

PROYECTO 2

# MANUAL TÉCNICO

---

Lenguajes Formales y de Programación B-  
Nombre: Ana Lucia Fletes Ordóñez  
Carné: 202010003

# INTRODUCCIÓN

El siguiente manual técnico explica a detalle el funcionamiento del programa realizado en Python para el Proyecto #2 del curso de Lenguajes Formales y de Programación. Este consiste en una aplicación con interfaz gráfica que trabaje como un editor de texto (permitiendo las acciones abrir, editar, guardar y crear un nuevo archivo); y a su vez tenga el papel de analizador, tanto léxico y sintáctico (basado en autómatas finitos obtenidos por el método del árbol), el cual reconozca un lenguaje con una estructura definida, generando un archivo destinado a la presentación de errores durante el análisis, y otro para mostrar los resultados con las instrucciones analizadas y traducidas a lenguaje Nosql para MongoDB.

# ÍNDICE

<b>DISEÑO DE LA SOLUCIÓN.....</b>	<b>5</b>
TABLA DE TOKENS.....	5
COMENTARIO DE UNA LÍNEA.....	6
ÁRBOL.....	6
AUTÓMATA FINITO.....	6
GRAMÁTICA.....	7
COMENTARIO DE VARIAS LÍNEAS.....	8
ÁRBOL.....	8
AUTÓMATA FINITO.....	9
GRAMÁTICA.....	9
FUNCIONES.....	10
ÁRBOL.....	10
AUTÓMATA FINITO.....	11
GRAMÁTICA.....	11
PARAMETROS.....	12
ÁRBOL.....	12
AUTÓMATA FINITO.....	13
GRAMÁTICA.....	13
DATOS.....	14
ÁRBOL.....	14
AUTÓMATA FINITO.....	15
GRAMÁTICA.....	15
JSON.....	16
ÁRBOL.....	16
AUTÓMATA FINITO.....	17
GRAMÁTICA.....	17
<b>ESTRUCTURA DEL PROYECTO.....</b>	<b>18</b>
<b>CLASE MENUPRINCIPAL.....</b>	<b>20</b>
CONSTRUCTOR.....	20
MÉTODOS.....	21
crear_componentes().....	21
crear_menu().....	21
nuevo().....	22
borrar().....	22

abrir().....	23
guardar().....	23
guardar_como().....	24
salir().....	24
generar_sentencias().....	24
ver_tokens().....	25
ver_errores().....	26
<b>CLASE LECTOR.....</b>	<b>27</b>
ATRIBUTOS.....	27
MÉTODOS.....	27
leer_archivo().....	27
<b>CLASE FUNCION.....</b>	<b>28</b>
ATRIBUTOS.....	28
MÉTODOS.....	28
Setters.....	28
<b>CLASE ESCRITORTRADUCCION.....</b>	<b>29</b>
ATRIBUTOS.....	29
MÉTODOS.....	29
escribir_funcion().....	29
<b>CLASE ERROR.....</b>	<b>31</b>
ATRIBUTOS.....	31
MÉTODOS.....	31
Getters.....	31
<b>CLASE ESCRITORERRORES.....</b>	<b>32</b>
ATRIBUTOS.....	32
MÉTODOS.....	32
escribir_errores().....	32
<b>CLASE ESCRITORTOKENS.....</b>	<b>34</b>
ATRIBUTOS.....	34
MÉTODOS.....	34
escribir_tokens().....	34
agregar_json().....	35
agregar_set().....	35
agregar_tokens().....	36
comprobar_tokens().....	37
<b>CLASE ANALIZADOR.....</b>	<b>38</b>
ATRIBUTOS.....	38
MÉTODOS.....	39

verificar_token().....	39
juntar_caracteres().....	39
analizar_coincidencia().....	40
verificar_id().....	40
verificar_cadena().....	41
comentario_linea().....	41
comentario_varias_lineas().....	42
json().....	42
datos().....	44
parametros().....	44
compilar().....	45
guardar_error().....	46
<b>CLASE MAIN.....</b>	<b>47</b>

## DISEÑO DE LA SOLUCIÓN

Previo a la elaboración del código se utilizó el método del árbol para hallar el autómata finito y la gramática respectiva; en donde se presentan los componentes léxicos o tokens válidos o aceptables que se deben reconocer durante cada paso o estado. Asimismo se llevó a cabo una tabla de los tokens aceptables.

### TABLA DE TOKENS

No.	Token	Patrón
1	Tk_Funcion	TK_CrearBD   TK_EliminarBD   Tk_CrearC   Tk_EliminarC   Tk_InsertarU   Tk_ActualizarU   Tk_EliminarU   Tk_BuscarT   Tk_BuscarU
2	Tk_CrearBD	("C")("r")("e")("a")("r")("B")("D")
3	Tk_EliminarBD	("E")("l")("m")("l")("n")("a")("r")("B")("D")
4	Tk_CrearC	("C")("r")("e")("a")("r")("C")("o")("l")("e")("c")("c")("l")("o")("n")
5	Tk_EliminarC	("E")("l")("m")("l")("n")("a")("r")("C")("o")("l")("e")("c")("c")("l")("o")("n")
6	Tk_InsertarU	("l")("n")("s")("e")("r")("t")("a")("r")("U")("n")("l")("c")("o")
7	Tk_ActualizarU	("A")("c")("l")("u")("a")("r")("z")("a")("r")("U")("n")("l")("c")("o")
8	Tk_EliminarU	("E")("l")("m")("l")("n")("a")("r")("U")("n")("l")("c")("o")
9	Tk_BuscarT	("B")("u")("s")("c")("r")("T")("o")("d")("o")
10	Tk_BuscarU	("B")("u")("s")("c")("a")("r")("U")("n")("l")("c")("o")
11	Tk_ID	CARACTER(CARACTER)*
12	Tk_Igual	"="
13	Tk_Nueva	("n")("u")("e")("v")("a")
14	Tk_ParA	"("
15	Tk_ParC	")"
16	Tk_PtoComa	"."
17	Tk_ComillaD	""
18	Tk_Coma	" , "
19	Tk_LLaveA	"{"
20	Tk_LLaveC	"}"
21	Tk_DosPts	"::"
22	Tk_Set	("\$")("s")("e")("t")

## COMENTARIO DE UNA LÍNEA

### ÁRBOL

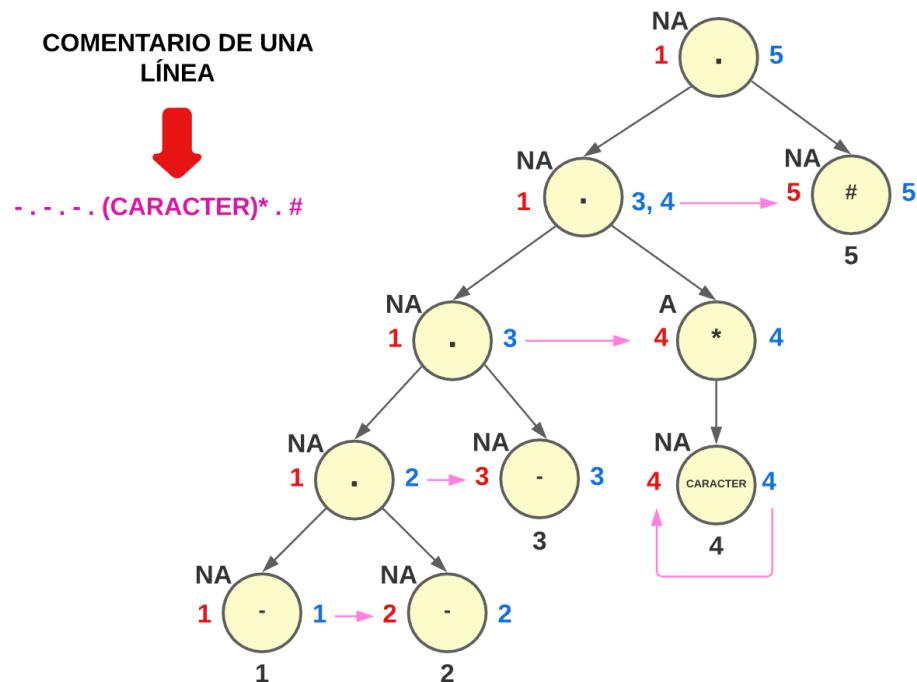


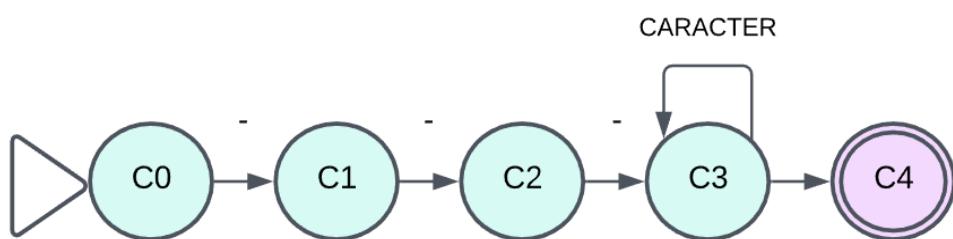
TABLA DE SIGUIENTE

i	SIGUIENTEPOS(i)
1	2
2	3
3	4, 5
4	4, 5
5	

TABLA DE TRANSICIONES

ESTADO	SIGUIENTEPOS(i)	-	CARACTER	#
C0	1	1		
C1	2	2		
C2	3	3		
C3	4,5		4	5
C4	5			5

### AUTÓMATA FINITO



---

## GRAMÁTICA

CARACTER = {A, a, B, b, … , Z, z, 0, 1, … , 9, +, -, …}

ESTADOS = {C0, C1, C2, C3, C4}

ESTADO INICIAL = C0

ESTADO FINAL = C4

**C0 -> - C1**

**C1 -> - C2**

**C2 -> - C3**

**C3 -> CARACTER C3 | C4**

**C4 -> ESTADO DE ACEPTACIÓN**

## COMENTARIO DE VARIAS LÍNEAS

### ÁRBOL

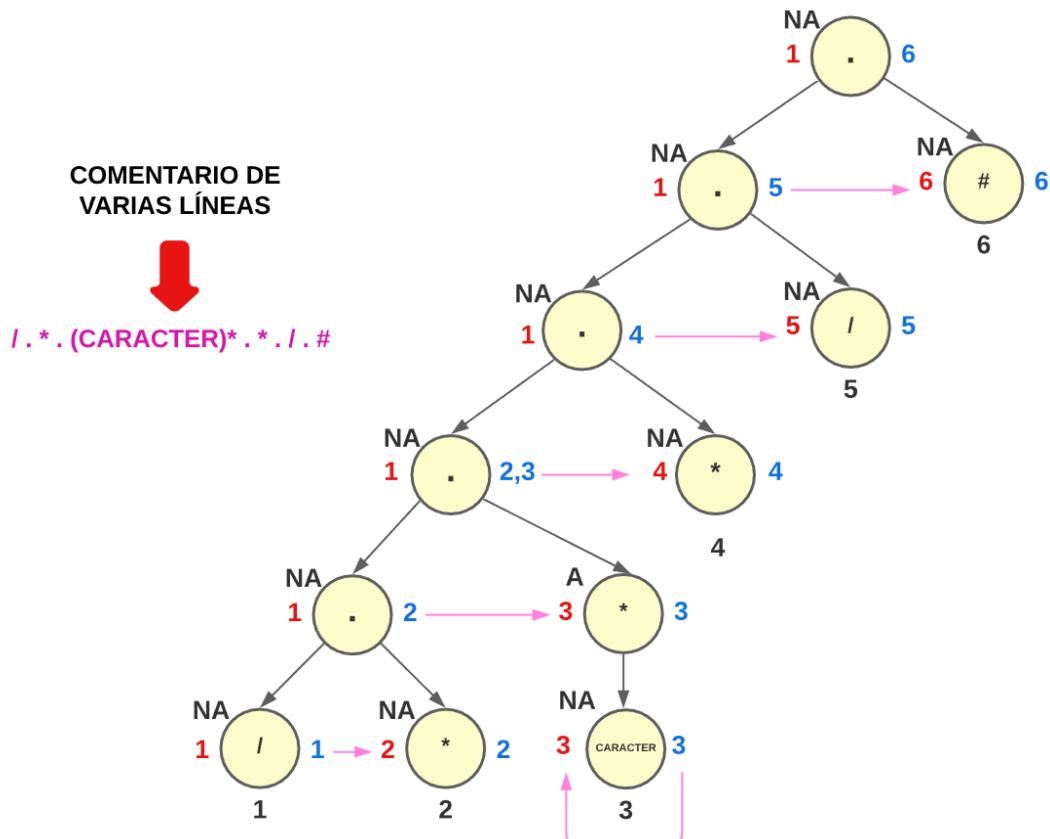


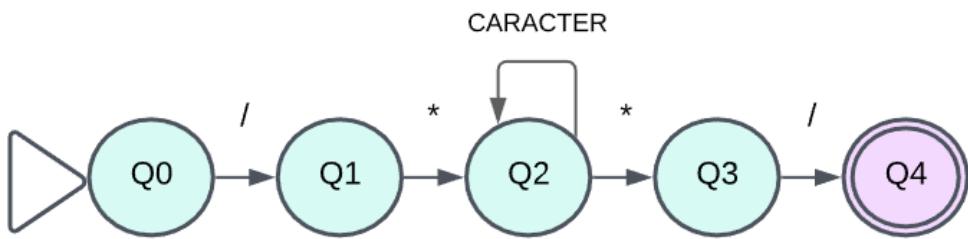
TABLA DE SIGUIENTE

i	SIGUIENTEPOS(i)
1	2
2	3, 4
3	3, 4
4	5
5	6
6	

TABLA DE TRANSICIONES

ESTADO	SIGUIENTEPOS(i)	/	*	CARACTER	#
Q0	1	1			
Q1	2		2		
Q2	3, 4		4	3	
Q3	5	5			
Q4	6				6

## AUTÓMATA FINITO



## GRAMÁTICA

CARACTER = {A, a, B, b, ..., Z, z, 0, 1, ..., 9, +, -, ...}

ESTADOS = {Q0, Q1, Q2, Q3, Q4}

ESTADO INICIAL = Q0

ESTADO FINAL = Q4

**Q0 -> / Q1**

**Q1 -> \* Q2**

**Q2 -> CARACTER Q2 | \* Q3**

**Q3 -> / Q4**

**Q4 -> ESTADO DE ACEPTACIÓN**

## FUNCIONES

### ÁRBOL

FUNCIONES      ↘ FUNCION . ID . = . nueva . FUNCION . ( . (PARAMETROS)? . ) . ; .#

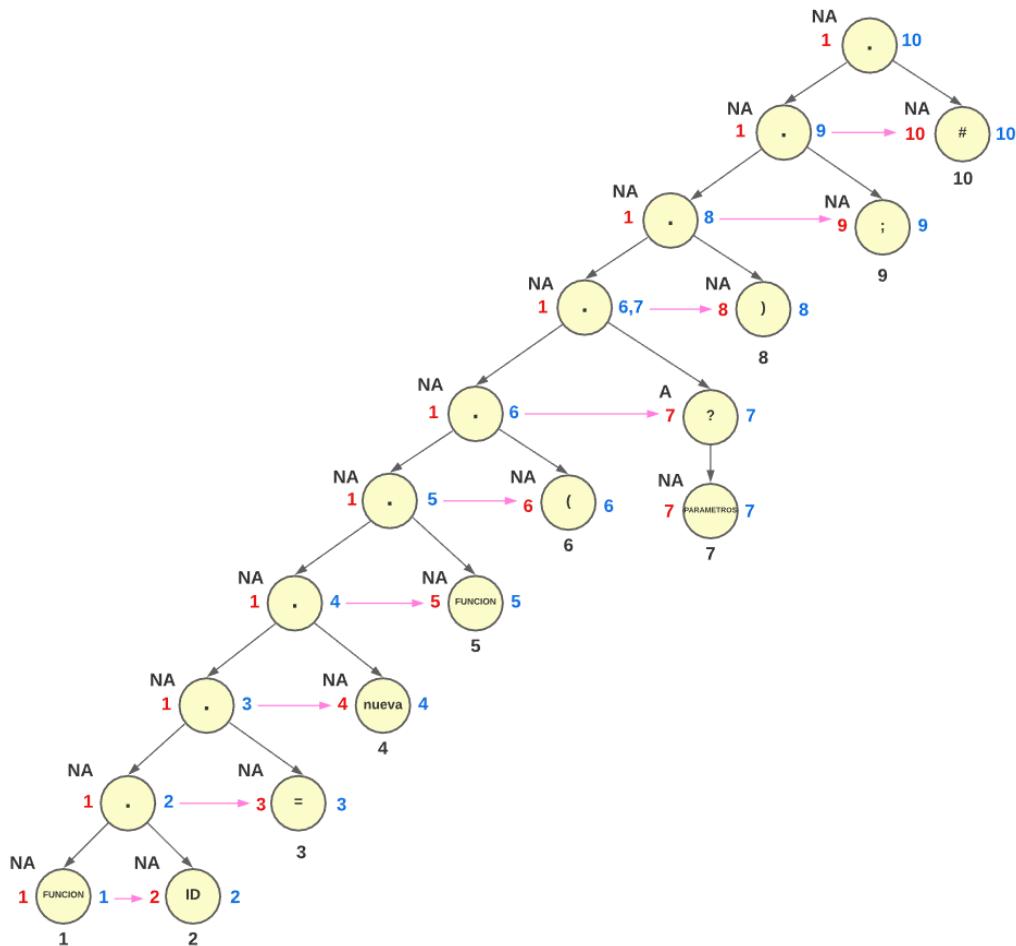


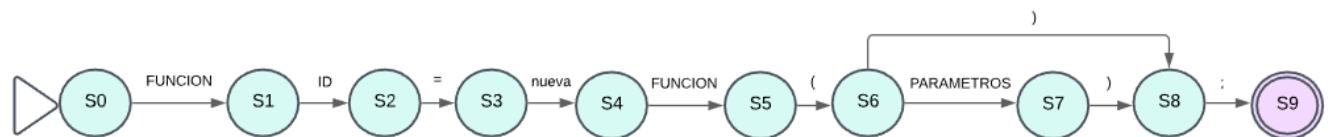
TABLA DE SIGUIENTE

i	SIGUIENTEPOS(i)
1	2
2	3
3	4
4	5
5	6
6	7,8
7	8
8	9
9	10
10	

TABLA DE TRANSICIONES

ESTADO	SIGUIENTEPOS(i)	FUNCION	ID	=	nueva	(	PARAMETROS	)	;	#
S0	1	1								
S1	2		2							
S2	3			3						
S3	4				4					
S4	5	5								
S5	6					6				
S6	7,8						7	8		
S7	8							8		
S8	9								9	
S9	10									10

## AUTÓMATA FINITO



## GRAMÁTICA

FUNCION = {CrearBD, EliminarBD, CrearColeccion,  
EliminarColeccion, InsertarUnico, ActualizarUnico,  
EliminarUnico, BuscarTodo, BuscarUnico}

ESTADOS = {S0, S1, S2, S3, S4, S5, S6, S7, S8, S9}

ESTADO INICIAL = S0

ESTADO FINAL = S9

**S0 -> FUNCION S1**  
**S1 -> ID S2**  
**S2 -> = S3**  
**S3 -> nueva S4**  
**S4 -> FUNCION S5**  
**S5 -> ( S6**  
**S6 -> PARAMETROS S7 | ) S8**  
**S7 -> ) S8**  
**S8 -> ; S9**  
**S9 -> ESTADO DE ACEPTACIÓN**

## PARAMETROS

### ÁRBOL

PARAMETROS → ". (ID)? . " . ( . " . JSON . ")? . #

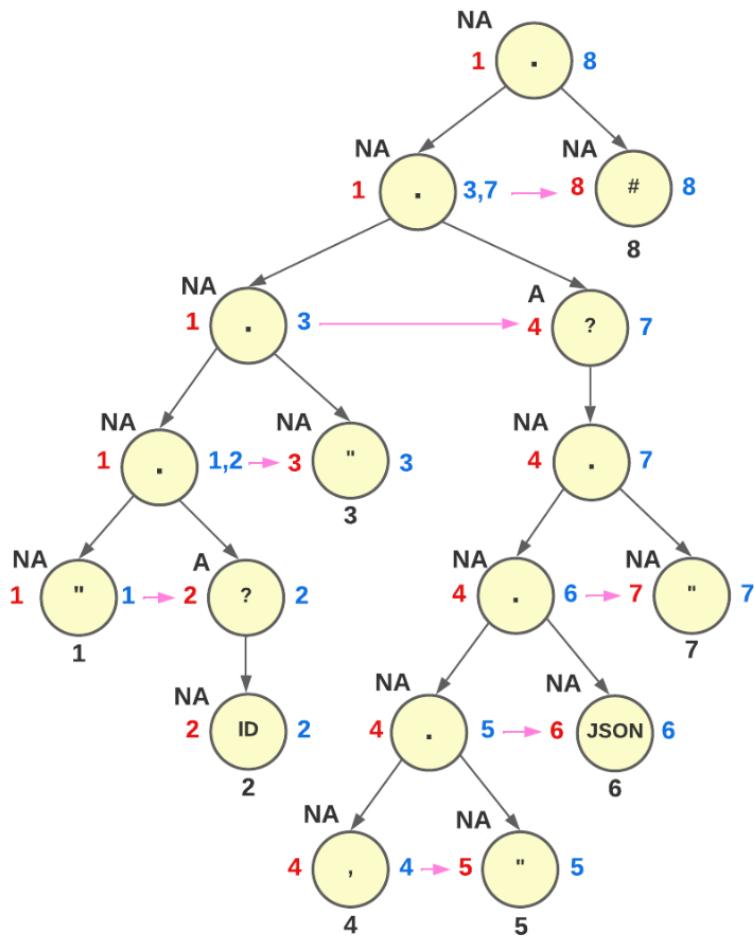


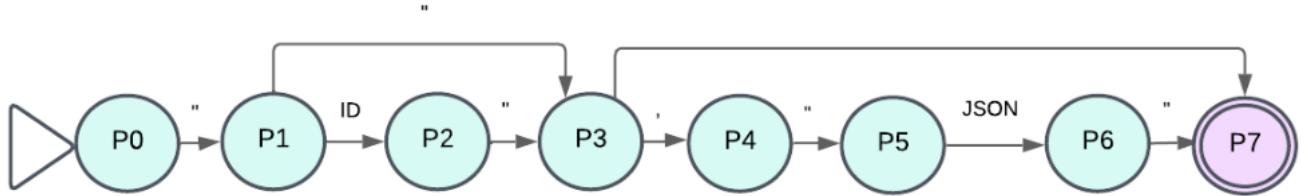
TABLA DE SIGUIENTE

i	SIGUIENTEPOS(i)
1	2,3
2	3
3	4,8
4	5
5	6
6	7
7	8
8	

TABLA DE TRANSICIONES

ESTADO	SIGUIENTEPOS(i)	"	ID	,	JSON	#
P0	1	1				
P1	2,3	3	2			
P2	3	3				
P3	4,8			4		8
P4	5	5				
P5	6				6	
P6	7	7				
P7	8					8

## AUTÓMATA FINITO



## GRAMÁTICA

ESTADOS = {P0, P1, P2, P3, P4, P5, P6, P7}

ESTADO INICIAL = P0

ESTADO FINAL = P7

**P0 -> " P1**  
**P1 -> ID P2 | " P3**  
**P2 -> " P3**  
**P3 -> , P4 | P7**  
**P4 -> " P5**  
**P5 -> JSON P6**  
**P6 -> " P7**  
**P7 -> ESTADO DE ACEPTACIÓN**

## DATOS

ÁRBOL

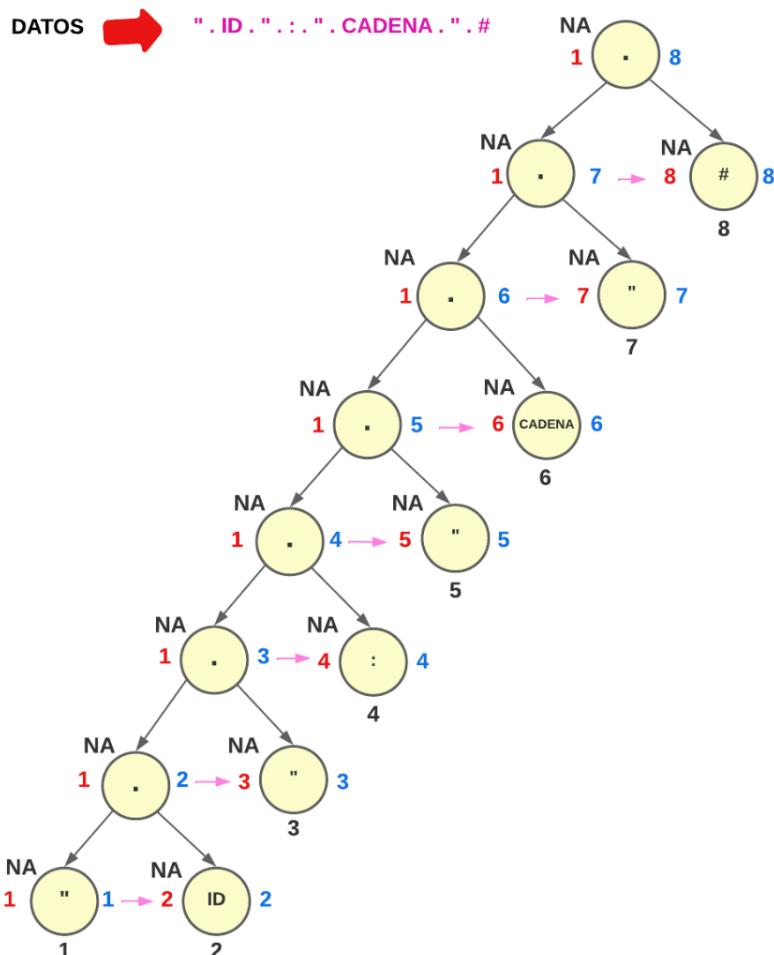
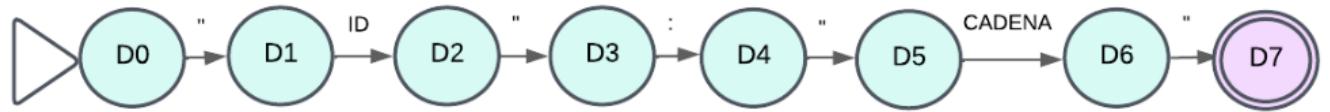


TABLA DE SIGUIENTE	
i	SIGUIENTEPOS(i)
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	

TABLA DE TRANSICIONES						
ESTADO	SIGUIENTEPOS(i)	"	ID	:	CADENA	#
D0	1	1				
D1	2		2			
D2	3	3				
D3	4			4		
D4	5	5				
D5	6				6	
D6	7	7				
D7	8					8

## AUTÓMATA FINITO



## GRAMÁTICA

ESTADOS = {D0, D1, D2, D3, D4, D5, D6, D7}

ESTADO INICIAL = D0

ESTADO FINAL = D7

D0 -> " D1

D1 -> ID D2

D2 -> " D3

D3 -> : D4

D4 -> " D5

D5 -> CADENA D6

D6 -> " D7

D7 -> ESTADO DE ACEPTACIÓN

## JSON

### ÁRBOL

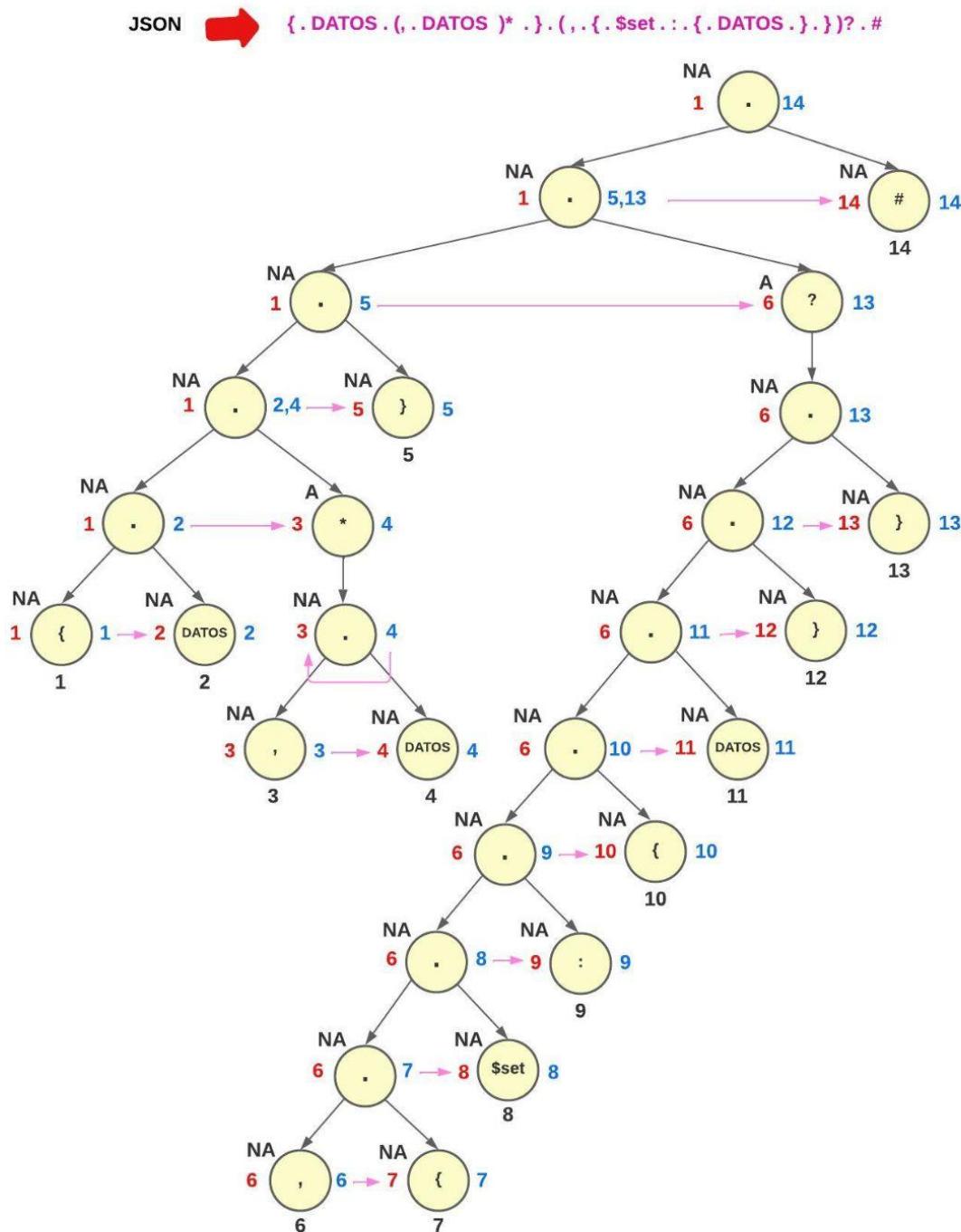


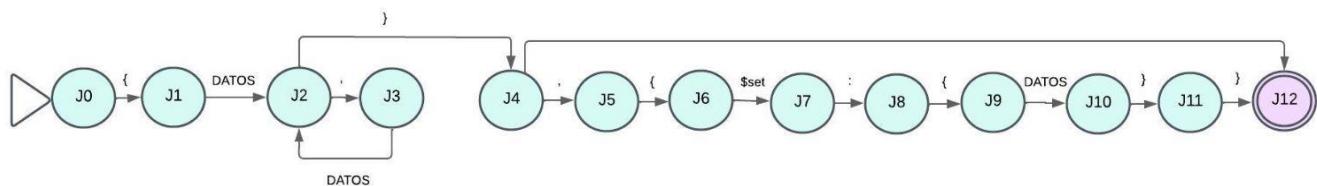
TABLA DE SIGUIENTE

i	SIGUIENTEPOS(i)
1	2
2	3,5
3	4
4	3,5
5	6,14
6	7
7	8
8	9
9	10
10	11
11	12
12	13
13	14
14	

TABLA DE TRANSICIONES

ESTADO	SIGUIENTEPOS(i)	{	DATOS	,	}	\$set	:	#
J0	1	1						
J1	2		2					
J2	3,5			3	5			
J3	4		4					
J4	6,14			6			14	
J5	7	7						
J6	8					8		
J7	9						9	
J8	10	10						
J9	11		11					
J10	12				12			
J11	13				13			
J12	14						14	

## AUTÓMATA FINITO



## GRAMÁTICA

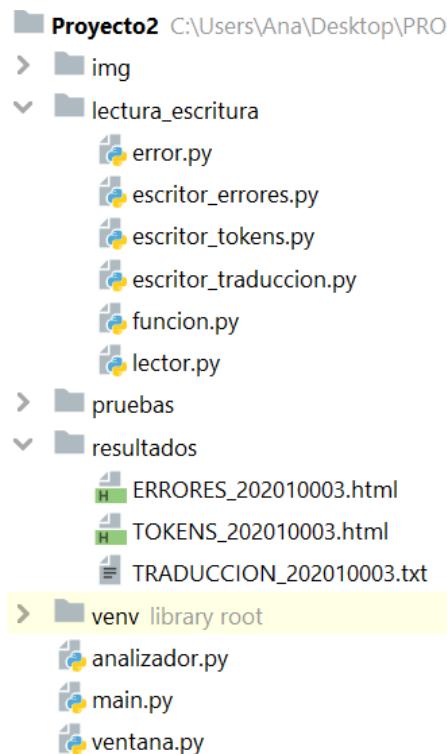
ESTADOS = {J0, J1, J2, J3, J4, J5, J6, J7, J8, J9, J10, J11, J12}

ESTADO INICIAL = J0

ESTADO FINAL = J12

**J0 -> { J1**  
**J1 -> DATOS J2**  
**J2 -> , J3 | } J4**  
**J3 -> DATOS J2**  
**J4 -> , J5 | J12**  
**J5 -> { J6**  
**J6 -> \$set J7**  
**J7 -> : J8**  
**J8 -> { J9**  
**J9 -> DATOS J10**  
**J10 -> } J11**  
**J11 -> } J12**  
**J12 -> ESTADO DE ACEPTACIÓN**

# ESTRUCTURA DEL PROYECTO



El proyecto consta de las siguientes carpetas y archivos .py, algunas de las cuales se detallarán a profundidad en las siguientes páginas:

- **Directorio img**

- Contiene el ícono a colocar en la ventana.

- **Directorio lectura\_escritura**

- error
- escritor\_errores
- escritor\_tokens
- escritor\_traducion
- funcion
- lector

- **Directorio pruebas**

- Contiene los archivos .txt para realizar pruebas de las distintas funciones de la aplicación.
- **Directorio resultados**
  - Destinado a contener el archivo .txt con las traducciones de las distintas sentencias analizadas; y los archivos .html para la tabla de tokens y para la tabla de errores encontrados durante el análisis.
- analizador
- main
- ventana

## CLASE MENUPRINCIAL

Esta clase contiene el código asociado a la creación de la ventana para la interfaz gráfica de la aplicación; por lo que contiene los métodos para la creación de componentes y administración de eventos de los mismos.

La clase ventana importa las siguientes librerías y clases para su uso posterior.

```
import sys
import subprocess
import tkinter as tk
from tkinter import messagebox, Menu
from tkinter.filedialog import askopenfile, asksaveasfilename
from lectura_escritura.lector import Lector
from analizador import Analizador
```

## CONSTRUCTOR

Dentro del constructor de la clase MenuPrincipal se tiene la configuración básica de la ventana, como lo es el tamaño, título e ícono de la misma. Asimismo, se tienen definidos algunos atributos o componentes que se usarán en los demás métodos de la clase, y se tiene llamadas a métodos para crear la barra de menús y el editor de texto.

```
class MenuPrincipal(tk.TK):
    def __init__(self):
        super().__init__()
        self.resizable(0, 0)
        self.title("Proyecto #2 - LFP")
        self.iconbitmap("img/cheems.ico")
        self.archivo = None
        self.archivo_activo = None
        self.fila = 1
        self.columna = 1
        self.label_ruta = tk.Label(self, text="")
        self.campo_texto = tk.Text(self)
        self.crear_menu()
        self.crear_componentes()
```

## MÉTODOS

### crear\_componentes()

Con este método se crea la etiqueta para almacenar la ruta del archivo abierto y se configura el campo de texto, posicionándolo al centro.

```
def crear_componentes(self):
    tk.Label(self, text="").grid(row=1, column=1, columnspan=3)
    tk.Label(self, text="\t").grid(row=2, column=1)
    tk.Label(self, text="- Ruta del archivo activo -").grid(row=2, column=2)
    tk.Label(self, text="\t").grid(row=2, column=3)
    self.label_ruta.grid(row=3, column=2)
    tk.Label(self, text="").grid(row=4, column=1, columnspan=3)
    self.campo_texto.grid(row=5, column=2)
    tk.Label(self, text="").grid(row=6, column=1, columnspan=3)
    tk.Label(self, text=f"Ln {self.fila}, Col{self.columna}").grid(row=7, column=1, columnspan=3)
    tk.Label(self, text="").grid(row=8, column=1, columnspan=3)
```

### crear\_menu()

Este método se encarga de la creación de los menús y submenús de la ventana, teniendo como menús principales “Archivo” con submenús “Nuevo”, “Abrir”, “Guardar”, “Guardar Como” y “Salir”; “Análisis” con submenú “Generar sentencias MongoDB”; “Tokens” con submenú “Ver tokens”; y “Errores” con submenú “Ver errores”. Cada submenú está ligado a un método para cumplir con determinada función.

```
def crear_menu(self):
    # Crear el menú principal
    menu_principal = Menu(self)
    # Crear submenús
    submenu_archivo = Menu(menu_principal, tearoff=False)
    submenu_analisis = Menu(menu_principal, tearoff=0)
    submenu_tokens = Menu(menu_principal, tearoff=0)
    submenu_errores = Menu(menu_principal, tearoff=0)
    # Agregar opciones y separadores en el submenú "Archivo"
    submenu_archivo.add_command(label="Nuevo", command=self.nuevo)
    submenu_archivo.add_separator()
    submenu_archivo.add_command(label="Abrir", command=self.abrir)
    submenu_archivo.add_separator()
    submenu_archivo.add_command(label="Guardar", command=self.guardar)
    submenu_archivo.add_separator()
    submenu_archivo.add_command(label="Guardar Como", command=self.guardar_como)
    submenu_archivo.add_separator()
    submenu_archivo.add_command(label="Salir", command=self.salir)
    # Agregar opción al submenú "Análisis"
    submenu_analisis.add_command(label="Generar sentencias MongoDB", command=self.generar_sentencias)
```

```

# Agregar opción al submenu "Tokens"
submenu_tokens.add_command(label="Ver tokens", command=self.ver_tokens)
# Agregar opción al submenu "Errores"
submenu_errores.add_command(label="Ver errores", command=self.ver_errores)
# Agregar los submenús al menú principal
menu_principal.add_cascade(menu=submenu_archivo, label="Archivo")
menu_principal.add_cascade(menu=submenu_analisis, label="Análisis")
menu_principal.add_cascade(menu=submenu_tokens, label="Tokens")
menu_principal.add_cascade(menu=submenu_errores, label="Errores")
# Mostrar el menú en la ventana principal
self.config(menu=menu_principal)

```

## nuevo()

El método nuevo, en caso de que exista un archivo activo, muestra una ventana de diálogo, en la que se permite decidir si guardar o no lo que esté escrito en el editor. Si se guardan los cambios, se mandará a llamar al método guardar\_como(); y en caso contrario, se limpia el editor de código.

```

def nuevo(self):
    # Verificar si hay un archivo abierto o activo
    if self.archivo_activo != None:
        respuesta = messagebox.askyesno("", "¿Desea guardar los cambios?")
        if respuesta:
            self.guardar_como()
            self.borrar()
        else:
            self.borrar()
    else:
        self.borrar()

```

## borrar()

El método borrar elimina el texto contenido en el campo de texto del editor, cambia la ruta del archivo a ninguna y establece archivo\_activo como None, ya que básicamente se cerró.

```

def borrar(self):
    # Eliminar el texto anterior
    self.campo_texto.delete(1.0, tk.END)
    self.label_ruta["text"] = ""
    self.archivo_activo = None

```

## abrir()

El método abrir desplegará una ventana del Explorador de archivos para seleccionar el archivo deseado, el cual se almacenará en la variable archivo\_activo. Se verifica que se haya seleccionado algún archivo y en caso de que sí, se abre el mismo en modo lectura, asignándose esto a la variable archivo. Se elimina el texto que se encuentre en el campo de texto de la ventana y se inserta el texto del archivo leído.

```
def abrir(self):
    try:
        # askopenfile (explorador para abrir archivo)
        self.archivo_activo = askopenfile(mode="r")
        # Verificar si se seleccionó un archivo a abrir
        if self.archivo_activo != None:
            with open(self.archivo_activo.name, "r") as self.archivo:
                texto = self.archivo.read()
                # Eliminar el texto anterior
                self.campo_texto.delete(1.0, tk.END)
                # Insertar el contenido del archivo
                self.campo_texto.insert(1.0, texto)
                self.label_ruta["text"] = f"{self.archivo.name}"
    except:
        messagebox.showerror("ERROR", "No pudo abrirse el archivo indicado")
        self.archivo_activo = None
```

## guardar()

El método guardar verificará que exista algún archivo abierto con anterioridad y que esté activo; si es así, entonces abre el archivo en modo escritura y lo asigna a la variable archivo. La variable texto almacenará todo el contenido del campo de texto y este se mandará como parámetro en el método write para escribir dicho contenido en el archivo. En el caso de que no haya ningún archivo activo y se quiera guardar el texto, se mandará a llamar al método guardar\_como().

```
def guardar(self):
    # Verificar si hay un archivo abierto o activo
    if self.archivo_activo != None:
        with open(self.archivo_activo.name, "w") as self.archivo:
            # Leer el contenido del cuadro de texto
            texto = self.campo_texto.get(1.0, tk.END)
            # Escribir contenido del cuadro de texto en el mismo archivo
            self.archivo.write(texto)
    else:
        self.guardar_como()
```

## guardar\_como()

El método guardar\_como tendrá algunas configuraciones iniciales, como la extensión del archivo a guardar, siendo por defecto txt; se desplegará una ventana del Explorador de archivos para escribir el nombre del archivo a guardar y seleccionar la ruta del mismo, asignándose esto al atributo archivo. En caso de que sí se vaya a guardar el archivo, se abre el archivo en modo escritura y lo asigna a la variable archivo. La variable texto almacenará todo el contenido del campo de texto y este se mandará como parámetro en el método write para escribir dicho contenido en el archivo.

```
def guardar_como(self):
    # Configuración respecto a la extensión del archivo
    self.archivo = asksaveasfilename(
        defaultextension=".txt",
        filetypes=[("Archivos de texto", "*.txt"), ("Todos los archivos", "*.*")]
    )
    # Verificar si se va a guardar un archivo
    if self.archivo:
        with open(self.archivo, "w") as self.archivo:
            # Leer el contenido del cuadro de texto
            texto = self.campo_texto.get(1.0, tk.END)
            # Escribir contenido del cuadro de texto en el nuevo archivo
            self.archivo.write(texto)
            self.label_ruta["text"] = f"{self.archivo.name}"
            self.archivo_activo = self.archivo
```

## salir()

El método salir cerrará y destruirá el objeto ventana, y terminará o saldrá del proceso de ejecución.

```
def salir(self):
    self.quit()
    self.destroy()
    sys.exit()
```

## generar\_sentencias()

El método generar\_sentencias primero verifica que haya un archivo abierto para poder realizar el análisis del mismo. Si es así, entonces se crea un objeto de la clase Lector, pasándose la ruta del archivo a leer; luego, se manda a llamar el método leer\_archivo(),

guardando las líneas leídas del archivo en la variable archivo\_leido. Después, se crea un objeto de la clase Analizador, en donde se envían las líneas leídas como parámetro, y se llama el método compilar() para ejecutar el análisis del archivo. Al finalizar el análisis, aparecerá una ventana de diálogo informando que el Análisis se ha realizado. Luego, se abre el archivo con la traducción de las funciones encontradas, se lee y escribe su contenido en el editor de código.

```
def generar_sentencias(self):
    # Verifica si hay algún archivo abierto con anterioridad
    if not self.archivo:
        messagebox.showerror("ERROR", "No se ha abierto ningún archivo para analizar")
        return
    archivo_leer = Lector(self.archivo_activo.name)
    archivo_leido = archivo_leer.leer_archivo()
    analizador = Analizador(archivo_leido)
    analizador.compilar()
    messagebox.showinfo("INFORMACIÓN", "Análisis realizado")

try:
    with open("resultados\TRADUCCION_202010003.txt", "r") as self.archivo:
        # Eliminar el texto anterior
        self.campo_texto.delete(1.0, tk.END)
        # Leer el contenido del archivo y guardarlo en la variable texto
        texto = self.archivo.read()
        # Insertar el contenido del archivo
        self.campo_texto.insert(1.0, texto)
        self.label_ruta["text"] = ""
        self.archivo = None
        self.archivo_activo = None
except:
    messagebox.showerror("ERROR", "No pudo abrirse el archivo de errores")
    self.archivo_activo = False
```

### ver\_tokens()

El método ver\_tokens abrirá en el navegador el archivo .html generado con la información de los tokens encontrados al realizar el análisis, si es que este existe.

```
def ver_tokens(self):
    try:
        ruta = "resultados\TOKENS_202010003.html"
        leer = open(ruta, "r")
        leer.close()
        subprocess.Popen([ruta], shell=True)
    except:
        messagebox.showerror("ERROR", "No existe un archivo de tokens")
```

### ver errores()

El método ver\_errores abrirá en el navegador el archivo .html generado con la información de los errores encontrados al realizar el análisis, si es que este existe.

```
def ver_errores(self):
    try:
        ruta = "resultados\ERRORES_202010003.html"
        leer = open(ruta, "r")
        leer.close()
        subprocess.Popen([ruta], shell=True)
    except:
        messagebox.showerror("ERROR", "No existe un archivo de errores")
```

## CLASE LECTOR

La clase Lector se encarga de leer un archivo y devolver la información del mismo.

### ATRIBUTOS

La clase Lector cuenta con dos atributos, uno denominado “ruta”, que representa la dirección del archivo a leer y del cual se recibe el valor como parámetro en el constructor; y lineas, que almacenará la información del archivo a leer.

```
class Lector():

    def __init__(self, ruta):
        self.ruta = ruta
        self.lineas = ""
```

### MÉTODOS

#### leer\_archivo()

El método leer\_archivo abre, en modo lectura, el documento con la dirección almacenada en el atributo ruta. Se recorre la información del archivo línea por línea, y esta se guarda en el atributo lineas. Al finalizar, se retorna el valor de la variable lineas.

```
def leer_archivo(self):
    archivo = open(self.ruta, "r")
    for linea in archivo.readlines():
        self.lineas += linea
    archivo.close()
    return self.lineas
```

## CLASE FUNCION

La clase Funcion es utilizada para crear objetos de la misma, que contengan las características de las funciones leídas durante el análisis.

### ATRIBUTOS

Los atributos de la clase Funcion representan las propiedades a almacenar de las funciones leídas para su posterior traducción a una sentencia de MongoDB.

```
class Funcion():

    def __init__(self, tipo, nombre):
        self.tipo = tipo
        self.nombre = nombre
        self.parametros = None
        self.id = None
        self.json = None
```

### MÉTODOS

#### Setters

Se tienen métodos set establecer el valor de parametros, id y json.

```
def set_parametros(self, parametros):
    self.parametros = parametros

def set_id(self):
    if self.parametros[0] != None:
        self.id = self.parametros[0]

def set_json(self):
    if self.parametros[1] != None:
        self.json = self.parametros[1]
```

## CLASE ESCRITORTRADUCCION

La clase EscritorTraduccion tiene el fin de escribir un archivo .txt con la traducción a sentencias MongoDB de cada función leída del archivo analizado.

### ATRIBUTOS

La clase EscritorTraduccion cuenta con el atributo de lista\_funciones (que contiene objetos de tipo Funcion, con información relevante de las funciones analizadas).

```
class EscritorTraduccion():

    def __init__(self, lista_funciones):
        self.lista_funciones = lista_funciones
```

### MÉTODOS

#### escribir\_funcion()

El método escribir\_funcion se encargará de generar un archivo de texto plano, llamado “TRADUCCION\_202010003” en la carpeta ‘resultados’, con la traducción de las funciones guardadas durante el análisis. En una variable llamada texto se almacenarán las oraciones traducidas de cada una de las funciones del arreglo. Esto se llevará a cabo por medio de un ciclo for, el cual recorrerá los objetos tipo Funcion de la lista\_funciones uno por uno, con el fin de poder establecer y acceder a las propiedades de los mismos.

```
def escribir_funcion(self):
    archivo = open("resultados\TRADUCCION_202010003.txt", "w+")
    texto = ""
    for funcion in self.lista_funciones:
        if funcion.parametros != None:
            funcion.set_id()
            funcion.set_json()
        if funcion.tipo == "CrearBD":
            texto += "use(\"" + str(funcion.nombre) + "\");\n"
        elif funcion.tipo == "EliminarBD":
            texto += "db.dropDatabase();\n"
        elif funcion.tipo == "CrearColeccion":
            texto += "db.createCollection(\"" + str(funcion.id) + "\");\n"
        elif funcion.tipo == "EliminarColeccion":
            texto += "db." + str(funcion.id) + ".drop();\n"
        elif funcion.tipo == "InsertarUnico":
            texto += "db." + str(funcion.id) + ".insertOne(" + str(funcion.json) + ");\n"
```

```
elif funcion.tipo == "ActualizarUnico":  
    texto += "db." + str(funcion.id) + ".updateOne(" + str(funcion.json) + ");\n"  
elif funcion.tipo == "EliminarUnico":  
    texto += "db." + str(funcion.id) + ".deleteOne(" + str(funcion.json) + ");\n"  
elif funcion.tipo == "BuscarTodo":  
    texto += "db." + str(funcion.id) + ".find();\n"  
elif funcion.tipo == "BuscarUnico":  
    texto += "db." + str(funcion.id) + ".findOne();\n"  
archivo.write(texto)  
archivo.close()
```

## CLASE ERROR

La clase Error es utilizada para crear objetos de la misma, que contengan las características de los errores generados durante el análisis.

### ATRIBUTOS

Los atributos de la clase Error representan las características a mostrar de los errores generados en el análisis del archivo, en donde está el tipo de error (“Léxico” o “Sintáctico”), el token que se esperaba, la descripción, la fila y columna en la que se encuentra el error.

```
class Error:  
    def __init__(self, tipo, token Esperado, descripcion, fila, columna):  
        self.tipo = tipo  
        self.token Esperado = token Esperado  
        self.descripcion = descripcion  
        self.fila = fila  
        self.columna = columna
```

### MÉTODOS

#### Getters

Se tienen métodos get para cada uno de los atributos, con los cuales se retorna el valor de los mismos.

```
def get_tipo(self):  
    return self.tipo  
  
def get_token Esperado(self):  
    return self.token Esperado  
  
def get_descripcion(self):  
    return self.descripcion  
  
def get_fila(self):  
    return self.fila  
  
def get_columna(self):  
    return self.columna
```

## CLASE ESCRITORERRORES

La clase EscritorErrores tiene el fin de escribir un archivo .html con la información de todos los errores generados durante el análisis y presentarlos en una tabla.

### ATRIBUTOS

La clase EscritorErrores cuenta con un único atributo, el cual es una lista llamada lista\_errores, la cual almacenará objetos de tipo Error, que representan los cada error generado en el análisis.

```
class EscritorErrores():

    def __init__(self, lista_errores):
        self.lista_errores = lista_errores
```

### MÉTODOS

#### escribir\_errores()

El método escribir\_errores se encargará de generar un archivo .html, llamado “ERRORES\_202010003” en la carpeta ‘resultados’, con la información de los errores generados en el análisis. En una variable llamada texto\_escribir se guardará toda la estructura y datos para crear la tabla con la información de los errores; esto se ejecutará con un ciclo for, el cual recorrerá los objetos tipo Error de la lista de errores.

```
def escribir_errores(self):
    archivo = open("resultados\ERRORES_202010003.html", "w+")
    numero_errores = 1
    texto_escribir = "<!DOCTYPE html>\n"
    texto_escribir += "<html lang=\"es\">\n"
    texto_escribir += "<head>\n"
    texto_escribir += "\t<title>Errores</title>\n"
    texto_escribir += "\t<meta charset=\"utf-8\">\n"
    texto_escribir += "</head>\n"
    texto_escribir += "<body>\n"
    texto_escribir += "\t<h1>TABLA DE ERRORES</h1>\n"
    texto_escribir += "\t<br>\n"
    texto_escribir += "\t<table border=\"1\">\n"
```

```

    texto_escribir += "\t<tr>\n"
    texto_escribir += "\t\t<td>No.</td>\n"
    texto_escribir += "\t\t<td>Tipo</td>\n"
    texto_escribir += "\t\t<td>Token esperado</td>\n"
    texto_escribir += "\t\t<td>Descripcion</td>\n"
    texto_escribir += "\t\t<td>Fila</td>\n"
    texto_escribir += "\t\t<td>Columna</td>\n"
    texto_escribir += "\t</tr>\n"

    for error in self.lista_errores:
        texto_escribir += "\t<tr>\n"
        texto_escribir += "\t\t<td>" + str(numero_errores) + "</td>\n"
        texto_escribir += "\t\t<td>" + str(error.get_tipo()) + "</td>\n"
        texto_escribir += "\t\t<td>" + str(error.get_token Esperado()) + "</td>\n"
        texto_escribir += "\t\t<td>" + str(error.get_descripcion()) + "</td>\n"
        texto_escribir += "\t\t<td>" + str(error.get_fila()) + "</td>\n"
        texto_escribir += "\t\t<td>" + str(error.get_columna()) + "</td>\n"
        texto_escribir += "\t</tr>\n"
        numero_errores += 1
    texto_escribir += "</table>\n"
    texto_escribir += "</body>\n"
    texto_escribir += "</html>\n"
    archivo.write(texto_escribir)
    archivo.close()

```

## CLASE ESCRITORTOKENS

La clase EscritorTokens tiene el fin de escribir un archivo .html con la información de todos los tokens encontrados durante el análisis y presentarlos en una tabla.

### ATRIBUTOS

La clase EscritorTokens cuenta con un único atributo, el cual es una lista llamada lista\_funciones, la cual guarda objetos de tipo Funcion, que representan cada una de las funciones analizadas correctamente del archivo.

```
class EscritorTokens():

    def __init__(self, lista_funciones):
        self.lista_funciones = lista_funciones
```

### MÉTODOS

#### escribir\_tokens()

El método escribir\_tokens se encargará de generar un archivo .html, llamado “TOKENS\_202010003” en la carpeta ‘resultados’, con la información de los tokens encontrados durante el análisis. En una variable llamada texto\_escribir se guardará toda la estructura y datos para crear la tabla con la información de los tokens.

```
def escribir_tokens(self):
    archivo = open("resultados\TOKENS_202010003.html", "w+")
    texto_escribir = "<!DOCTYPE html>\n"
    texto_escribir += "<html lang=\"es\">\n"
    texto_escribir += "<head>\n"
    texto_escribir += "\t<title>Tokens</title>\n"
    texto_escribir += "\t<meta charset=\"utf-8\">\n"
    texto_escribir += "</head>\n"
    texto_escribir += "<body>\n"
    texto_escribir += "\t<h1>TABLA DE TOKENS</h1>\n"
    texto_escribir += "\t<br>\n"
    texto_escribir += "\t<table border='1'>\n"
    texto_escribir += "\t<tr>\n"
    texto_escribir += "\t\t<td>No.</td>\n"
    texto_escribir += "\t\t<td>Token</td>\n"
    texto_escribir += "\t\t<td>Lexema</td>\n"
    texto_escribir += "\t</tr>\n"

    for funcion in self.lista_funciones:
        if funcion.parametros != None:
            funcion.set_id()
            funcion.set_json()
        numero, token = self.comprobar_tokens(funcion.tipo)
        texto_escribir += "\t<tr>\n"
        texto_escribir += "\t\t<td>" + str(numero) + "</td>\n"
        texto_escribir += "\t\t<td>" + str(token) + "</td>\n"
        texto_escribir += "\t\t<td>" + str(funcion.tipo) + "</td>\n"
        texto_escribir += "\t</tr>\n"
        texto_escribir += self.agregar_tokens(funcion)
        texto_escribir += "</table>\n"
        texto_escribir += "</body>\n"
        texto_escribir += "</html>\n"
    archivo.write(texto_escribir)
    archivo.close()
```

## agregar\_json()

El método agregar\_json se encargará de agregar los tokens, en la estructura de la tabla, que están relacionados al JSON, que tendrían la forma: { . “ . ID . ” . . . “ . ID . ” . . . “ . ID . ” )\* . } antes del \$set.

```
def agregar_json(self, funcion):
    texto_datos = ""
    texto_id = []
    texto = ""
    texto_escribir = ""
    if funcion.json != None:
        texto = funcion.json.replace("{", " ")
        texto = texto.replace("}", " ")
        texto_datos += texto
        texto_id = texto_datos.split("\n")
        texto_id.remove("")
        if len(texto_id) > 1:
            texto_id.pop()
        try:
            posicion = texto_id.index(",\$set:")
        except:
            posicion = -1
        if posicion != -1:
            texto_id.pop(posicion + 3)
            texto_id.pop(posicion + 2)
            texto_id.pop(posicion + 1)
            texto_id.pop(posicion)

    for valor in texto_id:
        if valor != ":" and valor != ",":
            texto_escribir += "\t<tr>\n"
            texto_escribir += "\t\t<td>17</td>\n"
            texto_escribir += "\t\t<td>Tk_ComillaD</td>\n"
            texto_escribir += "\t\t<td></td>\n"
            texto_escribir += "\t</tr>\n"
            texto_escribir += "\t<tr>\n"
            texto_escribir += "\t\t<td>\n"
            texto_escribir += "\t\t<td>11</td>\n"
            texto_escribir += "\t\t<td>Tk_ID</td>\n"
            texto_escribir += "\t\t<td>" + str(valor) + "</td>\n"
            texto_escribir += "\t</tr>\n"
            texto_escribir += "\t<tr>\n"
            texto_escribir += "\t\t<td>\n"
            texto_escribir += "\t\t<td>17</td>\n"
            texto_escribir += "\t\t<td>Tk_ComillaD</td>\n"
            texto_escribir += "\t\t<td></td>\n"
            texto_escribir += "\t</tr>\n"
            texto_escribir += "\t<tr>\n"

        elif valor == ":":
            texto_escribir += "\t<tr>\n"
            texto_escribir += "\t\t<td>18</td>\n"
            texto_escribir += "\t\t<td>Tk_Coma</td>\n"
            texto_escribir += "\t\t<td>,</td>\n"
            texto_escribir += "\t</tr>\n"
            texto_escribir += "\t<tr>\n"
        elif valor == ",":
            texto_escribir += "\t<tr>\n"
            texto_escribir += "\t\t<td>21</td>\n"
            texto_escribir += "\t\t<td>Tk_DosPts</td>\n"
            texto_escribir += "\t\t<td>:</td>\n"
            texto_escribir += "\t</tr>\n"
            texto_escribir += "\t<tr>\n"

    return texto_escribir
```

## agregar\_set()

El método agregar\_set se encargará de agregar los tokens, en la estructura de la tabla, que están relacionados al JSON, que tendrían la forma: “ . ID . ” . . . “ . ID . ” después del \$set.

```

def agregar_set(self, funcion):
    texto_datos = ""
    texto_id = []
    arreglo = []
    texto = ""
    texto_escribir = ""
    if funcion.json != None:
        texto = funcion.json.replace("{", "")
        texto = texto.replace("}", "")
        texto_datos += texto
        texto_id = texto_datos.split("\\")

        texto_id.remove("\\")
        if len(texto_id) > 1:
            texto_id.pop()
        try:
            posicion = texto_id.index("$set:")
        except:
            posicion = -1
        if posicion != -1:
            arreglo.append(texto_id[posicion + 1])
            arreglo.append(texto_id[posicion + 2])
            arreglo.append(texto_id[posicion + 3])
    for valor in arreglo:
        if valor != ":":
            texto_escribir += "\t<tr>\n"
            texto_escribir += "\t\t<td>17</td>\n"
            texto_escribir += "\t\t<td>Tk_ComillaB</td>\n"
            texto_escribir += "\t\t<td>\"/<td>\n"
            texto_escribir += "\t</tr>\n"
            texto_escribir += "\t<tr>\n"
            texto_escribir += "\t\t<td>\n"
            texto_escribir += "\t\t<td>11</td>\n"
            texto_escribir += "\t\t<td>Tk_ID</td>\n"
            texto_escribir += "\t\t<td>" + str(valor) + "</td>\n"
            texto_escribir += "\t</tr>\n"
            texto_escribir += "\t<tr>\n"
            texto_escribir += "\t\t<td>17</td>\n"
            texto_escribir += "\t\t<td>Tk_ComillaD</td>\n"
            texto_escribir += "\t\t<td>\"/<td>\n"
            texto_escribir += "\t</tr>\n"
            texto_escribir += "\t<tr>\n"
        elif valor == ":":
            texto_escribir += "\t<tr>\n"
            texto_escribir += "\t\t<td>21</td>\n"
            texto_escribir += "\t\t<td>Tk_DosPts</td>\n"
            texto_escribir += "\t\t<td>:</td>\n"
            texto_escribir += "\t</tr>\n"
            texto_escribir += "\t<tr>\n"
    return texto_escribir

```

## agregar\_tokens()

El método agregar\_tokens se encargará de agregar los tokens, en la estructura de la tabla, que están relacionados al JSON, que tendrían la forma: { . “ . ID . ” . : . “ . ID . ” ( . , . “ . ID . ” . : . “ . ID . ” ) \* . } después del \$set.

```

def agregar_tokens(self, funcion):
    texto = ""
    texto += "\t<tr>\n"
    texto += "\t\t<td>11</td>\n"
    texto += "\t\t<td>Tk_ID</td>\n"
    texto += "\t\t<td>" + str(funcion.nombre) + "</td>\n"
    texto += "\t</tr>\n"
    texto += "\t<tr>\n"
    texto += "\t\t<td>12</td>\n"
    texto += "\t\t<td>Tk_Igual</td>\n"
    texto += "\t\t<td>=</td>\n"
    texto += "\t</tr>\n"
    texto += "\t<tr>\n"
    texto += "\t\t<td>13</td>\n"
    texto += "\t\t<td>Tk_Nueva</td>\n"
    texto += "\t\t<td>nueva</td>\n"
    texto += "\t</tr>\n"
    texto += "\t<tr>\n"
    texto += "\t\t<td>14</td>\n"
    texto += "\t\t<td>Tk_ParA</td>\n"
    texto += "\t\t<td>(</td>\n"
    texto += "\t</tr>\n"
    if funcion.id != None:
        texto += "\t<tr>\n"
        texto += "\t\t<td>17</td>\n"
        texto += "\t\t<td>Tk_ComillaD</td>\n"
        texto += "\t\t<td>\"/<td>\n"
        texto += "\t</tr>\n"
        texto += "\t<tr>\n"
        texto += "\t\t<td>\n"
        texto += "\t\t<td>11</td>\n"
        texto += "\t\t<td>Tk_ID</td>\n"
        texto += "\t\t<td>" + str(funcion.id) + "</td>\n"
        texto += "\t</tr>\n"
        texto += "\t<tr>\n"
        texto += "\t\t<td>17</td>\n"
        texto += "\t\t<td>Tk_ComillaD</td>\n"
        texto += "\t\t<td>\"/<td>\n"
        texto += "\t</tr>\n"
    if funcion.tipo == "InsertarUnico" or \
        funcion.tipo == "ActualizarUnico" or \
        funcion.tipo == "EliminarUnico":

```

```

texto += "\t<tr>\n"
texto += "\t\t<td>18</td>\n"
texto += "\t\t<td>Tk_Coma</td>\n"
texto += "\t\t<td>,</td>\n"
texto += "\t</tr>\n"
texto += "\t<tr>\n"
texto += "\t\t<td>17</td>\n"
texto += "\t\t<td>Tk_ComillaD</td>\n"
texto += "\t</tr>\n"
texto += "\t<tr>\n"
texto += "\t\t<td>19</td>\n"
texto += "\t\t<td>Tk_LlaveA</td>\n"
texto += "\t\t<td>,</td>\n"
texto += "\t</tr>\n"
texto += "\t<tr>\n"
texto += "\t\t<td>19</td>\n"
texto += "\t\t<td>Tk_Set</td>\n"
texto += "\t\t<td>$set</td>\n"
texto += "\t</tr>\n"
texto += "\t<tr>\n"
texto += "\t\t<td>20</td>\n"
texto += "\t\t<td>Tk_DosPts</td>\n"
texto += "\t\t<td>:</td>\n"
texto += "\t</tr>\n"
texto += "\t<tr>\n"

if funcion.tipo == "ActualizarUnico":
    texto += "\t<tr>\n"
    texto += "\t\t<td>18</td>\n"
    texto += "\t\t<td>Tk_Coma</td>\n"
    texto += "\t\t<td>,</td>\n"
    texto += "\t</tr>\n"
    texto += "\t<tr>\n"
    texto += "\t\t<td>19</td>\n"
    texto += "\t\t<td>Tk_LlaveA</td>\n"
    texto += "\t\t<td>,</td>\n"
    texto += "\t</tr>\n"
    texto += "\t<tr>\n"
    texto += "\t\t<td>20</td>\n"
    texto += "\t\t<td>Tk_LlaveC</td>\n"
    texto += "\t\t<td>,</td>\n"
    texto += "\t</tr>\n"
    texto += "\t<tr>\n"
    texto += "\t\t<td>20</td>\n"
    texto += "\t\t<td>Tk_LlaveC</td>\n"
    texto += "\t\t<td>,</td>\n"
    texto += "\t</tr>\n"
    texto += "\t<tr>\n"
    texto += "\t\t<td>17</td>\n"
    texto += "\t\t<td>Tk_ComillaD</td>\n"
    texto += "\t\t<td>,</td>\n"
    texto += "\t</tr>\n"

def comprobar_tokens():
    texto += "\t<tr>\n"
    texto += "\t\t<td>15</td>\n"
    texto += "\t\t<td>Tk_ParC</td>\n"
    texto += "\t\t<td>,</td>\n"
    texto += "\t</tr>\n"
    texto += "\t<tr>\n"
    texto += "\t\t<td>16</td>\n"
    texto += "\t\t<td>Tk_PtoComa</td>\n"
    texto += "\t\t<td>,</td>\n"
    texto += "\t</tr>\n"
    return texto

```

## comprobar\_tokens()

El método comprobar\_tokens se encarga de verificar cuál es la función que está guardada y devolver el número y nombre de token correspondiente a la misma.

```

def comprobar_tokens(self, lexema):
    numero = None
    nombre_token = ""
    if lexema == "CrearBD":
        numero = 2
        nombre_token = "Tk_CrearBD"
    elif lexema == "EliminarBD":
        numero = 3
        nombre_token = "Tk_EliminarBD"
    elif lexema == "CrearColeccion":
        numero = 4
        nombre_token = "Tk_CrearC"
    elif lexema == "EliminarColeccion":
        numero = 5
        nombre_token = "Tk_EliminarC"
    elif lexema == "InsertarUnico":
        numero = 6
        nombre_token = "Tk_InsertarU"
    elif lexema == "ActualizarUnico":
        numero = 7
        nombre_token = "Tk_ActualizarU"
    elif lexema == "EliminarUnico":
        numero = 8
        nombre_token = "Tk_EliminarU"
    elif lexema == "BuscarTodo":
        numero = 9
        nombre_token = "Tk_BuscarT"
    elif lexema == "BuscarUnico":
        numero = 10
        nombre_token = "Tk_BuscarU"
    return numero, nombre_token

```

## CLASE ANALIZADOR

La clase Analizador tiene el fin de realizar el análisis léxico y sintáctico del contenido de un archivo de texto plano; y con el resultado del análisis, realizar la traducción de distintas sentencias a un lenguaje para MongoDB, y generar documentos donde se describen los tokens y errores encontrados. Para ello, se importan las siguientes clases:

```
from lectura_escritura.funcion import Funcion
from lectura_escritura.escriotor_traducion import EscritorTraducion
from lectura_escritura.escriotor_errores import EscritorErrores
from lectura_escritura.escriotor_tokens import EscritorTokens
from lectura_escritura.error import Error
```

## ATRIBUTOS

La clase Analizador cuenta con varios atributos, en donde se encuentra lineas, que es el contenido leído del archivo; fila, columna e index, que determinan la posición de un carácter; funciones, con los diferentes tipos de función que se pueden tener; funcion, que almacenará el valor de la función leída al principio; funcion\_leida que será un objeto de tipo Funcion; lista\_funciones, que contiene objetos tipo Funcion con los datos de las funciones analizadas correctamente; y lista\_errores, que almacena objetos tipo Error con la información de los errores generados

```
class Analizador():

    def __init__(self, lineas):
        self.lineas = lineas
        self.fila = 0
        self.columna = 0
        self.index = 0
        self.funciones = ["CrearBD", "EliminarBD", "CrearColeccion", "EliminarColeccion",
                          "InsertarUnico", "ActualizarUnico", "EliminarUnico",
                          "BuscarTodo", "BuscarUnico"]
        self.funcion = ""
        self.funcion_leida = None
        self.lista_funciones = []
        self.lista_errores = []
```

## MÉTODOS

### verificar\_token()

El método verificar\_token recibe el token a buscar y los estados actual y siguiente. Cuando encuentre el primer carácter distinto de espacio, a la variable texto se le asignará el valor que retorne el método juntar\_caracteres(). Luego se condiciona el valor que devuelva el método analizar\_coincidencia(), donde se manda el texto y token; si es verdadero, se pasa al estado siguiente, y en el caso contrario, se manda a llamar al método guardar\_error() y se retorna “ERROR” por no coincidir el token con la palabra encontrada en el archivo.

```
def verificar_token(self, token, estado_actual, estado_siguiente):
    if self.lineas[self.index] != " ":
        # Palabra encontrada en el archivo
        texto = self.juntar_caracteres(self.index, len(token))
        # Si la palabra coincide con el token esperado
        if self.analizar_coincidencia(token, texto):
            self.index += len(token) - 1
            self.columna += len(token) - 1
            return estado_siguiente
        else:
            self.guardar_error("Lexico", token, texto)
            return "ERROR"
    else:
        return estado_actual
```

### juntar\_caracteres()

El método juntar\_caracteres recibe un inicio y fin para concatenar los caracteres del archivo leído posicionados dentro de ese rango. Se retorna la palabra unificada.

```
def juntar_caracteres(self, inicio, fin):
    try:
        token_unificado = ""
        for caracter in range(inicio, inicio + fin):
            token_unificado += self.lineas[caracter]
        return token_unificado
    except:
        return None
```

## anализar\_coincidencia()

El método `anализар_коинциденция` recibe el token y el texto a comparar. Comprueba carácter a carácter que el texto sea igual al token esperado; si no coincide en algún carácter, retorna False; y si todos los caracteres coinciden, devuelve True.

```
def analizar_coincidencia(self, token, texto):
    try:
        posicion = 0
        token_temporal = ""
        for caracter in texto:
            if str(caracter) == str(token[posicion]):
                token_temporal += caracter
                posicion += 1
            else:
                return False
        print(f'***** ENCONTRE - {token_temporal} *****')
        return True
    except:
        return False
```

## verificar\_id()

El método `verificar_id` se ejecuta hasta que se cumpla con alguno de los casos de salida; asimismo cuando `estado_actual` sea igual a “ERROR”. Se comprueba el estado actual y se manda a verificar el token determinado, dependiendo el estado (carácter distinto a un espacio, salto de línea y comilla doble).

```
def verificar_id(self, estado_siguiente):
    estado_actual = "I0"
    estado_siguiente = estado_siguiente
    id = ""
    while self.lineas[self.index] != "":
        # Casos en los cuales se debe salir de la verificación de id
        if str(self.lineas[self.index]) == " " and id != "":
            self.index -= 1
            return [estado_siguiente, id]

        # Casos en los cuales se debe salir de la verificación de id
        elif str(self.lineas[self.index]) == "\n" and id != "":
            self.fila += 1
            self.columna = 0
            self.index -= 1
            return [estado_siguiente, id]

        # Casos en los cuales se debe salir de la verificación de id
        elif str(self.lineas[self.index]) == "\"":
            self.index -= 1
            return [estado_siguiente, id]

        # I0 -> CARACTER I1
        elif estado_actual == "I0":
            if self.lineas[self.index] != " ":
                id += self.lineas[self.index]
                estado_actual = "I1"

        # I1 -> CARACTER I1
        elif estado_actual == "I1":
            if self.lineas[self.index] != " ":
                id += self.lineas[self.index]
                estado_actual = "I1"
            else:
                estado_actual = "ERROR"

        if estado_actual == "ERROR":
            return ["ERROR", None]

        if self.index < len(self.lineas) - 1:
            self.index += 1
            self.columna += 1
        else:
            break
```

## verificar\_cadena()

El método verificar\_cadena se ejecuta hasta que se cumpla con alguno de los casos de salida; asimismo cuando estado\_actual sea igual a “ERROR”. Se comprueba el estado actual y se manda a verificar el token determinado, dependiendo el estado (carácter distinto a un salto de línea y comilla doble).

```
def verificar_cadena(self, estado_siguiente):
    estado_actual = "K0"
    estado_siguiente = estado_siguiente
    cadena = ""
    while self.lineas[self.index] != "":
        # Caso de salida de la verificación de cadena
        if str(self.lineas[self.index]) == "\"":
            self.index -= 1
            return [estado_siguiente, cadena]

        # K0 -> CARACTER K1
        elif estado_actual == "K0":
            cadena += self.lineas[self.index]
            estado_actual = "K1"

    # K1 -> CARACTER K1
    elif estado_actual == "K1":
        if self.lineas[self.index] != "\n":
            cadena += self.lineas[self.index]
            estado_actual = "K1"
        else:
            self.fila += 1
            self.columna = 0
            estado_actual = "ERROR"

    if estado_actual == "ERROR":
        return ["ERROR", None]

    if self.index < len(self.lineas) - 1:
        self.index += 1
        self.columna += 1
    else:
        break
```

## comentario\_linea()

El método comentario\_linea se ejecuta hasta que se cumpla con alguno de los casos de salida; asimismo cuando estado\_actual sea igual a “ERROR”. Se comprueba el estado actual y se manda a verificar el token determinado, dependiendo el estado. Se sigue la estructura: - . - . - . (CARACTER)\*

```
def comentario_linea(self, estado_siguiente):
    estado_actual = "C0"
    estado_siguiente = estado_siguiente
    comentario = ""
    while self.lineas[self.index] != "":
        # Casos en los cuales se debe salir del comentario de linea
        if str(self.lineas[self.index]) == "\n":
            self.fila += 1
            self.columna = 0
            self.index -= 1
            return [estado_siguiente, comentario]

        # C0 -> - C1
        elif estado_actual == "C0":
            estado_actual = self.verificar_token("-", "C0", "C1")

        # C1 -> - C2
        elif estado_actual == "C1":
            estado_actual = self.verificar_token("-", "C1", "C2")

        # C2 -> - C3
        elif estado_actual == "C2":
            estado_actual = self.verificar_token("-", "C2", "C3")

        # C3 -> CARACTER C3
        # C3 -> C4
        elif estado_actual == "C3":
            if self.lineas[self.index + 1] != "\n":
                comentario += self.lineas[self.index]
                estado_actual = "C3"
            else:
                comentario += self.lineas[self.index]
                print(f'***** ENCONTRE - {comentario} *****')
                estado_actual = "C4"

        # C4 -> ESTADO DE ACEPTACIÓN
        elif estado_actual == "C4":
            return [estado_siguiente, comentario]
```

```

    if estado_actual == "ERROR":
        return ["ERROR", None]

    if self.index < len(self.lineas) - 1:
        self.index += 1
        self.columna += 1
    else:
        break

```

## comentario\_varias\_lineas()

El método comentario\_varias\_lineas se ejecuta hasta que estado\_actual sea igual a “ERROR” o se llegue al estado de aceptación. Se comprueba el estado actual y se manda a verificar el token determinado, dependiendo el estado. Se sigue la estructura: / . \* . (CARACTER)\* . \* . /

```

def comentario_varias_lineas(self, estado_siguiente):
    estado_actual = "Q0"
    estado_siguiente = estado_siguiente
    comentario = ""
    while self.lineas[self.index] != "":
        if str(self.lineas[self.index]) == "\n":
            self.fila += 1
            self.columna = 0

        # Q0 -> / Q1
        if estado_actual == "Q0":
            estado_actual = self.verificar_token("/", "Q0", "Q1")

        # Q1 -> * Q2
        elif estado_actual == "Q1":
            estado_actual = self.verificar_token("*", "Q1", "Q2")

        # Q2 -> CARACTER Q2
        # Q2 -> * Q3
        elif estado_actual == "Q2":
            if self.lineas[self.index] != "*" and self.lineas[self.index + 1] != "/":
                comentario += self.lineas[self.index]
                estado_actual = "Q2"
            else:
                print(f'***** ENCONTRE - {comentario} *****')
                estado_actual = self.verificar_token("*", "Q2", "Q3")

        # Q3 -> / Q4
        elif estado_actual == "Q3":
            estado_actual = self.verificar_token("/", "Q3", "Q4")

        # Q4 -> ESTADO DE ACEPTACIÓN
        elif estado_actual == "Q4":
            return [estado_siguiente, comentario]

        if estado_actual == "ERROR":
            return ["ERROR", None]

        if self.index < len(self.lineas) - 1:
            self.index += 1
            self.columna += 1
        else:
            break

```

## json()

El método json se ejecuta hasta que estado\_actual sea igual a “ERROR” o se llegue al estado de aceptación. Se comprueba el estado actual y se manda a verificar el token determinado, dependiendo el estado. Se sigue la estructura: { . DATOS . ( , . DATOS )\* . } . ( , . { . \$set . : . { . DATOS . } . } )?

```

def json(self, estado_siguiente):
    estado_actual = "J0"
    estado_siguiente = estado_siguiente
    json = []
    set = ""
    datos = ""
    while self.lineas[self.index] != "":
        if str(self.lineas[self.index]) == "\n":
            self.fila += 1
            self.columna = 0

    # J0 -> { J1
    elif estado_actual == "J0":
        estado_actual = self.verificar_token("{", "J0", "J1")

# J1 -> DATOS J2
elif estado_actual == "J1":
    arreglo = self.datos("J2")
    if arreglo[0] == "ERROR":
        estado_actual = "ERROR"
    elif arreglo[0] == "J2":
        print(f'***** ENCONTRE - {arreglo[1]} *****')
        json.append(str(arreglo[1]))
        estado_actual = "J2"

# J2 -> , J3
# J2 -> } J4
elif estado_actual == "J2":
    if self.lineas[self.index] != " " or self.lineas[self.index] != "\n":
        if self.lineas[self.index] == ",":
            estado_actual = self.verificar_token(",", "J2", "J3")
        else:
            estado_actual = "J4"

# J3 -> DATOS J2
elif estado_actual == "J3":
    arreglo = self.datos("J2")
    if arreglo[0] == "ERROR":
        estado_actual = "ERROR"
    elif arreglo[0] == "J2":
        print(f'***** ENCONTRE - {arreglo[1]} *****')
        json.append(str(arreglo[1]))
        estado_actual = "J2"

# J4 -> , J5
# J4 -> J12
elif estado_actual == "J4":
    if self.lineas[self.index] != " " or self.lineas[self.index] != "\n":
        if self.lineas[self.index] == ":":
            estado_actual = self.verificar_token(":", "J4", "J5")
        else:
            estado_actual = "J12"

# J5 -> { J6
elif estado_actual == "J5":
    estado_actual = self.verificar_token("{", "J5", "J6")

# J6 -> $set J7
elif estado_actual == "J6":
    estado_actual = self.verificar_token("$set", "J6", "J7")

# J7 -> : J8
elif estado_actual == "J7":
    estado_actual = self.verificar_token(":", "J7", "J8")

# J8 -> { J9
elif estado_actual == "J8":
    estado_actual = self.verificar_token("{", "J8", "J9")

# J9 -> DATOS J10
elif estado_actual == "J9":
    arreglo = self.datos("J10")
    if arreglo[0] == "ERROR":
        estado_actual = "ERROR"
    elif arreglo[0] == "J10":
        print(f'***** ENCONTRE - {arreglo[1]} *****')
        set += arreglo[1]
        estado_actual = "J10"

# J10 -> } J11
elif estado_actual == "J10":
    estado_actual = self.verificar_token("}", "J10", "J11")

# J11 -> } J12
elif estado_actual == "J11":
    estado_actual = self.verificar_token("}", "J11", "J12")

# J12 -> ESTADO DE ACEPTACION
if estado_actual == "J12":
    datos += "{"
    contador = 1
    for dato in json:
        if contador != len(json):
            datos += dato + ","
        else:
            datos += dato
            contador += 1
    datos += "}"
    if set != "":
        datos += "{$set:{'"+ set + "'}}"
    return [estado_siguiente, datos]

if estado_actual == "ERROR":
    return ["ERROR", None]

if self.index < len(self.lineas) - 1:
    self.index += 1
    self.columna += 1
else:
    break

```

## datos()

El método datos se ejecuta hasta que estado\_actual sea igual a “ERROR” o se llegue al estado de aceptación. Se comprueba el estado actual y se manda a verificar el token determinado, dependiendo el estado. Se sigue la estructura: “ . ID . “ . : . ” . CADENA . “

```
def datos(self, estado_siguiente):
    estado_actual = "D0"
    estado_siguiente = estado_siguiente
    id = ""
    valor = ""
    dato = ""
    while self.lineas[self.index] != "":
        # D0 -> " D1
        if estado_actual == "D0":
            estado_actual = self.verificar_token("\", "D0", "D1")

        # D1 -> ID D2
        elif estado_actual == "D1":
            arreglo = self.verificar_id("D2")
            if arreglo[0] == "ERROR":
                estado_actual = "ERROR"
            elif arreglo[0] == "D2":
                print(f'***** ENCONTRE - {arreglo[1]} *****')
                id = arreglo[1]
                estado_actual = "D2"

        # D2 -> " D3
        elif estado_actual == "D2":
            estado_actual = self.verificar_token("\", "D2", "D3")

        # D3 -> : D4
        elif estado_actual == "D3":
            estado_actual = self.verificar_token(":", "D3", "D4")

        # D4 -> " D5
        elif estado_actual == "D4":
            estado_actual = self.verificar_token("\", "D4", "D5")

        # D5 -> CADENA D6
        elif estado_actual == "D5":
            arreglo = self.verificar_cadena("D6")
            if arreglo[0] == "ERROR":
                estado_actual = "ERROR"
            elif arreglo[0] == "D6":
                print(f'***** ENCONTRE - {arreglo[1]} *****')
                valor = arreglo[1]
                estado_actual = "D6"

        # D6 -> " D7
        elif estado_actual == "D6":
            estado_actual = self.verificar_token("\", "D6", "D7")

        # D7 -> ESTADO DE ACEPTACION
        if estado_actual == "D7":
            dato += "\\" + id + ":" + valor + "\\"
            return [estado_siguiente, dato]

        if estado_actual == "ERROR":
            return ["ERROR", None]

        if self.index < len(self.lineas) - 1:
            self.index += 1
            self.columna += 1
        else:
            break
```

## parametros()

El método parametros se ejecuta hasta que estado\_actual sea igual a “ERROR” o se llegue al estado de aceptación. Se comprueba el estado actual y se manda a verificar el token determinado, dependiendo el estado. Se sigue la estructura: “ . (ID)? . “ . ( , . ” . JSON . ”)?

```

def parametros(self, estado_siguiente):
    estado_actual = "P0"
    estado_siguiente = estado_siguiente
    id = ""
    json = ""
    while self.lineas[self.index] != "":
        if str(self.lineas[self.index]) == "\n":
            self.fila += 1
            self.columna = 0

    # P0 -> " P1
    elif estado_actual == "P0":
        estado_actual = self.verificar_token("", "P0", "P1")

    # P1 -> ID P2
    # P1 -> " P3
    elif estado_actual == "P1":
        if self.lineas[self.index] == "":
            estado_actual = self.verificar_token("", "P1", "P3")
        else:
            arreglo = self.verificar_id("P2")
            if arreglo[0] == "ERROR":
                estado_actual = "ERROR"
            elif arreglo[0] == "P2":
                id = arreglo[1]
                print(f'***** ENCONTRE - {arreglo[1]} *****')
            estado_actual = "P2"

    # P2 -> " P3
    elif estado_actual == "P2":
        estado_actual = self.verificar_token("", "P2", "P3")

# P3 -> , P4
# P3 -> P7
elif estado_actual == "P3":
    if self.funcion == "CrearColeccion" or self.funcion == "EliminarColeccion"\
        or self.funcion == "BuscarTodo" or self.funcion == "BuscarUnico":
        self.index -= 1
        estado_actual = "P7"
    elif self.lineas[self.index] == ",":
        estado_actual = self.verificar_token("", "P3", "P4")

# P4 -> " P5
elif estado_actual == "P4":
    estado_actual = self.verificar_token("", "P4", "P5")

# P5 -> JSON P6
elif estado_actual == "P5":
    arreglo = self.json("P6")
    if arreglo[0] == "ERROR":
        estado_actual = "ERROR"
    elif arreglo[0] == "P6":
        json = str(arreglo[1])
        print(f'***** ENCONTRE - {arreglo[1]} *****')
        if self.funcion == "EliminarUnico" or self.funcion == "InsertarUnico":
            self.index -= 1
        estado_actual = "P6"

```

```

# P6 -> P7
elif estado_actual == "P6":
    estado_actual = self.verificar_token("", "P6", "P7")

# P7 -> ESTADO DE ACEPTACION
if estado_actual == "P7":
    return [estado_siguiente, [id, json]]

if estado_actual == "ERROR":
    return ["ERROR", None]

if self.index < len(self.lineas) - 1:
    self.index += 1
    self.columna += 1
else:
    break

```

## compilar()

El método `compilar()` se ejecuta hasta leer por completo el archivo. Se comprueba el estado actual y se manda a verificar el token determinado, dependiendo el estado, siguiendo la estructura: FUNCION . ID . = . nueva . FUNCION . ( . (PARAMETROS)? . ) . ;

Al terminar la lectura del archivo, se escribirá crearán los objetos y se mandarán a llamar las funciones para generar los archivos de las traducciones, tokens y errores.

```

def compilar(self):
    estado_actual = "S0"
    while self.lineas[self.index] != "":
        # Si el carácter leído es un salto de linea,
        # entonces se aumenta 1 a fila y se reinicia la columna a 0
        if self.lineas[self.index] == "\n":
            self.fila += 1
            self.columna = 0

        # S0 -> FUNCION S1
        elif estado_actual == "S0":
            if self.lineas[self.index] == "-":
                arreglo = self.comentario_linea("S0")
                estado_actual = arreglo[0]
            elif self.lineas[self.index] == "/" and self.lineas[self.index + 1] == "*":
                arreglo = self.comentario_varias_lineas("S0")
                estado_actual = arreglo[0]
            else:
                estado_actual = "ERROR"
                for funcion in self.funciones:
                    estado_actual = self.verificar_token(funcion, "S0", "S1")
                    if estado_actual != "ERROR":
                        self.funcion = funcion
                        break
        # S1 -> ID S2
        elif estado_actual == "S1":
            arreglo = self.verificar_id("S2")
            if arreglo[0] == "ERROR":
                estado_actual = "ERROR"
            elif arreglo[0] == "S2":
                print(f'***** ENCONTRE - {arreglo[1]} *****')
                self.funcion_leida = Funcion(self.funcion, arreglo[1])
                estado_actual = "S2"

        # S2 -> S3
        elif estado_actual == "S2":
            estado_actual = self.verificar_token("=", "S2", "S3")

        # S3 -> nueva S4
        elif estado_actual == "S3":
            estado_actual = self.verificar_token("nueva", "S3", "S4")

        # S4 -> FUNCION S5
        elif estado_actual == "S4":
            estado_actual = "ERROR"
            for funcion in self.funciones:
                estado_actual = self.verificar_token(funcion, "S4", "S5")
                if estado_actual != "ERROR" and self.funcion == funcion:
                    break

# S5 -> ( S6
elif estado_actual == "S5":
    estado_actual = self.verificar_token("(", "S5", "S6")

# S6 -> PARAMETROS S7
# S6 -> ) S8
elif estado_actual == "S6":
    if self.funcion == "CrearBD" or self.funcion == "EliminarBD":
        estado_actual = self.verificar_token(")", "S6", "S8")
    else:
        arreglo = self.parametros("S7")
        if arreglo[0] == "ERROR":
            estado_actual = "ERROR"
        elif arreglo[0] == "S7":
            self.funcion_leida.set_parametros(arreglo[1])
            estado_actual = "S7"

# S7 -> ) S8
elif estado_actual == "S7":
    estado_actual = self.verificar_token(")", "S7", "S8")

# S8 -> ; S9
elif estado_actual == "S8":
    estado_actual = self.verificar_token(";", "S8", "S9")

# S9 -> ESTADO DE ACEPTACIÓN
if estado_actual == "S9":
    print("Cadena leída")
    self.lista_funciones.append(self.funcion_leida)
    estado_actual = "S0"

if estado_actual == "ERROR":
    estado_actual = "S0"

if self.index < len(self.lineas) - 1:
    self.index += 1
    self.columna += 1
else:
    print(self.lista_funciones)
    escribir_funciones = EscritorTraducción(self.lista_funciones)
    escribir_funciones.escribir_funcion()
    escribir_errores = EscritorErrores(self.lista_errores)
    escribir_errores.escribir_errores()
    escribir_tokens = EscritorTokens(self.lista_funciones)
    escribir_tokens.escribir_tokens()
    break

```

## guardar\_error()

El método guardar\_error crea un objeto de tipo Error, pasando los parámetros necesarios. Luego, se agrega este objeto a la lista lista\_errores.

```

def guardar_error(self, tipo, token, texto):
    tipo = tipo
    token Esperado = token
    descripción = texto.replace("\n", "")
    error = Error(tipo, token Esperado, descripción, self.fila, self.columna)
    self.lista_errores.append(error)

```

---

## CLASE MAIN

En la clase main, que es la principal, se importa la clase que contiene las instrucciones asociadas a la creación de la ventana; se crea un objeto de la misma y se muestra con el método mainloop().

```
from ventana import MenuPrincipal

ventana_principal = MenuPrincipal()
ventana_principal.mainloop()
```