

Manual Técnico

Guía técnica del analizador léxico y sintáctico (archivos .rmt).

Analizador Léxico.

Analizador Sintáctico.

Analizador Léxico e introducción al Sintáctico

El desarrollo de la aplicación consiste en poder hacer la limpieza de errores en todos los archivos que los contengan.

Los lenguaje definido y estructurado en Javascript, Css, Html y Rmt para la recopilación tendrán que ser analizado léxicamente y poder obtener los identificadores, comentarios, palabras reservadas, cadenas, operadores, números; en los archivos .rmt el analizador sintáctico verifica la estructura de una expresión aritmética. Los analizadores podrán verificar si existen errores léxicos o sintácticos (archivo .rmt), si esto ocurre se deben crear reportes y la limpieza de los mismos.

Requerimientos mínimos

- Windows 10 (64 bits)
- 15 MB de RAM
- 22 MB de espacio libre mínimo
- Archivo en formato .js, .css, .html, .rmt

Plataforma de desarrollo

- Microsoft Visual Studio Code
 - Versión: 1.48.2
- Python
 - Versión: 3.8.5
- Microsoft Windows 10

Analizador Léxico

1. Expresiones regulares Javascript

$C \rightarrow$ Todos los caracteres excepto “

$T \rightarrow$ Todos los caracteres

$S \rightarrow \{ \} () ; : . ,$

$L (L | D | _)^* \#1$

//Identificadores

$D^+ (. D^+)^? \#2$

//Numero entero o decimal

$S \#3$

//Símbolos

$" C^* " \#4$

//Cadena caracteres

$' C^* ' \#5$

//Cadena caracteres

$/ (* T^* * / | / T^* \backslash n)^? \#6$

//Comentarios – División

$\&\& \#7$

//Lógico

$\| \#8$

//Lógico

$+ (+ | =)^? \#9$

//Incremento – Concatenación

$- (- | =)^? \#10$

//Decremento

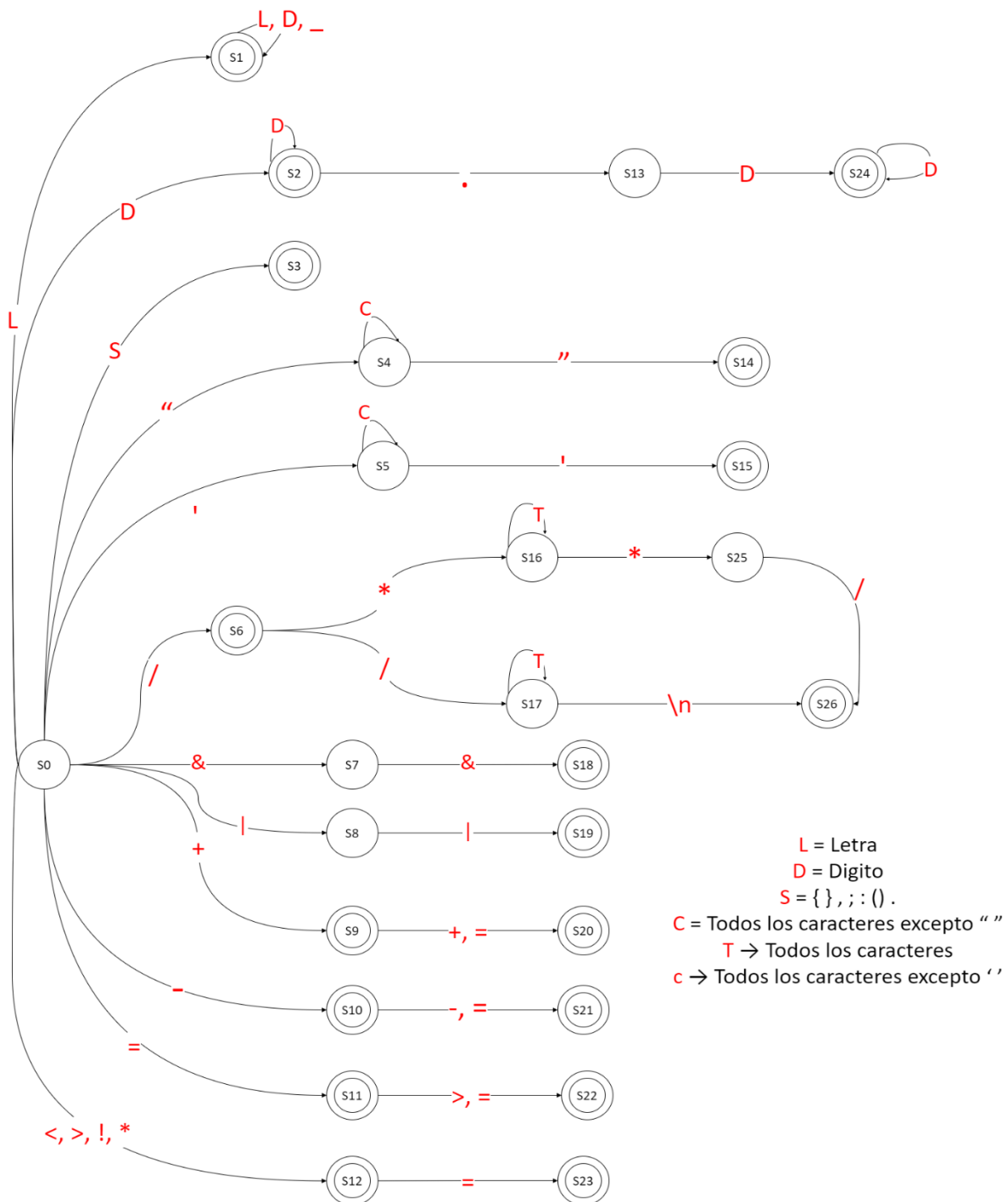
$= (> | =)^? \#11$

//Lamba o Igual

$(< | > | ! | *) =^? \#12$

//Relacionales – Asignación

2. Autómata finito determinista Javascript



3. Expresiones regulares Css

$C \rightarrow$ Todos los caracteres excepto “

$S \rightarrow \{ \} () ; : , . - * \% \#$

$L (L | D | -)^* \#1$

//Identificadores

$D^+ (. D^+)^? \#2$

//Numero entero o decimal

$S \#3$

//Símbolos

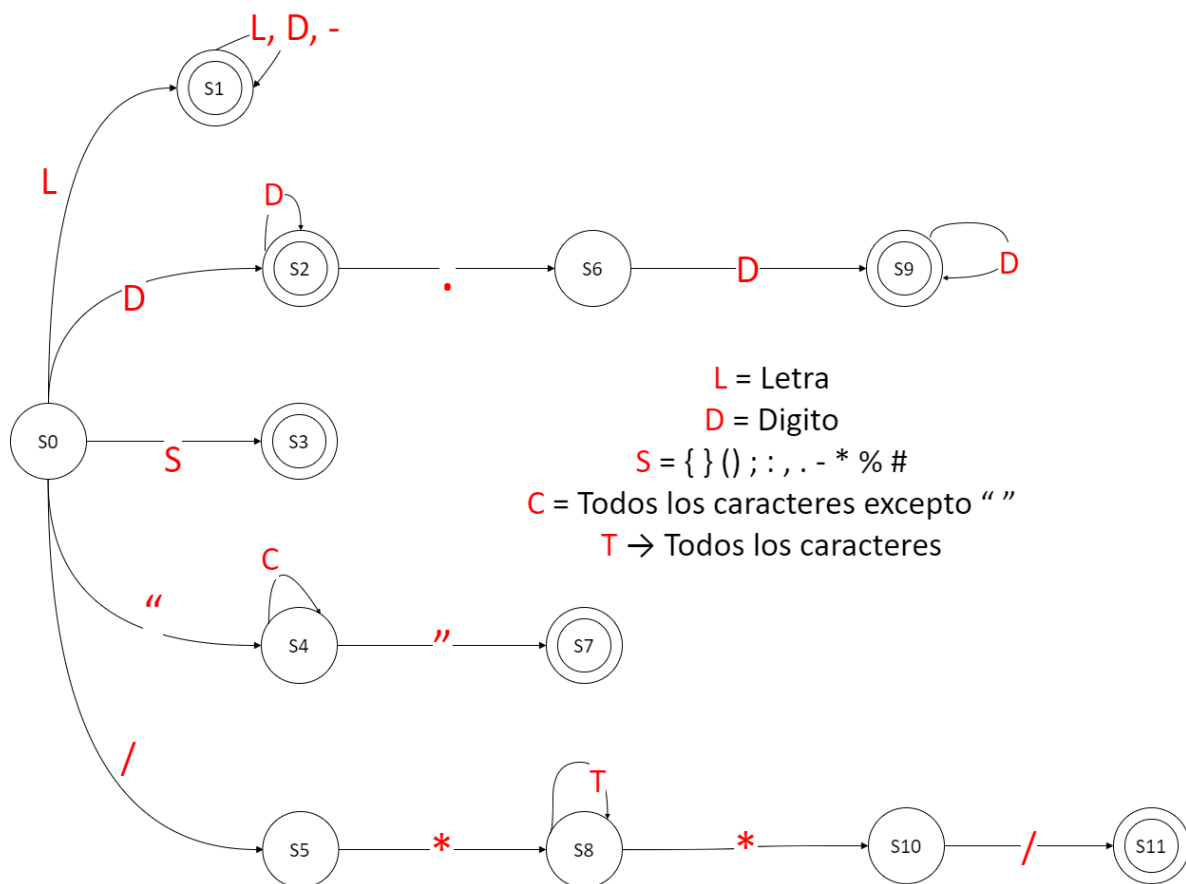
$" C^* " \#4$

//Cadena

$/ * T^* * / \#5$

//Comentarios

4. Autómata finito determinista Css



5. Expresiones regulares Html

$C \rightarrow$ Todos los caracteres excepto " o '

$S \rightarrow > / = . ()$

$L (L | D | -)^* \#1$

//Identificadores

$D^+ (. D^+)^? \#2$

//Numero entero o decimal

$S \#3$

//Símbolos

$" C^* " \#4$

//Cadena

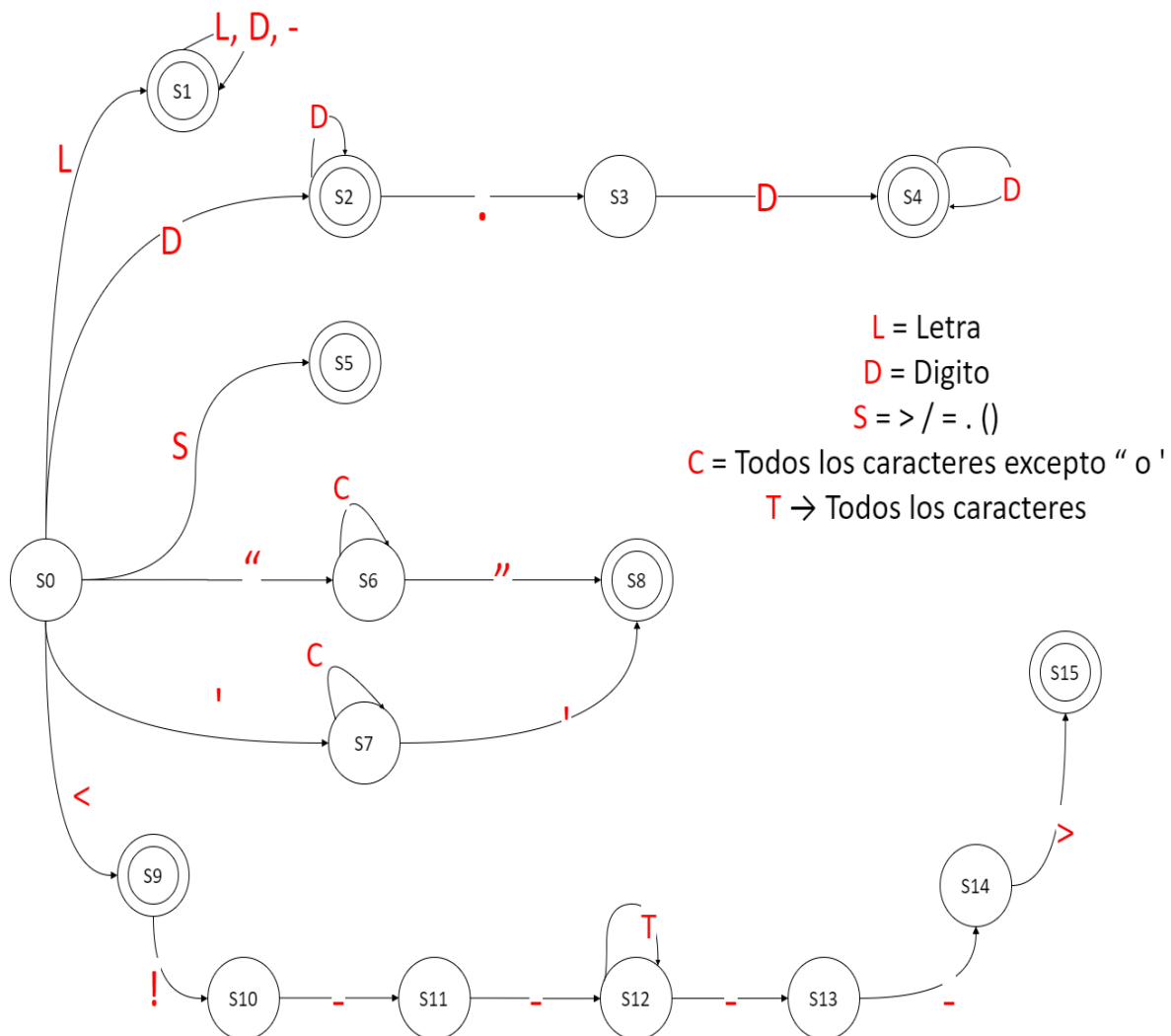
$' C^* ' \#5$

//Cadena caracteres

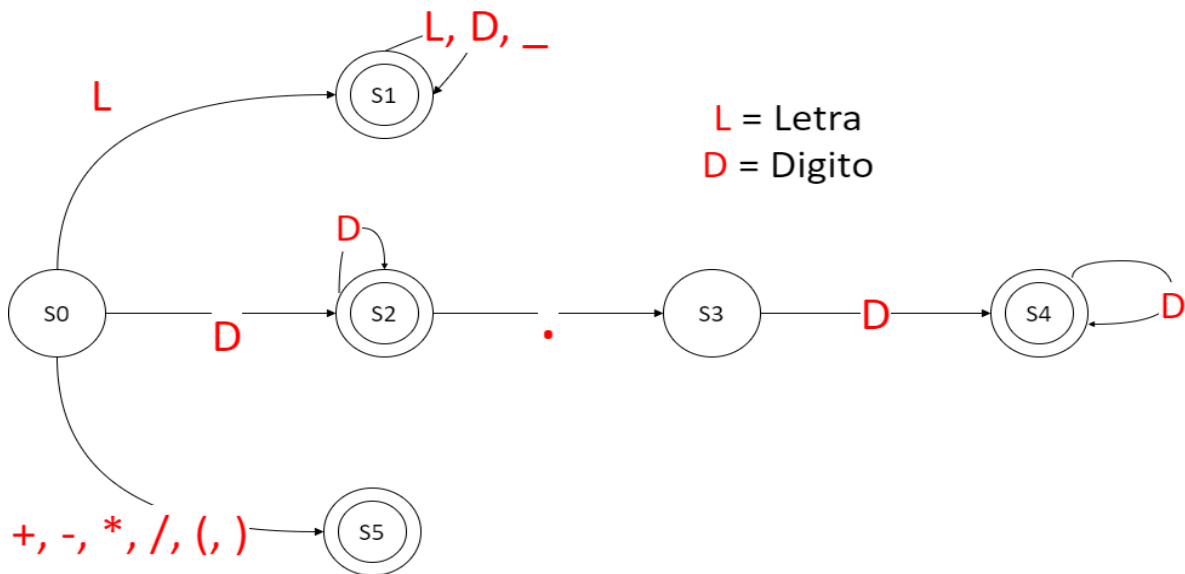
$< (- - T^* - - >)^? \#6$

//Menor que – Comentario

6. Autómata finito determinista Html



7. Autómata finito determinista Rmt



8. Gramática libre de contexto

GRAMATICA LIBRE DE CONTEXTO "EXPRESIONES ARITMETICAS" ARCHIVO RMT

$G = \{C, N, I, P\}$ donde:

C = Conjunto de terminales
N = Conjunto de no terminales
E = Símbolo inicial
P = Producciones

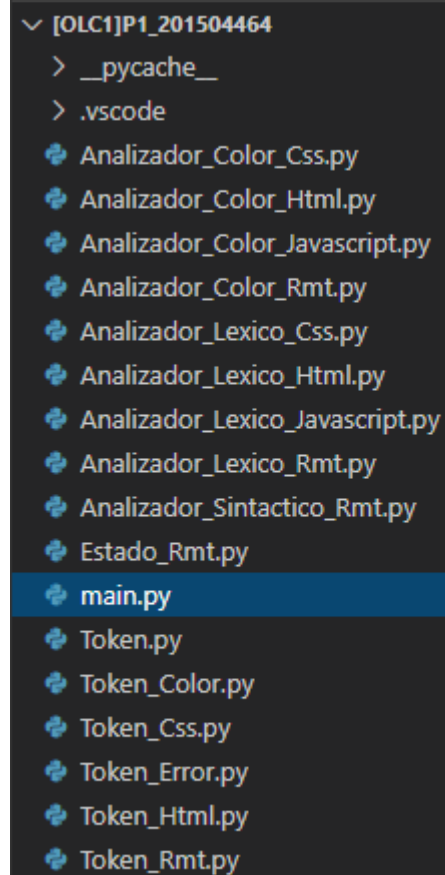
C = {
IDENTIFICADOR
NUMERO
PARENTESIS_IZQ
PARENTESIS_DER
SIGNO_MAS
SIGNO_MENOS
SIGNO_POR
SIGNO_DIVISION
}

N = {EXPRESION, E, E', T, T', F}

EXPRESION ::= E

```
E ::= T E'
E' ::= SIGNO_MAS T E'
      | SIGNO_MENOS T E'
      | ε
T ::= F T'
T' ::= SIGNO_POR F T'
      | SIGNO_DIVISION F T'
      | ε
F ::= IDENTIFICADOR
      | NUMERO
      | PARENTESIS_IZQ E PARENTESIS_DER
```

9. Estructura del programa

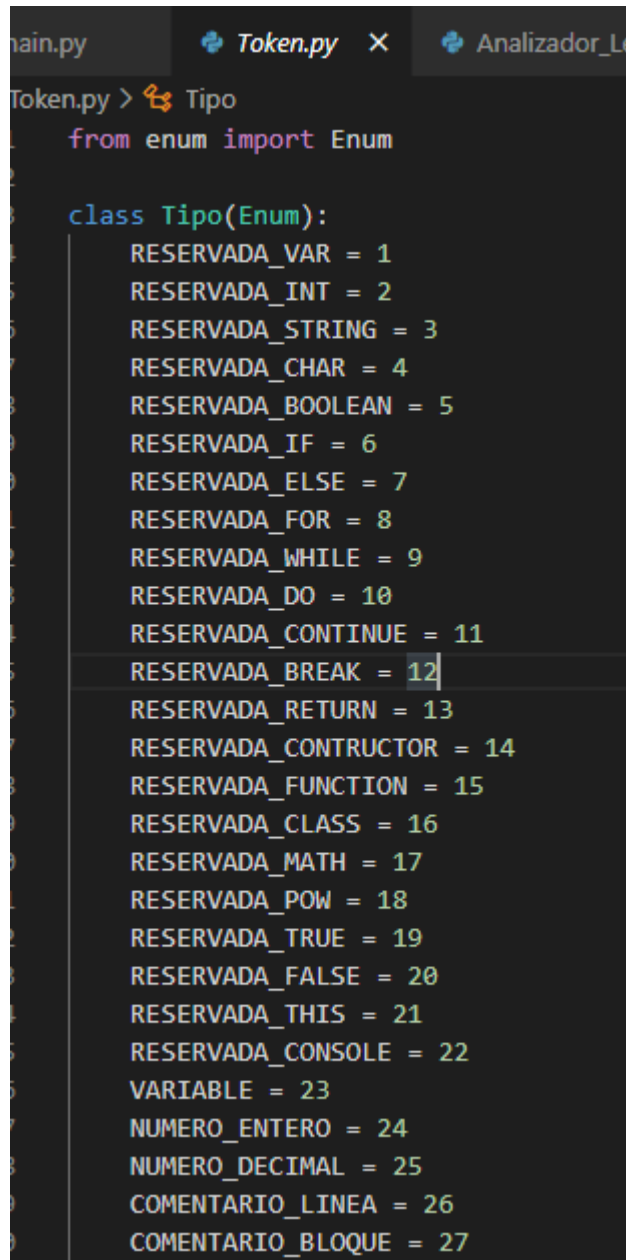


```

v [OLC1]P1_201504464
├── __pycache__
├── .vscode
├── Analizador_Color_Css.py
├── Analizador_Color_Html.py
├── Analizador_Color_Javascript.py
├── Analizador_Color_Rmt.py
├── Analizador_Lexico_Css.py
├── Analizador_Lexico_Html.py
├── Analizador_Lexico_Javascript.py
├── Analizador_Lexico_Rmt.py
├── Analizador_Sintactico_Rmt.py
├── Estado_Rmt.py
├── main.py
├── Token.py
├── Token_Color.py
├── Token_Css.py
├── Token_Error.py
├── Token_Html.py
├── Token_Rmt.py
```


10. Clase Token

Se declara un Enum llamado Tipo que representa al tipo de token que puede venir en el lenguaje definido.



```
main.py Token.py X Analizador_L  
Token.py > Tipo  
from enum import Enum  
  
class Tipo(Enum):  
    RESERVADA_VAR = 1  
    RESERVADA_INT = 2  
    RESERVADA_STRING = 3  
    RESERVADA_CHAR = 4  
    RESERVADA_BOOLEAN = 5  
    RESERVADA_IF = 6  
    RESERVADA_ELSE = 7  
    RESERVADA_FOR = 8  
    RESERVADA_WHILE = 9  
    RESERVADA_DO = 10  
    RESERVADA_CONTINUE = 11  
    RESERVADA_BREAK = 12  
    RESERVADA_RETURN = 13  
    RESERVADA_CONSTRUCTOR = 14  
    RESERVADA_FUNCTION = 15  
    RESERVADA_CLASS = 16  
    RESERVADA_MATH = 17  
    RESERVADA_POW = 18  
    RESERVADA_TRUE = 19  
    RESERVADA_FALSE = 20  
    RESERVADA_THIS = 21  
    RESERVADA_CONSOLE = 22  
    VARIABLE = 23  
    NUMERO_ENTERO = 24  
    NUMERO_DECIMAL = 25  
    COMENTARIO_LINEA = 26  
    COMENTARIO_BLOQUE = 27
```

Se declaran las variables que se utilizarán para obtener los datos respectivos a los nombres de cada una.

```
class Token:
    __tipoToken = Tipo
    __lexema = str
    __filaToken = int
    __columnaToken = int

    def __init__(self, tipo, auxlex, fila, columna):
        self.__tipoToken = tipo
        self.__lexema = auxlex
        self.__filaToken = fila
        self.__columnaToken = columna

    def getTipo(self):
        return self.__tipoToken

    def getLexema(self):
        return self.__lexema

    def getFila(self):
        return self.__filaToken

    def getColumna(self):
        return self.__columnaToken

    def getTipoEnString(self):
        self.nombreToken = ""

        if self.__tipoToken == Tipo.RESERVADA_VAR:
            self.nombreToken = "Reservada_Var"
        elif self.__tipoToken == Tipo.RESERVADA_INT:
            self.nombreToken = "Reservada_Int"
        elif self.__tipoToken == Tipo.RESERVADA_STRING:
            self.nombreToken = "Reservada_String"
        elif self.__tipoToken == Tipo.RESERVADA_CHAR:
```

El constructor de la clase Token recibe como parámetros los datos que corresponden a cada variable declarada anteriormente.

A cada variable se le establece su función get() para obtener el valor de cada una cuando se requiera.

11. Clase Token_Error

Se declara un enum llamado TipoError que representa al tipo de error en un token que puede venir en el lenguaje definido.

```
main.py  Token_Error.py X
Token_Error.py > ...
1  from enum import Enum
2
3  class TipoError(Enum):
4      LEXICO = 1
5      SINTACTICO = 2
6
7  class Token_Error:
8      __characterError = str
9      __tipoTokenError = TipoError
10     __descripcionError = str
11     __filaError = int
12     __columnaError = int
13
14     def __init__(self, caracter, tipoError, descripcion, fila, columna):
15         self.__characterError = caracter
```

Se declaran las variables que se utilizarán para obtener los datos respectivos a los nombres de cada una.

El constructor de la clase Token_Error recibe como parámetros los datos que corresponden a cada variable declarada anteriormente.

A cada variable se le establece su función get() para obtener el valor de cada una cuando se requiera.

```
class Token_Error:
    __characterError = str
    __tipoTokenError = TipoError
    __descripcionError = str
    __filaError = int
    __columnaError = int

    def __init__(self, caracter, tipoError, descripcion, fila, columna):
        self.__characterError = caracter
        self.__tipoTokenError = tipoError
        self.__descripcionError = descripcion
        self.__filaError = fila
        self.__columnaError = columna

    def getCaracterError(self):
        return self.__characterError

    def getTipoError(self):
        return self.__tipoTokenError

    def getDescripcionError(self):
        return self.__descripcionError

    def getFilaError(self):
        return self.__filaError

    def getColumnaError(self):
        return self.__columnaError
```

12. Clase Analizador_Lexico_Javascript

```
from Token import Token
from Token import Tipo
from Token_Error import Token_Error
from Token_Error import TipoError
from tkinter import messagebox

class Analizador_Lexico_Javascript:
    estado = int
    auxLexema = str
    textoDocumento = str
    lista_Tokens = list
    lista_Errores = list
    appendND = str
    dotID = str
    dotD = str
    dotC = str
```

Variables declaradas que servirán para el respectivo análisis del str textoDocumento obtenido del editor de texto.

```
def analizador_Javascript(self, textoDocumento):
    self.estado = 0
    self.auxLexema = ""
    self.textoDocumento = textoDocumento + "#"
    self.lista_Tokens = list()
    self.lista_Errores = list()
    self.c = ''
    self.columnaToken = 0
    self.filaToken = 1
    #---Validaciones para COMENTARIO_BLOQUE-----
    self.estadoComentario = 0 #Guarda el estado anterior del comentario para saber si es de linea o bloque
    self.filaComentarioBloque = 0
    self.columnaComentarioBloque = 0
    self.inicioComentarioBloque = False
    #---Validaciones para COMENTARIO_BLOQUE-----
    self.appendND = "" #Recolecta todos los caracteres omitiendo los errores
    self.dotID = ""
    self.dotD = ""
    self.dotC = ""
    self.terminoDotID, self.terminoDotD, self.terminoDotC = False, False, False
    self.lexemaDot = ""
    self.estado1Activo = False #Bandera para saber si el identificador tiene mas de 1 caracter
    self.estado2Activo, self.estado24Activo = False, False #Bandera para saber si hubo otro número antes de . o aceptacion
    self.estado16Activo = False #Bandera para saber si el comentario fue recursivo
```

Se inicializan las variables y listas que servirán para guardar los token y errores generados, así como las variables para validaciones dentro del analizador. Estas se inicializan dentro de la función “analizador_Javascript” que devolverá una lista de Token que recibe como parámetro un str textoDocumento.

El análisis inicia con un ciclo “while” que recorre la desfragmentación del str textoDocumento en caracteres individuales

Obteniendo los caracteres por medio del char llamado “c” se inicializa el estado en 0 y se verifica si “c” es una letra o un dígito o un carácter especial que pertenece al lenguaje o una comilla o si es un delimitador; si el carácter pertenece a alguno de estos entrara en el if correspondiente y el estado cambiara a conveniencia, la variable “auxLexema” ira guardando los caracteres hasta formar una palabra, cantidad o un carácter especial. Si el carácter que viene no entra en ninguno de estos se guardara en la lista de errores “lista_Errores” y se regresara al estado 0 y con el siguiente carácter a analizar.

```
i = 0
while i < len(self.textoDocumento):
    self.c = self.textoDocumento[i]
    self.columnaToken += 1

    #ESTADO S0
    if self.estado == 0:
        if self.c.isalpha():
            self.estado = 1
            self.auxLexema += self.c
            if self.terminoDotID == False:
                self.dotID = "digraph G{\nnode[shape = circle]\n\nS0 -> S1[label = \"" + self.c + "\"]\n"
        elif self.c.isdigit():
            self.estado = 2
            self.auxLexema += self.c
            if self.terminoDotD == False:
                self.dotD = "digraph G{\nnode[shape = circle]\n\nS0 -> S2[label = \"" + self.c + "\"]\n"
        elif (self.c == '{' or self.c == '}' or self.c == '(' or self.c == ')')
            or self.c == ';' or self.c == ':' or self.c == '.' or self.c == ','):
            self.estado = 3
            self.auxLexema += self.c
        elif self.c == "\"":
            self.estado = 4
            self.auxLexema += self.c
        elif self.c == "'":
            self.estado = 5
            self.auxLexema += self.c
        elif self.c == "/":
            self.estado = 6
            self.auxLexema += self.c
            if self.terminoDotC == False:
                self.lexemaDot = self.c + " "
        elif self.c == "&":
            self.estado = 7
```

En el estado 0 se determina en una condición si el carácter es # y si textoDocumento ha llegado a su último dato. Esto para finalizar el análisis léxico saliendo del "while", retornando la lista lista_Tokens con los tokens almacenados durante el análisis.

```
else:
    if self.c == "#" and i == len(self.textoDocumento) - 1:
        if len(self.lista_Errores) > 0:
            messagebox.showerror("Alerta", "Se han encontrado errores léxicos")
            #self.imprimirListaErrores()
            messagebox.showinfo("Aviso", "Análisis léxico satisfactorio")
        else:
            self.addTokenError(self.c, self.filaToken, self.columnaToken)
            #---validaciones para COMENTARIO_BLOQUE---
            self.terminoDotC = True
            i -= 1
            self.columnaToken -= 1
            i += 1

    return self.lista_Tokens
```

- La función analizador_Error() retorna la lista de errores encontrados.
- La función addToken() agrega un nuevo token formado por un patrón o carácter.
- La función addTokenError() agrega un nuevo token_Error el cual no esta definido en el lenguaje evaluado.
- La función imprimirListaErrores() recorre la lista de errores retornando un str con la impresión de estos.
- La función getRecolectorND() retorna un str que contiene toda la información del nuevo documento sin errores.
- La función getDotID() retorna un str que contiene el lenguaje dot formado durante el análisis para la expresión regular de un IDENTIFICADOR que forma un grafo utilizando la aplicación llamada Graphviz.
- La función getDotD() retorna un str que contiene el lenguaje dot formado durante el análisis para la expresión regular de un DIGITO que forma un grafo utilizando la aplicación llamada Graphviz.
- La función getDotC() retorna un str que contiene el lenguaje dot formado durante el análisis para la expresión regular de un COMENTARIO MULTILINEA que forma un grafo utilizando la aplicación llamada Graphviz.

```

def analizador_Error(self):
    return self.lista_Errores

def addToken(self, tipo, fila, columna):
    nuevoToken = Token(tipo, self.auxLexema, fila, columna)
    self.appendND += self.auxLexema # Recolector de caracteres
    self.lista_Tokens.append(nuevoToken)
    self.auxLexema = ""
    self.estado = 0

def addTokenError(self, caracter, fila, columna):
    if self.auxLexema == "":
        nuevoError = Token_Error(caracter, TipoError.LEXICO, "El simbolo no pertenece al lenguaje", fila, columna)
    else:
        nuevoError = Token_Error(caracter, TipoError.LEXICO, "Lexema no definido en el lenguaje", fila, columna)

    self.lista_Errores.append(nuevoError)
    self.auxLexema = ""
    self.estado = 0

def imprimirListaErrores(self):
    self.tokenError = Token_Error
    impresion = ""
    for self.tokenError in self.lista_Errores:
        #print(self.tokenError.getCaracterError() + " Tipo Error: " + self.tokenError.getTipoErrorEnString() + " " + self.tokenError.getDescr
        impresion += "Caracter: " + self.tokenError.getCaracterError() + " <-----> Tipo Error: " + self.tokenError.getTipoErrorEnString()

    return impresion

def getRecolectorND(self):
    return self.appendND

def getDotID(self):
    return self.dotID

```

13. Clase Analizador_Sintactico_Rmt

Variables declaradas que servirán para el respectivo análisis de la lista de tokens obtenida por el analizador léxico.

```
Analizador_Sintactico_Rmt.py > Analizador_Sintactico_Rmt > parsear

class Analizador_Sintactico_Rmt:
    numPreanalisis = int
    listaTokens = list()
    estadoOperacion = Estado

    def parsear(self, listaT):
        self.listaTokens = listaT
        self.preanalisis = Token_Rmt
        self.preanalisis = self.listaTokens[0]
        self.numPreanalisis = 0
        self.esCorrecto = True
        self.Expresion()

    def Expresion(self):
        self.E()
        if self.preanalisis.getTipo() == Tipo.DESCONOCIDO and self.esCorrecto:
            self.estadoOperacion = Estado.CORRECTO
        else:
            self.estadoOperacion = Estado.INCORRECTO

    def E(self):
        self.T()
        self.EP()

    def EP(self):
        if self.preanalisis.getTipo() == Tipo.SIGNO_MAS:
            self.match(Tipo.SIGNO_MAS)
            self.T()
            self.EP()
        elif self.preanalisis.getTipo() == Tipo.SIGNO_MENOS:
            self.match(Tipo.SIGNO_MENOS)
            self.T()
            self.EP()
```

El método parsear inicia el análisis obteniendo una lista de tokens, el método Expresión inicia las producciones.

Cada uno de los no terminales de la gramática libre de contexto se convertirán en una función que contendrán las llamadas a las funciones de los no terminales que producen y también a la función match que se le enviara como parámetro el terminal de la producción.

El método match es el que se encarga de comparar si el preanálisis de token es el esperado, si lo es este pasa al siguiente sin problema, de lo contrario producirá un error sintáctico. Este método tiene la capacidad al momento de encontrar un error poder reportarlo y tratar de recuperar la estructura en la que tienen que venir los tokens para poder generar la mayor cantidad de errores posibles.

```
def match(self, tipo):
    if not tipo == self.preanalisis.getTipo():
        print("Se esperaba " + self.getTipoParaError(tipo))
        self.esCorrecto = False

    if not self.preanalisis.getTipo() == Tipo.DESCONOCIDO:
        self.numPreanalisis += 1
        self.preanalisis = self.listaTokens[self.numPreanalisis]
```

Glosario de Términos

- **int estado:** valor entero iniciado en 0 que tomara el valor de los diferentes estados en el autómata finito.
- **str auxLexema:** variable en la que se concatenaran los caracteres analizados para formar las palabras, números o cadenas que se requieren.
- **str textoDocumento:** almacenara todo el texto recolectado en el editor de texto.
- **list lista_Tokens:** lista que almacenara palabras, números, cadenas como tipo Token.
- **List lista_Errores:** lista que almacenara caracteres, palabras, símbolos obtenidos como errores al lenguaje como tipo Token_Error.
- **char c:** variable tipo char que almacenara en cada ciclo del while el carácter obtenido de la desfragmentación del str textoDocumento.
- **int columnaToken:** tomara el valor de la posición en el que los caracteres se posicionan para indicar la columna en la que se encuentra el Token obtenido.
- **int filaToken:** indicara la fila en la que se encontró el token a guardar.