Un dibujo de una persona

El contenido generado por IA puede ser incorrecto.

**Universidad Nacional Autónoma de México**

**Computer Engineering**

**Compilers**

**Lexical Analyzer**

**Students:**

**Jiménez Elizalde Josue - 320334489**

**Medina Guzmán Santiago - 320067354**

**Tavera Castillo David Emmanuel - 320054831**

**Tenorio Martinez Jesus Alejandro – 320218666**

**Group: 5**

**Semester: 2025-2**

**México, CDMX. March 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 theorical 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 the classification of all lexemes.

1. **Theoretical framework**

The lexical analyzer, also known as 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).

The 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 produces results; and literals, such as logical, numeric, textual or reference values.

Examples of each category:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

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

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

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

*Left recursion*

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



It can be corrected as follows:

Diagrama

El contenido generado por IA puede ser incorrecto.

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

*Ambiguity*

The ambiguity occures when a input string can be derived in more than one way.

For example:

Imagen que contiene Gráfico

El contenido generado por IA puede ser incorrecto.

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

Tabla

El contenido generado por IA puede ser incorrecto.

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

1. **Development**

We developed a pyhton code that uses the library re, which is a “module that allows working with regular expressions (regex). Regular expressions are a useful tool for searching, extracting and manipulating text patterns.”

In the first part of the code, we define:

A screen shot of a computer code

AI-generated content may be incorrect.

**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).

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).

*A screen shot of a computer program

AI-generated content may be incorrect.*

*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.

def test\_variables\_y\_bucle(self): Tests that the lexer detects variables and control structures (while).

Texto

El contenido generado por IA puede ser incorrecto.

Code is used as a test code, results are verified using assert, and if all assertions are true, the test passes.

def test\_import\_y\_funcion(self): Test the detection import, def and return in functions. It executes the lexer with Code and verifies with assert. If all verifications pass, the test is successful.

Texto

El contenido generado por IA puede ser incorrecto.

Our code also follows Single Responsibility Principle, which allows 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.

1. **Results**

Running our program with the input:

Texto

El contenido generado por IA puede ser incorrecto.

The output is:

Texto

El contenido generado por IA puede ser incorrecto.

Which was the expected output.

1. **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.

**Referencias**

* *A. V. Aho, R. Sethi, y J. D. Ullman, Compiladores: principios, técnicas y herramientas. Pearson Educación, 1990.*
* *I. R. Martínez y L. Olivé, Compiladores. Epistemología evolucionista, 1977.*
* *A. V. Aho, M. S. Lam, y J. D. Ullman, Compilers: Principles, Techniques & Tools. Pearson Education, 2007.*