# INFO-F-403
# Introduction to language theory and compiling
# -
# ALGOL-0 Part 2

Antoine de Selys - Arthur Pierrot

000443288 - 000422751

Teachers Gilles GEERAERTS - Raphaël BERTHON - Sarah WINTER

Academic Year 2019-2020

November 2019

## Contents

## 1 Statement

In this second part of the project we had different purpose concerning our ALGOL-0 compiler. First, we had to transform the ALGOL-0 grammar given in the statement. In particular, the transformation that consists in modifying the grammar to take into account priority and associativity of operators allows one to remove (some) ambiguities of grammars. In our case with the grammar given, we had to :

- Remove unproductive and/or unreachable variables

- Make the grammar non-ambiguous by taking into account the priority and the associativity of the operators

- Remove left-recursion and apply factorisation where needed

In a second time, we had to check if our grammar is *LL(1)* and write the *action table* of an *LL(1)* parser for the transformed grammar. And last but not least we had to write a parser in Java for this grammar.

## 2 Introduction

As a reminder parsing is the second step of the compiling process. During this stage, the compiler analyses the syntax of the input program to check its correctness. In our project, we will talk about two families of parsers, namely the *top-down* and the *bottom-up* parsers. As their names indicate, these parsers work in two completely opposite ways: a *top-down* parser tries and build a derivation tree for the input string starting from the root, applying the grammar rules one after the other, until the sequence of tree leaves forms the desired string.
On the other hand, a *bottom-up* parser builds a derivation tree starting from the leaves (i.e.,the input string), follow backwards the derivation rules of the grammar, until it manages to reach the root of the tree. We will see that these two paradigms have their pros and cons. Top-down parsers are perhaps more intuitive, but bottom-up parsers are more powerful in term of complexity [1].

## 3 Choices and Hypotheses

### 3.1 Grammar transformations

#### 3.1.1 Removing useless symbols

Here below is the grammar given in our statement ; it doesn't contain any unproductive or unreachable variables. In this case, we had to pay attention to the fact that removing unproductive symbols can create unreachable ones, but that removing unreachable symbols will not make variables unproductive. So, to remove all useless symbols from a grammar we have to :
- remove unproductive variables
- remove unreachable symbols

After that, all variables are guaranteed to be productive, and all symbols to be reachable [1].

| | | | |
|---|---|---|---|
| [1] | <Program> | $\longrightarrow$ | begin <Code> end |
| [2] | <Code> | $\longrightarrow$ | $\epsilon$ |
| [3] | | $\longrightarrow$ | <InstList> |
| [4] | <InstList> | $\longrightarrow$ | <Instruction> |
| [5] | | $\longrightarrow$ | <Instruction> ; <InstList> |
| [6] | <Instruction> | $\longrightarrow$ | <Assign> |
| [7] | | $\longrightarrow$ | <If> |
| [8] | | $\longrightarrow$ | <While> |
| [9] | | $\longrightarrow$ | <For> |
| [10] | | $\longrightarrow$ | <Print> |
| [11] | | $\longrightarrow$ | <Read> |
| [12] | <Assign> | $\longrightarrow$ | [VarName] := <ExprArith> |
| [13] | <ExprArith> | $\longrightarrow$ | [VarName] |
| [14] | | $\longrightarrow$ | [Number] |
| [15] | | $\longrightarrow$ | ( <ExprArith> ) |
| [16] | | $\longrightarrow$ | - <ExprArith> |
| [17] | | $\longrightarrow$ | <ExprArith> <Op> <ExprArith> |
| [18] | <Op> | $\longrightarrow$ | + |
| [19] | | $\longrightarrow$ | - |
| [20] | | $\longrightarrow$ | * |
| [21] | | $\longrightarrow$ | / |
| [22] | <If> | $\longrightarrow$ | if <Cond> then <Code> endif |
| [23] | | $\longrightarrow$ | if <Cond> then <Code> else <Code> endif |
| [24] | <Cond> | $\longrightarrow$ | <Cond> <BinOp> <Cond> |

| | | | |
|---|---|---|---|
| [25] | | ⟶ | not <SimpleCond> |
| [26] | | ⟶ | <SimpleCond> |
| [27] | <SimpleCond> | ⟶ | <ExprArith> <Comp> <ExprArith> |
| [28] | <BinOp> | ⟶ | and |
| [29] | | ⟶ | or |
| [30] | <Comp> | ⟶ | = |
| [31] | | ⟶ | >= |
| [32] | | ⟶ | > |
| [33] | | ⟶ | <= |
| [34] | | ⟶ | < |
| [35] | | ⟶ | /= |
| [36] | <While> | ⟶ | while <Cond> do <Code> endwhile |
| [37] | <For> | ⟶ | for [VarName] from <ExprArith> by <ExprArith> to <ExprArith> do <Code> endwhile |
| [38] | <Print> | ⟶ | print([VarName]) |
| [39] | <Read> | ⟶ | read([VarName]) |

Table 1: Original grammar

### 3.1.2 Priority and associativity of operators

This is a technique that allows to remove ambiguities occurring typically in grammars designed for arithmetic or Boolean expressions. We had to remove ambiguities in the grammar by taking into account the priority and associativity of operators [1].

The following rules were removed.

| | | |
|---|---|---|
| <ExprArith> | ⟶ | [VarName] |
| | ⟶ | [Number] |
| | ⟶ | ( <ExprArith> ) |
| | ⟶ | - <ExprArith> |
| | ⟶ | <ExprArith> <Op> <ExprArith> |
| <Op> | ⟶ | + |
| | ⟶ | - |
| | ⟶ | * |
| | ⟶ | / |

Then we replaced them by those.

| | | |
|---|---|---|
| <ExprArith> | $\longrightarrow$ | <ExprArith> + <Prod> |
| | $\longrightarrow$ | <ExprArith> - <Prod> |
| | $\longrightarrow$ | <Prod> |
| <Prod> | $\longrightarrow$ | <Prod> * <Atom> |
| | $\longrightarrow$ | <Prod> / <Atom> |
| | $\longrightarrow$ | <Atom> |
| <Atom> | $\longrightarrow$ | - <Atom> |
| | $\longrightarrow$ | [Number] |
| | $\longrightarrow$ | [VarName] |
| | $\longrightarrow$ | ( <ExprArith> ) |

We did the same for the rules below.

| | | |
|---|---|---|
| <Cond> | $\longrightarrow$ | <Cond> <BinOp> <Cond> |
| | $\longrightarrow$ | not <SimpleCond> |
| | $\longrightarrow$ | <SimpleCond> |
| <SimpleCond> | $\longrightarrow$ | <ExprArith> <Comp> <ExprArith> |
| <BinOp> | $\longrightarrow$ | and |
| | $\longrightarrow$ | or |

| | | |
|---|---|---|
| <Cond> | $\longrightarrow$ | <Cond> or <CondAnd> |
| | $\longrightarrow$ | <CondAnd> |
| <CondAnd> | $\longrightarrow$ | <CondAnd> and <SimpleCond> |
| | $\longrightarrow$ | <SimpleCond> |
| <SimpleCond> | $\longrightarrow$ | <ExprArith> <Comp> <ExprArith> |

| | |
|---|---|
| $\longrightarrow$ | not <SimpleCond> |

### 3.1.3 Removing left-recursion

Now that we have added priority and associativity into the grammar, we need to remove the left-recursion created during the process. So, our technique to remove direct and indirect left-recursion will be as follows:
- First, transform indirect left-recursion into direct left-recursion;
- Then, transform left-recursion into right-recursion.

As a result we successfully removed the left-recursion [1].

$$\begin{array}{lcl}
\langle\text{ExprArith}\rangle & \longrightarrow & \langle\text{ExprArith}\rangle + \langle\text{Prod}\rangle \\
 & \longrightarrow & \langle\text{ExprArith}\rangle \text{ - } \langle\text{Prod}\rangle \\
 & \longrightarrow & \langle\text{Prod}\rangle \\
\langle\text{Prod}\rangle & \longrightarrow & \langle\text{Prod}\rangle * \langle\text{Atom}\rangle \\
 & \longrightarrow & \langle\text{Prod}\rangle \ / \ \langle\text{Atom}\rangle \\
 & \longrightarrow & \langle\text{Atom}\rangle
\end{array}$$

$$\begin{array}{lcl}
\langle\text{ExprArith}\rangle & \longrightarrow & \langle\text{Prod}\rangle \ \langle\text{ExprArith'}\rangle \\
\langle\text{ExprArith'}\rangle & \longrightarrow & + \ \langle\text{Prod}\rangle \ \langle\text{ExprArith'}\rangle \\
 & \longrightarrow & \text{- } \langle\text{Prod}\rangle \ \langle\text{ExprArith'}\rangle \\
 & \longrightarrow & \epsilon \\
\langle\text{Prod}\rangle & \longrightarrow & \langle\text{Atom}\rangle \ \langle\text{Prod'}\rangle \\
\langle\text{Prod'}\rangle & \longrightarrow & * \ \langle\text{Atom}\rangle \ \langle\text{Prod'}\rangle \\
 & \longrightarrow & / \ \langle\text{Atom}\rangle \ \langle\text{Prod'}\rangle \\
 & \longrightarrow & \epsilon
\end{array}$$

$$\begin{array}{lcl}
\langle\text{Cond}\rangle & \longrightarrow & \langle\text{Cond}\rangle \text{ or } \langle\text{CondAnd}\rangle \\
 & \longrightarrow & \langle\text{CondAnd}\rangle \\
\langle\text{CondAnd}\rangle & \longrightarrow & \langle\text{CondAnd}\rangle \text{ and } \langle\text{SimpleCond}\rangle \\
 & \longrightarrow & \langle\text{SimpleCond}\rangle
\end{array}$$

$$\begin{array}{lcl}
\langle\text{Cond}\rangle & \longrightarrow & \langle\text{CondAnd}\rangle \ \langle\text{Cond'}\rangle \\
\langle\text{Cond'}\rangle & \longrightarrow & \text{or } \langle\text{CondAnd}\rangle \ \langle\text{Cond'}\rangle \\
 & \longrightarrow & \epsilon \\
\langle\text{CondAnd}\rangle & \longrightarrow & \langle\text{SimpleCond}\rangle \ \langle\text{CondAnd'}\rangle \\
\langle\text{CondAnd'}\rangle & \longrightarrow & \text{and } \langle\text{SimpleCond}\rangle \ \langle\text{CondAnd'}\rangle \\
 & \longrightarrow & \epsilon
\end{array}$$

### 3.1.4 Factoring

The only step for finishing the grammar is the factoring. It can be applied when a grammar contains at least two rules with the same left-hand side, and a common prefix in the right-hand side. This step gives us a non-ambiguous grammar [1].

$$\begin{array}{lll} \text{<InstList>} & \longrightarrow & \text{<Instruction>} \\ & \longrightarrow & \text{<Instruction> ; <InstList>} \end{array}$$

$$\begin{array}{lll} \text{<InstList>} & \longrightarrow & \text{<Instruction> <NextInst>} \\ \text{<NextInst>} & \longrightarrow & \epsilon \\ & \longrightarrow & \text{; <InstList>} \end{array}$$

$$\begin{array}{lll} \text{<If>} & \longrightarrow & \text{if <Cond> then <Code> endif} \\ & \longrightarrow & \text{if <Cond> then <Code> else <Code> endif} \end{array}$$

$$\begin{array}{lll} \text{<If>} & \longrightarrow & \text{if <Cond> then <Code> <IfSeq>} \\ \text{<IfSeq>} & \longrightarrow & \text{endif} \\ & \longrightarrow & \text{else <Code> endif} \end{array}$$

### 3.1.5 Grammar

The grammar is finally complete and is shown below.

| | | | |
|---|---|---|---|
| [1] | <Program> | $\longrightarrow$ | begin <Code> end |
| [2] | <Code> | $\longrightarrow$ | $\epsilon$ |
| [3] | | $\longrightarrow$ | <InstList> |
| [4] | <InstList> | $\longrightarrow$ | <Instruction> <NextInst> |
| [5] | <NextInst> | $\longrightarrow$ | $\epsilon$ |
| [6] | | $\longrightarrow$ | ; <InstList> |
| [7] | <Instruction> | $\longrightarrow$ | <Assign> |
| [8] | | $\longrightarrow$ | <If> |
| [9] | | $\longrightarrow$ | <While> |
| [10] | | $\longrightarrow$ | <For> |
| [11] | | $\longrightarrow$ | <Print> |
| [12] | | $\longrightarrow$ | <Read> |
| [13] | <Assign> | $\longrightarrow$ | [VarName] := <ExprArith> |
| [14] | <ExprArith> | $\longrightarrow$ | <Prod> <ExprArith'> |
| [15] | <ExprArith'> | $\longrightarrow$ | + <Prod> <ExprArith'> |
| [16] | | $\longrightarrow$ | - <Prod> <ExprArith'> |
| [17] | | $\longrightarrow$ | $\epsilon$ |
| [18] | <Prod> | $\longrightarrow$ | <Atom> <Prod'> |
| [19] | <Prod'> | $\longrightarrow$ | * <Atom> <Prod'> |
| [20] | | $\longrightarrow$ | / <Atom> <Prod'> |
| [21] | | $\longrightarrow$ | $\epsilon$ |
| [22] | <Atom> | $\longrightarrow$ | - <Atom> |
| [23] | | $\longrightarrow$ | [Number] |
| [24] | | $\longrightarrow$ | [VarName] |
| [25] | | $\longrightarrow$ | ( <ExprArith> ) |
| [26] | <If> | $\longrightarrow$ | if <Cond> then <Code> <IfSeq> |
| [27] | <IfSeq> | $\longrightarrow$ | endif |
| [28] | | $\longrightarrow$ | else <Code> endif |
| [29] | <Cond> | $\longrightarrow$ | <CondAnd> <Cond'> |
| [30] | <Cond'> | $\longrightarrow$ | or <CondAnd> <Cond'> |
| [31] | | $\longrightarrow$ | $\epsilon$ |
| [32] | <CondAnd> | $\longrightarrow$ | <SimpleCond> <CondAnd'> |
| [33] | <CondAnd'> | $\longrightarrow$ | and <SimpleCond> <CondAnd'> |
| [34] | | $\longrightarrow$ | $\epsilon$ |
| [35] | <SimpleCond> | $\longrightarrow$ | <ExprArith> <Comp> <ExprArith> |
| [36] | | $\longrightarrow$ | not <SimpleCond> |
| [37] | <Comp> | $\longrightarrow$ | = |
| [38] | | $\longrightarrow$ | >= |
| [39] | | $\longrightarrow$ | > |
| [40] | | $\longrightarrow$ | <= |

| | | | |
|---|---|---|---|
| [41] | | $\longrightarrow$ | < |
| [42] | | $\longrightarrow$ | /= |
| [43] | <While> | $\longrightarrow$ | while <Cond> do <Code> endwhile |
| [44] | <For> | $\longrightarrow$ | for [VarName] from <ExprArith> by <ExprArith> to <ExprArith> do <Code> endwhile |
| [45] | <Print> | $\longrightarrow$ | print([VarName]) |
| [46] | <Read> | $\longrightarrow$ | read([VarName]) |

Table 2: Grammar after transformations

### 3.1.6 LL(1) grammar

We can check if the grammar is LL(1) by checking if it is Strong LL(1) because for all $k \geq 1$, for all CFG $G$: if $G$ is strong LL(k), then it is also LL(k).

A CFG $G =< V, T, P, S >$ is strong LL(k) iff, for all pairs of rules $A \rightarrow \alpha_1$ and $A \rightarrow \alpha_2$ in $P$ (with $\alpha_1 \neq \alpha_2$):

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

For each variable with more than one rule, we need to check if the property above is respected. Since $k = 1$, when $\alpha_1$ or $\alpha_2$ is different than $\epsilon$, we can ignore the $Follow(A)$.

*<Code>*

$$\begin{aligned} First(\epsilon Follow(< Code >)) &= Follow(< Code >) \\ &= \{end, endif, else, endwhile\} \\ First(< InstList >) &= \{[VarName], if, while, for, print, read\} \end{aligned}$$

*<NextInst>*

$$\begin{aligned} First(\epsilon Follow(< NextInst >)) &= Follow(< NextInst >) \\ &= \{end, endif, else, endwhile\} \\ First(; < InstList >) &= \{;\} \end{aligned}$$

*<Instruction>*

$$\begin{aligned}
First(<Assign>) &= \{[VarName]\}\\
First(<If>) &= \{if\}\\
First(<While>) &= \{while\}\\
First(<For>) &= \{for\}\\
First(<Print>) &= \{print\}\\
First(<Read>) &= \{read\}
\end{aligned}$$

*<ExprArith'>*

$$\begin{aligned}
First(+<Prod><ExprArith'>) &= \{+\}\\
First(-<Prod><ExprArith'>) &= \{-\}\\
First(\epsilon Follow(<ExprArith'>)) &= Follow(<ExprArith'>)\\
&= \{;, end, endif, else, endwhile, ), and, or,\\
&\quad then, do, by, to, =, >=, >, <=, <, /=\}
\end{aligned}$$

*<Prod'>*

$$\begin{aligned}
First(*<Atom><Prod'>) &= \{*\}\\
First(/<Atom><Prod'>) &= \{/\}\\
First(\epsilon Follow(<Prod'>)) &= Follow(<Prod'>)\\
&= \{+, -, ;, end, endif, else, endwhile, ), and,\\
&\quad or, then, do, by, to, =, >=, >, <=, <, /=\}
\end{aligned}$$

*<Atom>*

$$\begin{aligned}
First(-<Atom>) &= \{-\}\\
First([Number]) &= \{[Number]\}\\
First([VarName]) &= \{[VarName]\}\\
First((<ExprArith>)) &= \{(\}
\end{aligned}$$

*<IfSeq>*

$$\begin{aligned} First(endif) &= \{endif\} \\ First(else < Code > endif) &= \{else\} \end{aligned}$$

*<Cond'>*

$$\begin{aligned} First(or < CondAnd >< Cond' >) &= \{or\} \\ First(\epsilon Follow(< Cond' >)) &= Follow(< Cond' >) \\ &= \{then, do\} \end{aligned}$$

*<CondAnd'>*

$$\begin{aligned} First(and < SimpleCond >< CondAnd' >) &= \{and\} \\ First(\epsilon Follow(< CondAnd' >)) &= Follow(< CondAnd' >) \\ &= \{or, then, do\} \end{aligned}$$

*<SimpleCond>*

$$\begin{aligned} First(< ExprArith >< Comp >< ExprArith >) &= \{-, [Number], [VarName], (\} \\ First(not < SimpleCond >) &= \{not\} \end{aligned}$$

*<Comp>*

$$\begin{aligned} First(=) &= \{=\} \\ First(>=) &= \{>=\} \\ First(>) &= \{>\} \\ First(<=) &= \{<=\} \\ First(<) &= \{<\} \\ First(/=) &= \{/=\} \end{aligned}$$

The property is respected for all the variables with more than one rule. Thus the grammar is LL(1) and an action table can be built [1].

### 3.2 Action table

In order to compute the action table, we must find the First(X) and Follow(X) for each variable X.

#### 3.2.1 First(X)

| X | First(X) |
|---|---|
| <Program> | begin |
| <Code> | [VarName], if, while, for, print, read, $\epsilon$ |
| <InstList> | [VarName], if, while, for, print, read |
| <NextInst> | ;, $\epsilon$ |
| <Instruction> | [VarName], if, while, for, print, read |
| <Assign> | [VarName] |
| <ExprArith> | -, [Number], [VarName], ( |
| <ExprArith'> | +, -, $\epsilon$ |
| <Prod> | -, [Number], [VarName], ( |
| <Prod'> | *, /, $\epsilon$ |
| <Atom> | -, [Number], [VarName], ( |
| <If> | if |
| <IfSeq> | endif, else |
| <Cond> | not, -, [Number], [VarName], ( |
| <Cond'> | or, $\epsilon$ |
| <CondAnd> | not, -, [Number], [VarName], ( |
| <CondAnd'> | and, $\epsilon$ |
| <SimpleCond> | not, -, [Number], [VarName], ( |
| <Comp> | =, >=, >, <=, <, /= |
| <While> | while |
| <For> | for |
| <Print> | print |
| <Read> | read |

### 3.2.2 Follow(X)

| X | Follow(X) |
|---|---|
| <Program> | $\epsilon$ |
| <Code> | end, endif, else, endwhile |
| <InstList> | end, endif, else, endwhile |
| <NextInst> | end, endif, else, endwhile |
| <Instruction> | ;, end, endif, else, endwhile |
| <Assign> | ;, end, endif, else, endwhile |
| <ExprArith> | ;, end, endif, else, endwhile, ), and, or, then, do, by, to, =, >=, >, <=, <, /= |
| <ExprArith'> | ;, end, endif, else, endwhile, ), and, or, then, do, by, to, =, >=, >, <=, <, /= |
| <Prod> | +, -, ;, end, endif, else, endwhile, ), and, or, then, do, by, to, =, >=, >, <=, <, /= |
| <Prod'> | +, -, ;, end, endif, else, endwhile, ), and, or, then, do, by, to, =, >=, >, <=, <, /= |
| <Atom> | *, /, +, -, ;, end, endif, else, endwhile, ), and, or, then, do, by, to, =, >=, >, <=, <, /= |
| <If> | ;, end, endif, else, endwhile |
| <IfSeq> | ;, end, endif, else, endwhile |
| <Cond> | then, do |
| <Cond'> | then, do |
| <CondAnd> | or, then, do |
| <CondAnd'> | or, then, do |
| <SimpleCond> | and, or, then, do |
| <Comp> | -, [Number], [VarName], ( |
| <While> | ;, end, endif, else, endwhile |
| <For> | ;, end, endif, else, endwhile |
| <Print> | ;, end, endif, else, endwhile |
| <Read> | ;, end, endif, else, endwhile |

### 3.2.3 Action table

With the First(X) and Follow(X) table we can find the action table below cut in two (lines with no values where removed) [1].

| | begin | end | ; | := | [VarName] | [Number] | + | - | * | / | ( | ) | if | then | endif | else | or |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| <Program> | 1 | | | | | | | | | | | | | | | | |
| <Code> | | 2 | | | 3 | | | | | | | | 3 | | 2 | 2 | |
| <InstList> | | | | | 4 | | | | | | | | 4 | | | | |
| <NextInst> | | 5 | 6 | | | | | | | | | | | | 5 | 5 | |
| <Instruction> | | | | | 7 | | | | | | | | 8 | | | | |
| <Assign> | | | | | 13 | | | | | | | | | | | | |
| <ExprArith> | | | | | 14 | 14 | | 14 | | | 14 | | | 17 | 17 | 17 | 17 |
| <ExprArith'> | | 17 | 17 | | | | 15 | 16 | | | | | | | | | |
| <Prod> | | | | | 18 | 18 | | 18 | | | 18 | | | | | | |
| <Prod'> | | 21 | 21 | | | | 21 | 21 | 19 | 20 | | 21 | | 21 | 21 | 21 | 21 |
| <Atom> | | | | | 24 | 23 | | 22 | | | 25 | | | | | | |
| <If> | | | | | | | | | | | | 26 | | | | | |
| <IfSeq> | | | | | | | | | | | | | | | 27 | 28 | |
| <Cond> | | | | | 29 | 29 | | 29 | | | 29 | | | | | | |
| <Cond'> | | | | | | | | | | | | | | 31 | | | 30 |
| <CondAnd> | | | | | 32 | 32 | | 32 | | | 32 | | | | | | |
| <CondAnd'> | | | | | | | | | | | | | | 34 | | | 34 |
| <SimpleCond> | | | | | 35 | 35 | | 35 | | | 35 | | | | | | |

| | and | not | = | >= | > | <= | < | /= | while | do | endwhile | for | from | by | to | print | read |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| <Code> | | | | | | | | | 3 | | 2 | 3 | | | | 3 | 3 |
| <InstList> | | | | | | | | | 4 | | | 4 | | | | 4 | 4 |
| <NextInst> | | | | | | | | | | | 5 | | | | | | |
| <Instruction> | | | | | | | | | 9 | | | 10 | | | | 11 | 12 |
| <ExprArith'> | 17 | | 17 | 17 | 17 | 17 | 17 | 17 | | 17 | 17 | | | 17 | 17 | | |
| <Prod'> | 21 | | 21 | 21 | 21 | 21 | 21 | 21 | | 21 | 21 | | | 21 | 21 | | |
| <Cond> | | 29 | | | | | | | | | | | | | | | |
| <Cond'> | | | | | | | | | | 31 | | | | | | | |
| <CondAnd> | | 32 | | | | | | | | | | | | | | | |
| <CondAnd'> | 33 | | | | | | | | | 34 | | | | | | | |
| <SimpleCond> | | 36 | | | | | | | | | | | | | | | |
| <Comp> | | | 37 | 38 | 39 | 40 | 41 | 42 | | | | | | | | | |
| <While> | | | | | | | | | 43 | | | | | | | | |
| <For> | | | | | | | | | | | | 44 | | | | | |
| <Print> | | | | | | | | | | | | | | | | 45 | |
| <Read> | | | | | | | | | | | | | | | | | 46 |

### 3.3 Parsers

For the parser, we made a recursive descent LL(1) parser by hand using the action table. The derivation tree created from the file *euclide.alg* can be seen in the figure 1. We can recover the whole code by reading the leaves from left to right. The output generated for the same file is :
1 3 4 12 46 6 4 12 46 6 4 9 43 29 32 35 14 18 24 21 17 42 14 18 23 21 17 34 31 3 4 7 13 14 18 24 21 17 6 4 9 43 29 32 35 14 18 24 21 17 38 14 18 24 21 17 34 31 3 4 7 13 14 18 24 21 16 18 24 21 17 5 6 4 7 13 14 18 24 21 17 6 4 7 13 14 18 24 21 17 5 6 4 11 45 5

#### 3.3.1 Parse tree

We changed the class ParseTree so that It uses String instead of Symbol because non-terminal nodes have no symbol but can have a label like <*Program*>. We also added the line

```
\usepackage[T1]{fontenc}}
```

before writing the forest to be able to display < and > correctly in latex.

## 4 Tests and Results

We added 3 new files for the tests. *complex.alg* which contains a for loop and an if-else statement. *syntaxerror.alg* which contains a syntax error on line 8 and *syntaxerror2.alg* with a syntax error on line 14. The program runs correctly on those files and also on the *euclide.alg* file.

## 5 Conclusion

We now have accomplished the second step of the compilation and the program is capable of determining if the file given as parameter contains any syntax error. We first had to modify the grammar so that it would be LL(1). Then we created the action table on the basis of the First(X) and Follow(X) tables. Finally we wrote the parser using the action table.

## References

[1] Gilles Geeraerts, Guillermo A.Pérez, *Introduction to language theory and compiling*, 2019.

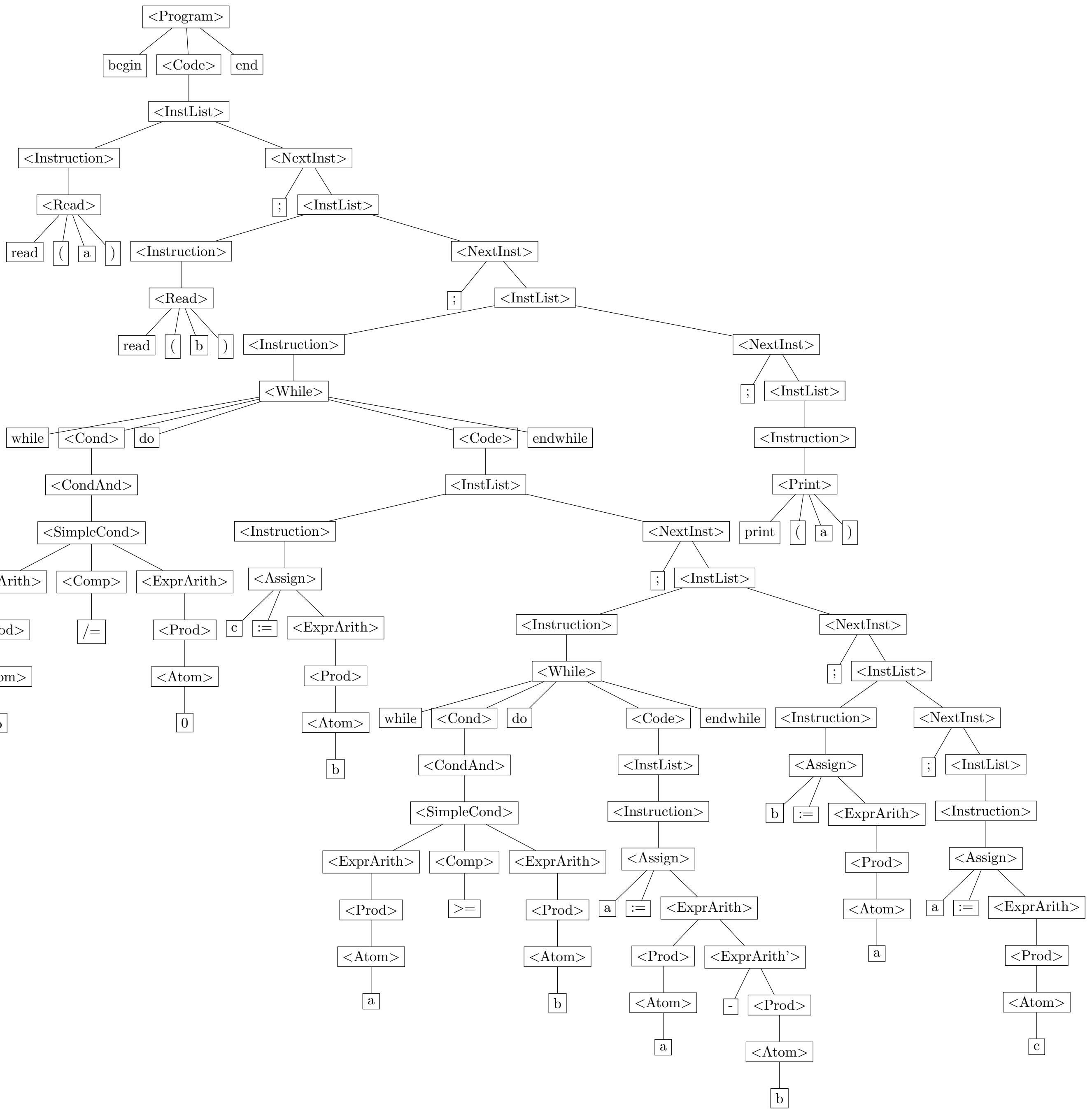

Figure 1: Derivation tree for the file *euclide.alg*