

NATIONAL AUTONOMOUS UNIVERSITY OF
MEXICO

FACULTY OF ENGINEERING



Computer Engineering

Compilers

Project 1: Lexical Analyzer

Professor: Eng. Rene Adrian Davila Perez

Semester: 2025-1

Team Members:

319050390

319103795

319024045

319240731

319321935

Group: 5

SEPTEMBER 10, 2024

Contents

1	Introduction	3
2	Theoretical Framework	3
2.1	Key Concepts	3
2.2	Token Categories	4
2.3	Stages of a Lexical Analyzer	5
3	Development	5
3.1	Deterministic Finite Automaton (DFA)	6
3.1.1	Automata Definitions	6
3.2	Abstracting Lexers	8
3.3	Interface with Tkinter	9
3.4	Parse Tree	10
4	Results	11
5	Conclusions	12
6	References	12

1 Introduction

What is the importance of a lexical analyzer?

When we talk about a lexical analyzer, we start with the concept of it being a phase in the compilation process of a program. It has the function of classifying substrings of an input file into sets of tokens that are previously defined by regular expressions. However, it only answers one question: *What is it?* and not the purpose it has in its broad use. Now, to explain its importance, let's look at another environment: a library, where there are many books of different types and the main idea is to quickly find the categories that make a specific book part of a set that shares similar content. If there were no tags and no person in charge of classifying these books, the search would be chaotic, and the library system would have problems.

Starting from this point, a lexical analyzer seeks to generate a catalog of tokens that can be viewed quickly, clearly ignoring blank spaces or comments. So that when a problem arises when reading a string and its classification cannot be distinguished, the analyzer, based on regular expressions, makes decisions to find the category to which it belongs.

The idea of automating the process of such classification (*tokenization*) is fundamental for the development of this project, since we do not prioritize the primary operation of the lexical analyzer. Instead, through the help of high-level language tools such as Python, we create a link between the tokenization process and the user, in such a way that it is not necessary to take into account the theory of how said process works internally and can quickly experience an automatic response of the classification of the input substring through the use of some file.

Finally, the objectives proposed to be developed throughout this report and which are part of the main concept by which this project was initiated, contemplate:

1. Exploring the essential role of a lexical analyzer in the compilation process by implementing a basic lexical analyzer in a programming language.
2. Designing and coding a tool that identifies and classifies tokens such as keywords, operators, literals, and identifiers from source code using regular expressions and finite automata.

2 Theoretical Framework

In order for the lexical analyzer to achieve the goal of splitting the input into parts, it must be able to decide for each of those parts whether it is a separate component and, if so, of what type. The lexical analyzer collects information about lexical components in their associated attributes. Lexical components influence parsing decisions, and attributes influence the translation of lexical components.

2.1 Key Concepts

- **Lexemes:** Sequence of characters in the source program that match the pattern for a token and that the lexical analyzer identifies as an instance of that token.
- **Tokenization:** The process of classifying lexemes into tokens using a set of rules that determine which lexemes form valid tokens in the programming language.

- **Regular Expressions:** Description of the form that token lexemes can take. In the case of a keyword as a token, the pattern is simply the sequence of characters that make up the keyword. For identifiers and some other tokens, the pattern is more complex and can match many strings.
- **Finite Automata:** Finite automata are like mathematical machines used to model how a lexer processes text. They have a limited number of states and transition between these states based on the characters they read from the source code. Depending on the sequence of these state transitions, finite automata determine whether a lexeme (a sequence of characters) is valid or not.
 - **Deterministic Finite Automata (DFA):** In a DFA, each state and input character combination leads to exactly one next state. This makes it predictable and straightforward to implement.
 - **Nondeterministic Finite Automata (NFA):** An NFA can transition to multiple possible states from a given state and input character. This allows for more flexibility in pattern recognition, though it can be more complex to implement. Every NFA can be converted into an equivalent DFA.
 - **Epsilon-NFA (ϵ -NFA):** This variant of NFA includes transitions that occur without consuming any input characters, known as epsilon (ϵ) transitions. This adds flexibility in handling patterns where transitions can happen without specific input.
- **Symbol Table:** This table holds important information during tokenization. It maps token types, positions, and other metadata.
- **Lexical Category:** The concept of a lexical category arises naturally and represents a type of elementary symbol in the programming language, such as identifiers, keywords, integers, etc.
- **Token:** A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol that represents a type of lexical unit, such as a specific keyword or a sequence of input characters denoting an identifier. Token names are the input symbols processed by the non-deterministic parser. From this point on, we will generally write the name of a token in bold, and we will often refer to a token by its name.

2.2 Token Categories

Some typical lexical category families of programming languages are:

- **Keywords:** Words with a specific meaning in the language. Examples of keywords in C are `while`, `if`, `return`, etc. Each keyword usually corresponds to a distinct category. Keywords are usually reserved. If they are not, the parser will need syntactic information to resolve the ambiguity.
- **Identifiers:** Names of variables, functions, user-defined types, etc. Examples of identifiers in C are `i`, `x10`, `read_value`, etc.

- **Operators:** Symbols that specify arithmetic, logical, string, and other operations. Examples of operators in C are `+`, `*`, `/`, `%`, `==`, `!=`, `&&`, etc.
- **Constants:** Numeric values like integers (decimal, octal, hexadecimal), floating-point numbers, etc. Examples of constants in C are `928`, `0xF6A5`, `83.3E+2`, etc.
- **Literals:** Characters or string values. Examples of string literals in C are `"a string"`, and examples of character literals are `'x'`, `'\0'`.
- **Special Characters:** Separators, delimiters, and terminators. Examples in C include `{`, `}`, `;`, etc. Each usually belongs to a distinct category.
- **Punctuation:** Characters such as commas, semicolons, and braces, used to separate expressions or define code blocks.

Understanding these token categories is essential for developing a lexical analyzer, which will be further discussed in the development section.

2.3 Stages of a Lexical Analyzer

The process of lexical analysis generally follows these key stages:

1. **Reading the Source Code:** The lexical analyzer reads through the code one character at a time, carefully going through the text to make sure nothing is missed.
2. **Pattern Recognition:** Using predefined rules—often based on regular expressions—the analyzer identifies patterns in the code. These patterns help the analyzer decide what each part of the code represents, such as whether it's a keyword, an identifier, or something else.
3. **Token Generation:** After recognizing a pattern, the analyzer converts it into a token. Think of tokens as small, easy-to-handle pieces of the code that the compiler will later use to understand and process the program.
4. **Handling Lexical Errors:** Sometimes, the analyzer may come across something unexpected—like a sequence of characters that doesn't fit any of the rules. When this happens, it reports a lexical error, signaling that there's something in the code that doesn't belong.

3 Development

In developing our token table, we chose Python for its ease of use and speed. We used Tkinter to create a graphical interface that displays the token table in a user-friendly manner.

A key part of our development process involved working with regular expressions (regex). Regex is an invaluable tool for defining patterns used in tokenization. We used regex to identify and categorize different types of tokens in the source code. This enabled us to efficiently match patterns for keywords, identifiers, operators, literals, and other components of the language.

From our perspective as developers, Python was the perfect choice. It's a language we're comfortable with, and it allowed us to implement the lexical analyzer quickly and effectively. Our main goal was to create an interface that would be intuitive for users who aren't familiar with the technical details of tokenization. We aimed to ensure that even those with no background in the program's inner workings could easily navigate the interface and understand the tokenization process.

In addition, Python's 're' library played a crucial role. It allowed us to generate token categories that suited our needs, particularly for reading C language files. Thanks to this intuitive library, abstracting the functionality of regular expressions was straightforward. The key was to understand where to direct our efforts and consult the Python documentation to find the best way to achieve our objectives.

Overall, we focused on building a tool that not only functions effectively but also provides a pleasant and accessible experience for all users.

3.1 Deterministic Finite Automaton (DFA)

We have implemented a Deterministic Finite Automaton (DFA) to identify various tokens such as identifiers, punctuation marks, operators, literals, and special characters.

This implementation provides a fundamental structure for developing the regular expressions that will guide the lexical analysis process. Specifically, these regular expressions will determine the paths and variants based on the patterns found in the substrings of the input file. By leveraging finite automata, we minimize ambiguities and gain the flexibility to modify the order in which regular expressions are analyzed, thereby achieving a more comprehensive tokenization.

Additionally, the DFA helps in evaluating potential risks and managing errors that may arise during the classification process.

3.1.1 Automata Definitions

Since we seek to build a robust lexical analyzer, we designed several specialized automata to address different types of tokens that our system needs to recognize. Each automaton is designed to handle a specific category (literals, operators, identifiers, and punctuation marks), each with its own unique set of challenges and quirks. In this way, we sought to detail each possible value of the automaton and obtain the best possible result for the analysis

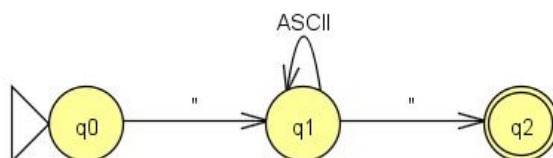


Figure 1: Automaton for literals

Literals Our automaton for literals first evaluates whether the character that follows matches the regular expression starting with quotation marks. So it doesn't matter what the content is as long as it starts and ends with " "

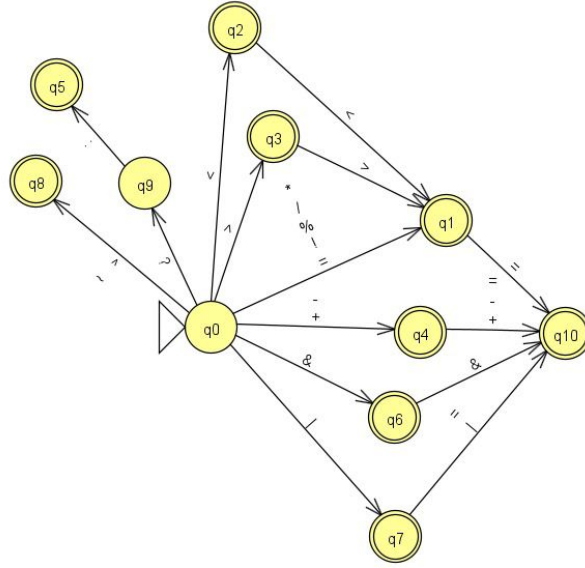


Figure 2: Automaton for operators

Operators The operator automaton is our preferred choice for recognizing symbols like ‘+’, ‘-’, ‘*’, and ‘=’. This part of the system needed some finesse to ensure it could handle simple and compound operators (e.g., ‘+=’, ‘==’). We faced challenges in distinguishing operators from other types of tokens and ensuring that complex expressions were parsed correctly. It’s all about getting precedence and associativity right, which required some careful tuning and testing. Likewise, we were able to detail the order in which regular expressions should be analyzed so as not to confuse or evaluate a classification without having a clear understanding of its concept or function within the program

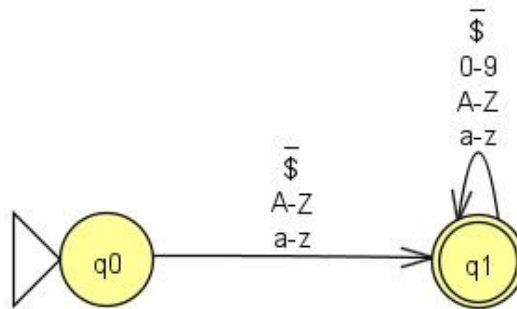


Figure 3: Automaton for identifiers

Identifiers When it comes to identifiers, we are talking about variable names and user-defined tokens that can vary widely. This automaton needed to be versatile enough to handle letters, digits, and underscores, while making sure to follow naming conventions and not confuse identifiers with reserved keywords. We had to be especially aware of case sensitivity and language-specific rules, which added a layer of complexity to its design. Evaluating possible cases like camelCase, snake_case, or PascalCase, as well as identifier values starting with an underscore

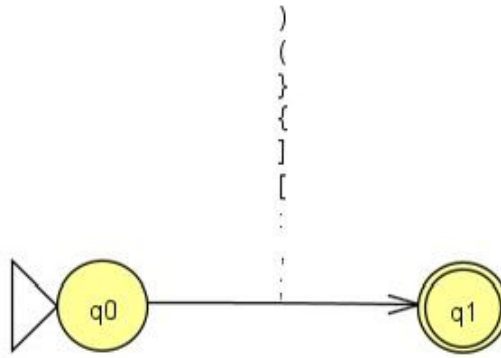


Figure 4: Punctuation automaton

Punctuation Finally, the punctuation automaton plays a crucial role in analyzing the structure of the code. It handles symbols like commas, semicolons, and parentheses, which are essential for proper code formatting. One of the main challenges in this case was ensuring that punctuation was not only recognized correctly, but also used appropriately in various contexts.

3.2 Abstracting Lexers

Now that we understand how each regular expression specifies the different token types, we can use Python’s regex capabilities to capture and save these lexical components. The format we use to record each regular expression is straightforward:

```
(r"[a-zA-Z_$][a-zA-Z_$0-9]*", "IDENTIFIER")
```

In this format, `(r"pattern", "label")` represents the following:

- **label:** The type of token being identified (e.g., IDENTIFIER).
- **pattern:** The regular expression pattern used to match the token.

The `lexer()` function operates by finding matches for these patterns as it reads through strings from an input file. It’s crucial to remember that the order in which categories are defined affects how the lexer processes the input. This order determines the sequence in which the lexer compares and identifies tokens within the core functionality.

To illustrate this, consider the following C code snippet:

```
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```


By carefully defining our regular expressions and their order, we ensure that the lexical analyzer correctly identifies and classifies each token in the input code. The following table is an example of a symbol table generated by the lexical analyzer, with respect to the input file:

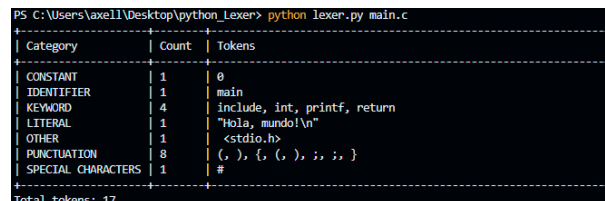
Category	Count	Tokens
Keyword	4	include, int, printf, return
Identifier	1	main
Operator	0	
Constant	1	0
Punctuation	5	(,), {, ;, }
Literal	1	"Hola, mundo!"
Special Characters	2	#

Table 1: Example Symbol Table Generated by the Lexer

3.3 Interface with Tkinter

In our journey of refining the lexer abstraction, we realized that moving beyond the technical details of tokenization and the internal structure of the code was crucial. It became evident that making our tool accessible and understandable to users, even those without a deep understanding of lexers, was just as important. Our goal was to create a visual interface that would resonate with users by presenting information in a familiar and intuitive way.

Thanks to the Tkinter library, we transformed our original, rather archaic interface into something far more engaging and user-friendly. Here's a glimpse of our progress:



```

PS C:\Users\axall\Desktop\python_lexer> python lexer.py main.c
+-----+-----+-----+
| Category | Count | Tokens |
+-----+-----+-----+
| CONSTANT | 1 | 0 |
| IDENTIFIER | 1 | main |
| KEYWORD | 4 | include, int, printf, return |
| LITERAL | 1 | "Hola, mundo!\n" |
| OTHER | 1 | <stdio.h> |
| PUNCTUATION | 8 | (, ), {, (, ), ;, ;, } |
| SPECIAL CHARACTERS | 1 | # |
+-----+-----+-----+
Total tokens: 17

```

Figure 5: Archaic Interface

As you can see, the initial interface was quite basic and did not offer much in terms of user engagement or visual appeal.

We set out to improve this by creating a more aesthetically pleasing and functional design. With Tkinter's help, we developed an interface that not only looks better but also provides a catalog-style view of the tokens. Additionally, by incorporating the Python Imaging Library (PIL), we added functionality to capture and display the token table with the click of a button.

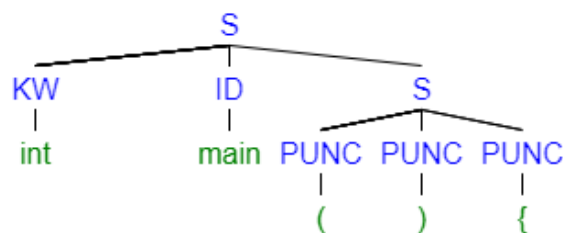
Category	Count	Tokens
CONSTANT	4	5, 15, 10, 0
IDENTIFIER	5	a, b, main, c, Operador
KEYWORD	5	if, int, return, printf, include
LITERAL	6	"El valor de c despu�s de decrementarlo es: %d\n", "z
OPERATOR	8	, /, -, =, <, +, &&, >
PUNCTUATION	6	;, {, (, ,, }
SPECIAL CHARAC	1	#

Save as Image

Total tokens: 138

The updated interface is designed to be much more user-friendly, making it easier for users to interact with and understand the tokenization process. The enhanced aesthetics and functionality not only make the tool more enjoyable to use but also bridge the gap between complex lexer processes and user comprehension.

3.4 Parse Tree



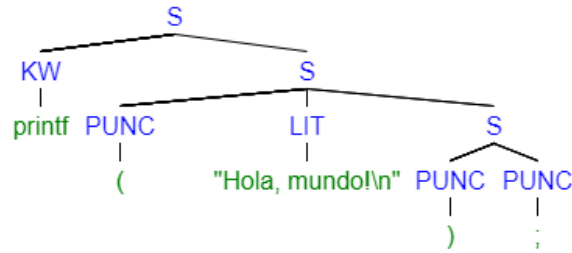


Figure 8: Parse Tree for second line

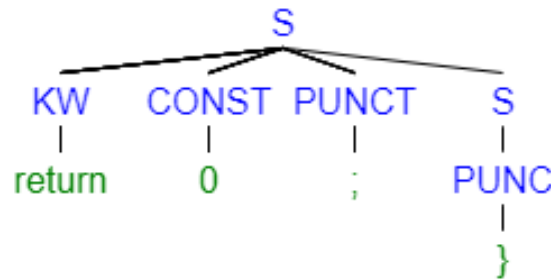


Figure 9: Parse Tree for final line

4 Results

Finally, and as a way of ensuring the correct functioning of our lexical analyzer, different tests were carried out with different input files, so that we could appreciate and evaluate points of view of a panorama as broad as the codes in C Language and, in case of any classification error, consider again the regular expression and the handling of Tokens.

```

#include <stdio.h>
int main() {
    int num = 10;
    int *ptr;
    ptr = &num;
    printf("El valor de num es: %d\n", num);
    printf("La dirección de num es: %p\n", &num);
    printf("El valor de ptr es: %p\n", ptr);
    printf("El valor al que apunta ptr es: %d\n", *ptr);
    return 0;
}
  
```

We can obtain the following table, where we can notice that since we define the regular expressions in a detailed manner, we do not have any consideration error for the case of any character. Therefore it means that the order and specification of the regular expressions is correct. In addition, we can notice that the table eliminates repeated tokens to avoid the trouble of viewing them and losing the sense of catalog, however the total token count continues to consider all of them.

Token's Summary		
Category	Count	Tokens
CONSTANT	2	0, 10
IDENTIFIER	3	num, ptr, main
KEYWORD	4	printf, return, include, int
LITERAL	4	"El valor de num es: %d\n", "El valor de ptr es: %p\n", "I
OPERATOR	2	=, *
PUNCTUATION	6), ,, }, ::, { {
SPECIAL CHARAC	2	#, &
<div> <div></div> <div>Save as Image</div> </div>		
Total tokens: 55		

- [2] A. V. Aho, *Compilers: Principles, Techniques, and Tools*, Addison Wesley Longman, 2000.
- [3] *Compiladores*, Naucalpan de Juárez, Estado de México: Pearson Educ., 2008.