Université Libre de Bruxelles

INFO-F-403
Introduction to language theory & compiling

Report

Étudiants :
Abderrahmane Faher, 000514457
M1-INFO
Eden Aroch, 000542373
M1-INFO

Enseignants :
G. GEERAERTS
M. BALACHANDER
M. SASSOLAS

24 novembre 2024

Table des matières

1 New Grammar Table

[1]	<Program>	→ LET [ProgName] BE <Code> END
[2]	<Code>	→ <Instruction> : <Code>
[3]		→ ϵ
[4]	<Instruction>	→ <Assign>
[5]		→ <If>
[6]		→ <While>
[7]		→ <Call>
[8]		→ <Output>
[9]		→ <Input>
[10]	<Assign>	→ [VarName] = <ExprArith>
[11]	<Cond>	→ <CompCond><cond-tail>
[12]	<cond-tail>	→ -> <Cond>
[13]		→ ϵ
[14]	<CompCond>	→ <AtomCond><CompCond'>
[15]	<CompCond'>	→ == <AtomCond><CompCond'>
[16]		→ <= <AtomCond><CompCond'>
[17]		→ < <AtomCond><CompCond'>
[18]		→ ϵ
[19]	<AtomCond>	→ |<Cond>|
[20]		→ <ExprArith>
[21]	<ExprArith>	→ <Prod><ExprArith'>
[22]	<ExprArith'>	→ +<Prod><ExprArith'>
[23]		→ -<Prod><ExprArith'>
[24]		→ ϵ
[25]	<Prod>	→ <Atom><Prod'>
[26]	<Prod'>	→ *<Atom><Prod'>
[27]		→ /<Atom><Prod'>
[28]		→ ϵ
[29]	<Atom>	→ -<Atom>
[30]		→ [VarName]
[31]		→ [Number]
[32]		→ (<ExprArith>)
[33]	<If>	→ IF { <Cond> } THEN <Code> <if-tail>
[34]	<if-tail>	→ END
[35]		→ ELSE <Code> END
[36]	<While>	→ WHILE { <Cond> } REPEAT <Code> END

[37]	<Output>	→ OUT ([VarName])
[38]	<Input>	→ IN ([VarName])

TABLE 1 – The GILLES Grammar (Modified) - Part 2

2 LL(1) check & Action Table

LL(1) check

To check if a grammar is LL(1), we have to check if it's **unambiguous**, contains no **left-recursion**, and contains no **common prefixes**.

The grammar presented in table **??** is LL(1) because it's **unambiguous**, since we applied the necessary transformations to take into account the priority and associavity of the operators. The grammar is also **left-recursion free**, as we turned all left-recursions into right-recursions. And finally, the grammar contains **no common prefixes**, because we factorized every rule that had common prefixes.

Action Table

	LET	BE	END	:	[VarName]	=	->	==	<=	<
<Program>	1									
<Code>			3		2					
<Instruction>					4					
<Assign>					10					
<Cond>					11					
<cond-tail>							12			
<CompCond>					14					
<CompCond'>							18	15	16	17
<AtomCond>					20					
<ExprArith>					21					
<ExprArith'>				24			24	24	24	24
<Prod>					25					
<Prod'>				28			28			
<Atom>					30					
<If>										
<if-tail>			34							
<While>										
<Output>										
<Input>										

	|	+	-	*	/	[Number]	(	)	IF	{
<Program>										
<Code>									2	
<Instruction>									5	
<Assign>										
<Cond>	11		11			11	11			
<cond-tail>	13									
<CompCond>	14		14			14	14			
<CompCond'>	18									
<AtomCond>	19		20			20	20			
<ExprArith>			21			21	21			
<ExprArith'>	24	22	23					24		
<Prod>			25			25	25			
<Prod'>	28	28	28	26	27			28		
<Atom>			29			31	32			
<If>									33	
<if-tail>										
<While>										
<Output>										
<Input>										

	}	THEN	ELSE	WHILE	REPEAT	OUT	IN			
<Program>										
<Code>			3	2		2	2			
<Instruction>				6		8	9			
<Assign>										
<Cond>										
<cond-tail>	13									
<CompCond>										
<CompCond'>	18									
<AtomCond>										
<ExprArith>										
<ExprArith'>	24									
<Prod>										
<Prod'>	28									
<Atom>										
<If>										
<if-tail>			35							
<While>				36						
<Output>						37				
<Input>							38			

3 Work Presentation

In this project, we implemented a parser in Java to analyze the output produced by the lexer (that was made in the first part of the project). The parser verifies whether the lexer's output follows the rules in the grammar table ??, ensuring syntactic correctness.

The folder 'src' contains all the source code that was made in order to build the parser. Here is a brief explanation of the content of each file we implemented :

— **Main.java** : This file initializes the lexer and the parser, and runs the parser on a file given as an argument. To use run the program : **java Main FILENAME -wt filename.tex** where FILENAME is the file that is parsed and filename.tex is the file in which the tree is written.

— **Parser.java** : This file contains the core logic of the parser. It utilizes the action table to determine the appropriate rules to apply.

— **ParseTree.java** : This file contains the necessary source code to construct a parse tree.

— **UnexpectedTokenException.java** : This file contains the exception that is used when an error is found in the parser.

4 Choices and Hypotheses

In this project, we found that the variable <Call> wasn't used anywhere, so we considered it a mistake and kept it in the grammar table, even though it doesn't appear on the left-hand side of any rule.

5 Description of Example Files

In the test folder, we created a file called **"ArithTest.gls"** that tests if the operators' precedence is respected. Another file called **"Euclid.gls"** (obtained on the UV from part one) is in the test folder. It is used to ensure that our parser can parse basic tokens and follow the correct production rules. And finally we have a file **"ErrorTest.gls"** that tests if the parser can detect a syntax error.