Universidad Nacional Autónoma de México

Computer Engineering

Compilers

Lexer documentation

TEAM 7:
320030772
320323719
320181520
423007262
424147295

Group:
5
Semester:
2026-I

Mexico City, Mexico. September 2025

Contents

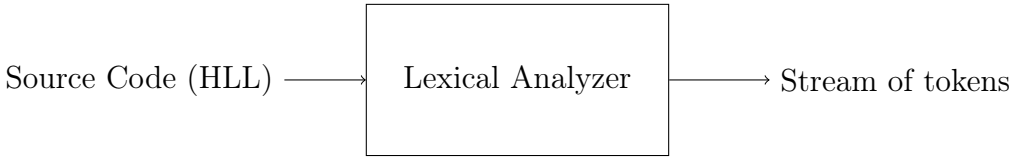
1 Introduction

In this project, the objective is to implement the first phase of a compiler, a lexical analyzer; in the field of compilers, a lexical analyzer is known as the scanning part.
In simple words, a lexical analyzer reads a source program or input (lexeme), where it must identify a basic series of tokens and produce that same list and its final count as output[1].
It is main purpose in this first phase is to prepare this specific output to be used later in the syntactic analyzer; in addition to this, to demonstrate the correct implementation, a context-free grammar (CFG) associated with an input will be presented through an example. This will show that the analyzer not only recognizes tokens but also makes sense within the language for the goal of the parser implementation[2].
Manually building this analyzer deepens and connects theory; for example, it involves the use of regular expressions, text input manipulation, and software testing, all of which are relevant for the subsequent compiler implementations.
Finally, it is expected to implement this phase through a module that reads a provided source code, either from a file or a string, detects and classifies this series of tokens, counts and reports the total number obtained by the implementation, and designs a CFG that represents one of the input examples.
For clarification, this section will only be limited to lexical analysis, will not include any extra implementation, and will not use tools dedicated to this study (for example, FLEX); it will only be limited to the use of the C language.

2 Theoretical Framework

2.0.1 Lexical Analysis

Lexical analysis is the first phase in compiler design and is one of the Front End phases, it consists in the analysis of the source code to convert HLL (high level language) into a stream of tokens.



2.0.2 Tokenization

The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form <token-name, attribute-value> that it passes on the subsequent phase, syntax analysis. This proccess is called tokenization [3]

2.0.3 Clasification of the tokens

The lexical analyzer depends on how we want to construct our language or code, normally the classification of the tokens are: identifiers, operators, constants, keywords, literals, punctuators and special characters. The white spaces, macros and comments are not contemplated as tokens and are removed from the output stream.

2.0.4 Regular expressions

To implement a lexical analyzer, we need to understand how the analysis of the language itself is. A language is a set of strings; a string is a finite sequence of symbols and the symbols themselves are taken from a finite alphabet. [4]
The regular expressions are tools for comprehending how strings are formed and if the string belongs to the language or not. So, regular expressions use a notation to recreate any string and corroborate its belonging.
The main elements to construct a regular expression are described below:
Symbol: any element in the alphabet of the language
Alternation: It is an operator that indicates if a string belongs to a language M or a language N, it is denoted with a vertical bar | or can be denoted with brackets [] [4]

$$M \mid N \text{ or } [MN]$$

Concatenation: Is the operator that indicates a string that is formed by two strings that belong to a language M and N, respectively. It is denoted by a dot but we can omit the symbol and write the strings concatenated. [4]

$$M \cdot N \text{ or } MN$$

Epsilon: ϵ represents a language whose only string is the empty string. [4]
Repetition: Basically, it is the Kleene closure of a regular expression M and is denoted by M*, that represents the concatenation of zero or more strings belonging to M. [4]
Other conventions are:

b-g means bcdefg
M? means (M|ϵ)
M+ means (M·M*)
. means any string

Once we apply and understand the regular expressions we have a general image of the language, now, to implement these expressions into a computer we need to use a Finite Automata.

2.0.5 Finite Automata

A finite automata has a finite set of states; edges lead from one state to another, and each edge is labeled with a symbol. One state is the start state, and certain of the states are distinguished as final states. [4] In a deterministic finite automaton (DFA), there is no more than one transition from the same state labeled with the same symbol, and there are no transitions with epsilon (ϵ). A DFA accepts or rejects a string as follows. Starting in the start state, for each character in the input string the automaton follows exactly one edge to get to the next state. The edge must be labeled with the input character. After making n transitions for an n-character string, if the automaton is in a final state, then it accepts the string. If it is not in a final state, or if at some point there was no appropriately labeled edge to follow, it rejects. The language recognized by an automaton is the set of strings that it accepts. [4]
So, the DFA is sufficient to implement the language in a computer. The programming languages have the sources to represent either the states of the DFA with the corresponding information or the transition between the states.

2.0.6 Context free grammar (CFG)

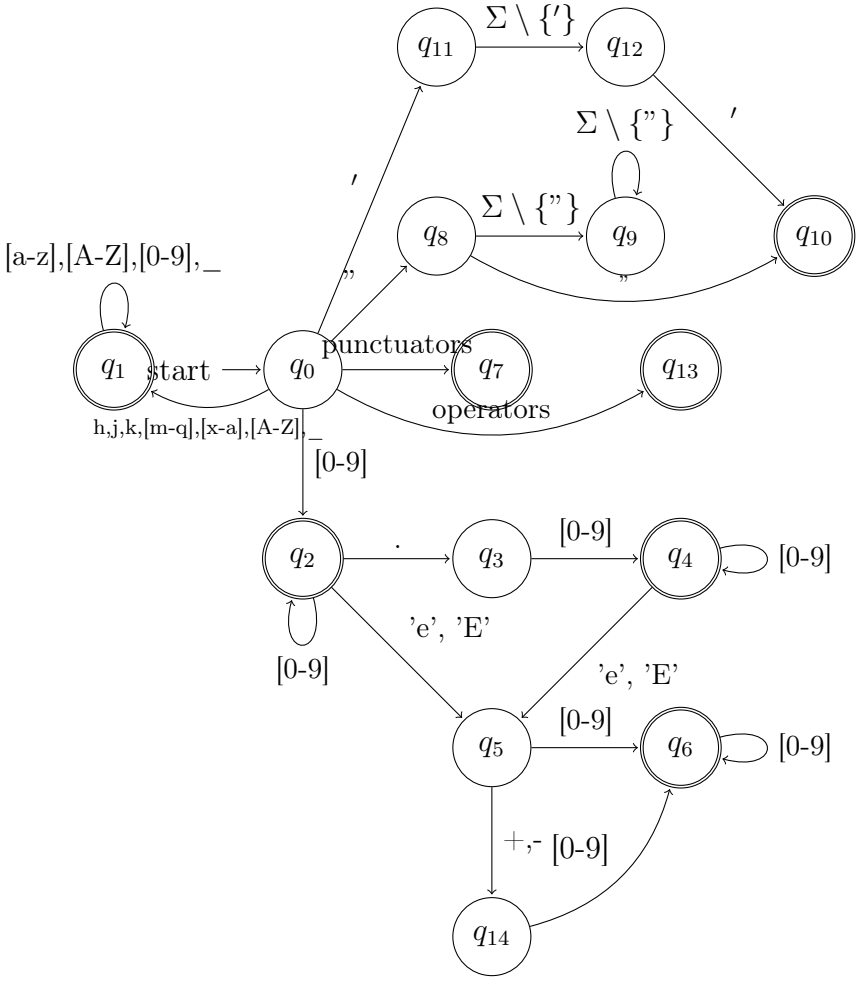
Finally we need the notion of the next phase of the compiler, the syntax analysis. We are not going to delve into the topic, just mention one of the elements used in this phase: the context free grammar. The context free grammar is the form we build the parser of the syntax analyzer. It is feautured by: initial nonterminal element, nonterminal elements, terminal elements and the set of production rules. Basically, it is the form in which the compiler builds the strings, with the tokens that are delivered from the lexical analyzer as inputs, the syntax analyzer verifies if the statements of tokens can be constructed by the CFG.

3 Development

The development of this lexical analyzer is composed of several functional blocks distributed into functions, which will be responsible for working together to transform the character stream into a sequence of tokens. As a first step, there will be a function that will mark the starting point in the input to subsequently begin the analysis, then a block will perform a character-by-character traversal, delimiting the beginning and end of each lexeme. Afterward, there will be a classification block to determine the type of token, functions where elements that do not correspond to the previous classification are discarded, and finally a function that will print the registered tokens with their final count.
With this concept, we ensure clarity in the code and its logic, isolating possible errors without affecting the entire program sequence.
In particular, it will be done using the C language because it provides suitable and necessary functions for the construction, such as the use of pointers or the classification of the program's logic.

3.1 Deterministic Finite Automaton (DFA)

Before starting with the code, the finite state automaton is defined, which will include the regular expressions for token classification. With this, we have the basis of the analyzer, since the expressions formally describe which patterns will be recognized.



Although keywords are important tokens, they take up a lot of space, so they were omitted from this diagram. However, the automaton will check each character; if there is a transition with an alphanumeric symbol different than the next symbol to be checked in the keyword or with underscore (_), it will automatically be transferred to q1.

3.1.1 List of keywords to be used for the C grammar.

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	volatile	while		

Table 1: Keywords in C

3.1.2 List of regular expressions.

- **Punctuators:** (|) , { | } | [|] | ; | , | : | ?
- **Operators:** = | == | + | ++ | += | - | - - | -= | -> | * | *= | / | /+ | % | %= | & | && | &= | | | || | = | ! | != | ^ | ≙ | ~ | < | « | <= | «= | > | » | >= | »= | .
- **Constants:** (0-9)+(.(0-9)+)?((e|E)(+|-)?(0-9)+)?
- **Literals:** "(Σ-("))*" | 'Σ-(')'
- **Identifiers:** [a-z|A-Z|_][a-z|A-Z|0-9|_]*

3.2 Description of Functions

3.2.1 Initialization Functions

This part prepares the analyzer before starting to read the input. It's responsible for placing the pointers that will mark the beginning and current position within the lexeme. The *initScanner* function is left to point to the start of the input, so when its execution begins, each token will reflect its beginning in the lexeme.
A lexical analysis always requires an initial state, therefore, by implementing pointers, we ensure an error-free analysis. This is important because this function works with *classifyToken*, *lookupKeyword* and *lookupOperator*

3.2.2 Token Classification Functions

Several functions were implemented, each performing a task in the process of token classification. The *getTokenType* function will receive a string extracted from the input, then perform a classification which will be the output that feeds *AssignCategory* and *ValidateToken* for the subsequent phases.
Next, *ValidateToken* will receive the token and the assigned type. Through a boolean, it will indicate if it's valid. This is important because if it fails, it proceeds to call *detectLexicalError*. With this output, *AssignCategory* receives the token and the validation, and as its name implies, it is limited to providing the category label.
Additionally, functions that apply implicit mechanisms such as a token filter, which will clean comments and spaces, and a lexical error detector, which will indicate a lexical error in the code, were implemented.

3.2.3 Auxiliary Recognition Functions

These were assigned as auxiliary tasks to help identify properties, whether basic character properties or token properties. They help us confirm if they belong to a certain category. *isLetter* will tell us if a character belongs to the alphabet, so the output should be a boolean value; the same applies, but in their own context, to *isDigit*, *isOperator*, *isPunctuator*, and *isLiteral*.

3.2.4 Special Lexeme Handling Functions

Their goal is to identify and process tokens that do not belong to a group like identifiers or operators, but to special elements as their name says, for example, escape sequences, combined symbols, or one with a particular context in the language. For instance, *handleComments* detects and omits comment lines or blocks; or *handleEscapeSequence* converts the escape sequence into valid literals.

3.2.5 Main Analysis Function

This is the most important part since it aims to coordinate all the handling and auxiliary functions that will convert the input into the expected sequence of tokens. Therefore, it is the core of the lexical analyzer because it mainly ensures that all functions are executed and all elements are recognized. This part emphasizes the *lexicalAnalysis* function.

3.3 Integration Tests

As a final step, a series of tests were conducted with representative code fragments. These should be entered into the terminal, and the expected result is a list of tokens with their final classification and count.

1. Input: `printf("Hello World!");`

   - `printf` → identifier
   - `(` → punctuator
   - `"Hello World!"` → literal
   - `)` → punctuator
   - `;` → punctuator

   Total tokens: 5

2. Input: `int x = 10;`

   - `int` → keyword
   - `x` → identifier
   - `=` → operator
   - `10` → constant
   - `;` → punctuator

   Total tokens: 5

3. Input: `x++; y = x * 5;`

   - `x` → identifier

- `++` → operator
- `;` → punctuator
- `y` → identifier
- `=` → operator
- `x` → identifier
- `*` → operator
- `5` → constant
- `;` → punctuator

Total tokens: 9

4. Input: `if (x == 5) { printf("ok"); }`

   - `if` → keyword
   - `(` → punctuator
   - `x` → identifier
   - `==` → operator
   - `5` → constant
   - `)` → punctuator
   - `{` → punctuator
   - `printf` → identifier
   - `(` → punctuator
   - `"ok"` → literal
   - `)` → punctuator
   - `;` → punctuator
   - `}` → punctuator

   Total tokens: 13

5. Input: `int x = 10; // this is a comment`

   - `int` → keyword
   - `x` → identifier
   - `=` → operator
   - `10` → constant
   - `;` → punctuator
   - Comment: ignored by the lexer

   Total tokens: 5

6. Input: `float y = 1.23e+10;`

   - `float` → keyword
   - `y` → identifier
   - `=` → operator
   - `1.23e+10` → constant
   - `;` → punctuator

   Total tokens: 5

4 Results

Our lexer has two ways of being used:

./lexer.out -s 'a statement'

./lexer.out filepath/filename.c

In these examples, the executable was named *lexer*. Based on this, we present some outputs of the lexer when processing different input strings. Screenshots are included to demonstrate token recognition.

Figure 1: Tokens in *Hello world* statement.

As shown in Figure 1, the lexer correctly identifies the function name printf, the punctuation symbols (,), , and the literal string. This statement can be derived from the following CFG:

⟨*statement*⟩ ::= ⟨*id*⟩ ⟨*left_par*⟩ ⟨*expr*⟩ ⟨*right_par*⟩ ⟨*semicolon*⟩

⟨*id*⟩ ::= `printf`

⟨*expr*⟩ ::= ⟨*string*⟩

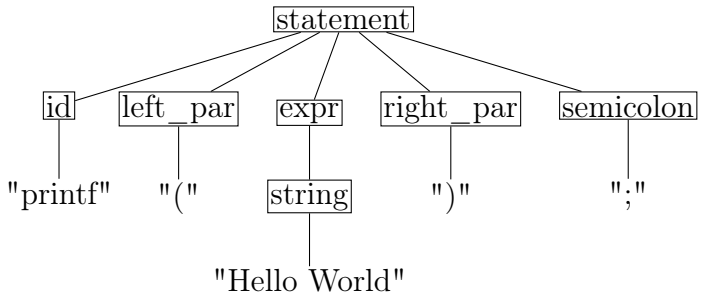
⟨*string*⟩ ::= `*-(")` | `*-(')`

⟨*left_par*⟩ ::= `(`

⟨*right_par*⟩ ::= `)`

⟨*semicolon*⟩ ::= `;`

The derivation process would be as follows.



```
santana@pc-bda-gss:~/Downloads$ ./lexer.out test1.c
KEYWORD('int')
IDENTIFIER('main')
PUNCTUATOR('(')
KEYWORD('void')
PUNCTUATOR(')')
PUNCTUATOR('{')
KEYWORD('int')
IDENTIFIER('x')
PUNCTUATOR(',')
IDENTIFIER('y')
OPERATOR('=')
CONST('0')
PUNCTUATOR(',')
IDENTIFIER('i')
PUNCTUATOR(';')
IDENTIFIER('x')
OPERATOR('=')
CONST('789')
IDENTIFIER('printf')
PUNCTUATOR('(')
LITERAL('"x + y = %d + %d = %d\n"')
PUNCTUATOR(',')
IDENTIFIER('x')
PUNCTUATOR(',')
IDENTIFIER('y')
PUNCTUATOR(',')
IDENTIFIER('x')
OPERATOR('+')
IDENTIFIER('y')
PUNCTUATOR(')')
PUNCTUATOR(';')
KEYWORD('return')
CONST('0')
PUNCTUATOR(';')
PUNCTUATOR('}')

TOTAL: 35
santana@pc-bda-gss:~/Downloads$
```

Figure 2: Tokens in *test1.c* file.

Figure 2 illustrates that our lexer can successfully tokenize every type of token. Finally, more outputs are shown.

```
santana@pc-bda-gss:~/Downloads$ ./lexer.out -s 'int x = 10;'
KEYWORD('int')
IDENTIFIER('x')
OPERATOR('=')
CONST('10')
PUNCTUATOR(';')

TOTAL: 5
santana@pc-bda-gss:~/Downloads$
```

Figure 3: Tokens in arbitrary statement.

```
santana@pc-bda-gss:~/Downloads$ ./lexer.out -s 'x++; y = x * 5;'
IDENTIFIER('x')
OPERATOR('++')
PUNCTUATOR(';')
IDENTIFIER('y')
OPERATOR('=')
IDENTIFIER('x')
OPERATOR('*')
CONST('5')
PUNCTUATOR(';')

TOTAL: 9
santana@pc-bda-gss:~/Downloads$
```

Figure 4: Tokens in arbitrary statement.

```
santana@pc-bda-gss:~/Downloads$ ./lexer.out -s 'if (x == 5) { print("ok"); }'
KEYWORD('if')
PUNCTUATOR('(')
IDENTIFIER('x')
OPERATOR('==')
CONST('5')
PUNCTUATOR(')')
PUNCTUATOR('{')
IDENTIFIER('print')
PUNCTUATOR('(')
LITERAL('"ok"')
PUNCTUATOR(')')
PUNCTUATOR(';')
PUNCTUATOR('}')

TOTAL: 13
santana@pc-bda-gss:~/Downloads$
```

Figure 5: Tokens in arbitrary statement.

```
santana@pc-bda-gss:~/Downloads$ ./lexer.out -s 'int x = 10; // this is a comment'
KEYWORD('int')
IDENTIFIER('x')
OPERATOR('=')
CONST('10')
PUNCTUATOR(';')

TOTAL: 5
santana@pc-bda-gss:~/Downloads$
```

Figure 6: Tokens in arbitrary statement.

```
santana@pc-bda-gss:~/Downloads$ ./lexer.out -s 'float y = 1.23e+10;'
KEYWORD('float')
IDENTIFIER('y')
OPERATOR('=')
CONST('1.23e+10')
PUNCTUATOR(';')

TOTAL: 5
santana@pc-bda-gss:~/Downloads$
```

Figure 7: Tokens in arbitrary statement.

5 Conclusions

The results show the classification of each token and the total of them. To reach this point it was important to understand how to implement the process of tokenization in code, in this case C. The regular expressions give us the notion of how to analyze the corresponding string to classify it. Moreover, we build a DFA on base in the regular expressions to have a formalism that helps us coding the language. Then we use the C language to recreate the DFA, mostly using cycles, pointers and if statements, the pointers to indicate the states and the if statements to do the transitions. Finally, check if our tokens can be represented by a CFG, this is important as the next phase of the compiler, the syntax analysis, needs a CFG to work. As we can see, the way we built the lexical analyzer was based on the theoretical framework using the concepts to implement the process of tokenization.

References

[1] GeeksforGeeks, *Working of lexical analyzer in compiler*, `https://www.geeksforgeeks.org/compiler-design/working-of-lexical-analyzer-in-compiler/`, Accessed: 2025-09-25, 2025.

[2] A. desconocido, *Función del analizador léxico*, `http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/21_funcin_del_analizador_lxico.html`, Accedido: 2025-09-24, 2022. [Online]. Available: `http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/21_funcin_del_analizador_lxico.html`.

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

[4] A. W. Appel, *Modern compiler implementation in C*, 1st ed. THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE, 2004.

[5] A. desconocido, *Análisis léxico*, `https://compiladores1.webnode.mx/ice-dea/`, Accedido: 2025-09-24, 2024. [Online]. Available: `https://compiladores1.webnode.mx/ice-dea/`.

[6] cppreference.com. "C keywords." Accedido: 2025-09-25. [Online]. Available: `https://en.cppreference.com/w/c/keyword.html`.

[7] cppreference.com. "Punctuation." Accedido: 2025-09-25. [Online]. Available: `https://en.cppreference.com/w/c/language/punctuators.html`.

[8] GeeksforGeeks. "Keywords in c." Accedido: 2025-09-25. [Online]. Available: `https://www.geeksforgeeks.org/c/keywords-in-c/`.