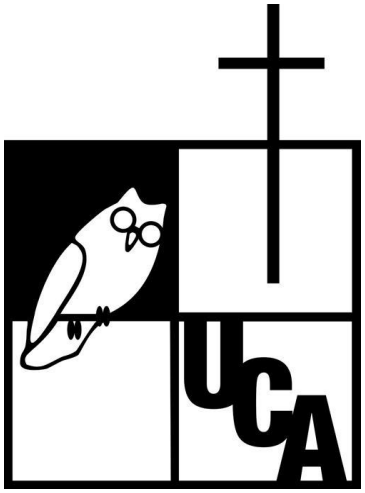
UNIVERSIDAD CENTROAMERICANA “JOSÉ SIMEÓN CAÑAS”



TEORÍAS DE LENGUAJES DE PROGRAMACIÓN

Catedrático: Jaime Roberto Clímaco

Analizador Lexicográfico Python para C++

Integrantes:
Grupo N 5

Anibal Ernesto Hernández Garcia	00401117
Rafael Alejandro Melara Garcia	00182516
Daniel Van Gabarrete Abrego	00015218
Nelson Manuel Garcia Flamenco	00088113
Rodrigo Arevalo Najarro	00303318
Mario Orlando Moisa Candray	00135115

INTRODUCCIÓN

Las extensiones punto cpp son un tipo de archivo que se utiliza para escribir código en el lenguaje de programación C++. Este lenguaje ofrece numerosas ventajas para el desarrollo de aplicaciones, como la eficiencia, la portabilidad, la orientación a objetos y la compatibilidad con C. Sin embargo, para aprovechar al máximo estas ventajas, es importante seguir unas buenas prácticas de programación que faciliten la comprensión, el mantenimiento y la depuración del código. Una de estas buenas prácticas es el uso de convenciones de codificación.

Las convenciones de codificación son un conjunto de reglas o normas que especifican cómo escribir, formatear y organizar el código fuente de un programa. Estas reglas pueden variar según el estilo personal, el equipo de trabajo o el proyecto, pero en general tienen como objetivo mejorar la legibilidad, la consistencia y la calidad del código. Algunos ejemplos de convenciones de codificación son:

- Usar nombres significativos y consistentes para las variables, funciones y clases, que reflejen su propósito y su tipo.
- Usar comentarios para explicar el objetivo y la lógica del código, especialmente cuando se trata de secciones complejas o poco claras.
- Usar sangrías y espacios en blanco para mejorar la estructura visual del código, respetando los niveles de anidación y alineando los elementos relacionados.
- Usar mayúsculas y minúsculas de forma coherente para diferenciar los identificadores, siguiendo alguna convención establecida como camelCase o snake_case.
- Usar paréntesis para agrupar las expresiones y evitar ambigüedades, especialmente cuando se usan operadores con distinta precedencia o asociatividad.

El uso de convenciones de codificación facilita la lectura y comprensión del código por parte de otros programadores, así como por parte del propio autor. Además, también ayuda a detectar y corregir posibles errores sintácticos o semánticos en el código.

Para verificar que el código cumple con las convenciones de codificación establecidas, se puede recurrir a herramientas automáticas que analizan el código fuente y lo validan según ciertos criterios. Una de estas herramientas es el analizador léxico.

El analizador léxico es la primera fase de un compilador que toma el código fuente como entrada y lo divide en tokens, que son las unidades mínimas de un lenguaje de programación, como palabras clave, identificadores, números, operadores, etc. Además, el lexer también implementa una tabla de símbolos para almacenar información sobre los tokens encontrados.

El analizador léxico funciona mediante el uso de expresiones regulares, que son patrones que describen conjuntos de cadenas de caracteres. Estas expresiones regulares se utilizan para definir las reglas léxicas del lenguaje de programación, es decir, cómo se forman los tokens válidos. Por ejemplo, una expresión regular para definir un identificador en C++ podría ser:

[a-zA-Z_][a-zA-Z0-9_]*

Esta expresión regular indica que un identificador debe empezar por una letra o un guión bajo (_), seguido por cero o más letras, números o guiones bajos.

Para crear un analizador léxico a partir de un código en base a Python que sea capaz de interpretar código de C++, se puede emplear el módulo re, que proporciona funciones para trabajar con expresiones regulares. Este módulo permite compilar una expresión regular en un objeto Pattern, que luego se puede usar para buscar coincidencias en una cadena de texto con el método match o search.

DESARROLLO

Se ha creado un código toma un archivo de código fuente en C++ y utiliza expresiones regulares para identificar diferentes tipos de tokens, como identificadores, números, operadores y paréntesis. Luego, construye una tabla de símbolos que almacena información sobre cada token, incluyendo el token en sí y su tipo.

Expresiones Regulares

En la parte superior del código, se definen una serie de expresiones regulares en la lista **token_patterns.** Cada tupla en esta lista contiene una expresión regular y un nombre de token (token_type) asociado. Estas expresiones regulares se utilizan para reconocer patrones específicos en el código fuente y asignarles un tipo de token.

```python
# Expresiones
token_patterns = [
    (r'\/\*[\s\S]*?\*\/', 'COMMENT_MULTILINE'),  # Token para comentarios multilínea "/* ... */"
    (r'\/\/[^\n]*', 'COMMENT_LINE'),  # Token para comentarios de línea "// ..."
    (r'if', 'IF'),  # Token para la palabra clave "if"
    (r'else', 'ELSE'),  # Token para la palabra clave "else"
    (r'while', 'WHILE'),  # Token para la palabra clave "while"
    (r'return', 'RETURN'),  # Token para la palabra clave "return"
    (r'class', 'CLASS'),  # Token para la palabra clave "class"
    (r'int|float|char|void', 'TYPE'),  # Token para tipos de datos (int, float, char, void)
    (r'\d+\.\d+[eE][+-]?\d+', 'DOUBLE'),  # Token para números de doble precisión en notación científica
    (r'\d+\.\d+', 'FLOAT'),         # Token para números de punto flotante
    (r'\d+', 'NUM'),                # Token para números enteros
    (r'\+', 'PLUS'),  # Token para el operador de suma "+"
    (r'-', 'MINUS'),  # Token para el operador de resta "-"
    (r'\*', 'TIMES'),  # Token para el operador de multiplicación "*"
    (r'/', 'DIVIDE'),  # Token para el operador de división "/"
    (r'\(', 'LPAREN'),  # Token para el paréntesis izquierdo "("
    (r'\)', 'RPAREN'),  # Token para el paréntesis derecho ")"
    (r'\{', 'LBRACE'),  # Token para la llave izquierda "{"
    (r'\}', 'RBRACE'),  # Token para la llave derecha "}"
    (r';', 'SEMICOLON'),  # Token para el punto y coma ";"
    (r',', 'COMMA'),  # Token para la coma ","
    (r'"[^"]*"', 'STRING'), # Token para cadenas de caracteres
    (r'=', 'ASSIGN'),      # Token para  "="
    (r'\+=', 'ADD_ASSIGN'), # Token para  "+="
    (r'-=', 'SUB_ASSIGN'), # Token para  "-="
    (r'>', 'GREATER_ASSIGN'), # Token para  "-="
    (r'<', 'LESSER_ASSIGN'), # Token para  "-="
    (r'#include\s*<.*?>', 'INCLUDE'), # Token para include
    (r'[a-zA-Z_][a-zA-Z0-9_]*', 'ID'),  # Token para identificadores
```

Clase SymbolTable

Se ha creado una clase denominada **SymbolTable** con el propósito de almacenar información referente a los símbolos identificados en el código. Estos símbolos pueden corresponder a variables, funciones, tipos de datos, entre otros. La clase **SymbolTable** ofrece métodos para llevar a cabo operaciones como la inserción, búsqueda y eliminación de símbolos en la tabla.

```python
# Leer codigo
class SymbolTable:
    def __init__(self):
        self.table = {}

    def insert(self, name, token_type, value=None):
        if name not in self.table:
            self.table[name] = {'type': token_type, 'value': value}
        else:
            # Manejar colisiones si es necesario
            pass

    def lookup(self, name):
        return self.table.get(name, None)

    def delete(self, name):
        if name in self.table:
            del self.table[name]

# Crear una instancia de la tabla de símbolos
symbol_table = SymbolTable()
```

Función tokenize_and_add_to_symbol_table

"Esta función recibe como entrada una línea de código fuente y un número de línea. Utiliza las expresiones regulares definidas en **token_patterns** para dividir la línea de código en tokens y luego los agrega a la tabla de símbolos (**symbol_table**). Además, se encarga de manejar casos

especiales como números enteros, números de punto flotante, cadenas y nombres de identificadores."

```
print (f"\n-----------------------------------------------")
print(f"Linea #{line_number}")
print(f"Cddigo: {line}")
print(f"Tokens son {tokens}")

# Propiedades de los tokens
print(f"\nLinea #{line_number} Propiedades")
for token_type, token_value in tokens:
    print(f"\nTipo es: {token_type}")
    print(f"Identificador: {token_value}")
```

Lectura del Código Fuente

Dentro del bloque if name == "main": en la parte principal del programa, se realiza la apertura de un archivo de código fuente y se procesa línea por línea. Cada línea se envía a la función **tokenize_and_add_to_symbol_table** para su tokenización, mientras se incrementa el número de líneas.

```
if __name__ == "__main__":
    file_path = "analizador_lexicografico/codigoenc.txt"

    try:
        with open(file_path, 'r') as file:
            line_number = 1
            for line in file:
                source_code = line
                tokenize_and_add_to_symbol_table(source_code, line_number)
                line_number += 1
```

Impresión de Tokens

Una vez que se han procesado todas las líneas del código fuente, se muestra la información de los tokens encontrados en cada línea. Esta información incluye el tipo de token y su valor.

```
print (f"\n-------------------------------------------------")
```

Manejo de Excepciones

El código maneja la excepción **FileNotFoundError** en caso de que el archivo de código fuente no se encuentre.

```
except FileNotFoundError:
    print(f"El archivo '{file_path}' no se encontró.")
```

Tokens

Los tokens son las unidades léxicas básicas en un programa de C++. Pueden clasificarse en varios tipos, que incluyen palabras clave, identificadores, constantes, literales de cadena y carácter, literales definidos por el usuario, operadores y signos de puntuación.

Palabras clave: Son las palabras reservadas del lenguaje que tienen un significado especial y no se pueden usar como identificadores. Algunos ejemplos son int, void, class, if, return, etc.

Identificadores: Son los nombres que se usan para referirse a variables, funciones, clases, objetos, etc. Deben empezar por una letra o un guión bajo y pueden contener letras, dígitos y guiones bajos. Algunos ejemplos son x, sum, cout, string, etc.

Constantes: Son valores fijos que no cambian durante la ejecución del programa. Pueden ser de tipo numérico, booleano o de puntero. Algunos ejemplos son 3.14, true, NULL, etc.

Literales de cadena y carácter: Son secuencias de caracteres que representan texto o un solo carácter. Se escriben entre comillas dobles o simples, respectivamente. Algunos ejemplos son "Hola", 'a', "C++", etc.

Literales definidos por el usuario: Son valores creados por el programador usando una sintaxis especial que permite especificar el tipo y el valor del literal. Se utilizan para crear objetos de clases definidas por el usuario. Algunos ejemplos son 12_km, 3.14_rad, "Hello"_s, etc.

Operadores: Son símbolos que se utilizan para realizar operaciones sobre los operandos, como operaciones aritméticas, lógicas, relacionales, etc. Suelen utilizar notación infija y se pueden aplicar a uno o varios operandos. Algunos ejemplos son +, -, *, /, %, &&, ||, <, >, =, etc.

Signos de puntuación: Son símbolos que se utilizan para separar o agrupar los tokens, como corchetes, llaves, paréntesis, comas, puntos y coma, etc. Algunos ejemplos son [ ], { }, ( ), , ,; , etc.

Los tokens suelen estar separados por espacios en blanco, que pueden ser uno o varios espacios, tabulaciones, saltos de línea, etc. También se pueden utilizar comentarios para explicar el código. Los comentarios pueden comenzar con // y terminar con un salto de línea, o pueden iniciar con /* y terminar con */

Esta versión reorganiza los términos en una estructura más clara y utiliza un lenguaje más conciso y preciso. Espero que te sea útil. Si tienes alguna otra pregunta, no dudes en hacerla.

Lista de palabras claves seleccionadas

float: los tipos numéricos de punto flotante representan números reales. Todos los tipos numéricos de punto flotante son tipos de valor	**comment multiline:** se refiere a las propiedades en la en la se podrá comentar en más de una línea
comment line: se emplea para comentar una sola línea.	**while:** estructura de control que permite ejecutar un bloque de código de forma repetida mientras se cumpla una condición.
if: es una estructura de control que permite ejecutar un bloque de código de forma condicional, es decir, solo si se cumple una expresión lógica.	**else:** es una estructura de control que se usa junto con la instrucción if para ejecutar un bloque de código alternativo cuando la condición del if es falsa.
type: se utiliza para indicar funciones del tipo char, int, float, y void	**string:** se utiliza para la indicación de una cadena de texto
class: Se define un nuevo tipo de dato que se puede usar para crear objetos. Una clase contiene los datos y el código que operan sobre esos datos, llamados variables miembro y funciones miembro, respectivamente.	**return:** Sirve para terminar la ejecución de una función y devolver el control a la función que la llamó, o al sistema operativo si se trata de la función main. La instrucción return también puede devolver un valor al punto de llamada, que debe ser del mismo tipo que el especificado en la declaración de la función.

DFA’s construidos

Para diseñar un autómata finito determinista (AFD) que reconozca estos tokens, primero debemos entender las expresiones regulares utilizadas en las tuplas (pattern, token_type) que has definido. Cada tupla describe un patrón y su correspondiente tipo de token. De esta manera, el analizador léxico se encarga de dividir el código fuente en tokens, que son unidades léxicas como palabras clave, identificadores, números, operadores, etc. Esto se logra mediante el uso de expresiones regulares y el diseño de un AFD que reconozca los patrones definidos en las tuplas.

El proceso de diseño de un analizador léxico implica la creación de autómatas finitos deterministas (AFD) que reconocen los patrones definidos en las expresiones regulares. Aquí hay una descripción general de cómo se podrían diseñar AFD para algunos de los tokens mencionados:

Palabras Clave: Un AFD para palabras clave podría tener estados que representan cada letra de las palabras clave y transiciones que siguen el orden de las letras.

Identificadores: El AFD para identificadores podría comenzar con un estado que acepta cualquier letra o guión bajo como primer carácter, seguido de transiciones para aceptar letras, dígitos o guiones bajos en cualquier orden.

Números Enteros: El AFD para números enteros podría tener un estado inicial que acepta cualquier dígito y luego transiciones para aceptar más dígitos.

Números de Punto Flotante: El AFD para números de punto flotante podría comenzar como el AFD para números enteros, pero con transiciones adicionales para aceptar el punto decimal y dígitos después del punto.

Cadenas: El AFD para cadenas podría comenzar en un estado que acepta comillas dobles y luego transiciones para aceptar cualquier carácter que no sea una comilla doble hasta que se alcance otra comilla doble.

REFERENCIAS

1. Tokens de c++ para sistema de expresiones regulares
   https://learn.microsoft.com/es-es/cpp/cpp/character-sets?view=msvc-170

2. ejemplos de analizador lexicográfico en python
   https://docs.python.org/es/3/reference/lexical_analysis.html