TD sur l'analyse syntaxique février 2008

Prérequis : Grammaire LL(1), récurivité à gauche, factorisation à gauche, calcul des ensembles *Vide*, *Premier*, *Suivant*, des symboles directeurs, descente récursive.
Durée : 1 h 50

TD 18 – Analyseur descendant récursif

Analyseur récursif LL(1)

Préambule

Vous avez découvert dans ce module deux types d'analyseurs syntaxiques descendants :

- les analyseurs récursifs
- les analyseurs non-récursifs

L'analyse syntaxique par descente récursive est une méthode d'analyse syntaxique descendante dans laquelle on exécute un ensemble de procédures récursives pour traiter la chaîne d'entrée. Une procédure est associée à chaque non-terminal d'une grammaire. Une analyse syntaxique récursive LL(1) se décompose en deux parties :

(*i*) **construction de l'analyseur** : on écrit le programme principal, la procédure d'analyse d'un terminal ainsi qu'une procédure d'analyse pour chaque non-terminal

(*ii*) **reconnaissance d'un mot** : on appelle la procédure principale. Le résultat de l'exécution fournit, si le mot est reconnu, l'arbre syntaxique du mot analysé en notation postfixée

Étude de la grammaire

Soit $G = (N, T, \rightarrow, LIST\_EXP)$ une grammaire telle que :

- $N = \{LIST\_EXP, EXP, COND, BOUCLE, AFFECTATION, \ldots\}$
- $T = \{if, then, else, endif, for, to, do, endfor, =, +, -, *, /, >, <, ==, <=, >=, a, b, c, d, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- la relation de production $\rightarrow$ est définie par les règles suivantes :

$$
\begin{cases}
LIST\_EXP & \rightarrow \epsilon \mid EXP\ \ LIST\_EXP \\
EXP & \rightarrow COND \mid BOUCLE \mid AFFECTATION \\
COND & \rightarrow if\ \ COMP\ \ then\ \ LIST\_EXP\ \ endif \\
 & \quad \mid if\ \ COMP\ \ then\ \ LIST\_EXP\ \ else\ \ LIST\_EXP\ \ endif \\
BOUCLE & \rightarrow for\ \ AFFECTATION\ \ to\ \ IDENT\ \ do\ \ LIST\_EXP\ \ endfor \\
AFFECTATION & \rightarrow VAR\ = \ OPERATION \mid VAR\ = \ IDENT \\
OPERATION & \rightarrow OPERATION\_BINAIRE \mid OPERATION\_UNAIRE \\
OPERATION\_BINAIRE & \rightarrow IDENT\ \ OPERATEUR\_BINAIRE\ \ IDENT \\
OPERATION\_UNAIRE & \rightarrow OPERATEUR\_UNAIRE\ \ IDENT \\
OPERATEUR\_BINAIRE & \rightarrow + \mid - \mid * \mid / \\
OPERATEUR\_UNAIRE & \rightarrow - \\
COMP & \rightarrow IDENT\ \ COMPARATEUR\ \ IDENT \\
COMPARATEUR & \rightarrow < \mid > \mid == \mid >= \mid <= \\
IDENT & \rightarrow VAR \mid CONSTANTE \\
VAR & \rightarrow a \mid b \mid c \mid d \\
CONSTANTE & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{cases}
$$

1. Deux propriétés de cette grammaire, que l'on peut voir sans entammer de calcul, font qu'elle n'est pas LL(1). Lesquelles ?
2. Résolvez ce problème afin d'obtenir, a priori, une grammaire LL(1) équivalente à G.
3. Effectuez les calculs qui permettent de vérifier que la grammaire obtenue est LL(1).

4. Y'a-t-il un problème ? Intuitivement, pensez-vous que la grammaire obtenue est LL(2) ?
5. Résolvez ce problème afin d'obtenir, enfin, une grammaire LL(1) équivalente à G.

Descente récursive

On dispose d'une classe `Analyseur` comportant les attributs suivants :

- `boolean erreur`
- `String res`
- `String y`

Ainsi que de la fonction `void lire()` qui affecte à `y` le terminal en cours de lecture. A partir de ces données, complétez la classe `Analyseur` en écrivant la descente récursive qui permet d'analyser la grammaire obtenue précédement et qui, en cas de réussite, retourne l'arbre syntaxique du « mot » analysé en notation postfixée.

Résultat

Exécuter l'analyse sur le texte suivant :

```
if a == b then
      a = ( c * d )
else
      a = b
endif
for a = 1 to b do
      a = ( c + 1 )
endfor
$
```

Appartient-il au langage engendré par G ? Si oui, dessinez son arbre syntaxique.