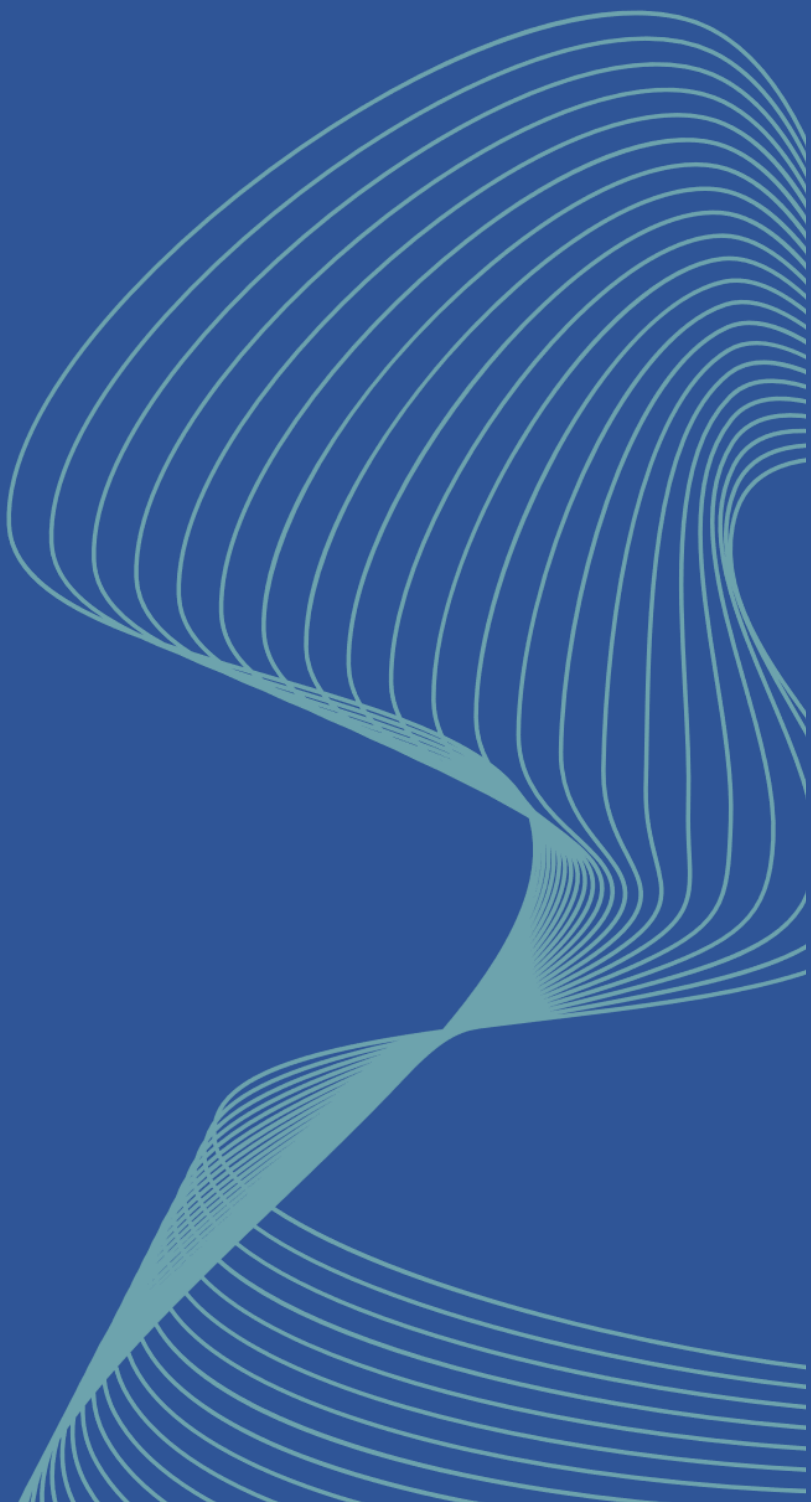
PROYECTO 1 LFP

S12024

Manual Técnico



Alumno
David Isaac García Mejía
202202077
Inga. Vivian Damaris Campos González

Manual Técnico

Esta aplicación está diseñada para analizar texto de entrada en formato json, identificar sus componentes léxicos, realizar análisis sintáctico y ser capaz de analizar y comprobar si las cadenas son aceptadas por una expresión regular. Como resultado se obtiene los reportes de los tokens, los errores y la tabla de la comprobación de las cadenas.

1. Requisitos del Sistema:

- Python 3.x

- Tkinter (incluido en la instalación estándar de Python)

- Graphviz (para la generación de árboles de derivación)

2. Estructura del proyecto:

El código fuente está organizado en varios archivos que representan diferentes aspectos de la aplicación:

ventana.py maneja la interfaz de usuario.

Parser.py realiza el análisis sintáctico.

Token.py *y* ***Error.py*** proporcionan clases auxiliares para el análisis léxico y la gestión de errores.

Analizador.py, Analizador_expresiones.py y ConvertidorHTML.py contienen funciones específicas de análisis.

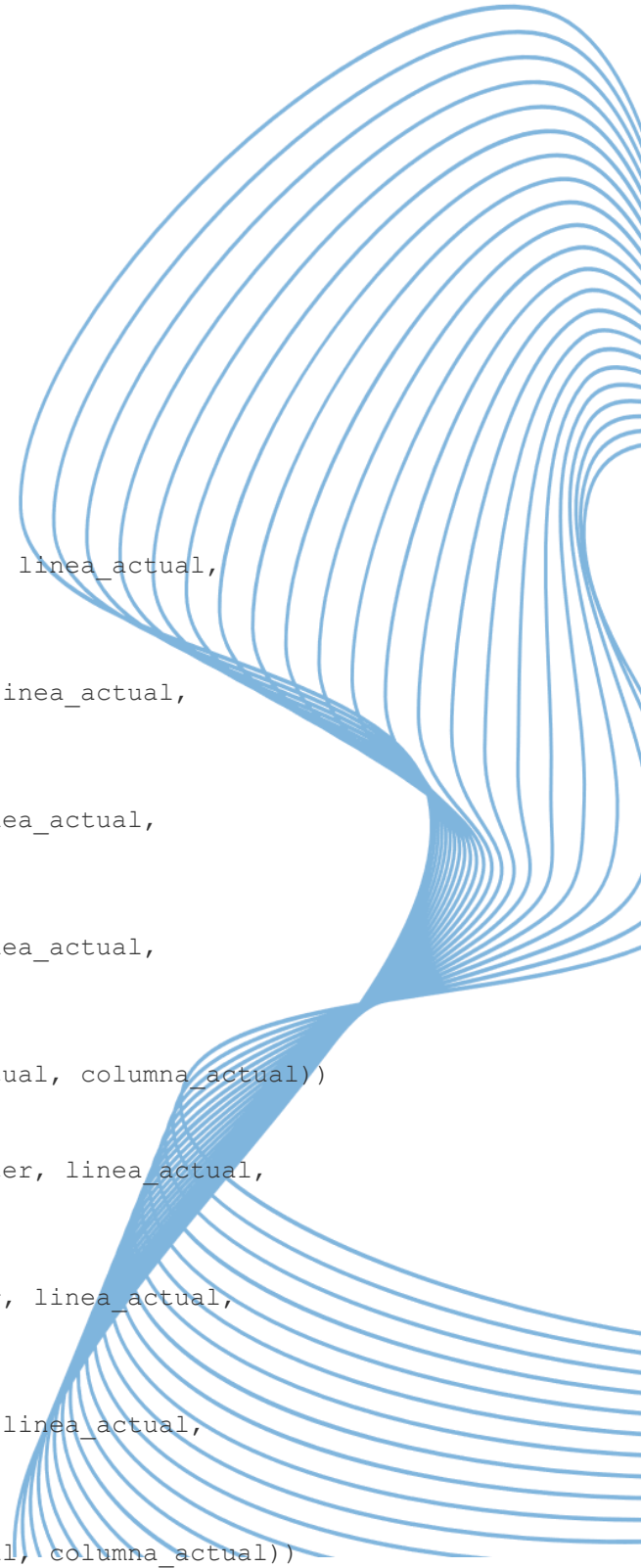
3. Funciones principales:

3.1 Análisis léxico:

El análisis léxico se realiza en el archivo de Analizador.py, en este se encuentra una función llamada *analizador_lexico(),* que recibe como parámetro una *entrada* que es la cadena de texto que se obtiene del archivo ingresado.

```
def analizador_lexico(self, entrada):
    ESTADO_INICIAL = 0
    ESTADO_TOKEN = 1
    ESTADO_NUMERO_ENTERO = 2
    ESTADO_NUMERO_DECIMAL = 3
    ESTADO_CADENA = 4
    ESTADO_COMENTARIO_LINEA = 5
    ESTADO_COMENTARIO_MULTILINEA = 6

    tokens_lexemas = []
    caracteres_no_permitidos = []
```

```
    lexema_actual = ''
    estado_actual = ESTADO_INICIAL
    posicion_actual = 0
    linea_actual = 1
    columna_actual = 1

    while posicion_actual < len(entrada):
      caracter = entrada[posicion_actual]

      if caracter == '\n':
        linea_actual += 1
        columna_actual = 1
      else:
        columna_actual += 1

      if estado_actual == ESTADO_INICIAL:
        if caracter.isspace():
          # Se ignora espacios en blanco
          pass
        elif caracter == '{':
          tokens_lexemas.append(
              Token('LLAVE_APERTURA', caracter, linea_actual,
columna_actual))
        elif caracter == '}':
          tokens_lexemas.append(
              Token('LLAVE_CIERRE', caracter, linea_actual,
columna_actual))
        elif caracter == ':':
          tokens_lexemas.append(
              Token('DOS_PUNTOS', caracter, linea_actual,
columna_actual))
        elif caracter == ';':
          tokens_lexemas.append(
              Token('PUNTO_COMA', caracter, linea_actual,
columna_actual))
        elif caracter == ',':
          tokens_lexemas.append(
              Token('COMA', caracter, linea_actual, columna_actual))
        elif caracter == '[':
          tokens_lexemas.append(
              Token('CORCHETE_APERTURA', caracter, linea_actual,
                    columna_actual))
        elif caracter == ']':
          tokens_lexemas.append(
              Token('CORCHETE_CIERRE', caracter, linea_actual,
columna_actual))
        elif caracter == '?':
          tokens_lexemas.append(
              Token('INTERROGACION', caracter, linea_actual,
columna_actual))
        elif caracter == '|':
          tokens_lexemas.append(
              Token('OR', caracter, linea_actual, columna_actual))
```



```
        elif caracter == '*':
          tokens_lexemas.append(
              Token('ASTERISCO', caracter, linea_actual, columna_actual))
        elif caracter == '(':
          tokens_lexemas.append(
              Token('PARENTESIS_ABRE', caracter, linea_actual,
columna_actual))
        elif caracter == ')':
          tokens_lexemas.append(
              Token('PARENTESIS_CIERRA', caracter, linea_actual,
                    columna_actual))
        elif caracter == '+':
          tokens_lexemas.append(
              Token('MAS', caracter, linea_actual, columna_actual))
        elif caracter == '=':
          tokens_lexemas.append(
              Token('SIGNO_IGUAL', caracter, linea_actual,
columna_actual))
        elif caracter == '"':
          lexema_actual += caracter
          estado_actual = ESTADO_CADENA
        elif caracter.isalpha():
          lexema_actual += caracter
          estado_actual = ESTADO_TOKEN
        elif caracter.isdigit():
          lexema_actual += caracter
          estado_actual = ESTADO_NUMERO_ENTERO
        elif caracter == '-':
          lexema_actual += caracter
          estado_actual = ESTADO_NUMERO_ENTERO
        elif caracter == '#':
          estado_actual = ESTADO_COMENTARIO_LINEA
        elif caracter == "'":
          estado_actual = ESTADO_COMENTARIO_MULTILINEA
        else:
          caracteres_no_permitidos.append(
              (caracter, linea_actual, columna_actual))

      elif estado_actual == ESTADO_TOKEN:
        if caracter.isalpha() or caracter.isdigit() or caracter == '_':
          lexema_actual += caracter
        else:
          if lexema_actual in self.palabras_reservadas:
            tokens_lexemas.append(
                Token('PALABRA_RESERVADA_' + lexema_actual,
lexema_actual,
                      linea_actual, columna_actual))
          else:
            caracteres_no_permitidos.append(
                (lexema_actual, linea_actual, columna_actual))
          lexema_actual = ''
          estado_actual = ESTADO_INICIAL
          continue
```

```
    elif estado_actual == ESTADO_NUMERO_ENTERO:
      if caracter.isdigit():
        lexema_actual += caracter
      elif caracter == '.':
        lexema_actual += caracter
        estado_actual = ESTADO_NUMERO_DECIMAL
      else:
        tokens_lexemas.append(
            Token('NUMERO_ENTERO', lexema_actual, linea_actual,
                  columna_actual))
        lexema_actual = ''
        estado_actual = ESTADO_INICIAL
        continue

    elif estado_actual == ESTADO_NUMERO_DECIMAL:
      if caracter.isdigit():
        lexema_actual += caracter
      else:
        tokens_lexemas.append(
            Token('NUMERO_DECIMAL', lexema_actual, linea_actual,
                  columna_actual))
        lexema_actual = ''
        estado_actual = ESTADO_INICIAL
        continue

    elif estado_actual == ESTADO_CADENA:
      if caracter == '"':
        lexema_actual += caracter
        tokens_lexemas.append(
            Token('CADENA', lexema_actual, linea_actual,
columna_actual))
        lexema_actual = ''
        estado_actual = ESTADO_INICIAL
      else:
        lexema_actual += caracter

    elif estado_actual == ESTADO_COMENTARIO_LINEA:
      if caracter == '\n':
        tokens_lexemas.append(
            Token('COMENTARIO_LINEA', lexema_actual, linea_actual,
                  columna_actual))
        lexema_actual = ''
        estado_actual = ESTADO_INICIAL
      else:
        lexema_actual += caracter

    elif estado_actual == ESTADO_COMENTARIO_MULTILINEA:
      if caracter == "'":
        lexema_actual += caracter
        if lexema_actual.endswith("'''"):
          tokens_lexemas.append(
```

```
                Token('COMENTARIO_MULTILINEA', lexema_actual,
linea_actual,
                      columna_actual))
            lexema_actual = ''
            estado_actual = ESTADO_INICIAL
        else:
          lexema_actual += caracter

      posicion_actual += 1

    return tokens_lexemas, caracteres_no_permitidos
```

1. Definición de estados y variables iniciales: Se definen varios estados para controlar el proceso de análisis léxico. Además, se inicializan las listas tokens_lexemas y caracteres_no_permitidos, y se inicializan las variables lexema_actual, estado_actual, posicion_actual, linea_actual y columna_actual.
2. Bucle principal: El análisis léxico se realiza mediante un bucle while que recorre la cadena de entrada caracter por caracter.
3. Actualización de la posición y la línea/columna actuales: Se actualizan las variables linea_actual y columna_actual cada vez que se encuentra un salto de línea o un caracter válido.
4. Manejo de caracteres en el estado inicial (ESTADO_INICIAL): En este estado, se analizan los caracteres para determinar si forman parte de algún token reconocido en el lenguaje. Se identifican caracteres como llaves, dos puntos, punto y coma, comas, corchetes, símbolos de operadores, comillas, letras, números, etc.
5. Manejo de tokens y palabras reservadas: Se maneja la lógica para identificar palabras reservadas del lenguaje, como palabras clave. Cuando se identifica una secuencia de caracteres que forma una palabra reservada, se agrega un token correspondiente a la lista tokens_lexemas.
6. Manejo de números enteros (ESTADO_NUMERO_ENTERO): Si se encuentra una secuencia de dígitos que forma un número entero, se agrega un token correspondiente a la lista tokens_lexemas.
7. Manejo de números decimales (ESTADO_NUMERO_DECIMAL): Si se encuentra una secuencia de dígitos con un punto decimal, se agrega un token correspondiente a la lista tokens_lexemas.
8. Manejo de cadenas de texto (ESTADO_CADENA): Se maneja la lógica para identificar cadenas de texto delimitadas por comillas. Cuando se encuentra el cierre de la cadena, se agrega un token correspondiente a la lista tokens_lexemas.
9. Manejo de comentarios de línea (ESTADO_COMENTARIO_LINEA): Se maneja la lógica para identificar comentarios de línea iniciados con el caracter #. Cuando se encuentra un salto de línea, se agrega un token correspondiente a la lista tokens_lexemas.
10. Manejo de comentarios multilinea (ESTADO_COMENTARIO_MULTILINEA): Se maneja la lógica para identificar comentarios multilinea iniciados y terminados con el caracter '. Cuando se encuentra el cierre del comentario multilinea, se agrega un token correspondiente a la lista tokens_lexemas.

11. Actualización de la posición actual: Se incrementa la variable posicion_actual para avanzar al siguiente caracter en la cadena de entrada.
12. Retorno de resultados: Se retornan las listas tokens_lexemas y caracteres_no_permitidos que contienen los tokens reconocidos y los caracteres no permitidos encontrados durante el análisis léxico, respectivamente.

3.2 Análisis Sintáctico:

1. Inicio del análisis sintáctico (parse):

- La función parse se llama para iniciar el análisis sintáctico del código fuente.
- Esta función es el punto de entrada principal para el análisis sintáctico en la clase Parser.

2. Inicio de la derivación (inicio):

- La función inicio se encarga de iniciar la derivación de la gramática.
- Llama a los métodos elemento y otro_elemento que representan los símbolos iniciales de la gramática.

3. Análisis del primer elemento (elemento):

- Se verifica si el primer token es una llave de apertura ({). Si es así, se procesa la instrucción contenida dentro de las llaves.
- Si no se encuentra una llave de apertura, se agrega un error y se intenta recuperar sincronizando con una llave de cierre.
- Dentro de las llaves, se procesan las instrucciones relacionadas con ID, ER (Expresión Regular) y cadenas de texto.

4. Análisis de otros elementos (otro_elemento):

- Esta función maneja la producción de la gramática que permite tener múltiples elementos separados por comas.
- Si se encuentra una coma después de un elemento, se analiza el siguiente elemento llamando a la función elemento.
- Este proceso se repite de manera recursiva hasta que no hay más elementos que analizar.

5. Análisis de instrucción de ID (instruccionID):

- Verifica si la instrucción comienza con la palabra reservada "ID", seguida de dos puntos, un número entero y un punto y coma.
- Si la sintaxis no coincide con esta estructura, se agrega un error y se intenta recuperar sincronizando con un punto y coma.
- Esta función se encarga de validar la sintaxis de las instrucciones relacionadas con ID.

6. Análisis de instrucción de ER (instruccionER):

- Verifica si la instrucción comienza con la palabra reservada "ER", seguida de dos puntos, una expresión regular y un punto y coma.
- Si la sintaxis no coincide con esta estructura, se agrega un error y se intenta recuperar sincronizando con un punto y coma.

- Esta función se encarga de validar la sintaxis de las instrucciones relacionadas con ER (Expresiones Regulares).

7. Análisis de expresiones (expresion):

- Analiza las expresiones regulares definidas dentro de las instrucciones de ER.
- Puede contener elementos individuales o expresiones compuestas por paréntesis.
- Se maneja la precedencia y asociatividad de los operadores.

8. Análisis de operadores (operador):

- Se encarga de analizar los operadores relacionados con las expresiones regulares.
- Puede ser un operador unario como "+" o "*", o el operador binario "OR".

9. Análisis de instrucción de cadenas (instruccionCadenas):

- Verifica si la instrucción comienza con la palabra reservada "CADENAS", seguida de dos puntos, una cadena de texto y opcionalmente otras cadenas separadas por comas.
- Si la sintaxis no coincide con esta estructura, se agrega un error y se intenta recuperar sincronizando con un punto y coma.
- Esta función se encarga de validar la sintaxis de las instrucciones relacionadas con cadenas de texto.

10. Análisis de otras cadenas (otraCadena):

- Esta función maneja la producción de la gramática que permite tener múltiples cadenas de texto separadas por comas.
- Se repite de manera recursiva hasta que no hay más cadenas que analizar.

11. Recuperación de errores (recuperar):

- Si se encuentra un error sintáctico, se llama a la función recuperar para intentar sincronizar el análisis y continuar desde un punto válido en la gramática.
- Esto ayuda a evitar que un único error detenga por completo el análisis sintáctico, permitiendo identificar múltiples errores en el código fuente.

12. Registro de errores (agregar_error):

- Durante todo el proceso de análisis sintáctico, se registran los errores encontrados en una lista para su posterior manejo.
- Cada error incluye información detallada como el tipo de error, la línea y columna donde ocurrió, y un mensaje descriptivo.