# INFO403 – Part 2: Syntactic Analysis of yaLcc
LL(1) Grammar Transformation and Parser Implementation

Zakaria M'Hamdi (000630126) & Wassim Turk (000630128)

November 2025

**Abstract**

This report presents the transformation of the yaLcc grammar and the implementation of an LL(1) parser. Following the lexical analysis phase of Part 1, this work focuses on grammar refinement, removal of ambiguities, and structural adjustments (elimination of left recursion and factorization) to ensure deterministic parsing. The resulting grammar forms the foundation for the implementation of the parser that interprets yaLcc source programs.

## Contents

## 1 Introduction

In the compilation process, **syntactic analysis** (or *parsing*) follows lexical analysis. Its goal is to verify whether a sequence of tokens generated by the scanner matches with syntactic rules of the language.

A **parser** is the compiler component responsible for this task. It takes as input the sequence of tokens produced by the lexical analyzer and checks whether it matches the language grammar. If so, it produces a hierarchical structure called a *parse tree* (or derivation tree), which represents the syntactic relationships between the elements of the program.

In this project, we design a deterministic **LL(1) recursive descent parser** for the *yaLcc* language. The term LL(1) stands for:

- **L**eft-to-right reading of the input,
- **L**eftmost derivation construction,
- and **1** lookahead token to decide which production rule to apply.

LL(1) grammars are non-ambiguous, non-left-recursive, and can be parsed deterministically without backtracking. They are therefore well-suited for top-down parsing implementations, where each non-terminal is mapped to a corresponding parsing function in code.

## 2 Objectives of Part 2

The main objective of this part is to construct the **syntactic analyzer (parser)** for the yaLcc language. To do so, we must:

1. Transform the original yaLcc grammar to a LL(1) grammar :
    - By removing unproductive or unreachable non-terminals (if any);
    - By Resolving ambiguities by encoding operator **precedence** and **associativity**;
    - By Eliminating left recursion and apply factorization where necessary.
2. Verify that the final grammar is LL(1) by computing its **FIRST** and **FOLLOW** sets.
3. Construct the LL(1) action table and justify it.
4. Finally, implement a recursive descent parser in Java.

## 3 Original Grammar of yaLcc

Before any modification, we start from the original grammar provided in the project. This grammar defines the full syntax of yaLcc before any transformation. Each rule is numbered as in Table 1 of the assignment:

| | | |
|---|---|---|
| [1] | $\langle Program \rangle$ | $\rightarrow$ Prog [ProgName] Is $\langle Code \rangle$ End |
| [2] | $\langle Code \rangle$ | $\rightarrow$ $\langle Instruction \rangle$ ; $\langle Code \rangle$ |
| [3] | | $\rightarrow \varepsilon$ |
| [4] | $\langle Instruction \rangle$ | $\rightarrow$ $\langle Assign \rangle$ |
| [5] | | $\rightarrow$ $\langle If \rangle$ |
| [6] | | $\rightarrow$ $\langle While \rangle$ |
| [7] | | $\rightarrow$ $\langle Call \rangle$ |
| [8] | | $\rightarrow$ $\langle Output \rangle$ |
| [9] | | $\rightarrow$ $\langle Input \rangle$ |
| [10] | $\langle Assign \rangle$ | $\rightarrow$ [VarName] = $\langle ExprArith \rangle$ |
| [11] | $\langle ExprArith \rangle$ | $\rightarrow$ [VarName] |
| [12] | | $\rightarrow$ [Number] |
| [13] | | $\rightarrow$ ( $\langle ExprArith \rangle$ ) |
| [14] | | $\rightarrow$ - $\langle ExprArith \rangle$ |
| [15] | | $\rightarrow$ $\langle ExprArith \rangle$ $\langle Op \rangle$ $\langle ExprArith \rangle$ |
| [16] | $\langle Op \rangle$ | $\rightarrow$ + |
| [17] | | $\rightarrow$ - |
| [18] | | $\rightarrow$ * |
| [19] | | $\rightarrow$ / |
| [20] | $\langle If \rangle$ | $\rightarrow$ If $\langle Cond \rangle$ Then $\langle Code \rangle$ End |
| [21] | | $\rightarrow$ If $\langle Cond \rangle$ Then $\langle Code \rangle$ Else $\langle Code \rangle$ End |
| [22] | $\langle Cond \rangle$ | $\rightarrow$ $\langle Cond \rangle$ `->` $\langle Cond \rangle$ |
| [23] | | $\rightarrow$ `\|` $\langle Cond \rangle$ `\|` |
| [24] | | $\rightarrow$ $\langle ExprArith \rangle$ $\langle Comp \rangle$ $\langle ExprArith \rangle$ |
| [25] | $\langle Comp \rangle$ | $\rightarrow$ == |
| [26] | | $\rightarrow$ `<=` |
| [27] | | $\rightarrow$ `<` |
| [28] | $\langle While \rangle$ | $\rightarrow$ While $\langle Cond \rangle$ Do $\langle Code \rangle$ End |
| [29] | $\langle Output \rangle$ | $\rightarrow$ Print([VarName]) |
| [30] | $\langle Input \rangle$ | $\rightarrow$ Input([VarName]) |

Table 1: The yaLcc original grammar.

Before implementing an LL(1) parser, the grammar must first be transformed to ensure that it can be parsed deterministically. The transformations required in steps (a), (b), and (c) have specific objectives:

- **Step (a)** — Removing unproductive and unreachable variables ensures that every non-terminal in the grammar can eventually produce a valid sequence of terminals and that all rules are accessible from the start symbol. This guarantees that the grammar defines only meaningful and reachable constructs.
- **Step (b)** — Resolving ambiguities by defining operator precedence and associativity eliminates multiple possible parse trees for the same expression. This is essential to make the grammar deterministic and to ensure that the parser can always select one unique production for each input.
- **Step (c)** — Removing left recursion and applying left factoring are mandatory transformations for top-down parsing. Left recursion causes infinite recursion in recursive descent parsers, while common prefixes between productions make prediction impossible. Eliminating them guarantees that the grammar can be parsed using a single lookahead token, as required by the LL(1) property.

Together, these transformations make the grammar suitable for predictive parsing and are a prerequisite for constructing the LL(1) action table and implementing the recursive descent parser.

# 4 Grammar Transformation Steps

## 4.1 (a) Removing Unproductive and Unreachable Variables

The original grammar (Table 1) contains a non-terminal `<Call>` referenced by `<Instruction>` but with no production rule defining it. Hence, `<Call>` is non-productive. We therefore removed the alternative `<Call>` from `<Instruction>`. All remaining non-terminals are productive and reachable from `<Program>`.

Removing such a symbol is essential for obtaining a consistent grammar. A non-productive non-terminal could never generate a valid terminal string and would lead the parser into dead ends during derivation. Eliminating `<Call>` ensures that every production is reachable and that the grammar forms a correct basis for LL(1) transformation.

## 4.2 (b) Resolving Ambiguity: Operator Precedence and Associativity

The original grammar contained ambiguous constructs for arithmetic and boolean expressions. In particular, the rule:

$$\langle ExprArith \rangle \rightarrow \langle ExprArith \rangle \ \langle Op \rangle \ \langle ExprArith \rangle$$

does not encode operator precedence or associativity. An expression such as `a - b - c` may be parsed either as `(a - b) - c` or `a - (b - c)`, producing ambiguity and preventing deterministic parsing.

To eliminate ambiguity, we introduced a layered structure for arithmetic expressions, following the precedence and associativity rules specified in the project:

- `*` and `/` have higher precedence than `+` and `-`.
- All binary arithmetic operators are left-associative.
- Unary minus has the highest precedence and is right-associative.

This leads to the following hierarchy:

$$\begin{array}{rcl}
\langle ExprArith \rangle & \rightarrow & \langle Prod \rangle \ \langle ExprArith' \rangle \\
\langle ExprArith' \rangle & \rightarrow & + \ \langle Prod \rangle \ \langle ExprArith' \rangle \ \mid \ - \ \langle Prod \rangle \ \langle ExprArith' \rangle \ \mid \ \epsilon \\
\langle Prod \rangle & \rightarrow & \langle Atom \rangle \ \langle Prod' \rangle \\
\langle Prod' \rangle & \rightarrow & * \ \langle Atom \rangle \ \langle Prod' \rangle \ \mid \ / \ \langle Atom \rangle \ \langle Prod' \rangle \ \mid \ \epsilon \\
\langle Atom \rangle & \rightarrow & - \ \langle Atom \rangle \ \mid \ [VarName] \ \mid \ [Number] \ \mid \ (\langle ExprArith \rangle)
\end{array}$$

For boolean expressions, the final grammar adopts a different structure from the arithmetic case. The implication operator `->` is right-associative, comparison operators bind more tightly, and the language includes the absolute-value form `|Cond|`. This is encoded through the following productions:

$$\begin{array}{rcl}
\langle Cond \rangle & \rightarrow & \langle Imp \rangle \\
\langle Imp \rangle & \rightarrow & \langle AtomImp \rangle \ \langle Imp' \rangle \\
\langle Imp' \rangle & \rightarrow & \texttt{->} \ \langle Imp \rangle \ \mid \ \epsilon \\
\langle AtomImp \rangle & \rightarrow & | \langle Cond \rangle | \ \mid \ \langle Rel \rangle \\
\langle Rel \rangle & \rightarrow & \langle ExprArith \rangle \ \langle RelTail \rangle \\
\langle RelTail \rangle & \rightarrow & \langle Comp \rangle \ \langle ExprArith \rangle
\end{array}$$

This structure ensures that:

- comparisons (`==`, `<=`, `<`) have higher precedence,
- implications form right-associative chains,
- `|Cond|` acts as an atomic boolean expression.

By explicitly encoding precedence, associativity, and the boolean constructs in this hierarchical manner, every expression has a unique parse tree. The grammar becomes unambiguous and suitable for predictive LL(1) parsing.

### 4.3 (c) Removing Left Recursion and Applying Factorization

Even after resolving ambiguity, some rules of the original grammar remained left-recursive. For instance:

$$\langle ExprArith\rangle \rightarrow \langle ExprArith\rangle\ \langle Op\rangle\ \langle ExprArith\rangle$$

was rewritten into the right-recursive and fully factored form described above. Right recursion is essential for recursive–descent parsing, as left recursion would lead to infinite recursion without consuming input.

The `If` rule also required left factoring. The original specification:

$$\begin{aligned}\langle If\rangle &\rightarrow If\{\langle Cond\rangle\}Then\langle Code\rangle End\\ &\mid If\{\langle Cond\rangle\}Then\langle Code\rangle Else\langle Code\rangle End\end{aligned}$$

contains a common prefix. With a single lookahead token, the parser cannot distinguish whether the next keyword will be `Else` or `End`. To resolve this, we factorized the rule:

$$\langle If\rangle \rightarrow If\{\langle Cond\rangle\}Then\langle Code\rangle\langle IfTail\rangle$$

$$\langle IfTail\rangle \rightarrow Else\langle Code\rangle End \mid End$$

All recursive occurrences are now rightmost, which ensures that each parsing step consumes input and guarantees determinism. The transformations performed — elimination of left recursion, left factoring, and precedence encoding — produce a grammar that is unambiguous, right-recursive, and fully compatible with LL(1) predictive parsing.

With these structural issues resolved, we can now compute the FIRST and FOLLOW sets to verify the LL(1) property formally.

## 5 Final Grammar

### 5.1 Introduction to the Final Grammar

After applying all transformations required in steps (a), (b), and (c), we obtain a new grammar that is unambiguous, left-recursion free, fully factorized, and therefore compatible with predictive LL(1) parsing. The grammar below is the **final specification** used in our recursive–descent parser.

All productions have been renumbered to ensure consistency with the leftmost derivation output required by the project instructions. This renumbering does not affect the semantics of the grammar but is essential for implementation.

To later justify that this grammar is LL(1), we compute its `FIRST` and `FOLLOW` sets. These sets are defined as follows:

- **FIRST(A)** is the set of terminals that may begin a string derived from the non-terminal $A$.
- **FOLLOW(A)** is the set of terminals that may immediately follow $A$ in any valid derivation.

If one alternative of a non-terminal derives $\varepsilon$, we must additionally ensure that:

$$\mathrm{FIRST}(\beta) \cap \mathrm{FOLLOW}(A) = \varnothing,$$

for all other alternatives $\beta$ of $A$. These constraints are necessary to guarantee that the grammar satisfies the LL(1) property.

The final grammar and its computed `FIRST` / `FOLLOW` sets are presented below.

## 5.2 Complete final grammar

```
[1]  <Program>     -> Prog [ProgName] Is <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>   -> <Prod> <ExprArith'>
[11] <ExprArith'>  -> <TermList>
[12] <TermList>    -> + <Prod> <TermList>
[13]               -> - <Prod> <TermList>
[14]               -> ϵ
[15] <Prod>        -> <Atom> <Prod'>
[16] <Prod'>       -> <FactorList>
[17] <FactorList>  -> * <Atom> <FactorList>
[18]               -> / <Atom> <FactorList>
[19]               -> ϵ
[20] <Atom>        -> - <Atom>
[21]               -> [VarName]
[22]               -> [Number]
[23]               -> ( <ExprArith> )
[24] <Rel>         -> <ExprArith> <RelTail>
[25] <RelTail>     -> <Comp> <ExprArith>
[26] <Comp>        -> ==
[27]               -> <=
[28]               -> <
[29] <Cond>        -> <Imp>
[30] <Imp>         -> <AtomImp> <Imp'>
[31] <Imp'>        -> -> <Imp>
[32]               -> ϵ
[33] <AtomImp>     -> |<Cond>|
[34]               -> <Rel>
[35] <If>          -> If {<Cond>} Then <Code> <IfTail>
[36] <IfTail>      -> Else <Code> End
[37]               -> End
[38] <While>       -> While {<Cond>} Do <Code> End
[39] <Output>      -> Print([VarName])
[40] <Input>       -> Input([VarName])
```

# 6 FIRST and FOLLOW Sets

### Computation of FIRST and FOLLOW Sets

To verify the LL(1) property of the transformed grammar, we manually computed the `FIRST` and `FOLLOW` sets for all non-terminals.

The computation follows the classical iterative method:

**FIRST sets.** The FIRST set of a non-terminal is computed by:

- adding terminals that appear at the beginning of its productions;
- propagating FIRST sets of non-terminals that appear first in a production;
- adding $\varepsilon$ if all symbols in a production may derive $\varepsilon$.

**FOLLOW sets.** The FOLLOW set of a non-terminal $A$ is computed by:

- adding \$ to FOLLOW(StartSymbol);
- adding FIRST sets of the symbols that follow $A$ in every production;
- propagating FOLLOW sets of the left-hand non-terminals when $A$ appears at the end of a production;
- propagating FOLLOW sets through nullable symbols when necessary.

The resulting FIRST and FOLLOW sets are shown in the tables below. These sets are then used in Section 7 to justify that the grammar is indeed LL(1).

## 6.1 FIRST sets

```
FIRST(Program)      = { Prog }

FIRST(Code)         = { [VarName], If, While, Print, Input, ϵ }

FIRST(Instruction)  = { [VarName], If, While, Print, Input }

FIRST(Assign)       = { [VarName] }

FIRST(ExprArith)    = { -, (, [VarName], [Number] }
FIRST(ExprArith')   = { +, -, ϵ }
FIRST(TermList)     = { +, -, ϵ }

FIRST(Prod)         = { -, (, [VarName], [Number] }
FIRST(Prod')        = { *, /, ϵ }
FIRST(FactorList)   = { *, /, ϵ }

FIRST(Atom)         = { -, (, [VarName], [Number] }

FIRST(Rel)          = { -, (, [VarName], [Number] }
FIRST(RelTail)      = { ==, <=, < }
FIRST(Comp)         = { ==, <=, < }

FIRST(Cond)         = { |, -, (, [VarName], [Number] }
FIRST(Imp)          = { |, -, (, [VarName], [Number] }
FIRST(Imp')         = { '->', ϵ }
FIRST(AtomImp)      = { |, -, (, [VarName], [Number] }

FIRST(If)           = { If }
FIRST(IfTail)       = { Else, End }

FIRST(While)        = { While }
FIRST(Output)       = { Print }
FIRST(Input)        = { Input }
```

## 6.2 FOLLOW sets

```
FOLLOW(Program)     = { '$' }

FOLLOW(Code)        = { End, Else }

FOLLOW(Instruction) = { ; }
FOLLOW(Assign)      = { ; }

FOLLOW(ExprArith)   = { ;, ), ==, <=, <, '->', '|', '}' }
FOLLOW(ExprArith')  = { ;, ), ==, <=, <, '->', '|', '}' }
FOLLOW(TermList)    = { ;, ), ==, <=, <, '->', '|', '}' }

FOLLOW(Prod)        = { +, -, ;, ), ==, <=, <, '->', '|', '}' }
FOLLOW(Prod')       = { +, -, ;, ), ==, <=, <, '->', '|', '}' }
FOLLOW(FactorList)  = { +, -, ;, ), ==, <=, <, '->', '|', '}' }

FOLLOW(Atom)        = { +, -, *, /, ;, ), ==, <=, <, '->', '|', '}' }

FOLLOW(Rel)         = { '->', '|', '}' }
FOLLOW(RelTail)     = { '->', '|', '}' }
FOLLOW(Comp)        = { -, (, [VarName], [Number] }

FOLLOW(Cond)        = { '|', '}' }
FOLLOW(Imp)         = { '|', '}' }
FOLLOW(Imp')        = { '|', '}' }
FOLLOW(AtomImp)     = { '->', '|', '}' }

FOLLOW(If)          = { ; }
FOLLOW(IfTail)      = { ; }
FOLLOW(While)       = { ; }
FOLLOW(Output)      = { ; }
FOLLOW(Input)       = { ; }
```

## 6.3 LL(1) Grammar Verification and Final Form

Once the grammar was disambiguated, factorized, and rewritten in a right-recursive form, we computed all **FIRST** and **FOLLOW** sets in order to verify the LL(1) property.

A grammar is LL(1) (also called strong LL(1)) if, for every non-terminal $A$ and for every pair of distinct alternatives $A \to \alpha_1$ and $A \to \alpha_2$, the following condition holds:

$$\text{FIRST}(\alpha_1\,\text{FOLLOW}(A)) \cap \text{FIRST}(\alpha_2\,\text{FOLLOW}(A)) = \varnothing.$$

Practically, this means that all alternatives of the same non-terminal must be distinguishable based on a single look-ahead symbol. In the case of nullable productions ($\alpha = \varepsilon$), the condition reduces to:

$$\big(\text{FIRST}(\alpha) \setminus \{\varepsilon\}\big) \cap \text{FOLLOW}(A) = \varnothing.$$

We applied this test to all non-terminals of the grammar. For example, $\text{FIRST}(\langle \mathit{ExprArith} \rangle) = \{-, (, [\mathit{VarName}], [\mathit{Numbe}$ and $\text{FOLLOW}(\langle \mathit{ExprArith} \rangle) = \{;, ), ==, <=, <, \texttt{->}, \texttt{|}, \texttt{"\}"}\}$. Similarly, the nullable non-terminals $\langle \mathit{ExprArith}' \rangle$, $\langle \mathit{TermList} \rangle$, $\langle \mathit{Prod}' \rangle$, $\langle \mathit{FactorList} \rangle$ and $\langle \mathit{Imp}' \rangle$ all satisfy the LL(1) constraint, since for each of them the FIRST/FOLLOW intersection associated with the $\varepsilon$ alternative is empty.

Boolean constructs were checked as well. The rule

$$\langle \mathit{Imp}' \rangle \to \texttt{->}\,\langle \mathit{Imp} \rangle \mid \varepsilon$$

has clearly disjoint FIRST sets, and $\text{FOLLOW}(\langle \mathit{Imp}' \rangle) = \{\texttt{|}, )\}$ does not create any conflict with the non-$\varepsilon$ alternative. Likewise,

$$\langle \mathit{AtomImp} \rangle \to \texttt{|}\,\langle \mathit{Cond} \rangle\,\texttt{|} \quad \mid \quad \langle \mathit{Rel} \rangle$$

yields disjoint FIRST sets: $\{|\}$ for the first alternative and $\{-, (, [VarName], [Number]\}$ for the second.

Although certain non-terminals such as $\langle Cond\rangle$ present an intersection between FIRST($A$) and FOLLOW($A$), this does not violate the LL(1) conditions. An intersection becomes problematic only when $A$ has an $\varepsilon$-production, because the parser must then choose between using FIRST($A$) or FOLLOW($A$) to predict the appropriate alternative. Since $\langle Cond\rangle$ does not derive $\varepsilon$, its FIRST and FOLLOW sets can overlap without creating any ambiguity.

All checks were therefore satisfied, and the resulting grammar is fully LL(1)-compliant. The following section verifies these conditions for each non-terminal and justifies the construction of the LL(1) parsing table.

## 7 Justification that the Grammar is LL(1)

As recalled in Section 6.3, a grammar is LL(1) if, for each non-terminal $A$ and every pair of productions $A \to \alpha$ and $A \to \beta$, the following conditions hold:

1. The FIRST sets of $\alpha$ and $\beta$ are disjoint:

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing.$$

2. If $\varepsilon \in \text{FIRST}(\alpha)$, then for every other alternative $\beta$ of $A$:

$$\text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \varnothing.$$

We now verify these conditions for each non-terminal in the transformed grammar.

### Program

$$Program \to Prog\ [ProgName]\ Is\ Code\ End$$

There is only one production. No conflict is possible. **LL(1) satisfied.**

### Code

$$Code \to Instruction\ ;\ Code \quad | \quad \epsilon$$

We have:

$$FIRST(Instruction) = \{[VarName],\ If,\ While,\ Print,\ Input\}, \quad FIRST(\epsilon) = \{\epsilon\},$$

$$FOLLOW(Code) = \{End,\ Else\}.$$

There is no overlap between $FIRST(Instruction)$ and $FOLLOW(Code)$, and the FIRST sets of both alternatives are disjoint. **LL(1) satisfied.**

### Instruction

$$Instruction \to Assign \mid If \mid While \mid Output \mid Input$$

Each alternative begins with a distinct terminal. **No conflict → LL(1) satisfied.**

### Assign

$$Assign \to [VarName]\ =\ ExprArith$$

Single production. **LL(1) satisfied.**

## ExprArith

$$ExprArith \to Prod\ ExprArith'$$

Single production. **LL(1) satisfied.**

## ExprArith'

$$ExprArith' \to TermList$$

There is only one production. By construction we have

$$FIRST(ExprArith') = FIRST(TermList) = \{+,\ -,\ \varepsilon\}.$$

All LL(1) conditions are therefore checked directly on the non-terminal `TermList` (see the discussion for `TermList` below in the action-table justification). **LL(1) satisfied.**

## Rel

$$Rel \to ExprArith\ RelTail$$

Single production. **LL(1) satisfied.**

## RelTail

$$\langle RelTail\rangle \ \to\ \langle Comp\rangle\ \langle ExprArith\rangle$$

Since `RelTail` has a **single production** and does **not** derive $\epsilon$, no FIRST/FOLLOW conflict can occur. The grammar does not require any disambiguation step for this non-terminal.

$$FIRST(RelTail) = \{==,\ <=,\ <\}$$

Because there is no $\epsilon$-alternative, FOLLOW sets do not play any role in predicting the correct rule. **LL(1) satisfied.**

## Comp

$$Comp \to ==\ \mid\ <=\ \mid\ <$$

Each alternative begins with a different terminal. **LL(1) satisfied.**

## AtomImp

$$AtomImp \to |\,Cond\,| \quad | \quad Rel$$

$$FIRST(|\,Cond\,|) = \{\text{'|'}\}, \qquad FIRST(Rel) = \{-,\ (,\ [VarName],\ [Number]\}.$$

These sets are disjoint. **LL(1) satisfied.**

## Imp'

$$Imp' \to ->\ Imp \mid \epsilon$$

$$FIRST(->\ Imp) = \{\text{'->'}\}, \qquad FIRST(\epsilon) = \{\epsilon\},$$

$$FOLLOW(Imp') = \{\text{'|'},\ \text{'}\}\text{'}\}.$$

No intersection occurs; thus the $\epsilon$-rule is safe. **LL(1) satisfied.**

### Imp

$$Imp \to AtomImp\ Imp'$$

Single production. **LL(1) satisfied.**

### Cond

$$Cond \to Imp$$

Single production. **LL(1) satisfied.**

### IfTail

$$IfTail \to Else\ Code\ End \quad | \quad End$$

$$FIRST(Else\ Code\ End) = \{Else\}, \qquad FIRST(End) = \{End\}.$$

These sets are disjoint. **LL(1) satisfied.**

### If, While, Output, Input

Each of these non-terminals has a single production. **LL(1) satisfied.**

### Conclusion

All non-terminals satisfy the LL(1) conditions:

- FIRST sets of alternative productions are disjoint.
- For productions that include $\epsilon$, their FOLLOW sets do not conflict with the FIRST sets of the other alternatives.

**Therefore, the grammar is LL(1).**

## 8 LL(1) Action Table and Justification

The LL(1) parsing (action) table is constructed using the FIRST and FOLLOW sets computed for the final transformed grammar.

For each non-terminal $A$ and each production $A \to \alpha$:

1. Compute $FIRST(\alpha)$.
2. For each terminal $a \in FIRST(\alpha)$ with $a \neq \epsilon$, place the production in the table entry:
$$M[A, a] = A \to \alpha.$$
3. If $\epsilon \in FIRST(\alpha)$, then for each terminal $b \in FOLLOW(A)$, place:
$$M[A, b] = A \to \alpha.$$

All other entries $M[A, a]$ are error entries. This construction uses exactly the FIRST and FOLLOW sets computed in the previous section.

### Justification with FIRST and FOLLOW Sets

We now justify the non-error entries of the table for each non-terminal, using the relevant FIRST and FOLLOW sets.

**Program**

$$Program \to Prog\ [ProgName]\ Is\ Code\ End$$

There is only one production. Therefore:

$$M[Program,\ Prog] = Program \to Prog\ [ProgName]\ Is\ Code\ End.$$

**Code**

$$Code \to Instruction\ ;\ Code\ \mid\ \epsilon$$

We have:

$$FIRST(Instruction) = \{[VarName], If, While, Print, Input\}, \quad FIRST(\epsilon) = \{\epsilon\},$$

$$FOLLOW(Code) = \{End, Else\}.$$

Thus:

$$\begin{aligned} M[Code,\ [VarName]] &= Code \to Instruction\ ;\ Code, \\ M[Code,\ If] &= Code \to Instruction\ ;\ Code, \\ M[Code,\ While] &= Code \to Instruction\ ;\ Code, \\ M[Code,\ Print] &= Code \to Instruction\ ;\ Code, \\ M[Code,\ Input] &= Code \to Instruction\ ;\ Code, \\ M[Code,\ End] &= Code \to \epsilon, \\ M[Code,\ Else] &= Code \to \epsilon. \end{aligned}$$

There is no conflict: the FIRST sets of the alternatives are disjoint, and $FIRST(\epsilon)$ does not clash with $FOLLOW(Code)$.

**Instruction**

$$Instruction \to Assign\ \mid\ If\ \mid\ While\ \mid\ Output\ \mid\ Input$$

The FIRST sets of the alternatives are:

$$FIRST(Assign) = \{[VarName]\}, \quad FIRST(If) = \{If\}, \quad FIRST(While) = \{While\}, \quad FIRST(Output) = \{Print\},$$

which are pairwise disjoint. The corresponding table entries are:

$$\begin{aligned} M[Instruction,\ [VarName]] &= Instruction \to Assign, \\ M[Instruction,\ If] &= Instruction \to If, \\ M[Instruction,\ While] &= Instruction \to While, \\ M[Instruction,\ Print] &= Instruction \to Output, \\ M[Instruction,\ Input] &= Instruction \to Input. \end{aligned}$$

**Assign**

$$Assign \to [VarName]\ =\ ExprArith$$

Only one production; we obtain:

$$M[Assign,\ [VarName]] = Assign \to [VarName]\ =\ ExprArith.$$

**ExprArith**

$$ExprArith \to Prod\ ExprArith'$$

With

$$FIRST(Prod) = \{-, (, [VarName], [Number]\},$$

we have:

$$\begin{aligned}
M[ExprArith,\ -] &= ExprArith \to Prod\ ExprArith',\\
M[ExprArith,\ (] &= ExprArith \to Prod\ ExprArith',\\
M[ExprArith,\ [VarName]] &= ExprArith \to Prod\ ExprArith',\\
M[ExprArith,\ [Number]] &= ExprArith \to Prod\ ExprArith'.
\end{aligned}$$

**ExprArith'**

$$ExprArith' \to TermList$$

There is only one production. Since

$$FIRST(TermList) = FIRST(ExprArith') = \{+, -, \epsilon\},$$

we get:

$$M[ExprArith',\ +] = ExprArith' \to TermList, \quad M[ExprArith',\ -] = ExprArith' \to TermList.$$

The internal choice between $+$, $-$, or $\epsilon$ is handled at non-terminal `TermList` (see below).

**TermList**

$$TermList \to +\ Prod\ TermList\ \mid\ -\ Prod\ TermList\ \mid\ \epsilon$$

$$\begin{gathered}
FIRST(+\dots) = \{+\}, \quad FIRST(-\dots) = \{-\}, \quad FIRST(\epsilon) = \{\epsilon\},\\
FOLLOW(TermList) = \{;,\ ), ==, <=, <, ->, |,\ \}\}.
\end{gathered}$$

Thus:

$$\begin{aligned}
& M[TermList,\ +] = TermList \to +\ Prod\ TermList,\\
& M[TermList,\ -] = TermList \to -\ Prod\ TermList,\\
\text{for all } a \in FOLLOW(TermList): \quad & M[TermList,\ a] = TermList \to \epsilon.
\end{aligned}$$

The FIRST sets of the non-$\epsilon$ alternatives are disjoint, and $FOLLOW(TermList)$ does not intersect $\{+, -\}$, hence the LL(1) conditions are satisfied.

**Prod**

$$Prod \to Atom\ Prod'$$

With

$$FIRST(Atom) = \{-, (, [VarName], [Number]\},$$

we obtain:

$$\begin{aligned}
M[Prod,\ -] &= Prod \to Atom\ Prod',\\
M[Prod,\ (] &= Prod \to Atom\ Prod',\\
M[Prod,\ [VarName]] &= Prod \to Atom\ Prod',\\
M[Prod,\ [Number]] &= Prod \to Atom\ Prod'.
\end{aligned}$$

**Prod'**

$$Prod' \to FactorList$$

A single production; the branching is handled by `FactorList`.

**FactorList**

$$FactorList \to *\ Atom\ FactorList\ \mid\ /\ Atom\ FactorList\ \mid\ \epsilon$$

$$FIRST(*\dots) = \{*\},\quad FIRST(/\dots) = \{/\},\quad FIRST(\epsilon) = \{\epsilon\},$$
$$FOLLOW(FactorList) = \{+, -, ;, ), ==, <=, <, ->, |,\ \}\}.$$

Thus:

$$\begin{aligned} & M[FactorList,\ *] = FactorList \to *\ Atom\ FactorList, \\ & M[FactorList,\ /] = FactorList \to /\ Atom\ FactorList, \\ \text{for all } a \in FOLLOW(FactorList):\quad & M[FactorList,\ a] = FactorList \to \epsilon. \end{aligned}$$

Again, the FIRST sets are disjoint and $FOLLOW(FactorList)$ does not contain $*$ or $/$, so the LL(1) condition holds.

**Atom**

$$Atom \to -\ Atom\ \mid\ [VarName]\ \mid\ [Number]\ \mid\ (\ ExprArith\ )$$

$$FIRST(Atom) = \{-, (, [VarName], [Number]\},$$

with each alternative starting with a distinct terminal. The corresponding entries are:

$$\begin{aligned} M[Atom,\ -] &= Atom \to -\ Atom, \\ M[Atom,\ [VarName]] &= Atom \to [VarName], \\ M[Atom,\ [Number]] &= Atom \to [Number], \\ M[Atom,\ (] &= Atom \to (\ ExprArith\ ). \end{aligned}$$

**Rel**

$$Rel \to ExprArith\ RelTail$$

Since

$$FIRST(ExprArith) = \{-, (, [VarName], [Number]\},$$

we obtain, for all $t$ in this set:

$$M[Rel,\ t] = Rel \to ExprArith\ RelTail.$$

**RelTail**

$$\langle RelTail\rangle \to \langle Comp\rangle\ \langle ExprArith\rangle$$

Since there is only one production for $\langle RelTail\rangle$, we have

$$FIRST(\langle RelTail\rangle) = \{==,\ <=,\ <\}.$$

Therefore:

$$\begin{aligned} M[\langle RelTail\rangle,\ ==] &= \langle RelTail\rangle \to \langle Comp\rangle\ \langle ExprArith\rangle, \\ M[\langle RelTail\rangle,\ <=] &= \langle RelTail\rangle \to \langle Comp\rangle\ \langle ExprArith\rangle, \\ M[\langle RelTail\rangle,\ <] &= \langle RelTail\rangle \to \langle Comp\rangle\ \langle ExprArith\rangle. \end{aligned}$$

For all other terminals, the entry for $\langle RelTail\rangle$ is an error. Since there is only one production, the non–terminal $\langle RelTail\rangle$ is trivially LL(1).

**Comp**

$$Comp \rightarrow == \mid <= \mid <$$

Each alternative starts with a distinct terminal:

$$\begin{gathered} M[Comp,\ ==] = Comp \rightarrow ==, \\ M[Comp,\ <=] = Comp \rightarrow <=, \\ M[Comp,\ <] = Comp \rightarrow < . \end{gathered}$$

**Cond**

$$Cond \rightarrow Imp$$

Single production. For all $t \in \{|,\ -, (, [VarName], [Number]\}$:

$$M[Cond,\ t] = Cond \rightarrow Imp.$$

**Imp**

$$Imp \rightarrow AtomImp\ Imp'$$

Single production. For all $t \in \{|,\ -, (, [VarName], [Number]\}$:

$$M[Imp,\ t] = Imp \rightarrow AtomImp\ Imp'.$$

**Imp'**

$$Imp' \rightarrow ->\ Imp\ \mid\ \epsilon$$

$$\begin{gathered} FIRST(->\ Imp) = \{->\}, \quad FIRST(\epsilon) = \{\epsilon\}, \\ FOLLOW(Imp') = \{|,\ \}\}. \end{gathered}$$

Thus:

$$\begin{aligned} & M[Imp',\ ->] = Imp' \rightarrow ->\ Imp, \\ \text{for all } b \in FOLLOW(Imp') : \quad & M[Imp',\ b] = Imp' \rightarrow \epsilon. \end{aligned}$$

There is no intersection between $\{->\}$ and $FOLLOW(Imp')$, so the LL(1) condition is satisfied.

**AtomImp**

$$AtomImp \rightarrow |\,Cond\,|\ \mid\ Rel$$

$$FIRST(|\,Cond\,|) = \{|\}, \quad FIRST(Rel) = \{-, (, [VarName], [Number]\},$$

which are disjoint. Therefore:

$$\begin{aligned} & M[AtomImp,\ |] = AtomImp \rightarrow |\,Cond\,|, \\ \text{for all } t \in \{-, (, [VarName], [Number]\} : \quad & M[AtomImp,\ t] = AtomImp \rightarrow Rel. \end{aligned}$$

**If**

$$If \rightarrow If\ \{Cond\}\ Then\ Code\ IfTail$$

Single production:

$$M[If,\ If] = If \rightarrow If\ \{Cond\}\ Then\ Code\ IfTail.$$

**IfTail**

$$IfTail \to Else\ Code\ End \ \mid \ End$$

$$FIRST(Else\ Code\ End) = \{Else\}, \quad FIRST(End) = \{End\},$$

so:

$$M[IfTail,\ Else] = IfTail \to Else\ Code\ End,$$
$$M[IfTail,\ End] = IfTail \to End.$$

**While**

$$While \to While\ \{Cond\}\ Do\ Code\ End$$

Single production:

$$M[While,\ While] = While \to While\ \{Cond\}\ Do\ Code\ End.$$

**Output**

$$Output \to Print([VarName])$$

Single production:

$$M[Output,\ Print] = Output \to Print([VarName]).$$

**Input**

$$Input \to Input([VarName])$$

Single production:

$$M[Input,\ Input] = Input \to Input([VarName]).$$

**Note on the structure of the LL(1) table.** A common misunderstanding is that each non-terminal should appear with only one production in its corresponding row of the LL(1) table. This is not required. Since many non-terminals in the grammar have multiple alternatives (e.g., $Code \to Instruction\ ;\ Code$ and $Code \to \epsilon$), it is normal for a row to contain several different production numbers.

What the LL(1) property strictly requires is that *each table entry* $M[A, a]$ contains **at most one** production. That is, for every pair consisting of a non-terminal $A$ and a terminal $a$, the parser must never face a choice between two different productions. In our table, every cell contains either a single production or an error entry, which confirms that the grammar is indeed LL(1).

## LL(1) action table – statements

Terminals (columns): $\{Prog,\ [VarName],\ If,\ While,\ Print,\ Input,\ End,\ Else\}$

| Non-terminal | Prog | [VarName] | If | While | Print | Input | End | Else |
|---|---|---|---|---|---|---|---|---|
| Program | [1] | error | error | error | error | error | error | error |
| Code | error | [2] | [2] | [2] | [2] | [2] | [3] | [3] |
| Instruction | error | [4] | [5] | [6] | [7] | [8] | error | error |
| Assign | error | [9] | error | error | error | error | error | error |
| If | error | error | [35] | error | error | error | error | error |
| IfTail | error | error | error | error | error | error | [37] | [36] |
| While | error | error | error | [38] | error | error | error | error |
| Output | error | error | error | error | [39] | error | error | error |
| Input | error | error | error | error | error | [40] | error | error |

## LL(1) action table – arithmetic and relations

Terminals (columns): $\{[VarName],\ [Number],\ -, (, +, *, /, ), ;, ==, <=, <, |, ->, \}\}$

| Non-term. | `[VarName]` | `[Number]` | `-` | `(` | `+` | `*` | `/` | `)` | `;` | `==` | `<=` | `<` | `\|` | `->` | } |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ExprArith | [10] | [10] | [10] | [10] | error | error | error | error | error | error | error | error | error | error | error |
| ExprArith' | error | error | [11] | error | [11] | error | error | [11] | [11] | [11] | [11] | [11] | [11] | [11] | [11] |
| TermList | error | error | [13] | error | [12] | error | error | [14] | [14] | [14] | [14] | [14] | [14] | [14] | [14] |
| Prod | [15] | [15] | [15] | [15] | error | error | error | error | error | error | error | error | error | error | error |
| Prod' | error | error | [16] | error | [16] | [16] | [16] | [16] | [16] | [16] | [16] | [16] | [16] | [16] | [16] |
| FactorList | error | error | [19] | error | [19] | [17] | [18] | [19] | [19] | [19] | [19] | [19] | [19] | [19] | [19] |
| Atom | [21] | [22] | [20] | [23] | error | error | error | error | error | error | error | error | error | error | error |
| RelTail | error | error | error | error | error | error | error | error | error | [25] | [25] | [25] | error | error | error |
| Rel | [24] | [24] | [24] | [24] | error | error | error | error | error | error | error | error | error | error | error |
| Comp | error | error | error | error | error | error | error | error | error | [26] | [27] | [28] | error | error | error |

## LL(1) action table – conditions and implications

Terminals (columns): $\{|,\ [VarName],\ [Number],\ -, (, ->, \}\}$

| Non-term. | `\|` | `[VarName]` | `[Number]` | `-` | `(` | `->` | } |
|---|---|---|---|---|---|---|---|
| Cond | [29] | [29] | [29] | [29] | [29] | error | error |
| Imp | [30] | [30] | [30] | [30] | [30] | error | error |
| Imp' | [32] | error | error | error | error | [31] | [32] |
| AtomImp | [33] | [34] | [34] | [34] | [34] | error | error |

## 9 Complete Action table

| | `Prog` | `[ProgName]` | `Is` | `End` | `If` | `Then` | `Else` | `While` | `Do` | `Print` | `Input` | `[VarName]` | `[Number]` | `=` | `+` | `-` | `*` | `/` | `(` | `)` | `{` | `}` | `;` | `==` | `<=` | `<` | `\|` | `->` | `$` |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Program | 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Code | | | | 3 | 2 | | 3 | 2 | | 2 | 2 | 2 | | | | | | | | | | | | | | | | | |
| Instruction | | | | | 5 | | | 6 | | 7 | 8 | 4 | | | | | | | | | | | | | | | | | |
| Assign | | | | | | | | | | | | 9 | | | | | | | | | | | | | | | | | |
| ExprArith | | | | | | | | | | | | 10 | 10 | | | 10 | | | 10 | | | | | | | | | | |
| ExprArith’ | | | | | | | | | | | | | | | 11 | 11 | | | | 11 | | 11 | 11 | 11 | 11 | 11 | 11 | | |
| TermList | | | | | | | | | | | | | | | 12 | 13 | | | | 14 | | 14 | 14 | 14 | 14 | 14 | 14 | | |
| Prod | | | | | | | | | | | | 15 | 15 | | | 15 | | | 15 | | | | | | | | | | |
| Prod’ | | | | | | | | | | | | | | | 16 | 16 | 16 | 16 | | 16 | | 16 | 16 | 16 | 16 | 16 | 16 | | |
| FactorList | | | | | | | | | | | | | | | 19 | 19 | 17 | 18 | | 19 | | 19 | 19 | 19 | 19 | 19 | 19 | | |
| Atom | | | | | | | | | | | | 21 | 22 | | | 20 | | | 23 | | | | | | | | | | |
| Rel | | | | | | | | | | | | 24 | 24 | | | 24 | | | 24 | | | | | | | | | | |
| RelTail | | | | | | | | | | | | | | | | | | | | | | | | 25 | 25 | 25 | | | |
| Comp | | | | | | | | | | | | | | | | | | | | | | | | 26 | 27 | 28 | | | |
| Cond | | | | | | | | | | | | 29 | 29 | | | 29 | | | 29 | | | | | | | | 29 | | |
| Imp | | | | | | | | | | | | 30 | 30 | | | 30 | | | 30 | | | | | | | | 30 | | |
| Imp’ | | | | | | | | | | | | | | | | | | | | | 32 | | | | | | 31 | 31 | |
| AtomImp | | | | | | | | | | | | 34 | 34 | | | 34 | | | 34 | | | | | | | | 33 | | |
| If | | | | | 35 | | | | | | | | | | | | | | | | | | | | | | | | |
| IfTail | | | 37 | | | 36 | | | | | | | | | | | | | | | | | | | | | | | |
| While | | | | | | | | 38 | | | | | | | | | | | | | | | | | | | | | |
| Output | | | | | | | | | | 39 | | | | | | | | | | | | | | | | | | | |
| Input | | | | | | | | | | | 40 | | | | | | | | | | | | | | | | | | |

Table 2: Table LL(1) — action table

## 10 Implementation of the LL(1) Parser

Having defined and verified the LL(1) grammar, we now present the architecture and implementation choices of the recursive–descent parser developed in Java. The parser is structured around a set of mutually recursive methods, one for each non-terminal of the grammar. In addition to these parsing functions defined in the main parsing class, some auxiliary classes support the construction of the parse tree and the representation of the grammar symbols.

This section gives an overview of the core components involved:

- the `Parser` class, which implements the LL(1) parsing algorithms and the grammar productions;
- the `ParseTree` class, which help builds the syntactic derivation tree in latex;
- the modifications added to the `Symbol` class(which was used in part 1 of the project) supporting the derivation output and tree construction;
- the `Main` class, which integrates the scanner and the parser to build the left most derivation list.

Our implementation directly follows the final version of the grammar. Each non-terminal of the grammar is mapped to a dedicated method in the parser, and the behaviour of these methods is determined by the LL(1) rules studied earlier. This means that every parsing decision is made using only one lookahead token, and no backtracking is needed.

### 10.1 Design of the Parser Class

The `Parser` class is the central component of the syntactic analysis. It consumes the token stream produced by the lexical analyzer and attempts to derive the input according to the grammar rules. Each method in the parser corresponds to a specific grammar production and implements it deterministically using the lookahead of 1.

#### Token Management and Lookahead

At any given time, the parser maintains a single lookahead token, obtained from the lexer. This fulfils the “1” in LL(1). The parser checks the lookahead before selecting a production. If the lookahead does not match any terminal in the FIRST set of the current non-terminal, a syntax error is thrown.

Advancing the input is centralized through a method `match(LexicalUnit expected)`, which:

1. verifies that the lookahead token matches `expected` which is a token;

2. if it matches requests the next token from the lexer to get the next symbol of lookahead or throw an syntax Exception if it doesn't match.

**Structure of the Parser and Methods for Non-Terminals**

The `Parser` class is a direct implementation of a recursive–descent LL(1) parser. Each non-terminal of the grammar is represented by one dedicated Java method (`parseCode`, `parseInstruction`, `parseExprArith`, etc.), and the algorithms inside these methods strictly follows the LL(1) structure of the grammar.

Unlike the theoretical description, the implementation does not rely on explicit tables of FIRST and FOLLOW sets. Instead, these sets are encoded directly into the code through compact helper such as `isFirstOfInstruction`, `isInFollowOfCode`, or `isInFollowOfFactorList`. This keeps the parser readable.

**Method structure.** Each parsing method follows the same overall pattern:

1. **Select the production.** The parser inspects `lookahead.getType()` to identify which grammar rule must be applied. This corresponds to FIRST/FOLLOW reasoning implemented through `if/else` blocks.
2. **Record the rule number.** The call to `pushRule(n)` store the derivation rule, printed later by `printDerivation()`.
3. **Match the terminals.** The `match(expected)` method consumes the next token or raises a `SyntaxException` if the token does not match.
4. **Recursive descent.** Non-terminals appearing on the right-hand side are parsed by calling their associated methods in order.
5. **Optional parse tree construction.** When the `buildTree` flag is set, the method creates a `ParseTree` node and attaches the children as the recursive calls unfold. Without this flag, the same parsing logic applies but no tree nodes are created.

This design guarantees that the parser remains deterministic: at each step, the lookahead token uniquely identifies the correct production, so no backtracking is ever required.

**Example: arithmetic expressions.** Consider the rule

$$\langle \mathit{ExprArith} \rangle \to \langle \mathit{Prod} \rangle \ \langle \mathit{ExprArith}' \rangle.$$

In the implementation, the method `parseExprArith()` simply calls `parseProd()` followed by `parseExprArithPrime()`, and optionally wraps both subtrees into a `ParseTree` node. This pattern is applied consistently across the grammar, including relational expressions, implication operators, `if/else` constructs, and the `while` loop.

**FIRST/FOLLOW utility methods.** The helper functions at the bottom of the class (`isFirstOfInstruction`, `isInFollowOfFactorList`, etc.) play the role of FIRST and FOLLOW sets: they resolve ambiguity and indicate when an $\varepsilon$-production must be used. For instance, `parseCode()` decides whether to parse another instruction or apply the $\varepsilon$-rule depending on whether the lookahead token lies in FIRST(Code) or FOLLOW(Code).

## 10.2 Construction of the Parse Tree

The construction of the parse tree is handled by the `ParseTree` class. Whenever the parser applies a production rule, it creates a new tree node whose label corresponds to the grammar symbol (terminal or non-terminal) on the left-hand side, and then attaches child nodes for each symbol on the right-hand side.

This structure allows the tree to reflect exactly how the input program is reduced according to the grammar.

**Tree Node Structure**

Each node of the tree contains:

- the *Symbol* it represents (terminal or non-terminal);
- a list of children, stored as `List<ParseTree>`;
- for terminal nodes, the concrete lexeme (number, variable name, operator, etc.), which is carried by the `Symbol`.

Since the grammar is designed to be LL(1) and right-recursive, the resulting tree closely matches the syntactic structure of the language. Nested expressions, conditionals, and sequences of instructions naturally appear inside the tree as hierarchical blocks.

## 10.3 Extension of the `Symbol` Class

To support the parse tree visualisation, we extended the `Symbol` class with an additional functionality. The addition is the method `toTexString()`, which converts any symbol into a LaTeX-safe string suitable for use in `forest` or TikZ environments.

The new features provide:

- a distinction between terminal and non-terminal symbols;
- a uniform mechanism to represent non-terminals using the standard notation `<NonTerminal>` when generating the parse tree;
- automatic escaping of special LaTeX characters (`$`, `_`, `%`, `#`, `{`, `}`, etc.);
- a readable string form for printing derivations and debugging.

A non-terminal is therefore displayed as `<Name>`, while a terminal is shown using its concrete token name. This unified representation ensures that the structure of the tree printed from `ParseTree` matches exactly the sequence of reductions used during parsing.

The following method implements this behaviour:

```
public String toTexString() {
    String s;

    if (this.isTerminal()) {
        if (this.value != null) s = this.value.toString();
        else if (this.type != null) s = this.type.toString();
        else s = "TERMINAL";
    } else {
        if (this.value != null) s = "<" + this.value.toString() + ">";
        else s = "$<NON_TERMINAL>$";
    }

    // Escape LaTeX special characters
    s = s.replace("\\", "\\textbackslash{}")
         .replace("_", "\\_")
         .replace("#", "\\#")
         .replace("$", "\\$")
         .replace("%", "\\%")
         .replace("&", "\\&")
         .replace("{", "\\{")
         .replace("}", "\\}")
         .replace("^", "\\textasciicircum{}")
        .replace("~", "\\textasciitilde{}");

    return s;
}
```

## 10.4 Role of the `Main` Class

The `Main` class serves as the entry point of the syntactic analysis pipeline. Its role is to orchestrate the interaction between the scanner, the parser, and the optional construction of the parse tree.

Its behaviour can be summarised as follows:

1. It interprets the command-line arguments. Using `-wt` activates the parse tree construction mode and specifies the output file in which the tree's LaTeX code will be written.
2. It initializes the lexical analyzer (`LexicalAnalyzer`) with the input file.
3. It creates a `Parser` instance and invokes `parser.parseProgram()` to start the analysis from the grammar's start symbol.
4. Upon successful parsing:
   - it prints the leftmost derivation produced by the parser;
   - if `-wt` is active, it writes the LaTeX representation of the parse tree using `ParseTree.toLaTeX()` in the /more folder.
5. It handles several kinds of errors:
   - `FileNotFoundException` (input file cannot be opened);
   - `UnkownLexicalUnitException` (scanner encountered an invalid token);
   - `SyntaxException` (parser encountered an unexpected token).

The `Main` class therefore links all components of the front-end and provides the user-facing interface to run the syntactic analyser in either normal mode or tree-generation mode.

## 10.5 Summary

Our parser is a straightforward, from scratch, implementation of the LL(1) approach, closely following the structure of the transformed grammar. Each non-terminal in the grammar corresponds to a dedicated method in the `Parser` class, which uses lookahead tokens to deterministically choose the correct production and recursively expand it.

The `ParseTree` class captures the program's syntactic structure, while the extended `Symbol` class provides a uniform representation for terminals and non-terminals for a latex friendly output.

Finally, the `Main` class ties everything together, handling input, invoking the parser, printing the derivation, and optionally generating a LaTeX representation of the parse tree. Overall, the system is deterministic, predictable, and faithful to the grammar and LL(1) principles.

## Note on Parser Output

Before discussing the test results, we briefly recall what the parser outputs. The *leftmost derivation* is the sequence of grammar rules applied while always expanding the leftmost non-terminal in the current sentential form. Since the final grammar is LL(1), this derivation is deterministic: each rule is selected based exclusively on the current non-terminal and the lookahead symbol.

The second output is the *parse tree*, a hierarchical representation of the program in which each internal node corresponds to a non-terminal and its children correspond to the right-hand side of the selected production. Nullable productions appear explicitly as $\varepsilon$ leaves.

Together, the derivation and the parse tree provide a clear and complete view of how the program is parsed according to the grammar.

# 11 Testing and Validation

To ensure that the LL(1) parser correctly implements the final grammar, we executed a series of systematic tests covering all syntactic constructs of the language. These tests verify the correctness of the derivations, the structure of the parse trees, and the handling of both valid and invalid programs. All test files are located in the `test/` directory and are executed automatically using the Makefile targets described earlier.

The parser was tested in two modes:

- **Standard mode** (`make run FILE=...`) Prints the leftmost derivation or a syntax error message.
- **Tree mode** (`make runWT FILE=...`) Generates a `.tex` file containing the full LaTeX parse tree (Forest package), compiles it to `.pdf`, and removes all LaTeX temporary files.

Additionally, the command `make runWTTest` applies this process to every file in `test/`, generating a complete suite of parse-tree PDFs in the `more/` directory.

## 11.1 11.1 Testing Methodology

The purpose of the test suite is threefold:

1. **Check the correctness of the LL(1) derivation.** Each program produces a deterministic leftmost derivation according to the 40 grammar rules.
2. **Validate the structure of the parse tree.** The resulting tree must correspond exactly to the productions used during parsing.
3. **Verify that FIRST/FOLLOW decisions apply correctly.** In particular, epsilon-rules must be selected only when the lookahead belongs to FOLLOW.

The tests cover: valid simple programs, nested conditionals, implications, arithmetic expressions, loops, whitespace handling, comments, and various error cases such as unclosed comments or invalid program names.

## 11.2 Example Test: `TooMuchSpace.ycc`

We illustrate the behaviour of our parser using a simple example containing excessive whitespace:

```
!! Too much space test !!

Prog MyProgram    Is
  Input(a)   ;
  Print(a)   ;
End
```

Even though the code contains irregular spacing, the lexical analyzer collapses whitespace and produces the correct sequence of tokens. Running the test using:

```
make runWT FILE=test/TooMuchSpace.ycc
```

produces:

```
Initialization
Variables
a 4
Done
Leftmost derivation:
1 2 8 40 2 7 39 3
Parse tree written to more/TooMuchSpace.tex
==> Compiling more/TooMuchSpace.tex to PDF...
```

The derivation corresponds to:

- Rule 1 : `<Program>`
- Rule 2 : `<Code> → <Instruction> ; <Code>`
- Rule 8 : `<Instruction> → <Input>`
- Rule 40 : `<Input>`
- Rule 2 : second instruction
- Rule 7 : `<Instruction> → <Output>`
- Rule 39 : `<Output>`
- Rule 3 : `<Code> → ε`

This matches the expected structure for:

```
Input(a);
Print(a);
```

The corresponding parse tree is generated automatically as a PDF in the `more/` folder (see Figure 1).

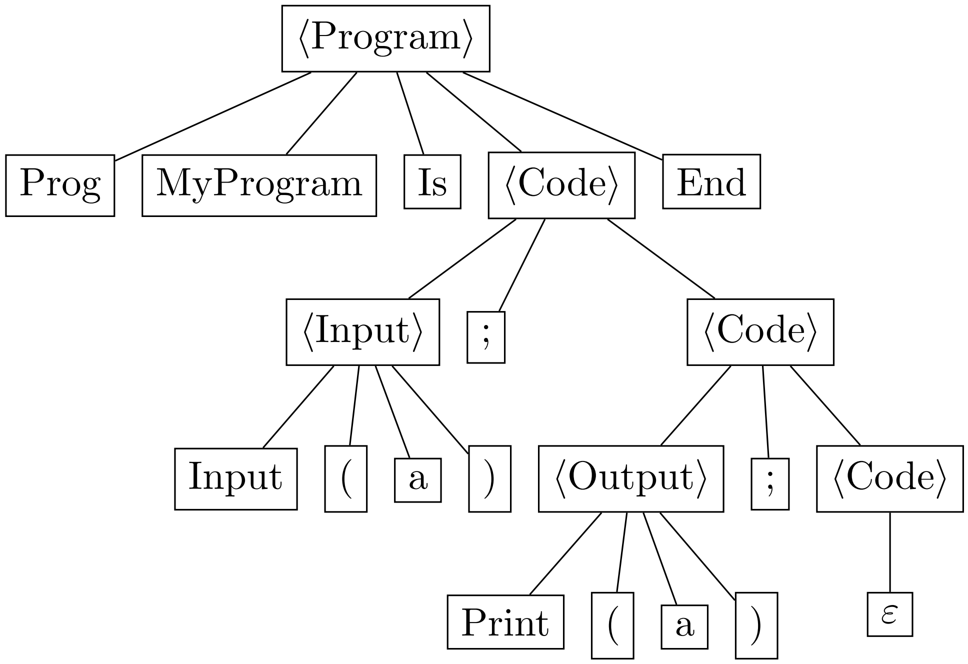

Figure 1: Parse tree generated for the test file `TooMuchSpace.ycc`.

## 11.3 Other example Test: `WhileOne.ycc`

During testing, the file `WhileOne.ycc` produced the following error:

```
Syntax error at line 6, column 10:  While parsing <RelTail>:  unexpected token RBRACK('}')
```

This behaviour is correct given the final grammar. In our language, the condition of a `While`-statement must be a well-formed logical or relational expression. According to the grammar:

$$\langle \mathit{While} \rangle \rightarrow \texttt{While}\ \{\ \langle \mathit{Cond} \rangle\ \}\ \texttt{Do}\ \langle \mathit{Code} \rangle\ \texttt{End}$$

$$\langle \mathit{Cond} \rangle \rightarrow \langle \mathit{Imp} \rangle$$

$$\langle \mathit{Imp} \rangle \rightarrow \langle \mathit{AtomImp} \rangle \, \langle \mathit{Imp'} \rangle$$

$$\langle \mathit{AtomImp} \rangle \rightarrow |\langle \mathit{Cond} \rangle|$$

$$\langle \mathit{AtomImp} \rangle \rightarrow \langle \mathit{Rel} \rangle$$

$$\langle \mathit{Rel} \rangle \rightarrow \langle \mathit{ExprArith} \rangle \, \langle \mathit{RelTail} \rangle$$

$$\langle \mathit{RelTail} \rangle \rightarrow \langle \mathit{Comp} \rangle \, \langle \mathit{ExprArith} \rangle$$

A condition therefore cannot consist solely of a numeric literal such as `1`. The input fragment

```
While {1} Do
```

is parsed as a relational expression; after reading the arithmetic expression `1`, the parser expects a comparison operator (`==`, `<=`, or `<`) in order to complete the production for $\langle \mathit{RelTail} \rangle$. Encountering the closing brace } instead leads to the reported syntax error.

This confirms that the parser behaves as specified by the grammar: a standalone number is not a valid condition in this language. To express an unconditional loop, one must use a valid relational or logical expression, for example {`1 == 1`}.

## 11.4 Discussion of Correctness

This test validates several essential aspects of the LL(1) parser:

**Correct handling of whitespace.** Whitespace is ignored by the lexical analyzer, ensuring that parsing is unaffected by formatting irregularities.

**Correct rule selection using FIRST and FOLLOW.** After reading a semicolon, the parser chooses between rule 2 and rule 3 based solely on the lookahead. Since the lookahead before `End` belongs to FOLLOW(`Code`), the epsilon rule 3 is correctly selected.

**Correct behaviour of right recursion.** The grammar uses right recursion for `Code`, and the resulting tree matches this structure precisely: `Instruction ; Code` is expanded until `Code → ε`.

**Structural compliance with the final grammar.** Every node in the tree corresponds exactly to one of the 40 productions defined in the final grammar, demonstrating that the parser respects the formal specification.

## 11.5 Summary of Test Results

All provided test files were executed successfully using both `make runTest` and `make runWTTest`. The parser correctly identified syntax errors, handled nested constructs, expanded epsilon-productions appropriately, and produced well-formed parse trees for all valid programs. These results confirm the functional correctness of our LL(1) parser and its conformity to the final grammar.

## 12 Conclusion

This project led to the design and implementation of a complete LL(1) parsing system, starting from an initially ambiguous grammar and ending with a functional recursive–descent parser capable of handling the yaLcc language. Through the successive transformations applied to the grammar—removing ambiguity, eliminating left recursion, and applying factorization—we obtained a form that can be parsed deterministically with a single lookahead symbol.

The implementation reflects the structure of this transformed grammar: each non-terminal is associated with a dedicated parsing routine, and FIRST/FOLLOW information guides the selection of productions. The resulting leftmost derivation and parse tree confirm, step by step, how the program is expanded according to the grammar.

The extensive test suite, which includes simple programs, nested constructs, implication chains, loops, whitespace variations, comment handling, and a range of error cases, demonstrated that the parser behaves predictably and that the grammar supports deterministic LL(1) parsing.

## References

- G. Geeraerts, A. Leponce, *INFO403 — Language Theory and Compiling*, Université Libre de Bruxelles, 2025.
- Aho, Lam, Sethi, Ullman — *Compilers: Principles, Techniques, and Tools* (2nd Edition), Addison-Wesley, 2006.
- JFlex Official Documentation — `https://www.jflex.de/`
- Oracle Java SE Documentation — `https://docs.oracle.com/javase/`
- ANTLR Documentation — Concepts of grammar transformations and LL parsing, `https://www.antlr.org/`