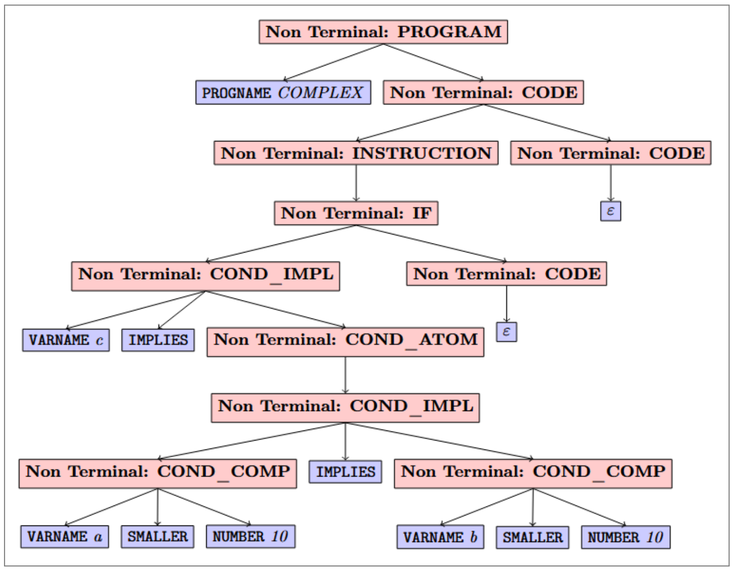
INFO—F403
Introduction to language theory and compiling
Project Report - Part 2

Berthion Antoine — 566199   Bontridder Milan - 493709

November 21, 2025



Abstract

This project implements the **parser phase** of our compiler for the toy language **YaLCC** (Yet Another Language for the Compiler Course). The parser consumes a `stream of tokens` and produces a corresponding **parse tree**, which will later be used by the code generation phase. This report presents the **design and implementation** of the parser, the underlying **LL(1) grammar** and **LL(1) table**, illustrative `input` and `output` examples, and the adopted **testing strategy**.

1 Introduction

The **parser** constitutes the second major phase of the YaLCC compiler. After **lexical analysis** has converted the raw `source code` into a structured stream of **tokens**, the parser is responsible for determining whether this sequence forms a *syntactically valid* program according to the **rules of the language**. Its primary output is a well-formed **parse tree** that reflects the hierarchical structure of the input.

In contrast with the lexer's role of recognizing individual lexical units, the parser operates at the level of **grammatical constructs**. It verifies that tokens appear in an order permitted by the language specification and that they combine to form meaningful statements, expressions, and program blocks. When the token sequence deviates from the grammar, the parser detects and reports the corresponding syntax error.

The YaLCC parser is based on an **LL(1)** specification, relying on a predictive parsing table and a deterministic top-down strategy. This approach enforces a clear separation between the **grammar design** and the **parsing algorithm**, ensuring that both the structure of the language and the implementation of the parser remain transparent and easy to reason about.

2 Project Overview

The project is organized as follows:

— `src/` : Contains all source files necessary for the **parser implementation**, including `NonTerminal.java`, `ParseException.java`, `Parser.java`, `ParserTree.java`, and `Main.java`.

— `doc/` : Contains the **Javadoc documentation** and this report PDF.

— `test/` : Contains example **YaLCC source files** and unit tests implemented using **JUnit**.

— `dist/` : Contains the executable JAR file `part2.jar`.

— `more/` : Contains other supporting files, including the project specification document.

The main **workflow** of the project can be summarized as:

```
YaLCC source file (.ycc) → LexicalAnalyzer → sequence of lexical tokens → Parser →
                                  Parse Tree
```

3 Transforming the Grammar

The initial **YaLCC grammar** was not directly suitable for predictive parsing. Several transformations were therefore required to obtain an **LL(1) grammar** and construct the final *parsing table* presented in Appendix A. The process involved **resolving ambiguities**, enforcing operator **precedence** and **associativity**, removing **left recursion**, and applying grammar factorization when necessary.

Ambiguity arises when a single input string can be parsed into **multiple valid parse trees** according to the grammar. In such cases, a predictive parser cannot deterministically choose which production to apply. **Figure 1** shows an example of a possible ambiguity that could arise.

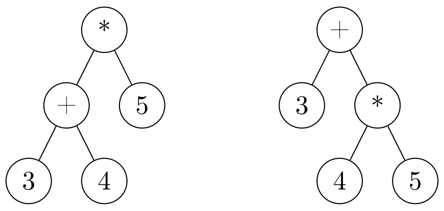


Figure 1: Ambiguity in parsing the expression $3 + 4 * 5$ without operator precedence.

Many of the transformations described below were specifically aimed at eliminating this ambiguity to allow **deterministic LL(1) parsing**.

3.1 Operator Priority and Associativity

The original specification defines multiple classes of operators for both arithmetic and boolean expressions. These operators do not all share the same precedence, and some are associative. Without making these relationships explicit in the grammar, the parser would face **ambiguities** when interpreting expressions. Appendix B summarizes the required priorities, from **highest** to **lowest**.

Parentheses `( )` act as usual in arithmetic expressions, while the symbol `| |` plays an analogous role for grouping boolean subexpressions in conditions.

To reflect these priorities explicitly, the grammar was rewritten into a sequence of **non-terminals**, each corresponding to one precedence level. For instance, `ExprAddSub` expands into `ExprMulDiv` optionally followed by repeated `(+|-) ExprMulDiv`, enforcing left associativity of addition and subtraction. The same pattern is used between `ExprMulDiv` and `ExprUnary`. Comparison and boolean operators are handled similarly within the `Cond`, `CondImpl`, and `CondAtom` rules.

3.2 Removing Left Recursion

Several expression rules in the initial grammar were naturally expressed using **left recursion**, such as

$$Expr \rightarrow Expr + Term \tag{1}$$

which is incompatible with **LL(1)** predictive parsing. These rules were restructured into **right-recursive** or iterative forms. For example, left-recursive arithmetic productions were rewritten as follows:

$$ExprAddSub \rightarrow ExprMulDiv\ \{(+|-)\ ExprMulDiv\}^* \tag{2}$$

This transformation preserves **left associativity** while eliminating **left recursion** entirely.

3.3 Grammar Factorization

In some cases, the grammar contained productions whose prefixes overlapped, making them unsuitable for **LL(1) parsing** without further transformation. The conditional construct `If` is a representative example: both the `if-then-end` and `if-then-else-end` forms share an identical prefix. The grammar was therefore factored so that the parser first recognizes the mandatory prefix

```
If {Cond} Then Code
```

before determining through a **lookahead** (1 token of lookahead) whether an `Else` branch follows. A similar strategy was applied to the definition of conditions, in particular within the `CondImpl` rules.

3.4 Resulting LL(1) Grammar

After these transformations, the grammar became suitable for predictive parsing. All ambiguities were resolved, operator precedence and associativity were explicitly enforced, left recursion was eliminated, and overlapping prefixes were factored. The resulting grammar is presented in the LL(1) parsing table in Appendix A, which serves as the basis for the recursive descent parser implementation.

4 Parser Design and Implementation

The **YaLCC parser** is implemented in **Java** as a **recursive descent parser**, leveraging the LL(1) grammar described in Section 3. The parser consumes tokens produced by the **lexer** and constructs a **parse tree** reflecting the hierarchical structure of the source code.

Key features of the parser design include:

— **Recursive Descent Structure:** Each non-terminal in the grammar has a corresponding parsing method. For example, `parseExprAddSub()` handles addition and subtraction expressions, while `parseIf()` handles conditional statements.

— **Token Matching and Lookahead:** The parser maintains a `currentToken` obtained from the lexer. The `match()` method ensures that the current token matches the expected type, throwing a `ParseException` otherwise. This design guarantees deterministic parsing consistent with the LL(1) table.

— **Parse Tree Construction:** The parser builds a `ParseTree` incrementally. Each parsing method creates a tree node representing a non-terminal or terminal, and child nodes are attached recursively to represent the derivation. Non-terminal nodes are represented using `dummy()` symbols, while terminal nodes contain the actual token information.

— **Operator Precedence and Associativity:** Expressions are parsed in a hierarchy reflecting precedence and associativity. For instance, `ExprAddSub` calls `ExprMulDiv`, which in turn calls `ExprUnary`. Loops in `parseExprAddSub()` and `parseExprMulDiv()` implement left-associativity for binary operators while preserving correct operator priority.

— **Error Handling:** Syntax errors are detected as soon as an unexpected token is encountered. The parser reports the type of expected token, the actual token found, and its position (line and column), which aids debugging and error localization.

— **Optional Productions:** Some grammar rules include optional components, such as the `Else` branch in `If` statements. The parser handles these through conditional checks and selectively adds child nodes for optional branches.

— **Integration with the Lexer:** The parser relies on the lexer to provide tokens sequentially. This modular separation allows the parser to focus purely on syntactic structure without concerning itself with character-level scanning or regular expression matching.

— **Extensible Tree Representation:** The `ParseTree` class supports conversion to LaTeX `forest` with TikZ representations. Each node stores its rule number, allowing for easy debugging and visualization of derivations.

Overall, the design prioritizes **clarity**, **maintainability**, and **deterministic parsing**. By explicitly encoding operator precedence, handling left recursion, and factoring ambiguous rules, the parser is capable of producing unambiguous parse trees suitable for subsequent semantic analysis and code generation.

5 Sample Programs and Parser Results

To evaluate our **YaLCC parser**, we executed it on a variety of `.ycc` programs to analyze its **parse tree output** and verify correct derivation of syntactic structures. We also implemented extensive **unit tests** to ensure the parser correctly constructs parse trees according to the grammar.

Several of the `.ycc` files used for testing are provided in **Appendix C**. The main focus of these tests was to observe how the parser handles:

— Minimal programs with **empty code blocks**.

— Programs with a **single statement**, such as `Input`, `Print`, or variable assignments.

— Arithmetic expressions with **operator precedence and associativity**.

— Conditional statements, including `If` and **optional `Else` branches**.

— **Complex** logical conditions with implication (`->`) and **nested expressions** inside `|...|`.

— Programs containing **multiple instructions** to test recursive parsing and tree construction.

Each test confirmed that the resulting `ParseTree` has the correct **structure**, **node types** (terminals and non-terminals), and **children counts** according to the grammar rules.

6 Parser Verification via Unit Tests

The **parser** has been rigorously tested using **JUnit 5**. The test suite ensures that programs are correctly parsed into trees with accurate hierarchy, node labeling, and operator handling. Key aspects of the testing methodology include:

— **Non-terminal Validation:** Each parsing method corresponds to a grammar production. Tests assert that non-terminal nodes in the parse tree match the expected grammar symbols, e.g., `PROGRAM`, `CODE`, `INSTRUCTION`.

— **Terminal Validation:** Leaf nodes containing variables, numbers, or operators are verified for correctness using `LexicalUnit` or actual values.

— **Tree Structure Verification:** Helper methods check that each parse tree node has the expected number of children, ensuring hierarchical integrity and correct recursive parsing.

— **Expression Parsing:** Arithmetic expressions are validated for correct operator precedence and left-associativity by inspecting the parse tree structure for nodes like `EXPR_ADDSUB` and `EXPR_MULDIV`.

— **Conditional Parsing:** `If` and `If-Else` statements are tested to ensure that optional branches are correctly represented in the parse tree.

— **Complex Conditions:** Nested conditions, implication operators (`->`), and pipe-delimited expressions (`|...|`) are verified for proper tree hierarchy.

— **Minimal Program Testing:** Programs with no statements or a single statement are used to test base cases and ensure the parser handles empty code blocks gracefully.

— **Integration with Lexer:** The parser tests depend on the `LexicalAnalyzer` to supply tokens, ensuring end-to-end validation from input string to parse tree.

The complete test suite can be executed using the command `make test`, providing automated verification of parser correctness and robustness. Each unit test asserts not only the node types and values but also the structural integrity of the parse tree, giving high confidence in the parser's behavior across all supported `.ycc` programs.

7 Use of LLMs

In this brief section, we discuss the use of large language models (LLMs) in the context of this project. No LLM was used for the implementation or for understanding the project itself. However, this report was reviewed for syntax and grammar by DeepL as well as a GPT model. It should be noted that none of the information contained in this report was generated by anyone other than the authors.

8 Conclusion

The YaLCC parser successfully implements a **deterministic LL(1) recursive descent** strategy, correctly transforming token streams into structured **parse trees**. Through grammar transformations, careful handling of operator precedence and associativity, and extensive unit testing, the parser reliably detects syntax errors and produces unambiguous derivations.

This implementation lays a solid foundation for subsequent compiler phases, including semantic analysis and code generation, while remaining clear, maintainable, and extensible.

A LL(1) Parsing Table

This appendix presents the **LL(1) parsing table** used by the YaLCC parser. The table encodes the decisions made by the parser when analyzing a stream of tokens generated by the lexer. Each row corresponds to a combination of a **non-terminal symbol** and an **input token**, specifying which production rule the parser should apply.

The table is implemented in the parser to allow **deterministic top-down parsing** with a single-token lookahead, ensuring efficient and correct syntactic analysis of YaLCC programs.

LL(1) Parsing Table

Non-Terminal	**Token(s) (Terminal)**	**Production**	**Rule #**
Program	PROG	Prog PROGNAME Is Code End	1
Code	VARNAME, IF, WHILE, PRINT, INPUT	Instruction ; Code	2
Code	END	ε	3
Instruction	VARNAME	Assign	4
Instruction	IF	If	5
Instruction	WHILE	While	6
Instruction	PRINT	Output	7
Instruction	INPUT	Input	8
Assign	VARNAME	VarName = ExprArith	9
If	IF	If { Cond } Then Code End	10
If	IF	If { Cond } Then Code Else Code End	11
While	WHILE	While { Cond } Do Code End	12
Output	PRINT	Print (VarName)	13
Input	INPUT	Input (VarName)	14
Cond	PIPE, VARNAME, NUMBER, LPAREN, MINUS	CondImpl	15
CondImpl	PIPE, VARNAME, NUMBER, LPAREN, MINUS	CondAtom → CondImpl	16
CondImpl	PIPE, VARNAME, NUMBER, LPAREN, MINUS	CondAtom	16*
CondAtom	PIPE	| Cond |	17
CondAtom	VARNAME, NUMBER, LPAREN, MINUS	CondComp	18
CondComp	VARNAME, NUMBER, LPAREN, MINUS	ExprArith Comp ExprArith	19
ExprArith	VARNAME, NUMBER, LPAREN, MINUS	ExprAddSub	20
ExprAddSub	VARNAME, NUMBER, LPAREN, MINUS	ExprMulDiv { (+|-) ExprMulDiv }*	21
ExprMulDiv	VARNAME, NUMBER, LPAREN, MINUS	ExprUnary { (*|/) ExprUnary }*	22
ExprUnary	MINUS	- ExprPrimary	23
ExprUnary	VARNAME, NUMBER, LPAREN	ExprPrimary	24
ExprPrimary	VARNAME	VarName	25
ExprPrimary	NUMBER	Number	26
ExprPrimary	LPAREN	(ExprArith)	27

B Operator Precedence and Associativity

The following table summarizes the **operator precedence** and **associativity** used by the YaLCC parser. These rules were applied when transforming the original grammar to remove ambiguity and correctly parse arithmetic and boolean expressions.

YaLCC Operator Precedence

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

The precedence order is from top (highest) to bottom (lowest). Right-associative operators are parsed from right to left, and left-associative operators are parsed from left to right.

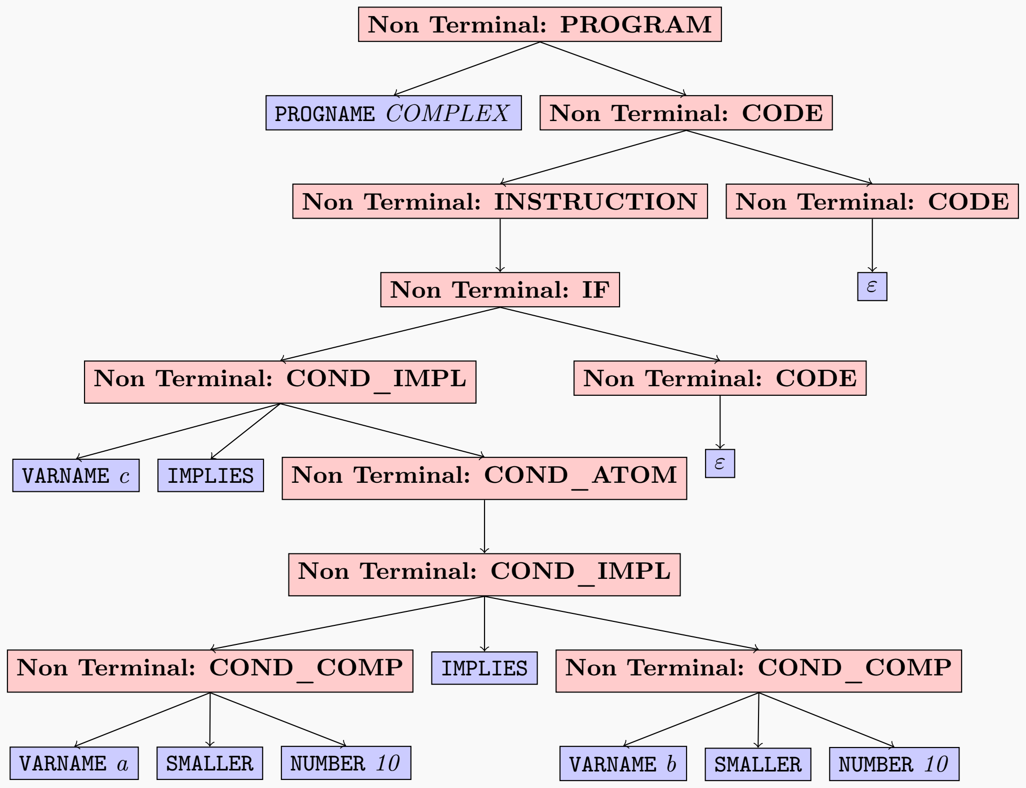
C Example YaLCC Programs and Parser Output

The following examples demonstrate the **YaLCC parser** applied to sample programs. For each program, we present the source code and the resulting parse tree.

Code 1

```
Prog COMPLEX Is
    If { c -> | a < 10 -> b < 10 | } Then
        End;
End
```

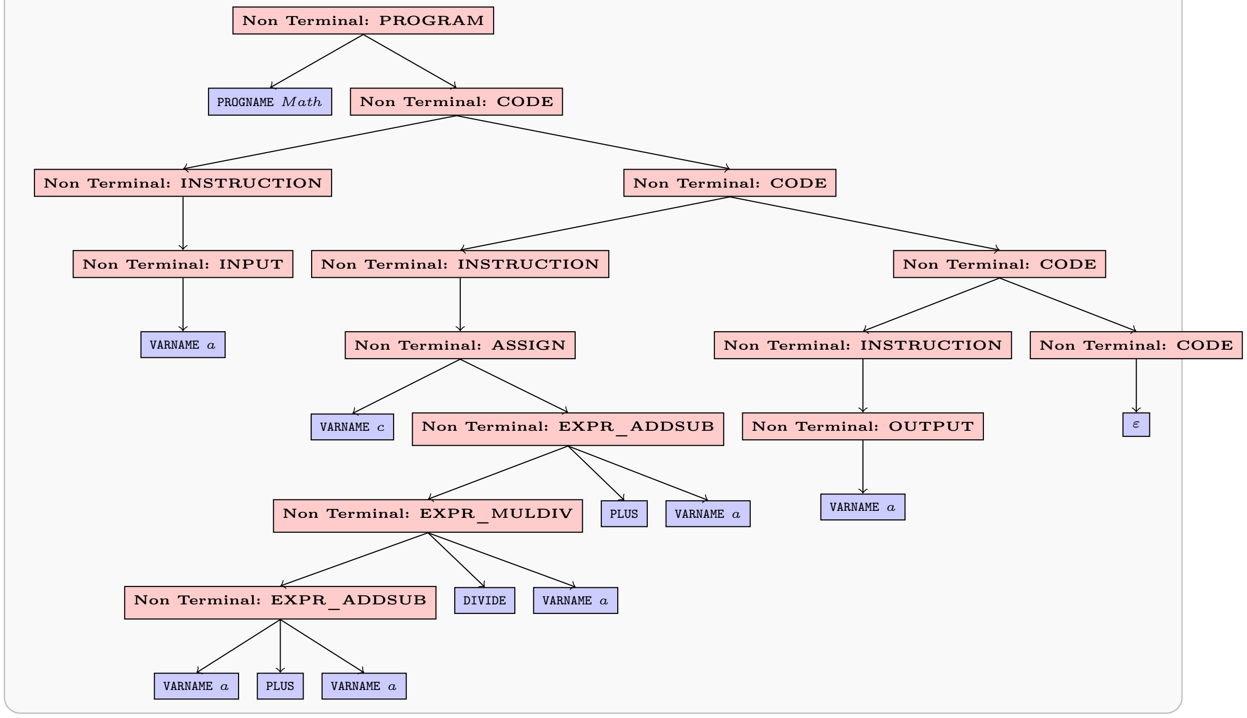
Resulting Parse Tree 1



Code 2

```
Prog Math Is
    Input(a);
    c = (a + a) / a + a;
    Print(a);
End;
```

Resulting Parse Tree 2



In both figures, the **blue nodes** represent **terminal symbols** (that is, `variable names`, punctuation symbols, `Progname`, and ε). The **red nodes** correspond to **nonterminal symbols**. In these examples, it is straightforward to verify that all **leaves** of the parse tree are **terminals**, while all **internal nodes** are **nonterminals**.