Université libre de Bruxelles
École Polytechnique de Bruxelles
M-IRIFS

INFO-F403 - Introduction to language theory and compiling - 2023/2024

PASCALmaispresque

Project – Part 2

Memberss :
Younes El Mokhtari
Ismail Jeq

Professeur :
Gilles GEERAERTS

21 novembre 2023

1 Description of the project

This project aims to build a compiler for PASCALMAISPRESQUE, a simple imperative language. In this second part, we designed a recursive descent LL(1) parser for the PASCALMAISPRESQUE grammar. The goal is to parse an input string, output on the terminal its leftmost derivation and generate its parse tree in LaTeX format.

1.1 Format

The `Format` class serves as a utility class for defining characters corresponding to the grammar format in a file and their LaTeX equivalents that will be used for the building process of the parse tree.

1.2 LexicalUnit

The `LexicalUnit` enum defines lexical units or token types recognized by the `LexicalAnalyzer`. Each constant in the enum corresponds to a specific token type encountered during scanning and parsing.

1.3 Symbol

The `Symbol` class represents tokens for the scanner and symbols (terminal or variable) for the grammar.

1.4 Rule

The `Rule` class represents grammar rules, including left-hand side, right-hand side, and rule number.

1.5 ContextFreeGrammar

The `ContextFreeGrammar` class reads grammar rules from a file, sets up variables, terminals, rules, and the start symbol.

1.6 ParseTools

The `ParseTools` class contains different grammar-related tools. It computes First and Follow sets, checks LL(k) grammar properties, constructs LL(1) action tables. The constructed action table will then be used by the parser for the parsing.

1.7 LexicalAnalyzer

The `LexicalAnalyzer` class is the scanner and will be used for scanning the input file.

1.8 Parser

The `Parser` class is the implemented recursive descent LL(1) parser that will be used for parsing the input for outputing its left most derivation and building its parse tree.

1.9 ParseTree

The `ParseTree` class represents parse trees and provides methods to output them in LaTeX in TikZ or Forest format.

1.10 Main

The `Main` class checks command-line options, scans the input file, and then passes the input to the parser for parsing. There are two ways to proceed : either parse the input and output the leftmost derivation, or parse the input, produce the leftmost derivation and then construct the parse tree.

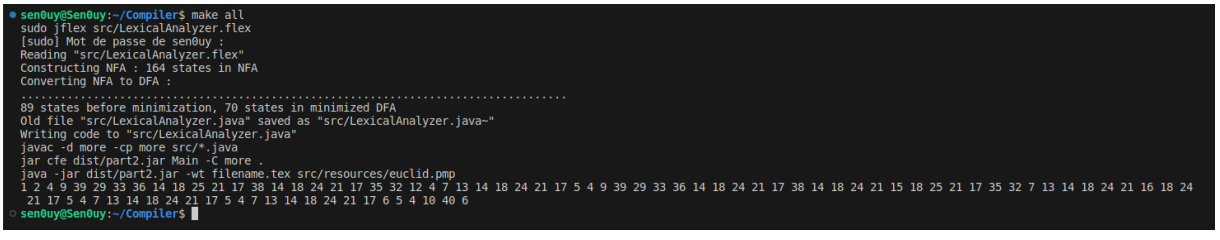


FIGURE 1 – Left most derivation for euclid.pmp

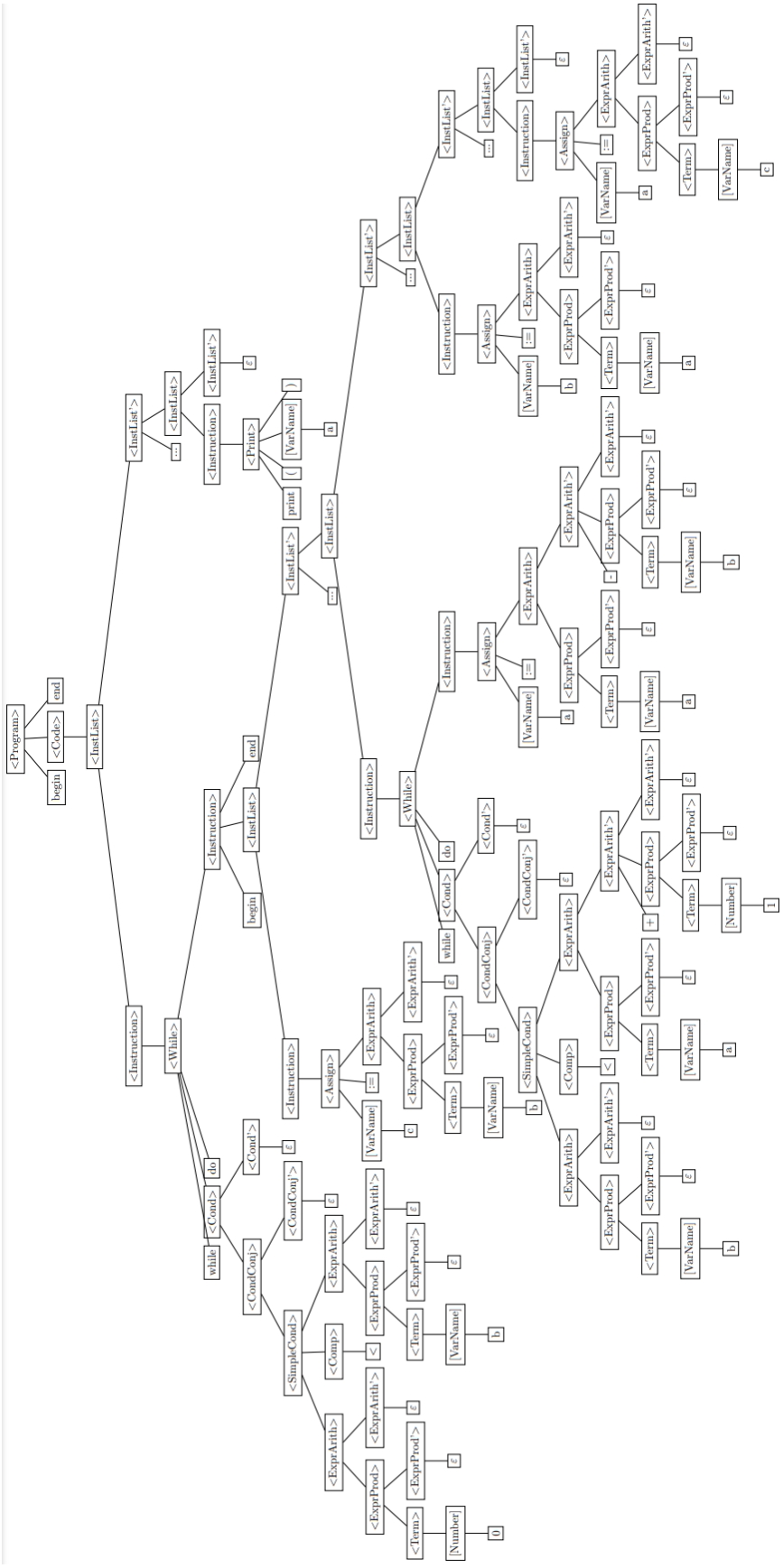


FIGURE 2 – Parse tree for euclid.pmp

2 Implementation choices

For this project, we did different implementation choices in our program :

2.1 ContextFreeGrammar

The context-free grammar is created from the file cfg.pmp in the folder src/resources. The file must respect a certain format defined in Format.java. It means that any grammar can be implemented and will be checked whether it is LL(1) or not.

2.2 ParseTools

ParseTools has differents tools related to a context-free grammar :

- First$^k(X)$
- Follow$^k(X)$
- First$^k(\alpha$Follow$^k(A))$

It means that for any context-free grammar, it can compute First sets, Follow sets, check if the grammar is LL(1) and construct its LL(1) action table automatically.

2.3 Parser

The parser implemented an LL(1) recursive descent algorithm, where the input is processed from left to right, and the corresponding leftmost derivation is output on stdout. Then, we build the parse tree using a recursive function that generates child nodes for the parent node based on this leftmost derivation.

3 Modified grammar

For this project, it was asked to transform the PASCALMAISPRESQUE grammar in order to :

1. Remove unproductive and/or unreachable variables
2. Make the grammar non-ambiguous by taking into account the priority and the associativity of the operators (detailled in section 3.3)
3. Remove left-recursion and apply factorisation

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

FIGURE 3 – PASCALMAISPRESQUE grammar

3.1 Remove unproductive and unreachable variables

To remove unproductive variables, we computed the set of productive symbols for each rule. All symbols are productive except the <For> symbol. So we have to remove the rule number 9.

To remove unreachable variables, we computed the set of reachable symbols for each rule. We obtained that all symbols are reachable and we obtain the following grammar at this step :

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

FIGURE 4 – Grammar after removing unproductive and unreachable variables

3.2 Make the grammar non-ambiguous

To make the grammar non-ambiguous, we have to take into account the priority and associativity of these operators :

Operators	Associativity
− (unary)	right
⋆, /	left
+, − (binary)	left
<, =	left
and	left
or	left

FIGURE 5 – Priority and associativity of the PASCALMAISPRESQUEoperators (operators are sorted indecreasing order of priority)

We obtained the following grammar at this step :

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

FIGURE 6 – Grammar after making it non-ambiguous

3.3 Remove left-recursion and apply factorisation

We have left recursion for the variables <ExprArith>, <ExprProd>, <Cond> and <CondConj> so for the rules 13, 14, 16, 17, 25, and 28.

We applied factorisation for the variables <InstList> and <If>. At the end, our transformed grammar is the following grammar :

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

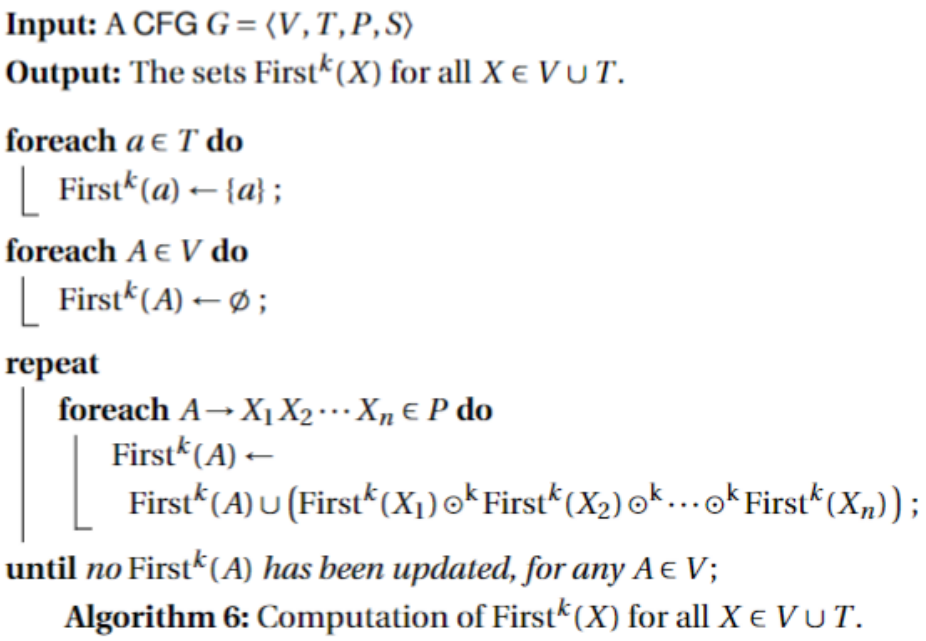
FIGURE 7 – Transformed grammar

4 Action table

For this project, it was asked to construct the action table of an LL(1) parser for the transformed grammar. To do this, we computed the First sets and Follow sets for the transformed grammar using algorithms provided in the syllabus. We then construct the action table using the algorithm.

4.1 Computation of First sets

To compute the First sets, we used the following algorithm :



Input: A CFG $G = \langle V, T, P, S \rangle$
Output: The sets $\text{First}^k(X)$ for all $X \in V \cup T$.

foreach $a \in T$ **do**
  $\text{First}^k(a) \leftarrow \{a\}$;
foreach $A \in V$ **do**
  $\text{First}^k(A) \leftarrow \emptyset$;
repeat
  foreach $A \rightarrow X_1 X_2 \cdots X_n \in P$ **do**
    $\text{First}^k(A) \leftarrow$
    $\text{First}^k(A) \cup \left(\text{First}^k(X_1) \odot^k \text{First}^k(X_2) \odot^k \cdots \odot^k \text{First}^k(X_n)\right)$;
until *no* $\text{First}^k(A)$ *has been updated, for any* $A \in V$;

Algorithm 6: Computation of $\text{First}^k(X)$ for all $X \in V \cup T$.

FIGURE 8 – First(k) algorithm

We obtained the following First sets :

```
First1(<Program>) = [begin]
First1(<Code>) = [ε, print, read, begin, [VarName], while, if]
First1(<InstList>) = [print, read, begin, [VarName], while, if]
First1(<InstList'>) = [ε, ...]
First1(<Instruction>) = [print, read, begin, [VarName], while, if]
First1(<Assign>) = [[VarName]]
First1(<ExprArith>) = [[VarName], [Number], -, ()]
First1(<ExprArith'>) = [ε, +, -]
First1(<ExprProd>) = [[VarName], [Number], -, ()]
First1(<ExprProd'>) = [ε, *, /]
First1(<Term>) = [[VarName], [Number], -, ()]
First1(<If>) = [if]
First1(<If'>) = [ε, print, read, begin, [VarName], while, if]
First1(<Cond>) = [[VarName], [Number], {, -, ()]
First1(<Cond'>) = [ε, or]
First1(<CondConj>) = [[VarName], [Number], -, ()]
First1(<CondConj'>) = [ε, and]
First1(<SimpleCond>) = [[VarName], [Number], -, ()]
First1(<Comp>) = [=, <]
First1(<While>) = [while]
First1(<Print>) = [print]
First1(<Read>) = [read]
```

FIGURE 9 – First(1) sets

4.2 Computation of Follow sets

To compute the Follow sets, we used the following algorithm :

Input: A CFG $G = \langle V, T, P, S \rangle$
Output: The sets $\text{Follow}^k(X)$ for all $X \in V$.

foreach $A \in V \setminus \{S\}$ **do**
  $\text{Follow}^k(A) \leftarrow \emptyset$;
$\text{Follow}^k(S) \leftarrow \{\varepsilon\}$;
repeat
  foreach $A \rightarrow \alpha B \beta \in P$ *(with* $B \in V$ *and* $\alpha, \beta \in (V \cup T)^*$*)* **do**
    $\text{Follow}^k(B) \leftarrow \text{Follow}^k(B) \cup \left(\text{First}^k(\beta) \odot^k \text{Follow}^k(A)\right)$;
until *no* $\text{Follow}^k(A)$ *has been updated, for any* $A \in V$;

Algorithm 7: Computation of $\text{Follow}^k(X)$ for all $X \in V$.

FIGURE 10 – Follow(k) algorithm

We obtained the following Follow sets :

```
Follow1(<Program>) = [ε]
Follow1(<Code>) = [end]
Follow1(<InstList>) = [end]
Follow1(<InstList'>) = [end]
Follow1(<Instruction>) = [..., end, else]
Follow1(<Assign>) = [..., end, else]
Follow1(<ExprArith>) = [..., end, =, <, and, or, then, else, }, do]
Follow1(<ExprArith'>) = [..., end, =, <, and, or, then, else, }, do]
Follow1(<ExprProd>) = [+, -, ..., end, =, <, and, or, then, else, }, do]
Follow1(<ExprProd'>) = [+, -, ..., end, =, <, and, or, then, else, }, do]
Follow1(<Term>) = [*, /, +, -, ..., end, ), =, <, and, or, then, else, }, do]
Follow1(<If>) = [..., end, else]
Follow1(<If'>) = [..., end, else]
Follow1(<Cond>) = [then, }, do]
Follow1(<Cond'>) = [then, }, do]
Follow1(<CondConj>) = [or, then, }, do]
Follow1(<CondConj'>) = [or, then, }, do]
Follow1(<SimpleCond>) = [and, or, then, }, do]
Follow1(<Comp>) = [[VarName], [Number], -, ()]
Follow1(<While>) = [..., end, else]
Follow1(<Print>) = [..., end, else]
Follow1(<Read>) = [..., end, else]
```

FIGURE 11 – Follow(1) sets

4.3 Check that transformed grammar is LL(1)

To check if the transformed grammar is LL(1), we used the definition of strong LL(1) :

Definition 5.12 (Strong LL(k) CFG). A CFG $G = \langle V, T, P, S \rangle$ is *strong* LL(k) iff, for all pairs of rules $A \to \alpha_1$ and $A \to \alpha_2$ in P (with $\alpha_1 \neq \alpha_2$):

$$\text{First}^k\left(\alpha_1 \text{Follow}^k(A)\right) \cap \text{First}^k\left(\alpha_2 \text{Follow}^k(A)\right) = \emptyset$$

FIGURE 12 – Strong LL(k) definition

For each rule with the same left hand side, the intersection set is empty, meaning that the transformed grammar is LL1 :

```
[2] First1([<InstList>, Follow, <Code>]) = [print, read, begin, [VarName], while, if]
[3] First1([ε, Follow, <Code>]) = [end]

[5] First1([..., <InstList>, Follow, <InstList'>]) = [...]
[6] First1([ε, Follow, <InstList'>]) = [end]

[7] First1([<Assign>, Follow, <Instruction>]) = [[VarName]]
[8] First1([<If>, Follow, <Instruction>]) = [if]
[9] First1([<While>, Follow, <Instruction>]) = [while]
[10] First1([<Print>, Follow, <Instruction>]) = [print]
[11] First1([<Read>, Follow, <Instruction>]) = [read]
[12] First1([begin, <InstList>, end, Follow, <Instruction>]) = [begin]

[15] First1([+, <ExprProd>, <ExprArith'>, Follow, <ExprArith'>]) = [+]
[16] First1([-, <ExprProd>, <ExprArith'>, Follow, <ExprArith'>]) = [-]
[17] First1([ε, Follow, <ExprArith'>]) = [..., end, =, <, and, or, then, else, }, do]

[19] First1([*, <Term>, <ExprProd'>, Follow, <ExprProd'>]) = [*]
[20] First1([/, <Term>, <ExprProd'>, Follow, <ExprProd'>]) = [/]
[21] First1([ε, Follow, <ExprProd'>]) = [+, -, ..., end, =, <, and, or, then, else, }, do]

[23] First1([-, <Term>, Follow, <Term>]) = [-]
[24] First1([(, <Term>, ), Follow, <Term>]) = [(]
[25] First1([[VarName], Follow, <Term>]) = [[VarName]]
[26] First1([[Number], Follow, <Term>]) = [[Number]]

[28] First1([<Instruction>, Follow, <If'>]) = [print, read, begin, [VarName], while, if]
[29] First1([ε, Follow, <If'>]) = [..., end, else]

[31] First1([<CondConj>, <Cond'>, Follow, <Cond>]) = [[VarName], [Number], -, (]
[32] First1([{, <Cond>, }, Follow, <Cond>]) = [{]

[34] First1([or, <CondConj>, <Cond'>, Follow, <Cond'>]) = [or]
[35] First1([ε, Follow, <Cond'>]) = [then, }, do]

[37] First1([and, <SimpleCond>, <CondConj'>, Follow, <CondConj'>]) = [and]
[38] First1([ε, Follow, <CondConj'>]) = [or, then, }, do]

[40] First1([=, Follow, <Comp>]) = [=]
[41] First1([<, Follow, <Comp>]) = [<]
```

FIGURE 13 – First(1)(alpha Follow(1)(A)

4.4 Construct the action table

To construct the action table, we used the following algorithm :

Input: A CFG $G = \langle P, T, V, S \rangle$.
Output: The LL(1) action M table of G

```
/* Initialisation of the table                                    */
foreach a ∈ T do
    foreach A ∈ V do
        M[A, a] ← ∅ ;
    foreach b ∈ T \ {a} do
        M[b, a] ← ∅ ;
    M[a, a] ← {M} ;
M[$, $] ← {A} ;
/* Adding the 'Produce' actions                                   */
foreach rule A → α ∈ P with number i do
    foreach a ∈ First(α · Follow(A)) do
        M[A, a] ← M[A, a] ∪ {i} ;
return M ;
```

Algorithm 8: Systematic construction of the LL(1) table of a CFG.

FIGURE 14 – LL(1) action table algorithm

We obtained the following action table :

	begin	end	...	[VarName]	:=	+	-	*	/	(	)	[Number]	if	then	else	{	}	or	and	=	<	while	do	print	read
<Program>	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<Code>	2	3	0	2	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	2	0	2	2
<Instlist>	4	0	0	4	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	4	0	4	4
<Instlist'>	0	6	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<Instruction>	12	0	0	7	0	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	9	0	10	11
<Assign>	0	0	0	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<Exprarith>	0	0	0	14	0	0	14	0	0	14	0	14	0	0	0	0	0	0	0	0	0	0	0	0	0
<Exprarith'>	0	17	17	0	0	15	16	0	0	0	0	0	0	17	17	0	17	17	17	17	17	0	17	0	0
<ExprProd>	0	0	0	18	0	0	18	0	0	18	0	18	0	0	0	0	0	0	0	0	0	0	0	0	0
<ExprProd'>	0	21	21	0	0	21	21	19	20	0	0	0	0	21	21	0	21	21	21	21	21	0	21	0	0
<Term>	0	0	0	24	0	0	22	0	0	23	0	25	0	0	0	0	0	0	0	0	0	0	0	0	0
<If>	0	0	0	0	0	0	0	0	0	0	0	0	26	0	0	0	0	0	0	0	0	0	0	0	0
<If'>	27	28	28	27	0	0	0	0	0	0	0	0	27	0	28	0	0	0	0	0	0	27	0	27	27
<Cond>	0	0	0	29	0	0	29	0	0	29	0	29	0	0	0	30	0	0	0	0	0	0	0	0	0
<Cond'>	0	0	0	0	0	0	0	0	0	0	0	0	0	32	0	0	32	31	0	0	0	0	32	0	0
<CondConj>	0	0	0	33	0	0	33	0	0	35	0	33	0	0	0	0	0	0	0	0	0	0	0	0	0
<CondConj'>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	35	35	34	0	0	0	35	0	0
<SimpleCond>	0	0	0	36	0	0	36	0	0	36	0	36	0	0	0	0	0	0	0	0	0	0	0	0	0
<Comp>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	37	38	0	0	0	0
<While>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	39	0	0	0
<Print>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	40	0
<Read>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	41
begin	M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
end	0	M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
...	0	0	M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[VarName]	0	0	M	M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:=	0	0	0	0	M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+	0	0	0	0	M	M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-	0	0	0	0	M	M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
*	0	0	0	0	M	M	M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
/	0	0	0	0	M	M	M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(	0	0	0	0	0	0	0	0	0	M	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
)	0	0	0	0	0	0	0	0	0	0	M	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[Number]	0	0	0	0	0	0	0	0	0	0	0	M	0	0	0	0	0	0	0	0	0	0	0	0	0
if	0	0	0	0	0	0	0	0	0	0	0	0	M	0	0	0	0	0	0	0	0	0	0	0	0
then	0	0	0	0	0	0	0	0	0	0	0	0	0	M	M	0	0	0	0	0	0	0	0	0	0
else	0	0	0	0	0	0	0	0	0	0	0	0	0	M	M	0	0	0	0	0	0	0	0	0	0
{	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	M	0	0	0	0	0	0	0	0	0
}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	M	M	0	0	0	0	0	0	0
or	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	M	0	0	0	0	0	0	0
and	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	M	M	0	0	0	0	0	0
=	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	M	M	0	0	0	0	0	0
<	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	M	M	0	0	0	0	0	0
while	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	M	0	0	0
do	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	M	0	0
print	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	M	0
read	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	M

FIGURE 15 – LL(1) action table

5 Parser

We implemented the parser by writing a parser generator which generate it from the action table that we computed previously. To do so, we implemented the function `parse` in Parser class using the algorithm in the syllabus and rendering it recursive.

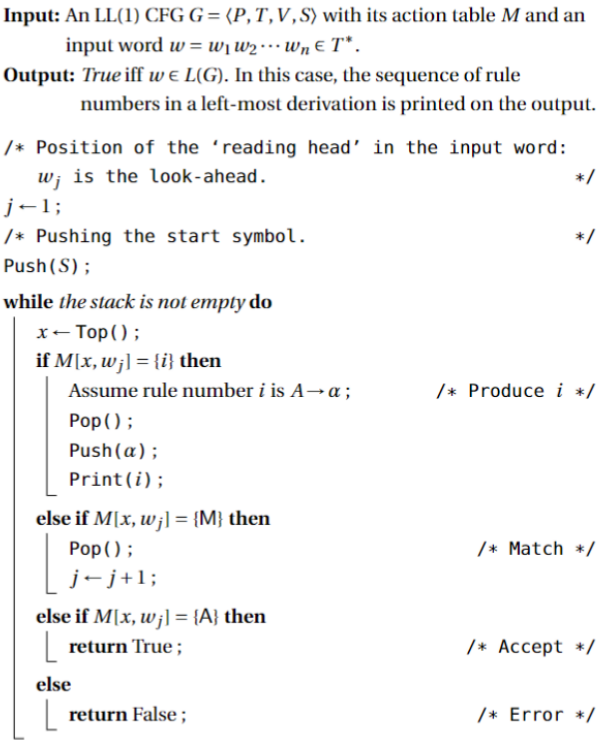


FIGURE 16 – Algorithm of the parser

5.1 Parsing algorithm

The parsing algorithm in Figure 16, follows a specific sequence : initially, the start symbol is placed onto the stack. Subsequently, the algorithm consistently reads the symbol 'x' from the stack's top, where 'x' may represent either a variable or a terminal. Simultaneously, it assesses the lookahead symbol 'a' and executes the action specified by the action table in cell M[x, a], then the symbol 'x' is replaced in the stack by its derivation. Whenever a match occurs between the terminal on the stack's top and the lookahead symbol, the corresponding rule number is printed. This process continues for all matching states.

It proceeds this way until the 'Accept' action is executed, or until an error is encountered (which is the case when the cell is empty). But our implementation doesn't have a 'Accept' state since no final terminal exist (end cannot be the one since we can have multiple ones), so we just stop at the end of the input string reading or when the stack is empty. At the end, the program output on stdout the leftmost derivation of the input if and only if it is correct meaning that the input respect the grammar.

When the input is not correct, the parse function detects an error and throw an explanotary message : "Unexpected symbol "symbol" encountered. Expected symbols : (+ - *". The unexpected symbol is the look-ahead and the expected symbols are coming from the First set of x (top of the stack).

```
1
2
3   '' Euclid's algorithm ''
4   begin
5     while 0 < b do
6       begin
7         c := b...
8         while b < a+1 do     ** computation of modulo
9           a := a-b...
10        b := a...
11        a := c...
12      end...
13    print(a)
14  end
15
Error, unexpected symbol encountered: end
At line and column : 12:4
```

FIGURE 17 – Example

6 Example files

For this project, it was required to test and provide our own example files, ensuring their relevance. To do so, we identified different scenarios in which our parser implementation can be tested :

6.1 Check that a transformed grammar is LL(1)

We tested for different grammars that we know they are LL(1) or not LL(1) (taken from syllabus or proved during exercise sessions) that our function `isGrammarLLK` is correct :

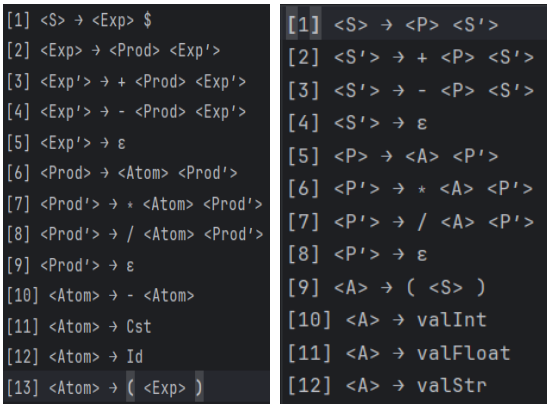


FIGURE 18 – Known LL1 grammars

They are both LL1 according to our program which is correct. We do the same with a different grammar that we know it is not LL(1) and the program throw an error telling that it is not LL1. These grammars can be found in the directory test/resources/grammars.

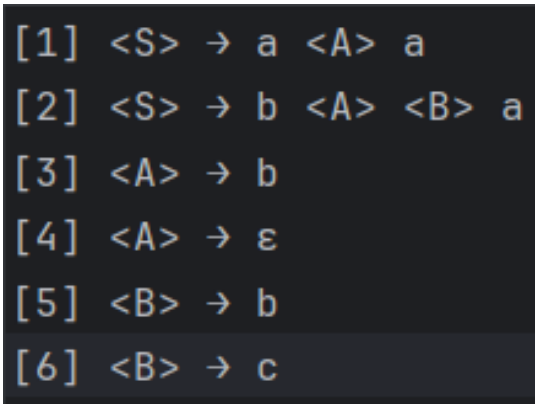


FIGURE 19 – Known non LL1 grammars

6.2 Construct action table for known transformed grammar

For two different grammars that we know they are LL(1), we tested if the constructed action table using our function `constructLL1ActionTable` is correct. They can be found in test/resources/action_table.

	$	+	-	*	/	Cst	Id	(	)
<S>	0	0	1	0	0	1	1	1	0
<Exp>	0	0	2	0	0	2	2	2	0
<Exp'>	5	3	4	0	0	0	0	0	5
<Prod>	0	0	6	0	0	6	6	6	0
<Prod'>	9	9	9	7	8	0	0	0	9
<Atom>	0	0	10	0	0	11	12	13	0
$	M	0	0	0	0	0	0	0	0
+	0	M	0	0	0	0	0	0	0
-	0	0	M	0	0	0	0	0	0
*	0	0	0	M	0	0	0	0	0
/	0	0	0	0	M	0	0	0	0
Cst	0	0	0	0	0	M	0	0	0
Id	0	0	0	0	0	0	M	0	0
(	0	0	0	0	0	0	0	M	0
)	0	0	0	0	0	0	0	0	M

	+	-	*	/	(	)	valInt	valFloat	valStr
<S>	0	0	0	0	1	0	1	1	1
<S'>	2	3	0	0	0	4	0	0	0
<P>	0	0	0	0	5	0	5	5	5
<P'>	8	8	6	7	0	8	0	0	0
<A>	0	0	0	0	9	0	10	11	12
+	M	0	0	0	0	0	0	0	0
-	0	M	0	0	0	0	0	0	0
*	0	0	M	0	0	0	0	0	0
/	0	0	0	M	0	0	0	0	0
(	0	0	0	0	M	0	0	0	0
)	0	0	0	0	0	M	0	0	0
valInt	0	0	0	0	0	0	M	0	0
valFloat	0	0	0	0	0	0	0	M	0
valStr	0	0	0	0	0	0	0	0	M

FIGURE 20 – Action table constructed for LL1 grammars

6.3 Parsing input and build parse tree

We check if our parser works and the output of left most derivation of the input is correct :

```
'' Euclid's algorithm ''
begin
  while 0 < b do
    begin
      c := b...
      while b < a+1 do     ** computation of modulo
        a := a-b...
      b := a...
      a := c
    end...
  print(a)
end
```

```
1 2 4 9 39 29 33 36 14 18 25 21 17 38 14 18 24 21 17 35 32 12 4
7 13 14 18 24 21 17 5 4 9 39 29 33 36 14 18 24 21 17 38 14 18 24
21 15 18 25 21 17 35 32 7 13 14 18 24 21 16 18 24 21 17 5 4 7 13
14 18 24 21 17 5 4 7 13 14 18 24 21 17 6 5 4 10 40 6
```

```
'' Factorial ''
begin
  read(n)...
  fact := 1...
  while n < 0 do ** condition was changed for scanner, it is normally n > 0
    begin
      fact := fact * n...
      n := n - 1
    end...
  print(fact)   ** Print the calculated factorial
end
```

```
1 2 4 11 41 5 4 7 13 14 18 25 21 17 5 4 9 39 29 33 36 14 18 24 21
17 38 14 18 25 21 17 35 32 12 4 7 13 14 18 24 19 24 21 17 5 4 7 13
14 18 24 21 16 18 25 21 17 6 5 4 10 40 6
```

FIGURE 21 – Examples of inputs for the parser and the left most derivation

These inputs are available in test/resources/parser/input The parse trees for theses inputs can be found in test/resources/parser/parse_tree

6.3.1 Finish before the file / Accepting state

A problem that we encounter when we first contructed the action table of the transform is the parser finished the parsing of the input before the end of the file. It was because the terminal "end" was considered as an accepting state. We fixed this problem by considering the terminal "end" as a matching state and the input will be correct for the parser if, after reading it, the stack is empty meaning the input was parsed completely.

6.3.2 Explainatory message / Error

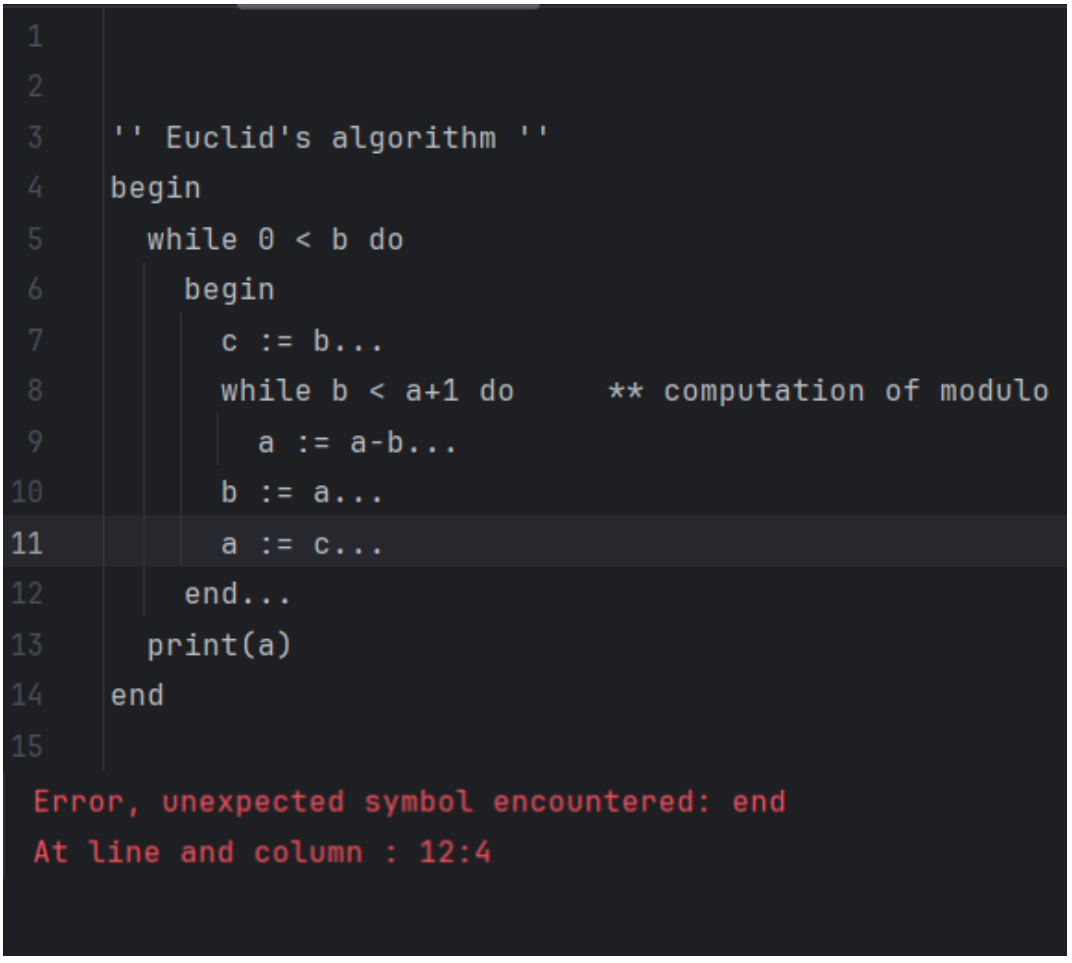


FIGURE 22 – Example when there is an error