Université Libre de Bruxelles

Introduction to language theory and compiling
INFO-F403

# Project Pascalmaispresque Part 3 - LLVM

**Students :**

| | |
|---|---|
| Achten Alexandre | 494484 |
| Bienfait Alexandre | 513930 |

**Professors :**

Geeraerts Gilles

Sassolas Mathieu

Brice Léonard

December 2023

Academic year 2023-2024

# Table des matières

# 1 Lexical Analyzer

## 1.1 Regular expressions

**VARNAME : [a-z][a-zA-z0-9]***
A VARNAME must be a string of letters and digits starting with a lower case letter. The corresponding regular expression ensure that a varname starts by a lowercase letter. It is followed by a Kleene closure of letters and digits

**NUMBER : [0-9]+**
A number is a string of digits. There must be at least one digit in it.

## 1.2 MACROS

**WHITESPACE : [ \t \r \n]+**
The whitespaces have been defined as the return of line, space, tabulation and the carriage return.

## 1.3 States

The lexical analyzer can be in the following states.

There are 3 different states :

1. **Initial state**
   It is the basic state. It detects every key words of the language and return a symbol with the corresponding Lexical Unit. When encountering a character that isn't in the Lexical Unit, it raises an Illegal Character message.

2. **Long comment state**
   This state starts when there is a ' ' in the initial state. It ignores any other character except another ' ' that sends back to the initial state.

3. **Short comment state**
   This state start with the characters ** and end where there is a newline character (\n).

We could implement nested comments and recursively call the long comment state to "count" the depth of comments. But the difficulty is that in our language we identify long comments with " ' ' ". There is no difference between the beginning and the end of long comments.
To use nested comments, we would have to create a new character to distinguish the beginning and end of comments (e.g. "/*" and "*/").

## 1.4 Main file

The script in the main file takes as argument a file to analyse with the LexicalAnalyzer. It prints each token and it's lexical unit to the standard output. To print the symbol table, a `Hashmap` is used to store the line of the first occurrence of each variable. Then the symbol table is printed to the standard output.

### 1.5 Tests

Tests have been carried out to ensure that all functionalities concerning the scanner are working. Here's a list of the different tests we've done :

- Test if spaces are not taken into account (regardless of the number of spaces)
- Test with an illegal character such as "A" or " ?"
- Test the recognition of variable names such as "whileif" or "alex83"
- Test the short and long comments
- Test a number followed by characters without spaces between them (the program don't work correctly for that case and just return a NUMBER followed by a VARNAME)
- Test if a NUMBER starts with 0 (we could change the definition of the regular expression NUMBER by [1-9][0-9]*, but we don't know if it's the purpose of the lexical analyzer)

To run all the tests, please use the `make tests` command.

## 2 Parser

### 2.1 Pascalmaispresque grammar

The table of the grammar rules of Pascalmaispresque is shown in the Figure 1.

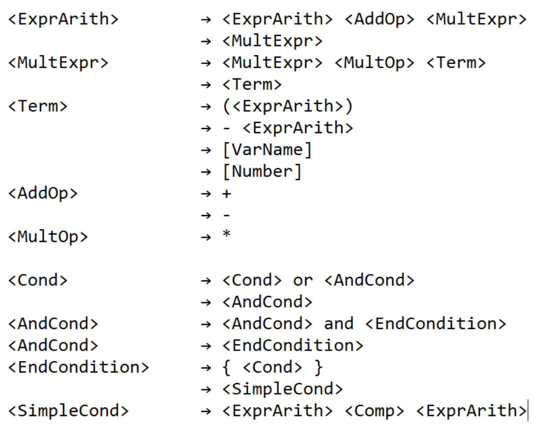
| | | |
|---|---|---|
| [1] | <Program> | → begin <Code> end |
| [2] | <Code> | → $\varepsilon$ |
| [3] | | → <InstList> |
| [4] | <InstList> | → <Instruction> |
| [5] | | → <Instruction> ... <InstList> |
| [6] | <Instruction> | → <Assign> |
| [7] | | → <If> |
| [8] | | → <While> |
| [9] | | → <For> |
| [10] | | → <Print> |
| [11] | | → <Read> |
| [12] | | → begin <InstList> end |
| [13] | <Assign> | → [VarName] := <ExprArith> |
| [14] | <ExprArith> | → [VarName] |
| [15] | | → [Number] |
| [16] | | → ( <ExprArith> ) |
| [17] | | → − <ExprArith> |
| [18] | | → <ExprArith> <Op> <ExprArith> |
| [19] | <Op> | → + |
| [20] | | → − |
| [21] | | → * |
| [22] | | → / |
| [23] | <If> | → if <Cond> then <Instruction> else |
| [24] | | → if <Cond> then <Instruction> else <Instruction> |
| [25] | <Cond> | → <Cond> and <Cond> |
| [26] | | → <Cond> or <Cond> |
| [27] | | → { <Cond> } |
| [28] | | → <SimpleCond> |
| [29] | <SimpleCond> | → <ExprArith> <Comp> <ExprArith> |
| [30] | <Comp> | → = |
| [31] | | → < |
| [32] | <While> | → while <Cond> do <Instruction> |
| [33] | <Print> | → print ([VarName]) |
| [34] | <Read> | → read ([VarName]) |

FIGURE 1 : Initial Pascalmaispresque grammar

In order to get an LL(1) grammar, we must first remove unproductive and unreachable expression.

The only unproductive expression is <For> so we removed it. There a are no unreachable expression .

There is two groupe of operation that have priority, arithmetic expression with {+,-,*,/} and condition with {*and* and *or*}. To handle those priority <ExprArith> and <Cond> were changed to the following form :



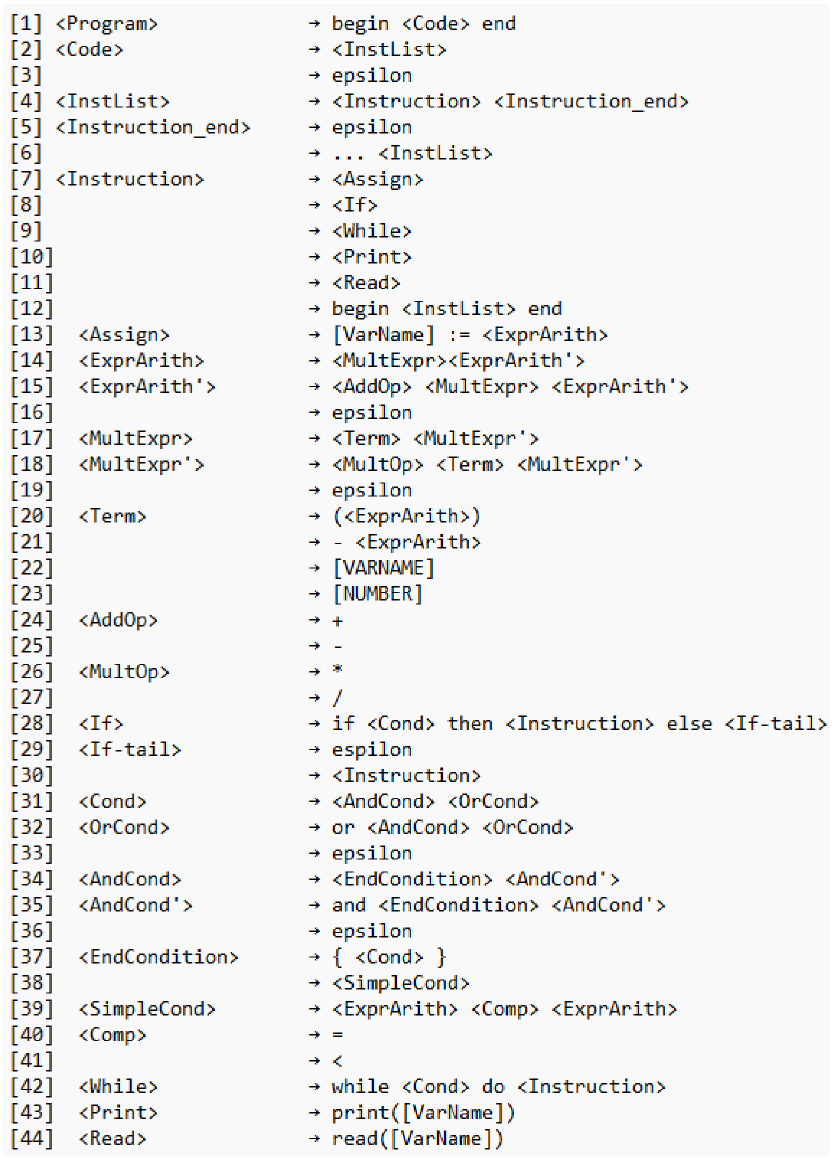
```
<ExprArith>        → <ExprArith> <AddOp> <MultExpr>
                   → <MultExpr>
<MultExpr>         → <MultExpr> <MultOp> <Term>
                   → <Term>
<Term>             → (<ExprArith>)
                   → - <ExprArith>
                   → [VarName]
                   → [Number]
<AddOp>            → +
                   → -
<MultOp>           → *

<Cond>             → <Cond> or <AndCond>
                   → <AndCond>
<AndCond>          → <AndCond> and <EndCondition>
<AndCond>          → <EndCondition>
<EndCondition>     → { <Cond> }
                   → <SimpleCond>
<SimpleCond>       → <ExprArith> <Comp> <ExprArith>
```

FIGURE 2 : right priority grammar

There still ambiguity and left recursion in the grammar. It can be changed by factorising expression and by adding intermediate state.

Factorisation has been used for <InstList> and <If> and we add new state for the different rules of arithmetic expression and condition

The final grammar is presented at the Figure 3.



```
[1] <Program>            → begin <Code> end
[2] <Code>               → <InstList>
[3]                      → epsilon
[4] <InstList>           → <Instruction> <Instruction_end>
[5] <Instruction_end>    → epsilon
[6]                      → ... <InstList>
[7] <Instruction>        → <Assign>
[8]                      → <If>
[9]                      → <While>
[10]                     → <Print>
[11]                     → <Read>
[12]                     → begin <InstList> end
[13]  <Assign>           → [VarName] := <ExprArith>
[14]  <ExprArith>        → <MultExpr><ExprArith'>
[15]  <ExprArith'>       → <AddOp> <MultExpr> <ExprArith'>
[16]                     → epsilon
[17]  <MultExpr>         → <Term> <MultExpr'>
[18]  <MultExpr'>        → <MultOp> <Term> <MultExpr'>
[19]                     → epsilon
[20]  <Term>             → (<ExprArith>)
[21]                     → - <ExprArith>
[22]                     → [VARNAME]
[23]                     → [NUMBER]
[24]  <AddOp>            → +
[25]                     → -
[26]  <MultOp>           → *
[27]                     → /
[28]  <If>               → if <Cond> then <Instruction> else <If-tail>
[29]  <If-tail>          → espilon
[30]                     → <Instruction>
[31]  <Cond>             → <AndCond> <OrCond>
[32]  <OrCond>           → or <AndCond> <OrCond>
[33]                     → epsilon
[34]  <AndCond>          → <EndCondition> <AndCond'>
[35]  <AndCond'>         → and <EndCondition> <AndCond'>
[36]                     → epsilon
[37]  <EndCondition>     → { <Cond> }
[38]                     → <SimpleCond>
[39]  <SimpleCond>       → <ExprArith> <Comp> <ExprArith>
[40]  <Comp>             → =
[41]                     → <
[42]  <While>            → while <Cond> do <Instruction>
[43]  <Print>            → print([VarName])
[44]  <Read>             → read([VarName])
```

FIGURE 3 : Final Pascalmaispresque grammar

## 2.2 Grammar properties

The first thing we can deduce from the grammar rules is the First[1] and Follow[1] table. This is shown in the Table 1.

| Symbol | $\text{First}^1$ | $\text{Follow}^1$ |
|---|---|---|
| Program | begin | $\varepsilon$ |
| Code | $\varepsilon$ [VARNAME] if while print read begin | end |
| InstList | [VARNAME] if while print read begin | end |
| Instruction_end | $\varepsilon$ ... | end |
| Instruction | [VARNAME] if while print read begin | ... else end |
| Assign | [VARNAME] | ... else end |
| ExprArith | ( - [VARNAME] [NUMBER] | ... else end ) = < and or then do |
| ExprArith' | + - $\varepsilon$ | ... else end ) = < and or then do |
| MultExpr | ( - [VARNAME] [NUMBER] | ... else end ) = < and or then + - do |
| MultExpr' | * / $\varepsilon$ | ... else end ) = < and or then + - do |
| Term | ( - [VARNAME] [NUMBER] | ... else end ) * / = < and or then + - do |
| AddOp | + - | ( - [VARNAME] [NUMBER] |
| MultOp | * / | ( - [VARNAME] [NUMBER] |
| If | if | ... else end |
| If-tail | $\varepsilon$ [VARNAME] if while print read begin | ... else end |
| Cond | { ( - [VARNAME] [NUMBER] | then } do |
| OrCond | or $\varepsilon$ | then } do |
| AndCond | { ( - [VARNAME] [NUMBER] | or then } |
| AndCond' | and $\varepsilon$ | or then } do |
| EndCondition | { ( - [VARNAME] [NUMBER] | and or then } do |
| SimpleCondition | ( - [VARNAME] [NUMBER] | and or then } do |
| Comp | #SYNTAX | ( - [VARNAME] [NUMBER] |
| While | while | ... else end |
| Print | print | ... else end |
| Read | read | ... else end |

TABLE 1 : $\text{First}^1$ and $\text{Follow}^1$ of the grammar

This grammar is strong LL(1) if :

$$First^1(\alpha_1 Follow^1(A)) \cap First^1(\alpha_2 Follow^1(A)) = \varnothing$$

With all the pair rules from the grammar under the $A \longrightarrow \alpha_n$.

**PROOF** :
First we check the definition on the variable **code** :
$First^1(\varepsilon Follow^1(code)) = First^1(Follow^1(code)) = \{end\}$
$First^1(< InstList > Follow^1(code)) = \{[VARNAME], if, while, print, read, begin\}$
And the intersection between these two sets is the empty set.

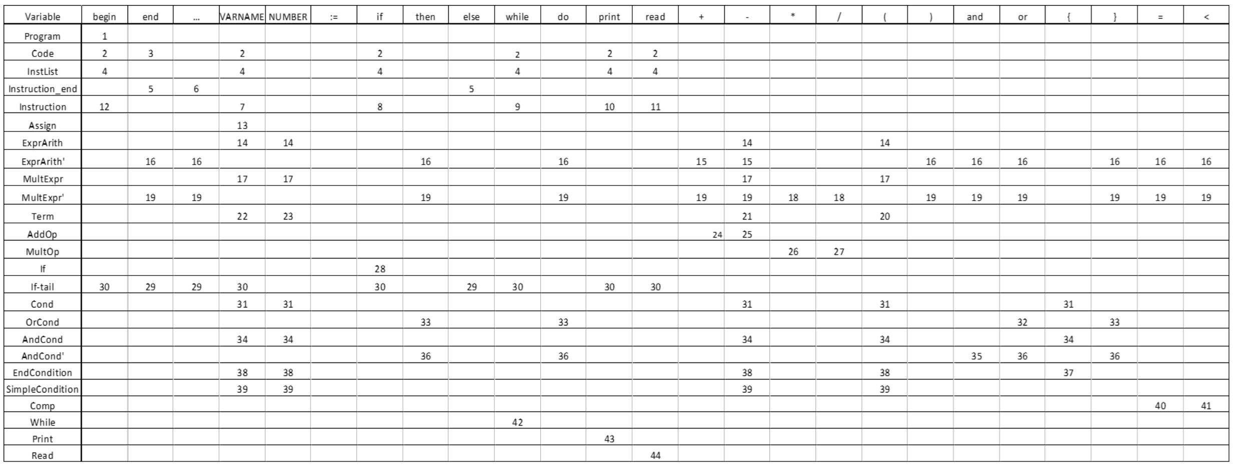
Now for the variable **Instruction_end** :
$First^1(\varepsilon Follow^1(Instruction\_end)) = \{end\}$
$First^1(... < InstList > Follow^1(Instruction\_end)) = \{...\}$
For the **Instruction** variable :
$First^1(< Assign > Follow^1(< Instruction >)) = \{[VARNAME]\}$
$First^1(< if > Follow^1(< Instruction >)) = \{if\}$
$First^1(< while > Follow^1(< Instruction >)) = \{while\}$
$First^1(< for > Follow^1(< Instruction >)) = \{for\}$
$First^1(< print > Follow^1(< Instruction >)) = \{print\}$
$First^1(< read > Follow^1(< Instruction >)) = \{read\}$
$First^1(begin < InstList > endFollow^1(< Instruction >)) = \{begin\}$
For the **ExprArith'** variable :
$First^1(< AddOp >< MultExpr >< ExprArith' > Follow^1(ExprArith')) = \{+, -\}$
$First^1(\varepsilon Follow^1(ExprArith')) = \{..., else, end, ), =, <, and, or, then, do\}$
$First^1(< MultExpr > Follow^1(ExprArith')) = \{(, -, [VARNAME], [NUMBER]\}$
The minus sign are not the same.
For the **MultExpr'** variable :
$First^1(< MultOp >< Term >< MultExpr' > Follow^1(MultExpr')) = \{*, /\}$
$First^1(\varepsilon Follow^1(MultExpr')) = \{..., else, end, ), =, <, and, or, then, +, -, do\}$
For the **Term** variable :
$First^1((< ExprArith >)Follow^1(Term)) = \{(\}$
$First^1(- < ExprArith > Follow^1(Term)) = \{-\}$
$First^1([VARNAME]]Follow^1(Term)) = \{[VARNAME]\}$
$First^1([NUMBER]]Follow^1(Term)) = \{[NUMBER]\}$
For the **AddOp** and **MultOp**, we don't need to check the definition because the terminals are differents.
For the **If-tail** variable :
$First^1(\varepsilon Follow^1(If - tail)) = \{..., else, end\}$
$First^1(< Instruction > Follow^1(If{-}tail)) = \{[VARNAME], if, while, print, read, begin\}$
For the **OrCond** variable :
$First^1(or < AndCond >< OrCond > Follow^1(OrCond)) = \{or\}$
$First^1(\varepsilon Follow^1(OrCond)) = \{then, \}, do\}$
For the **AndCond'** variable :
$First^1(and < EndCondition >< AndCond' > Follow^1(AndCond')) = \{and\}$
$First^1(\varepsilon Follow^1(AndCond')) = \{or, then, \}\}$
For the **EndCondition** variable :
$First^1(\{< Cond >\}Follow^1(EndCondition)) = \{\{\}$
$First^1(< SimpleCond > Follow^1(EndCondition)) = \{(, -, [VARNAME, [Number]\}$

This proves that the grammar is strong LL(1). In conclusion, the grammar is LL(1), because all strong LL(1) grammars are LL(1) grammars.

## 2.3 Action Table

The action table completely describes the behaviour of the parser. Each cell indicates which rule should be used depending on the line (potential tops of stack) and rows (potential look-aheads). The action table of the grammar in Figure 3 is shown

in Figure 4.
To build the action table we use the First[1] and Follow[1]. If a cell is empty, it means that this case is not possible, an error is raised.



| Variable | begin | end | ... | VARNAME | NUMBER | := | if | then | else | while | do | print | read | + | - | * | / | ( | ) | and | or | { | } | = | < |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Program | 1 | | | | | | | | | | | | | | | | | | | | | | | | |
| Code | 2 | 3 | | 2 | | | 2 | | | 2 | | 2 | 2 | | | | | | | | | | | | |
| InstList | 4 | | | 4 | | | 4 | | | 4 | | 4 | 4 | | | | | | | | | | | | |
| Instruction_end | | 5 | 6 | | | | | | 5 | | | | | | | | | | | | | | | | |
| Instruction | 12 | | | 7 | | | 8 | | | 9 | | 10 | 11 | | | | | | | | | | | | |
| Assign | | | | 13 | | | | | | | | | | | | | | | | | | | | | |
| ExprArith | | | | 14 | 14 | | | | | | | | | | 14 | | | 14 | | | | | | | |
| ExprArith' | | 16 | 16 | | | | | 16 | | | 16 | | | 15 | 15 | | | | 16 | 16 | 16 | | 16 | 16 | 16 |
| MultExpr | | | | 17 | 17 | | | | | | | | | | 17 | | | 17 | | | | | | | |
| MultExpr' | | 19 | 19 | | | | | 19 | | | 19 | | | 19 | 19 | 18 | 18 | | 19 | 19 | 19 | | 19 | 19 | 19 |
| Term | | | | 22 | 23 | | | | | | | | | | 21 | | | 20 | | | | | | | |
| AddOp | | | | | | | | | | | | | | 24 | 25 | | | | | | | | | | |
| MultOp | | | | | | | | | | | | | | | | 26 | 27 | | | | | | | | |
| If | | | | | | | 28 | | | | | | | | | | | | | | | | | | |
| If-tail | 30 | 29 | 29 | 30 | | | 30 | | 29 | 30 | | 30 | 30 | | | | | | | | | | | | |
| Cond | | | | 31 | 31 | | | | | | | | | | 31 | | | 31 | | | | 31 | | | |
| OrCond | | | | | | | | 33 | | | 33 | | | | | | | | | | 32 | | 33 | | |
| AndCond | | | | 34 | 34 | | | | | | | | | | 34 | | | 34 | | | | 34 | | | |
| AndCond' | | | | | | | | 36 | | | 36 | | | | | | | | | 35 | 36 | | 36 | | |
| EndCondition | | | | 38 | 38 | | | | | | | | | | 38 | | | 38 | | | | 37 | | | |
| SimpleCondition | | | | 39 | 39 | | | | | | | | | | 39 | | | 39 | | | | | | | |
| Comp | | | | | | | | | | | | | | | | | | | | | | | | 40 | 41 |
| While | | | | | | | | | | 42 | | | | | | | | | | | | | | | |
| Print | | | | | | | | | | | | 43 | | | | | | | | | | | | | |
| Read | | | | | | | | | | | | | 44 | | | | | | | | | | | | |

Figure 4 : Action table

## 2.4 Implementation of the parser

To create our parser, we needed to incorporate our grammar and action table into our program. For this purpose, we created three classes : ActionTableReader, GrammarAnalyzer, and State.
The **State** class is essentially an enumeration of all possible variables, referred to as states.
The purpose of the **ActionTableReader** class is to create a table in which it is possible to retrieve the number of a rule based on a state and a lexical unit.
The **GrammarAnalyzer** class serves a similar purpose. It creates a table in which one can retrieve all the right-hand side elements of a rule based on a rule number and a state.
The arrays in the **ActionTableReader** and **GrammarAnalyzer** classes are created from the corresponding CSV files located in the 'doc/grammar_doc' directory.

Finally, we created the **Parser** class, in which we implemented a recursive descent parser. This parser is capable of generating the leftmost derivation as well as the parse tree of the input.
The parser is initialized with a starting state <Program> and the first symbol of the input is read thanks to the **LexicalAnalyzer**.
The parse function serves as the core of the parser implementation. Given a specific parsing state, it realize the recursive parsing process. The function begins by updating the current state and retrieving the corresponding rule number from the action table using the **ActionTableReader**. The rule number is recorded in the *usedrules* list for reference.

To handle potential errors, the function checks if a rule is found for the current state and symbol. If no rule is found, it means that we are in a grammatically incorrect situation. An error message is then displayed, and the program terminates.

Subsequently, the function retrieves the elements produced by the identified rule from the grammar using the **GrammarAnalyzer**. These elements represent the components of the rule, including states and lexical units. The function then constructs a parse tree, initializing a node with the current state.
The parsing process iterates through each element of the rule. If the element is a state, the function recursively calls itself to parse that state, generating a subtree. This subtree will be added to the children of the tree created above. If the element is a lexical unit, it is compared to the current symbol. If a match occurs, a corresponding node is created in the parse tree, and the parser advances to the next symbol. $\varepsilon$ elements, representing empty productions, are appropriately handled.
In case of a symbol mismatch, the function reports an error and terminates the program.

Finally, the function returns the constructed parse tree, representing the hierarchical structure of the input based on the applied grammar rules.

With the Euclid.pmp file we get the following list of rules and tree :

1 2 4 11 44 6 4 11 44 6 4 9 42 31 34 38 39 14 17 23 19 16 41 14 17 22 19 16 36 33 12 4 7 13 14 17 22 19 16 6 4 9 42 31 34 38 39 14 17 22 19 16 41 14 17 22 19 15 24 17 23 19 16 36 33 7 13 14 17 22 19 15 25 17 22 19 16 6 4 7 13 14 17 22 19 16 6 4 7 13 14 17 22 19 16 5 6 4 10 43 5
The parse tree is represented at the Figure 5.

## 3 LLVM

This part is about generating LLVM code from the .pmp source.

### 3.1 LLVMWriter class

A class called **LLVMWriter** has been created to delegate this task. It takes in its constructor the list of rules used during the recursive descent of the parser. In addition, it receives a list of all variable names linked to the lexical unit VARNAME in the order of their apparition. It also gets a list of numbers linked to the lexical unit NUMBER. The class **LLVMWriter** is represented in the Table 2.

| LLVMWriter |
|---|
| - llvmCode : StringBuilder<br>- ruleCoutner : int<br>- varCounter : int<br>- nbCounter : int<br>- tempVarCounter : int<br>- labelCounter : int<br><br>- rules : ArrayList<Integer><br>- varList : ArrayList<String><br>- nbList : ArrayList<String><br>- varMap : Map<String, Boolean> |
| + LLVMWriter()<br>- begin()<br>+ writeToFile()<br>- one function per rule |

Table 2 : LLVMWriter class

- llvmCode : It is used to build the llvm code and it will be print in the correct *output.ll* file a the end of the program.
- ruleCounter : It allows to iterate over every rule used during the recursive descent in the parser.
- varCounter : It allows to iterate over every variable that appear in the code. This list of variables is store in *varList* .
- nbCounter : As varCounter, it allows to iterate over each number that appears in the code. This list of number is store in *nbList*.
- tempVarCounter : The integer allow to create virtual registers. It will be incremented every time a new virtual register is needed.
- labelCounter : Allows to create labels with changing names for every new label (if or while).
- rules : The list of the rules used to parse the input file.
- varList : List of each variable that appears in the input file in the right order.
- nbList : List of each number that appears in the input file in the right order.
- varMap : Map a boolean to each distinct variable that appears in the input file. Every booleans are initialized to false. When a variable is encountered for the first time, the corresponding boolean is set to true. It allows not allocating memory for an already allocated variable.

## 3.2 LLVMWriter logic

### 3.2.1 Algorithm

At the beginning, the program check if a *print()* or *read()* is used. If one of them is used, the definition of *println()* and *readint()* are written at the beginning of llvm-Code.

The algorithm inside the class uses the list of used rules from the parser to determine which method to call. There is one method per grammar rule.
When a method is called, it adds string corresponding to the rule in the *llvmCode* StringBuilder attribute. Since the methods are called in a pseudo-recursive way, the llvm code is writen in the right order.
The algorithm of writing code :

1. Method is called
2. Add code to the LLVMCode
3. Call the next method by checking the next rule (1). If there are no rules associated go to (4)
4. Add the end part of the code to the LLVMCode and exit the method

### 3.2.2 Operations

For each operation, LLVM must have the 2 operands stored in registers. Because the program uses registers with names that contain an incrementing number, it has to remember which operands to use in the operations.
Let's analyse the following exemple :

$$a := 1 + 2 * 3 \tag{1}$$

Assuming it's the only line in the input code.

1. The parser first see a 1 and save it in %0
2. It see a + and knows it has to do an addition.
3. It see a 2 and store it in %1
4. It see a * and knows it has to do a multiplication
5. It see a 3 and store it in %2
6. save %1 * %2 in %3
7. save %0 + %3 in %4

For step (6) the algorithm has to remember in step (4) which was the last tempVar saved. It will then multiply this tempVar by the last tempVar saved in step 6, which in this case is %2.

The same logic is used to do the addition.

### 3.2.3 Labels

For `WHILE` and `IF`, Llvm uses labels to execute only a specific block of code based on a condition. Each label in LLvm must have a unique name. The program uses the same logic for the label name as for the tempVar name. So the LLvm code will have labels if1, if2, if3, etc... To put everything in the in the right order, the program uses this kind of logic : (note that the number is not relevant).

- For if construction

    1. Condition code
    2. Put the following line :
       ```
       br i1 %8, label %if.then1, label %if.else1
       ifthen1:
       ```
    3. Instruction code for *then* label
    4. Put the following line :
       ```
       br label %if.end1
       if.else1:
       ```
    5. Instruction code for *else* label
    6. Put the following line :
       ```
       br label %if.end1
       if.end1:
       ```

  The same logic is used for while construction

# 4 Conclusion

In conclusion, the compiler uses all three components of this project : the scanner, the parser and the translation to LLVM to generate low-level code from our input code. All the tests included in the project were successful, ensuring the correct functioning of the program.

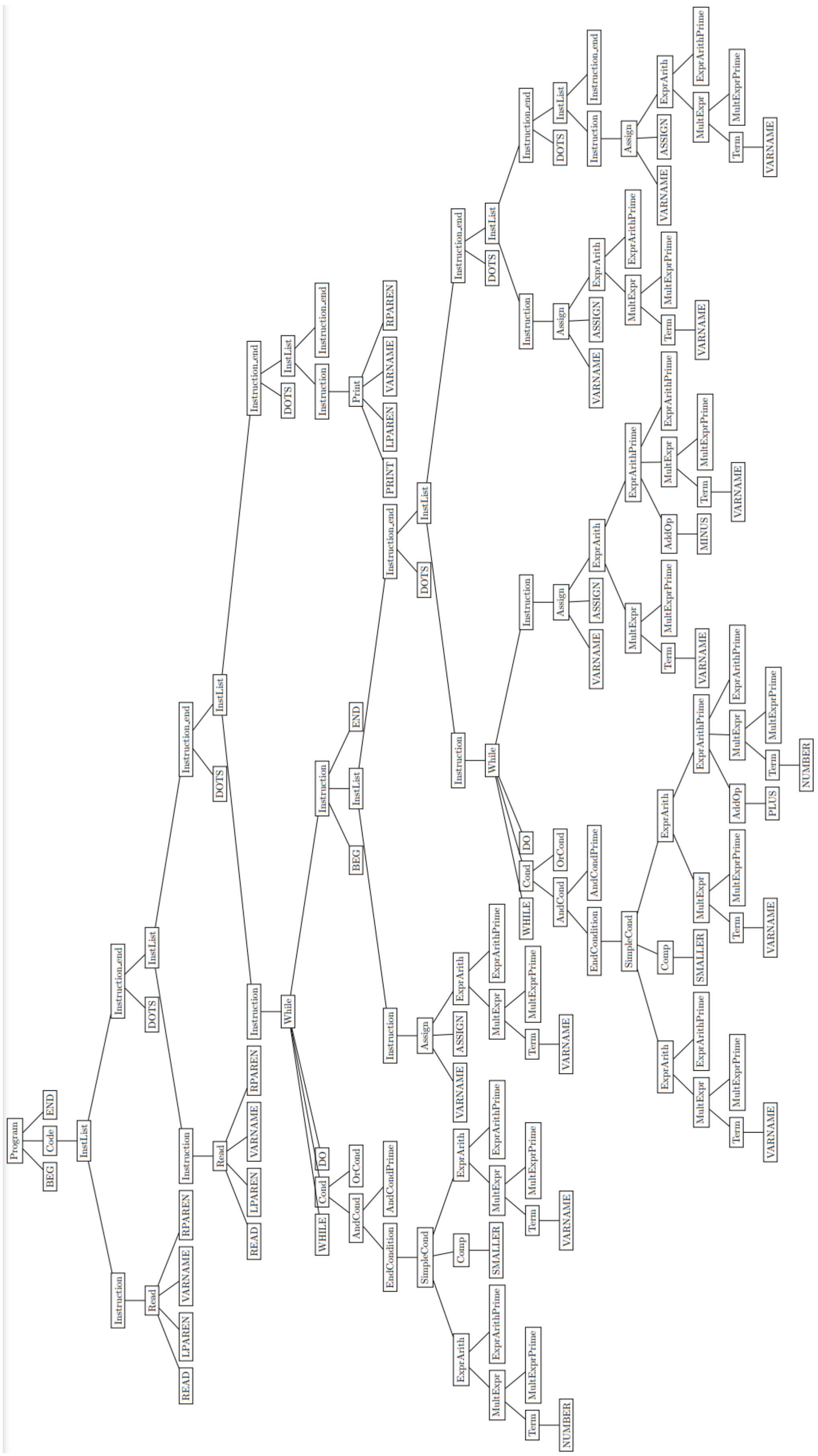


FIGURE 5 : Parse tree of the euclid.pmp file