

Centro Universitario de Ciencias Exactas e Ingeniería

SEM. de Solución de Problemas de Traductores 2

Practica 1

LUIS FELIPE MUNOZ MENDOZA

Juan Pablo Hernández Orozco

219294285

Objetivo:

Crea un analizador léxico que valide varias cadenas dadas por el usuario.

el programa debe identificar los siguientes Token y asignar su correspondiente categoría

0 --- Tipo de dato	1 --- Identificador	2 --- constante	3 --- ;	4 --- ,
5 --- (6 ---)	7 --- {	8 --- }	9 --- =
9 --- if	10 --- while	11 --- return	12 --- else	13 --- for
14 --- opAdición	15 --- opMultiplicacion	16 --- opLogico	17 --- opRelacional	18 -- \$

Siendo:

opAdision + -

opMultiplicacion * / << >>

opLogico && ||

opelacional < > >= <= == !=

constante cualquier numero #, y pi

tipo de dato int float char void string

recuerda mostrar al final la cantidad de tokens encontrados de cada categoría o si encontró algún error.

Puede ser implementado en cualquier lenguaje, solo que cumpla los requerimientos mencionados.

Les comparto los códigos vistos en clase por si quieren tomarlos de base o apoyo.

Se adjuntan códigos fuente, el reporte del desarrollo de la actividad y un video corto demostrando el funcionamiento.

Introducción:

El análisis léxico es una etapa fundamental en la compilación de programas, donde una secuencia de caracteres es transformada en una lista de tokens que representan las unidades léxicas del lenguaje. En este proyecto, se implementa un analizador léxico en Python que procesa código fuente almacenado en un archivo de texto (entrada.txt).

El objetivo de este analizador es identificar palabras clave, operadores, identificadores, constantes y otros elementos sintácticos de un lenguaje de programación similar a C. Mediante el uso de expresiones condicionales y estructuras de control, el programa clasifica cada fragmento de la entrada en su categoría correspondiente y genera un informe con los tokens detectados.

Esta implementación facilita la detección temprana de errores léxicos, permitiendo el desarrollo de herramientas más avanzadas para el procesamiento de código, como analizadores sintácticos e intérpretes.

Desarrollo:

¿Qué es un analizador léxico y cómo funciona?

Un analizador léxico es el primer paso en el proceso de compilación de un programa. Su función principal es leer el código fuente, identificar los componentes básicos del lenguaje y clasificarlos en tokens, que son las unidades mínimas significativas del código.

Por ejemplo, en la línea: `int x = 10 ;`

El analizador léxico dividiría la entrada en los siguientes tokens:

- `int` → tipo de dato
- `x` → identificador
- `=` → operador de asignación
- `10` → constante
- `;` → punto y coma (delimitador)

El analizador sigue reglas definidas para reconocer palabras clave, operadores, identificadores, números y otros símbolos. Si encuentra una secuencia no válida (como `#` en un lenguaje donde no es un operador), la marca como error.

El flujo general del analizador léxico es:

1. Leer el código fuente carácter por carácter.
2. Ignorar espacios en blanco y saltos de línea.
3. Identificar tokens de múltiples caracteres (como `==`, `>=`).
4. Identificar tokens de un solo carácter (como `;`, `+`).
5. Clasificar los tokens y almacenarlos en una lista.
6. Detectar errores si se encuentran caracteres no reconocidos.

Esta práctica procesa un archivo de texto con código fuente, genera una lista de tokens y muestra un resumen de la cantidad de cada tipo de token encontrado.

Código

Definición de la función `lexer(cadena)`

```
def lexer(cadena):
```

Esta función recibe una cadena de texto (el código fuente) y devuelve una lista de tokens.

```
    tokens = []
```

```
    i = 0
```

```
    n = len(cadena)
```

`tokens`: lista donde se almacenarán los tokens encontrados.

i: índice que recorre la cadena carácter por carácter.

n: longitud de la cadena para evitar que i salga del rango.

Proceso de análisis léxico

```
while i < n:
```

Un bucle que recorre la cadena mientras haya caracteres por analizar.

```
    if cadena[i].isspace():
```

```
        i += 1
```

Si el carácter actual es un espacio o salto de línea, se ignora y se avanza al siguiente.

Detección de tokens de dos caracteres

```
    dos_chars = cadena[i:i+2]
```

```
    if dos_chars in ['<<', '>>']:
```

```
        tokens.append(('opMultiplicacion', dos_chars))
```

```
        i += 2
```

Si el carácter y el siguiente forman un operador como << o >>, se agrega como token y se avanza dos posiciones.

```
    if dos_chars in ['&&', '||']:
```

```
        tokens.append(('opLogico', dos_chars))
```

```
        i += 2
```

Reconoce operadores lógicos (&&, ||).

```
    if dos_chars in ['>=', '<=', '==', '!=']:
```

```
        tokens.append(('opRelacional', dos_chars))
```

```
        i +=
```

Reconoce operadores relacionales (>=, <=, ==, !=).

Detección de tokens de un solo carácter

```
    ch = cadena[i]
```

```
    if ch == ';':
```

```
        tokens.append(('punto_y_coma', ';'))
```

Si el carácter es ;, se agrega como token de punto y coma. Este mismo método se usa para otros símbolos como ,, {, }, (,), =, +, -, *, /, <, >, y \$.

Detección de constantes y palabras clave

```
if cadena[i:i+2] == 'pi' and (i+2 == n or not cadena[i+2].isalnum()):  
    tokens.append(('constante', 'pi'))  
    i += 2
```

Si encuentra pi, lo reconoce como una constante especial.

```
if cadena[i].isdigit():  
    num = ""  
    dot_count = 0  
    while i < n and (cadena[i].isdigit() or cadena[i] == '.'):   
        if cadena[i] == '.':  
            dot_count += 1  
            if dot_count > 1:  
                break  
        num += cadena[i]  
        i += 1  
    tokens.append(('constante', num))
```

Este bloque detecta números enteros y flotantes, asegurándose de que solo haya un punto decimal en los números flotantes.

```
if cadena[i].isalpha():  
    palabra = ""  
    while i < n and (cadena[i].isalnum() or cadena[i] == '_'):  
        palabra += cadena[i]  
        i += 1
```

Si el carácter es una letra, se trata como una posible palabra clave o identificador. Se sigue leyendo hasta que aparezca un carácter que no sea alfanumérico o _.

```
if palabra in ['if', 'while', 'return', 'else', 'for']:  
    tokens.append((palabra, palabra))  
elif palabra in ['int', 'float', 'char', 'void', 'string']:  
    tokens.append(('tipo_dato', palabra))  
else:
```

```
tokens.append(('identificador', palabra))
```

Si la palabra es una palabra clave (if, while, return, etc.), se clasifica como tal. Si es un tipo de dato (int, float, etc.), se clasifica como tipo de dato. Si no es ninguna de estas, se trata como un identificador.

Manejo de errores

```
tokens.append(('error', ch))
```

```
i += 1
```

Si un carácter no pertenece a ninguna categoría, se marca como error.

Función main() para leer el archivo y analizar el código

```
def main():
```

```
    with open("entrada.txt", "r") as archivo:
```

```
        cadena = archivo.read()
```

Abre y lee el contenido del archivo entrada.txt.

```
tokens = lexer(cadena)
```

Llama a la función lexer() para analizar la cadena y obtener la lista de tokens.

```
print("\nTokens encontrados:")
```

```
for ttipo, lexema in tokens:
```

```
    print(f"Tipo: {ttipo}, Lexema: {lexema}")
```

Imprime los tokens encontrados.

```
conteo = {}
```

```
for ttipo, _ in tokens:
```

```
    conteo[ttipo] = conteo.get(ttipo, 0) + 1
```

Cuenta cuántos tokens de cada tipo hay.

```
print("\nConteo de tokens por categoría:")
```

```
for categoria, cantidad in conteo.items():
```

```
    print(f"{categoria}: {cantidad}")
```

Muestra el conteo de cada categoría de token.

```
if 'error' in conteo:
```

```
    print("\nSe encontraron errores en el análisis léxico.")
```

Este es el ejemplo del archivo de entrada

Y obtenemos de salida :

Tokens encontrados:

Tipo: tipo_datos, Lexema: int

Tipo: identificador, Lexema: main

Tipo: simbolo, Lexema: (

Tipo: simbolo, Lexema:)

Tipo: simbolo, Lexema: {

Tipo: tipo_datos, Lexema: float

Tipo: constante, Lexema: pi

Tipo: simbolo, Lexema: =

Tipo: constante, Lexema: 3.14

Tipo: simbolo, Lexema: ;

Tipo: tipo_datos, Lexema: char

Tipo: identificador, Lexema: letra

Tipo: simbolo, Lexema: =

Tipo: error, Lexema: '

Tipo: identificador, Lexema: A

Tipo: error, Lexema: '

Tipo: simbolo, Lexema: ;

Tipo: tipo_datos, Lexema: string

Tipo: identificador, Lexema: mensaje

Tipo: simbolo, Lexema: =

Tipo: error, Lexema: "

Tipo: identificador, Lexema: Hola

Tipo: simbolo, Lexema: ,

Tipo: identificador, Lexema: mundo

Tipo: error, Lexema: "

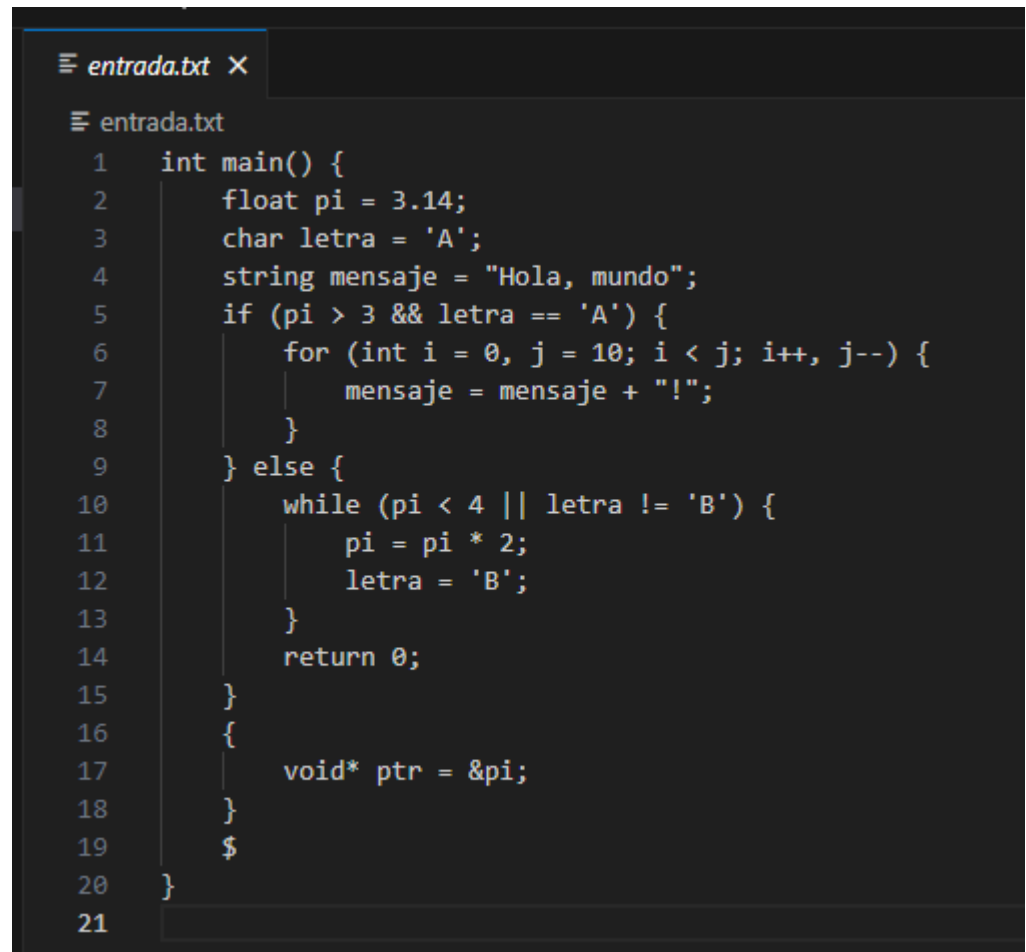
Tipo: simbolo, Lexema: ;

Tipo: if, Lexema: if

Tipo: simbolo, Lexema: (

Tipo: constante, Lexema: pi

Tipo: simbolo, Lexema: >



```

1  int main() {
2      float pi = 3.14;
3      char letra = 'A';
4      string mensaje = "Hola, mundo";
5      if (pi > 3 && letra == 'A') {
6          for (int i = 0, j = 10; i < j; i++, j--) {
7              mensaje = mensaje + "!";
8          }
9      } else {
10         while (pi < 4 || letra != 'B') {
11             pi = pi * 2;
12             letra = 'B';
13         }
14         return 0;
15     }
16     {
17         void* ptr = &pi;
18     }
19     $
20 }
21

```

Tipo: constante, Lexema: 3
Tipo: opLogico, Lexema: &&
Tipo: identificador, Lexema: letra
Tipo: opRelacional, Lexema: ==
Tipo: error, Lexema: '
Tipo: identificador, Lexema: A
Tipo: error, Lexema: '
Tipo: simbolo, Lexema:)
Tipo: simbolo, Lexema: {
Tipo: for, Lexema: for
Tipo: simbolo, Lexema: (
Tipo: tipo_dato, Lexema: int
Tipo: identificador, Lexema: i
Tipo: simbolo, Lexema: =
Tipo: constante, Lexema: 0
Tipo: simbolo, Lexema: ,
Tipo: identificador, Lexema: j
Tipo: simbolo, Lexema: =
Tipo: constante, Lexema: 10
Tipo: simbolo, Lexema: ;
Tipo: identificador, Lexema: i
Tipo: simbolo, Lexema: <
Tipo: identificador, Lexema: j
Tipo: simbolo, Lexema: ;
Tipo: identificador, Lexema: i
Tipo: simbolo, Lexema: +
Tipo: simbolo, Lexema: +
Tipo: simbolo, Lexema: ,
Tipo: identificador, Lexema: j
Tipo: simbolo, Lexema: -
Tipo: simbolo, Lexema: -
Tipo: simbolo, Lexema:)
Tipo: simbolo, Lexema: {

Tipo: identificador, Lexema: mensaje

Tipo: simbolo, Lexema: =

Tipo: identificador, Lexema: mensaje

Tipo: simbolo, Lexema: +

Tipo: error, Lexema: "

Tipo: error, Lexema: !

Tipo: error, Lexema: "

Tipo: simbolo, Lexema: ;

Tipo: simbolo, Lexema: }

Tipo: simbolo, Lexema: }

Tipo: else, Lexema: else

Tipo: simbolo, Lexema: {

Tipo: while, Lexema: while

Tipo: simbolo, Lexema: (

Tipo: constante, Lexema: pi

Tipo: simbolo, Lexema: <

Tipo: constante, Lexema: 4

Tipo: opLogico, Lexema: ||

Tipo: identificador, Lexema: letra

Tipo: opRelacional, Lexema: !=

Tipo: error, Lexema: '

Tipo: identificador, Lexema: B

Tipo: error, Lexema: '

Tipo: simbolo, Lexema:)

Tipo: simbolo, Lexema: {

Tipo: constante, Lexema: pi

Tipo: simbolo, Lexema: =

Tipo: constante, Lexema: pi

Tipo: simbolo, Lexema: *

Tipo: constante, Lexema: 2

Tipo: simbolo, Lexema: ;

Tipo: identificador, Lexema: letra

Tipo: simbolo, Lexema: =

Tipo: error, Lexema: '
Tipo: identificador, Lexema: B
Tipo: error, Lexema: '
Tipo: simbolo, Lexema: ;
Tipo: simbolo, Lexema: }
Tipo: return, Lexema: return
Tipo: constante, Lexema: 0
Tipo: simbolo, Lexema: ;
Tipo: simbolo, Lexema: }
Tipo: simbolo, Lexema: {
Tipo: tipo_dato, Lexema: void
Tipo: simbolo, Lexema: *
Tipo: identificador, Lexema: ptr
Tipo: simbolo, Lexema: =
Tipo: error, Lexema: &
Tipo: constante, Lexema: pi
Tipo: simbolo, Lexema: ;
Tipo: simbolo, Lexema: }
Tipo: simbolo, Lexema: \$
Tipo: simbolo, Lexema: }

Conteo de tokens por categoría:

tipo_dato: 6
identificador: 21
simbolo: 53
constante: 13
error: 14
if: 1
opLogico: 2
opRelacional: 2
for: 1
else: 1
while: 1

```
return: 1
```

Conclusión:

El analizador léxico en Python desarrollado en esta práctica constituye un elemento esencial en el proceso de compilación. Al leer el código fuente desde un archivo, identificar y clasificar cada token—ya sean palabras clave, identificadores, constantes u operadores—se establece una base sólida para la construcción de compiladores e intérpretes más avanzados. La capacidad de detectar errores léxicos de manera temprana no solo mejora la calidad del código, sino que también simplifica el posterior análisis sintáctico y semántico. En definitiva, esta práctica refuerza la comprensión de los fundamentos del procesamiento de lenguajes y demuestra la importancia de una correcta estructuración y manejo de datos en el desarrollo de herramientas de programación.

Repositorio

https://github.com/ELJuanP/Seminario_de_Traductores_de_lenguajes_II/tree/main/Practica%201