SEG 2506
Devoir 3 - Compilateur pour un sous-ensemble de Pascal

par
Philippe Doré

presentee à
M. Gregor v. Bochmann

dans le cadre du cours
SEG 2506

Note :
36/50
= 72%

le 25 mars 2008

SEG 2506 – hiver 2008
Barême de correction – devoir 3

Bonus :
remis à la première échéance 5pts.
code-test supplémentaire 2pts

- 5/5 grammaire LL(1) équivalente réécrite
- 5/5 First & Follow
- 5/5 table d'analyse
- 0/5 attributs sémantiques
- 0/5 règles d'évaluation
- 6/6 programme d'analyse syntaxiques
  - 2/2 fichier input.txt correct fourni
  - 0/2 fichier output.txt correct fourni
- 6/6 programme d'analyse lexical
  - 0/2 fichier input.txt corrct fourni
  - 0/2 fichier output.txt correct fourni (incluant rapport d'erreur s'il y a lieu)
- 0/5 Explications pour chaque élément du travail à remettre

SEG 2506
Devoir 3 - Compilateur pour un sous-ensemble de Pascal

DÉFINITION DU SOUS-ENSEMBLE DE PASCAL :

En prenant les expressions données dans le fichier, et en éliminant les non-terminaux non voulus, j'ai obtenue la grammaire suivante.
Pour la rendre LL(1), on doit éliminer la récursivité a gauche. Pour ce faire, j'ai appliquer l'algorithme qui se trouve dans les notes pour modifier les non-terminaux « statement_list », « expression », « simple_expression » et « term ». Cela a rendu la grammaire précédente, LL(1).

La voici :

S → program **$**

program →
PROGRAM ID
compound_statement **.**

compound_statement →
BEGIN
optional_statements
END

optional_statements →
statement_list | **e**

statement_list →
statement statement_list′

statement_list′ →
; statement statement_list′ | **e**

statement →
ID ASSIGNOP expression
| compound_statement
| **WHILE** expression **DO** statement

expression →
simple_expression expression′

expression′ →
RELOP simple_expression | **e**

simple_expression →
term simple_expression′

simple_expression′ →
ADDOP term simple_expression′ | **e**

term →
factor term′

term′ →
MULOP factor term′ | **e**

factor →
ID | **NUM** | **(** expression **)**

Pour ce sous-ensemble de Pascal, voici les définitions des non-terminaux spéciaux :

ID → letter (letter | digit)*
letter → [a-zA-Z]
digit → [0-9]
NUM → digits
digits → digit digit*
RELOP → = | <> | < | <= | >= | >
ADDOP → + | -
MULOP → * | /
ASSIGNOP → :=

FIRST ET FOLLOWS:

FIRST(program) = **{PROGRAM}**
FIRST(compound_statement) = **{BEGIN}**
FIRST(optional_statements) = FIRST(statement_list) + {e}
= **{BEGIN , ID , WHILE , e}**
FIRST(statement_list) = FIRST(statement)
= FIRST(variable) + FIRST(compound_statement) + {WHILE}
= **{BEGIN , ID , WHILE}**
FIRST(statement_list') = **{; , e}**
FIRST(statement) = FIRST(compound_statement) + {ID , WHILE}
= **{BEGIN , ID , WHILE}**

FIRST(expression) = FIRST(simple_expression)
= FIRST(term)
= FIRST(factor)

= **{ID , NUM , (}**

FIRST(expression') = **{RELOP , e}**

FIRST(simple_expression) = FIRST(term)
= FIRST(factor)
= **{ID , NUM , (}**

FIRST(simple_expression') = **{ADDOP , e}**

FIRST(term) = FIRST(factor)
= **{ID , NUM , (}**

FIRST(term') = **{MULOP , e}**

FIRST(factor) = **{ID , NUM , (}**

FOLLOW(program) = **{$}**

FOLLOW(compound_statement) = {. , FOLLOW(statement)}
= **{. , ; , END}**

FOLLOW(optional_statements) = **{END}**

FOLLOW(statement_list) = FOLLOW(optional_statement)
= **{END}**

FOLLOW(statement_list') = FOLLOW(statement_list)
= **{END}**

FOLLOW(statement) = FIRST(statement_list')
= {; , FOLLOW(statement_list')}
= **{; , END}**

FOLLOW(expression) = **{DO ,)}**

FOLLOW(expression') = FOLLOW(expression)
= **{DO}**

FOLLOW(simple_expression) = FOLLOW(expression')
= **{DO}**

FOLLOW(simple_expression') = FOLLOW(simple_expression)
= **{DO}**

FOLLOW(term) = FIRST(simple_expression')
= {ADDOP , FOLLOW simple_expression'}
= **{ADDOP , DO}**

FOLLOW(term') = FOLLOW(term)
= **{ADDOP , DO}**

FOLLOW(factor) = FIRST(term')
= {MULOP , FOLLOW(term')}
= **{MULOP , ADDOP , DO}**

Apres avoir déterminé les FIRST et les FOLLOW de chaque non-terminaux, on a qu'à comparer les FIRST et FOLLOW de chacun d'eux respectivement pour conclure si la grammaire en question est sous forme LL(1).
Ici, les FIRST de chaque non-terminaux sont disjoints de leurs FOLLOW, alors, la grammaire ci-haute est effectivement sous forme LL(1).

TABLE D'ANALYSE :

Table 1

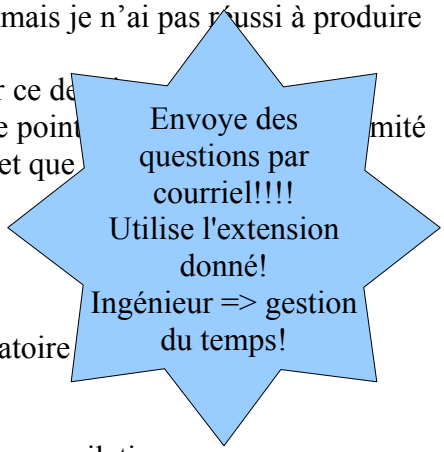
Non-terminaux	**Terminaux**							
	PROGRAM	**BEGIN**	**END**	**;**	**ASSIGN OP**	**WHILE**	**DO**	**ID**
program	#	#	--	--	--	--	--	--
compound_ statement	--	#	--	--	--	--	--	--
optional_ statement	--	*o_s* → s_l	*o_s* → **e**	--	--	*o_s* → s_l	--	*o_s* → s_l
statement_ list	--	#	--	--	--	#	--	#
statement_ list'	--	--	*s_l'* → **e**	*s_l'* → **;** s s_l	--	--	--	--
statement	--	*s* → c_s	--	--	--	--	--	*s* → v **AOP** e
variable	--	--	--	--	--	--	--	#
expression	--	--	--	--	--	--	--	#
expression'	--	--	--	--	--	--	*e'* → **e**	--
simple_ expression	--	--	--	--	--	--	--	#
simple_ expression'	--	--	--	--	--	--	*s_e'* → **e**	--
term	--	--	--	--	--	--	--	#
term'	--	--	--	--	--	--	*t'* → **e**	--
factor	--	--	--	--	--	--	--	*f* → **ID**

** Ici, j'ai utiliser des abréviations des non-terminaux pour économiser de l'espace.
** A cause du manque de place dans la table, le symbole # a été utiliser lorsque l'expression n'avait aucune alternative, donc le # signifie que l'expression respective est la seule qui se trouve dans la grammaire ci-haute.
** Le -- symbolise que le terminal ne figure pas avec le non-terminal respectif.

Table 2

Non-terminaux	Terminaux							
	RELOP	**ADDOP**	**MULOP**	**NUM**	**(**	**)**	**.**	**$**
program	--	--	--	--	--	--	--	--
compound_ statement	--	--	--	--	--	--	--	--
optional_ statement	--	--	--	--	--	--	--	--
statement_ list	--	--	--	--	--	--	--	--
statement_ list'	--	--	--	--	--	--	--	--
statement	--	--	--	--	--	--	--	--
variable	--	--	--	--	--	--	--	--
expression	--	--	--	#	#	--	--	--
expression'	*e'* → **RELOP** s_e	--	--	--	--	--	--	--
simple_ expression	--	--	--	#	#	--	--	--
simple_ expression'	--	*s_e'* → **ADDOP** t s_e'	--	--	--	--	--	--
term	--	--	--	#	#	--	--	--
term'	--	*t'* → **e**	*t'* → **MULOP** f t'	--	--	--	--	--
factor	--	--	--	f → **NUM**	f → **(e)**	--	--	--

Ensuite, la démarche demande de définir des « attributs sémantiques » pour pouvoir évaluer des expressions sans variables et d'ensuite de définir des règles d'évaluation des attributs pour effectuer une évaluation statique des attributs.
J'ai beau examiner le laboratoire 6 et les explications supplémentaires données pour ce devoir, je n'ai pas pu comprendre au juste ce que je devais faire.
Par contre, j'ai bien réussi à développer un compilateur, mais je n'ai pas réussi à produire un fichier de sortie comme voulu.
Je me suis donc baser sur le laboratoire 6 pour compléter ce de
J'espère seulement que vous n'allez pas pénaliser trop de point mité et du fait que c'était la grande fin de semaine de Pâques et que occupées mon temps.



PROGRAMME DE COMPILATION EN JAVA :

En modifiant les fichiers « Lexer » et « Syner » du laboratoire compilateur du sous-ensemble de langage Pascale.

Les fichiers « Lexer » et « Syner » inclus effectuent cette compilation.

TESTS :

Pour pouvoir vérifier si le programme faisait une compilation adéquate, j'ai rédiger quelques programmes tests :

1.
```
PROGRAM test
BEGIN
END
.
```
2.
```
PROGRAM test
BEGIN
v0 := 3
END
.
```
3.
```
PROGRAM test
BEGIN
WHILE 9 < 2 DO v0 := 6
END
.
```
4.
```
PROGRAM test
BEGIN
v0 := 5 * 5
END
.
```
5.
```
PROGRAM test
BEGIN
v0 := 3 ;
v1 := 6 * 8
END
.
```
6.
```
PROGRAM test
BEGIN
v0 := 9;
v1 := 1 * 2;
BEGIN
WHILE 8 > 2 DO x := 0 + 11
END
END
.
```

Puisque tous ces testes passent, nous avons effectivement effectué un compilateur pour le sous-ensemble de Pascale.

Donc, dans ce devoir, nous avons modifié une grammaire existante pour la rendre plus simple. Ensuite, nous l'avons transformé pour que nous puissions effectuer une analyse LL(1). Puis, en évaluant les FIRST et FOLLOW de chaque non-terminal et en dressant une table d'analyse, nous avons confirmé que la grammaire était bien LL(1). Par la suite, nous avons modifié le code du compilateur du VSPL du laboratoire 6 pour qu'il effectue la compilation de notre grammaire.