PROJET DE COMPILATION - ANALYSE SYNTAXIQUE DES EXPRESSIONS

1. Objectif

L'objectif de ce TP est de programmer un analyseur syntaxique minimaliste pour les expressions arithmétiques en langage L. Nous traiterons pour le moment uniquement les expressions sur les nombres, avec les opérateurs somme ($+$) e multiplication ($*$), et avec gestion des parenthèses.

2. La Grammaire

Il existent plusieurs grammaires possibles pour reconnaître les expressions arithmétiques. Par exemple, la grammaire ci-dessous décrit le langage des expressions de manière assez intuitive :

$$E \longrightarrow E + E \mid E * E \mid (E) \mid nombre$$

Cependant, pour simplifier la traduction grammaire $\rightarrow$ analyseur syntaxique, il convient d'utiliser une grammaire non ambiguë, non récursive à gauche et factorisée, ce qui n'est pas le cas de la grammaire ci-dessus. La grammaire proposée ci-dessous respecte ces critères, et est donc une meilleure candidate pour guider notre implémentation :

$$\begin{array}{lll} E & \longrightarrow & T\ E' \\ E' & \longrightarrow & + E \mid \varepsilon \\ T & \longrightarrow & F\ T' \\ T' & \longrightarrow & \times F \mid \varepsilon \\ F & \longrightarrow & (E) \mid nombre \end{array}$$

3. L'analyseur syntaxique

Nous utiliserons une approche d'analyse prédictive descendante. Chaque non terminal de la grammaire sera directement traduit vers une fonction en C. Ainsi, notre analyseur aura cinq fonctions plus le `main`, qui fait appel à la fonction correspondant à l'axiome (E). Chaque fonction correspondant à un non terminal suit le schéma suivant :

```
void NonTerminal( void ) {
  if( entrée trouvée = entrée attendue ) {
    suite de l'analyse;
    return;
  }
  else {
    printf( "Erreur de syntaxe" );
    exit( -1 );
  }
}
```

À tout moment, l'analyseur syntaxique a accès à une variable globale contenant le lexème correspondant à l'unité courante. Une fois qu'elle a été reconnue, on peut demander à l'analyseur lexical de fournir la prochaine unité `uniteCourante = yylex()`

Dans le corps de chaque fonction, il faut implémenter les règles de la grammaire. Ainsi, si la partie droite de la règle contient un symbole terminal, l'analyseur syntaxique doit vérifier que l'unité courante est égale à ce symbole non terminal, et le consommer. Si la partie droite de la règle contient un symbole non terminal, nous faisons appel à la fonction correspondante.

Par exemple, pour le terminal A et ses règles de production $A \longrightarrow b\ C \mid C\ d$, nous aurons la fonction suivante :

```
void A( void ) {
  if( uniteCourante == 'b' ) {
    uniteCourante = yylex();
    C();
  }
  else {
    C();
    if( uniteCourante == 'd' ) {
      uniteCourante = yylex();
    }
    else {
      printf( "Erreur de syntaxe" );
      exit( -1 );
    }
  }
}
```

Les productions vides sont traduites par un `return` sans test.

4. Communication analyseur lexical - analyseur syntaxique

Nous allons dès maintenant utiliser l'analyseur lexical programmé au TP1. Cependant, nous n'utiliserons qu'un sous-ensemble de ses fonctionnalités comprenant les nombres, parenthèses et opérateurs. Par exemple, pour le moment nous ne traitons pas les mots-clefs.
Quand vous testez si l'unité courante est la parenthèse ouvrante, par exemple, au lieu de comparer directement le caractère, vous devez utiliser les constantes renvoyées par l'analyseur lexical. Autrement dit, vous ne devez pas écrire `if( uniteCourante == '(' )` mais plutôt `if( uniteCourante == PARENTHESE_OUVRANTE )`
Pour la clarté du code, nous vous conseillons de créer un fichier `analyseur_syntaxique.c` avec son en-tête `.h` correspondant. Ensuite, vous pouvez copier le code de `test_yylex.c` vers votre compilateur, en remplaçant la boucle d'affichage par l'appel à la fonction correspondant à l'axiome de la grammaire dans l'analyseur syntaxique.
N'oubliez pas les initialisations et terminaisons. Rappelez-vous que, avant de démarrer l'analyse, il faut initialiser l'unité courante avec le premier lexème du fichier à compiler. À la fin, il faut vérifier que vous avez atteint la fin de fichier en vérifiant que le symbole courant est la constante `FIN`

5. Tests et exemples

Vous pouvez tester votre analyseur sur des fichiers contenant des expressions comme ci-dessous :
`12 + 4 * (5 + 10)`
N'oubliez pas de tester votre analyseur sur des expressions mal formées. Par exemple, il doit afficher un message d'erreur l'entrée `10 * + 5`.
Pour vérifier votre analyse syntaxique, nous vous conseillons d'afficher l'arbre de dérivation. Pour cela, vous pouvez utiliser du XML, en affichant une balise ouvrante à l'entrée de la fonction et une balise fermante à la sortie. Cette balise porte le nom du non terminal de la fonction.

6. Extensions

Maintenant que vous avez un analyseur syntaxique minimaliste pour les expressions, vous pouvez étendre la grammaire et l'analyseur pour prendre en compte :

- Les variables
- Les autres opérateurs de même précédence en L (`-` et `/`)
- Les autres opérateurs de précédénce inférieure, comme les opérateurs de comparaison (`<` et `=`) et logiques (`&`, `|` et `!`)
- Les cases de tableaux
- Les appels à fonction