2023-2024

INFOF403 - Introduction to Language Theory and Compilation

PascalMaisPresque Parser

Ferreira Brenno

Mutkowski Philippe

Gilles GEERAERTS

2023

Table des matières

1 Parser

1.1 Introduction

This report will talk about the implementation of a parser in java for an imaginary language called PascalMaisPresque. The goal of the second part of this project was changing the original grammar into something usable, writing the corresponding action table and finally writing the parser to checks if the code respects the language. The lexer was already build in the first part of this project, and we choose to keep our own instead of the one provided on the U.V.. A minor flaw was discovered. It's talked about in the parser section.

1.2 Grammar Transformation

First, the grammar had to be changed :

1. To remove unproductive and unreachable variables
2. To avoid ambiguity (and respect priority and associativity of operators)
3. To remove left-recursion

1.2.1 Unproductive and Unreachable Rules

At the original grammar, there was no unproductive and unreachable rules except for the <For> variable at rule 9 which is unproductive. Indeed, in the grammar there is no other rule <For> $\Rightarrow^*_G w$.

1.2.2 Ambiguity, Priority and Associativity

In order to solve ambiguity in the grammar and apply priority and associativity principles, we have to take a look at rule 14-22, <ExprArith>, and rule 25-29, <Cond>.

<ExprArith> : Rules 14-22

First of all, we consider that an arithmetic expression can produce either an <Expr>±<Prod>, or just <Prod>. This allows to set priority of $*, /$ over $+, -$. Then, we can produce the multiplication and division, or resolve the variables to terminals by producing an atom. After this steps, the priority is set, the last step was to delete left recursion. This was made by adding the "prime" variables.

14	<ExprArith>	→	[VarName] :=<ExprArith>
15		→	[Number]
16		→	(<ExprArith>)
17		→	- <ExpArith>
18		→	<ExprArith> <Op> <ExprArith>
19	<Op>	→	+
20		→	-
21		→	*
22		→	/

FIGURE 1.1 – Version original des règles 14-22.

<ExprArith>	→	<Exp> + <Prod>
	→	<Exp> - <Prod>
	→	<Prod>
<Prod>	→	<Prod>*<Atom>
	→	<Prod>/<Atom>
	→	<Atom>
<Atom>	→	[VarName]
	→	[Number]
	→	(<ExprArith>)
	→	- Atom

FIGURE 1.2 – Ambiguity lifted, priority and associativity set.

14	<ExprArith>	→	<Prod> <ExprArith>'
15	<ExprArith'>	→	+<Prod><ExprArith'>
16		→	-<Prod><ExprArith'>
17		→	ε
18	<Prod>	→	<Atom><Prod'>
19	<Prod'>	→	*<Atom><Prod'>
20		→	/<Atom><Prod'>
21		→	ε
22	<Atom>	→	[VarName]
23		→	[Number]
24		→	(<ExprArith>)
25		→	-<Atom>

FIGURE 1.3 – Left-recursion lifted.

One can also notice the definition of the unary $-$. The rule 25 on the last image can turn any terminal or expression between parenthesis into a negative one. It is also put at the lowest level of recursion, very close to a terminal, to ensure that it has the maximum priority, over all other operators.

<Cond> : Rules 25-29

For <Cond> variable, the task was almost the same. We have to ensure that 'and' has priority over 'or'. This was made by producing first an 'or' then 'and'. The 'or' becomes the main operation, thus giving priority to 'and'.

25	<Cond>	→	<Cond> and <Cond>
26		→	<Cond> or <Cond>
27		→	{<Cond>}
28		→	<SimpleCond>
29	<SimpleCond>	→	<ExprArith> <Comp> <ExprArith>

FIGURE 1.4 – Original version of rules 25-29.

29	<Cond>	→	<And><Cond'>
30	<Cond'>	→	or<And><Cond'>
31		→	ε
32	<And>	→	<CondAtom><And'>
33	<And'>	→	and<CondAtom><And'>
34		→	ε
35	<CondAtom>	→	{<Cond>}
36		→	<ExprArith> <Comp> <ExprArith>

FIGURE 1.5 – Ambiguity lifted, priority and associativity set.

1.2.3 Left-Recursion

In addition to rules <ExprArith> and <Cond>, we had to modify the rules <InstList>, <If> because they contain a left-recursion to their definitions. This is done by factoring the production rules, adding a tail to each expression.

4	<InstList>	→	<Instruction><InstTail>
5	<InstTail>	→	...<InstList>
6		→	ε

FIGURE 1.6 – Modification de la règle InstList.

26	<If>	→	if<Cond>then<Instruction>else<ElseTail>
27	<ElseTail>	→	<Instruction>
28		→	ε

FIGURE 1.7 – Modification de la règle If

The whole modified grammar can be found at the end of the report.

1.2.4 Note

The production rules of <Instruction> could directly produces the correct terminals, deleting a few rules (like 39, 40, 41). The reason of this choice is because it made the code cleaner and easier to understand.

1.3 Action Table

1.3.1 Python Script

At first, a draft of an action table has been written by hand and constructed by checking the first and follows on the fly. The calculation of First and Follow sets turned out to be very time-consuming and not reliable if hand-made. With the idea to speed up the process, we wrote a python script to compute the First and Follow sets and then, based in those information, construct the action table. By doing so, we could allocate more time in double checking the grammar and its rules. With the hand-made action table already pretty complete, we made sure to check discrepancies and made our script reliable.

1.3.2 Terminals, Variables and Grammar

The terminals and variables were encoded as Enum elements in order to ease the identification process.

The grammar was hard-coded inside the script, in the form of a dictionary. The keys were the left-handside variables and the values were the list of rules.

The algorithm also includes detailed debugging logs to aid in understanding its execution. The logging provides insights into the decisions made at each step of the computation. We invite you to check the python script and its output files located in the folder more/playground in order to see its code.

```python
class Terminals(Enum):
    VARNAME = "[VarName]",
    NUMBER = "[Number]",
    BEGIN = "begin",
    END = "end",
    DOTS = "...",
    ASSIGN = ":=",
    LPAREN = "(",
    RPAREN = ")",
    MINUS = "-",
    PLUS = "+",
    TIMES = "*",
    DIVIDE = "/",
    IF = "if",
    THEN = "then",
    ELSE = "else",
    AND = "and",
    OR = "or",
    LBRACK = "{",
    RBRACK = "}",
    EQUAL = "=",
    SMALLER = "<",
    WHILE = "while",
    DO = "do",
    PRINT = "print",
    READ = "read",
    EPSILON = "eps",
    EOF = "$"
```

FIGURE 1.8 – Terminal Enum Class.

```python
class Variables(Enum):
    PROGRAM = "<Program>",
    CODE = "<Code>",
    INSTLIST = "<InstList>",
    INSTTAIL = "<InstTail>",
    INSTRUCTION = "<Instruction>",
    ASSIGN = "<Assign>",
    EXPRARITH = "<ExprArith>",
    EXPRARITHPRIME = "<ExprArith'>",
    PROD = "<Prod>",
    PRODPRIME = "<Prod'>",
    ATOM = "<Atom>",
    IF = "<If>",
    ELSETAIL = "<ElseTail>",
    COND = "<Cond>",
    CONDPRIME = "<Cond'>",
    AND = "<And>",
    ANDPRIME = "<And'>",
    CONDATOM = "<CondAtom>",
    COMP = "<Comp>",
    WHILE = "<While>",
    PRINT = "<Print>",
    READ = "<Read>",
```

FIGURE 1.9 – Variable Enum Class.

```python
grammar = {
            Variables.PROGRAM: [
                [Terminals.BEGIN, Variables.CODE, Terminals.END]
                ],
            Variables.CODE : [
                [Variables.INSTLIST],
                [Terminals.EPSILON]
                ],
            Variables.INSTLIST : [
                [Variables.INSTRUCTION, Variables.INSTTAIL]
                ],
            Variables.INSTTAIL : [
                [Terminals.DOTS, Variables.INSTLIST],
                [Terminals.EPSILON]
                ],
             Variables.INSTRUCTION : [
                 [Variables.ASSIGN],
                 [Variables.IF],
                 [Variables.WHILE],
                 [Variables.PRINT],
                 [Variables.READ],
                 [Terminals.BEGIN, Variables.INSTLIST, Terminals.END]
                 ],
             Variables.ASSIGN : [
                 [Terminals.VARNAME, Terminals.ASSIGN, Variables.EXPRARITH]
                 ],
             Variables.EXPRARITH : [
                 [Variables.PROD, Variables.EXPRARITHPRIME]
                 ],
             Variables.EXPRARITHPRIME : [
                 [Terminals.PLUS, Variables.PROD, Variables.EXPRARITHPRIME],
                 [Terminals.MINUS, Variables.PROD, Variables.EXPRARITHPRIME],
                 [Terminals.EPSILON]
                 ],
```

FIGURE 1.10 – Grammar.

1.3.3 First Set Computation Algorithm

The algorithm implemented in `compute_first(variable)` of the script is derived from the pseudo-code presented in the lecture notes. It begins by checking whether the first set for a given variable has already been computed. This step is crucial due to the recursive nature of the function. If the first set for the variable is not yet computed, the algorithm initializes and applies the following rules :

1. **Terminal Handling :**
   — If the variable is a terminal, it is included in the first set. It is noteworthy that a terminal can have a first set containing itself. While this might seem non-sensical, it plays a crucial role in the recursive functionality of the function.
2. **Production Rule Iteration :**
   — The algorithm iterates through each production rule of the variable, performing the following steps :

(a) If the first symbol in the production rule is a terminal, it is added to the first set.

(b) If the first symbol is a variable, the first set is updated by taking the union with the first set of the respective variable.

(c) Whenever ϵ is present in the first set of the variable, the first set of the subsequent symbol needs to be computed.

The computed first set is then returned.

1	Variable	FIRST
2	PROGRAM	{<Terminals.BEGIN: ('begin',)>}
3	CODE	{<Terminals.READ: ('read',)>, <Terminals.EPSILON: ('eps',)>, <Terminals.PRINT: ('print',)>, <Terminals.WHILE: ('while',)>, <Terminals.VARNAME: ('[VarName]',)>, <Terminals.BEGIN: ('begin',)>, <Terminals.IF: ('if',)>}
4	INSTLIST	{<Terminals.READ: ('read',)>, <Terminals.PRINT: ('print',)>, <Terminals.WHILE: ('while',)>, <Terminals.VARNAME: ('[VarName]',)>, <Terminals.BEGIN: ('begin',)>, <Terminals.IF: ('if',)>}
5	INSTTAIL	{<Terminals.EPSILON: ('eps',)>, <Terminals.DOTS: ('...',)>}
6	INSTRUCTION	{<Terminals.READ: ('read',)>, <Terminals.PRINT: ('print',)>, <Terminals.WHILE: ('while',)>, <Terminals.VARNAME: ('[VarName]',)>, <Terminals.BEGIN: ('begin',)>, <Terminals.IF: ('if',)>}
7	ASSIGN	{<Terminals.VARNAME: ('[VarName]',)>}
8	EXPRARITH	{<Terminals.NUMBER: ('[Number]',)>, <Terminals.MINUS: ('-',)>, <Terminals.LPAREN: ('(',)>, <Terminals.VARNAME: ('[VarName]',)>}
9	EXPRARITHPRIME	{<Terminals.PLUS: ('+',)>, <Terminals.EPSILON: ('eps',)>, <Terminals.MINUS: ('-',)>}
10	PROD	{<Terminals.NUMBER: ('[Number]',)>, <Terminals.MINUS: ('-',)>, <Terminals.LPAREN: ('(',)>, <Terminals.VARNAME: ('[VarName]',)>}
11	PRODPRIME	{<Terminals.DIVIDE: ('/',)>, <Terminals.EPSILON: ('eps',)>, <Terminals.TIMES: ('*',)>}
12	ATOM	{<Terminals.NUMBER: ('[Number]',)>, <Terminals.MINUS: ('-',)>, <Terminals.LPAREN: ('(',)>, <Terminals.VARNAME: ('[VarName]',)>}
13	IF	{<Terminals.IF: ('if',)>}
14	ELSETAIL	{<Terminals.READ: ('read',)>, <Terminals.EPSILON: ('eps',)>, <Terminals.PRINT: ('print',)>, <Terminals.WHILE: ('while',)>, <Terminals.VARNAME: ('[VarName]',)>, <Terminals.BEGIN: ('begin',)>, <Terminals.IF: ('if',)>}
15	COND	{<Terminals.LBRACK: ('{',)>, <Terminals.NUMBER: ('[Number]',)>, <Terminals.VARNAME: ('[VarName]',)>, <Terminals.LPAREN: ('(',)>, <Terminals.MINUS: ('-',)>}
16	CONDPRIME	{<Terminals.OR: ('or',)>, <Terminals.EPSILON: ('eps',)>}
17	AND	{<Terminals.LBRACK: ('{',)>, <Terminals.NUMBER: ('[Number]',)>, <Terminals.VARNAME: ('[VarName]',)>, <Terminals.LPAREN: ('(',)>, <Terminals.MINUS: ('-',)>}
18	ANDPRIME	{<Terminals.EPSILON: ('eps',)>, <Terminals.AND: ('and',)>}
19	CONDATOM	{<Terminals.LBRACK: ('{',)>, <Terminals.NUMBER: ('[Number]',)>, <Terminals.VARNAME: ('[VarName]',)>, <Terminals.LPAREN: ('(',)>, <Terminals.MINUS: ('-',)>}
20	COMP	{<Terminals.SMALLER: ('<',)>, <Terminals.EQUAL: ('=',)>}
21	WHILE	{<Terminals.WHILE: ('while',)>}
22	PRINT	{<Terminals.PRINT: ('print',)>}
23	READ	{<Terminals.READ: ('read',)>}

FIGURE 1.11 – First Set Result.

1.3.4 Follow Set

Algorithm

The `compute_follow(variable)` function calculates the follow set of a given variable in the grammar while preventing infinite recursion by tracking computed follow sets. The algorithm follows these steps :

1. **Base Case Check :**
   — Verify if the follow set for the variable is already computed. If so, return the computed follow set.
2. **Initialization :**
   — Initialize an empty set for the follow set of the current variable.
3. **Start Symbol Check :**
   — If the variable is the start symbol (`Variables.PROGRAM`), add `Terminals.EOF` to its follow set.
4. **Iterate Over Grammar Productions :**
   — For each variable in the grammar, iterate through its production rules.
5. **Production Rule Analysis :**

— For each production rule containing the current variable :

(a) **Case 1 : Variable at the End :**

— If the variable is at the end of the production, and it's not the same as the current variable, update its follow set with the computed follow set of the variable where the production rule is found.

(b) **Case 2 : Variable not at the End :**

— If the variable is not at the end of the production :

— If the next symbol is a terminal, add it to the follow set.

— If the next symbol is a variable :

i. If epsilon is in the first set of the next symbol :

— Update the follow set by excluding epsilon from the first set of the next symbol.

— Consider applying the rule with epsilon in the next iteration.

— Update the follow set with the computed follow set of the next symbol.

— If the next symbol is the last in the production, update the follow set with the computed follow set of the variable where the production rule is found.

ii. If epsilon is not in the first set, update the follow set with the first set of the next symbol.

6. **Logging :**

— Log debugging information for each step of the computation.

7. **Return Result :**

— Return the computed follow set for the variable.

This algorithm ensures that the follow sets are computed accurately and efficiently for the given grammar.

1.3.5 Action Table

Algorithm

— **Initialization :**

— Initialize an empty action table, associating each variable and terminal with a `None` value.

1	Variable	FOLLOW
2	PROGRAM	{<Terminals.EOF: '$'>}
3	CODE	{<Terminals.END: ('end',)>}
4	INSTLIST	{<Terminals.END: ('end',)>}
5	INSTTAIL	{<Terminals.END: ('end',)>}
6	INSTRUCTION	{<Terminals.ELSE: ('else',)>, <Terminals.DOTS: ('...',)>, <Terminals.END: ('end',)>}
7	ASSIGN	{<Terminals.ELSE: ('else',)>, <Terminals.DOTS: ('...',)>, <Terminals.END: ('end',)>}
8	EXPRARITH	{<Terminals.OR: ('or',)>, <Terminals.SMALLER: ('<',)>, <Terminals.RPAREN: (')',)>, <Terminals.ELSE: ('else',)>, <Terminals.END: ('end',)>, <Terminals.DO: ('do',)>, <Terminals.EQUAL: ('=',)>, <Terminals.RBRACK: ('}',)>, <Terminals.AND: ('and',)>, <Terminals.DOTS: ('...',)>, <Terminals.THEN: ('then',)>}
9	EXPRARITHPRIME	{<Terminals.OR: ('or',)>, <Terminals.SMALLER: ('<',)>, <Terminals.RPAREN: (')',)>, <Terminals.ELSE: ('else',)>, <Terminals.END: ('end',)>, <Terminals.DO: ('do',)>, <Terminals.EQUAL: ('=',)>, <Terminals.RBRACK: ('}',)>, <Terminals.AND: ('and',)>, <Terminals.DOTS: ('...',)>, <Terminals.THEN: ('then',)>}
10	PROD	{<Terminals.OR: ('or',)>, <Terminals.SMALLER: ('<',)>, <Terminals.PLUS: ('+',)>, <Terminals.RPAREN: (')',)>, <Terminals.ELSE: ('else',)>, <Terminals.MINUS: ('-',)>, <Terminals.DO: ('do',)>, <Terminals.DOTS: ('...',)>, <Terminals.THEN: ('then',)>, <Terminals.END: ('end',)>, <Terminals.EQUAL: ('=',)>, <Terminals.RBRACK: ('}',)>, <Terminals.AND: ('and',)>}
11	PRODPRIME	{<Terminals.OR: ('or',)>, <Terminals.SMALLER: ('<',)>, <Terminals.PLUS: ('+',)>, <Terminals.RPAREN: (')',)>, <Terminals.ELSE: ('else',)>, <Terminals.END: ('end',)>, <Terminals.MINUS: ('-',)>, <Terminals.DO: ('do',)>, <Terminals.EQUAL: ('=',)>, <Terminals.RBRACK: ('}',)>, <Terminals.AND: ('and',)>, <Terminals.DOTS: ('...',)>, <Terminals.THEN: ('then',)>}
12	ATOM	{<Terminals.OR: ('or',)>, <Terminals.SMALLER: ('<',)>, <Terminals.PLUS: ('+',)>, <Terminals.RPAREN: (')',)>, <Terminals.ELSE: ('else',)>, <Terminals.MINUS: ('-',)>, <Terminals.DO: ('do',)>, <Terminals.DOTS: ('...',)>, <Terminals.TIMES: ('*',)>, <Terminals.THEN: ('then',)>, <Terminals.DIVIDE: ('/',)>, <Terminals.END: ('end',)>, <Terminals.EQUAL: ('=',)>, <Terminals.RBRACK: ('}',)>, <Terminals.AND: ('and',)>}
13	IF	{<Terminals.ELSE: ('else',)>, <Terminals.DOTS: ('...',)>, <Terminals.END: ('end',)>}
14	ELSETAIL	{<Terminals.ELSE: ('else',)>, <Terminals.DOTS: ('...',)>, <Terminals.END: ('end',)>}
15	COND	{<Terminals.DO: ('do',)>, <Terminals.RBRACK: ('}',)>, <Terminals.THEN: ('then',)>}
16	CONDPRIME	{<Terminals.DO: ('do',)>, <Terminals.RBRACK: ('}',)>, <Terminals.THEN: ('then',)>}
17	AND	{<Terminals.DO: ('do',)>, <Terminals.RBRACK: ('}',)>, <Terminals.OR: ('or',)>, <Terminals.THEN: ('then',)>}
18	ANDPRIME	{<Terminals.DO: ('do',)>, <Terminals.RBRACK: ('}',)>, <Terminals.OR: ('or',)>, <Terminals.THEN: ('then',)>}
19	CONDATOM	{<Terminals.OR: ('or',)>, <Terminals.DO: ('do',)>, <Terminals.RBRACK: ('}',)>, <Terminals.AND: ('and',)>, <Terminals.THEN: ('then',)>}
20	COMP	{<Terminals.NUMBER: ('[Number]',)>, <Terminals.MINUS: ('-',)>, <Terminals.LPAREN: ('(',)>, <Terminals.VARNAME: ('[VarName]',)>}
21	WHILE	{<Terminals.ELSE: ('else',)>, <Terminals.DOTS: ('...',)>, <Terminals.END: ('end',)>}
22	PRINT	{<Terminals.ELSE: ('else',)>, <Terminals.DOTS: ('...',)>, <Terminals.END: ('end',)>}
23	READ	{<Terminals.ELSE: ('else',)>, <Terminals.DOTS: ('...',)>, <Terminals.END: ('end',)>}

FIGURE 1.12 – Follow Set Result.

- **Action Table Filling :**
  - Iterate over each variable in the grammar.
  - For each production of the variable :
    1. Compute the first set of the first symbol in the production.
    2. For each terminal in the first set :
       - If the terminal is not epsilon, assign the absolute number of the rule to the corresponding cell in the action table.
    3. If epsilon is in the first set :
       - Compute the follow set of the current variable.
       - For each terminal in the follow set, assign the absolute number of the rule to the corresponding cell in the action table.
- **Production Number Update :**
  - Increment the production number for each processed production.
- **Return Result :**
  - Return the computed action table.

This algorithm constructs an action table by determining the rule numbers for each variable and terminal combination. It efficiently handles both terminals in the first set and terminals in the follow set, ensuring a comprehensive and accurate action table for the grammar.

1.4 Parser

1.4.1 Recursive Descent LL(1) Parser

The design of the parser was largely inspired by the lecture note's example of a recursinve descent LL(1) parser in python, shown in 1.14.

	VARNAME	NUMBER	BEGIN	END	DOTS	ASSIGN	LPAREN	RPAREN	MINUS	PLUS	TIMES	DIVIDE	IF	THEN	ELSE	AND	OR	LBRACK	RBRACK	EQUAL	SMALLER	WHILE	DO	PRINT	READ	EPSILON	EOF
PROGRAM			1																								
CODE	2		2	3									2									2		2	2		
INSTLIST	4		4										4									4		4	4		
INSTTAIL				6	5																						
INSTRUCTION	7		12										8									9		10	11		
ASSIGN	13																										
EXPRARITH	14	14					14		14																		
EXPRARITHPRIME				17	17			17	16	15				17	17	17	17		17	17	17		17				
PROD	18	18					18		18																		
PRODPRIME				21	21			21	21	21	19	20		21	21	21	21		21	21	21		21				
ATOM	22	23					24		25																		
IF													26														
ELSETAIL	27		27	28	28								27		28							27		27	27		
COND	29	29					29		29									29									
CONDPRIME														31			30		31				31				
AND	32	32					32		32									32									
ANDPRIME														34		33	34		34				34				
CONDATOM	36	36					36		36									35									
COMP																				37	38						
WHILE																						39					
PRINT																								40			
READ																									41		

FIGURE 1.13 – Action Table Result.

```
def S():
    n = get_next_character() # This is a look-ahead

    # If the look-ahead is in First(A b Follow(S))
    if n == 'a1' or n == 'a2' or ... or n == 'an' :
        read_next_character() # Discard the look-ahead
        r = A()
        if (!r): return False
        n = read_next_character()
        if (n == 'b'): return True
        else: return False

    # If the look-ahead is in First(B c Follow(S))
    if n == 'b1' or n == 'b2' or ... or n == 'bn' :
        read_next_character() # Discard the look-ahead
        r = B()
        if (!r): return False
        n = read_next_character()
        if (n == 'c'): return True
        else: return False

    # Otherwise, the input cannot be valid.
    return False
```

FIGURE 1.14 – Recursive Descent LL(1) Parser in Python as described in the lecture notes.

Parser attributes The most important attributes of the Parser class are the `lexer`, the `currentToken`, `usedRules` array and the `root` ParseTree.

— The `lexer` attribute holds an instance of the `LexicalAnalyzer`, responsible for scanning the input file and segmenting it into tokens. During the second phase of the project, we opted to utilize the lexer developed in part 1. However, this decision led us to identify a minor flaw in its implementation. Specifically, the token `DOTS` ('...') was absent from the original lexer. Upon detecting this flaw, we addressed it by adding the missing token into the lexer's code.

— The `currentToken` holds a symbol given by the lexer. This attribute is equivalent to a look-ahead from the theory of PDAs. The choice of which rule to apply is based in this symbol.

— The `usedRules` arrays contains a sequence of number rules used in order to accept the word. Whenever a rule is applied, its number is append to the end of the array.

— The `root` ParseTree is just the <Program> ParseTree, but it was names root because it's the beginning of the whole tree.

Parser Methods Each production rule of the grammar was coded as a method of the parser class. Whenever one have to apply rule 13, `<Assign>`, for example, one have to call the method `assign()`. Other important methods are `match()`, `nextToken()` and `syntaxError()` ensures the running of the parser.

— The `program()` method serves as the initial step in the parser. It is called whenever a file needs parsing, as it corresponds to the first rule and represents the start symbol. This method creates the `ParseTree` root and attempts to match the first token, which is `Terminals.BEG`. If the token is successfully matched, the node for `Terminals.BEG` is inserted into the root's children list.

Given that the next token is a `Variable.CODE`, a node is created and inserted into the root children list but the subtree of this node is generated by calling the `code(ParseTree parentTree)` function and passing the code tree as an argument. Consequently, `code(ParseTree parentTree)` is responsible for creating its children nodes and inserting them into their respective children lists. This pattern is repeated whenever the current token is a variable.

Upon the completion of the `code()` execution, it returns the complete parse tree. The `program()` method can then attempt to match the next token, which is `Terminal.END`. If successful, the end node is inserted into the root's children list. After processing `Terminal.END`, the parser should generate the desired output, namely a sequence of applied rules for the derivation. This sequence is precisely maintained by the `usedRules` array.

To conclude the parser's execution, the `program()` method returns the root `ParseTree`. This root can be obtained by the Main Class to subsequently write it to a file.

All production rules were coded in this very same way. First, the current token is matched with some LexicalUnit according to the production rule and the node is inserted into the tree. When the token is a variable, the corresponding tree node is created and the method of this variables is responsible to generate the subtree below.

— The `match(Symbol expected)` is the method responsible of checking if the syntax is correct, which means checking if the current token is the token needed to continue the parsing. If it is successful, it reads the token from the input file stream thanks to the method `nextToken()`. If it fails, it calls the `syntaxError(Symbol token, LexicalUnit[] expected)` method.

— The `nextToken()` uses the lexer to obtain the next token of the file. When this function is called, the lexer give the next token and the last token is consumed. It fails if the lexer could not scan the file.

— The `syntaxError(Symbol token, LexicalUnit[] expected)` raises an error to the standard output alerting that the current token is not valid. It shows the line and column of the token and what were the expected tokens that the parser attempt to identify. It gives also the last state of the parser which is the last rule or method executed, but this is more useful for debugging purposes.

Parse Tree By having each method create its children nodes and append them to their respective children lists, then returning their tree alongside the recursive calls of methods, we can construct the parse tree for any input file.

1.4.2 Changes in LexicalUnit.java

With the idea of detecting if the current token is within the grammar, the LexicalUnit.java file was updated to include the variables names too. It allowed to compare the type of the token with the expected token using `currentType.getType() == LexicalUnit.AND`, for example. There was also a need to include the new variables generated by the grammar transformation like `<Prod>, <ExprArithPrime>, ...`

1.4.3 Main Class

The main class has two commands possible :

- `java -jar ./dist/part2.jar <inputFilePath>`
  - The first one is to only pass as argument the input file path. Thus, the main class calls the method `parse(String filepath)` that initialize an object Parser with the file path and calls the method `program()` of it, which is the first rule. When the parser execution is done, the function `parse(String filepath)` returns the parse tree generated by the run. In this case, nothing is done with it
- `java -jar ./dist/part2.jar -wt <LatexFilePath>.tex <inputFilePath>`
  - If the option `-wt <latexFilePath>.tex` is passed in the arguments, the execution is exactly the same, except that the main class write down a file containing the LaTex of the parse tree.
- In both case, the standard output will contain the sequence of the rules used for this derivation if it is successful. If not, a syntax error will raise and stop the execution. The main execution could be also stopped whenever some other major errors, not related to the parser algorithm, occurs, like wrong command or wrong filepath.

1.5 Tests

1.5.1 All Lexical Units test

This test file has all lexical units written with the correct grammar. This can also be used to check that the parser respects the priority of operations.

1.5.2 Euclid

This was the files provided with the project statement. It was the main file used to debug.

1.5.3 ExprArith

This highlights the priority principle by looking at tree level of operators (* is lower than + then, * has more priority over +).
It highlights also the change of priority in one of the expression by the simple addition of a parenthesis Lastly, it highlights the left associativity when two operators of same priority are together (2*3/1, where the * should be lower than the / because it is on the left)

1.5.4 Fibonacci

This is another code that should be runnable.

1.5.5 Comments

Here it highlisghts that all comments are completely ignored because there are not even a token

.1 PascalMaisPresque Grammar

1	<Program>	→	begin <Code> end
2	<Code>	→	<InstList>
3		→	ϵ
4	<InstList>	→	<Instruction><InstTail>
5	<InstTail>	→	...<InstList>
6		→	ϵ
7	<Instruction>	→	<Assign>
8		→	<If>
9		→	<While>
10		→	<Print>
11		→	<Read>
12		→	begin<InstList>end
13	<Assign>	→	[VarName] :=<ExprArith>
14	<ExprArith>	→	<Prod> <ExprArith>’
15	<ExprArith’>	→	+<Prod><ExprArith’>
16		→	-<Prod><ExprArith’>
17		→	ϵ
18	<Prod>	→	<Atom><Prod’>
19	<Prod’>	→	*<Atom><Prod’>
20		→	/<Atom><Prod’>
21		→	ϵ
22	<Atom>	→	[VarName]
23		→	[Number]
24		→	(<ExprArith>)
25		→	-<Atom>
26	<If>	→	if<Cond>then<Instruction>else<ElseTail>
27	<ElseTail>	→	<Instruction>
28		→	ϵ
29	<Cond>	→	<And><Cond’>
30	<Cond’>	→	or<And><Cond’>
31		→	ϵ
32	<And>	→	<CondAtom><And’>
33	<And’>	→	and<CondAtom><And’>
34		→	ϵ
35	<CondAtom>	→	{<Cond>}
36		→	<ExprArith> <Comp> <ExprArith>
37	<Comp>	→	=
38		→	<
39	<While>	→	while<Cond>do<Instruction>
40	<Print>	→	print([varName])
41	<Read>	→	read([varName])