INSTITUTO TECNOLOGICO DE MEXICO

CAMPUS PACHUCA

LENGUAJES Y AUTOMATAS

4.2 Documentación AR y AF para Analizador léxico

Baume Lazcano Rodolfo

Espinosa Antelis Gabriela Dyvheke

Garcia Cruz Andres

14 MAYO 2024

El análisis léxico es la primera etapa en el proceso de compilación de un programa, donde se convierte el flujo de caracteres de entrada en una secuencia de tokens significativos para el compilador. Este reporte analiza el código de un analizador léxico implementado en Python utilizando expresiones regulares y diccionarios para reconocer tokens específicos en el texto de entrada.

Utilización de tabla de tokens:

- El código utiliza diccionarios para definir distintas categorías de tokens, tales como operadores, palabras reservadas, tipos de datos, símbolos de puntuación y variables. Estos diccionarios son estructuras de datos clave-valor que mapean un token (la clave) a su categoría correspondiente (el valor). Por ejemplo:

operators = {'=': 'Operador de Asignación', '+': 'Operador de Suma', '-': 'Operador de Resta', ...}

- Cada diccionario proporciona una lista completa y organizada de los tokens que el analizador debe reconocer en el texto de entrada, facilitando así la identificación y categorización de los mismos.

Escritura de reglas de coincidencia:

- Se emplean condicionales if para verificar si un token específico se encuentra dentro de los diccionarios de tokens previamente definidos. Por ejemplo:

if token in operators: tokens.append((line_count, 'Operador', operators[token]))

- Estas reglas de coincidencia determinan cómo se reconocen los distintos tipos de tokens en el texto de entrada, identificando cada token y asignándole su categoría correspondiente.

Asociación de acciones a las reglas:

- Cada regla de coincidencia está asociada con una acción específica que se ejecuta cuando se reconoce un token según la regla correspondiente. Por ejemplo, cuando se detecta un operador, se agrega el token a la lista de tokens junto con su categoría. Esto se realiza para cada tipo de token reconocido en el texto de entrada.
- Las acciones asociadas a cada regla garantizan que los tokens sean correctamente identificados y categorizados de acuerdo a su significado y función en el código fuente.

Manejo de casos especiales:

- Se han considerado casos especiales, como tokens que no coinciden con ningún patrón conocido, y se manejan adecuadamente etiquetándolos como "No se encontró patrón". Esto se logra mediante un condicional else al final del proceso de análisis léxico, que captura cualquier token que no coincida con ninguna de las reglas previamente definidas.
- Además, se resuelven ambigüedades en las reglas de análisis mediante condiciones específicas que priorizan ciertos patrones sobre otros, asegurando así un análisis léxico preciso y consistente. Por ejemplo, la diferenciación entre un numero entero, un numero de teléfono o un código postal.

Pruebas de las reglas:

- Durante el desarrollo del analizador léxico, se llevan a cabo pruebas exhaustivas utilizando una variedad de ejemplos de texto de entrada para asegurar su correcto funcionamiento. Se prueban diferentes combinaciones de tokens, así como casos límite y situaciones de error para validar la efectividad y precisión del analizador.
- Se ajustan las reglas según sea necesario durante las pruebas para corregir cualquier error o comportamiento inesperado encontrado, garantizando así que el analizador léxico funcione correctamente en una variedad de situaciones y escenarios.

DOCUMENTACION DEL CODIGO

```
import re  # Importa el módulo para trabajar con expresiones regulares
import tkinter as tk  # Importa la biblioteca para crear interfaces gráficas
from tkinter import ttk  # Importa un módulo específico para widgets de ttk

# Diccionarios de tokens y sus categorías
operators = {'=': 'Operador de Asignación', '+': 'Operador de Suma', '-':
'Operador de Resta', '/': 'Operador de División', '*': 'Operador de
Multiplicación', '<': 'Operador Menor que', '>': 'Operador Mayor que'}  #
Define un diccionario para operadores y sus categorías
reservwords = {
    'if': 'Condición',
    'then': 'Entonces',
    'else': 'Sino',
    'case': 'Caso',
    'switch': 'Alterna',
    'while': 'Mientras',
    'for': 'Para'
}
para palabras reservadas y sus categorías
data_type = {'int': 'Entero', 'float': 'Flotante', 'char': 'Carácter',
'long': 'Entero Largo'}  # Define un diccionario para tipos de datos y sus
categorías
punctuation_symbol = {':': 'Dos Puntos', ';': 'Punto y Coma', '.': 'Punto',
',': 'Coma'}  # Define un diccionario para símbolos de puntuación y sus
categorías
variables = {chr(i): 'Variable' for i in range(ord('a'), ord('z') + 1)}  #
Define un diccionario para variables, asignándoles la categoría "Variable"

# Función para el análisis léxico
def lexico(code):
    tokens = []  # Inicializa una lista para almacenar los tokens
    line_count = 0  # Inicializa un contador de líneas

    program = code.split("\n")  # Divide el código en líneas
    for line in program:
        if line.strip() == "":  # Ignora las líneas vacías
            continue
        line_count += 1  # Incrementa el contador de líneas
        line_tokens = line.split(' ')  # Divide la línea en tokens

        for token in line_tokens:
            if re.match(r'^__([a-zA-Z][a-zA-Z0-9]*)*$', token):  # Expresión
regular para Identificador
```

```
                tokens.append((line_count, 'Identificador', token))  #
Agrega el token a la lista con su categoría
            elif re.match(r'^\/\/[a-zA-Z0-9]*$', token):  # Expresión
regular para Comentario
                tokens.append((line_count, 'Comentario', token))  # Agrega
el token a la lista con su categoría
            elif re.match(r'^(?:=|\+|-|\/|\*|<|%|=|>=|<=|>|&|\|\|){0,2}$',
token):  # Expresión regular para Símbolos
                tokens.append((line_count, 'Simbolos', token))  # Agrega el
token a la lista con su categoría
            elif token in operators:  # Si el token es un operador
                tokens.append((line_count, 'Operador', operators[token]))  #
Agrega el token a la lista con su categoría
            elif token in data_type:  # Si el token es un tipo de dato
                tokens.append((line_count, 'Tipo de Dato',
data_type[token]))  # Agrega el token a la lista con su categoría
            elif token in punctuation_symbol:  # Si el token es un símbolo
de puntuación
                tokens.append((line_count, 'Símbolo de Puntuación',
punctuation_symbol[token]))  # Agrega el token a la lista con su categoría
            elif token in reservwords:  # Si el token es una palabra
reservada
                tokens.append((line_count, 'Palabra Reservada',
reservwords[token]))  # Agrega el token a la lista con su categoría
            elif token in variables:  # Si el token es una variable
                tokens.append((line_count, 'Variable', variables[token]))  #
Agrega el token a la lista con su categoría
            elif re.match(r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-
Z]{2,}$', token):  # Si el token es un correo electrónico
                tokens.append((line_count, 'Correo Electrónico', token))  #
Agrega el token a la lista con su categoría
            elif re.match(r'^\d{10}$', token):  # Si el token es un número
de teléfono
                tokens.append((line_count, 'Número de Teléfono', token))  #
Agrega el token a la lista con su categoría
            elif re.match(r'^\d{5}$', token) and token.startswith('42'):  #
Si el token es un código postal que comienza con '42'
                tokens.append((line_count, 'Código Postal', token))  #
Agrega el token a la lista con su categoría
            elif re.match(r'^\d+$', token):  # Si el token es un número
entero
                tokens.append((line_count, 'Número Entero', token))  #
Agrega el token a la lista con su categoría
            elif re.match(r'^\d+\.\d+$', token):  # Si el token es un número
flotante
```

```
                tokens.append((line_count, 'Número Flotante', token))  # Agrega el token a la lista con su categoría
            elif re.match(r'^\s*\d+(\.\d+)?(\s*[\+\-\*/]\s*\d+(\.\d+)?)+\s*$', token):  # Si el token es una operación matemática compuesta
                unique_operators = set(re.findall(r'[\+\-\*/]', token))  # Encuentra los operadores únicos en la expresión
                if len(unique_operators) > 1:  # Si hay más de un tipo de operador
                    tokens.append((line_count, 'Operación', 'Operación Compuesta', token))  # Agrega el token a la lista como una operación compuesta
                else:  # Si solo hay un tipo de operador
                    operator = next(iter(unique_operators))  # Obtiene el operador
                    tokens.append((line_count, 'Operación', operators[operator]))  # Agrega el token a la lista como una operación simple
            else:
                tokens.append((line_count, 'No se encontró patrón', token))  # Si no coincide con ningún patrón conocido, agrega el token con la categoría "No se encontró patrón"

    return tokens  # Devuelve la lista de tokens

def update_lexical_table(code, table):
    table.delete(*table.get_children())  # Limpia la tabla
    code = code.strip()  # Elimina las líneas vacías al final
    tokens = lexico(code)  # Realiza el análisis léxico del código
    for token in tokens:
        if token[2] == 'No se encontró patrón':
            table.insert('', tk.END, values=(token[0], token[1], 'No se encontró patrón'))  # Agrega el token a la tabla
        else:
            table.insert('', tk.END, values=token)  # Agrega el token a la tabla

def create_gui():
    # Configuración de la ventana principal
    app = tk.Tk()  # Crea la ventana principal
    app.geometry('900x700')  # Establece el tamaño de la ventana
    app.title('Analizador Léxico')  # Establece el título de la ventana
    app.configure(bg='#860e4e')  # Establece el color de fondo

    # Estilo para la tabla
```

```
    style = ttk.Style()  # Crea un objeto de estilo
    style.theme_use("clam")  # Usa un tema de estilo específico
    style.configure("Treeview", background="#dedede",
fieldbackground="#dedede", foreground="black", font=('Century Gothic',
12))  # Configura el estilo de la tabla

    # Etiqueta de título
    title_label = tk.Label(app, text="Analizador Léxico", font=('Century
Gothic', 24), bg='#860e4e', fg='white')  # Crea una etiqueta para el título
    title_label.pack(pady=20)  # Empaqueta la etiqueta en la ventana
principal

    # Área de entrada de texto
    text_frame = tk.Frame(app, bg='#860e4e')  # Crea un marco para el área
de entrada de texto
    text_frame.pack(pady=5)  # Empaqueta el marco en la ventana principal

    input_label = tk.Label(text_frame, text='Expresión:', font=('Century
Gothic', 14), bg='#860e4e', fg='white')  # Crea una etiqueta para la entrada
de texto
    input_label.pack()  # Empaqueta la etiqueta en el marco

    text_input = tk.Text(text_frame, width=60, height=7, font=('Century
Gothic', 12))  # Crea un área de texto para la entrada
    text_input.pack()  # Empaqueta el área de texto en el marco

    # Sección de salida léxica
    output_frame = tk.Frame(app, bg='#dedede')  # Crea un marco para la
salida léxica
    output_frame.pack(pady=20)  # Empaqueta el marco en la ventana principal

    output_label = tk.Label(output_frame, text='Resultado del Análisis
Léxico', font=('Century Gothic', 16), bg='#dedede', fg='#860e4e')  # Crea
una etiqueta para la salida léxica
    output_label.pack()  # Empaqueta la etiqueta en el marco

    # Creando tabla de datos
    table_frame = tk.Frame(output_frame, bg='#dedede')  # Crea un marco para
la tabla de datos
    table_frame.pack()  # Empaqueta el marco en la salida léxica

    columns = ('Línea', 'Tipo', 'Token')  # Define las columnas de la tabla
    token_table = ttk.Treeview(table_frame, columns=columns,
show='headings')  # Crea una tabla de árbol con las columnas especificadas
```

```
    token_table.heading('Línea', text='Línea')  # Configura la cabecera de
la columna 'Línea'
    token_table.heading('Tipo', text='Tipo')  # Configura la cabecera de la
columna 'Tipo'
    token_table.heading('Token', text='Token')  # Configura la cabecera de
la columna 'Token'

    # Establecer el ancho de las columnas
    total_width = 800  # Ancho total de la tabla
    line_width = total_width // 9  # Ancho de la columna 'Línea'
    type_width = total_width * 4 // 9  # Ancho de la columna 'Tipo'
    token_width = total_width * 4 // 9  # Ancho de la columna 'Token'

    # Configurar el peso de cada columna
    token_table.column('Línea', width=line_width, anchor=tk.CENTER)  #
Configura el ancho y la alineación de la columna 'Línea'
    token_table.column('Tipo', width=type_width, anchor=tk.CENTER)  #
Configura el ancho y la alineación de la columna 'Tipo'
    token_table.column('Token', width=token_width, anchor=tk.CENTER)  #
Configura el ancho y la alineación de la columna 'Token'

    token_table.pack(side='left')  # Empaqueta la tabla en el marco

    scrollbar = ttk.Scrollbar(table_frame, orient=tk.VERTICAL,
command=token_table.yview)  # Crea una barra de desplazamiento vertical para
la tabla
    token_table.configure(yscroll=scrollbar.set)  # Configura la tabla para
que pueda desplazarse verticalmente
    scrollbar.pack(side='right', fill='y')  # Empaqueta la barra de
desplazamiento en el lado derecho del marco

    # Botón de validación
    validate_frame = tk.Frame(app, bg='#860e4e')  # Crea un marco para el
botón de validación
    validate_frame.pack(pady=10)  # Empaqueta el marco en la ventana
principal

    validate_button = tk.Button(validate_frame, text='Validar',
font=('Century Gothic', 14), bg='white', fg='black', relief=tk.FLAT,
command=lambda: update_lexical_table(text_input.get("1.0", tk.END),
token_table))  # Crea un botón de validación
    validate_button.pack()  # Empaqueta el botón en el marco de validación

    # Cambiar color del botón al pasar el puntero
```

```
    validate_button.bind("<Enter>", lambda e:
validate_button.config(bg='black', fg='white'))  # Cambia el color del botón
al pasar el puntero sobre él
    validate_button.bind("<Leave>", lambda e:
validate_button.config(bg='white', fg='black'))  # Restaura el color
original del botón al salir el puntero

    # Pie de página
    footer_frame = tk.Frame(app, bg='#860e4e')  # Crea un marco para el pie
de página
    footer_frame.pack(pady=10)  # Empaqueta el marco en la ventana principal

    footer_label = tk.Label(footer_frame, text='Unidad 4', font=('Century
Gothic', 12, 'italic'), bg='#860e4e', fg='white')  # Crea una etiqueta para
el pie de página
    footer_label.pack()  # Empaqueta la etiqueta en el marco

    # Centrar la ventana en la pantalla
    app.update_idletasks()  # Actualiza las tareas pendientes en la ventana
    width = app.winfo_width()  # Obtiene el ancho de la ventana
    height = app.winfo_height()  # Obtiene la altura de la ventana
    x = (app.winfo_screenwidth() // 2) - (width // 2)  # Calcula la posición
x para centrar la ventana
    y = (app.winfo_screenheight() // 2) - (height // 2)  # Calcula la
posición y para centrar la ventana
    app.geometry('+{}+{}'.format(x, y))  # Establece la geometría de la
ventana para centrarla

    app.mainloop()  # Ejecuta el bucle principal de la aplicación

# Iniciar la interfaz gráfica
create_gui()  # Llama a la función para crear la interfaz gráfica
```