_0, q_1, q_2\}$
$\Gamma = \{A, B, N, (, )\}$
$F =$ Reconnaissance par pile vide

$$T = \left\{ \begin{array}{c} q_0,\ (,\ B,\ (,\ q_0,\ B,\ (( \\ q_0,\ (,\ \delta,\ \delta,\ q_0,\ \delta,\ ( \\ q_0,\ (,\ B,\ (,\ q_0,\ \delta,\ (( \\ q_0,\ a,\ B,\ (,\ q_1,\ A,\ ( \\ q_0,\ a,\ B,\ \delta,\ q_1,\ A,\ \delta \\ q_0,\ a,\ \delta,\ \delta,\ q_1,\ A,\ \delta \\ q_0,\ a,\ \delta,\ (,\ q_1,\ A,\ ( \\ q_0,\ n,\ B,\ \delta,\ q_1,\ N,\ \delta \\ q_0,\ n,\ B,\ (,\ q_1,\ N,\ ( \\ q_0,\ n,\ \delta,\ \delta,\ q_1,\ N,\ \delta \\ q_0,\ n,\ \delta,\ (,\ q_1,\ N,\ ( \\ q_0,\ a,\ B,\ \delta,\ q_2,\ \varepsilon,\ \delta \\ q_0,\ a,\ B,\ (,\ q_2,\ \varepsilon,\ ( \\ q_1,\ ),\ \delta,\ (,\ q_1,\ \delta,\ \epsilon \\ q_1,\ ),\ A,\ (,\ q_1,\ A,\ \epsilon \\ q_1,\ a,\ N,\ (,\ q_1,\ A,\ ( \\ q_1,\ a,\ N,\ \delta,\ q_1,\ A,\ \delta \\ q_1,\ b,\ A,\ (,\ q_0,\ B,\ ( \\ q_1,\ b,\ A,\ \delta,\ q_0,\ B,\ \delta \\ q_1,\ a,\ N,\ (,\ q_2,\ \varepsilon,( \\ q_1,\ a,\ N,\ \varepsilon,\ q_2,\ \delta,\ \delta \\ q_2,\ ),\ \delta,\ (,\ q_2,\ \delta,\ \varepsilon \end{array} \right\}$$

# 2 Fonction principale du programme

Pour rappel, l'idée générale du projet est de mettre en argument une expression booléenne à notre programme et qu'il en ressort le résultat. Pour cela, on créé une liste de *tokens* qui prend tous les éléments de l'expression booléenne. On réalise ensuite notre arbre en fonction de la liste de *token*. Finalement on calcule le résultat de notre expression à partir de l'arbre qui lui correspond. Le résultat est ensuite affiché sur le terminal.

# 3 Affichage

Le programme possède un affichage simple montrant qu'il a bien pris en compte chaque *token* et un affichage plus graphique (toujours dans le terminal) affichant l'arbre. Toutes ces fonctions sont dans le fichier `affichage.c`. L'affichage graphique se présente sous la forme suivante :

```
1=>(1.1)
                ~
            1
                ~
        .
                ~
            1
                ~
=>
            ~
        1
            ~
```

L'arbre est à lire horizontalement. Par exemple ici, => a pour fils gauche 1 et pour fils droit . .