Licence Sciences et Technologie **UGA**
Parcours Informatique **UE INF404**

Devoir Surveillé du 13 mars 2020

Durée : 1h - Document autorisé : une feuille A4 recto-verso - Barême indicatif

Les programmes demandés peuvent êtres écrits en (pseudo-) C ou en notation algorithmique ...

Introduction

On s'intéresse à un langage d'expressions booléennes. Ce langage est constitué des constantes `tt` et `ff`, représentant les valeurs "vrai" et "faux", de l'opérateur unaire `!`, représentant la négation, et de l'expression conditionnelle (`if-then-else`). Chaque expression de ce langage correspond à une valeur booléenne. Ainsi :

- `tt` vaut vrai
- `!tt` vaut faux
- `if !tt then (if tt then ff else tt) else ff` vaut faux

L'objectif de ce travail est d'écrire un programme sur le même modèle que ce qui a été fait en TP pour la "calculatrice". Ce programme prendra donc en entrée une expression lue au clavier et effectuera :

- une analyse lexicale du langage : partie 1 ;
- une analyse syntaxique : partie 2 ;
- un traitement effectué à partir de l'analyse syntaxique : partie 3.

Partie 1 : analyse lexicale (3 points)

Les lexèmes du langage sont les suivants :

- `TT`, représente la constante vrai : `tt`
- `FF`, représente la constante faux : `ff`
- `NOT`, représente l'opérateur de négation : `!`
- `PARO`, représente la parenthèse ouvrante : `(`
- `PARF`, représente la parenthèse fermante : `)`
- `IF`, représente le mot-clé `if`
- `THEN`, représente le mot-clé `then`
- `ELSE`, représente le mot-clé `else`

Les caractères séparateurs sont "espace" et "fin ligne".

Q1. Dessinez un automate reconnaissant les lexèmes du langage. Cet automate lit en entrée **une séquence de caractères** et il atteint un état final lorsqu'un **lexème** a été reconnu. Les transitions seront étiquetées uniquement par le caractère courant. On ne fera pas apparaitre le traitement des erreurs lexicales.

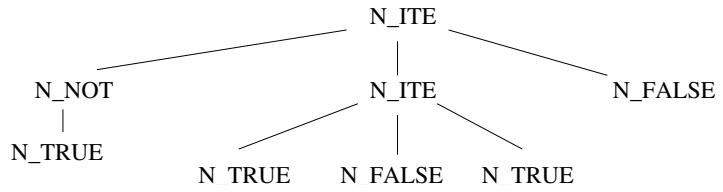

FIGURE 1 – AST de `if !tt then (if tt then ff else tt) else ff`

Partie 2 : analyse syntaxique (9 points)

La grammaire du langage est la suivante :

$$
\begin{array}{rcl}
Exp & \rightarrow & \texttt{IF}\ Terme\ \texttt{THEN}\ Terme\ \texttt{ELSE}\ Terme \\
Exp & \rightarrow & Terme \\
Terme & \rightarrow & \texttt{NOT}\ \ Facteur \\
Terme & \rightarrow & Facteur \\
Facteur & \rightarrow & \texttt{TRUE} \\
Facteur & \rightarrow & \texttt{FALSE} \\
Facteur & \rightarrow & \texttt{PARO}\ Exp\ \texttt{PARF}
\end{array}
$$

Q2. Dessinez `l'arbre de dérivation` obtenu pour l'expression E suivante :

```
if !tt then (if tt then ff else tt) else ff
```

Q3. Donnez un exemple d'expression comportant une erreur lexicale et un exemple d'expression comportant une erreur syntaxique.

Q4. En utilisant les primitives de `analyse_lexicale.h`, fournies en Annexe A, écrivez le corps de la procédure `analyser` spécifiée ci-dessous. Vous pouvez écrire des procédures auxilliaires (comme `Rec_Exp`, `Rec_Terme` et `Rec_Facteur`). Cette question peut être groupée avec la question Q5

```
void analyser(char *nom_fichier) ;
// e.i. : indifferent
// e.f. : une sequence de lexemes a ete lue dans le fichier nom_fichier,
//         une erreur syntaxique est signalee  si elle ne respecte pas la grammaire
```

Partie 3 : construction de l'arbre abstrait et évaluation (8 points)

L'**arbre abstrait** (Ast) de l'expression `if !tt then (if tt then ff else tt) else ff` est représentée sur la Figure 1. Vous noterez en particulier :

- qu'un noeud de nature `N_NOT`, représentant l'opérateur de négation, n'a qu'un seul fils (central)
- qu'un noeud de nature `N_ITE`, représentant l'opérateur if-then-else a trois fils (gauche, central, droit)
- que les noeuds de nature `N_TRUE` et `N_FALSE` sont des feuilles de l'arbre abstrait.

Q5. En vous aidant des fichiers `type_ast.h` et `ast_construction.h`, fournis en Annexe B, complétez le code de la question **Q4** pour que la procédure `analyser` fournisse en paramètre résultat l'arbre abstrait de l'expression lue (lorsque cette expresion ne contient pas d'erreurs) comme spécifié ci-après :

```
void analyser (char *nom_fichier, Ast *arbre) ;
// e.i. : indifferent
// e.f. : une sequence de lexemes a ete lue, une erreur est signalee si elle ne respecte pas
//        la grammaire, sinon arbre contient l'arbre abstrait de cette expression
```

Q6. Ecrivez une fonction `evaluer` qui prend en paramètre l'arbre abstrait d'une expression booléenne et qui renvoie sa valeur (0 pour faux, différent de 0 pour vrai). On rappelle les formules logiques suivantes :

$$\begin{aligned} \texttt{if } e1 \texttt{ then } e2 \texttt{ else } e3 &\equiv (e1 \Rightarrow e2) \wedge ((\neg e1) \Rightarrow e3) \\ e1 \Rightarrow e2 &\equiv (\neg e1) \vee e2 \end{aligned}$$

Annexe A : le fichier `analyse_lexicale.h`

```
typedef enum {TT, FF, NOT, PARO, PARF, IF, THEN, ELSE} Nature_Lexeme ;

typedef struct {
   Nature_Lexeme nature;    // nature du lexeme
   char chaine[256];           // chaine de caracteres
} Lexeme ;

void demarrer(char *nom_fichier);
// initialise l'analyse lexicale

void avancer();
// lit le lexeme suivant

Lexeme lexeme_courant();       // pourra etre abrege en "LC"
// valeur du lexeme courant

int fin_de_sequence();
// vrai ssi la fin de sequence est atteinte
```

Annexe B : le fichier `type_ast.h`

```
typedef enum {N_NOT, N_ITE, N_TRUE, N_FALSE} TypeAst ;

typedef struct noeud {
TypeAst nature ;
struct noeud *gauche, *centre, *droit ;
} NoeudAst ;

typedef NoeudAst *Ast ;
```

Annexe C : le fichier `ast_construction.h`

```
#include "type_ast.h"

Ast creer_ite(Ast ast1, Ast ast2, Ast ast3) ;
// renvoie un arbre abstrait de nature N_ITE,  d'operande gauche ast1,
// d'operande central ast2 et d'operande droit ast3
```

```
Ast creer_negation(Ast ast) ;
// renvoie un arbre abstrait de nature N_NOT, d'operande central ast

Ast creer_vrai() ;
// renvoie un arbre abstrait "feuille", de nature N_TRUE

Ast creer_faux() ;
// renvoie un arbre abstrait "feuille", de nature N_FALSE
```