Projet IN406 - Évaluation d'expressions booléennes

DANKOU Mathis et SOURSOU Adrien

Mai 2019

Question 1 :

On définit G la grammaire reconnaissant les expressions booléennes :

$G = (\Sigma,\ V,\ S,\ P)$
$\Sigma = \{\ 0,\ 1,\ NON,\ +,\ .,\ \Rightarrow,\ \Leftrightarrow\ \}$
$V = \{\ E,\ C,\ OP\ \}$
$S = E$
$P = \{$
$\quad E \rightarrow C \mid (E) \mid NON\ E \mid E\ OP\ E,$
$\quad C \rightarrow\ 0 \mid 1,$
$\quad OP \rightarrow\ + \mid . \mid\ \Rightarrow\ \mid\ \Leftrightarrow$
$\}$

Question 2 :

```
typedef enum { CONSTANTE, OPERATEUR, PARENTHESE } e_type;

typedef enum
{
    FAUX = 0, VRAI = 1,
    NON, ET, OU, IMPLICATION, EQUIVALENCE,
    GAUCHE, DROITE
}   e_valeur;

typedef struct token* liste_token;
struct token
{
    e_type type;
    e_valeur valeur;
    liste_token suivant;
};
```

Question 3 :

Soient L le langage ainsi que c et e deux expressions régulières :
$c = [\ NON \mid (^*\ ]^*\ [\ 0 \mid 1\ ]\ )^*$
$e = c\ [\ [\ + \mid .\ \mid\ \Rightarrow\ \mid\ \Leftrightarrow\ ]\ c\ ]^*$
$L = \{\ w \in e,\ n\ \geq 0,\ |w|_(= |w|_) = n\ \}$

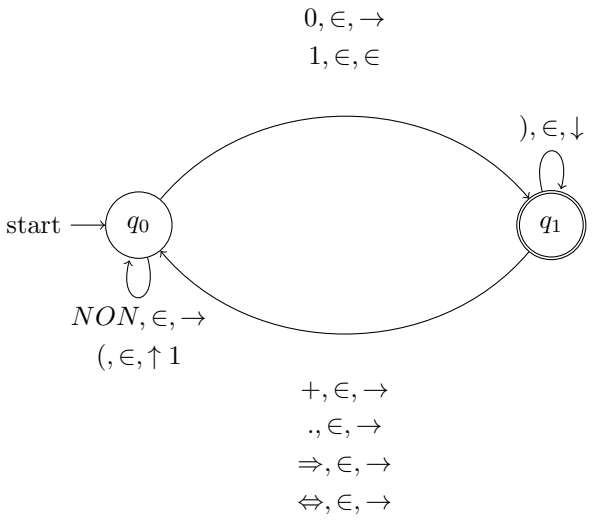

Figure 1: Automate à pile reconnaissant le langage par état final et pile vide.

Définition formelle :

$A = (\Sigma,\ Q,\ \Gamma,\ q_0,\ F,\ T)$
$\Sigma = \{\ 0,\ 1,\ NON,\ +,\ .,\ \Rightarrow,\ \Leftrightarrow\ \}$
$Q = \{\ q_0,\ q_1\ \}$
$\Gamma = \{\ 1\ \}$
$F = \{\ q_0\ \}$
$T = \{$
$(q_0,\ \in,\ NON,\ q_0,\ \rightarrow),\ (q_0,\ \in,\ (,\ q_0,\ \uparrow 1),$
$(q_0,\ \in,\ 0,\ q_1,\ \rightarrow),\ (q_0,\ \in,\ 1,\ q_1,\ \rightarrow),$
$(q_1,\ \in,\ +,\ q_0,\ \rightarrow),\ (q_1,\ \in,\ .,\ q_0,\ \rightarrow),$
$(q_1,\ \in,\ \Rightarrow,\ q_0,\ \rightarrow),\ (q_1,\ \in,\ \Leftrightarrow,\ q_0,\ \rightarrow),$
$(q_1,\ \in,\ ),\ q_1,\ \downarrow),$
$\}$

Question 4 :

```
typedef struct arbre* arbre_token;
struct arbre
{
    e_type type;
    e_valeur valeur;
    arbre_token gauche;
    arbre_token droite;
};
```

Question 5 :

```
int resoudre(e_valeur a, e_valeur b, e_valeur op)
{
    if (op == NON)
        return ! a;
    else if (op == OU)
        return a | b;
    else if (op == ET)
        return a & b;
    else if (op == IMPLICATION)
        return (! a) | b;
    // else (op == EQUIVALENCE)
    return ! a ^ b;
}

int arbre_to_int(arbre_token at)
{
    if (! at)
        return 0;
    else if (at->type == CONSTANTE)
        return at->valeur;
    return resoudre(arbre_to_int(at->gauche), arbre_to_int(at->droite), at->valeur);
}
```

Question 7 :

La priorité des opérateurs est décrite par le tableau de priorité ci-dessous :

Opérateur	Priorité
NON	1
.	2
+	3
$\Rightarrow$	4
$\Leftrightarrow$	5

Commentaires

Le programme par défaut n'affiche que le résultat de l'expression. Cependant, une variable DEBUG peut être modifiée dans le fichier *eval.c* afin de visualiser l'arbre construit.

En plus de la fonction *liste_token_to_arbre_token*, nous utilisons la fonction *liste_token_to_postfixe*, permettant de transformer notre liste de tokens en liste postfixe. Au sein de la fonction *liste_token_to_arbre_token*, les éléments de type constante de la liste sont ajoutés à une pile d'arbre_token implémentée sous forme de tableau. Les arbres sont combinés ensemble lorsqu'un token de type opérateur est rencontré.

La conversion de la chaîne de caractère en liste de tokens se fait en un seul parcours.
La liste de tokens est ensuite parcourue à quatre reprises : une première fois afin de la transformer en liste de tokens postfixe, une seconde afin de calculer la taille maximum de notre pile d'arbres, une troisième pour construire notre arbre de tokens, et une dernière pour libérer la mémoire allouée.

On peut ainsi estimer que la complexité de notre programme est en $O(n)$ linéaire.