TDS - AST

1 Table des symboles

Soit le programme suivant écrit dans un langage impératif à structure de blocs : dessiner les tables de symboles lors de l'analyse de l'instruction `retourne( P2(a+b+z) )`

PP

```
Programme PP
   var a, b, c : entier

   fonction P1(a: entier; z: entier): entier
     var x: entier

     fonction P2(aa: entier): entier
      debut
       retourner( Q1(aa+1)
      fin

     fonction Q1(x: entier): entier
      var aa: entier
      debut
       aa := x
       retourne( aa )
      fin

    debut
     retourne( P2(a+b+z) )
    fin
 debut
  a := 10
  b := 20
  c := P1(a,b)
 fin
```

2 Arbre abstrait

Soit la grammaire suivante qui reconnaît un sous-ensemble d'instruction du langage C (Nous donnons le fichier au format Antlr expleC.g)

- **Question 1.** Définir la grammaire abstraite correspondant à ce langage
- **Question 2.** Introduire dans la grammaire donnée ci-dessus les actions permettant de construire l'AST
- **Question 3.** Dessiner l'AST correspondant au texte source suivant :

Exp

```
char c;
int x;
int f(int y, char a) {
  int i;
  for(i=0; i!=3; i=i+1) {
    x = 3;
    y = 5;
  }
}
```

- **Question 4.** Définir la grammaire d'arbre (*tree-grammar*) correspondant à la grammaire précédente.
- **Question 5.** Écrire les actions permettant d'imprimer toutes les variables et fonctions définies dans un texte source analysé. Par exemple, sur le texte source précédent, la sortie doit être la suivante :

TdS

```
variable: char c
variable: int x
variable: int i
fonction: f()
```

3 Annexes

expleC.g

```
grammar expleC;

options { langage = C;
          output = AST;
          ...
}

tokens {
        // ajouter ici les "tokens imaginaires" nécessaires
}

//Définitions des expressions régulières reconnaissant les tokens
ID  : ('a'..'z' | 'A'..'Z')('a'..'z' | 'A'..'Z' | '0'..'9')* ;
INT : ('0'..'9')+ ;
WS  : (' ' | '\t' | '\n' | '\r' )+ {$channel = HIDDEN;}

program
   : declaration+
   ;

declaration
   : variable
   | function
   ;

variable
   : type ID ';'
   ;

type
   : 'int'
   | 'char'
   ;

function
   : type ID '(' (formalParam (',' formalParam)* )? ')' block
   ;

formalParam
   : type ID
   ;

block
   : '{' variable* inst* '}'
   ;

instr
   : forInst
   | expr ';'
   | block
   | affect ';'
   | ';'
   ;

forInst
   : 'for' '(' affect ';' expr ';' affect ')' block
   ;

affect
   : ID '=' expr
   ;

expr
   : condExpr
   ;

condExpr
   : expr1 ( ('==' | '!=') expr1 )?
   ;

expr1
   : expr2 ('+' expr2)*
   ;

expr2
   : atom ('*' atom)*
   ;

atom
   : ID
   | INT
   | '(' expr ')'
   ;


```