# SEG 2506

**Construction de logiciels**

# DEVOIR 3 :

# Compilateur d'un sous ensemble Pascal

présenté à:

prof. G. Bochmann, Ph.D.

Note:
110%


par:

Renaud Bougueng Tchemeube

#4333634

rboug039@uottawa.ca

et

Boniface Mbouzao

#4275992

mbouzaoboni@hotmail.com

École d'Ingénierie et de Technologie de l'Information.

Université d'Ottawa.


Le lundi 24 mars 2008

Commentaire :
Hors-pair! Bonnes observations sur les environnements de développement. En passant, j'utilise flex pour traitement de la langue.

Suggestion :
continuez de même et commencez à observer les professeurs pour trouver un superviseur de maîtrise. Cherchez quelqu'un qui pourra vous guider dans un champ d'études sérieux et avec des possibilités professionnelles véritables.

Bonus :
-saisie des buts explicites et implicites de l'exercice : 5pts

SEG 2506 – hiver 2008
Barême de correction – devoir 3

- 5/5 grammaire LL(1) équivalente réécrite
- 5/5 First & Follow
- 5/5 table d'analyse
- 5/5 attributs sémantiques
- 5/5 règles d'évaluation
- 6/6 programme d'analyse syntaxiques
  - 2/2 fichier input.txt correct fourni
  - 2/2 fichier output.txt correct fourni
- 6/6 programme d'analyse lexical
  - 2/2 fichier input.txt corrct fourni
  - 2/2 fichier output.txt correct fourni (incluant rapport d'erreur s'il y a lieu)
- 5/5 Explications pour chaque élément du travail à remettre

# TABLES DES MATIERES

**INTRODUCTION....3**

**ANALYSE DE LA GRAMMAIRE PROPOSEE....3**

**GRAMMAIRE LL(1) ....4**

Définition des Terminaux :....6
Classe de caractères (pour Lex):....7

**FIRST ET FOLLOW....7**

**TABLE D'ANALYSE SYNTAXIQUE....8**

**ATTRIBUTS SEMANTIQUES....9**

**REGLES D'EVALUATION....9**

**DISCUSSION....10**

Le choix de Java (Eclipse pour le JDE)?....11
Le choix de Lex?....11
Le choix d'un tableau à 3D ?....11
Les problèmes dans l'approche d'implémentation ?....12
Les compromis (efficacité, flexibilité etc)?....12
La connexion Lex – Java ?....13
Langage d'expression du compilateur ?....13

**EXECUTION DU COMPILATEUR....13**

**ANNEXE....13**

GRAPHES....13
*Schéma d'approche du compilateur....13*
*Graphe syntaxique....14*
CODE-SOURCE....15
*Fichier d'exécution principale « compiler »....15*
*Fichier Lex....16*
*Classe Symbol....19*
*Classe NonTerminal....20*
*Classe Terminal....20*
*Classe ParsingTable....20*
*Classe PascalCompiler....25*
DOCUMENTS UTILISES....31
*Figure 1 - Eliminer la récursivité à gauche....31*
*Figure 2 - Algorithme de Aho....32*
*Figure 3 – Exemple de définition d'attributs sémantiques....33*

**REFERENCES....33**

# INTRODUCTION

Il s'agit dans ce devoir de concevoir un compilateur pour un sous-ensemble Pascal. Dans la liste des non-terminaux de la grammaire donnée, nous avons, comme recommandé, enlevé les suivants: **identifier_list, declarations, type, standard_type, procedure_statement, if-statement, expression_list, sign**. Et les terminaux : **not, and, or, div,** et **mod** ont aussi été enlevés. Puis, nous avons tenu compte du fait que le terminal **num** devrait seulement considérer des *entiers*, et pas de valeurs *réelles*.

Néanmoins, l'objectif de ce devoir n'était pas moins de comprendre l'importance de grammaires hors-contexte, leur potentialité pour l'analyse de langage et les enjeux qu'elles impliquent dans le domaine d'application de leur programmation, que le simple fait de programmer un compilateur Pascal. Ceci dit, la conception d'un tel compilateur demande de passer par plusieurs étapes indispensables pour une bonne implémentation. Mais premièrement, commencer par analyser la grammaire proposée.

# ANALYSE DE LA GRAMMAIRE PROPOSEE

La grammaire proposée est une grammaire assez simple dans la mesure où elle est aisément compréhensible par le simple « homme de science ». Et il apparait clairement que les termes employés pour sa définition ont été minimisés. Mais, pour une implémentation d'un analyseur syntaxique pour cette grammaire, il sera préférable (voire même conseillé), de trouver une grammaire LL(1) équivalente.

Voici la grammaire simplifiée donnée :

*program* → program id ;
compound_statement
.

compound_statement → **begin**

optional_statements
**end**

optional_statements → statement_list | ϵ

statement_list → statement | statement_list **;** statement

statement → variable **assignop** expression
| compound_statement
| **while** expression **do** statement

variable → **id**

expression → simple_expression | simple_expression **relop** simple_expression

simple_expression → term | simple_expression **addop** term

term → factor | term **mulop** factor

*factor* → **id**
| num
| **(**expression**)**

# GRAMMAIRE LL(1)

La grammaire LL(1), comme son nom l'indique, consiste à analyser la chaine de caractère à évaluer en la parcourant de gauche à droite (**L**eft-to-right) et en ne considérant que, le non-terminal le plus à gauche dans l'évaluation (**L**eft-most derivation), et le prochain caractère (un **seul** (1) caractère) de la chaine étudiée.
L'avantage ou plus exactement, la nécessité d'une telle grammaire s'explique par le fait qu'elle permet une analyse syntaxique systématique, et de ce fait, la possibilité de programmer la vérification de grammaire avec les langages standards.
Cependant, L'approche LL(1) demande une certaine syntaxe (méta-syntaxe vu que c'est la syntaxe des règles pour les syntaxes) à respecter dans la définition des règles pour permettre l'élaboration d'un algorithme systématique et efficace: Eviter les ambigüités et les récursivités à gauche. Il est donc évident que la grammaire proposée n'est pas une grammaire LL(1).

En effet, on remarque que certaines règles sont ambigües :

Ex. expression → simple_expression | simple_expression **relop** simple_expression

On décèle aussi de nombreuses récursivités à gauche susceptible d'engendrer des boucles infinies dans une implémentation avec approche LL(1).
Ex. 1) statement_list → statement | statement_list **;** statement
2) simple_expression → term | simple_expression **addop** term
3) term → factor | term **mulop** factor

Nous avons donc appliqué les transformations suivantes vues en cours pour transformer cette grammaire :
Pour l'ambigüité :
A --> a B c | a B A *peut être remplacé par*
A --> a B A' et A' --> C | A.

Pour la récursivité :
A --> A $\alpha_1$ | A $\alpha_2$ | … | A $\alpha_m$ | $\beta_1$ | $\beta_2$ |…| $\beta_n$ *peut être remplacé par*
A --> $\beta_1$ A' | $\beta_2$ A' | …| $\beta_n$A'
A' --> $\alpha_1$ A' | $\alpha_2$ A' |… $\alpha_m$ A' | ε

Voila donc ce que l'on obtient :

*program* → program id ;
compound_statement
.

compound_statement → **begin**
optional_statements
**end**

optional_statements → statement_list | ϵ

statement_list → statement recpart_statement
recpart_statement → **;** statement recpart_statement
| ϵ

statement → **id assignop** expression
| compound_statement
| **while** expression **do** statement

expression → simple_expression alt_Expression
alt_expression → **relop** simple_expression | ϵ

simple_expression → term recpart_simpleExpression
recpart_simpleExpression → **addop** term recpart_simpleExpression | ϵ

term → factor recpart_term
recpart_term → **mulop** factor recpart_term | ϵ

*factor* → **id**
| num
| **(**expression**)**

Nous nous sommes ensuite assuré que la grammaire obtenu était bien LL(1), c'est-à-dire :

Les ensembles *First* des parties droites de toutes les alternatives du non-terminal sont disjoints.
Si le *First* d'une partie droite contient *vide* alors le *Follow* du non-terminal doit être disjoint avec tous les *First* des parties droites.

## Définition des Terminaux :

Il est important de recenser clairement les terminaux utilisés par la grammaire :
program
begin
end
while
do
;
.
(
)
:= aussi appelé **assignop**
Groupe de terminaux
id
num
relop
addop
mulop

## Classe de caractères (pour Lex):

Comme vous avez pu le remarqué, certains terminaux peuvent regrouper en classe de caractères. Cette astuce rendre plus facile, l'analyse lexicale du programme. En fait, cette approche est idéale pour l'outil Lex. Lex est une application permettant de générer un analyseur lexicale. Il compile en langage C, le fichier écrit. Cela automatise et minimise donc l'effort de développement à effectuer par le programmeur car Lex propose un langage pseudo-descriptif (une partie peut aussi être écrite en C).

Voici donc les classes de caractères utilisables pour Lex, qui nous permettront d'obtenir un analyseur lexical:

| | |
|---|---|
| LETTER | [a-zA-Z] |
| DIGIT | [0-9] |
| DIGITS | {DIGIT}{DIGIT}* |
| OPTIONAL_FRACTION | .{DIGITS} \| ϵ |
| OPTIONAL_EXPONENT | (E (+\|-\|ϵ) {DIGITS})\| ϵ |

| | |
|---|---|
| **id** | {LETTER}({LETTER}\|{DIGIT})* (à définir la longueur limite) |
| **num** | {DIGITS}{OPTIONAL_FRACTION}{OPTIONAL_EXPONENT} |
| **relop** | {=, <>, <, <=, >=, >} |
| **addop** | {+, -} |
| **mulop** | {*, /} |

# FIRST ET FOLLOW

Le FIRST d'un non-terminal d'une grammaire est le premier terminal que peut générer la partie droite de ce non-terminal.
Le FOLLOW d'un non-terminal est tout simplement le premier terminal susceptible de suivre ce non-terminal suivant les règles définies par sa grammaire.

Le calcul du FIRST et du FOLLOW est une étape capitale dans la conception du compilateur parce qu'elle permet d'ériger la table d'analyse qui sera utilisée ultérieurement dans notre travail. Ils nous permettent de définir l'action systématique à prendre pour chaque couple (non-terminal ; terminal) donné rencontré durant l'analyse

Les calculs de FIRST et de FOLLOW pour chaque non-terminal ont donc donné ceci:

FIRST (*program*) = {program}

FIRST (*compound_statement*) = {begin}

FIRST (*optional_statements*) = {id, begin, while, ϵ}

FIRST (*statement_list*) = FIRST (*statement*) = {id, begin, while}

FIRST (*recpart_statement*) = {;, ϵ }

FIRST (*expression*) = FIRST(*simple_expression*) = FIRST(*term*) = FIRST(*factor*) = {id, num, (}

FIRST (alt_*expression*) = {relop, ϵ}

FIRST (recpart_simpleExpression) = {addop, ϵ}

FIRST (*recpart_term*) = {mulop, ϵ}

FOLLOW (*program*) = {$}

FOLLOW (*compound_statement*) = {., ;, end}

FOLLOW (optional_statements) = FOLLOW (statement_list) = FOLLOW (recpart_statement) = {end}

FOLLOW (*statement*) = {;, end}

FOLLOW (*expression*) = FOLLOW (alt_*expression*) = {do, ), ;, end }

FOLLOW (*simple_expression*) = FOLLOW (*recpart_simpleExpression*) = {do, ), ;, end, relop }

FOLLOW (*term*) = FOLLOW (*recpart_term*) = {addop, do, ), ;, end, relop }

FOLLOW (*factor*) = {mulop, addop, do, ), ;, end, relop}

# TABLE D'ANALYSE SYNTAXIQUE

A partir des First et Follow obtenus, nous avons été en mesure de déterminer la composition de la table d'analyse. Les cellules vides dans la table correspondent aux erreurs de syntaxe. La construction de la table d'analyse demande de la concentration et de la minutie. Une erreur dons la table peut engendrer des bogues dans le

programme et son débogage est d'autant plus difficile lorsqu'on a beaucoup de symbole dans la grammaire (Terminaux et non-Terminaux).
cf. Fichier Excel « Table d'analyse.xls » dans le dossier principal.

# ATTRIBUTS SEMANTIQUES

Définir des attributs sémantiques dans une grammaire permet d'assurer le respect de la sémantique d'un programme. En effet, un programme a une sémantique qui est directement relié à son utilité et le paradigme dans lequel il s'applique. Dans notre cas particulier (le compilateur d'un sous-ensemble Pascal), il s'agit de s'assurer que le programme pascal analysé respecte les contraintes sémantiques d'un tel langage impératif. Plus précisément, vérifier les concordances de types dans les contextes de compréhension particuliers et dans l'usage des opérateurs sur ces types.

# REGLES D'EVALUATION

Les règles d'évaluation des attributs définies pour réaliser l'évaluation statique des expressions sont les suivantes :

1)
statement → **id assignop** expression
Règle sémantique: <expression>.expected_type ← <id>.actual_type

2)
statement → **while** expression **do** statement
Règle sémantique: <expression>.expected_type ← boolean

3)
expression → simple_expression(1) alt_Expression
alt_expression → **relop** simple_expression(2)

Règle sémantique: <expression>.actual_type ←
if(< simple_expression(1)>.actual_type = <simple_expression(2)>.actual_type)
then boolean
else error
end if

Predicat: <expression>.actual_type = <expression>.expected_type

4)
simple_expression → term(1) recpart_simpleExpression
recpart_simpleExpression → **addop** term(2) recpart_simpleExpression

Règle sémantique : <simple_expression>.actual_type ←
if (< term(1)>.actual_type = double) and
(<term(2)>.actual_type = double)
then double
else error
end if

Predicat: <simple_expression>.actual_type = <simple_expression>.expected_type

5)
term → factor recpart_term
recpart_term → **mulop** factor recpart_term

Règle sémantique 1: < term>.actual_type ←
if(< factor(1)>.actual_type = double) and
(<factor (2)>.actual_type = double)
then double
else error
end if

Predicat : < term>.actual_type = <term>.expected_type

6)
factor → **id**
Predicat: <factor>.actual_type = <id>.actual_type

7)
*factor* → **num**
Predicat: <factor>.actual_type = double

factor → **(**expression**)**
Predicat: <expression>.actual_type = <factor>.actual_type

# DISCUSSION

## Le choix de Java (Eclipse pour le JDE)?

En théorie, le choix d'un langage de programmation pour la construction d'un système logiciel devrait se faire suivant les exigences et les contraintes que l'équipe de développement s'est fixées. Mais, en réalité, et comme nous l'a souvent dit un professeur, le choix du langage dépend malheureusement de qui connaît plus quel langage. Donc « **oui** », nous avons choisi Java en partie (75% environ) parce que l'on est plus à l'aise avec. Qui voudrait s'aventurer dans un langage qu'il ne maitrise pas ; en plus, lorsque un programme comme un compilateur n'est pas de tout repos pour le cerveau. Mais, quand même, Java dispose de la **portabilité**, d'un **API riche**, et du fait qu'il est totalement **orienté-objet** ; un choix de paradigme qui, comme on le verra, nous facilitera grandement la vie.
Le choix d'Eclipse s'explique beaucoup pour sa réputation, son large champ d'utilisation, sa capacité **automatique** de **compilation**, sa **rétroaction** précise et de sa grande aide dans le débogage.

## Le choix de Lex?

Nous pensons que la réponse à cette question se trouve plus haut. En effet Lex est un générateur d'analyseur syntaxique. Il est très flexible et malléable. C'était l'outil rêvé pour l'analyse lexicale. Nous voulons juste mentionné que puisque qu'il nous fallait connaître le type de chaque lexème retourné par Lex pour opérer avec la table d'analyse, nous avons décidé que Lex retournerait sur une **seule** ligne, une succession d'entiers indiquant le type de l'élément suivant, et le contenu de l'élément.
Ex.
```
begin
    Var1 := 2 * X
end.
```

Devient : 1 begin 10 Var1 9 := 12 2 13 * 10 X 2 end 6 .
(cf. Section sur la définition des Terminaux).

## Le choix d'un tableau à 3D ?

La modélisation de la table d'analyse syntaxique n'a pas été de tout repos. En effet, c'est conceptuellement une table à 2D dont chaque cellule peut contenir un nombre variable de données non-primitives. La question reflexe survient donc : Comment modéliser

cela ? Et quels sont les contraintes de qualités que doivent respecter le choix d'implémentation ?
Nous parlons d'une table qui peut regrouper des centaines voire des milliers de cellules contenant des informations complexes. Dans notre cas, nous avions de multiples choix avec la réutilisation de types abstraits de données classiques de l'API Java. Je pense ici à un tableau 2D contenant des listes ou des piles, une liste contenant un tableau de tableau, une table de hachage contenant des listes, ou encore, une composition plus homogène avec trois listes imbriquées. Mais bien que chacun de ces choix semblait tentant, il n'était pas difficile de remarquer que de telle implémentation rendrait l'accès lent et qu'une table vaste provoquerait rapidement un alourdissement du système. Nous avons finalement opté pour l'efficacité : Un tableau à 3D. Ce dernier permet, à notre avis, un accès plus direct et rapide, et est manipulé plus facilement par le système vu que les tableaux sont des « built-in » chez Java. Il est en outre plus facile à utiliser lors de la fastidieuse initialisation de la table.

## Les problèmes dans l'approche d'implémentation ?

L'autre problème similaire à celle de la table d'analyse est le problème des symboles. En considérant que ces symboles sont ceux qui constituent les index de la table d'analyse, on aurait opté pour des structures primitives telles que les entiers, qui iraient parfaitement avec le tableau à 3D. Cependant, il a été demandé dans le devoir d'afficher en sortie chaque règle syntaxique employé par le programme pour une trace de son analyse. Cette contrainte nous exige de revoir nos considérations.
Nos symboles doivent pouvoir être à la fois affichable en langage naturelle, et servir d'index. Ce qui est un parfait exemple d'objet. Nos symboles encapsulent des attributs qui les définissent : Un entier pour les index, et une chaine de caractères pour leurs noms dans la grammaire. De cette manière, nous avons résolu ce problème. Cette implémentation apportait de nombreux avantages. Entre autres, une meilleure lisibilité de la table d'analyse dans le programme qui présentait maintenant des index référencés en langage humain, au lieu d'avoir de longs paragraphes de chiffres qui prendraient probablement trois fois plus de temps pour un débogage en cas d'erreurs.

## Les compromis (efficacité, flexibilité etc)?

Nous devons concevoir un compilateur. Donc dans un contexte pratique, il se doit d'être flexible, portable, mais aussi rapide, efficace dans l'analyse, et précis dans la rétroaction avec l'usager. Evidemment, nous n'en sommes pas là, mais nous avons essayé au maximum de respecter ces critères. Ce genre d'exigences de qualité sont affaire

commune pour ce type logiciel, mais demande toujours autant de prises de décisions et de concessions.

## La connexion Lex – Java ?

Pour relier la sortie de Lex avec la machine virtuelle Java, il nous a fallu écrire un petit script (bash file) contenant des exécutions systèmes Linux. Cela permet de lancer l'analyseur avec la simple commande « compile ».

## Langage d'expression du compilateur ?

Equation quasi-triviale : nous avons choisi l'anglais comme langage d'expression pour la communication avec l'usager dans l'interface de commande. Evidemment, un plus gros compilateur destiné à un usage commercial dans le monde se devrait d'être polyglotte et demanderait de supporter plusieurs langues humaines différentes. Mais, une fois de plus, on n'en est pas là.

# EXECUTION DU COMPILATEUR

L'utilisation de Lex demande d'utiliser d'exécuter le compilateur dans un environnement Linux. Pour ce faire :

- Copier votre fichier-programme à compiler dans le répertoire contenant l'analyseur et le renommer « Program.txt ».
- Taper ensuite la commande « compile » dans le Terminal Linux.

Vous pouvez aussi utiliser les programmes-tests soumis en le renommant aussi « Program.txt ».
Le fichier « LexicalOutput.txt » contient l'analyse lexicale (effectuée par Lex) du programme à compiler. Vous pouvez le consulter pour vérifier l'exécution.

# ANNEXE

## GRAPHES

**Schéma d'approche du compilateur**

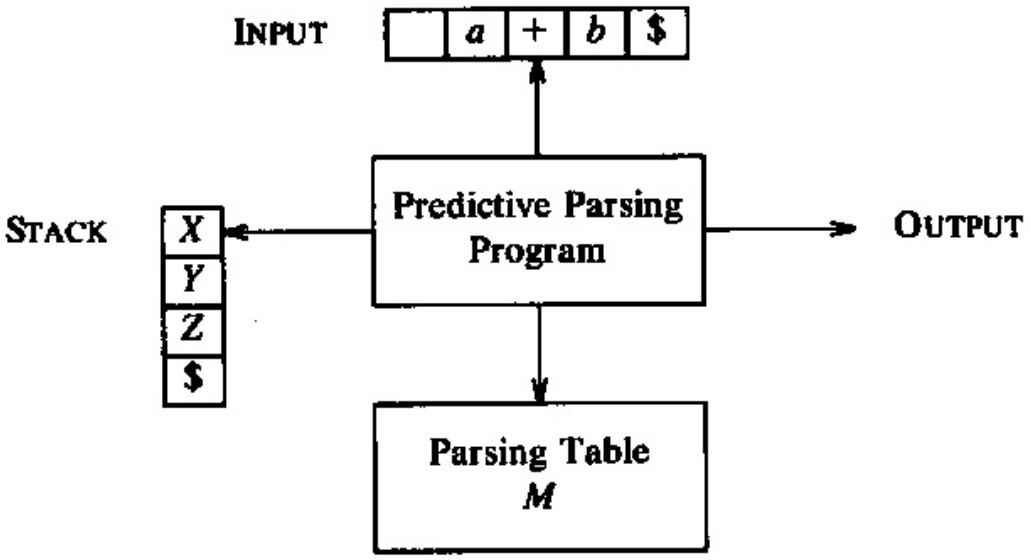
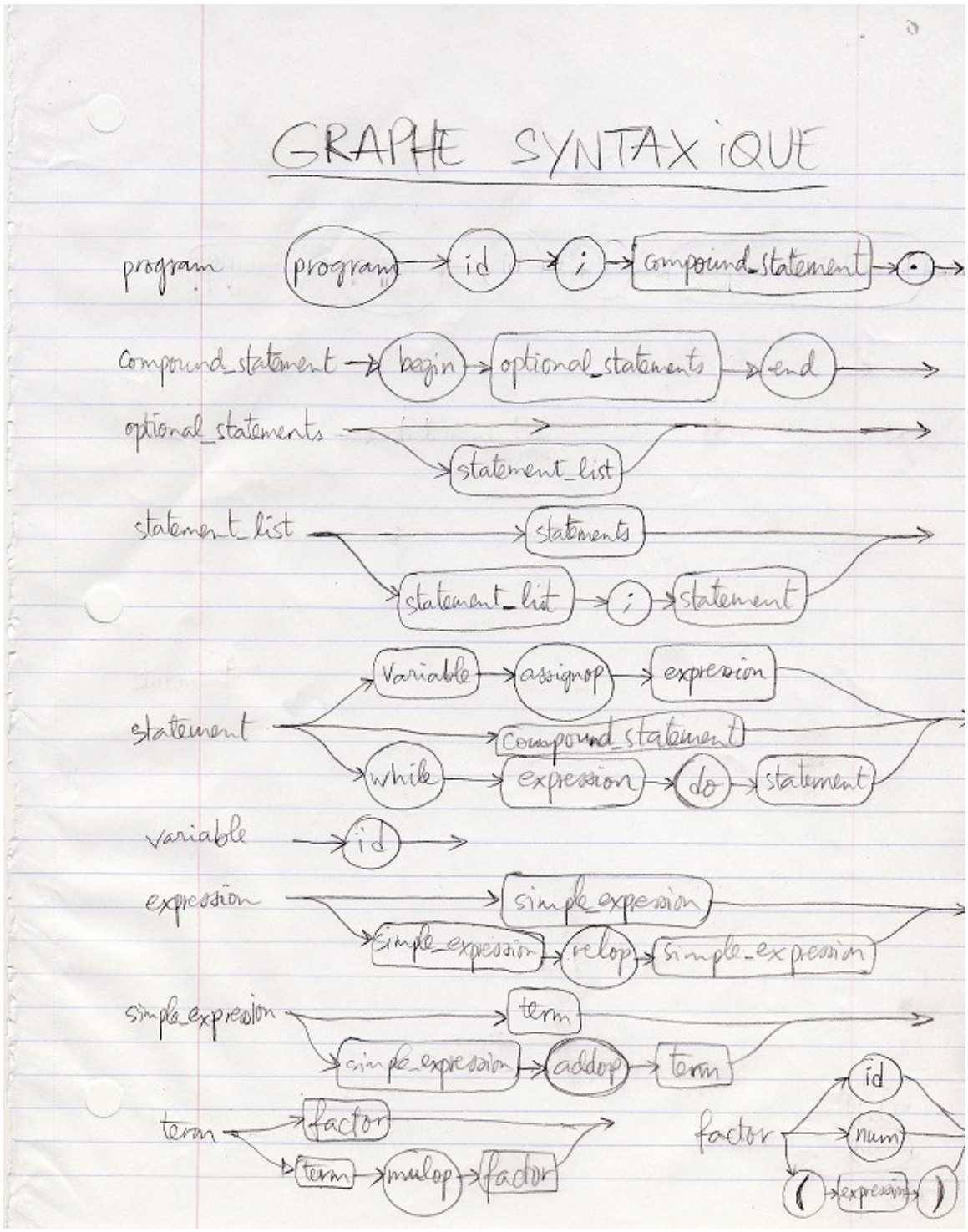



**Fig. 4.13. Model of a nonrecursive predictive parser.**

**Graphe syntaxique**




## CODE-SOURCE

**Fichier d'exécution principale « compiler »**

Permet d'exécuter complètement le compilateur Pascal :

```
#!/bin/bash
flex AnalyseurPascal.lex
gcc -o AnalyseurPascal lex.yy.c
AnalyseurPascal < Program.txt > LexicalOutput.txt
java PascalCompiler LexicalOutput.txt
```

## Fichier Lex

Permet d'effectuer l'analyse lexicale du programme et génère le fichier LexicalOutput.txt :

```
/* File........ AnalPascal.lex
 * Contents.... Analyseur lexical couvrant un sous-ensemble specifique
du langage Pascal
 */

/* compilation:
 *    flex AnalPascal.lex
 *    gcc -o AnaPascal.exe lex.yy.c
 */


/* ---------------- Definitions space ----------------- */
%option noyywrap
%{
#include <assert.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

  int comment = 0; /*zones de commentaires de style  */

/*Mots reserves de Pascal*/

  int Program = 0;
  int Begin = 1;
  int End = 2;
  int While = 3;
  int Do = 4;
  int PointVirgule = 5;
  int Point = 6;
  int ParentheseOuvrante = 7;
  int ParentheseFermante = 8;
  int Assignop = 9;
  int Id = 10;
  int Num = 11;
  int Relop = 12;
  int Addop = 13;
  int Mulop = 14;
```

```
%}

LETTER  [a-zA-Z]
DIGIT   [0-9]
DIGITS  {DIGIT}+
OPTIONAL_FRACTION  (.{DIGITS})?
OPTIONAL_EXPONENT  (E({ADDOP})?{DIGITS})?

ID      {LETTER}({LETTER}|{DIGIT})*
NUM     {DIGITS}{OPTIONAL_FRACTION}{OPTIONAL_EXPONENT}

KEYWORDS ((p|P)(r|R)(o|O)(g|G)(r|R)(a|A)(m|M))|((b|B)(e|E)(g|G)(i|I)(n|
N))|((e|E)(n|N)(d|D))|((w|W)(h|H)(i|I)(l|L)(e|E))|((d|D)(o|O))

APOST            ["]
RELOP            ["="|"<"|">"]
ADDOP            [+-]
MULOP            [*/]

/* ------------------- Rules space -------------------- */
%%

{APOST}        {if(comment != 1){
            printf("\n>> Lexical Error: Invalid token - %s.\n", yytext);
                  yyterminate();
                  }
             }


"{"           {if(comment == 0){
            comment = 1;
            }else{
            printf("\n>> Lexical Error: Cannot put '{' after '{'.\n");
            yyterminate();
            }
             }
"}"           {if(comment == 1){
            comment = 0;
            }else{
      printf("\n>>Lexical Error: Cannot put '}' without a '{'
before.\n");
            yyterminate();
            }
             }

";"           {if(comment != 1){
            printf(" %i ;", PointVirgule);
            }
             }

"."            {if(comment != 1){
            printf(" %i .", Point);
            }
             }

"("            {if(comment != 1){
```

```
            printf(" %i (", ParentheseOuvrante);
            }
             }

")"            {if(comment != 1){
            printf(" %i )", ParentheseFermante);
            }
             }

":="           {if(comment != 1){
            printf(" %i :=", Assignop);
            }
             }

"<>"       {if(comment != 1){
                  printf(" %i %s", Relop, yytext);
                  }
            }

"<="       {if(comment != 1){
                  printf(" %i %s", Relop, yytext);
                  }
            }

">="       {if(comment != 1){
                  printf(" %i %s", Relop, yytext);
                  }
            }

((p|P)(r|R)(o|O)(g|G)(r|R)(a|A)(m|M)){1}   {if(comment != 1){
                  printf(" %i program", Program);
                  }
            }

((b|B)(e|E)(g|G)(i|I)(n|N)){1}      {if(comment != 1){
                  printf(" %i begin", Begin);
                  }
            }

((e|E)(n|N)(d|D)){1}    {if(comment != 1){
                  printf(" %i end", End);
                  }
            }

((w|W)(h|H)(i|I)(l|L)(e|E)){1}      {if(comment != 1){
                  printf(" %i while", While);
                  }
            }

((d|D)(o|O)){1}         {if(comment != 1){
                  printf(" %i do", Do);
                  }
            }

({ID}{1})       {if(comment != 1){
                  printf(" %i %s", Id, yytext);
                  }
```

```
              }

{NUM}{1}         {if(comment != 1){
                   printf(" %i %s", Num, yytext);
                   }
             }

{RELOP}{1}       {if(comment != 1){
                   printf(" %i %s", Relop, yytext);
                   }
             }

{ADDOP}{1}       {if(comment != 1){
                   printf(" %i %s", Addop, yytext);
                   }
             }

{MULOP}{1}       {if(comment != 1){
                   printf(" %i %s", Mulop, yytext);
                   }
             }

(" "|\n|\t|\r|\f)               { }

.               {if(comment != 1){
            printf("\n>> Lexical Error: Invalid token - \"%s\".\n",
yytext);
                   yyterminate();
                   }
              }
%%

/* ----------------- User code space ------------------ */
main()
{
  yylex();

  if(comment == 1){
      printf("\n>> Lexical Error: Unterminated quote.\n");
      yyterminate();
  }
}
```

## Classe Symbol

Elle représente les différents symboles de la grammaire :

```
public class Symbol{
  public int index; //Valeur numérique identifiant le symbole.
  public String value; //Ecriture normale du symbole.

  public Symbol(int indx){
```

```
    index = indx;
  }
  public Symbol (int indx, String val){
    index = indx;
    value = val;
  }
}
```

## Classe NonTerminal

```
public class NonTerminal extends Symbol{

      public NonTerminal(int indx){
            super(indx);
      }
      public NonTerminal (int indx, String val){
            super(indx, val);
      }
}
```

## Classe Terminal

```
public class Terminal extends Symbol{

      public Terminal(int indx){
            super(indx);
      }
      public Terminal (int indx, String val){
            super(indx, val);
      }
}
```

## Classe ParsingTable

Regroupe les activités (remplissage) sur la table d'analyse et les déclarations de symboles :

```
public class ParsingTable {

      protected static Symbol [][][] parsingTable; //Table d'analyse
predictive pour l'analyse syntaxique.

      //Variables finales
**********************************************************************

      //Non-terminaux
```

```
	public final static NonTerminal PROGRAM = new NonTerminal(0,
"Program");

	public final static NonTerminal COMPOUND_STATEMENT = new
NonTerminal(1, "Compound_Statement");

	public final static NonTerminal OPTIONAL_STATEMENTS = new
NonTerminal(2, "Optional_Statement");

	public final static NonTerminal STATEMENT_LIST = new
NonTerminal(3, "Statement_List");

	public final static NonTerminal STATEMENT = new NonTerminal(4,
"Statement");

	public final static NonTerminal RECPART_STATEMENT = new
NonTerminal(5, "Recpart_Statement");

	public final static NonTerminal EXPRESSION = new NonTerminal(6,
"Expression");

	public final static NonTerminal ALT_EXPRESSION = new
NonTerminal(7, "Alt_Expression");

	public final static NonTerminal SIMPLE_EXPRESSION = new
NonTerminal(8, "Simple_Expression");

	public final static NonTerminal  RECPART_SIMPLEEXPRESSION = new
NonTerminal(9, "Recpart_SimpleExpression");

	public final static NonTerminal TERM = new NonTerminal(10,
"Term");

	public final static NonTerminal RECPART_TERM = new
NonTerminal(11, "Recpart_Term");

	public final static NonTerminal FACTOR = new NonTerminal(12,
"Factor");


	//Terminaux

	public final static Terminal Prog = new Terminal(0, "program");

	public final static Terminal Begin = new Terminal(1, "begin");

	public final static Terminal End = new Terminal(2, "end");

	public final static Terminal While = new Terminal(3, "while");

	public final static Terminal Do = new Terminal(4, "do");

	public final static Terminal PointVirgule = new Terminal(5, ";");

	public final static Terminal Point = new Terminal(6, ".");
```

```
	public final static Terminal ParentheseOuvrante = new Terminal(7,
"(");

	public final static Terminal ParentheseFermante = new Terminal(8,
")");

	public final static Terminal Assignop = new Terminal(9, ":=");

	public final static Terminal Id = new Terminal(10, "identifier");

	public final static Terminal Num = new Terminal(11, "number");

	public final static Terminal Relop = new Terminal(12, "relop");
// "=, <>, <, <=, >= or >"

	public final static Terminal Addop = new Terminal(13, "addop");
// "\"+\" or \"-\""

	public final static Terminal Mulop = new Terminal(14, "mulop");
// "\"*\" or \"/\""

	//Le Symbole - Transition spontanée
	public final static Symbol TRANSITION = new Symbol(-1, "epsilon");

	//Construction de la table d'analyse
*********************************************************

	/*Suivant les calculs de First et Follow des non-terminaux, nous
entrons dans la table d'analyse,
	 * pour chaque couple "Non-Terminaul - Terminal", la partie
droite correspondante du non-terminal. */

	public static void initialize(){

		parsingTable = new Symbol[13][15][5];

		parsingTable[PROGRAM.index][Prog.index][0] = Prog;
		parsingTable[PROGRAM.index][Prog.index][1] = Id;
		parsingTable[PROGRAM.index][Prog.index][2] = PointVirgule;
		parsingTable[PROGRAM.index][Prog.index][3] =
COMPOUND_STATEMENT;
		parsingTable[PROGRAM.index][Prog.index][4] = Point;

		parsingTable[COMPOUND_STATEMENT.index][Begin.index][0] =
Begin;
		parsingTable[COMPOUND_STATEMENT.index][Begin.index][1] =
OPTIONAL_STATEMENTS;
		parsingTable[COMPOUND_STATEMENT.index][Begin.index][2] =
End;

		parsingTable[OPTIONAL_STATEMENTS.index][Begin.index][0] =
STATEMENT_LIST;
		parsingTable[OPTIONAL_STATEMENTS.index][End.index][0] =
TRANSITION;
		parsingTable[OPTIONAL_STATEMENTS.index][While.index][0] =
STATEMENT_LIST;
```

```
            parsingTable[OPTIONAL_STATEMENTS.index][Id.index][0] =
STATEMENT_LIST;

            parsingTable[STATEMENT_LIST.index][Begin.index][0] =
STATEMENT;
            parsingTable[STATEMENT_LIST.index][Begin.index][1] =
RECPART_STATEMENT;
            parsingTable[STATEMENT_LIST.index][While.index][0] =
STATEMENT;
            parsingTable[STATEMENT_LIST.index][While.index][1] =
RECPART_STATEMENT;
            parsingTable[STATEMENT_LIST.index][Id.index][0] = STATEMENT;
            parsingTable[STATEMENT_LIST.index][Id.index][1] =
RECPART_STATEMENT;

            parsingTable[STATEMENT.index][Begin.index][0] =
COMPOUND_STATEMENT;
            parsingTable[STATEMENT.index][While.index][0] = While;
            parsingTable[STATEMENT.index][While.index][1] = EXPRESSION;
            parsingTable[STATEMENT.index][While.index][2] = Do;
            parsingTable[STATEMENT.index][While.index][3] = STATEMENT;
            parsingTable[STATEMENT.index][Id.index][0] = Id;
            parsingTable[STATEMENT.index][Id.index][1] = Assignop;
            parsingTable[STATEMENT.index][Id.index][2] = EXPRESSION;

            parsingTable[RECPART_STATEMENT.index][PointVirgule.index]
[0] = PointVirgule;
            parsingTable[RECPART_STATEMENT.index][PointVirgule.index]
[1] = STATEMENT;
            parsingTable[RECPART_STATEMENT.index][PointVirgule.index]
[2] = RECPART_STATEMENT;
            parsingTable[RECPART_STATEMENT.index][End.index][0] =
TRANSITION;

            parsingTable[EXPRESSION.index][ParentheseOuvrante.index][0]
= SIMPLE_EXPRESSION;
            parsingTable[EXPRESSION.index][ParentheseOuvrante.index][1]
= ALT_EXPRESSION;
            parsingTable[EXPRESSION.index][Id.index][0] =
SIMPLE_EXPRESSION;
            parsingTable[EXPRESSION.index][Id.index][1] =
ALT_EXPRESSION;
            parsingTable[EXPRESSION.index][Num.index][0] =
SIMPLE_EXPRESSION;
            parsingTable[EXPRESSION.index][Num.index][1] =
ALT_EXPRESSION;

            parsingTable[ALT_EXPRESSION.index][End.index][0] =
TRANSITION;
            parsingTable[ALT_EXPRESSION.index][Do.index][0] =
TRANSITION;
            parsingTable[ALT_EXPRESSION.index][PointVirgule.index][0] =
TRANSITION;
            parsingTable[ALT_EXPRESSION.index]
[ParentheseFermante.index][0] = TRANSITION;
            parsingTable[ALT_EXPRESSION.index][Relop.index][0] = Relop;
```

```
            parsingTable[ALT_EXPRESSION.index][Relop.index][1] =
SIMPLE_EXPRESSION;

            parsingTable[SIMPLE_EXPRESSION.index]
[ParentheseOuvrante.index][0] = TERM;
            parsingTable[SIMPLE_EXPRESSION.index]
[ParentheseOuvrante.index][1] = RECPART_SIMPLEEXPRESSION;
            parsingTable[SIMPLE_EXPRESSION.index][Id.index][0] = TERM;
            parsingTable[SIMPLE_EXPRESSION.index][Id.index][1] =
RECPART_SIMPLEEXPRESSION;
            parsingTable[SIMPLE_EXPRESSION.index][Num.index][0] = TERM;
            parsingTable[SIMPLE_EXPRESSION.index][Num.index][1] =
RECPART_SIMPLEEXPRESSION;

            parsingTable[RECPART_SIMPLEEXPRESSION.index][End.index][0]
= TRANSITION;
            parsingTable[RECPART_SIMPLEEXPRESSION.index][Do.index][0] =
TRANSITION;
            parsingTable[RECPART_SIMPLEEXPRESSION.index]
[PointVirgule.index][0] = TRANSITION;
            parsingTable[RECPART_SIMPLEEXPRESSION.index]
[ParentheseFermante.index][0] = TRANSITION;
            parsingTable[RECPART_SIMPLEEXPRESSION.index][Relop.index]
[0] = TRANSITION;
            parsingTable[RECPART_SIMPLEEXPRESSION.index][Addop.index]
[0] = Addop;
            parsingTable[RECPART_SIMPLEEXPRESSION.index][Addop.index]
[1] = TERM;
            parsingTable[RECPART_SIMPLEEXPRESSION.index][Addop.index]
[2] = RECPART_SIMPLEEXPRESSION;

            parsingTable[TERM.index][ParentheseOuvrante.index][0] =
FACTOR;
            parsingTable[TERM.index][ParentheseOuvrante.index][1] =
RECPART_TERM;
            parsingTable[TERM.index][Id.index][0] = FACTOR;
            parsingTable[TERM.index][Id.index][1] = RECPART_TERM;
            parsingTable[TERM.index][Num.index][0] = FACTOR;
            parsingTable[TERM.index][Num.index][1] = RECPART_TERM;

            parsingTable[RECPART_TERM.index][End.index][0] = TRANSITION;
            parsingTable[RECPART_TERM.index][Do.index][0] = TRANSITION;
            parsingTable[RECPART_TERM.index][PointVirgule.index][0] =
TRANSITION;
            parsingTable[RECPART_TERM.index][ParentheseFermante.index]
[0] = TRANSITION;
            parsingTable[RECPART_TERM.index][Relop.index][0] =
TRANSITION;
            parsingTable[RECPART_TERM.index][Addop.index][0] =
TRANSITION;
            parsingTable[RECPART_TERM.index][Mulop.index][0] = Mulop;
            parsingTable[RECPART_TERM.index][Mulop.index][1] = FACTOR;
            parsingTable[RECPART_TERM.index][Mulop.index][2] =
RECPART_TERM;

            parsingTable[FACTOR.index][ParentheseOuvrante.index][0] =
ParentheseOuvrante;
```

```
            parsingTable[FACTOR.index][ParentheseOuvrante.index][1] =
EXPRESSION;
            parsingTable[FACTOR.index][ParentheseOuvrante.index][2] =
ParentheseFermante;
            parsingTable[FACTOR.index][Id.index][0] = Id;
            parsingTable[FACTOR.index][Num.index][0] = Num;
      }
}
```

## Classe PascalCompiler

Classe effectuant l'analyse syntaxique de programme et l'évaluation statique des expressions:

```
import java.io.*;
import java.lang.Exception;
import java.util.Stack;
import java.util.StringTokenizer;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
public class PascalCompiler {

      //Variables d'instance *********************************

      private BufferedReader input; //BufferedReader du fichier
d'entrée qui contient les tokens analysés par l'outil Lex.

      private String line; //Chaine de caractères obtemu du
BufferedReader.

      private StringTokenizer st; //Utilisé pour accéder chaque lexème
de la chaine "line".

      //Messages ****************************

      public static final String COMPILATION_ABORTED = ">> Compilation
aborted.";
      public static final String COMPILATION_SUCCEEDED = ">> Compiled
successfully.";

      // Constructeur

      public PascalCompiler (String inputFile){
            try{
                  input = new BufferedReader(new
FileReader(inputFile)); //Lecture du fichier.
                  line = input.readLine(); //On obtient le contenu du
fichier sur une ligne.
                  st = new StringTokenizer(line, " ");
            }catch(IOException e) {e.printStackTrace();}
            ParsingTable.initialize(); //Créer la table danalyse.
```

```
	}


	public void start() throws SyntaxError, LexicalError,
SemanticError, CriticalError{

		if(st.countTokens() == 0)
			throw new CriticalError(CriticalError.CRITICAL_ERROR
+ "Possible Empty File.\n" + COMPILATION_ABORTED);

		// On regarde si on avait une erreur lexicale avec Lex.
		try{
			String lex = input.readLine();
			if(lex != null)throw new LexicalError("\n"+lex);
		}catch(IOException e){}

		if(st.countTokens()%2 == 1)
			throw new CriticalError(CriticalError.CRITICAL_ERROR
+ "Check the lexical analysis.\n" + COMPILATION_ABORTED);

		Stack<Symbol> stack = new Stack<Symbol>(); //Pile d'analyse.
		Stack<String> eval = new Stack<String>(); //Pile
d'evaluation semantique.

		boolean expr = false; //Permet de savoir si on se trouve a
une expression.
		boolean evaluable = false; //Indique si l'expression
courante peut etre evalué de facon statique.
		Terminal before = ParsingTable.Begin; //Indique quel
terminal vient avant une expression à evaluer ("while", "(" ou
"assignop").

		Symbol symbol = ParsingTable.PROGRAM;
		stack.push(symbol);

		int type = Integer.parseInt(st.nextToken()); //On prend le
type du lexème suivant.
		String token = st.nextToken(); //On prend le premier lexème
du programme.

		while(!stack.isEmpty()){
			symbol = stack.peek();

			if (symbol instanceof Terminal){ //Le symbole du
dessus de la pile est un terminal.
				Terminal t = (Terminal)symbol;
				if(t.index == type){ //On regarde si le lexème
du programme est le terminal attendu.
					stack.pop(); //On enlève le Terminal
identifié.

					//****************************
Evaluation semantique statique   ************************

					if
((t.equals(ParsingTable.Do)&&before.equals(ParsingTable.While))
```

```
                                        ||
(t.equals(ParsingTable.PointVirgule) &&
before.equals(ParsingTable.Assignop))
                                        ||
(t.equals(ParsingTable.End) && before.equals(ParsingTable.Assignop))
                                        ||
(t.equals(ParsingTable.ParentheseFermante) &&
before.equals(ParsingTable.ParentheseOuvrante))){
                                  String expression = "";
//L'expression.
                                  if
(t.equals(ParsingTable.ParentheseFermante)){
                                        eval.push(")");
                                  }
                                  while(!eval.isEmpty()){
                                        String s = eval.pop();
                                        expression = s + " " +
expression;
                                  }
                                  if (expr){
                                        if (evaluable){
                                              //Il sert a
l'evaluation progressive. l'entier "n" indique sa taille
                                              String aexp =
convert(new StringTokenizer(expression, " ")); //On convertit
l'expression en syntaxe Java.

                                              try{
                                                    Object result =
evaluate(aexp); //On evalue l'expression.


      if(result.toString().equalsIgnoreCase("Infinity")) //division
pour zero.
                                                          throw new
SemanticError(SemanticError.SEMANTIC_ERROR + "Division by zero.\n" +
COMPILATION_ABORTED);


      System.out.println("\n********** The expression: " + expression +
" = "+ result + ".\n");
                                              } catch(ScriptException
se) {
                                                    throw new
SemanticError(SemanticError.SEMANTIC_ERROR + "Check "+ expression + ".\
n" + COMPILATION_ABORTED);
                                              }

                                        }else{

      System.out.println("\n********** The expression: " + expression +
" cannot be evaluated.\n");
                                        }
                                        expr = false;
                                  }
                            }
```

```
                                    if (expr) eval.push(token);

                                    if ((t.equals(ParsingTable.While) ||
t.equals(ParsingTable.Assignop)
                                                ||
t.equals(ParsingTable.ParentheseOuvrante))&&!expr){
                                          before = t;
                                          expr = true;
                                          evaluable = true;
                                    }

                                    if (t.equals(ParsingTable.Id) && expr)
evaluable = false;

                                    //*********************************
Fin de l'evaluation   *******************************


                                    if(!stack.isEmpty()){
                                          //On passe au lexème suivant s'il y
a.
                                          if(st.countTokens() > 0){
                                                type =
Integer.parseInt(st.nextToken()); //Son type.
                                                token = st.nextToken(); //Son
contenu.
                                          }else{
                                                throw new
SyntaxError(SyntaxError.SYNTAX_ERROR + "Possible missing tokens at end
of file.\n" + COMPILATION_ABORTED);
                                          }
                                    }
                              }else{
                                    throw new
SyntaxError(SyntaxError.SYNTAX_ERROR + t.value + " expected
before: \""+ token +"\".\n" + COMPILATION_ABORTED);
                              }

                        }else if(symbol instanceof NonTerminal){ //Le symbole
du dessus de la pile est un non-terminal.
                              NonTerminal nt = (NonTerminal)symbol;

                              Symbol [] relation =
ParsingTable.parsingTable[nt.index][type]; //On identifie la règle de
grammaire à appliquer.

                              if(relation[0] == null){ //Cela signifie que le
terminal du programme analysé n'est pas celui attendu.
                                    throw new
SyntaxError(SyntaxError.SYNTAX_ERROR + "Unexpected token at: \""+ token
+"\".\n" + COMPILATION_ABORTED);
                              }

                              //On remplace dans la pile le non-terminal par
sa partie droite.
```

```
                        stack.pop();

                        Stack<Symbol> tmp = new Stack<Symbol>(); //pour
inverser la liste.
                        String regle =  nt.value + " -> ";
                        int i;
                        for (i = 0; i < relation.length &&  relation[i]
!= null; i++){
                              tmp.push(relation[i]);
                              regle = regle + relation[i].value + " ";
                        }

                        //On affiche la regle utilisée.
                        System.out.println(regle);

                        while(!tmp.isEmpty()){
                              stack.push(tmp.pop());
                        }

                  }else{ //Le symbole est une transition spontanée.
                        stack.pop();
                  }
            }
            if(st.hasMoreTokens()) throw new
SyntaxError(SyntaxError.SYNTAX_ERROR + "Nothing expected after \".\"\n"
+ COMPILATION_ABORTED);
            System.out.println(COMPILATION_SUCCEEDED);
      }

      public String convert (StringTokenizer st){
            String aexp = "";
            while(st.hasMoreTokens()){
                  String s = st.nextToken();
                  if(s.equals("=")){
                        aexp = aexp + "=="; //On convertit.
                  }else if(s.equals("<>")){
                        aexp = aexp + "!="; //On convertit.
                  }else{
                        int i = s.indexOf("E");
                        if(i != -1){ //On a un nombre a exposant.
                              String num = s.substring(0, i);
                              String exposant = s.substring(i+1);
                              aexp = aexp + "Math.pow(" + num + ", " +
exposant + ")"; //On convertit.
                        }else{
                              aexp = aexp + s;
                        }
                  }
            }
            return aexp;
      }

      public Object evaluate(String expression) throws ScriptException{

        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("js");
```

```
            Object result = engine.eval(expression);    //On evalue
l'expression.
            //***********System.out.println(expression+" = "+result);

            return result;
      }



      public static void main (String[] args)throws SyntaxError,
LexicalError, SemanticError, CriticalError{
            if(args.length != 1){
                  System.out.println("Usage: java PascalCompiler
ProgramName"); //Format d'usage de la classe.
            }else{
                  PascalCompiler pc = new PascalCompiler(args[0]);
//Construction du compilateur avec le fichier d'entrée.
                  try{
                        pc.start(); //On commence l'analyse syntaxique.
                  }catch(SyntaxError e)
{System.out.println(e.toString());
                  }catch(LexicalError e)
{System.out.println(e.toString());}
            }
      }

//    Classes imbriquées
      public class CriticalError extends Exception {
            //Quelques constantes
            public static final String CRITICAL_ERROR = "\n>> Critical
Error: ";

            public CriticalError(){super();}
            public CriticalError(String msg){super(msg);}
      }

      public class SyntaxError extends Exception {
            //Quelques constantes
            public static final String SYNTAX_ERROR = "\n>> Syntax
Error: ";

            public SyntaxError(){super();}
            public SyntaxError(String msg){super(msg);}
      }


      public class SemanticError extends Exception {
            //Quelques constantes
            public static final String SEMANTIC_ERROR = "\n>> Semantic
Error: ";

            public SemanticError(){super();}
            public SemanticError(String msg){super(msg);}
      }

      public class LexicalError extends Exception {
            public LexicalError(){super();}
```

```
            public LexicalError(String msg){super(msg);}
        }

}
```

## DOCUMENTS UTILISES

*Input.* Grammar $G$ with no cycles or $\epsilon$-productions.

*Output.* An equivalent grammar with no left recursion.

*Method.* Apply the algorithm in Fig. 4.7 to $G$. Note that the resulting non-left-recursive grammar may have $\epsilon$-productions. □

1. Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2. **for** $i := 1$ **to** $n$ **do begin**
   **for** $j := 1$ **to** $i-1$ **do begin**
   replace each production of the form $A_i \rightarrow A_j\gamma$
   by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$,
   where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all the current $A_j$-productions;
   **end**
   eliminate the immediate left recursion among the $A_i$-productions
   **end**

Figure 1 - Eliminer la récursivité à gauche

*Input.* A string $w$ and a parsing table $M$ for grammar $G$.

*Output.* If $w$ is in $L(G)$, a leftmost derivation of $w$; otherwise, an error indication.

*Method.* Initially, the parser is in a configuration in which it has \$$S$ on the stack with $S$, the start symbol of $G$ on top, and $w$\$ in the input buffer. The program that utilizes the predictive parsing table $M$ to produce a parse for the input is shown in Fig. 4.14. □

```
set ip to point to the first symbol of w$;
repeat
    let X be the top stack symbol and a the symbol pointed to by ip;
    if X is a terminal or $ then
        if X = a then
            pop X from the stack and advance ip
        else error()
    else    /* X is a nonterminal */
        if M[X, a] = X → Y1Y2 · · · Yk then begin
            pop X from the stack;
            push Yk, Yk−1, . . . , Y1 onto the stack, with Y1 on top;
            output the production X → Y1Y2 · · · Yk
        end
        else error()
until X = $    /* stack is empty */
```

**Figure 2 - Algorithme de Aho**

**Example 3.6** **An Attribute Grammar for Simple Assignment Statements**

```
<assign> → <var> := <expr>
<expr> → <var> + <var>
         | <var>
<var> → A | B | C
```

1. Syntax rule: <assign> → <var> := <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type = <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type = <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look-up(<var>.string)

The look-up function looks up a given variable name in the symb table and returns the variable's type.

**Figure 3 – Exemple de définition d'attributs sémantiques**

# REFERENCES

Notes de cours SEG2506 du professeur Gregor v. Bohmann incluant :
Sujets abordés :
Grammaire hors-contexte, analyse descendante, grammaire LL(1), First et Follow, table d'analyse, algorithme pour interpréteur d'une table d'analyse, diagramme syntaxique, traitement des erreurs syntaxiques, attributs sémantiques, règles d'évaluation.

http://www.site.uottawa.ca/~bochmann/SEG2506/devoirs/devoir-3/devoir-3.html
http://www.site.uottawa.ca/~bochmann/SEG2506/Notes/Module-2/IntroGrammaires/index.html
http://www.site.uottawa.ca/~bochmann/SEG2506/Notes/Module-2/IntroGrammaires/First-Follow.html

Ressources Lab 6 SEG2506
http://www.site.uottawa.ca/~bochmann/SEG2506/labs/lab6/index.html
http://www.site.uottawa.ca/~bochmann/SEG2506/labs/lab6/Syner.java

Traduction
http://babelfish.altavista.com/tr

Ressources d'implémentation
http://java.sun.com/j2se/1.4.2/docs/api/java/util/StringTokenizer.html
http://java.sun.com/j2se/1.4.2/docs/api/java/io/BufferedReader.html
http://java.sun.com/j2se/1.4.2/docs/api/java/util/Stack.html
http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html
http://forum.java.sun.com/thread.jspa?threadID=5123390&messageID=9434122
http://forum.java.sun.com/thread.jspa?threadID=5144807&messageID=9536652