Luis Rodrigo Morales Florián

202208521

Laboratorio Lenguajes Formales y de Programación

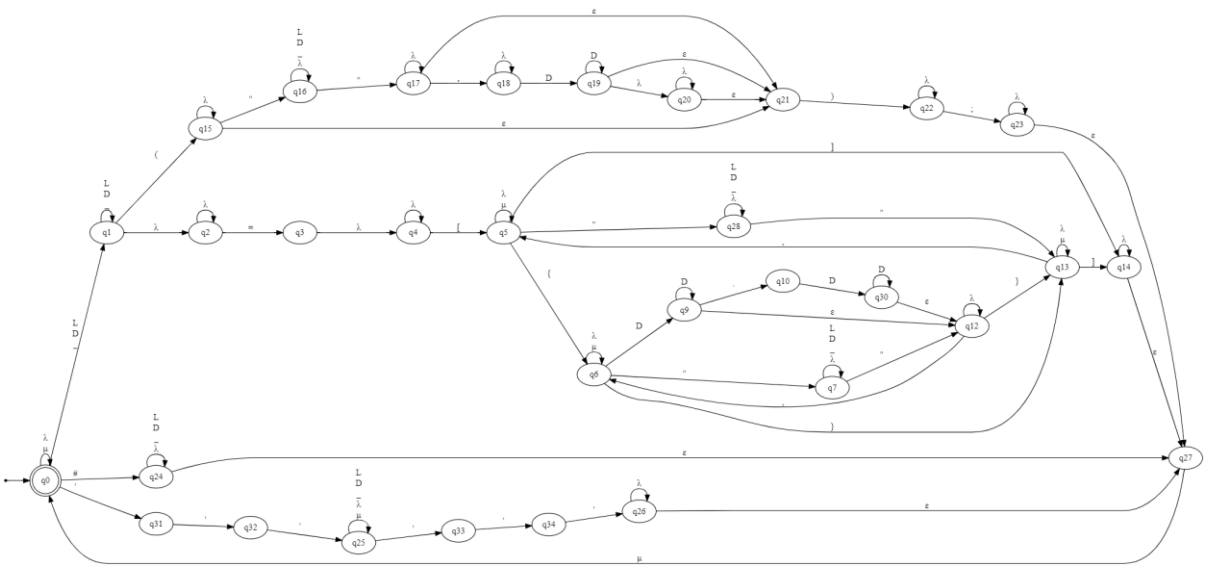
Sección B-

Manual Técnico – Proyecto #2

Contenido

Descripción del AFD

Gráfica:



Para ver la gráfica más clara en SVG, obtén el código de grafo.dot (que se ubica en la raíz del proyecto) y genéralo en Graphviz o visita el siguiente link:
https://res.cloudinary.com/dyin2bpxi/image/upload/v1698450860/2023/Lenguajes%20Formales/Proyecto%202/mubdq8pqr9xavg1f4old.svg

Descripción

$Q = \{q0, q1, q2, q3, q4, q5, q6, q7, q9, q10, q12, q13, q14,$

$q15, q16, q17, q18, q19, q20, q22, q23, q24, q25, q26, q27, q28, q30, q31, q32, q33, q34\}$

$\Sigma = \{L, D, \{, \}, [,], \mu, \varepsilon, \lambda, =, , (,); , "\}$

$Estado\ inicial = \{q0\}$

$Funciones\ de\ transición$:

- $\delta(q0, \lambda) = q0$
- $\delta(q0, \mu) = q0$
- $\delta(q0, L) = q1$
- $\delta(q0, D) = q1$
- $\delta(q0, \_) = q1$
- $\delta(q1, L) = q1$
- $\delta(q1, D) = q1$
- $\delta(q1, \_) = q1$

- $\delta(q1, \lambda) = q2$
- $\delta(q2, \lambda) = q2$
- $\delta(q2, =) = q3$
- $\delta(q3, \lambda) = q4$
- $\delta(q4, \lambda) = q4$
- $\delta(q4, [) = q5$
- $\delta(q5, \lambda) = q5$
- $\delta(q5, \mu) = q5$
- $\delta(q5, L) = q6$
- $\delta(q17, \lambda) = q17$
- $\delta(q17, ,) = q18$
- $\delta(q18, \lambda) = q18$
- $\delta(q18, D) = q19$
- $\delta(q19, D) = q19$
- $\delta(q19, \lambda) = q20$
- $\delta(q20, \lambda) = q20$
- $\delta(q20, \varepsilon) = q21$
- $\delta(q19, \varepsilon) = q21$
- $\delta(q15, \varepsilon\ ) = q1$
- $\delta(q17, \varepsilon) = q21$
- $\delta(q21,)) = q22$
- $\delta(q22, \lambda) = q22$
- $\delta(q22, ;) = q23$
- $\delta(q23, \lambda) = q23$
- $\delta(q0, \#) = q24$
- $\delta(q0, ") = q31$
- $\delta(q31, ") = q32$
- $\delta(q32, ") = q25$
- $\delta(q25, L) = q25$
- $\delta(q25, D) = q25$
- $\delta(q25, \_) = q25$
- $\delta(q25, \lambda) = q25$

- $\delta(q25, ") = q33$
- $\delta(q33, ") = q34$
- $\delta(q34, ") = q26$
- $\delta(q26, \lambda) = q26$
- $\delta(q26, \varepsilon) = q27$
- $\delta(q27, \mu) = q0$
- $\delta(q5, \{) = q6$
- $\delta(q6, \}) = q13$
- $\delta(q12, \}) = q13$
- $\delta(q13,]) = q14$
- $\delta(q12, ,) = q6$
- $\delta(q13, ,) = q5$
- $\delta(q5, \lambda) = q5$
- $\delta(q5, \mu) = q5$
- $\delta(q5, 23) = q6$
- $\delta(q13, \lambda) = q14$
- $\delta(q14, D) = q15$
- $\delta(q18, \lambda) = q18$
- $\delta(q18, \mu) = q29$
- $\delta(q19, D) = q19$
- $\delta(q19, \lambda) = q20$
- $\delta(q20, \lambda) = q20$
- $\delta(q20, \varepsilon) = q21$
- $\delta(q18, L) = q21$
- $\delta(q25, \lambda) = q23$
- $\delta(q11, \mu) = q26$
- $\delta(q5, 23) = q6$
- $\delta(q17, \lambda) = q17$
- $\delta(q17, \_) = q18$
- $\delta(q18, \lambda) = q18$
- $\delta(q30, D) = q19$
- $\delta(q31, L) = q19$

- $\delta(q32, \mu) = q33$
- $\delta(q32, \lambda) = q23$
- $\delta(q34, \varepsilon) = q20$

$Estado\ de\ aceptación = \{q0\}$

Descripción de Gramática Libre de Contexto

$\boldsymbol{V_t} = \{< \text{registros} >, < \text{reg}_{\text{array}} >, < \text{reg}_{\text{element}} >, < \text{elemento} >, < \text{claves} >, < \text{string}_{\text{array}} >, < \text{comentario} >, < \text{com}_{\text{contenido}} >, < \text{comentario}_{\text{multilinea}} >, < \text{inicio}_{\text{comentario}} >, < \text{fin}_{\text{comentario}} >, < \text{linea}_{\text{comentario}} >, < \text{llamada}_{\text{funcion}} >, < \text{valor} >, < \text{string} >, < \text{cadena} >, < \text{entero} >, < \text{digito} >, < \text{abecedario} >\}$

V_n = {“Registros”, “=”, “[”, “]”, “,”, “Claves”, “a”,”b”,”c”…,”z”, “0”,”1”, “2”…,”9”, “’”}

Producciones:

<S> :== <registros> | <claves> | <comentario> | <comentario_multilinea> | <llamada_funcion>

<registros> :== "Registros" "=" "[" <reg_array> "]"

<reg_array> - :== "{" <reg_element> "}" | <reg_array> "," "{" < reg_element> "}"

< reg_element> :== <elemento> | <element> "|" <elemento>

<elemento> :== "numero" | "string" | "float"

<claves> :== "Claves" "=" "[" <string_array> "]"

<string_array> :== <string> | <string_array> "," <string>

<comentario> :== "#" <com_contenido>

< com_contenido> :== < elemento> | <contenido> <elemento>

<comentario_multilinea> :== <inicio_comentario> <contenido> <fin_comentario>

<inicio_comentario> :== "’’’”

<fin_comentario> :== “’’”

<contenido> :== <linea_comentario> | <contenido> <linea_comentario>

<linea_comentario> :== <elemento> | <linea_comentario> <elemento>

<llamada_funcion> :== <identificador_funcion> "(" ")" ";"

| <identificador_funcion> "(" <valor> ")" ";"

| <identificador_funcion> "(" <valor> "," <valor> ")" ";"

<valor> :== <string> | <entero>

<string> :== "\"" <cadena> "\""

<cadena> :== <caracter> | <cadena> <caracter>

<caracter> :== <digito> | <abecedario> | “_” | “.”

<entero> :== <digito> | <digito> <entero>

<digito> :== "0" | "1" | ... | "9"

<abecedario> :== “a” | “b” | “c” … “z”

Símbolo inicial = <S>

Funcionamiento técnico

```
if __name__ == '__main__':
  ViewMethod().exec()
```

Ahí es donde empieza todo el programa. Se hace una instancia anónima de la clase ViewMethod() y se ejecuta.

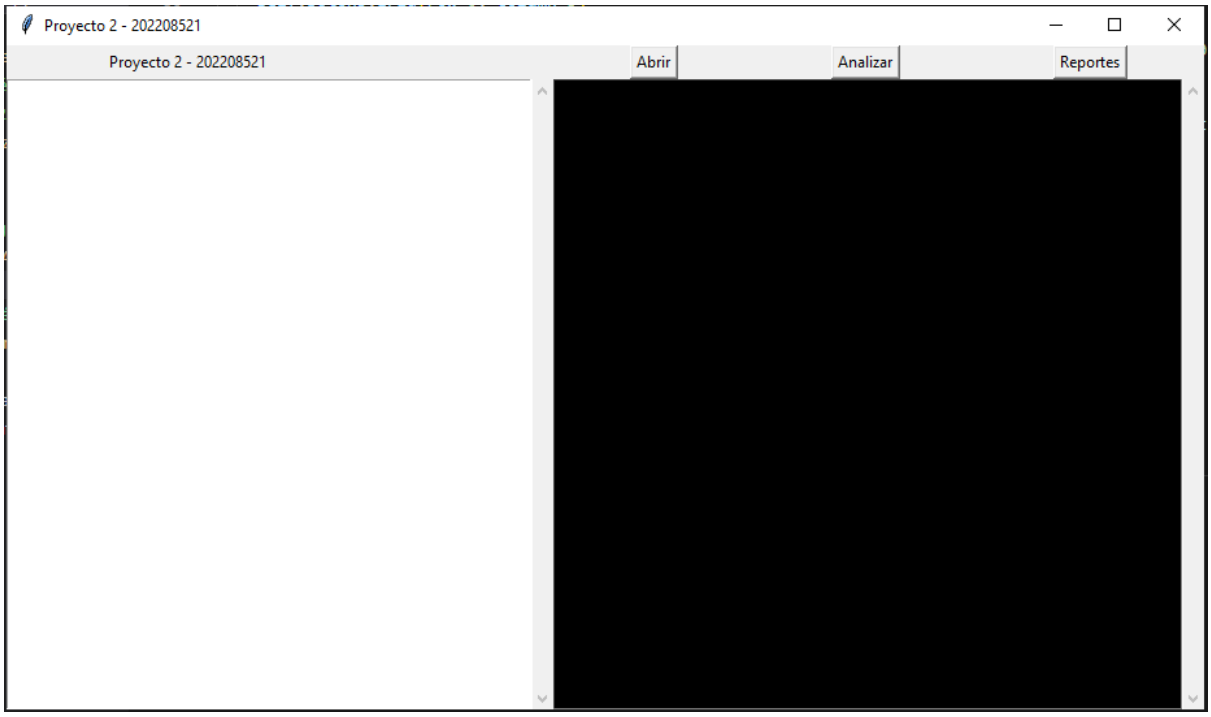
```python
class ViewMethod:
  def __init__(self) -> None:
    self.ventana = tk.Tk()
    self.ventana.title("Proyecto 2 - 202208521")

    self.titulo = tk.Label(self.ventana, text="Proyecto 2 - 202208521", padx=10)
    self.boton1 = tk.Button(self.ventana, text="Abrir", command=self.boton1_clicado)
    self.boton2 = tk.Button(self.ventana, text="Analizar", command=self.boton2_clicado)
    self.boton3 = tk.Button(self.ventana, text="Reportes", command=self.boton3_clicado)

    self.titulo.grid(row=0, column=0, columnspan=2)
    self.boton1.grid(row=0, column=3)
    self.boton2.grid(row=0, column=4)
    self.boton3.grid(row=0, column=5)

    self.texto_editable = scrolledtext.ScrolledText(self.ventana, wrap=tk.WORD, width=50, height=30)
    self.texto_editable.grid(row=1, column=0, columnspan=3)
```

Podemos ver una porción de código que representa nada más y nada menos que una vista de tkinter de Python. Se ejecuta y se tiene la siguiente vista:



En el lado izquierdo, se puede ingresar código con cierta estructura (previamente mencionada en el AFD y la gramática) y el lado derecho es una consola capaz de mostrar lo que nuestro programa interpretó y está programado para “reaccionar” a tal código. También es capaz de mostrar errores semánticos, sintácticos y léxicos. A continuación una muestra de código que se puede ingresar en el lado izquierdo:

```
Claves = ["codigo", "producto", "precio_compra", "precio_venta", "stock"]

Registros = [
  {1, "Barbacoa", 10.50, 20.00, 6},
```

```
  {2, "Salsa", 13.00, 16.00, 7},
  {3, "Mayonesa", 15.00, 18.00, 8},
  {4, "Mostaza", 14.00, 16.00, 4}
]

# Comentarios

'''
comentario multilinea
'''

imprimir("1. Reporte de");
imprimir("Abarroteria");

imprimirln("2. Reporte de");
imprimirln("Abarroteria");

conteo();

promedio("stock");

contarsi("stock", 0);

datos();

sumar("stock");

max("campo");

min("campo");

exportarReporte("titulo");
```

Al darle click al botón Analizar, se apela a la siguiente clase:

```python
class ADFAnalizer:
  def __init__(self, text_to_analize) -> None:
    self.there_is_an_error = False
    self.text_to_analize = text_to_analize
    self.acum_case = ""
    self.acum_stack = Stack()
    self.current_case = ""
    self.node = 0
    self.acum_temp = ""
    self.case_1_list = ""
    self.case_2_params = []
    self.errors = []
    self.logs = []
    self.alf_L = (
```

Donde se manejan los datos que necesita para funcionar. Nótese la clase Stack, que nos sirve para comprobar si por ejemplo la misma comilla que ingresé “ | ‘, sea la misma que se recibe al final de una cadena.

Se va analizando carácter por carácter, tal y como lo indica el AFD, si se encuentra algo inesperado, se cancela la operación y se lanza un error. El análisis de los nodos se ve de la siguiente manera:

```python
# node12 es un método
elif self.node == 12:
  self.node_12(val_1, row_count, indx_1)

elif self.node == 13:
  print("NODE 13")

  print(val_1)
  print(val_1 == "]")
  if val_1 == ",":
    self.case_1_list += ","
    self.node = 5
  elif val_1 == "]":
    self.case_1_list += "]"
    self.node = 14
  elif val_1 != " " and val_1 != "\n":
    self.set_error(f"Se espera únicamente un cierre de lista ']' o una coma ','. Caracter inválido: {val_1}; fila
```

Se analiza cada caso posible, y por ejemplo, en el nodo 12, se usa en vez un método separado, ya que su “análisis” es necesario para continuación de otros nodos, así que es una buena manera de mantenerlo todo centralizado.

Es así como se van acumulando lo que se va obteniendo siempre y cuando de manera sintácticamente esté bien (caso contrario, lanza un error), se manda a los siguientes métodos:

```
def check_case_1(self, val, set_er = True):
  if val == "Claves":
    return "Claves"
  elif val == "Registros":
    return "Registros"
  else:
    if set_er:
      self.set_error(f"{val} no es un comando conocido")
    return None

def check_case_2(self, val, set_er = True):
  if val == "imprimir":
    return "imprimir"
  elif val == "imprimirln":
    return "imprimirln"
  elif val == "conteo":
    return "conteo"
  elif val == "promedio":
```

Se chequea si es un método válido, y si lo es se procede a la siguiente clase:

```
class GramaticAnalizer:
  def __init__(self) -> None:
    self.db = DB()

  def an_case_1(self, key_w, cadena_entrada):
    errores = []
    mensajes = []

    if key_w == "Claves":
      print(cadena_entrada)
      if len(self.db.obtener_claves()) == 0:
        lista_resultante = []

        if cadena_entrada.startswith('[') and cadena_entrada.endswith(']'):
          contenido = cadena_entrada[1:-1]

          elementos = contenido.split(',')
```

Una vez chequeados, se chequean si los parámetros internos de cada “comando” están bien. Si lo están, se procede a enviar un mensaje, sino, se procede a mandar un error y cancelar la ejecución de esa instrucción.

En este proyecto también es posible generar reportes ya sea a través del código exportarReporte(“nombre_del_reporte”);, o con el botón que se encuentra en la interfaz gráfica.