**Universidad de Guadalajara**



Ingeniería en Computación

Compiladores

**Practica 1 – Analizador Léxico**

28-FEBRERO-2025

Roberto Carlos Rodríguez Rivas

En esta práctica nos encargaremos de desarrollar un programa en Python que se encargue de realizar un análisis léxico de diferentes cadenas para poder corroborar que las cadenas que introduzcamos sean permitidas en el lenguaje que el maestro previamente y definió. Esta actividad será la primera de 4 practicas, para poder crear un compilador funcional al final del curso.

**Explicación del código**

En el desarrollo del código decidí utilizar una librería llamada ‘***collections’*** para que me permitiera crear un diccionario que devuelve un valor por defecto si una clave no existe. Decidí trabajar con una clase llamada ‘***lexer’*** que tendrá el será nuestro analizador léxico, dentro de la clase defino una lista de tokens con sus respectivas expresiones regulares y a cada token le asigno un numero para identificar su tipo (un diccionario). Creo también 1 diccionario para contar cuantas veces aparece cada tipo de token, creo dos listas, la primera para almacenar los errores léxicos encontrados y la segunda para almacenar los tokens reconocidos junto con su tipo.

Creo una función llamada ‘***analize’*** que se encargará de la cadena de entrada y extraerá los tokens. Dentro de la función me encargo de recorrer los caracteres que analizaré, siempre ignorando los espacios. Si encuentro un match en la posición actual con ayuda del regex, extraigo el token, lo guardo en la lista de coincidencias e incremento el contador de este tipo de token, muevo la posición hacia delante según el tamaño del de token encontrado, marcamos que encontramos un token y salimos del bucle. En el caso de no encontrar ningún token valido en la posición actual lo consideramos un error léxico y seguimos analizando el siguiente.

**Código fuente con comentarios:**

# Importamos la librería 're' para trabajar con expresiones regulares

import re

# Importamos 'defaultdict' de la librería 'collections'

# Nos permite crear un diccionario que devuelve un valor por defecto si una clave no existe

from collections import defaultdict

# Definimos la clase 'Lexer', que será nuestro analizador léxico

class Lexer:

    def \_\_init\_\_(self):

        """

        Constructor de la clase Lexer.

        Define los tokens a reconocer y crea estructuras para almacenar los resultados.

        """

        # Lista de tokens con sus respectivas expresiones regulares

        # Cada token tiene un número asociado para identificar su tipo

        self.tokens = [

            (0, r'\b(int|float|char|void|string)\b'),  # Tipos de datos

            (10, r'\bif\b'),  # Palabra clave "if"

            (11, r'\bwhile\b'),  # Palabra clave "while"

            (12, r'\breturn\b'),  # Palabra clave "return"

            (13, r'\belse\b'),  # Palabra clave "else"

            (14, r'\bfor\b'),  # Palabra clave "for"

            (2, r'\b\d+(\.\d+)?\b|\bpi\b|#'),  # Constantes numéricas y la constante 'pi'

            (1, r'\b[a-zA-Z\_][a-zA-Z0-9\_]\*\b'),  # Identificadores (variables, nombres de funciones)

            (3, r';'),  # Punto y coma (;)

            (4, r','),  # Coma (,)

            (5, r'\('),  # Paréntesis izquierdo '('

            (6, r'\)'),  # Paréntesis derecho ')'

            (7, r'\{'),  # Llave izquierda '{'

            (8, r'\}'),  # Llave derecha '}'

            (9, r'='),  # Operador de asignación '='

            (15, r'[\+\-]'),  # Operadores aritméticos de suma/resta ('+', '-')

            (16, r'[\\*/<>]{1,2}'),  # Operadores multiplicativos y relacionales ('\*', '/', '<', '>')

            (17, r'&&|\|\|'),  # Operadores lógicos ('&&', '||')

            (18, r'[<>]=?|==|!='),  # Operadores relacionales ('<', '>', '>=', '<=', '==', '!=')

            (19, r'\$'),  # Símbolo especial ('$')

        ]

        # Diccionario para contar cuántas veces aparece cada tipo de token

        self.token\_counts = defaultdict(int)

        # Lista para almacenar los errores léxicos encontrados

        self.errors = []

        # Lista donde almacenaremos los tokens reconocidos junto con su tipo

        self.matches = []

    def analyze(self, input\_string):

        """

        Analiza una cadena de entrada y extrae los tokens.

        """

        pos = 0

        # Recorremos toda la cadena mientras queden caracteres por analizar

        while pos < len(input\_string):

            # Ignoramos los espacios en blanco y tabulaciones

            if input\_string[pos].isspace():

                pos += 1

                continue

            match\_found = False  # Variable para saber si encontramos un token válido en esta posición

            # Recorremos la lista de tokens definidos para intentar hacer una coincidencia

            for token\_type, patron in self.tokens:

                regex = re.compile(patron)  # Compilamos la expresión regular

                match = regex.match(input\_string, pos)  # Intentamos encontrar un match en la posición actual

                # Si encontramos un token válido

                if match:

                    lexeme = match.group(0)  # Extraemos el texto del token

                    self.matches.append((lexeme, token\_type))  # Guardamos el token en la lista de coincidencias

                    self.token\_counts[token\_type] += 1  # Incrementamos el contador de este tipo de token

                    pos += len(lexeme)  # Movemos la posición hacia adelante según el tamaño del token encontrado

                    match\_found = True  # Marcamos que encontramos un token

                    break  # Salimos del bucle porque ya encontramos un token en esta posición

            # Si no encontramos ningún token válido en la posición actual, lo consideramos un error léxico

            if not match\_found:

                self.errors.append(f"Error léxico en posición {pos}: '{input\_string[pos]}'")

                pos += 1  # Avanzamos al siguiente carácter para seguir analizando

    def display\_results(self):

        """

        Muestra los resultados del análisis léxico: tokens encontrados, conteo por categoría y errores.

        """

        print("\nTokens encontrados:")

        # Mostramos todos los tokens encontrados junto con su tipo

        for lexeme, token\_type in self.matches:

            print(f"{lexeme} -> {token\_type}")

        print("\nCantidad de tokens por categoría:")

        # Mostramos cuántos tokens de cada tipo fueron encontrados

        for token\_type, count in sorted(self.token\_counts.items()):

            print(f"Categoría {token\_type}: {count}")

        # Si hubo errores léxicos, los mostramos

        if self.errors:

            print("\nErrores encontrados:")

            for error in self.errors:

                print(error)

# Bloque principal: solo se ejecuta si el script es ejecutado directamente

if \_\_name\_\_ == "\_\_main\_\_":

    input\_string = input("Ingrese el código a analizar: ")  # Pedimos al usuario que ingrese código fuente

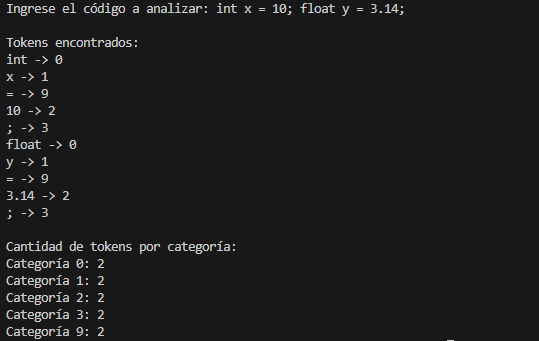
    lexer = Lexer()  # Creamos una instancia del analizador léxico

    lexer.analyze(input\_string)  # Analizamos la entrada del usuario

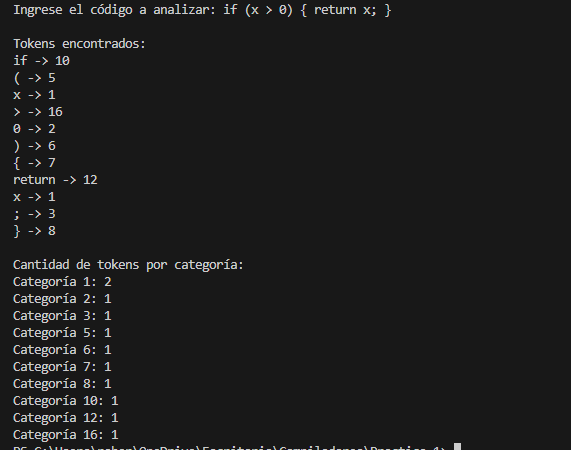
    lexer.display\_results()  # Mostramos los resultados del análisis

**Ejemplos de ejecución con capturas de pantalla.**

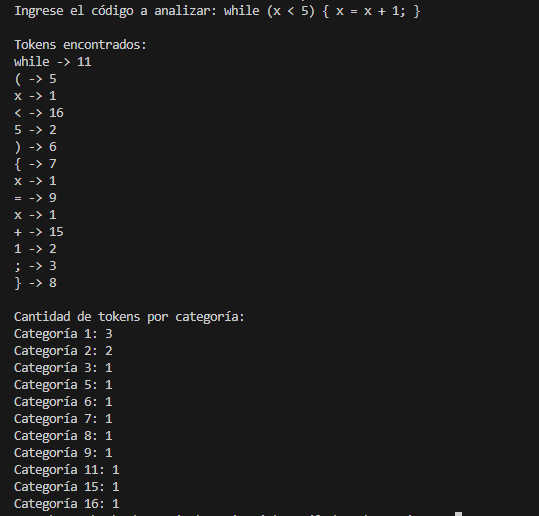
* **int x = 10; float y = 3.14;**



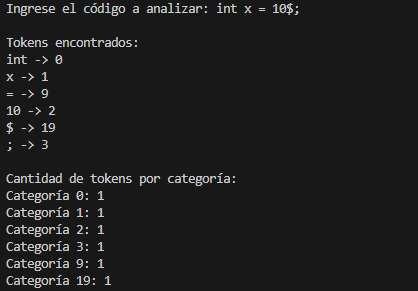
* **if (x > 0) { return x; }**



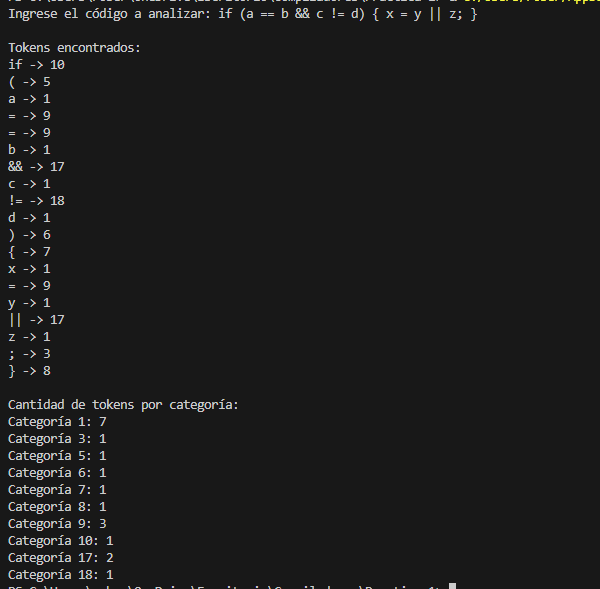
* **while (x < 5) { x = x + 1; }**



* **int x = 10$;**



* **if (a == b && c != d) { x = y || z; }**



**Justifique los resultados obtenidos.**

El desarrollo y prueba del analizador léxico permitieron verificar su correcto funcionamiento en la identificación de tokens y detección de errores. Se confirmó que el analizador reconoce adecuadamente los tipos de datos (int, float, etc.), identificadores (x, variable1), operadores (=, +, -), símbolos (;, {, }) y constantes numéricas, incluyendo flotantes (3.14, 0.5).

Los resultados obtenidos confirman que el analizador cumple su función de convertir una secuencia de caracteres en una lista de tokens y reportar errores léxicos. Este proceso es fundamental en la compilación, ya que proporciona una base estructurada para la siguiente fase del análisis sintáctico.