Devoir 3 : Analyse Syntaxique Récursive Descendante

Dû : le 22 mars à 23h45

SEG2506 – Construction de Logiciels

Partie A

Analyse Syntaxique Récursive Descendante

L'analyse syntaxique récursive descendante est une technique pour analyser une grammaire compatible avec les parseurs LL(1) sans calculer les ensembles FIRST et FOLLOW et sans créer un tableau syntaxique. Une grammaire LL(1) ne peut pas être ambiguë ou récursive à gauche, et doit être factorisée à gauche. Malgré que la technique soit plus facile à implémenter, les parseurs traditionnels LL(1) qui utilisent les tableaux syntaxiques fournissent une meilleure performance.

Considérez la grammaire suivante qu'on a déjà vue en classe :

```
<expr> ::= <term><expr'>
<expr'> ::= +<expr>
      | -<expr>
      | ε
<term> ::= <factor><term'>
<term'> ::= *<term>
      | /<term>
      | ε
<factor> ::= num
      | id
```

On peut produire un analyseur syntaxique récursif descendant simple à partir de la grammaire en associant une procédure avec chaque symbole non-terminal. Le pseudo code d'un tel mécanisme est montré ci-dessous. Notez que "token" est une variable globale partagée entre toutes les procédures données.

```
Procedure: main ()
      token ← getNextToken();
      if (expr() == ERROR || token != "$") then
            return ERROR;
      else
            return OK;
```

```
Procedure: expr ()
      if (term() == ERROR) then
            return ERROR;
      else return expr_prime();
```

```
Procedure: expr_prime ()
      if (token == "+") then
            token ← getNextToken();
            return expr();
      else if (token == "-") then
            token ← getNextToken();
            return expr();
      else return OK;
```

```
Procedure: term ()
      if (factor() == ERROR) then
            return ERROR;
      else return term_prime();
```

```
Procedure: term_prime()
      if (token == "*") then
            token ← getNextToken();
            return term();
      else if (token == "/") then
            token ← getNextToken();
            return term();
      else return OK;
```

```
Procedure: factor ()
            if (token == "num") then
                  token ← getNextToken();
                  return OK;
            else if (token == "id") then
                  token ← getNextToken();
                  return OK;
            else return ERROR;
```

Étudiez bien le pseudo code ci-dessus. Assurez-vous de bien comprendre tous ses détails.

Partie B

Étant donné la grammaire suivante pour un « **Very Simple Programming Language »** (VSPL) :

```
<program> ::= begin <statement_list> end
<statement_list> ::= <statement> ; <statement_list>
<statement_list> ::= <statement> ;
<statement> ::= id = <expression>
<expression> ::= <factor> + <factor>
<expression> ::= <factor> - <factor>
<expression> ::= <factor>
<factor> ::= id | num
```

Les symboles grammaticaux suivants sont des non-terminaux :

program
statement_list
statement
expression
factor

Les symboles grammaticaux suivants sont des terminaux :

begin
end
;
id
num
=
+
-

Ci-dessous, voici un échantillon de programme écrit en VSPL :

```
begin
      a = 15;
      b = 20;
      c = a + b;
end
```

Exercice 1 (15 points)

Convertissez la grammaire non-contextuelle **VSPL** en grammaire LL(1). Effectuez tous les modifications nécessaires (si nécessaire).

Exercice 2 (60 points)

Écrivez un programme Java qui effectue l'analyse syntaxique récursive descendante sur la grammaire LL(1) que vous avez produite précédemment. Votre analyseur syntaxique obtient ses tokens à partir d'un fichier (au lieu de les obtenir d'un analyseur lexical). Lorsque l'activité d'analyse syntaxique est terminée, votre programme doit afficher un des messages suivants :

- **SUCCESS**: le code a bien été analysé *(dans le cas de succès)*
- **ERROR**: le code contient une erreur syntaxique *(dans le cas d'un échec)*

Deux fichiers de données contenant une liste de tokens ont été attachés à ce devoir. Le premier est appelé `input1.txt` et ne contient pas d'erreurs syntaxiques. Le deuxième est appelé `input2.txt` et contient une erreur syntaxique (un point-virgule manque). Dans ces fichiers, chaque token est écrit sur une ligne séparée. Conséquemment, afin d'obtenir un token du fichier, vous pouvez simplement utiliser un appel `readline()`.

Notez que le programme Java développé doit recevoir le nom du fichier comme argument passé à sa méthode « main ».

Assurez-vous d'inclure vos fichiers .java dans votre soumission du devoir.

Exercice 3 (25 points)

1. Écrivez les ensembles FIRST et FOLLOW pour tous les non-terminaux de la grammaire LL(1) produite dans l'**Exercice 1**
2. Développez le tableau syntaxique pour la grammaire LL(1) produite dans l'**Exercice 1**

C'est tout, Bonne Chance !