UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

COMPUTER ENGINEERING

COMPILERS

Group: 5

Teacher: M.C. RENE ADRIAN DAVILA PEREZ

Delivery date: March 13, 2025

Lexical Analyzer

Team: 8

Account Number	Last Name	Middle Name	First Name(s)
320334489	Jiménez	Elizalde	Josue
320067354	Medina	Guzmán	Santiago
320054831	Tavera	Castillo	David Emmanuel
320218666	Tenorio	Martinez	Jesus Alejandro
117004023	Salazar	Rubi	Héctor Manuel

Semester 2025-2

Compilers
Group: 05
UNAM / FI / DIE
Semester 2025-2

Lexical Analyzer

Delivery date: March 13, 2025

1 Introduction

In this project, we built and implemented a lexical analyzer using the Python programming language, with help of the topics covered in the theoretical class.

We developed the lexical analyzer so that it reads an input string and processes each symbol one by one. In this way, it classifies each lexeme into a specific type, such as a keyword, constant, identifier, literal, etc. At the end of execution, it will show the number of tokens, as well as the classification of all lexemes.

2 Theoretical Framework

The lexical analyzer, also known as a lexicographic analyzer (or scanner/lexer), is the first phase of a compiler. It consists of a program that takes source code as input and produces tokens and symbols as an output. These tokens are then used in a later state of the translation process, serving as input for the syntactic analyzer (parser).

Programming languages include rules based on regular expressions that define the set of possible character sequences for a token or lexeme.

The lexical token, or simply token, is a string with an assigned meaning and, therefore, can be identified. It is structured as a pair consisting of a "token name" and an optional "token value." The token name represents a category of lexical units. Common token names include:

- **Identifier**: Names chosen by the programmer.
- **Keyword**: Unique names in the programming language.
- **Punctuation**: Punctuation characters.
- **Operator**: Symbols that operate over arguments and produce results.
- **Literals**: Logical, numeric, textual, or reference values.

Examples of each category:

Category	**Example**
Identifier	`variableName`
Keyword	`if`, `while`, `return`
Punctuation	`;`, `,`, `()`
Operator	`+`, `-`, `*`, `/`
Literals	`42`, `"Hello World"`, `true`

Table 1: Examples of lexical categories

Finally, the lexical analyzer will produce something like the following sequence of tokens:

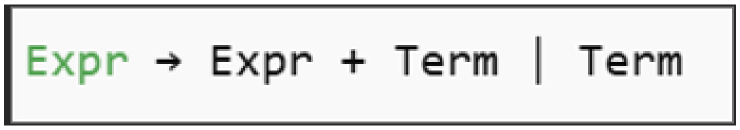
Type	Value
Identifier	x
Operator	$=$
Identifier	a
Operator	$+$
Identifier	b
Operator	$*$
Literal	2
Separator	;

Lexical Analyzers and Regular Expressions

The grammar used for lexical analyzers is based on regular expressions. These regular expressions define the character sequences that can form a token.

Left Recursion

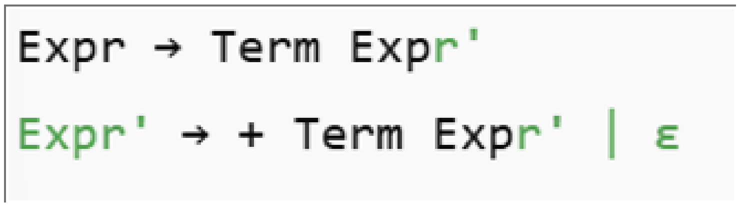
It occurs when a context-free grammar has a production that calls itself at the beginning.



```
Expr → Expr + Term | Term
```

Figure 1: Production rule for expressions in a context-free grammar

It can be corrected as follows:



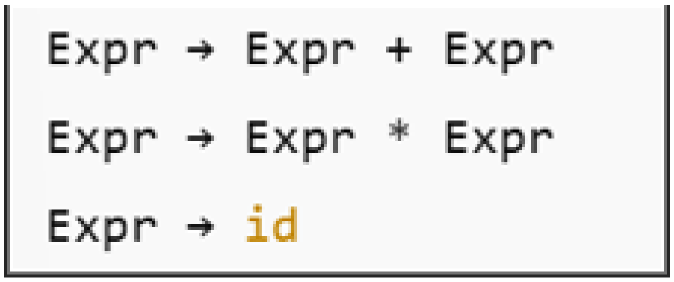
```
Expr → Term Expr'
Expr' → + Term Expr' | ε
```

Figure 2: Non-left recursive form of the grammar for arithmetic expressions.

In this way, the recursion is converted into an iterative form.

Ambiguity

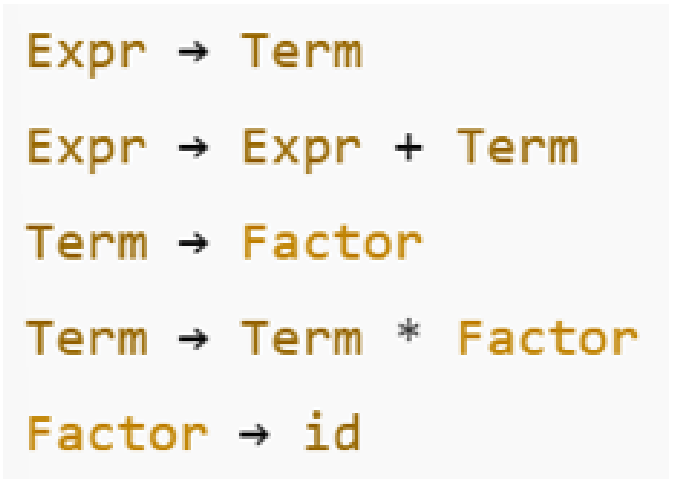
The ambiguity occurs when an input string can be derived in more than one way. For example:



```
Expr → Expr + Expr
Expr → Expr * Expr
Expr → id
```

Figure 3: An example of an ambiguous grammar, where an input string can have multiple derivations.

In the previous grammar, the input: a + b + c can be derived as (a + b) * c or a + (b * c). + and * have no precedence over each other, this can be corrected as follows:



```
Expr → Term
Expr → Expr + Term
Term → Factor
Term → Term * Factor
Factor → id
```

Figure 4: An ambiguous grammar where addition and multiplication have no precedence, leading to multiple derivations.

* is resolved first because Term → Term * Factor its lower in the hierarchy.

3 Development

The Python module provides operations for regular expressions similar to those in Perl. Regular expressions use the backslash (\) to indicate special forms or to allow special characters to be used without invoking their special meaning. This can create some issues when using Python with these characters.

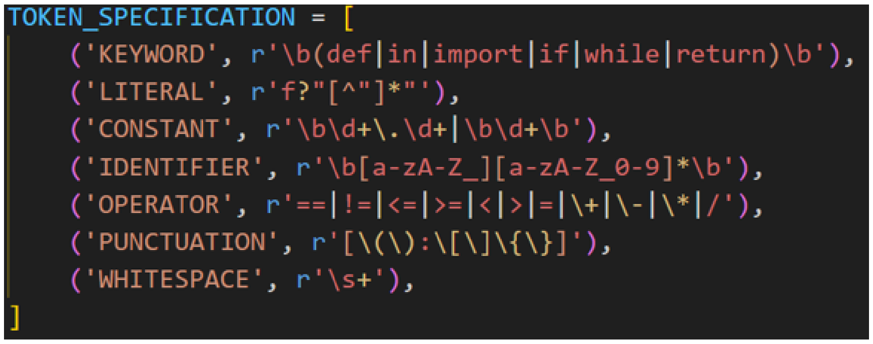
For example, to match a backslash, one must write ‘

‘ as a pattern because the regular expression must be ‘
‘, and each backslash must be represented as ‘
‘ within a regular Python string literal. To avoid this issue, it is recommended to use Python's raw string notation for regular expression patterns, as the backslash is not treated in a special way in a raw string. This, ‘r"\n"‘ is a two-character string containing ‘\‘ and ‘n‘, whereas ‘"\n"‘ is a single-character string representing a newline.

Regular expressions can contain special and ordinary characters. Most ordinary characters, such as ‘A‘, ‘a‘, or ‘0‘, are the simplest regular expressions; they match themselves exactly. Ordinary characters can be concatenated, so ‘last‘ matches the string ‘last‘. Some characters, like ‘—‘ or ‘(‘, are special and represent character classes or influence how regular expressions are interpreted in their environment. It is important to highlight that most regular expressions are available as functions and methods at the module level in compiled regular expressions.

We also used the ‘sys‘ module to access terminal arguments like ‘sys.argv‘.

In the first part of the code, we define the tokens using regular expressions:



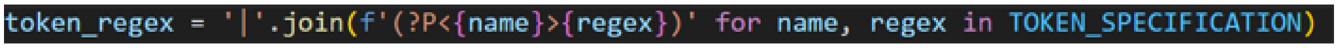
```
TOKEN_SPECIFICATION = [
    ('KEYWORD', r'\b(def|in|import|if|while|return)\b'),
    ('LITERAL', r'f?"[^"]*"'),
    ('CONSTANT', r'\b\d+\.\d+|\b\d+\b'),
    ('IDENTIFIER', r'\b[a-zA-Z_][a-zA-Z_0-9]*\b'),
    ('OPERATOR', r'==|!=|<=|>=|<|>|=|\+|\-|\*|/'),
    ('PUNCTUATION', r'[\(\):\[\]\{\}]'),
    ('WHITESPACE', r'\s+'),
]
```

Figure 5: tokens using regular expressions

Each line defines a token type with its own regular expression. Are detected:

- Keywords (`def`, `import`, `if`, `while`, `return`).
- String literals (`"text"`, including f-strings).
- Numeric constants (`123`, `3.14`).
- Identifiers (`var_name`).
- Operators (`+`, `-`, `*`, `==`, `!=` etc.).
- Punctuation marks (`()`, `[]`, {}).
- Whitespaces (just to ignore them later).

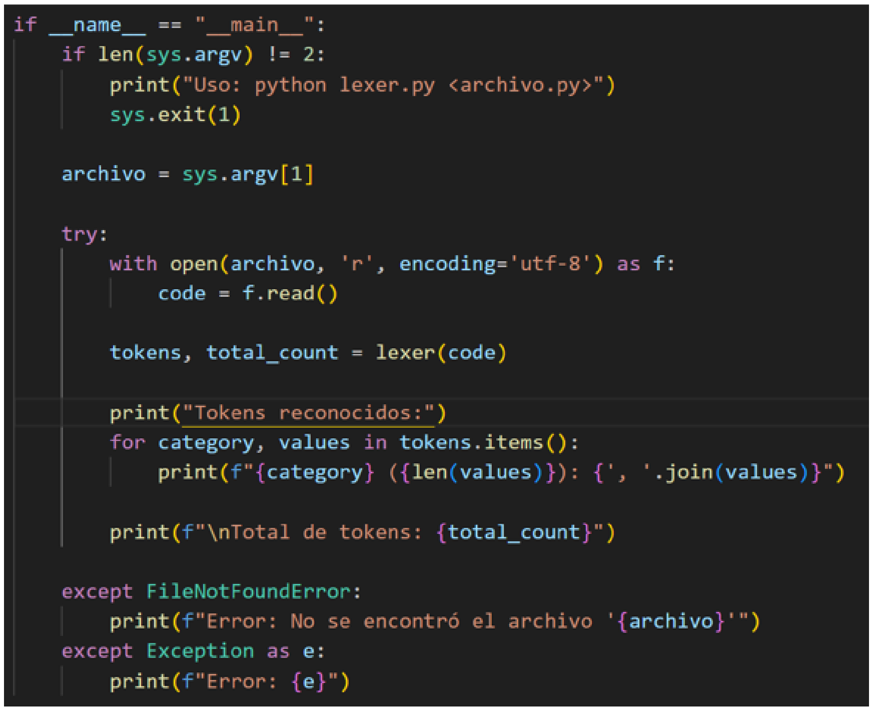
We created a unique regular expression unified where each group has a name, this is useful to be able to classify easily each token in the analysis.



```
token_regex = '|'.join(f'(?P<{name}>{regex})' for name, regex in TOKEN_SPECIFICATION)
```

Figure 6: Unique regular expression unified where each group has a name

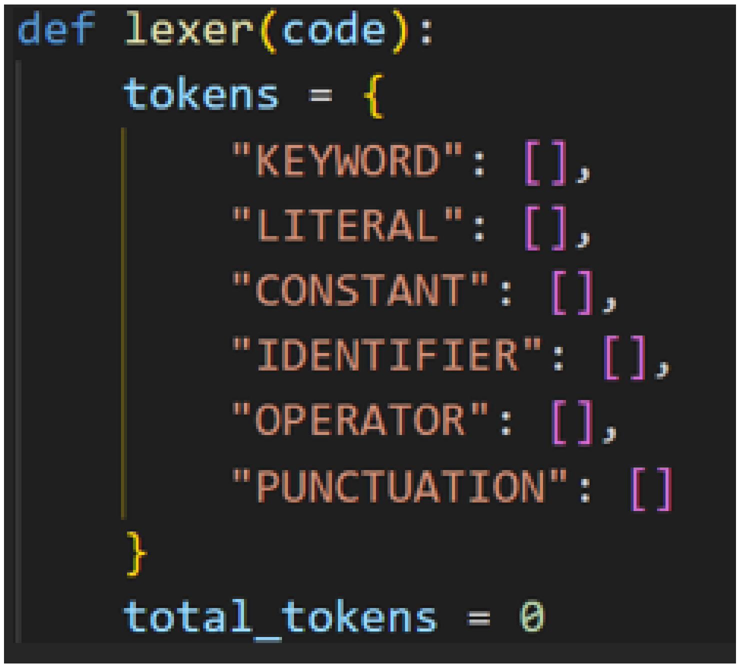
Below, we first verify that the user has provided a file as an argument, if not, we handle errors. If a file was provided as an argument, save its name, open it for reading its content and if not, we handle the error. To avoid problems with accents or special characters we use utf-8. If the file was able to reading, we call lexer() function with the file content and print the results: the tokens classification and the total amount.



```python
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Uso: python lexer.py <archivo.py>")
        sys.exit(1)

    archivo = sys.argv[1]

    try:
        with open(archivo, 'r', encoding='utf-8') as f:
            code = f.read()

        tokens, total_count = lexer(code)

        print("Tokens reconocidos:")
        for category, values in tokens.items():
            print(f"{category} ({len(values)}): {', '.join(values)}")

        print(f"\nTotal de tokens: {total_count}")

    except FileNotFoundError:
        print(f"Error: No se encontró el archivo '{archivo}'")
    except Exception as e:
        print(f"Error: {e}")
```

Figure 7: Main function

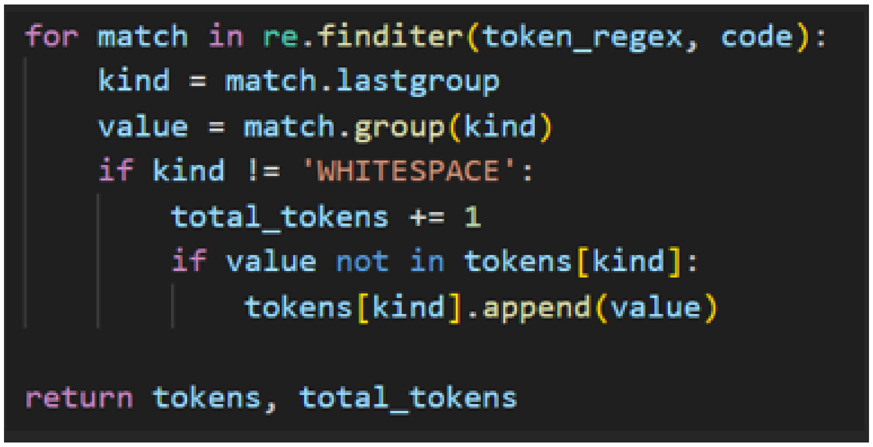
We define the lexer() function to analyze the code:



```python
def lexer(code):
    tokens = {
        "KEYWORD": [],
        "LITERAL": [],
        "CONSTANT": [],
        "IDENTIFIER": [],
        "OPERATOR": [],
        "PUNCTUATION": []
    }
    total_tokens = 0
```

Figure 8: Lexer function.

We create an empty set, here is where we are going to put each token.



```
for match in re.finditer(token_regex, code):
    kind = match.lastgroup
    value = match.group(kind)
    if kind != 'WHITESPACE':
        total_tokens += 1
        if value not in tokens[kind]:
            tokens[kind].append(value)

return tokens, total_tokens
```

Figure 9: Here we are going to put each token.

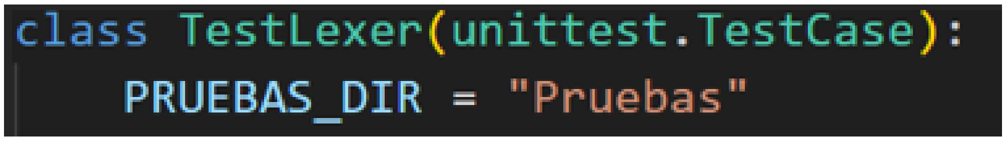
Then we find every token with help of re.finditer(), ignore whitespaces, count the total amount of tokens and save them in the set. Basically, the lexer function counts each token as it finds it. re.finditer() searches for all regex matches in the source code. It ignores whitespaces and classifies each token by its type (keywords, identifiers, etc). Return the token list and the total amount.

Best practices in software engineering

We applied testing to this program using another program `text_lexer`, which uses the `unittest` module, designed to verify the correct functionality of the lexical analyzer (lexer) that was written before.

`class TestLexer(unittest.TestCase)`: Inherits from `unittest.TestCase`, which means that it is a unit test class. It contains tests to verify if `lexer()` classifies the tokens correctly.

We define the folder where the tests are going to be:

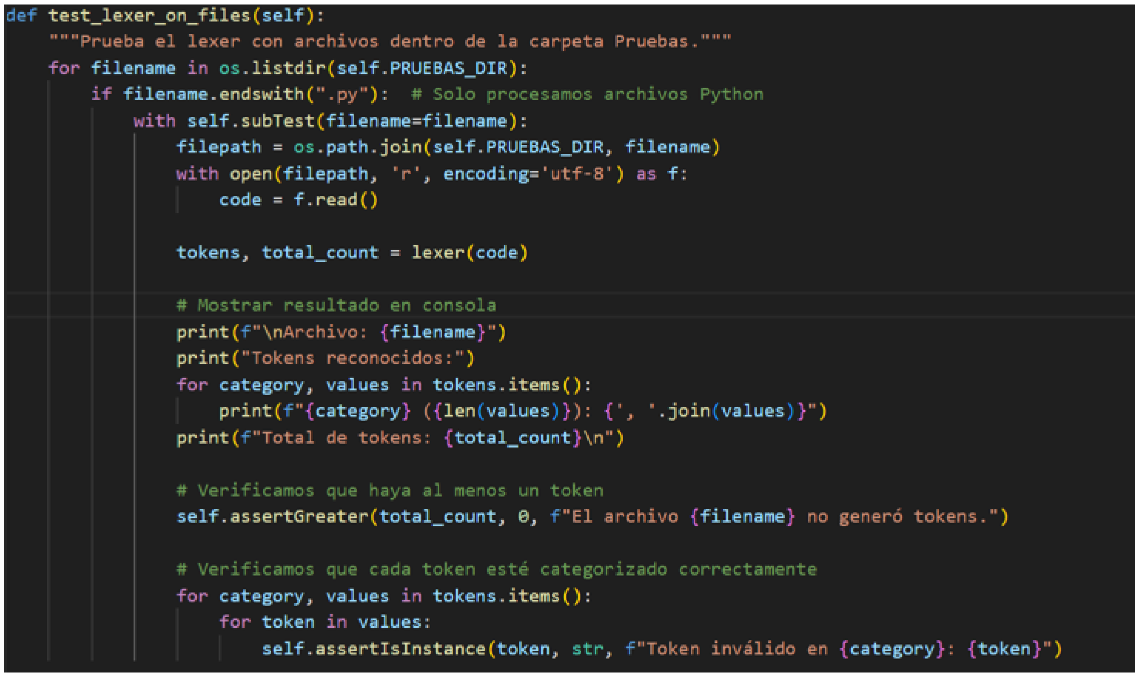


```
class TestLexer(unittest.TestCase):
    PRUEBAS_DIR = "Pruebas"
```

Figure 10: TestLexer function.

We select only the python files, build the absolute path for each one, open the file in reading mode and assign to code the file content. Call lexer() to analyze the code to get the total amount and the classification. Then verify the file has at least 1 token, if not, the test fails.

Finally verify that all the detected tokens are strings, if there are an empty one or bad processed the test also fails.



```python
def test_lexer_on_files(self):
    """Prueba el lexer con archivos dentro de la carpeta Pruebas."""
    for filename in os.listdir(self.PRUEBAS_DIR):
        if filename.endswith(".py"):  # Solo procesamos archivos Python
            with self.subTest(filename=filename):
                filepath = os.path.join(self.PRUEBAS_DIR, filename)
                with open(filepath, 'r', encoding='utf-8') as f:
                    code = f.read()

                tokens, total_count = lexer(code)

                # Mostrar resultado en consola
                print(f"\nArchivo: {filename}")
                print("Tokens reconocidos:")
                for category, values in tokens.items():
                    print(f"{category} ({len(values)}): {', '.join(values)}")
                print(f"Total de tokens: {total_count}\n")

                # Verificamos que haya al menos un token
                self.assertGreater(total_count, 0, f"El archivo {filename} no generó tokens.")

                # Verificamos que cada token esté categorizado correctamente
                for category, values in tokens.items():
                    for token in values:
                        self.assertIsInstance(token, str, f"Token inválido en {category}: {token}")
```

Figure 11: Python unit test for a lexer: Reads and analyzes Python files, verifies token classification, and ensures correctness.

Our code also follows Single Responsibility Principle, which allows us to identify and correct mistakes in classes with only one responsibility, as well reduce the probability of cascading errors, since a change in one class does not affect others.

Grammar example

We have this grammar for an 'if' sentence in python:

$G = ($
$NT = \{S, E, B', B'', B, F, O, T\},$
$T = \{$if, print, el, se, id, "Hola mundo", ==, ¿=, ¡=, !=, ¿, ¡$\},$
$P = \{$
  $S \to \text{if}(E)B'$
  $B' \to B \mid B \text{ else } B \mid B \text{ elif } B$
  $E \to FOF$
  $F \to \text{id}$
  $O \to == \mid >= \mid <= \mid ! = \mid > \mid <$
  $B \to \text{print}(T)$
  $T \to "Holamundo"$
$\},$
$S)$

This grammar has redundance and left recursion, so we are going to apply the left factory and right recursion to get the following grammar:

$G = ($
$NT = \{S, E, B', B'', B, F, O, T\}$,
$T =$ {if, print, el, se, id, ”Hola mundo”, ==, ¿=, ¡=, !=, ¿, ¡},
$P = \{$

$S \to$ if$(E)B''$
$B' \to B \mid B$ el B
$B'' \to$ if$(E)B' \mid$ se$B$
$E \to FOF$
$F \to$ id
$O \to == | >= | <= | ! = | > | <$
$B \to$ print(T)
$T \to$ ”$Holamundo$”

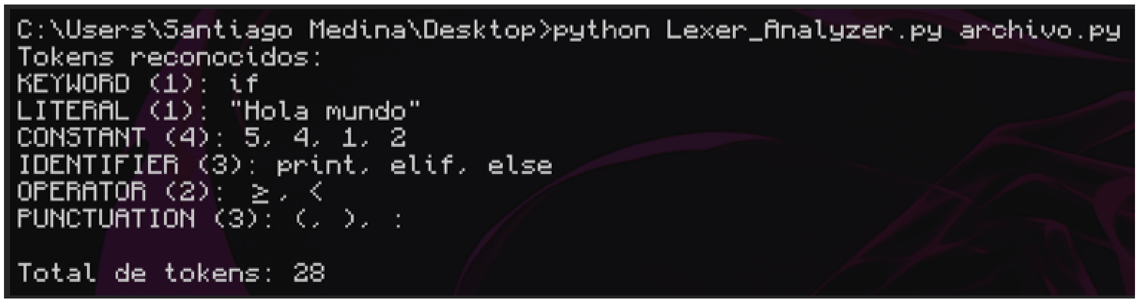
},
$S)$

We finished working with the grammar, so we can generate the automaton:

State (No Terminal)	**Input (Terminal)**	**Following state**
S	if	E
E	(	E
E	id	F
F	==, ¿=, ¡=, !=, ¿, ¡	O
O	id	F
F	)	B’
B’	print	B
B’	if	B”
B”	if	B’
B”	se	B
B	”Hola mundo”	T
T	-	Acceptance

Table 2: Transition Table

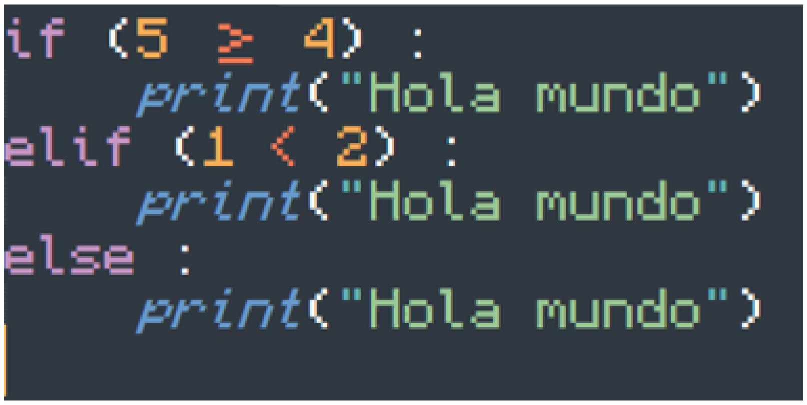
When we run a generated code by the automaton in our Lexical Analyzer we get the following output:



```
C:\Users\Santiago Medina\Desktop>python Lexer_Analyzer.py archivo.py
Tokens reconocidos:
KEYWORD (1): if
LITERAL (1): "Hola mundo"
CONSTANT (4): 5, 4, 1, 2
IDENTIFIER (3): print, elif, else
OPERATOR (2): ≥, <
PUNCTUATION (3): (, ), :

Total de tokens: 28
```

Figure 12: Code output.

Analyzed code:

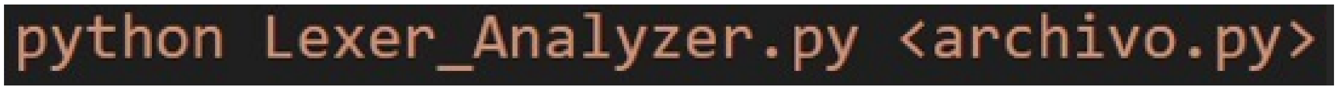


```
if (5 ≥ 4) :
    print("Hola mundo")
elif (1 < 2) :
    print("Hola mundo")
else :
    print("Hola mundo")
```

Figure 13: Program conditionals

4 Results

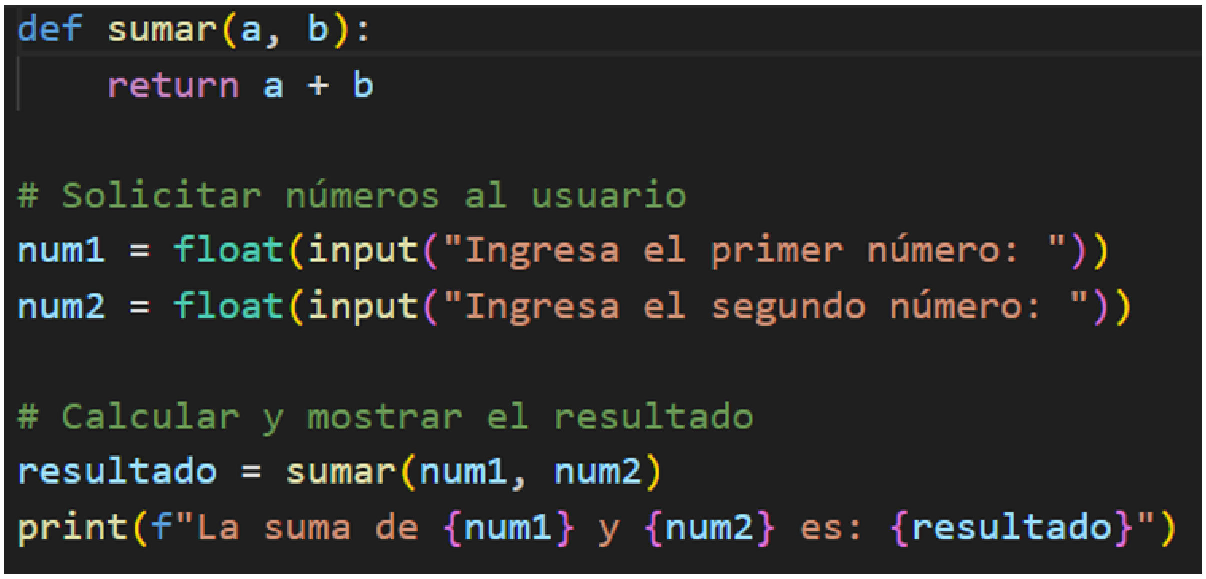
Our program can run as follows:



```
python Lexer_Analyzer.py <archivo.py>
```

Figure 14: How to execute the code

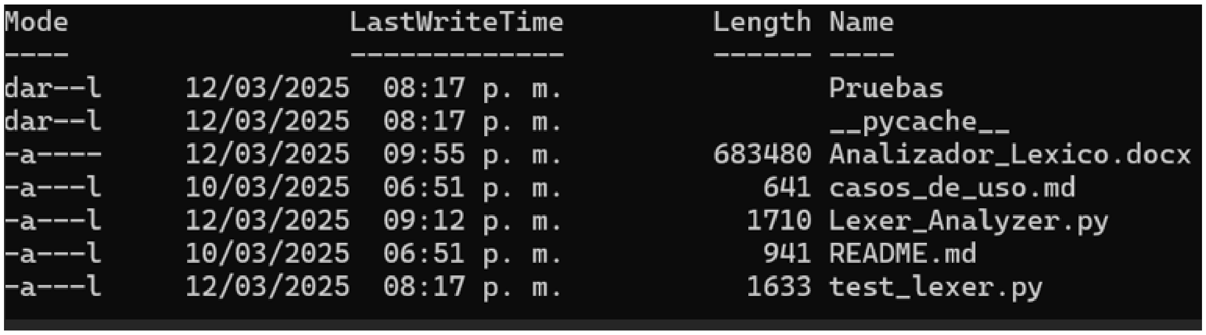
An example is "`python Lexer_Analyzer.py Sumador.py`"
Pass as an argument the file to analyze.
Let's consider the following code fragment:



```
def sumar(a, b):
    return a + b

# Solicitar números al usuario
num1 = float(input("Ingresa el primer número: "))
num2 = float(input("Ingresa el segundo número: "))

# Calcular y mostrar el resultado
resultado = sumar(num1, num2)
print(f"La suma de {num1} y {num2} es: {resultado}")
```

Figure 15: Sumar function

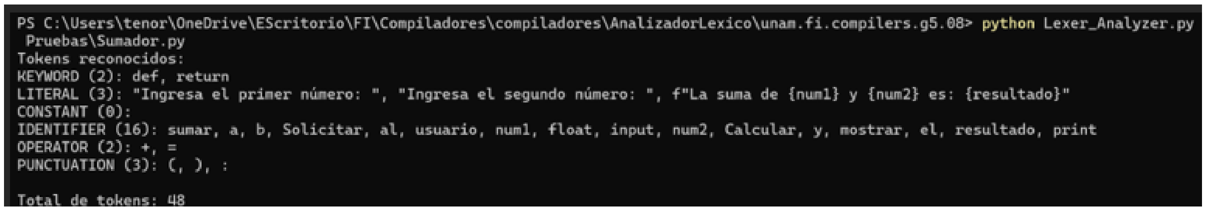
This is contained in the test file carpet.



```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
dar--l     12/03/2025  08:17 p. m.                Pruebas
dar--l     12/03/2025  08:17 p. m.                __pycache__
-a----     12/03/2025  09:55 p. m.         683480 Analizador_Lexico.docx
-a---l     10/03/2025  06:51 p. m.            641 casos_de_uso.md
-a---l     12/03/2025  09:12 p. m.           1710 Lexer_Analyzer.py
-a---l     10/03/2025  06:51 p. m.            941 README.md
-a---l     12/03/2025  08:17 p. m.           1633 test_lexer.py
```

Figure 16: File carpet.

So, the program runs as follows:



```
PS C:\Users\tenor\OneDrive\EScritorio\FI\Compiladores\compiladores\AnalizadorLexico\unam.fi.compilers.g5.08> python Lexer_Analyzer.py
 Pruebas\Sumador.py
Tokens reconocidos:
KEYWORD (2): def, return
LITERAL (3): "Ingresa el primer número: ", "Ingresa el segundo número: ", f"La suma de {num1} y {num2} es: {resultado}"
CONSTANT (0):
IDENTIFIER (16): sumar, a, b, Solicitar, al, usuario, num1, float, input, num2, Calcular, y, mostrar, el, resultado, print
OPERATOR (2): +, =
PUNCTUATION (3): (, ), :

Total de tokens: 48
```

Figure 17: File carpet.

The program immediately will classify and count the tokens of the .py file.

5 Conclusions

While building the lexical analyzer, we learned several important points about how this phase compilation works, such as identifying and classifying parts of the code into tokens. We also defined regular expressions. We applied the SRP principle, as the initial code which mixes multiple responsibilities in the lexer() function. We improved it by separating rule definition, regular expressions generation and tokens analysis. We used unittests for testing our code and ensure that it classifies the tokens correctly. In conclusion, we are ready to move on to the parser.

References

[1] A. V. Aho, R. Sethi, y J. D. Ullman, *Compiladores: principios, técnicas y herramientas*. Pearson Educación, 1990.

[2] I. R. Martínez y L. Olivé, *Compiladores*. Epistemología evolucionista, 1977.

[3] A. V. Aho, M. S. Lam, y J. D. Ullman, *Compilers: Principles, Techniques & Tools*. Pearson Education, 2007.

[4] Python Software Foundation, “re — Operaciones con expresiones regulares,” Documentación de Python 3.13. [En línea]. Disponible en: `https://docs.python.org/es/3.13/library/re.html`. [Accedido: 13-mar-2025].