INFO-F403: Introduction to language theory and compiling
Part 2: Parser of GILLES

Loïc Blommaert (542721, M-INFOS), Hugo Charels (544051, M-INFOS)

November 24, 2024

1 Introduction

We previously discussed the design and implementation of the lexical analyzer for *GILLES*. The next step in the implementation of *GILLES* is the construction of its LL(1) parser, which will be the focus of this report. In the following sections, we will describe the modifications made to the grammar to make it LL(1), compute the action table, and detail the implementation of the parser, which was done in Java.

2 Transforming the Grammar into LL(1) Form

The raw grammar that describes GILLES is as follows:

[1]	⟨Program⟩	→ LET [ProgName] BE ⟨Code⟩ END
[2]	⟨Code⟩	→ ⟨Instruction⟩ : ⟨Code⟩
[3]		→ ε
[4]	⟨Instruction⟩	→ ⟨Assign⟩
[5]		→ ⟨If⟩
[6]		→ ⟨While⟩
[7]		→ ⟨Call⟩
[8]		→ ⟨Output⟩
[9]		→ ⟨Input⟩
[10]	⟨Assign⟩	→ [VarName]=⟨ExprArith⟩
[11]	⟨ExprArith⟩	→ [VarName]
[12]		→ [Number]
[13]		→ (⟨ExprArith⟩)
[14]		→ -⟨ExprArith⟩
[15]		→ ⟨ExprArith⟩ ⟨Op⟩ ⟨ExprArith⟩
[16]	⟨Op⟩	→ +
[17]		→ -
[18]		→ *
[19]		→ /
[20]	⟨If⟩	→ IF {⟨Cond⟩} THEN ⟨Code⟩ END
[21]		→ IF {⟨Cond⟩} THEN ⟨Code⟩ ELSE ⟨Code⟩ END
[22]	⟨Cond⟩	→ ⟨Cond⟩ -> ⟨Cond⟩
[23]		→ | ⟨Cond⟩ |
[24]		→ ⟨ExprArith⟩ ⟨Comp⟩ ⟨ExprArith⟩
[25]	⟨Comp⟩	→ ==
[26]		→ <=
[27]		→ <
[28]	⟨While⟩	→ WHILE {⟨Cond⟩} REPEAT ⟨Code⟩ END
[29]	⟨Output⟩	→ OUT([VarName])
[30]	⟨Input⟩	→ IN([VarName])

To transform the grammar into LL(1) form, we need to remove unproductive and unreachable variables, resolve ambiguities, and finally eliminate left recursion and apply factorization where necessary.

2.1 Removing Unproductive and Unreachable Variables

The first step in making the grammar `LL(1)` to design an `LL(1)` parser is to remove unproductive and unreachable variables. This can be achieved by removing rule 7 in this case. Indeed, the variable ⟨Call⟩ has no rule associated, meaning it cannot derive a terminal and is therefore unproductive. Since all other variables are reachable and productive this step is complete.

2.2 Making the Grammar Non-Ambiguous

An ambiguous grammar is one in which the same expression can be derived using multiple parse trees, leading to different interpretations. This is the case for our grammar, as illustrated by the expression `1+2*3`. Depending on how the operations are grouped, this expression can produce different results. For instance, as shown in Figure 1, evaluating the addition first yields `(1 + 2) * 3 = 7`. On the other hand, as shown in Figure 2, prioritizing multiplication results in `1 + (2 * 3) = 9`.

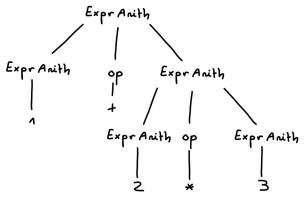

Figure 1: An execution leading to `1+2*3 = 7`

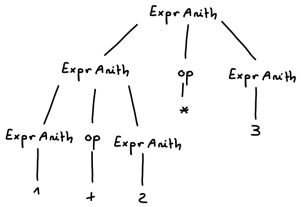

Figure 2: Another execution leading to `1+2*3 = 9`

This ambiguity arises because the grammar does not explicitly encode operator precedence or associativity. To resolve this, it is necessary to rewrite the grammar to enforce the operators precedence and to define the left-associativity of these operators explicitly. By doing so, we can ensure that all expressions are parsed in a consistent and unambiguous manner. Note that the precedence of operators are described in table 1.

Operators	Associativity
- (unary)	right
*, /	left
+, - (binary)	left
==, <=, <	left
->	right

Table 1: Operators precedence of `GILLES` (decreasing order)

Since the first result of evaluation of expressions comes from the bottom of the tree, it is necessary to introduce intermediate states to represent the hierarchy of operators. This restructuring ensures that higher-priority operators are evaluated first, as dictated by the tree's depth. By increasing the depth of the tree through additional intermediate states, we can correctly enforce operator precedence and associativity. To achieve this, we applied the following transformations:

		⋮				⋮
[10]	⟨ExprArith⟩	→ [VarName]		[10]	⟨ExprArith⟩	→ ⟨ExprArith⟩+⟨ProdArith⟩
[11]		→ [Number]		[11]		→ ⟨ExprArith⟩-⟨ProdArith⟩
[12]		→ (⟨ExprArith⟩)		[12]		→ ⟨ProdArith⟩
[13]		→ -⟨ExprArith⟩		[13]	⟨ProdArith⟩	→ ⟨ProdArith⟩*⟨Atom⟩
[14]		→ ⟨ExprArith⟩ ⟨Op⟩ ⟨ExprArith⟩		[14]		→ ⟨ProdArith⟩/⟨Atom⟩
[15]	⟨Op⟩	→ +		[15]		→ ⟨Atom⟩
[16]		→ -		[16]	⟨Atom⟩	→ [VarName]
[17]		→ *		[17]		→ [Number]
[18]		→ /	→	[18]		→ (⟨ExprArith⟩)
		⋮		[19]		→ -⟨Atom⟩
[21]	⟨Cond⟩	→ ⟨Cond⟩ -> ⟨Cond⟩				⋮
[22]		→ | ⟨Cond⟩ |		[22]	⟨Cond⟩	→ ⟨CondSimple⟩->⟨Cond⟩
[23]		→ ⟨ExprArith⟩ ⟨Comp⟩ ⟨ExprArith⟩		[23]		→ ⟨CondSimple⟩
[24]	⟨Comp⟩	→ ==		[24]	⟨CondSimple⟩	→ | ⟨Cond⟩ |
[25]		→ <=		[25]		→ ⟨ExprArith⟩ ⟨Comp⟩ ⟨ExprArith⟩
[26]		→ <		[26]	⟨Comp⟩	→ ==
		⋮		[27]		→ <=
				[28]		→ <
						⋮

At this stage, the grammar has been transformed into the following form:

[1]	⟨Program⟩	→ LET [ProgName] BE ⟨Code⟩ END
[2]	⟨Code⟩	→ ⟨Instruction⟩ : ⟨Code⟩
[3]		→ ε
[4]	⟨Instruction⟩	→ ⟨Assign⟩
[5]		→ ⟨If⟩
[6]		→ ⟨While⟩
[7]		→ ⟨Output⟩
[8]		→ ⟨Input⟩
[9]	⟨Assign⟩	→ [VarName]=⟨ExprArith⟩
[10]	⟨ExprArith⟩	→ ⟨ExprArith⟩+⟨ProdArith⟩
[11]		→ ⟨ExprArith⟩-⟨ProdArith⟩
[12]		→ ⟨ProdArith⟩
[13]	⟨ProdArith⟩	→ ⟨ProdArith⟩*⟨Atom⟩
[14]		→ ⟨ProdArith⟩/⟨Atom⟩
[15]		→ ⟨Atom⟩
[16]	⟨Atom⟩	→ [VarName]
[17]		→ [Number]
[18]		→ (⟨ExprArith⟩)
[19]		→ -⟨Atom⟩
[20]	⟨If⟩	→ IF {⟨Cond⟩} THEN ⟨Code⟩ ⟨END⟩
[21]		→ IF {⟨Cond⟩} THEN ⟨Code⟩ ELSE ⟨Code⟩ END
[22]	⟨Cond⟩	→ ⟨CondSimple⟩->⟨Cond⟩
[23]		→ ⟨CondSimple⟩
[24]	⟨CondSimple⟩	→ | ⟨Cond⟩ |
[25]		→ ⟨ExprArith⟩ ⟨Comp⟩ ⟨ExprArith⟩
[26]	⟨Comp⟩	→ ==
[27]		→ <=
[28]		→ <
[29]	⟨While⟩	→ WHILE {⟨Cond⟩} REPEAT ⟨Code⟩ END
[30]	⟨Output⟩	→ OUT([VarName])
[31]	⟨Input⟩	→ IN([VarName])

2.3 Remove Left Recursion and Factorization

Left recursion occurs when a non-terminal rewrites the same non-terminal in the head position. Since we only have one character of look-ahead, left recursions leads to infinite loops in the execution of the parser, explaining why it must eliminated. Factorization is also necessary, as we have only one character to decide which rule to apply. Otherwise, we would be unable to choose which rule to apply.

As we can see, ⟨ExprArith⟩ and ⟨ProdArith⟩ are subject to left recursion while ⟨If⟩ and ⟨Cond⟩ require factorization, as they begin with the same terminals. To eliminate left recursion, we can rewrite the rule of the form

$$\begin{array}{lll} \langle A\rangle & \to \langle A\rangle\alpha \\ & \to \beta \end{array} \qquad \text{in} \qquad \begin{array}{lll} \langle A\rangle & \to \beta\langle A'\rangle \\ \langle A'\rangle & \to \alpha\langle A'\rangle \\ & \to \varepsilon \end{array}$$

Factorization can be achieved by applying the following transformation:

$$\begin{array}{lll} \langle A\rangle & \to \pi\Delta_1 \\ & \to \pi\Delta_2 \\ & \dots \\ & \to \pi\Delta_n \end{array} \qquad \to \qquad \begin{array}{lll} \langle A\rangle & \to \pi \\ \langle A'\rangle & \to \Delta_1 \\ & \to \Delta_2 \\ & \dots \\ & \to \Delta_n \end{array}$$

where π is the largest prefix (either terminal or non-terminal) and Δ_1, Δ_2 represent the rest of the expression. Therefore, we applied the following transformations to our grammar :

		⋮				⋮
[10]	⟨ExprArith⟩	→ ⟨ExprArith⟩`+`⟨ProdArith⟩		[10]	⟨ExprArith⟩	→ ⟨ProdArith⟩⟨ExprArith′⟩
[11]		→ ⟨ExprArith⟩`-`⟨ProdArith⟩		[11]	⟨ExprArith′⟩	→ `+`⟨ProdArith⟩⟨ExprArith′⟩
[12]		→ ⟨ProdArith⟩		[12]		→ `-`⟨ProdArith⟩⟨ExprArith′⟩
[13]	⟨ProdArith⟩	→ ⟨ProdArith⟩`*`⟨Atom⟩		[13]		→ ε
[14]		→ ⟨ProdArith⟩`/`⟨Atom⟩		[14]	⟨ProdArith⟩	→ ⟨Atom⟩⟨ProdArith′⟩
[15]		→ ⟨Atom⟩		[15]	⟨ProdArith′⟩	→ `*`⟨Atom⟩⟨ProdArith′⟩
[16]	⟨Atom⟩	→ [`VarName`]		[16]		→ `/`⟨Atom⟩⟨ProdArith′⟩
[17]		→ [`Number`]		[17]		→ ε
[18]		→ `(`⟨ExprArith⟩`)`	→	[18]	⟨Atom⟩	→ [`Number`]
[19]		→ `-`⟨Atom⟩		[19]		→ [`VarName`]
				[20]		→ `(`⟨ExprArith⟩`)`
[20]	⟨If⟩	→ `IF` {⟨Cond⟩} `THEN` ⟨Code⟩ `END`		[21]		→ `-`⟨Atom⟩
[21]		→ `IF` {⟨Cond⟩} `THEN` ⟨Code⟩ `ELSE` ⟨Code⟩ `END`				
				[22]	⟨If⟩	→ `IF` {⟨Cond⟩} `THEN` ⟨Code⟩ ⟨IfSeq⟩
[22]	⟨Cond⟩	→ ⟨CondSimple⟩`->`⟨Cond⟩		[23]	⟨IfSeq⟩	→ `END`
[23]		→ ⟨CondSimple⟩		[24]		→ `ELSE` ⟨Code⟩ `END`
		⋮				
				[25]	⟨Cond⟩	→ ⟨CondSimple⟩⟨NextCond⟩
				[26]	⟨NextCond⟩	→ `->`⟨CondSimple⟩⟨NextCond⟩
				[27]		→ ε
						⋮

Note that rule 26 is equivalent to ⟨NextCond⟩ → `->`⟨Cond⟩ (as visible in the new rule 25), but it has been rewritten in this form to simplify the computation of `FOLLOW`, which will be discussed in section 3.

In the end, the LL(1) grammar is :

[1]	⟨Program⟩	→ `LET` [`ProgName`] `BE` ⟨Code⟩ `END`
[2]	⟨Code⟩	→ ⟨Instruction⟩ `:` ⟨Code⟩
[3]		→ ε
[4]	⟨Instruction⟩	→ ⟨Assign⟩
[5]		→ ⟨If⟩
[6]		→ ⟨While⟩
[7]		→ ⟨Output⟩
[8]		→ ⟨Input⟩
[9]	⟨Assign⟩	→ [`VarName`]`=`⟨ExprArith⟩
[10]	⟨ExprArith⟩	→ ⟨ProdArith⟩⟨ExprArith′⟩
[11]	⟨ExprArith′⟩	→ `+`⟨ProdArith⟩⟨ExprArith′⟩
[12]		→ `-`⟨ProdArith⟩⟨ExprArith′⟩
[13]		→ ε
[14]	⟨ProdArith⟩	→ ⟨Atom⟩⟨ProdArith′⟩
[15]	⟨ProdArith′⟩	→ `*`⟨Atom⟩⟨ProdArith′⟩
[16]		→ `/`⟨Atom⟩⟨ProdArith′⟩
[17]		→ ε
[18]	⟨Atom⟩	→ [`Number`]
[19]		→ [`VarName`]
[20]		→ `(`⟨ExprArith⟩`)`
[21]		→ `-`⟨Atom⟩
[22]	⟨If⟩	→ `IF` {⟨Cond⟩} `THEN` ⟨Code⟩ ⟨IfSeq⟩
[23]	⟨IfSeq⟩	→ `END`
[24]		→ `ELSE` ⟨Code⟩ `END`
[25]	⟨Cond⟩	→ ⟨CondSimple⟩⟨NextCond⟩
[26]	⟨NextCond⟩	→ `->`⟨CondSimple⟩⟨NextCond⟩
[27]		→ ε
[28]	⟨CondSimple⟩	→ `|`⟨Cond⟩`|`
[29]		→ ⟨ExprArith⟩ ⟨Comp⟩ ⟨ExprArith⟩
[30]	⟨Comp⟩	→ `==`
[31]		→ `<=`
[32]		→ `<`
[33]	⟨While⟩	→ `WHILE` {⟨Cond⟩} `REPEAT` ⟨Code⟩ `END`
[34]	⟨Output⟩	→ `OUT`([`VarName`])
[35]	⟨Input⟩	→ `IN`([`VarName`])

Figure 3: LL(1) grammar

3 Verifying LL(1) Transformation

To ensure that we have removed all left recursions and common prefixes, we can use the definition of strong `LL(k)`: A context-free grammar is strong `LL(k)` iff, for all pairs of rules $A \to \alpha_1$ and $A \to \alpha_2$ in P (with $\alpha_1 \neq \alpha_2$):

$$\texttt{FIRST}^k\left(\alpha_1\texttt{FOLLOW}^k(A)\right) \cap \texttt{FIRST}^k\left(\alpha_2\texttt{FOLLOW}^k(A)\right) = \varnothing. \tag{3.0.1}$$

We can apply this definition since for $k = 1$, all `LL(k)` grammars are also strong `LL(k)`. Therefore, we will first compute the sets of $\texttt{FIRST}^1$ and $\texttt{FOLLOW}^1$ (abbreviated by `FIRST` and `FOLLOW`).

3.1 FIRST Sets Computation

`FIRST`(⟨A⟩) denotes a function that takes ⟨A⟩ as a parameter and returns the first terminals that could appear when the parser encounters ⟨A⟩. Therefore, it is relatively easy to compute,we simply need to follow the path of variables until reaching a terminal.
For example, referring to figure 3, the first token in (⟨Code⟩) is ⟨Instruction⟩ (rule 2), so we need to look for `FIRST`(⟨Instruction⟩) which is given by :

FIRST(⟨Assign⟩) = {[VarName]} (rule 4 then 9), FIRST(⟨If⟩) = {IF} (rule 5 then 22), ...

Thus, the entire set of FIRST for our grammar is as follows :

$$
\begin{aligned}
\texttt{FIRST}(\langle\text{Program}\rangle) &= \{\texttt{LET}\} \\
\texttt{FIRST}(\langle\text{Code}\rangle) &= \{\texttt{IF}, \texttt{WHILE}, \texttt{OUT}, \texttt{IN}, [\texttt{VarName}], \varepsilon\} \\
\texttt{FIRST}(\langle\text{Instruction}\rangle) &= \{\texttt{IF}, \texttt{WHILE}, \texttt{OUT}, \texttt{IN}, [\texttt{VarName}]\} \\
\texttt{FIRST}(\langle\text{Assign}\rangle) &= \{[\texttt{VarName}]\} \\
\texttt{FIRST}(\langle\text{ExprArith}\rangle) &= \{[\texttt{VarName}], [\texttt{Number}], \texttt{(}, \texttt{-}\} \\
\texttt{FIRST}(\langle\text{ExprArith}'\rangle) &= \{\texttt{+}, \texttt{-}, \varepsilon\} \\
\texttt{FIRST}(\langle\text{ProdArith}\rangle) &= \{[\texttt{VarName}], [\texttt{Number}], \texttt{(}, \texttt{-}\} \\
\texttt{FIRST}(\langle\text{ProdArith}'\rangle) &= \{\texttt{*}, \texttt{/}, \varepsilon\} \\
\texttt{FIRST}(\langle\text{Atom}\rangle) &= \{[\texttt{VarName}], [\texttt{Number}], \texttt{(}, \texttt{-}\} \\
\texttt{FIRST}(\langle\text{If}\rangle) &= \{\texttt{IF}\} \\
\texttt{FIRST}(\langle\text{Ifseq}\rangle) &= \{\texttt{END}, \texttt{ELSE}\} \\
\texttt{FIRST}(\langle\text{Cond}\rangle) &= \{[\texttt{VarName}], [\texttt{Number}], \texttt{(}, \texttt{-}, \texttt{!}\} \\
\texttt{FIRST}(\langle\text{NextCond}\rangle) &= \{\texttt{->}, \varepsilon\} \\
\texttt{FIRST}(\langle\text{CondSimple}\rangle) &= \{[\texttt{VarName}], [\texttt{Number}], \texttt{(}, \texttt{-}, \texttt{!}\} \\
\texttt{FIRST}(\langle\text{Comp}\rangle) &= \{\texttt{==}, \texttt{<=}, \texttt{<}\} \\
\texttt{FIRST}(\langle\text{While}\rangle) &= \{\texttt{WHILE}\} \\
\texttt{FIRST}(\langle\text{Output}\rangle) &= \{\texttt{OUT}\} \\
\texttt{FIRST}(\langle\text{Input}\rangle) &= \{\texttt{IN}\}
\end{aligned}
$$

3.2 FOLLOW Sets Computation

Computing the set of follows is more challenging. In this case we are looking for what terminals could follow $\langle A\rangle$ once it has been read. Therefore, we need to examine the rules in which $\langle A\rangle$ appears and determine what can follow it.
For example to compute FOLLOW(⟨Code⟩) we must examine rules 1, 2, 22, 24 and 33 (the rules in which ⟨Code⟩ appears in our grammar on figure 3). In rule 1, we see that ⟨Code⟩ is followed by END. In rule 2, ⟨Code⟩ appears at the end, meaning we need to check what can follow the variable being called. Since it is ⟨Code⟩ itself, which is our current concern, this rule has no importance here. In rule 22, ⟨Code⟩ is followed by ⟨IfSeq⟩, which begin with END and ELSE, so ⟨Code⟩ can be followed by these terminals. In rules 24 and 33 ⟨Code⟩ is followed by END. Thus, FOLLOW(⟨Code⟩) is {END, ELSE}.

Note that in cases where $\langle A\rangle \to \varepsilon$, we need to check what follows $\langle A\rangle$ in the rules where $\langle A\rangle$ is called and compute their follow sets to determine the follow of $\langle A\rangle$.

Thus, the entire set of FOLLOW for our grammar is as follows :

$$
\begin{aligned}
\texttt{FOLLOW}(\langle \text{Program} \rangle) &= \{\texttt{EOS}\} \\
\texttt{FOLLOW}(\langle \text{Code} \rangle) &= \{\texttt{END}, \texttt{ELSE}\} \\
\texttt{FOLLOW}(\langle \text{Instruction} \rangle) &= \{\texttt{:}\} \\
\texttt{FOLLOW}(\langle \text{Assign} \rangle) &= \{\texttt{:}\} \\
\texttt{FOLLOW}(\langle \text{ExprArith} \rangle) &= \{\texttt{)}, \texttt{:}, \texttt{\}}, \texttt{==}, \texttt{<=}, \texttt{<}, \texttt{->}, \texttt{|}\} \\
\texttt{FOLLOW}(\langle \text{ExprArith}' \rangle) &= \{\texttt{)}, \texttt{:}, \texttt{\}}, \texttt{==}, \texttt{<=}, \texttt{<}, \texttt{->}, \texttt{|}\} \\
\texttt{FOLLOW}(\langle \text{ProdArith} \rangle) &= \{\texttt{)}, \texttt{:}, \texttt{\}}, \texttt{+}, \texttt{-}, \texttt{==}, \texttt{<=}, \texttt{<}, \texttt{->}, \texttt{|}\} \\
\texttt{FOLLOW}(\langle \text{ProdArith}' \rangle) &= \{\texttt{)}, \texttt{:}, \texttt{\}}, \texttt{+}, \texttt{-}, \texttt{==}, \texttt{<=}, \texttt{<}, \texttt{->}, \texttt{|}\} \\
\texttt{FOLLOW}(\langle \text{Atom} \rangle) &= \{\texttt{)}, \texttt{:}, \texttt{\}}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{==}, \texttt{<=}, \texttt{<}, \texttt{->}, \texttt{|}\} \\
\texttt{FOLLOW}(\langle \text{If} \rangle) &= \{\texttt{:}\} \\
\texttt{FOLLOW}(\langle \text{Ifseq} \rangle) &= \{\texttt{:}\} \\
\texttt{FOLLOW}(\langle \text{Cond} \rangle) &= \{\texttt{\}}, \texttt{|}\} \\
\texttt{FOLLOW}(\langle \text{NextCond} \rangle) &= \{\texttt{\}}, \texttt{|}\} \\
\texttt{FOLLOW}(\langle \text{CondSimple} \rangle) &= \{\texttt{\}}, \texttt{->}, \texttt{|}\} \\
\texttt{FOLLOW}(\langle \text{Comp} \rangle) &= \{\texttt{[VarName]}, \texttt{[Number]}, \texttt{(}, \texttt{-}\} \\
\texttt{FOLLOW}(\langle \text{While} \rangle) &= \{\texttt{:}\} \\
\texttt{FOLLOW}(\langle \text{Output} \rangle) &= \{\texttt{:}\} \\
\texttt{FOLLOW}(\langle \text{Input} \rangle) &= \{\texttt{:}\}
\end{aligned}
$$

Note that these results can be computed using our program, which is safer (and more suitable for maintenance) than computing by hand, which can quickly lead to errors due to concentration lapses.This is especially true for computations involving ⟨ExprArith⟩) to ⟨Atom⟩, which are long and interdependent. Figure 4 illustrates the computation of FOLLOW(⟨ExprArith⟩). The edges and their labels indicate the rule where the variable is called. When the variable is called several times within the same rule, the leftmost is labeled $[r]_1$, the next one as $[r]_2$, and so on (where r is the rule number).

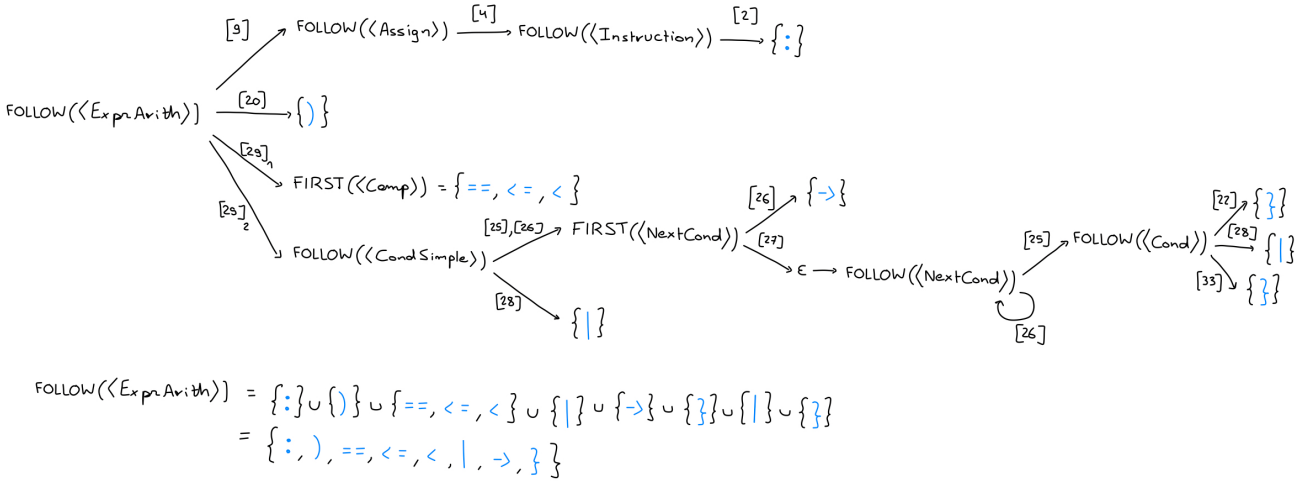

Figure 4: FOLLOW(⟨ExprArith⟩) computation

3.3 Verifications

Let's apply the definition 3.0.1 for each variable. Note that, for readability, we will only write $\texttt{FIRST}(\alpha)$ instead of $\texttt{FIRST}\left(\alpha\texttt{FOLLOW}^k(A)\right)$ when $\alpha \neq \varepsilon$, as this is equivalent.

- ⟨Program⟩ : it has only one rule, so it is not concerned.
- ⟨Code⟩: $\texttt{FIRST}(\langle\text{Instruction}\rangle) \cap \texttt{FIRST}(\varepsilon\texttt{FOLLOW}(\langle\text{Code}\rangle))$
  $= \{[\texttt{VarName}], \texttt{IF}, \texttt{WHILE}, \texttt{OUT}, \texttt{IN}\} \cap \{\texttt{END}, \texttt{ELSE}\} = \varnothing.$
- ⟨Instruction⟩: $\texttt{FIRST}(\langle\text{Assign}\rangle) \cap \texttt{FIRST}(\langle\text{If}\rangle) \cap \texttt{FIRST}(\langle\text{While}\rangle) \cap \texttt{FIRST}(\langle\text{Output}\rangle) \cap \texttt{FIRST}(\langle\text{Input}\rangle)$
  $= \{[\texttt{VarName}]\} \cap \{\texttt{IF}\} \cap \{\texttt{WHILE}\} \cap \{\texttt{OUT}\} \cap \{\texttt{IN}\} = \varnothing.$
- ⟨Assign⟩ : it has only one rule.
- ⟨ExprArith⟩ : it has only one rule.
- ⟨ExprArith'⟩ : $\texttt{FIRST}(\texttt{+}) \cap \texttt{FIRST}(\texttt{-}) \cap \texttt{FIRST}(\varepsilon\texttt{FOLLOW}(\langle\text{ExprArith'}\rangle))$
  $= \{\texttt{+}\} \cap \{\texttt{-}\} \cap \{\texttt{:}, \texttt{)}, \texttt{}\}, \texttt{==}, \texttt{<=}, \texttt{<}, \texttt{|}, \texttt{->}\} = \varnothing.$
- ⟨ProdArith⟩ : it has only one rule.
- ⟨ProdArith'⟩ : $\texttt{FIRST}(\texttt{*}) \cap \texttt{FIRST}(\texttt{/}) \cap \texttt{FIRST}(\varepsilon\texttt{FOLLOW}(\langle\text{ProdArith'}\rangle))$
  $= \{\texttt{*}\} \cap \{\texttt{/}\} \cap \{\texttt{:}, \texttt{)}, \texttt{}\}, \texttt{==}, \texttt{<=}, \texttt{<}, \texttt{+}, \texttt{-}, \texttt{|}, \texttt{->}\} = \varnothing.$
- ⟨Atom⟩ : $\texttt{FIRST}([\texttt{Number}]) \cap \texttt{FIRST}([\texttt{VarName}]) \cap \texttt{FIRST}(\texttt{(}) \cap \texttt{FIRST}(\texttt{-})$
  $= \{[\texttt{Number}]\} \cap \{[\texttt{VarName}]\} \cap \{\texttt{(}\} \cap \{\texttt{-}\} = \varnothing.$
- ⟨If⟩ : it has only one rule.
- ⟨IfSeq⟩ : $\texttt{FIRST}(\texttt{END}) \cap \texttt{FIRST}(\texttt{ELSE})$
  $= \{\texttt{END}\} \cap \{\texttt{ELSE}\} = \varnothing$
- ⟨Cond⟩ : it has only one rule.
- ⟨NextCond⟩ : $\texttt{FIRST}(\texttt{->}) \cap \texttt{FIRST}(\varepsilon\texttt{FOLLOW}(\langle\text{NextCond}\rangle))$
  $= \{\texttt{->}\} \cap \{\texttt{}\}, \texttt{|}\} = \varnothing$
- ⟨CondSimple⟩ : $\texttt{FIRST}(\texttt{|}) \cap \texttt{FIRST}(\langle\text{ExprArith}\rangle)$
  $= \{\texttt{|}\} \cap \{[\texttt{Number}], [\texttt{VarName}], \texttt{(}, \texttt{-}\ \} = \varnothing$
- ⟨Comp⟩ : $\texttt{FIRST}(\texttt{<}) \cap \texttt{FIRST}(\texttt{<=}) \cap \texttt{FIRST}(\texttt{==})$
  $= \{\texttt{<}\} \cap \{\texttt{<=}\} \cap \{\texttt{==}\} = \varnothing.$
- ⟨While⟩ : it has only one rule.
- ⟨Output⟩ : it has only one rule.
- ⟨Input⟩ : it has only one rule.

Hence, since the definition holds up in every aspect, we are certain that the grammar is strong LL1 and therefore LL1.

4 Action Table

Table 2 shows the action table for our grammar. It tells us which rule to produce for each possible symbol on the top of the stack, and each possible first character on the input. The table is constructed using the `FIRST`s, and also `FOLLOW`s when the variable has an ε-rule.

Variable	LET	[ProgName]	BE	END	ELSE	IF	THEN	WHILE	REPEAT	OUT	IN	[VarName]	[Number]	(	)	:	{	}	=	+	-	*	/	==	<=	<	->	|	EOS
⟨Program⟩	1																												
⟨Code⟩				3	3	2		2		2	2	2																	
⟨Instruction⟩						5		6		7	8	4																	
⟨Assign⟩												9																	
⟨ExprArith⟩												10	10	10							10								
⟨ExprArith′⟩															13	13		13		11	12			13	13	13	13	13	
⟨ProdArith⟩												14	14	14							14								
⟨ProdArith′⟩															17	17		17		17	17	15	16	17	17	17	17	17	
⟨Atom⟩												19	18	20							21								
⟨If⟩						22																							
⟨Ifseq⟩				23	24																								
⟨Cond⟩												25	25	25							25							25	
⟨NextCond⟩																		27									26	27	
⟨CondSimple⟩												29	29	29							29							28	
⟨Comp⟩																								30	31	32			
⟨While⟩								33																					
⟨Output⟩										34																			
⟨Input⟩											35																		

Table 2: `GILLES` Action Table

5 Details of Implementation

Below, is an overview of the main components of the implementation and their functionalities:

- `GlsGrammar`: This class represents the set of production rules for the grammar. It includes methods to compute the `FIRST` and `FOLLOW` sets, as well as to build the action table necessary for the parsing process.
- `GlsParser`: This class contains the `parse` method, which uses the lexer and the grammar to parse a given file and construct a derivation tree.
- `GlsTerminal` and `GlsVariable`: These classes represent the terminal and non-terminal symbols in the production rules of the grammar, respectively.
- `LexicalSymbol`: This class encapsulates the tokens returned by the lexer, providing additional metadata such as line and column numbers for error reporting and debugging.
- `Main`: This class orchestrates the execution of the parser. It parses the input file, prints the leftmost derivation, and optionally writes the derivation tree to a file depending on the command-line arguments.
- `ParseTree`: Based on the project-provided implementation, this class models the parse tree with additional methods for updating the children of a node.
- `ProductionRule`: This class represents production rules in the form $\alpha \to \beta$, where α is a variable and β is a sequence of symbols.
- `Symbol Interface`: Implemented by `GlsVariable` and `GlsTerminal`, this interface ensures consistency in the representation of grammar symbols.

Note that the parsing algorithm and action table algorithm are directly implemented from the pseudocodes in the lecture notes. The program can be run with `java -jar part2.jar [-wt <out>.tex] <file>.gls`.

5.1 Example of Execution

The figure 5 presents a `GILLES` file on which the parser returns the following leftmost derivation : `1 2 4 9 10 14 18 17 11 14 18 15 18 17 13 3`, where each number denotes the rule applied. Namely :

[1]	⟨Program⟩	→ `LET` [`ProgName`] `BE` ⟨Code⟩ `END`	[11]	⟨ExprArith′⟩	→ `+`⟨ProdArith⟩⟨ExprArith′⟩
[2]	⟨Code⟩	→ ⟨Instruction⟩`:`⟨Code⟩	[13]		→ `-`⟨ProdArith⟩⟨ExprArith′⟩
[3]		→ ε	[14]		→ ε
[4]	⟨Instruction⟩	→ ⟨Assign⟩	[15]	⟨ProdArith⟩	→ ⟨Atom⟩⟨ProdArith′⟩
[9]	⟨Assign⟩	→ [`VarName`]`=`⟨ExprArith⟩	[17]		→ `/`⟨Atom⟩⟨ProdArith′⟩
[10]	⟨ExprArith⟩	→ ⟨ProdArith⟩⟨ExprArith′⟩	[18]		→ ε

```
1 LET Priority BE
2   x = 1+2*3:
3 END
```

Figure 5: Example of `GILLES` file

Figure 6 displays the corresponding parse tree, which clearly corresponds to our explanation through this report.

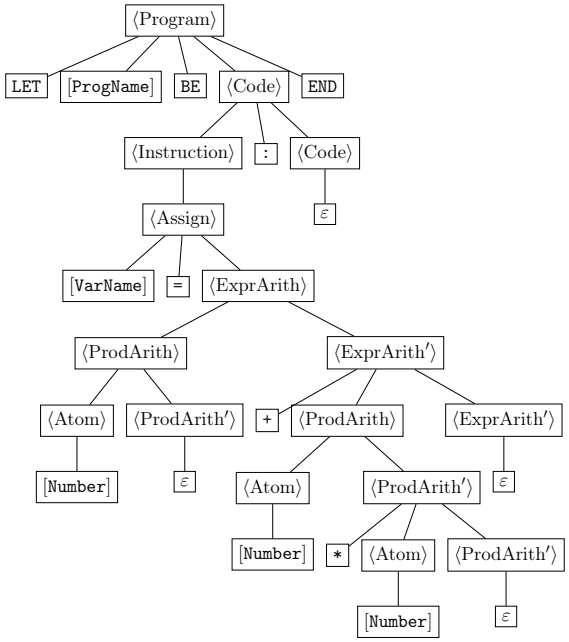

Figure 6: Parse tree of the example of `GILLES` file

Note that several test files are provided to illustrate other behaviors of the parser.

6 Conclusion

We have previously developed the lexer for `GILLES`, capable of parsing a `GILLES` file and extracting its corresponding tokens. In this phase, we focused on implementing an `LL(1)` parser for `GILLES`. We outlined the entire process, beginning with grammar modifications to satisfy the `LL(1)` requirements, followed by constructing the parsing table and implementing the parser in Java.

The final milestone in the construction of `GILLES` will be its translation into machine language, completing the pipeline from source code to executable instructions.