RAPPORT IN406 : Evaluation d'expressions booléennes

Mohand Arezki ACHERIR 21921002 – Clémence DUMOULIN 21903986

25 mai 2021

Sommaire

1 Fonctionnement du programme

1.1 Organisation des fichiers

L'implémentation de ce projet est en C, décomposée en deux fichiers. L'un correspondant à l'évaluation d'expression booléenne sans priorité entre les opérateurs **eval.c**, et l'autre à l'évaluation d'expression booléenne avec une priorité entre les opérateurs **evalprio.c**.

1.2 Matériel de compilation

Pour la compilation on utilise un **Makefile**, il suffit d'écrire la commande **make** ou **make test** pour compiler **eval.c** et lancer 5 exemples pertinents sur l'évaluation d'une expression booléenne. Pareillement, écrire la commande **make priorite** permet de compiler **evalprio.c** et lancer 6 exemples montrant les différences d'évaluation liées à la priorité des opérateurs. Ces expressions sont celles respectivement indiquées dans la cible **test** et dans la cible **priorite**. On peut les changer dans le Makefile, ou écrire **./eval "expression booléenne"** en ligne de commande. De plus, écrire **make debug** ou **make debugp** permet de vérifier à l'aide de Valgrind si la mémoire a bien été utilisée (allocation et libération de mémoire) dans chacun des cas énoncés ci-dessus. Enfin, la cible **clean** permet d'effacer les exécutables et de lister le contenu du répertoire courant.

2 Raisonnement et description du programme

Cette partie est divisée en quatre sous parties. Une pour la première question, une autre pour les questions deux et trois, la troisième pour les questions quatre et cinq, et la dernière pour la partie facultative.

2.1 Transformation d'une chaine de caractère en liste de tokens

On souhaite ici, lire une expression en argument, vérifier s'il s'agit bien d'une expression écrite avec l'alphabet du langage pour ensuite l'affecter à une liste chainée de token. On ne veut pas créer une liste de tokens si le mot est composé d'autres lettres que celles du langage. Par exemple, le mot **1+a** n'est pas transformé en liste de token puisque la lettre a n'appartient pas à notre alphabet. On supprime également les espaces s'il y en a pour ne pas stocker inutilement un espace dans un token. De plus, on considère les mots en C, composés de plusieurs caractère, comme des lettres. Par exemple, NON est ajouté dans la liste, seulement si on lit un N, puis un O et un autre N, sinon NON n'est pas considéré comme une lettre de l'alphabet. Ainsi, la fonction en C :

```
liste_token creer_liste(char *string);
```

permet de vérifier l'appartenance de chaque lettre à notre alphabet et de construire la liste de tokens comme liste chainée avec les fonctions

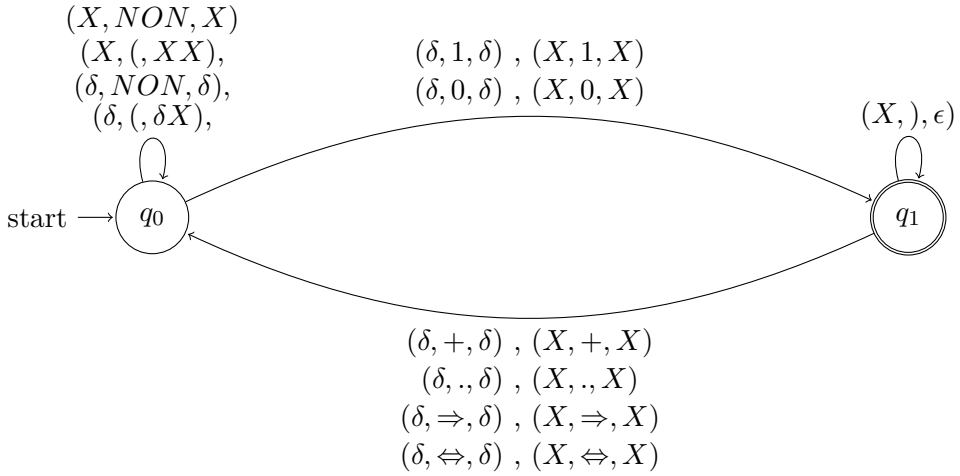
```
liste_token creer_token(int type, char *op, char par, int value);
liste_token ajouter_fin_liste(liste_token current, liste_token add);
```

2.2 Reconnaissance par automate

On crée l'automate à pile $A = \{\Sigma, Q, q0, \Gamma, \delta, F, T\}$, tel que :
$\Sigma = \{0, 1, NON, (,), ., +, =>, <=>\}$
$Q = \{q0, q1\}$
$\Gamma = \{X\}$
$F = \{q1\}$

$$T = \left\{ \begin{array}{lllll} (q0, & \delta, & (, & \delta X, & q0), \\ (q0, & X, & (, & XX, & q0), \\ (q0, & \delta, & NON, & \delta, & q0), \\ (q0, & X, & NON, & X, & q0), \\ (q0, & \delta, & 0, & \delta, & q1), \\ (q0, & X, & 0, & X, & q1), \\ (q0, & \delta, & 1, & \delta, & q1), \\ (q0, & X, & 1, & X, & q1), \\ (q1, & X, &), & \epsilon, & q1), \\ (q1, & \delta, & +, & \delta, & q0), \\ (q1, & X, & +, & X, & q0), \\ (q1, & \delta, & ., & \delta, & q0), \\ (q1, & X, & ., & X, & q0), \\ (q1, & \delta, & \Rightarrow, & \delta, & q0), \\ (q1, & X, & \Rightarrow, & X, & q0), \\ (q1, & \delta, & \Leftrightarrow, & \delta, & q0), \\ (q1, & X, & \Leftrightarrow, & X, & q0), \end{array} \right\}$$

Ce qui nous donne, graphiquement :



On fait une reconnaissance par pile vide et état final.
La pile de l'automate permet de vérifier la bonne position, et le bon nombre de parenthèses (ouvrantes et fermantes). L'état final permet de ne pas reconnaître une expression ne finissant ni par une constante, ni par une parenthèse fermante.
Pour construire cet automate, on a utilisé le tableau ci-dessous, qui permet de savoir quelle lettre peut directement se trouver à la suite d'une autre.

courant \ *suivant*	(	)	0	1	NON	+	.	⇒	⇔	Fin
(	X		X	X	X					
)		X				X	X	X	X	X
0		X				X	X	X	X	X
1		X				X	X	X	X	X
NON	X		X	X	X					
+	X		X	X	X					
.	X		X	X	X					
⇒	X		X	X	X					
⇔	X		X	X	X					
Initial	X		X	X	X					

On remarque deux statuts différents, correspondant aux deux états de notre automate. Les boucles sur un état se justifient par un élément qui peut se suivre lui-même (la diagonale du tableau).
L'implémentation en C se fait avec la fonction, ci-dessous, qui renvoie 0 quand l'expression n'est pas reconnu par l'automate et 1 sinon.

```
int est_valide(liste_token lt);
```

2.3 Arbre et évaluation de l'expression

Dans cette partie, l'objectif est de rechercher l'opérateur de priorité la plus faible. Cependant, on ne considère pas la priorité entre les opérateurs, ainsi l'opérateur le moins prioritaire devient le dernier dans l'ordre de lecture en dehors des parenthèses. On considère tout de même que sans parenthèse l'opérateur unaire NON est prioritaire. Par exemple, dans l'expression $1 \Rightarrow NON1$, $NON1$ sera effectué avant l'implication. L'opérateur le moins prioritaire sera effectué en dernier. Cette recherche se fait dans la fonction :

```
liste_token chrch_opp(liste_token lt, int nb)
```

Une fois cet opérateur trouvé, on l'affecte en racine de l'arbre et on découpe la liste chaînée en deux, la partie précédant l'opérateur et la partie suivant l'opérateur. Le fils gauche de la racine sera la partie précédant l'opérateur, et le fils droit, la partie le suivant. Ceci est permis grâce aux fonctions :

```
liste_token get_prev(liste_token lt, liste_token t, int nb);
liste_token get_next(liste_token lt, liste_token t, int nb);
```

On répète ces étapes récursivement jusqu'à finir la construction de l'arbre.
Nous avons choisi d'afficher l'arbre en parcours postfixe puisque combiné à l'expression en argument, qui correspond à un parcours infixe, on peut vérifier que l'arbre est correctement construit.

Après avoir construit l'arbre, on évalue l'expression remontant l'arbre. Lorsqu'on atteint une feuille, on stocke sa valeur dans une variable et on remonte au père pour effectuer l'opération correspondante à l'opérateur stocké dans ce noeud. L'évaluation se fait grâce à la fonction :

```
int arbre_to_int(arbre_token at);
```

2.4 Avec priorité des opérateurs

Dans cette partie, il suffit seulement d'intégrer une fonction qui indique la priorité suivie par les opérateurs. On définit les opérateurs du plus prioritaire au moins prioritaire :

- ()
- NON
- .
- $+$
- $\Rightarrow$
- $\Leftrightarrow$

On implémente cette priorité par la fonction :

```
int is_weaker(liste_token curr, liste_token opp);
```

Il faut également une petite modification dans la recherche de l'opérateur racine de l'arbre, lorsqu'on compare deux opérateurs entre eux, dans la fonction :

```
liste_token chrch_opp(liste_token lt, int nb);
```