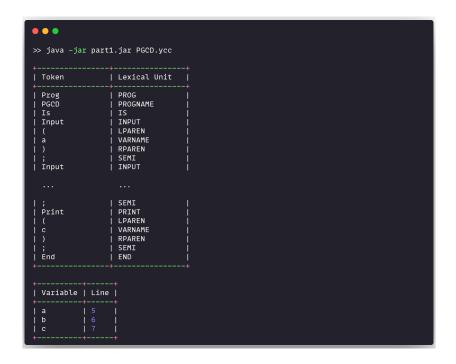
INFO-F-403

Introduction to language theory and compiling Project Report

Berthion Antoine - 566199 Bontridder Milan - 493709

15 October 2025



Abstract

This project implements the **lexical analysis phase** of a compiler for the toy language **YaLCC** (Yet Another Language for the Compiler Course). The **lexer** reads source code files and produces a stream of **tokens**, which can later be processed by a **parser**. This report details the **design and implementation** of the lexer, the **regular expressions** used for token recognition, example **input** and output files, and the **testing strategy**.

1 Introduction

The YaLCC lexer transforms source code into a well-defined sequence of tokens, accurately identifying keywords, variable names, numerical constants, operators, and comments. The objective of the lexical analysis phase is to systematically recognize and classify the fundamental units of the program that are relevant to the compiler.

Conceptually, the lexer functions as a "spelling checker", ensuring that the individual elements of the code are correctly identified. Subsequently, the **parser** serves as a "grammar checker", verifying whether these tokens are syntactically arranged to form meaningful constructs according to the rules of the language.

The YaLCC language provides the following features :

- Program declarations using Prog [ProgName] Is ... End
- Variable declarations and assignments, allowing the definition of identifiers for storing values
- Basic arithmetic expressions and operators (+, -, *, /)
- Conditional statements (If ... Then ... Else ... End)
- Loops (While ... Do ... End)
- Input and output statements (Input(...) and Print(...))
- Short (\$) and long (!!) comments, which are ignored by the parser

The complete symbol table, listing all recognized tokens, is provided in Appendix A.

2 Project Overview

The project is organized as follows :

- src/: Contains all source files necessary for the lexer implementation, including LexicalAnalyzer.flex, the generated LexicalAnalyzer.java, Symbol.java, LexicalUnit.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 part1.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 → symbol table
```

3 Regular Expressions

The **lexer** is implemented using **JFlex**, which generates a **nondeterministic finite automaton** (**NFA**) responsible for accepting valid **tokens** and rejecting invalid ones, while associating each accepted token with its corresponding **lexical unit** for subsequent processing by the **parser**. The NFA is then converted into a **deterministic finite automaton** (**DFA**) and optimized to minimize the number of states ¹, improving **efficiency** during lexical analysis.

To configure the JFlex-generated DFA, it is necessary to provide a set of **regular expressions**. These expressions define the patterns of valid **tokens**—such as **keywords**, **identifiers**, **numbers**, and **operators**—so that any input not matching one of these patterns is automatically rejected, ensuring that the lexer correctly identifies only the valid **lexical units**.

3.1 Specific Tokens (Keywords, Operators, and Symbols)

The lexer first identifies specific tokens corresponding to reserved keywords, operators, and punctuation. These are matched literally in the input stream. Examples include:

- Prog, Is, End denote the beginning and end of a program.
- If, Then, Else represent conditional statements.
- While, Do denote loops.
- Input, Print handle input/output operations.

^{1.} The DFA initially had 63 states and was optimized down to 54 states, improving efficiency during lexical analysis.

— Operators and punctuation symbols: +, -, *, /, =, ==, <, (,), ;.

These tokens are recognized by **exact string matches**, ensuring that reserved words and syntactic symbols are correctly identified before general pattern matching.

3.2 Pattern-Based Tokens (Identifiers, Numbers, Comments, Whitespace)

The following regular expressions are used to recognize variable identifiers, numbers, comments, and whitespace :

- ID_PROG = [A-Z][A-Za-z0-9_]*
 - Matches program identifiers that start with an uppercase letter followed by any combination (or None) of letters, digits, or underscores. This is case sensitive.
- $ID_VAR = [a-z][A-Za-z0-9_]*$
 - Matches variable identifiers beginning with a lowercase letter, followed by letters, digits, or underscores. This enforces case sensitivity and distinguishes variables from program identifiers.
- NUMBER = [0-9] +
 - Matches integer constants composed of one or more digits. Only positive integers are considered, and negative numbers are handled via the minus operator rule.
- WHITESPACE = $[\t r\n] +$
 - Matches sequences of spaces, tabs, and newline characters. These are ignored by the lexer and do not produce tokens, as they are not meaningful for the parser.
- SHORT_COMMENT = \\$.*
 - Matches short comments starting with a dollar sign and continuing to the end of the line. These are skipped by the lexer and do not contribute to the token stream.
- -- LONG_COMMENT = \!\!([^!]|!\+[^!])*?\!\!
 - Matches long comments enclosed between double exclamation marks. The pattern allows any content that does not include the closing !!, ensuring that nested comments are not accepted, as per the language specification.

Together, these regular expressions ensure that the lexer can accurately identify all valid lexical units while ignoring irrelevant or non-semantic content.

4 Lexer Design and Implementation

The YaLCC lexer is implemented in Java using JFlex. The core definition resides in LexicalAnalyzer.flex, which specifies the regular expressions and the corresponding actions executed for each matched token.

Upon recognizing a token, the lexer constructs a Symbol object containing the **token text**, its **lexical unit**, and its **position** (line and column) within the source file. These **Symbol** objects are then sequentially passed to the **parser** for **syntactic analysis**.

Key design considerations include:

- Modular Token Representation: Each lexical unit is encapsulated in a Symbol object, ensuring a consistent interface between the lexer and parser.
- Error Handling: Unrecognized or invalid characters trigger immediate error messages to System.err, including the precise location (yy) in the source code, facilitating debugging.
- Lexer-Parser Integration: Tokens are returned in the order they appear in the source file, preserving program structure and enabling deterministic parsing without ambiguity.

This design ensures the **lexer** is **efficient**, **maintainable**, and **robust**, providing a solid foundation for the subsequent **parsing** and **semantic analysis** phases.

5 Sample Programs and Lexical Analysis Results

To evaluate our YaLCC lexer, we executed it on various .ycc programs to analyze its output behavior. Additionally, we employed unit testing to ensure that our lexical analyzer functioned as intended. For further details on the unit testing methodology, see Section 6.

Several of the .ycc files used for testing are provided in the **Appendix B**. The primary objective was to observe how the lexer responded to specific situations, including, but not limited to : excessive or irregular whitespace, invalid tokens, extensive long comments, syntactically incorrect code, and complex sequences of tokens.

6 Lexer Verification via Unit Tests

The lexical analyzer has been rigorously tested using **JUnit 5**. The test suite is designed to verify that the lexer correctly identifies tokens, handles errors, and maintains a consistent symbol table. Key aspects of the testing strategy include:

- **Lexical Unit Validation**: Tests confirm that each token is correctly classified as a keyword, identifier, number, operator, or punctuation symbol, ensuring accurate token recognition.
- Error Handling: Tests provide input containing invalid or unrecognized characters to verify that the lexer reports errors with precise location information, in line with the design requirements.
- Comment and Edge Case Handling: Input files with short (\$) and long (!!) comments, empty files, or unusual sequences (such as consecutive semicolons) are tested to ensure the lexer correctly ignores irrelevant content and maintains proper token sequencing.
- Symbol Table Integrity: Tests verify that the symbol table correctly records all variable identifiers in lexicographical order, along with the line numbers of their first occurrence, ensuring both completeness and ordering consistency.

The complete test suite can be executed using the command make test, providing automated verification of lexer correctness and robustness.

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 author.

8 Conclusion

In this project, we successfully implemented the lexical analysis phase of the YaLCC compiler. The lexer accurately identifies keywords, variable names, numerical constants, operators, and both short and long comments, converting source code into a well-defined sequence of tokens. By leveraging JFlex and carefully designed regular expressions, the lexer ensures correctness, efficiency, and robust error handling.

Comprehensive testing using **JUnit** confirmed that the **lexer** handles valid inputs, rejects invalid characters, and maintains a consistent **symbol table**, forming a solid foundation for the subsequent **parsing** and **semantic analysis** phases. This project demonstrates the critical role of **lexical analysis** in the compilation process and provides a **modular**, **maintainable** implementation suitable for extension in future **compiler stages**.

A Symbol Table (YaLCC Grammar)

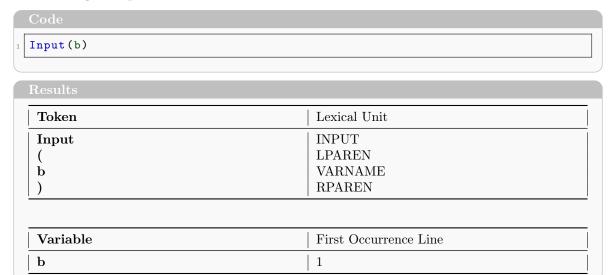
The following table presents the YaLCC grammar. Non-terminal symbols are enclosed in angle brackets $\langle \cdot \rangle$, terminal symbols are written in plain text, and ε denotes the empty string.

aLCC Grammar		
Non-terminal	Production	
(Program)	Prog [ProgName] Is (Code) End	
$\langle \text{Code} \rangle$	$\langle \text{Instruction} \rangle$; $\langle \text{Code} \rangle$	
$\langle \text{Code} \rangle$	ε	
$\langle Instruction \rangle$	$\langle \mathrm{Assign} \rangle$	
$\langle Instruction \rangle$	$\langle \mathrm{If} angle$	
$\langle Instruction \rangle$	$\langle \mathrm{While} \rangle$	
$\langle Instruction \rangle$	$\langle \mathrm{Call} \rangle$	
$\langle {\rm Instruction} \rangle$	(Output)	
$\langle {\rm Instruction} \rangle$	$\langle ext{Input} angle$	
$\langle \mathrm{Assign} \rangle$	$[VarName] = \langle ExprArith \rangle$	
$\langle \mathrm{ExprArith} \rangle$	[VarName]	
$\langle \text{ExprArith} \rangle$	[Number]	
$\langle \text{ExprArith} \rangle$	((ExprArith))	
$\langle \text{ExprArith} \rangle$	- (ExprArith)	
$\langle \text{ExprArith} \rangle$	$\langle \text{ExprArith} \rangle \langle \text{Op} \rangle \langle \text{ExprArith} \rangle$	
$\langle \mathrm{Op} \rangle$	+	
$\langle \mathrm{Op} \rangle$	-	
$\langle \mathrm{Op} \rangle$	*	
$\langle \mathrm{Op} \rangle$	/	
$\langle \mathrm{If} \rangle$	If (Cond) Then (Code) End	
$\langle \mathrm{If} \rangle$	If (Cond) Then (Code) Else (Code) End	
$\langle \text{Cond} \rangle$	$\langle \text{Cond} \rangle \to \langle \text{Cond} \rangle$	
$\langle \text{Cond} \rangle$	$ \langle \operatorname{Cond} \rangle $	
$\langle \text{Cond} \rangle$	$\langle \text{ExprArith} \rangle \langle \text{Comp} \rangle \langle \text{ExprArith} \rangle$	
$\langle \text{Comp} \rangle$	==	
$\langle \mathrm{Comp} \rangle$	<=	
$\langle \text{Comp} \rangle$	<	
$\langle \text{While} \rangle$	While (Cond) Do (Code) End	
$\langle \text{Output} \rangle$	Print([VarName])	
$\langle \text{Input} \rangle$	Input([VarName])	

B Example YaLCC Programs and Lexer Output

The following examples demonstrate the **YaLCC lexer** applied to sample programs. For each program, we present the source code, the resulting lexical tokens, and the symbol table.

B.1 Single Input



B.2 Statement with spaces

Token	Lexical Unit
(LPAREN
If	IF
abbb	VARNAME
Then	THEN
x	VARNAME
=	ASSIGN
3	NUMBER
;	SEMI

Variable	First Occurrence Line
abbb	2
x	5