

INFO-F403

INTRODUCTION TO LANGUAGE THEORY AND COMPILING

PROJECT – PART 2

Professor :

Gilles GEERAERTS

Authors :

Adel Nehili

Assistants :

Florian Noussa Yao

L eonard Brice

Sarah WINTER

2023-2024

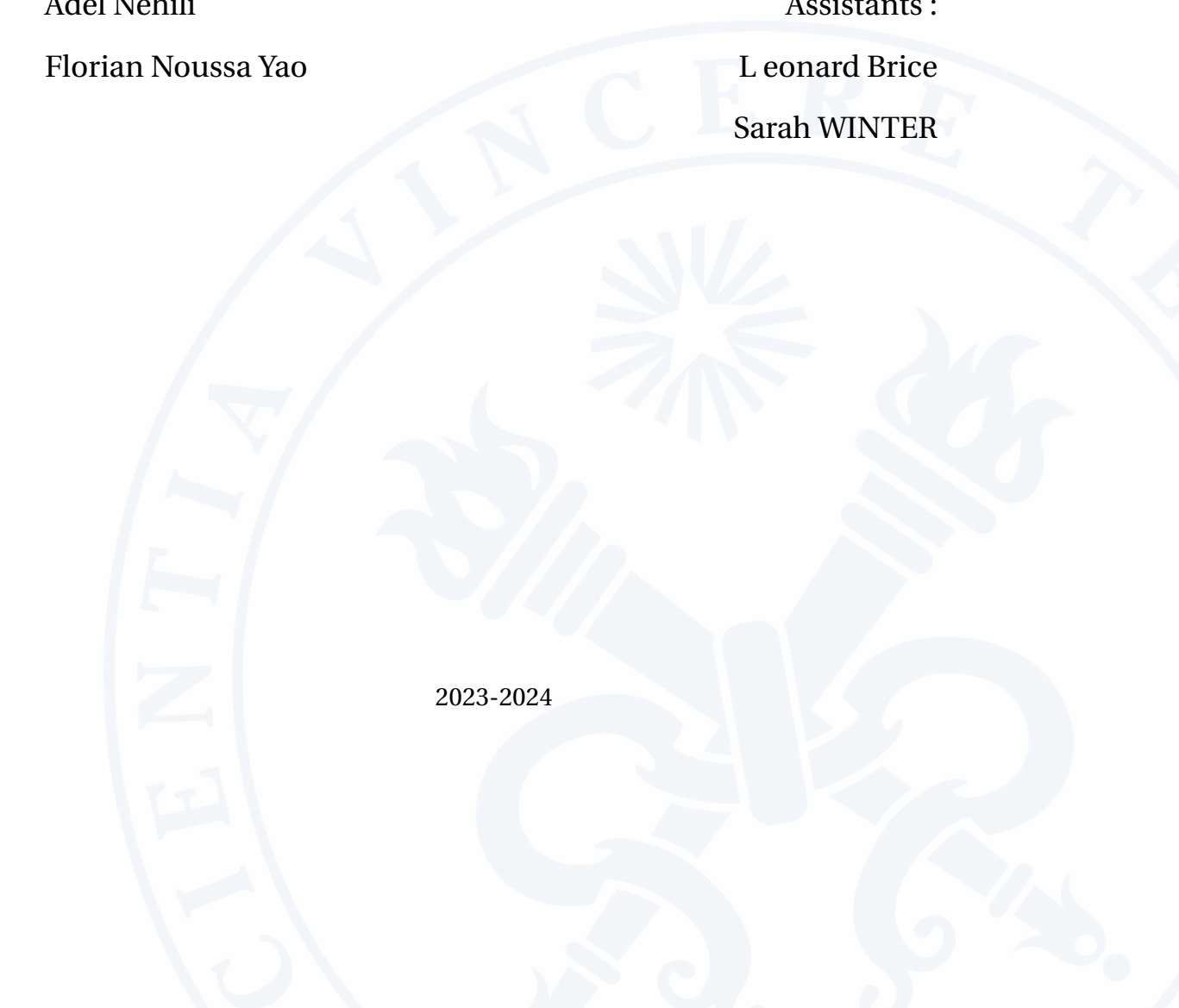


Table des matières

1	Introduction	1
2	Pmp Grammar Transformation	3
2.1	Unreachable/Unproductive Variables Removal	3
2.1.1	Unreachable Variables	3
2.1.2	Unproductive Variables	4
2.2	Non-Ambiguity of Grammar	5
2.3	Left-recursion Removal and Factorisation	6
2.3.1	Understanding Left-Recursion and Factorisation :	6
2.3.2	Eliminating Left Recursion from Grammar	6
3	LL(1) Grammar and Action Table	8
3.1	LL(1) Grammar	8
3.1.1	Understanding LL(1) Grammar	8
3.1.2	Computing the Action Table for LL(1) Grammar	8
3.1.3	Final Grammar	9
3.1.4	First(k) and Follow(k)	11
3.1.5	Action table	12
4	Java Recursive descent LL(1) Parser	15
5	Testing	17
6	Conclusion	19

After lexical analysis, students are required to create a parser for the PASCALmaispresque programming language that checks the syntactic correctness of programs read by the lexer. This will be done through a given grammar rules.

This report describes the various transformations performed on the original grammar of the PASCALmaispresque(Pmp) language, the action table for checking the LL(1) property of the resulting grammar, and the implementation of the parser. It details the Java code written to achieve this purpose. Below is an overview of each section :

Pmp Grammar Transformation

- This section details the process of refining the grammar for effective parsing. It includes :
 - **Unproductive/Unreachable Variables Removal** : Identifying and eliminating variables that do not contribute to productive grammar.
 - **Non-Ambiguity of Grammar** : Ensuring the grammar is unambiguous by considering the priority and associativity of operators.
 - **Left-Recursion Removal and Factorisation** : Addressing the challenges posed by left recursion in the grammar and implementing factorisation to simplify the grammar structure.

LL(1) Grammar and Action Table

- Central to the report, this section focuses on :
 - **LL(1) Grammar** : Discussing the characteristics of LL(1) grammar and its importance in deterministic parsing.
 - **First(k) and Follow(k)** : Computing these sets to aid in constructing the action table, crucial for directing the parsing process.
 - **Action Table** : Developing a table that guides the parser's decisions based on lookahead characters.

Java Recursive Descent LL(1) Parser

- This segment outlines the design and implementation of the parser. It includes :
 - The use of a three-class structure to manage grammar processing, set computation, and parsing methods.
 - Emphasis on the flexibility of the parser to adapt to various context-free grammars.

Pmp Grammar Transformation

The first step in building the parser is the processing of the grammar itself. In order to understand how this is done, it is necessary to define what a grammar is.

A grammar is a quadruplet $G = \langle V, T, P, S \rangle$ where :

- V is a finite set of variables;
- T is a finite set of terminals;
- P is a finite set of production rules of the form $\alpha \rightarrow \beta$ with :
 - $\alpha \in (V \cup T)^* V (V \cup T)^*$
 - $\beta \in (V \cup T)^*$
- $S \in V$ is a variable called the start symbol.

As mentioned earlier, the upcoming parser is designed as a recursive descent LL(1) parser. This implies that the grammar being parsed must adhere to the LL(1) characteristics. In this context, "LL(1)" signifies that the input scanning occurs from left to right, resulting in a leftmost derivation. Furthermore, the "1" indicates that only a single input symbol of lookahead is utilized in the parsing process.

2.1 Unreachable/Unproductive Variables Removal

The **GrammarTransformation class**' initial goal is to search for unproductive and/or unreachable variables. In the following section, we'll explain how we handled the issues linked to the given grammar

2.1.1 Unreachable Variables

Given the original Pmp grammar, if no word w in T^* can be produced from the productions of a variable A , then that variable is said to be unproductive.

Based on this definition and on our project, it was always possible to create words out of arbitrary variables given the original syntax except the variable `<for>` in rule9, therefore this variable will be remove.

[6]	<Instruction>	→ <Assign>
[7]		→ <If>
[8]		→ <While>
[9]		→ <For>
[10]		→ <Print>

FIGURE 2.1 – `<For>` Expection

2.1.2 Unproductive Variables

In our Java implementation, the grammar processing begins with a pre-mannually cleaned grammar input. Once done, the "**PmpGrammar**" class can be filled with the correct grammar. After that, the "**GrammarTransformation**" class will finish to clean the grammar.

```
/**
 * Constructs a Parser
 *
 * @param grammar the grammar of the considered language
 * @param symbolList the list of the symbols supported by the language
 * @param terminalsMap a map having the terminals of the grammar as keys and the corresponding lexical units as values
 */
public Parser(Grammar grammar, ArrayList<Symbol> symbolList, Map<String, LexicalUnit> terminalsMap){
    Grammar reducedGrammar = GrammarTransformation.removeUseless(grammar);
    reducedGrammar.removeLeftRecursion();
    reducedGrammar.leftFactor();
    this.grammar = reducedGrammar;
    //System.out.println(this.grammar.toString());
    this.actionTable = new ActionTable(reducedGrammar);
    System.out.println(actionTable);
    this.symbolList = symbolList;
    this.terminalsMap = terminalsMap;
    this.variablesNumbers = this.grammar.findVariablesNumbers();
    this.rulesSequence = new ArrayList<>();
}
```

FIGURE 2.2 – GrammarTransformation call

Based on that, here's a quick overview on the

removeUnproductive(Grammar G) :

- **Objective :** It's primary goal to identify and eliminate unproductive symbols from a grammar.
- **Working Mechanism :** This function operates by maintaining two sets of variables, previousV and currentV. Initially, both sets are empty. The function then repeatedly updates currentV with variables guessed productive as found by the `findProductiveVariables` method. This process continues in a loop until there is no change in the set of productive variables.
- **Output :** Upon stabilizing the set of productive variables, the function constructs a new grammar, Gprime. This grammar includes only the productive variables (Vprime), the origi-

nal terminal symbols of G , and the productive rules (P_{prime}) that were identified during the process. The result is a streamlined grammar free from unproductive elements.

2.2 Non-Ambiguity of Grammar

In the following section, we'll explain how we considered the priority and associativity of operators to eliminate ambiguities.

In the context of compiler design, an ambiguity in grammar occurs when a string of symbols can be parsed in more than one way, leading to multiple possible interpretations.

Our approach to eliminating ambiguities in the grammar, particularly in arithmetic expressions, involved the introduction of new symbols and the restructuring of production rules. This was necessary as ambiguities primarily arose from the variable $\langle \text{ExprArith} \rangle$.

[14]	$\langle \text{ExprArith} \rangle$	$\rightarrow [\text{VarName}]$
[15]		$\rightarrow [\text{Number}]$
[16]		$\rightarrow (\langle \text{ExprArith} \rangle)$
[17]		$\rightarrow - \langle \text{ExprArith} \rangle$
[18]		$\rightarrow \langle \text{ExprArith} \rangle \langle \text{Op} \rangle \langle \text{ExprArith} \rangle$

FIGURE 2.3 – Left-hand side ambiguity

Since the grammar rules concerned by an ambiguity has as left-hand side the variable $\langle \text{ExprArith} \rangle$, we replace its production rules with a rule producing an expression in which two new symbols appear one after the other. These are the symbols $\langle \text{ExprArith}' \rangle$ and $\langle \text{ExprArith}'' \rangle$.

The second symbol, denoted as $\langle \text{ExprArith}'' \rangle$, is associated with binary addition and subtraction operations to ensure their execution as the final steps.

Conversely, the first symbol, $\langle \text{ExprArith}' \rangle$, is associated with a rule generating a new symbol labeled $\langle T \rangle$.

This newly introduced symbol $\langle T \rangle$ is then linked to a rule that produces an expression featuring two additional symbols sequentially : $\langle T' \rangle$ and $\langle T'' \rangle$.

Similarly, the second symbol, $\langle T'' \rangle$, is associated with multiplication and division operations to ensure their execution before binary addition and subtraction but after unary operations. Regarding unary operations, the first symbol, $\langle T' \rangle$, is linked to a rule that generates a singular new symbol, $\langle U \rangle$, to which unary operations, especially the application of a minus sign to a variable,

are associated. This arrangement guarantees that unary operations precede multiplication and division operations. The introduction of these new symbols, coupled with their hierarchy and order in the production rules, effectively accounts for the priority and associativity of the operators : unary minus, multiplication and division, and binary addition and subtraction.

A similar Process is applied to the rules related to the <cond> variable. there is no need to apply on <comp> variable because there isn't a priority between "<" and "=". The resulting grammar is shown below.

2.3 Left-recursion Removal and Factorisation

2.3.1 Understanding Left-Recursion and Factorisation :

In compiler design, left-recursion occurs in grammar when a production rule can be rewritten such that **it refers to itself as its first symbol**, potentially leading to infinite recursion during parsing.

Factorisation, on the other hand, involves restructuring grammar to resolve conflicts where two or more productions for a **nonterminal begin with the same symbol**. Both left-recursion removal and factorisation are crucial for making a grammar suitable for certain types of parsers, like LL(1) parsers.

2.3.2 Eliminating Left Recursion from Grammar

In our implementation, there is a set of methods designed to address and eliminate left recursion within a given grammar.

Here's an overview of our **GrammarTransformation class** implementation :

1. **Initial Identification with findRecursiveVariables** : This method identifies variables in the grammar that are associated with at least one left-recursive rule.
2. **Systematic Removal Using removeLeftRecursion** : The removeLeftRecursion function is used to systematically eliminate left recursion by processing each identified variable iteratively.
3. **Rule Extraction and Variable Introduction** :
 - For every variable identified as left-recursive, its rules are extracted from the grammar.

- Two new variables, U and V, are introduced to aid in eliminating left recursion.
- 4. **Updating Grammar Rules** : The grammar rules are updated to incorporate the changes after introducing new variables, thereby effectively removing left recursion.
- 5. **Iterative Process** : This process is repeated until no more left-recursive variables are identified in the grammar.
- 6. **Alignment with Grammar Transformation Principles** : The methods used align with the fundamental principles of grammar transformation, contributing to a non-left-recursive grammar.
- 7. **Enhancing LL(1) Parser Effectiveness and Correctness** : By removing left recursion, the effectiveness and correctness of LL(1) parsers are ensured, preventing recursive cycles that can impact parsing accuracy.

The only rules that may be factored are **<If>** and **<InstList>** as the left-hand side : this variable produces production rules with similar prefixes. This have been corrected by adding the **<If>**' and **<InstList>**' variable. Our final Grammar can be found in the next section.

LL(1) Grammar and Action Table

3.1 LL(1) Grammar

3.1.1 Understanding LL(1) Grammar

LL(1) grammars are a specific type of context-free grammar that are particularly suitable for top-down parsing with a **single lookahead character**. The "LL" stands for Left-to-right scanning of the input and Leftmost derivation, while the "1" signifies the use of one lookahead character. These grammars allow for **deterministic parsing**, which means the parser can decide which rule to apply based solely on the next input character and its current state. This makes parsing more efficient and straightforward.

3.1.2 Computing the Action Table for LL(1) Grammar

The action table is the main component in LL(1) parsing. It dictates how the parser should react based on the **current input symbol and the top of the stack**. All this work is done by the "ActionTable class".

1. Role of First and Follow Sets :

- **First Set** : This set is composed of the first terminals that can appear in the strings derived from a variable. It's used to determine the initial characters to expect when a particular production rule is applied.
- **Follow Set** : This set includes all the possible terminals that can immediately follow a particular non-terminal in some "sentential" form.

2. Determining the Action Table :

- The action table is constructed by combining the First and Follow sets. For each production rule, we examine the First set of the right-hand side. If it contains a terminal,

```

/**
 * This method builds the first set of the grammar (containing the first sets of variables and terminals)
 *
 * @return first the first set of the grammar (containing the first sets of each symbol)
 */
public Map<String, Set<String>> buildFirst(){
    Map<String, Set<String>> first = new HashMap<>();
    for(String a : this.grammar.getTerminals()){
        Set<String> firsta = new HashSet<>();
        firsta.add(a);
        first.put(a, firsta);
    }
    for(String A : this.grammar.getVariables()){
        Set<String> firstA = new HashSet<>();
        first.put(A, firstA);
    }
    boolean stable;
    do{
        stable = true;
        for(String A : this.grammar.getVariables()){
            Set<String> previousFirstA = first.get(A);
            Set<String> currentFirstA = buildCurrentFirstA(A, previousFirstA, first);
            if(!stable || (stable = previousFirstA.equals(currentFirstA))){
                first.put(A, currentFirstA);
            }
        }
    }while(!stable);
    return first;
}

```

```

/**
 * This method builds the follow set of the grammar (containing the follow sets of the variables of the grammar)
 *
 * @return follow the follow set of the grammar (containing the follow sets of each variable)
 */
public Map<String, Set<String>> buildFollow(){
    Map<String, Set<String>> follow = new HashMap<>();
    for(String A : this.grammar.getVariables()){
        if(A != this.grammar.getStartSymbol()){follow.put(A, new HashSet<>());}
        else {follow.put(A, null);}
    }
    boolean stable;
    do{
        stable = true;
        Map<String, ArrayList<ArrayList<String>>> rules = this.grammar.getRules();
        for(Map.Entry<String, ArrayList<ArrayList<String>>> P : rules.entrySet()){
            String B = P.getKey();
            if(this.grammar.getVariables().contains(B)){
                for(ArrayList<String> ruleOfB = P.getValue()){
                    for(int i = 0; i < ruleOfB.size(); i++){
                        String A = ruleOfB.get(i);
                        if(this.grammar.getVariables().contains(A)){
                            if(i-1 < ruleOfB.size()){beta = ruleOfB.get(i-1);}
                            Set<String> previousFollowA = follow.get(A);
                            if(previousFollowA != null){
                                Set<String> currentFollowA = buildCurrentFollowA(B, beta, follow, previousFollowA);
                                if(!stable || (stable = previousFollowA.equals(currentFollowA))){
                                    follow.put(A, currentFollowA);
                                }
                            }
                        }
                    }
                }
            }
        }
    }while(!stable);
    return follow;
}

```

FIGURE 3.1 – ActionTable class : First/Follow Computation

that terminal and the production rule are added to the action table.

- In cases where the First set includes an epsilon, the Follow set of the left-hand side non-terminal is used to determine the appropriate entries in the action table.

3. Utility of the Action Table :

- The action table simplifies parsing by providing a clear set of instructions on what action the **parser should take for each combination of the current state** (non-terminal) and the lookahead character.
- This table helps in avoiding guessing or backtracking since each decision is made based on the table's entries, leading to more efficient and faster parsing.

3.1.3 Final Grammar

As shown below, our grammar is LL(1) because we can check that for each variable input symbol combination, there can be at most one rule which can be specified to let the parser know which action to do.

Grammar :

- [1] $\langle \text{Program} \rangle \rightarrow \text{begin} \langle \text{Code} \rangle \text{end}$
- [2] $\langle \text{Code} \rangle \rightarrow \epsilon$
- [3] $\langle \text{Code} \rangle \rightarrow \langle \text{InstList} \rangle$
- [4] $\langle \text{InstList} \rangle \rightarrow \langle \text{Instruction} \rangle \langle \text{InstList} \rangle'$
- [5] $\langle \text{InstList} \rangle' \rightarrow \dots \langle \text{InstList} \rangle$
- [6] $\langle \text{InstList} \rangle' \rightarrow \epsilon$

- [7] $\langle \text{Instruction} \rangle \rightarrow \langle \text{Assign} \rangle$
- [8] $\langle \text{Instruction} \rangle \rightarrow \langle \text{If} \rangle$
- [9] $\langle \text{Instruction} \rangle \rightarrow \langle \text{While} \rangle$
- [10] $\langle \text{Instruction} \rangle \rightarrow \langle \text{Print} \rangle$
- [11] $\langle \text{Instruction} \rangle \rightarrow \langle \text{Read} \rangle$
- [12] $\langle \text{Instruction} \rangle \rightarrow \text{begin} \langle \text{InstList} \rangle \text{end}$
- [13] $\langle \text{Assign} \rangle \rightarrow [\text{VarName}] := \langle \text{ExprArith} \rangle$
- [14] $\langle \text{ExprArith} \rangle \rightarrow \langle \text{ExprArith} \rangle' \langle \text{ExprArith} \rangle''$
- [15] $\langle \text{ExprArith} \rangle' \rightarrow \langle T \rangle$
- [16] $\langle \text{ExprArith} \rangle'' \rightarrow + \langle T \rangle \langle \text{ExprArith} \rangle''$
- [17] $\langle \text{ExprArith} \rangle'' \rightarrow - \langle T \rangle \langle \text{ExprArith} \rangle''$
- [18] $\langle \text{ExprArith} \rangle'' \rightarrow \epsilon$
- [19] $\langle T \rangle \rightarrow \langle T \rangle' \langle T \rangle''$
- [20] $\langle T \rangle' \rightarrow \langle U \rangle$
- [21] $\langle T \rangle'' \rightarrow * \langle U \rangle \langle T \rangle''$
- [22] $\langle T \rangle'' \rightarrow / \langle U \rangle \langle T \rangle''$
- [23] $\langle T \rangle'' \rightarrow \epsilon$
- [24] $\langle U \rangle \rightarrow [\text{VarName}]$
- [25] $\langle U \rangle \rightarrow [\text{Number}]$
- [26] $\langle U \rangle \rightarrow (\langle \text{ExprArith} \rangle)$
- [27] $\langle U \rangle \rightarrow - \langle U \rangle$
- [28] $\langle \text{If} \rangle \rightarrow \text{if} \langle \text{Cond} \rangle \text{then} \langle \text{Instruction} \rangle \text{else} \langle \text{If} \rangle'$
- [29] $\langle \text{If} \rangle' \rightarrow \langle \text{Instruction} \rangle$
- [30] $\langle \text{If} \rangle' \rightarrow \epsilon$
- [31] $\langle \text{Cond} \rangle \rightarrow \langle \text{Cond} \rangle' \langle \text{Cond} \rangle''$
- [32] $\langle \text{Cond} \rangle' \rightarrow \langle V \rangle$
- [33] $\langle \text{Cond} \rangle'' \rightarrow \text{or} \langle V \rangle \langle \text{Cond} \rangle''$
- [34] $\langle \text{Cond} \rangle'' \rightarrow \epsilon$
- [35] $\langle V \rangle \rightarrow \langle V \rangle' \langle V \rangle''$

- [36] $\langle V \rangle' \rightarrow \langle W \rangle$
- [37] $\langle V \rangle'' \rightarrow \text{and } \langle W \rangle \langle V \rangle''$
- [38] $\langle V \rangle'' \rightarrow \epsilon$
- [39] $\langle W \rangle \rightarrow \{ \langle \text{Cond} \rangle \}$
- [40] $\langle W \rangle \rightarrow \langle \text{SimpleCond} \rangle$
- [41] $\langle \text{SimpleCond} \rangle \rightarrow \langle \text{ExprArith} \rangle \langle \text{Comp} \rangle \langle \text{ExprArith} \rangle$
- [42] $\langle \text{Comp} \rangle \rightarrow =$
- [43] $\langle \text{Comp} \rangle \rightarrow <$
- [44] $\langle \text{While} \rangle \rightarrow \text{while } \langle \text{Cond} \rangle \text{ do } \langle \text{Instruction} \rangle$
- [45] $\langle \text{Print} \rangle \rightarrow \text{print } ([\text{VarName}])$
- [46] $\langle \text{Read} \rangle \rightarrow \text{read } ([\text{VarName}])$

3.1.4 First(k) and Follow(k)

As explained previously, we have to determine the First(k)/Follow(k) sets to get the corresponding action table. Based on that, here's the sets :

k	$First^1(k)$	$Follow^1(k)$
$\langle \text{Program} \rangle$	begin	
$\langle \text{Code} \rangle$	ϵ begin [VarName] if while print read	end
$\langle \text{InstList} \rangle$	begin [VarName] if while print read	end
$\langle \text{InstList} \rangle'$	$\epsilon \dots$	end
$\langle \text{Instruction} \rangle$	begin [VarName] if while print read	end ... else
$\langle \text{Assign} \rangle$	[VarName]	end ... else
$\langle \text{ExprArith} \rangle$	- [VarName] [Number] (or and else) end do then $\leq =$ } ...
$\langle \text{ExprArith} \rangle'$	- [VarName] [Number] (or) + do then - ... and else end $\leq =$ }
$\langle \text{ExprArith} \rangle''$	$\epsilon + -$	or and else) end do then $\leq =$ } ...
$\langle T \rangle$	- [VarName] [Number] (or) + do then - ... and else end $\leq =$ }
$\langle T \rangle'$	- [VarName] [Number] (or) * + do then - ... / and else end $\leq =$ }
$\langle T \rangle''$	$\epsilon * /$	or) + do then - ... and else end $\leq =$ }

<U>	- [VarName] [Number] (or) * + do then - ... / and else end < = }
<If>	if	end ... else
<If>'	ϵ begin [VarName] if while print read	end ... else
<Cond>	{ - [VarName] [Number] (then } do
<Cond>'	{ - [VarName] [Number] (or then } do
<Cond>''	or ϵ	then } do
<V>	{ - [VarName] [Number] (or then } do
<V>'	{ - [VarName] [Number] (and or then } do
<V>''	and ϵ	or then } do
<W>	{ - [VarName] [Number] (and or then } do
<SimpleCond>	- [VarName] [Number] (and or then } do
<Comp>	= <	- [VarName] [Number] (
<While>	while	end ... else
<Print>	Print	end ... else
<Read>	read	end ... else

3.1.5 Action table

As previously explained, to build the action table based on the First(1) and Follow(1) sets, it is essential to analyze each production rule in the grammar.

For every rule, the First(1) set of the first symbol on the right-hand side must be considered. If this set does not include the empty word, the action table entries must be populated. For each terminal in the First(1) set of the symbol, the corresponding entry in the action table, determined by the variable on the left-hand side of the rule and the considered terminal, needs to be filled with the rule number. It's important to note that for terminals, the First(1) set consists solely of the terminal itself.

On the other hand, if the First(1) set of the first symbol in the rule's right-hand side contains the empty word, the elements from the Follow(1) set of the variable on the left-hand side of the rule are considered. These elements determine the content of action table entries, which will then be assigned the rule number. Following this logic and applying it to the previously identified First(1) and Follow(1) sets, the resulting action table is obtained.

```

public void buildActionTable(){
    // Retrieve the grammar rules as a map, where the key is a non-terminal and the value is a list of production rules
    Map<String, ArrayList<ArrayList<String>>> rules = this.grammar.getRules();

    // Iterate over each entry (non-terminal and its production rules) in the grammar rules map
    for(Map.Entry<String, ArrayList<ArrayList<String>>> P : rules.entrySet()){
        String A = P.getKey(); // 'A' is the current non-terminal

        // Iterate over the production rules of the current non-terminal
        for(int i = 0; i < P.getValue().size(); i++){
            ArrayList<String> ruleOfA = P.getValue().get(i); // 'ruleOfA' is the current production rule for 'A'

            // Iterate over each symbol in the current production rule
            for(int j = 0; j < ruleOfA.size(); j++){
                String alpha = ruleOfA.get(j); // 'alpha' is the current symbol in the production rule
                boolean containsEpsilon = false; // Flag to check if the First set contains an epsilon

                // Iterate over the First set of the current symbol 'alpha'
                for(String a : this.first.get(alpha)){
                    if(Objects.equals(a, "")){ ...
                    }
                    else{
                        // If 'a' is not epsilon, add an entry to the action table for the combination of 'A' and 'a'
                        addActionTableEntry(A, a, ruleOfA);
                    }
                }

                // If epsilon is in the First set of 'alpha'
                if (containsEpsilon){
                    // Check if there are Follow set entries for 'A'
                    if(this.follow.get(A) != null){...
                    }
                }
                else{ ...
            }
        }
    }
}

```

FIGURE 3.2 – Action Table Building based on First(1)/Follow(1) sets

	begin	end	[VarName]	[Number]	(-)	{	--	}	+	*	/	=	<	if	else	then	and	or	do	print	while	read
<Program>	1																							
<Code>	3	2	3													3						3	3	3
<InstList>	4		4													4						4	4	4
<InstList>		6		5																				
<Instruction>	12		7													8						10	9	11
<Assign>			13																					
<ExprArith>			14	14		14																		
<ExprArith>			15	15		15																		
<ExprArith>		18				17	18		18	18	16			18	18		18	18	18	18	18			
<ExprArith>			19	19	19	19																		
<T>			20	20	20	20																		
<T>		23		25	26	27	23	23	23	23	23	21	22	23	23		23	23	23	23	23			
<U>																								
<If>																28								
<If>	29	30	29						30							29	30				29	29		29
<Cond>			31	31	31	31		31																
<Cond>			32	32	32	32		32																
<Cond>										34								34		33	34			
<V>			35	35	35	35		35																
<V>			36	36	36	36		36																
<V>										38														
<W>			40	40	40	40		39										38	37	38	38			
<SimpleCond>			41	41	41	41																		
<Comp>														42	43									
<While>																								
<Print>																						44		
<Read>																					45			45

FIGURE 3.3 – Action table

Java Recursive descent LL(1) Parser

In the following section, we'll break down the whole chosen code architecture.

Several strategies can be considered for constructing a recursive descent LL(1) parser. One approach involves manually coding a parser tailored to a specific grammar. However, this method comes with the drawback of requiring extensive code adjustments to accommodate a new grammar.

The chosen approach here involves a four-class structure :

- [1] **src/Grammar/PmpGrammar.java** : Subclass of Grammar, it defines what's our grammar.
- [2] **src/Grammar/GrammarTransformation.java** : Classes implements the processing methods for any context-free grammar
- [3] **src/Grammar/ActionTable.java** : Class that handle the computation methods for First(1), Follow(1), and the action table of any context-free grammar
- [4] **src/Parser/Parser.java** : third class deals with the parser methods.

In this design, the computation of the First(1) and Follow(1) sets, along with the construction of the action table, occurs independently of the grammar's language. The parser utilizes the First(1) and Follow(1) sets of any context-free grammar, enabling the creation of a parser tailored to the specific grammar under consideration. This approach offers the advantage of allowing changes to the context-free grammar **without necessitating code adaptations. Only the rules of the new grammar need to be coded.**

The parsing process operates as follows :

Starting from the **Parser** and the initial symbol, we'll checks whether the extracted token is part of the Follow(1) set of the symbol and whether any production rules associated with the symbol

Alongside the `euclid.pmp` file furnished in the initial project phase, four additional test files are offered for parser evaluation. The program can undergo testing through two distinct methods. To execute a test with an input file and exclusively showcase the leftmost derivation of the input on the standard output stream, the following command must be executed :

”`java -jar dist/part2.jar test/sourceFile.pmp`” with `sourceFile.pmp` is the name of the file to test. To generate the parse tree of the input in a LaTeX file and display the leftmost derivation on the standard output stream, execute the following command in the terminal :

```
java -jar dist/part2.jar -wt sourceFile.tex test/sourceFile.pmp
```

Here, `sourceFile.pmp` is the name of the file to be tested, and `sourceFile.tex` is the name of the LaTeX file where the parse tree will be written.

Regarding the new test files, a file named `TestGoodpmp.pmp` allows testing a correct program in the PASCALmaispresque language. This program encompasses all the terminals of the grammar, various types of instructions, and possible operations within a concise program adhering to the language’s grammar. To test the program with this file, execute the command :

```
make test_goodpmp
```

Another test file, `TestOperations.pmp`, enables testing the handling of operation priority. This file features a sequence of mathematical operations on variables and constants within a program adhering to the grammar. It facilitates visualizing the priority of operations in the parse tree. To test the program with this file, use the command :

```
make test_operations
```

Additionally, two test files, `TestIncorrectIf.pmp` and `TestBadRead.pmp`, are provided to observe syntax error detection and parser behavior in such scenarios. `TestIncorrectIf.pmp` contains

a program where the word THEN is omitted in an If statement. The second file attempts to read a variable named IF. To test the program with these files, execute the commands respectively :

```
make test_incorrectIf
```

and

```
make test_badread
```

Note that all the provided commands allow testing the program with the option of writing the parse tree of the input file to a LaTeX file.

In conclusion, the latter portion of the project serves as an exploration into the construction and design of descending recursive parsers, offering insights into the connection between a lexer and a parser. Delving into the practical aspects of parsing necessitates a solid understanding of language theory principles such as the elimination of inefficient and/or inaccessible variables, considerations of operator precedence and associativity, identification of left recursion, and an awareness of LL(1) properties within the grammar. This deeper understanding not only enhances our knowledge of parsing techniques but also underscores the importance of theoretical foundations in effectively implementing parsers for various grammatical structures.