UNIVERSITÉ LIBRE DE BRUXELLES

INFO-F403 - Introduction To Language Theory And Compiling

Rapport 2ème Partie

Auteurs :

Yasin Arslan

Jacky Trinh

Titulaires :

Gilles Geeraerts

Assistants :

Marie Van den Bogaard

Leo Exibard

Table des matières

1 Introduction **2**

2 Grammaire **2**

3 Contruction de la table d'action LL(1) **4**

3.1 Tableau d'action LL(1) 5

4 Implantation Java **6**

4.1 Classe 6

4.1.1 Rules 6

4.1.2 Parser 6

4.1.3 GenericStack 6

4.1.4 AnalyzingException 6

4.1.5 ParsingException 6

4.2 Structure utilisée 6

5 Exécution & Makefile **7**

6 Exemple d'exécution **7**

7 Conclusion **7**

1 Introduction

Dans le cadre du cours INFO-F403, il nous a été demandé d'implanter un compilateur pour le langage IMP. Ce projet est divisé en 3 parties. La deuxième partie consiste en l'implantation d'un parser qui va vérifier la syntaxe d'un code fourni. Pour ce faire, nous avons dû utiliser ce que nous avions implanté lors de la première partie pour récupérer les symboles ainsi que des notions et concepts vus aux cours théoriques et pratiques.

2 Grammaire

Afin de concevoir un parser, la première approche a été de modifier la grammaire originale du langage *IMP* afin de corriger les incohérences de cette dernière. Dans un premier temps, les variables inatteignables et non-productives devaient être retirées. Vraisemblablement, il n'en n'existait pas. Ensuite, puisque ce langage possède des opérations, les priorités et l'associativité des opérateurs existent. Nous avons donc dû rendre la grammaire non-ambiguë en tenant compte de ces détails. Pour terminer, nous avons retirer toutes les récursions de gauche et nous avons appliquer une factorisation là où c'était nécessaire. La nouvelle grammaire modifiée est donc celle-ci :

Règle	**Variable**	**Terminaison**
1	<Program>	⇒ begin <Code> end
2	<Code>	⇒ ε
3		⇒ <InstList>
4	<InstList>	⇒ <Instruction> <AfterInstruction>
5	<Instruction>	⇒ <Assign>
6		⇒ <If>
7		⇒ <While>
8		⇒ <For>
9		⇒ <Print>
10		⇒ <Read>
11	<AfterInstruction>	⇒ ε
12		⇒ ; <InstList>
13	<Assign>	⇒ [VarName] := <Expr>
14	<Expr>	⇒ <ProdOrDiv> <Expr'>
15	<Expr'>	⇒ <SecondOp> <Expr>
16		⇒ ε

17	<ProdOrDiv>	⇒ <Atom> <ProdOrDiv'>
18	<ProdOrDiv'>	⇒ <FirstOp> <ProdOrDiv>
19		⇒ ε
20	<Atom>	⇒ -<Atom>
21		⇒ [VarName]
22		⇒[Number]
23		⇒ (<Expr>)
24	<FirstOp>	⇒ *
25		⇒ /
26	<SecondOp>	⇒ +
27		⇒ -
28	<If>	⇒ if <Cond> then <Code> <AfterIf>
29	<AfterIf>	⇒ endif
30		⇒ else <Code> endif
31	<Cond>	⇒ <AndCond> <Cond'>
32	<Cond'>	⇒ or <Cond>
33		⇒ ε
34	<AndCond>	⇒ <SimpleCond> <AndCond'>
35	<AndCond'>	⇒ and <AndCond>
36		⇒ ε
37	<SimpleCond>	⇒ <NotCond> <Expr> <Comp> <Expr>
38	<NotCond>	⇒ not
39		⇒ ε
40	<Comp>	⇒ =
41		⇒ <=
42		⇒ >
43		⇒ >=
44		⇒ <
45		⇒ <>
46	<While>	⇒ while <Cond> do <Code> done
47	<For>	⇒ for [VarName] from <Expr> <AfterFor>
48	<AfterFor>	⇒ <ByExpr> to <Expr> do <Code> done
49	<ByExpr>	⇒ by <Expr>
50		⇒ ε
51	<Print>	⇒ print([VarName])
52	<Read>	⇒ read([VarName])

3 Contruction de la table d'action LL(1)

Pour construire cette table aisément, il nous a fallu recourir au tableau *First & Follow*. Pour la construire, nous avons dû trouver pour chaque variable les terminaux initiaux possibles, représentés par le *First*, ainsi que les terminaux possibles qui suivent cette variable représentés par le *Follow*. Une fois fait, il nous a fallu reprendre les *First* pour la construction de la table d'action. Il fallait tout de même faire attention au cas où une variable pouvait commencer par ε, auquel cas, il suffisait d'ajouter le *Follow*.

Variable	**First**	**Follow**
<Program>	begin	
<Code>	ε [VarName] if while for print read	end endif else done
<InstList>	[VarName] if while for print read	end endif else done
<Instruction>	[VarName] if while for print read	end endif else done ;
<AfterInstruction>	ε ;	end endif else done
<Assign>	[VarName]	end endif else done ;
<Expr>	[VarName] [Number] (-	end endif else done ;) do to and or then by = <= > >= < <>
<Expr'>	ε + -	end endif else done ;) do to and or then by = <= > >= < <>
<ProdOrDiv>	[VarName] [Number] (-	end endif else done ;) do to and or then by, = <= > >= < <> + -
<ProdOrDiv'>	ε * /	end endif else done ;) do to and or then by = <= > >= < <> + -
<Atom>	[VarName] [Number] (-	end endif else done ;) do to and or then by = <= > >= < <> + - * /
<FirstOp>	* /	[VarName] [Number] (-
<SecondOp>	+ -	[VarName] [Number] (-

<If>	if	end endif else done ;
<AfterIf>	endif else	end endif else done ;
<Cond>	not [VarName] [Number] (-	then do
<Cond'>	ε or	then do
<AndCond>	not [VarName] [Number] (-	or then do
<AndCond'>	ε and	or then do
<SimpleCond>	not [VarName] [Number] (-	and or then do
<NotCond>	not ε	[VarName] [Number] (-
<Comp>	= <= > >= < <>	[VarName] [Number] (-
<While>	while	end endif else done ;
<For>	for	end endif else done ;
<AfterFor>	by to	end endif else done ;
<ByExpr>	ε by	to
<Print>	print	end endif else done ;
<Read>	read	end endif else done ;

3.1 Tableau d'action LL(1)

	begin	[Varname]	if	while	for	print	read	;	[Number]	(	)	-	+	*	/	end	endif	else	done	not	or	and	=	<=	>	>=	<	<>	by	to	do	then
Program	1																															
Code		3	3	3	3	3	3									2	2	2	2													
InstList		4	4	4	4	4	4																									
Instruction		5	6	7	8	9	10																									
AfterInstruction								12								11	11	11	11													
Assign		13																														
Expr		14							14	14		14																				
Expr'								16			16	15	15			16	16	16	16		16	16	16	16	16	16	16	16	16	16	16	16
ProdOrDiv		17							17	17		17																				
ProdOrDiv'								19			19			18	18	19	19	19	19		19	19	19	19	19	19	19	19	19	19	19	19
Atom		21							22	23		20																				
FirstOp														24	25																	
SecondOp												27	26																			
If			28																													
AfterIf																	29	30														
Cond		31							31	31		31								31												
Cond'																					32									33	33
AndCond		34							34	34		34								34												
AndCond'																					36	35								36	36
SimpleCond		37							37	37		37								37												
NotCond		39							39	39		39								38												
Comp																							40	41	42	43	44	45				
While				46																												
For					47																											
AfterFor																													48	48		
ByExpr																													49	50		
Print						51																										
Read							52																									

FIGURE 1 – Table d'action LL(1)

4 Implantation Java

4.1 Classe

De nouvelles classes ont vu le jour afin de rendre l'existence du Parser.

4.1.1 Rules

Cette classe contient toutes les règles de la nouvelle grammaire et vérifie ainsi si l'input actuel est bien ce qu'on attendait ou pas.

4.1.2 Parser

C'est elle qui va se charger d'analyser syntaxiquement la cohérence des inputs à l'aide de la classe *Rules* en appliquant les règles en conséquence.

4.1.3 GenericStack

Cette classe va nous permettre de traiter notre liste de Symbole comme étant un Stack. Ainsi nous pouvons uniquement pop l'élément au sommet du Stack, l'analyser, et si la grammaire l'accepte, passer au suivant.

4.1.4 AnalyzingException

Au vu des remarques concernant la première partie du projet, nous avons par conséquent créé une classe d'exception vis-à-vis du *LexicalAnalyzer* lorsqu'un *token* non reconnu par le langage est lu. Le programme s'arrêtera en affichant l'erreur.

4.1.5 ParsingException

Même chose que la classe au-dessus, lorsque la syntaxe n'est pas respectée, une erreur survient et le programme s'arrête.

4.2 Structure utilisée

Nous avons implanté et utilisé une classe générique que nous avons appelé GenericStack. Cette dernière nous a permis de simuler un Stack contenant les symboles identifiés dans le code *.imp* fourni. Si le symbole est satisfaisant, on applique la règle approprié, sinon une erreur est renvoyé à l'utilisateur.

5 Exécution & Makefile

Afin de pouvoir exécuter notre parser, il faut se situer dans le dossier */dist/* et rentrer la commande suivante dans un terminal : *java -jar part2.jar sourceFile* où le *sourceFile* est supposé être un fichier *.imp*.
Nous avons également créé un makefile afin de rendre les choses beaucoup plus simple. Pour l'utiliser, il faut se rendre dans le dossier */more/*.

- *make* permet de compiler le projet.
- *make euclid* permet de tester les fichiers *.class* avec le fichier d'input *Euclid.imp* provenant de l'Université Virtuelle.
- *make test* même chose qu'au-dessus mais avec le fichier d'input Test.imp.
- *make aa* même chose qu'au-dessus mais avec le fichier d'input *aa.imp*.
- *make jar* permet de créer un fichier *jar* dans le dossier */dist/* ainsi que de l'exécuter avec le fichier de base *Euclid.imp*
- *make clean* supprime tous les fichiers *.class*.

6 Exemple d'exécution

Afin de vérifier si notre parser fonctionnait correctement, nous l'avons testé sur trois fichiers dont un étant le fichier fourni par l'Université Virtuelle, à savoir *Euclid.imp*. Ayant vérifié un par un les règles que nous affichait notre parser, nous pouvons affirmer qu'il fonctionne correctement.

```
#-----------------
yasinrjs@yasinrjs-ThinkPad-T440p:~/Bureau/Compil/Projet/Part2/more$ make euclid
#-----------------
#LAUNCH EUCLID.imp
#-----------------
java Main ../test/Euclid.imp
1 3 4 10 52 12 4 10 52 12 4 7 46 31 34 37 39 14 17 21 19 16 45 14 17 22 19 16 36 33 3 4 5 13 14 17 21 19 15 27 14 17 23 14 17 22 19 16 19 16 12 4 7 46 31 34 37 39 14 17 21 19 1
6 43 14 17 21 19 16 36 33 3 4 5 13 14 17 21 19 15 27 14 17 21 19 16 11 12 4 5 13 14 17 21 19 16 12 4 5 13 14 17 21 19 16 11 12 4 9 51 11
------- Parsing done with success -----------
#
#-----------------
#
#-----------------
```

FIGURE 2 – Exemple d'exécution

7 Conclusion

Cette deuxième partie nous a permis d'analyser concrètement les éventuelles erreurs syntaxiques d'un code IMP fourni. Nous pouvons être sûrs que tout code ayant passé le test du Parser pourra donc être exécuté. Nous avons opté pour une implantation récursive en nous basant uniquement sur le Stack de Symbole reçu par notre Scanner de la première partie. Nous attendons avec impatience la troisième partie pour tester notre compilateur universel.