PROJET DE COMPILATION - ANALYSE SYNTAXIQUE LL(1)

1. Objectif

L'objectif de ce TP est de programmer l'analyseur syntaxique complet du langage L. Cet analyseur doit accepter uniquement des programmes bien formés syntaxiquement, selon la grammaire, auquel cas il donne en sortie l'arbre de dérivation sous format XML. Il doit afficher un message d'erreur et interrompre l'analyse en cas de désaccord avec la grammaire.
Nous utiliserons comme point de départ l'analyseur syntaxique des expressions simples. Le principe à appliquer est toujours identique, avec une fonction par non terminal de la grammaire.
Nous vous conseillons d'utiliser la grammaire donnée à la fin de l'énoncé. Cela vous permettra de comparer la sortie de votre analyseur syntaxique avec les fichiers xml de référence.

2. La table d'analyse LL(1)

Pour rendre l'implémentation de notre analyseur plus systématique, nous vous conseillons d'utiliser la table d'analyse LL(1), qui rend l'application de chaque production déterministe.
Voici un exemple de fonction pour un non terminal A dont les productions sont affichées ci-dessous :

$A \longrightarrow a\ B \mid C\ D \mid \varepsilon$

```
void A( void ) {
  // Cas 1 : La partie droite commence par un terminal
  if( uniteCourante == 'a' ) {
    uniteCourante = yylex();
    B();
  }
  // Cas 2 : La partie droite commence par un non terminal
  else if( uniteCourante est dans Premier(C) ) {
    C();    D();
  }
  // Cas 3 : La partie droite est epsilon
  else if( uniteCourante est dans Suivant(A) )
    return;
  else {
    printf( "Erreur de syntaxe" );
    exit( -1 );
  }
}
```

Pour vérifier qu'un symbole quelconque appartient à l'ensemble PREMIER ou SUIVANT d'un non terminal, vous pourrez utiliser les fonctions `est_premier` et `est_suivant` fournies. Ainsi, nous vous conseillons de remplir les tables fournies dans les fichiers `premiers.c` et `suivants.c`. Vous pouvez copier directement ce qui a été fait en TD, car la grammaire est similaire à celle que vous devez implémenter. La seule différence est que, dans la version à implémenter, reproduite ci-dessous, nous avons donné des noms plus longs et plus explicites pour les symboles non terminaux.
Attention : Si $\varepsilon \in$ PREMIER(C), alors il faut aller plus loin dans la partie droite, et appliquer cette production aussi si le symbole courant est dans PREMIER(D), et ainsi de suite.

3. Sortie XML

Pour générer l'arbre de dérivation, nous conseillons d'utiliser le format XML. Ainsi, à chaque fois que l'on entre dans une fonction correspondant à un non terminal, on ouvre une balise avec son nom. À la fin, avant tout return, on ferme cette balise. Vous pouvez utiliser les fonctions `affiche_balise_ouvrante` et `affiche_balise_fermante` fournies dans `util.c`, qui gèrent automatiquement l'indentation. Pour savoir le nom de la fonction où vous vous trouvez, vous pouvez utiliser la variable automatique de C `__FUNCTION__`. Pour afficher des symboles simples, préférez la fonction `affiche_xml_texte`, qui échappe les caractères spéciaux, notamment pour afficher le symbole inférieur `<`

4. Grammaire complète de L

Grammaire LL(1) –non ambiguë, non récursive à gauche et factorisée à gauche– du langage L.

```
pg -> odv ldf                #(1)  programme -> optDecVariables listeDecFonctions
odv -> ldv ';'               #(2)  optDecVariables -> listeDecVariables ';'
odv ->                       #(3)                  |
ldv -> dv ldvb               #(4)  listeDecVariables -> declarationVariable listeDecVariablesBis
ldvb -> ',' dv ldvb          #(5)  listeDecVariablesBis -> ',' declarationVariable listeDecVariablesBis
ldvb ->                      #(6)                        |
dv -> INT IDV ott            #(7)  declarationVariable -> ENTIER ID_VAR optTailleTableau
ott -> '[' NB ']'            #(8)  optTailleTableau -> '[' NOMBRE ']'
ott ->                       #(9)                    |
ldf -> df ldf                #(10) listeDecFonctions -> declarationFonction listeDecFonctions
ldf ->                       #(11)                   |
df -> IDF lp odv ib          #(12) declarationFonction -> ID_FCT listeParam optDecVariables instructionBloc
lp -> '(' oldv ')'           #(13) listeParam -> '(' optListeDecVariables ')'
oldv -> ldv                  #(14) optListeDecVariables -> listeDecVariables
oldv ->                      #(15)                       |
i -> iaff                    #(16) instruction -> instructionAffect
i -> ib                      #(17)              | instructionBloc
i -> isi                     #(18)              | instructionSi
i -> itq                     #(19)              | instructionTantque
i -> iapp                    #(20)              | instructionAppel
i -> iret                    #(21)              | instructionRetour
i -> iecr                    #(22)              | instructionEcriture
i -> ivide                   #(23)              | instructionVide
iaff -> var '=' exp ';'      #(24) instructionAffect -> var '=' expression ';'
ib -> '{' li '}'             #(25) instructionBloc -> '{' listeInstructions '}'
li -> i li                   #(26) listeInstructions -> instruction listeInstructions
li ->                        #(27)                      |
isi -> SI exp ALR ib osinon  #(28) instructionSi -> SI expression ALORS instructionBloc optSinon
osinon -> SIN ib             #(29) optSinon -> SINON instructionBloc
osinon ->                    #(30)           |
itq -> TQ exp FR ib          #(31) instructionTantque -> TANTQUE expression FAIRE instructionBloc
iapp -> appf ';'             #(32) instructionAppel -> appelFct ';'
iret -> RET exp ';'          #(33) instructionRetour -> RETOUR expression ';'
iecr -> ECR '(' exp ')'      #(34) instructionEcriture -> ECRIRE '(' expression ')' ';'
ivide -> ';'                 #(35) instructionVide -> ';'
exp -> conj expB             #(36) expression -> conjonction expressionBis
expB -> '|' conj expB        #(37) expressionBis -> '|' conjonction expressionBis
expB ->                      #(38)                 |
conj -> neg conjB            #(39) conjonction -> negation conjonctionBis
conjB -> '&' neg conjB       #(40) conjonctionBis -> '&' negation conjonctionBis
conjB ->                     #(41)                   |
neg -> '!' comp              #(42) negation -> '!' comparaison
neg -> comp                  #(43)           | comparaison
comp -> e compb              #(44) comparaison -> expArith comparaisonBis
compb -> '=' e compb         #(45) comparaisonBis -> '=' expArith comparaisonBis
compb -> '<' e compb         #(46)                 | '<' expArith comparaisonBis
compb ->                     #(47)                 |
e -> t eb                    #(48) expArith -> terme expArithBis
eb -> '+' t eb               #(49) expArithBis -> '+' terme expArithBis
eb -> '-' t eb               #(50)              | '-' terme expArithBis
eb ->                        #(51)              |
t -> f tb                    #(52) terme -> facteur termeBis
tb -> '*' f tb               #(53) termeBis -> '*' facteur termeBis
tb -> '/' f tb               #(54)           | '/' facteur termeBis
tb ->                        #(55)           |
f -> '(' exp ')'             #(56) facteur -> '(' expression ')'
f -> NB                      #(57)          | NOMBRE
f -> appf                    #(58)          | appelFct
f -> var                     #(59)          | var
f -> LIRE '(' ')'            #(60)          | LIRE '(' ')'
var -> IDV oind              #(61) var -> ID_VAR optIndice
oind -> '[' exp ']'          #(62) optIndice -> '[' expression ']'
oind ->                      #(63)            |
appf -> IDF '(' lexp ')'     #(64) appelFct -> ID_FCT '(' listeExpressions ')'
lexp -> exp lexpb            #(65) listeExpressions -> expression listeExpressionsBis
lexp ->                      #(66)                     |
lexpb -> ',' exp lexpb       #(67) listeExpressionsBis -> ',' expression listeExpressionsBis
lexpb ->                     #(68)                       |
```