Universidad Nacional Autónoma de México

Computer Engineering

Compilers

LEXICAL ANALYZER

ALUMNO:
423032833
423529834
320318931
320332825
320341704

Grupo:
5
Semestre:
2026-I

México,CDMX. September 2025

Contents

1 Introduction

In the area of programming languages, lexical analysis is a fundamental stage in the compilation of a program, since its process consists of taking source code as input in order to analyze and classify it into tokens, such as keywords, identifiers, operators, constants, punctuation, etc. A correct classification of tokens is the foundation for the subsequent stages of the compiler; if the lexical analyzer has errors when identifying and classifying a token, the compiler will not be able to correctly interpret the programmer's instructions. Therefore, a correct lexical analyzer is essential to ensure that the compiler works properly.

The main objective is, therefore, to design and implement a lexical analyzer that fulfills two important functions: the first is to correctly identify each token and assign its corresponding category; the second is to provide an accurate count of the total number of tokens identified, as well as display such classification. The correct implementation of this analyzer ensures that the next stage, syntactic analysis, receives the proper classification of the tokens.

2 Theoretical framework

The first phase of a compiler is lexical analysis or scanning. The lexical analyzer scans the stream of characters from the source code and groups it into lexemes. A lexeme is "a sequence of characters in the source program that matches the pattern for a token" [1]. The output of each lexeme is a token, which is "a pair consisting of a token name and an optional attribute value" [1]. The token name will be used in the syntax analysis and the value will be used in the semantic analysis.

There are different categories of tokens:

- **Identifiers:** name of variables, functions, and classes with the format `[a-zA-Z ][a-zA-Z0-9 ]*`
- **Keywords:** `int, float, for, while, if, else, return, print`
- **Operators:** `==, !=, <=, >=, ++, --, +=, -=, *=, /=, %=, &&,` |, +, -, *, /,
- **Punctuators:** ( ) , ; `{ }`
- **Constants:** numerical values with the format d+(? : .d+)?)
- **Literals:** strings with the format

```
"([^"\\]|\\.)*"|'([^'\\]|\\.)*'
```

To perform the tokenization of source code, the lexical analyzer must recognize the pattern of each lexeme to decide if it's a keyword, an identifier, an operator, etc. This pattern recognition process relies on regular expressions (regex), which provide a declarative notation for defining the valid patterns that make up the lexemes. Each token type is described by its corresponding regular expression, ensuring that the syntax of each lexical unit is unambiguous.

Each regex corresponds to a regular language that can be recognized by a deterministic finite automaton (DFA). During this phase, whitespace is ignored, as defined by the regular expression `\s+`. The analyzer is also responsible for rejecting invalid characters, which typically results in a syntax error.

3 Development

3.1 Functions

lexer(code: str): The lexer function takes a piece of code text, goes through it from start to finish, and ignores spaces. At each position, it tries to recognize a valid token according to the regular expressions defined in the tokens list, which is a list of tuples. This list is previously compiled to make execution more efficient. If it finds a match within tokens, it extracts the lexeme and adds it to the result list as a tuple and updates the counts counter for that lexeme. If none of the regular expressions match any fragment, it means that it contains an invalid character. At the end of the text scan, the function returns the result list with all the tokens in the order they appeared and the counts dictionary that indicates how many tokens of each type were recognized. [3]

3.2 Token's Automatas

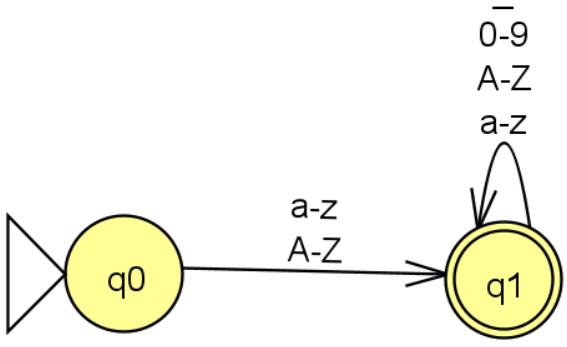

Figure 1: Identifier Automata

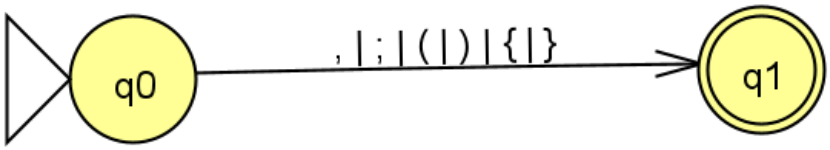

Figure 2: Punctuator Automata

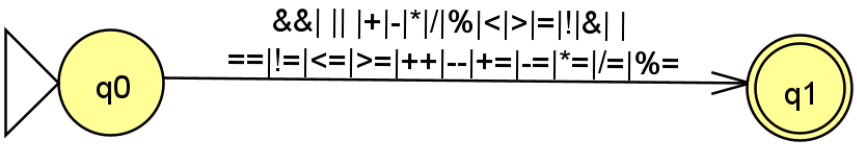

Figure 3: Operator Automata

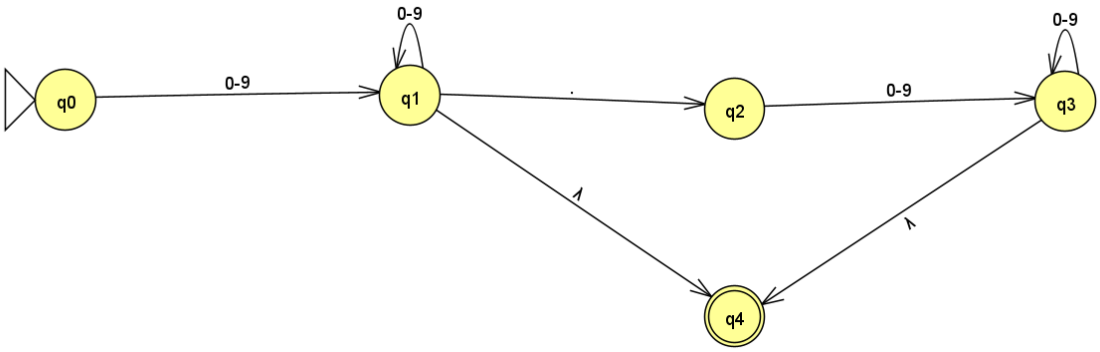

Figure 4: Constant Automata

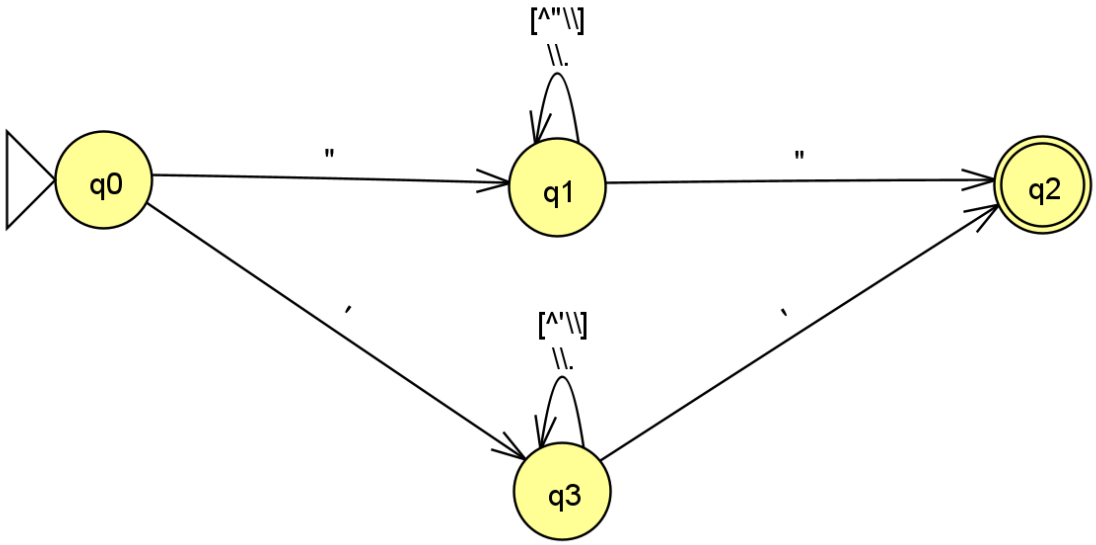

Figure 5: Literal Automata

3.3 Context-Free Grammar (CFG) [2]

Once tokens are available, we can describe how they combine into valid programs. A **Context-Free Grammar (CFG)** $G = (V, \Sigma, R, S)$ is defined as:

- Non-terminals V: syntactic categories (abstract structures like `Expr` or `Stmt`).
- Terminals Σ: actual tokens from the lexer (`int`, identifiers, operators, literals, etc.).
- Start symbol S: the top-level category, here `Program`.
- Productions R: rewriting rules that describe valid sentence structures.

3.3.1 Grammar rules

A simplified CFG for this language is:

$$
\begin{aligned}
Program &\rightarrow StmtList \\
StmtList &\rightarrow Stmt\ StmtList \mid \epsilon \\
Stmt &\rightarrow Decl \mid Assign \mid PrintStmt \mid ReturnStmt \\
Decl &\rightarrow Type\ Id\ OptAssign\ ";" \\
Type &\rightarrow "int" \mid "float" \\
OptAssign &\rightarrow "=" \ Expr \mid \epsilon \\
Assign &\rightarrow Id\ AssignOp\ Expr\ ";" \\
AssignOp &\rightarrow "=" \mid "+=" \mid "-=" \mid "*=" \mid "/=" \mid "\%=" \\
PrintStmt &\rightarrow "print""("ArgList")"";" \\
ArgList &\rightarrow Expr\ ArgTail \mid \epsilon \\
ArgTail &\rightarrow ","ExprArgTail \mid \epsilon \\
ReturnStmt &\rightarrow "return"Expr";" \\
Expr &\rightarrow Id \mid Const \mid Literal \mid IdOpExpr \mid "("Expr")" \\
Op &\rightarrow "+" \mid "-" \mid "*" \mid "/" \mid "\%" \\
Id &\rightarrow \text{identifier} \\
Const &\rightarrow \text{constant} \\
Literal &\rightarrow \text{literal}
\end{aligned}
$$

3.3.2 What each non-terminal means

- **Program:** the full source file, consisting of a list of statements.
- **StmtList:** one or more statements in sequence, or empty (epsilon).
- **Stmt:** a single statement, which could be a declaration, assignment, print call, or return.
- **Decl:** declaration of a variable, possibly with an initial value.
- **Type:** the variable type, e.g. `int` or `float`.
- **OptAssign:** optional assignment of a value to the variable upon declaration.
- **Assign:** re-assignment of a value to an existing variable.
- **AssignOp:** assignment operators supported, like = or +=.
- **PrintStmt:** statement for printing values to output.
- **ArgList:** arguments passed to a print function, possibly empty.
- **ReturnStmt:** statement to return a value from a function.

- **Expr:** an expression, which can be an identifier, constant, literal, operation, or grouped subexpression.
- **Op:** arithmetic operators allowed.
- **Id, Const, Literal:** terminal categories corresponding to tokens.

3.4 Issues in the Original CFG [4]

The initial CFG had several problems:

- **Ambiguity:**
  - The original expression production `Expr → Id` Const | Literal | Id Op Expr | (Expr)| allows multiple derivations for strings like `x + y`, making it ambiguous.
- **Non-determinism (LL(1) conflicts):**
  - With a single lookahead token, the parser cannot always decide which production to apply.
  - For example, upon seeing an `Id` token, it cannot distinguish between `Id` alone or `Id Op Expr`.
  - Similarly, | ArgList → Expr ArgTail | ϵ | is ambiguous at lookahead if the next token could start an expression or indicate an empty list.
- **Operator precedence not enforced:**
  - The original grammar does not separate terms and operations by precedence levels, causing ambiguity in parsing arithmetic expressions.

These issues make the original CFG unsuitable for predictive (recursive-descent) parsers.

3.5 Revised Context-Free Grammar (CFG)

To make the CFG **unambiguous**, **right-recursive**, and **deterministic LL(1)**, expressions and lists are rewritten carefully.

3.5.1 Grammar rules (LL(1) version)

$$
\begin{aligned}
Program &\rightarrow StmtList \\
StmtList &\rightarrow Stmt\ StmtList \mid \epsilon \\
Stmt &\rightarrow Decl \mid Assign \mid PrintStmt \mid ReturnStmt \\
Decl &\rightarrow Type\ Id\ OptAssign\ ";" \\
Type &\rightarrow "int" \mid "float" \\
OptAssign &\rightarrow "=" \ Expr \mid \epsilon \\
Assign &\rightarrow Id\ AssignOp\ Expr\ ";" \\
AssignOp &\rightarrow "=" \mid "+=" \mid "-=" \mid "*=" \mid "/=" \mid "\%=" \\
PrintStmt &\rightarrow "print""("\ ArgList")""";" \\
ArgList &\rightarrow Expr\ ArgTail \mid \epsilon \\
ArgTail &\rightarrow ","ExprArgTail \mid \epsilon \\
ReturnStmt &\rightarrow "return" Expr";" \\
Expr &\rightarrow Term\ Expr' \\
Expr' &\rightarrow Op\ Term\ Expr' \mid \epsilon \\
Term &\rightarrow Id \mid Const \mid Literal \mid "("Expr")" \\
Op &\rightarrow "+" \mid "-" \mid "*" \mid "/" \mid "\%" \\
Id &\rightarrow \text{identifier} \\
Const &\rightarrow \text{constant} \\
Literal &\rightarrow \text{literal}
\end{aligned}
$$

3.5.2 Justification

- **Unambiguity:** Expr is split into Term + Expr' to prevent multiple derivations for the same string.
- **Right recursion:** StmtList, ArgTail, and Expr' are right-recursive, which is convenient for recursive-descent parsing.
- **Deterministic LL(1):** Each production can be chosen with a single lookahead token.

4 Results

4.1 Test 1

The program receives a file named Archivo_Prueba1.txt, and the output can be observed with the correct classification of the tokens, the total token count was 64.

4.1.1 Input

```
int main(){
    int x, a=1, b=2, c=3;
    x=a+b*c;
    print("The result is %D",x);
    return 0;
    if(a<b){
        print("hola")
    }else if(b==a){
        print("Adios")
    }
}
```

Figure 6: Archivo_Prueba1.txt

4.1.2 Output

```
El archivo a utilizar es: Archivo_Prueba1.txt

TOKENS agrupados:
keywords: 'int', 'return', 'if', 'else'
identifier: 'main', 'x', 'a', 'b', 'c', 'print'
punctuacion: '(', ')', '{', ',', ';', '}'
operator: '=', '+', '*', '<', '=='
constant: '1', '2', '3', '0'
literal: '"The result is %D"', '"hola"', '"Adios"'

CANTIDADES:
keywords: 6
identifier: 17
punctuacion: 26
operator: 8
constant: 4
literal: 3

Total= 64
```

Figure 7: Archivo_Prueba1.txt Output

4.2 Test 2

The program receives a file named Archivo_Prueba2.txt, and the output can be observed with the correct classification of the tokens, the total token count was 58.

4.2.1 Input

```
int a = 10, b = 20, c = 30;
int x;
x = a + b * c;
print("El resultado es", x);

if (a < b) {
    print("a es menor que b");
} else if (b == a) {
    print("a es igual a b");
}
```

Figure 8: Archivo_Prueba2.txt

4.2.2 Output

```
El archivo a utilizar es: Archivo_Prueba2.txt

TOKENS agrupados:
keywords: 'int', 'if', 'else'
identifier: 'a', 'b', 'c', 'x', 'print'
operator: '=', '+', '*', '<', '=='
constant: '10', '20', '30'
punctuacion: ',', ';', '(', ')', '{', '}'
literal: '"El resultado es"', '"a es menor que b"', '"a es igual a b"'

CANTIDADES:
keywords: 5
identifier: 16
punctuacion: 23
operator: 8
constant: 3
literal: 3

Total= 58
PS C:\Users\DELL\Downloads\Lexer\Prueba>
```

Figure 9: Archivo_Prueba2.txt Output

4.3 Test 3

The program receives a file named Archivo_Prueba3.txt, and the output can be observed with the correct classification of the tokens, the total token count was 63.

4.3.1 Input

```
float promedio = 0.0;
for (int i = 0; i < 10; i++) {
    float nota = i * 1.5 + 2.0;
    promedio += nota;
}
if (promedio >= 50.0 && promedio <= 100.0) {
    print("Promedio final:", promedio);
} else {
    print("Promedio fuera de rango:", promedio);
}
```

Figure 10: Archivo_Prueba3.txt

4.3.2 Output

```
El archivo a utilizar es: Archivo_Prueba3.txt

TOKENS agrupados:
keywords: 'float', 'for', 'int', 'if', 'else'
identifier: 'promedio', 'i', 'nota', 'print'
operator: '=', '<', '++', '*', '+', '+=', '>=', '&&', '<='
constant: '0.0', '0', '10', '1.5', '2.0', '50.0', '100.0'
punctuacion: ';', '(', ')', '{', '}', ','
literal: '"Promedio final:"', '"Promedio fuera de rango:"'

CANTIDADES:
keywords: 6
identifier: 14
punctuacion: 23
operator: 11
constant: 7
literal: 2

Total= 63
PS C:\Users\DELL\Downloads\Lexer\Prueba>
                                    Lín. 13, col. 1   Espacios: 4   UTF-8
```

Figure 11: Archivo_Prueba2.txt Output

4.4 Test 4

The program receives a file named Archivo_Prueba4.c. The output generates an error because the extension is incorrect; only .txt files are accepted. Additionally, if a non-existent file is provided, an error will also occur.

4.4.1 Input

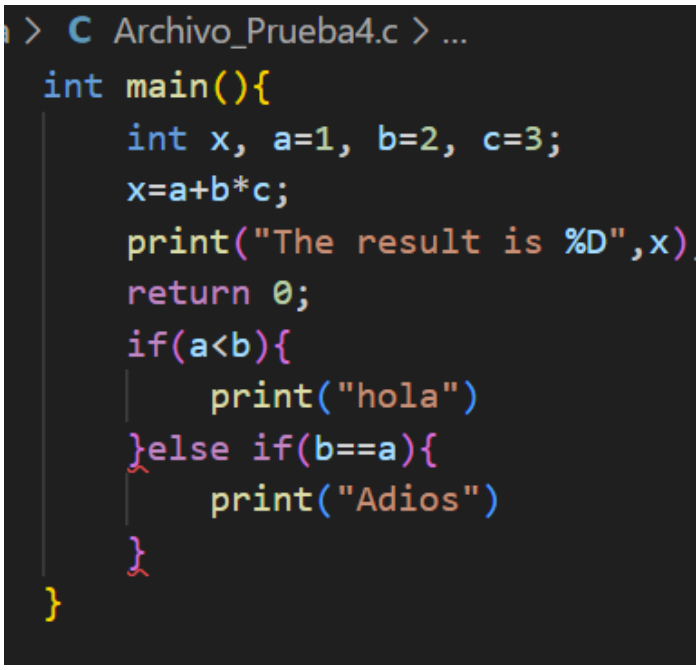

Figure 12: Archivo_Prueba4.c

4.4.2 Output

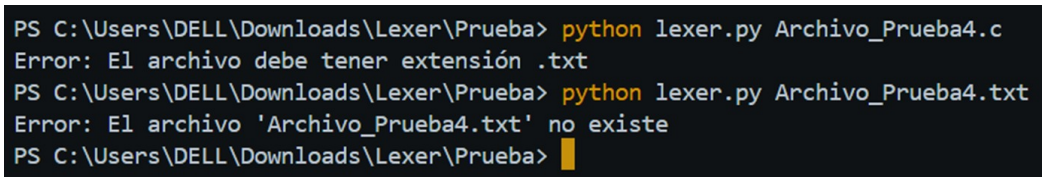

Figure 13: Archivo_Prueba4.c Output and File Non-Existence

4.5 Test 5

When no input file is supplied, the program executes a predefined default code embedded in the script, the total token count was 21.

4.5.1 Input

```
int main() {
    int x = 10;
    print("Hola mundo", x);
    return 0;
}
```

Figure 14: Default code

4.5.2 Output

```
PS C:\Users\DELL\Downloads\Lexer\Prueba> python lexer.py

TOKENS agrupados:
keywords: 'int', 'return'
identifier: 'main', 'x', 'print'
punctuacion: '(', ')', '{', ';', ',', '}'
operator: '='
constant: '10', '0'
literal: '"Hola mundo"'

CANTIDADES:
keywords: 3
identifier: 4
punctuacion: 10
operator: 1
constant: 2
literal: 1

Total= 21
```

Figure 15: Default code Output

5 Conclusion

The implementation of the lexical analyzer confirms the central role of lexical analysis as the first stage in the compilation process. By applying regular expressions to recognize the patterns of lexemes, the analyzer reliably classifies the source code into tokens such as keywords, identifiers, operators, constants, literals and punctuation. Each regular expression corresponds to a regular language that can be recognized by a deterministic finite automaton, ensuring that token recognition is both precise and efficient. The design also correctly handles whitespace and rejects invalid characters, thereby guaranteeing that only valid lexical units are processed.

The analyzer successfully identified each token and assigned it to its corresponding category, while also maintaining an accurate count of the total tokens recognized. This correct classification provides the essential input required for subsequent stages of compilation, particularly syntax analysis. In this way, the project demonstrates how the theoretical concepts of regular expressions and finite automata directly support the practical construction of a reliable lexical analyzer, reinforcing the foundational link between theory and implementation in compiler design.

References

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd. Pearson, 2007.

[2] *Context Free-Grammars*. Department of Computer Science - University of Rochester. URL: https://www.cs.rochester.edu/u/brown/173/readings/05_grammars.pdf.

[3] Python Software Foundation. *re — Regular expression operations*. Version 3.13.7. Python Software Foundation. 2024. URL: https://docs.python.org/3/library/re.html.

[4] A. Jiménez Hernández. *APUNTES DE COMPILADORES*. Facultad de Ingeniería - UNIVERSIDAD NACIONAL AUTOMOMA DE MEXICO. URL: http://www.ptolomeo.unam.mx:8080/xmlui/bitstream/handle/132.248.52.100/10230/7J%20APUNTES%20DE%20COMPILADORES_OCR.pdf?sequence=1&isAllowed=y.