


UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ingeniería

Computer Engineering
Compilers
Lexer

Students:

320317460
423112645
423002205
320262614

Team: 09

Professor:

Rene Adrián Dávila Pérez

Group 5, Semester 2025-2

México, CDMX

Deadline:

13 de marzo de 2025

Contents

1 Introduction

1.1 Problem Statement

A compiler is a fundamental tool in computer science that translates source code written in a high-level language into a lower-level language, such as machine code or assembly. The compilation process consists of several stages, with the analysis phase being one of the most crucial. Within this phase, the lexical analyzer, or scanner, is responsible for processing the input code and breaking it down into a stream of tokens. These tokens serve as the fundamental building blocks for later stages, such as syntax and semantic analysis.

The main challenge in lexical analysis is correctly identifying lexemes while handling variations in syntax, whitespace, and comments. Additionally, an efficient lexical analyzer must minimize processing time and ensure error detection at an early stage. Without a properly functioning lexical analyzer, subsequent compilation phases may fail, leading to incorrect code translation or unexpected behavior.

1.2 Motivation

The correct design of a lexical analyzer is crucial for the development of any programming language compiler or interpreter. This phase significantly influences the performance and accuracy of the overall compilation process. A poorly implemented lexical analyzer can introduce inefficiencies, misinterpret tokens, or fail to handle lexical errors effectively, impacting the entire compilation pipeline.

By studying and implementing a lexical analyzer, we gain a deeper understanding of how programming languages are processed. This knowledge is valuable not only for compiler design but also for applications such as code editors, interpreters, and static analysis tools. Additionally, exploring lexical analysis helps reinforce concepts in formal languages, automata theory, and regular expressions.

1.3 Objectives

- Identify the role and importance of lexical analysis within the compilation process.
- Design and implement a lexical analyzer capable of recognizing tokens in a given programming language.
- Utilize regular expressions and finite automata to define and detect lexical patterns.
- Optimize the efficiency and correctness of the lexical analyzer to ensure accurate tokenization.

2 Theoretical Framework

To develop a lexical analyzer, it is essential to understand several fundamental concepts related to compiler design, formal languages, and automata theory. This section provides an overview of the key theoretical foundations applied in this project.

2.1 Compilers and Their Phases

A compiler is a software system that translates a program written in a high-level programming language into machine code or an intermediate representation. The compilation process consists of multiple phases, which can be categorized into **analysis and synthesis**. The analysis phase breaks down the source code into a structured representation, while the synthesis phase generates the final translated code.

The **lexical analysis** phase is the first step in this process, responsible for converting the input source code into a sequence of tokens. These tokens serve as the building blocks for the subsequent phases: **syntax analysis**, **semantic analysis**, **intermediate code generation**, **optimization**, and **target code generation** [1].

2.2 Lexical Analysis and Tokenization

Lexical analysis is the process of scanning the input source code and recognizing meaningful sequences of characters called **lexemes**, which are then categorized into **tokens**. Tokens represent basic elements of the language, such as **keywords**, **identifiers**, **literals**, **operators**, and **punctuation symbols**.

The lexical analyzer must also handle **whitespace and comments** to facilitate a smooth transition to the syntax analysis phase. To achieve this, lexical analyzers often use **finite automata** and **regular expressions** to define and recognize patterns in the input text.

2.3 Formal Languages and Automata

Lexical analysis relies heavily on concepts from formal language theory, specifically **regular languages**, which can be recognized using **finite state machines**. The theoretical foundations of this approach include:

- **Regular Expressions (REs)**: A notation used to define patterns in a language. Each regular expression corresponds to a **regular language** [2].
- **Finite Automata (FA)**: A computational model used to process and recognize regular languages. These include **deterministic finite automata (DFA)** and **non-deterministic finite automata (NFA)**.
- **Lexical Specification**: The process of formally defining the token categories using regular expressions and then converting them into a DFA for efficient scanning.

Lexical analyzers typically use **lexical specification tools**, such as **Flex** (which we are not allowed to use), to automate the generation of scanners from regular expressions.

2.4 Error Handling in Lexical Analysis

One of the primary responsibilities of a lexical analyzer is detecting and handling errors efficiently. Common lexical errors include:

- Unexpected characters that do not match any token definition.
- Unterminated strings or comments.
- Invalid numerical formats.

Although error handling is an important aspect of lexical analysis, **this implementation does not include mechanisms for lexical error detection or recovery**. Instead, the analyzer assumes that the input source code follows the expected lexical structure, focusing solely on token recognition. Future improvements could incorporate error detection strategies such as skipping invalid tokens, generating meaningful error messages, or integrating error recovery techniques [3].

2.5 Integration with the Compilation Process

The lexical analyzer serves as the **first component of the compiler pipeline**, providing structured token sequences for the **parser** (syntax analyzer). The accuracy and efficiency of the lexical analyzer directly impact the performance of the entire compiler.

A well-implemented lexical analyzer enhances code readability, facilitates debugging, and ensures proper interpretation of the source code. Therefore, its design must balance **speed**, **accuracy**, and **robust error handling**.

3 Development

3.1 Automaton

The implementation is based on a custom-designed automaton, which represents the fundamental structure of the Lexer and defines how tokens are recognized and categorized. This automaton models the transitions between different states based on input characters, allowing the lexical analyzer to efficiently classify keywords, identifiers, operators, constants, and other elements of the language. For more details on the automaton, refer to **Appendix .1**.

3.2 Implementation

The implementation of the lexical analyzer was carried out in Java, focusing on the use of **regular expressions** and **finite automat** to recognize different types of tokens in a given input source code. The project consists of three main components: the **Lexer**, the **Token** class, and the **Program** class.

3.2.1 Lexer

The `Lexer` class is responsible for scanning the input string and identifying tokens based on predefined **regular expressions**. It recognizes different types of tokens, including:

- **Keywords**: Reserved words such as `if`, `else`, `while`, `return`, etc.
- **Identifiers**: Variable names, function names, etc.
- **Operators**: Arithmetic, logical, and assignment operators such as `+`, `-`, `=`, `==`, etc.
- **Constants**: Numeric values, including integers and floating-point numbers.
- **Literals**: String literals enclosed in double quotes.
- **Punctuation**: Symbols such as parentheses, commas, semicolons, brackets, etc.
- **Whitespace and Comments**: Spaces, tabs, newlines, and comments are identified but ignored in token processing.

The lexer processes the input using Java's regex API, applying a pattern-matching approach to extract tokens while ignoring whitespace and comments. It also keeps track of the number of occurrences of each token.

3.2.2 Token Class

The `Token` class defines an enumeration of token types and provides a structure for storing tokens. Each token consists of:

- A **type**, which classifies the token (e.g., keyword, identifier, operator).
- A **value**, which stores the actual lexeme recognized.

This class enables easy categorization and retrieval of tokens for further analysis.

3.2.3 Program Execution

The `Program` class serves as the entry point for executing the lexical analyzer. It performs the following tasks:

1. Opens a file selection dialog to allow the user to choose a source code file.
2. Reads the content of the selected file into a string.
3. Passes the string to the `Lexer` class for tokenization.
4. Prints the categorized tokens along with their respective occurrences.
5. Prints the total occurrences.

3.3 Tests and Results

To validate the functionality of the lexical analyzer, several test cases were conducted using input files containing C-like syntax. The results were analyzed by comparing the expected and actual output.

3.3.1 Test Case 1: Basic C Code

Input:

```
  int main() {
//comments
      int x = 10;
      float y = 5.5;
/*
several comments
*/
      if (x > y) {
          return x;
      } else {
          return y;
      }
// comments
  }

```

Expected Output:

- Keywords: `int[2]`, `float[1]`, `if[1]`, `return[2]`, `else[1]`
- Identifiers: `main[1]`, `x[2]`, `y[3]`
- Operators: `=[2]`, `>[1]`
- Constants: `10[1]`, `5.5[1]`
- Punctuators: ([2],)[2], {[3], ;[4], }[3]
- **Total tokens found**: 33

Actual Output: The results obtained matched the expected output, confirming correct token recognition.

3.3.2 Test Case 2: While Loop

Input:

```
        int main() {
            int num = 10;
            int factorial = 1;

            while (num > 1) {
                factorial *= num;
                num--;
            }

            printf("Factorial: %d\n", factorial);
            return 0;
        }

```

Expected Output:

- **Keywords**: int[3], while[1], return[1]
- **Identifiers**: main[1], num[4], factorial[3], printf[1]
- **Operators**: =[2], >[1], *=[1], –[1]
- **Constants**: 10[1], 1[2], 0[1]
- **Punctuators**: ([3],)[3], {[2], ;[6], }[2], ,[1]
- **Literal**: ”Factorial: %d \n”[1]
- **Total tokens found**: 41.

Actual Output: The lexical analyzer correctly identified all tokens, including keywords, operators, and numerical constants.

3.3.3 Test Case 3: Print Statement

Input:

```
    int main()
    {
        printf("Introducci n a la programaci n en lenguaje C\n");
        return 0;
    }

```

Expected Output:

- **Keywords**: int[1], return[1]
- **Identifiers**: main[1], printf[1]
- **Punctuators**: ([2],)[2], {[1], ;[2], }[1]
- **Literal**: ”Introducción a la programación en lenguaje C\n”[1]
- **Constant**: 0[1]
- **Total tokens found**: 14.

Actual Output: The lexical analyzer correctly identified all tokens, including keywords, function calls, literals, and punctuation.

3.3.4 Limitations

Although the lexer successfully tokenizes input code, it does not handle **error detection** for unrecognized symbols, incorrect numeric formats, or unterminated string literals. These limitations could be addressed in future improvements by incorporating error-handling mechanisms.

3.4 Grammar

3.4.1 Initial Grammar

$$
\begin{aligned}
S &\rightarrow Token\ S \mid \epsilon \\
Token &\rightarrow Keyword \mid Identifier \mid Punctuator \mid Literal \mid Constant \\
Keyword &\rightarrow int \mid return \\
Identifier &\rightarrow main \mid printf \\
Punctuator &\rightarrow (\mid) \mid \{ \mid \} \mid ; \\
Literal &\rightarrow \text{"}(Character)^*\text{"} \\
Constant &\rightarrow 0
\end{aligned}
$$

3.4.2 Example: Target Code

```c
int main()
{
    printf("Introducci n a la programaci n en lenguaje C\n");
    return 0;
}

```

Applying the grammar:

$S \rightarrow Token\ S$ (Repeat 14 times)

$\rightarrow Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ S$

$\rightarrow Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ Token\ \epsilon$

$\rightarrow Keyword\ Identifier\ Punctuator\ Punctuator\ Punctuator\ Identifier\ Punctuator\ Literal$
$Punctuator\ Punctuator\ Keyword\ Constant\ Punctuator\ Punctuator$

$\rightarrow int\ main\ (\ )\ \{\ printf\ (\ \text{"}Introducci\acute{o}n\ a\ la\ programaci\acute{o}n\ en\ lenguaje\ C\text{"}\ )\ ;\ return\ 0\ ;\}$

3.4.3 Left Recursion Case

The previous grammar does not have a left recursion or left factoring problem. However, for the sake of demonstrating the transformation process, let's consider the following grammar:

$$
\begin{aligned}
S &\rightarrow S\ Token \mid \epsilon \\
Token &\rightarrow Keyword \mid Identifier \mid Punctuator \mid Literal \mid Constant \\
Keyword &\rightarrow Keyword\ \text{"}Keyword\text{"} \mid int \mid return \\
Identifier &\rightarrow main \mid printf \\
Punctuator &\rightarrow (\mid) \mid \{ \mid \} \mid ; \\
Literal &\rightarrow \text{"}(Character)^*\text{"} \\
Constant &\rightarrow 0
\end{aligned}
$$

We can observe that there is a **left recursion problem** in 'S' and 'Keyword'. Left recursion can cause an **infinite loop**, so we need to eliminate it.

Using the transformation:

$$\begin{aligned} A &\to A\alpha \mid \beta \\ A &\to \beta A' \\ A' &\to \alpha A' \mid \epsilon \end{aligned}$$

Applying this to the grammar:

$$\begin{aligned} S &\to \epsilon S' \\ S' &\to Token\ S' \mid \epsilon \end{aligned}$$

For the 'Keyword' rule:

$$\begin{aligned} Keyword &\to (int \mid return)\ Keyword' \\ Keyword' &\to \text{"}Keyword\text{"}\ Keyword' \mid \epsilon \end{aligned}$$

Now, the grammar is **free of left recursion** and can be **efficiently parsed using an LL(1) parser**.

3.4.4 Left Factoring Case

Given the following grammar:

$$\begin{aligned} S &\to Token\ S \mid \epsilon \\ Token &\to Keyword'\ int \mid Keyword'\ return \mid Identifier\ main \mid Identifier\ printf \mid Punctuator \mid Literal \mid Constant \\ Punctuator &\to (\mid) \mid \{ \mid \} \mid ; \\ Literal &\to \text{"}(Character)^*\text{"} \\ Constant &\to 0 \\ Keyword' &\to \epsilon \\ Identifier' &\to \epsilon \end{aligned}$$

The second rule presents an opportunity to apply **left factoring**.
Using the transformation:

$$\begin{aligned} A &\to \alpha\beta \mid \alpha\gamma \\ A &\to \alpha X \\ X &\to \beta \mid \gamma \end{aligned}$$

Applying this transformation:

$$\begin{aligned} Token &\to Keyword'X \mid Identifier'Y \mid Punctuator \mid Literal \mid Constant \\ X &\to int \mid return \\ Y &\to main \mid printf \\ Keyword' &\to \epsilon \\ Identifier' &\to \epsilon \end{aligned}$$

Thus, the final grammar, after applying **left factoring**, is:

$$
\begin{aligned}
S &\rightarrow Token\ S \mid \epsilon \\
Token &\rightarrow Keyword'X \mid Identifier'Y \mid Punctuator \mid Literal \mid Constant \\
X &\rightarrow int \mid return \\
Y &\rightarrow main \mid printf \\
Keyword' &\rightarrow \epsilon \\
Identifier' &\rightarrow \epsilon \\
Punctuator &\rightarrow (\mid) \mid \{ \mid \} \mid ; \\
Literal &\rightarrow \text{"}(Character)^*\text{"} \\
Constant &\rightarrow 0
\end{aligned}
$$

4 Results

4.1 Test Case 1: Basic C Code

The results for this first test case were exactly as expected. The lexer correctly identified all token categories presented in the example, ensuring that no expected tokens were omitted. Additionally, the classification of each token matched the predefined categories, demonstrating the accuracy of the lexical analysis process. A simple example that includes keywords such as `if`, `return`, `int`, `float`, and `else`; along with constants, identifiers, operators, punctuators, and the total count of tokens. It also proves the capability of the lexer to manage comments.

```
C:\Users\danyh\LexicalAnalyzer>java -cp bin unam.fi.compilers.g5.E09.Lexer.P
rogram
KEYWORD: int[2], float[1], if[1], return[2], else[1]
IDENTIFIER: main[1], x[3], y[3]
OPERATOR: =[2], >[1]
CONSTANT: 10[1], 5.5[1]
PUNCTUATOR: ([2], )[2], {[3], ;[4], }[3]

Total tokens found: 33
```

Figure 4.1: Test Case 1: Results

4.2 Test Case 2: While Loop

The results for this second test case were exactly as expected. The lexer correctly identified all token categories presented in the example, ensuring that no expected tokens were omitted. Additionally, the classification of each token matched the predefined categories, demonstrating the accuracy of the lexical analysis process. In this case, we tested additional tokens; specifically, this test includes the keyword `while`, various new operators such as `*=`, `--`, and `>`, and also evaluates the literal category.

```
C:\Users\danyh\LexicalAnalyzer>java -cp bin unam.fi.compilers.g5.E09.Lexer.Program
KEYWORD: int[3], while[1], return[1]
IDENTIFIER: main[1], num[4], factorial[3], printf[1]
OPERATOR: =[2], >[1], *=[1], --[1]
CONSTANT: 10[1], 1[2], 0[1]
LITERAL: "Factorial: %d\n"[1]
PUNCTUATOR: ([3], )[3], {[2], ;[6], }[2], ,[1]

Total tokens found: 41
```

Figure 4.2: Test Case 2: Results

4.3 Test Case 3: Print Statement

The results for this third test case were exactly as expected. The lexer correctly identified all token categories presented in the example, ensuring that no expected tokens were omitted. This is a very simple example that verifies the results of the previous test cases, but it is perfect for demonstrating our knowledge about generating grammars and optimizing them.

```
C:\Users\danyh\LexicalAnalyzer>java -cp bin unam.fi.compilers.g5.E09.Lexer.Program
KEYWORD: int[1], return[1]
IDENTIFIER: main[1], printf[1]
CONSTANT: 0[1]
LITERAL: "Introducción a la programación en lenguaje C\n"[1]
PUNCTUATOR: ([2], )[2], {[1], ;[2], }[1]

Total tokens found: 14
```

Figure 4.3: Test case 3: Results

5 Conclusion

The development of the lexical analyzer demonstrated the practical application of key theoretical concepts, such as formal languages, finite automata, and regular expressions, in the process of tokenizing source code. The implementation of a custom lexer successfully recognized and categorized tokens, proving the effectiveness of using **pattern-matching techniques** for lexical analysis.

By designing and testing different **context-free grammars (CFGs)**, we reinforced the importance of **left factoring** and **right recursion** to ensure an efficient and unambiguous grammar structure. The removal of **left recursion** prevented infinite recursion issues, making the grammar more suitable for **top-down parsing methods**, while **left factoring** helped eliminate ambiguity and improved the predictability of parsing decisions. These transformations ensured that the grammar could be effectively used in an LL(1) parser.

The conducted test cases confirmed the correctness of the lexer, as all expected tokens were identified accurately. The ability of the lexer to recognize **keywords, identifiers, literals, constants, operators, and punctuators** validates the theoretical foundation applied in its construction. Additionally, the integration of **deterministic finite automaton (DFA)** played a crucial role in modeling the state transitions necessary for token classification.

Furthermore, the **objectives established at the beginning of this work were successfully achieved**. The implementation correctly identified and classified tokens, demonstrating a clear understanding of the **lexical analysis process**. Additionally, the use of **regular expressions and finite automata** effectively defined lexical patterns, and the grammar transformations, including **left factoring and left recursion elimination**, ensured that the structure remained **efficient and parseable**.

Although the lexical analyzer does not include **error detection mechanisms**, its modular structure allows for future enhancements, such as **handling invalid tokens, identifying malformed constants, and implementing error recovery strategies**. This demonstrates how lexical analysis is an **essential** yet **evolving** phase in compiler design, setting the foundation for subsequent **syntax and semantic analysis stages**.

In summary, the successful implementation of the lexer highlights the **practical application of compiler theory** and its role in software development. Future improvements could focus on **optimizing token recognition speed, expanding token categories, and integrating with a syntax analyzer** to further extend the compilation pipeline.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd. Addison-Wesley, 2006, ISBN: 978-0321486813.

[2] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd. Addison-Wesley, 2006, ISBN: 978-0321455369.

[3] A. V. Aho, *Foundations of Compiler Design*. Pearson, 1995, ISBN: 978-0201403851.

.1 Automaton

The automaton used in the lexical analyzer can be found in the following repository:

GitHub Repository: `https://github.com/D0CTT/LexicalAnalyzer/tree/main/Automaton`

Imgur: `https://i.imgur.com/cvJ1zfc.jpeg`

This resource contains the formal representation of the finite automaton used for token recognition. It is also worth mentioning that the automaton was created using the JFLAP tool. However, it is purely a technical representation that neither functions nor can be tested, but it effectively illustrates the concept of how the lexer operates.