INFO-F403

Compiler for
PASCALMAISPRESQUE - Part 2
Report

Mohammad SECUNDAR –> 504107
Dave Pikop Pokam –> 506979

INFO-F403 - Introduction to language theory and compiling
MA1-SECUC
Université Libre de Bruxelles

Table des matières

1 Introduction

The objective of this project is to create a compiler for PASCALMAISPRESQUE, a straightforward imperative language. The compiler development encompasses four main stages : lexical analysis (scanning), syntax analysis (parsing), semantic analysis, and synthesis. In the initial project phase, we implemented the lexical analyzer using the JFlex tool in Java. The core idea involves taking a program written in the PASCALMAISPRESQUE language as input and systematically analyzing, identifying, and tokenizing each symbol according to the language's rules.

The subsequent phase of the project focused on creating a recursive-descent LL(1) parser for the PASCALMAISPRESQUE grammar. The goal is to execute the parser with a PASCALMAISPRESQUE program as input and the provided grammar to generate the sequence of rules applied by the parser or a parse tree. To achieve this, it is crucial to ensure that the grammar is LL(1), meaning it must be non-ambiguous.

2 Grammar transformation

An LL(1) grammar is characterized by its non-ambiguity. Ambiguity in a grammar refers to a situation where there is at least one string that can be derived by multiple parse trees. This ambiguity poses challenges in interpreting sentences and can complicate the task of parsers in determining the correct syntactic structure. Therefore, the pursuit of a non-ambiguous grammar is driven by the desire for clarity and predictability in language processing.

Various techniques can be employed to eliminate ambiguity within a grammar. These include removing unproductive rules and unreachable variables, considering the priority and associativity of operators, eliminating left-recursion, and applying factorization where necessary.

In the upcoming steps, we will denote **Variables** as non-terminals, **Constants** as terminals, and **Symbols** as terminals or non-terminals within our grammar.

2.1 Removing unproductive and variables

A non-productive variable (or non-terminal) is one that cannot be substituted by a sequence of terminal symbols using production rules. When the grammar encounters the variable <For>, it becomes "stuck" because there are no rules available to further decompose or replace <For> with a sequence of terminal symbols—those being the actual characters or tokens in the language being defined. This designates **<For>** as a non-productive variable since it does not contribute to generating valid strings within the language specified by the grammar.

Consequently, we eliminated the rule containing this variable, a process referred to as removing an unproductive rule.

2.2 Removing unreachable variables

A variable is called reachable if it can be reached from the starting variable. The starting variable is reachable by default. After a deep analysis on the reachability of the variables we concluded that all variables are reachable. Here is a draft about our analysis :

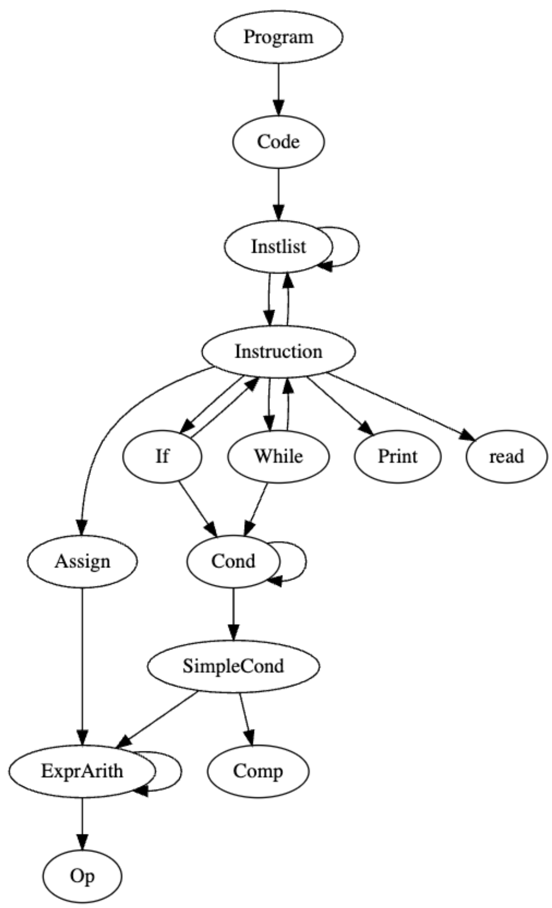

FIGURE 1 – Reachable variables from Program

2.3 Priority and the associativity of the operators

Another way to remove the ambiguity in the grammar is to take into account the natural priority and associativity properties of the arithmetic and Boolean operators.

Before applying the priority of operators

$$
\begin{array}{rl}
< ExprArith > & \rightarrow \text{varname} \\
 & \rightarrow \text{number} \\
 & \rightarrow \text{lparen} < ExprArith > \text{rparen} \\
 & \rightarrow \text{minus} < ExprArith > \\
 & \rightarrow < ExprArith >< Op >< ExprArith > \\
< Op > & \rightarrow \text{plus} \\
 & \rightarrow \text{minus} \\
 & \rightarrow \text{times} \\
 & \rightarrow \text{divide} \\
... & \rightarrow ... \\
< Cond > & \rightarrow < Cond > \ \text{and} \ < Cond > \\
 & \rightarrow < Cond > \ \text{or} \ < Cond > \\
 & \rightarrow \text{lbrack} < Cond > \text{rbrack} \\
 & \rightarrow < SimpleCond >
\end{array}
$$

After applying the priority of operators

$$
\begin{array}{rl}
< ExprArith > & \rightarrow < ExprArith > \ \text{plus} \ < ExprArith2 > \\
 & \rightarrow < ExprArith > \ \text{minus} \ < ExprArith2 > \\
 & \rightarrow < ExprArith2 > \\
< ExprArith2 > & \rightarrow < ExprArith2 > \ \text{times} \ < ExprArith3 > \\
 & \rightarrow < ExprArith2 > \ \text{divide} \ < ExprArith3 > \\
 & \rightarrow < ExprArith3 > \\
< ExprArith3 > & \rightarrow \text{lparen} < ExprArith > \text{rparen} \\
 & \rightarrow \text{minus} < ExprArith > \\
 & \rightarrow \text{varname} \\
 & \rightarrow \text{number} \\
... & \rightarrow ... \\
< Cond > & \rightarrow < Cond > \ \text{or} \ < Cond2 > \\
 & \rightarrow < Cond2 > \\
< Cond2 > & \rightarrow < Cond2 > \ \text{and} \ < Cond3 > \\
 & \rightarrow < Cond3 > \\
< Cond3 > & \rightarrow \text{lbrack} < Cond > \text{rbrack} \\
 & \rightarrow < SimpleCond >
\end{array}
$$

2.4 Left-recursion removal

After applying Left-recursion removal

$$
\begin{aligned}
<ExprArith> &\rightarrow <ExprArith2><ExprArithPrim> \\
<ExprArithPrim> &\rightarrow plus <ExprArith2><ExprArithPrim> \\
&\rightarrow minus <ExprArith2><ExprArithPrim> \\
&\rightarrow \varepsilon \\
<ExprArith2> &\rightarrow <ExprArith3><ExprArith2Prim> \\
<ExprArith2Prim> &\rightarrow times <ExprArith3><ExprArith2Prim> \\
&\rightarrow divide <ExprArith3><ExprArith2Prim> \\
&\rightarrow \varepsilon \\
<ExprArith3> &\rightarrow lparen <ExprArith> rparen \\
&\rightarrow minus <ExprArith> \\
&\rightarrow varname \\
&\rightarrow number \\
... &\rightarrow ... \\
<Cond> &\rightarrow <Cond2><CondPrim> \\
<CondPrim> &\rightarrow or <Cond2><CondPrim> \\
&\rightarrow \varepsilon \\
<Cond2> &\rightarrow <Cond3><Cond2Prim> \\
<Cond2Prim> &\rightarrow and <Cond3><Cond2Prim> \\
&\rightarrow \varepsilon \\
<Cond3> &\rightarrow lbrack <Cond> rbrack \\
&\rightarrow <SimpleCond>
\end{aligned}
$$

2.5 Left-factoring

Before applying Left-factoring

$$
\begin{aligned}
<InstList> &\rightarrow <Instruction> \\
&\rightarrow <Instruction> dots <InstList> \\
... &\rightarrow ... \\
<If> &\rightarrow if <Cond> then <Instruction> else \\
&\rightarrow if <Cond> then <Instruction> else <Instruction>
\end{aligned}
$$

After applying Left-factoring

$$
\begin{aligned}
<InstList> &\rightarrow <Instruction><DotsInstList> \\
<DotsInstList> &\rightarrow \varepsilon \\
&\rightarrow dots <InstList> \\
... &\rightarrow ... \\
<If> &\rightarrow if <Cond> then <Instruction> else <Statement> \\
<Statement> &\rightarrow <Instruction> \\
&\rightarrow \varepsilon
\end{aligned}
$$

3 LL(1) Grammar Verification

After the transformation of the initial grammar, we got a new grammar and we are going to check whether this grammar is LL(1) or not. The method we used is to build the action table of this grammar. For that, we will need the computation of the First and Follow for each variables.

3.1 First and Follow Computation

Variables	First	Follow
Program	begin	$
Code	ε, begin, varname, if, while, print, read	end
Instlist	begin, varname, if, while, print, read	end
DotsInstList	ε, ...	end
Instruction	begin, varname, if, while, print, read	end, ..., else
Assign	varname	end, ..., else
ExprArith	(, -, varname, number	end, ..., (, then, else, and, or, }, =, <, do
ExprArithPrim	+, -, ε	end, ...,), then, else, and, or, }, =, <, do
AddOp	+,-	(, -, varname, number
ExprArith2	(, -, varname, number	end, ...,), -, +, then, else, and, or, }, =, <, do
ExprArith2Prim	*, /, ε	end, ...,), -, +, then, else, and, or, }, =, <, do
MultOp	*, /	(, -, varname, number
ExprArith3	(, -, varname, number	end, ...,), -, +, *, /, then, else, and, or, }, =, <, do
If	if	end, ..., else
Statement	begin, varname, if, while, print, read, ε	end, ..., else
Cond	{, (, -, varname, number	then, }, do
CondPrim	or, ε	then, }, do
Cond2	{, (, -, varname, number	then, or, }, do
Cond2Prim	and,ε	then, or, }, do
Cond3	{, (, -, varname, number	then, and, or, }, do
SimpleCond	(, -, varname, number	then, and, or, }, do
Comp	<, =	(, -, varname, number
While	while	end, ..., else
Print	print	end, ..., else
Read	read	end, ..., else

FIGURE 2 – First and Follow of variables

3.2 Action table

Variables	begin	end	assign	dots	(	)	-	+	*	/	if	then	else	and	or	{	}	=	<	while	do	print	read	varname	number
Program	1																								
Code	3	2									3									3		3	3	3	
Instlist	4										4									4		4	4	4	
DotsInstList		5		6																					
Instruction	12		7								8									9		10	11		
Assign																								13	
ExprArith					14		14																	14	14
ExprArithPrim		16		16	16		15	15				16	16	16	16		16	16	16		16				
AddOp							18	17																	
ExprArith2					19		19																	19	19
ExprArith2Prim		21		21		21	21	21	20	20		21	21	21	21		21	21	21		21				
MultOp									22	23															
ExprArith3					24		25																	26	27
If											28														
Statement	29	30		30							29		30							29		29	29	29	
Cond					31		31									31								31	31
CondPrim												33			32		33				33				
Cond2					34		34									34								34	34
Cond2Prim												36		35	36		36				36				
Cond3					38		38									37								38	38
SimpleCond					39		39									39								39	39
Comp																		40	41						
While																				42					
Print																						43			
Read																							44		

FIGURE 3 – Action table

Observing the table, we find no detected collisions. Consequently, based on the definition, we can conclude that this grammar is LL(1).

4 Implementation

Initially, we modified the grammar to ensure it adheres to LL(1) requirements. Subsequently, we calculated the parser, and finally, we constructed the parse tree.

4.1 Processing the grammar

Initially, we developed the grammarReader lexer to read the grammar and break it down into identifiable sequences of tokens.

Subsequently, the computation of First, Follow, and Action Tables, along with their printing, was implemented in the Grammar.java class. This implementation closely follows the algorithm outlined in Practical Session 5.3. The executable corresponding to this process is implemented in the ProcessGrammar.java class. It's worth noting that Pair.java serves as a simple implementation of the pair type, facilitating the mapping of pairs to specific values (as opposed to using a Map of Maps).

Finally, the main class grammarProcess.java was implemented to manage the grammar and apply the aforementioned implementations.

4.2 Symbol Class Enhancements

In order to enhance flexibility in handling terminals and variables, we made modifications to the Symbol class, dividing it into distinct classes : the **NonTerminal** and **Terminal** classes. These classes function as simple enumerations of non-terminals and terminals, respectively. The Symbol class now serves as both the **Terminal** and **NonTerminal**, employed by the LexicalAnalyser lexer. Additionally, the Token class has taken on the role of the former Symbol class, now utilized by the grammarReader lexer. The union of **NonTerminal** and **Terminal** is encapsulated under the TreeLabel class, which represents labels of parse tree nodes (refer to the ParseTree class). It's important to note that while functional, this may not be the optimal approach in terms of code quality.

4.3 Parse tree construction

The parser keeps track of the current token—here, "current" refers to the token immediately following the reading head. The parser will never need to inspect the token currently pointed to by the reading head. It can examine it (as indicated by the switch(current.getType()) construct) and advance the reading head using the match(term) function. This function checks if the current token is the expected terminal, consumes it, and moves to the next token, mirroring the Match operation discussed in class.

The parse tree is constructed during parsing and is represented using the ParseTree class. This class represents a tree with a root label and a list of its children, each of which is also a parse tree. This aligns with the conventional recursive definition of labeled trees.

Consequently, there exists a function for each non-terminal in the grammar, and each function returns the parse tree rooted in the corresponding non-terminal. For example, as illustrated in Figure 4, when the parser has the starting symbol **Program** as input, it invokes the function **program()**. It's essential to note that the rule for **Program** is **Program -> BEGIN Code END**. Therefore, the function **code()** is called since **code** is a non-terminal. Subsequently, **code()** invokes **instlist()**, which calls **instruction()** and **DotsInstList()**. Finally, **instruction()** invokes **print()**.

The employed algorithm is the LL(1) descent parser [1]

Here are the results of parsing the **printText.pmp** program : **1 3 4 10 44 5**

1. https ://irvifa.medium.com/recursive-descent-parser-573641b461ed : :text=LL(1)

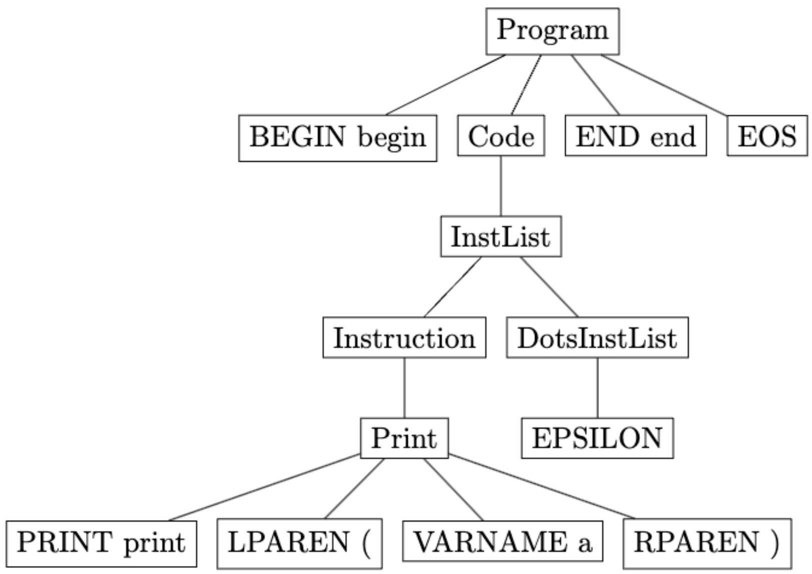

Figure 4 – Parse tree for printText.pmp

5 New grammar built

1	$<Program>$	$\rightarrow$ begin $<Code>$ end
2	$<Code>$	$\rightarrow \varepsilon$
3		$\rightarrow <InstList>$
4	$<InstList>$	$\rightarrow <Instruction><DotsInstList>$
5	$<DotsInstList>$	$\rightarrow \varepsilon$
6		$\rightarrow \ldots <InstList>$
7	$<Instruction>$	$\rightarrow <Assign>$
8		$\rightarrow <If>$
9		$\rightarrow <While>$
10		$\rightarrow <Print>$
11		$\rightarrow <Read>$
12		$\rightarrow$ begin $<InstList>$ end
13	$<Assign>$	$\rightarrow$ varname assign $<ExprArith>$
14	$<ExprArith>$	$\rightarrow <ExprArith2><ExprArithPrim>$
15	$<ExprArithPrim>$	$\rightarrow <AddOp><ExprArith2><ExprArithPrim>$
16		$\rightarrow \varepsilon$
17	$<AddOp>$	$\rightarrow +$
18		$\rightarrow -$
19	$<ExprArith2>$	$\rightarrow <ExprArith3><ExprArith2Prim>$
20	$<ExprArith2Prim>$	$\rightarrow <MultOp><ExprArith3><ExprArith2Prim>$
21		$\rightarrow \varepsilon$
22	$<MultOp>$	$\rightarrow *$
23		$\rightarrow /$
24	$<ExprArith3>$	$\rightarrow (<ExprArith>)$
25		$\rightarrow - <ExprArith3>$
26		$\rightarrow$ varname
27		$\rightarrow$ number
28	$<If>$	$\rightarrow$ if $<Cond>$ then $<Instruction>$ else $<Statement>$
29	$<Statement>$	$\rightarrow <Instruction>$
30		$\rightarrow \varepsilon$
31	$<Cond>$	$\rightarrow <Cond2><CondPrim>$
32	$<CondPrim>$	$\rightarrow$ or $<Cond2><CondPrim>$
33		$\rightarrow \varepsilon$
34	$<Cond2>$	$\rightarrow <Cond3><Cond2Prim>$
35	$<Cond2Prim>$	$\rightarrow$ and $<Cond3><Cond2Prim>$
36		$\rightarrow \varepsilon$
37	$<Cond3>$	$\rightarrow \{<Cond>\}$
38		$\rightarrow <SimpleCond>$
39	$<SimpleCond>$	$\rightarrow <ExprArith><Comp><ExprArith>$
40	$<Comp>$	$\rightarrow =$
41		$\rightarrow <$
42	$<While>$	$\rightarrow$ while $<Cond>$ do $<Instruction>$
43	$<Print>$	$\rightarrow$ print(varname)
44	$<Read>$	$\rightarrow$ read(varname)