Práctica 1

Ian Mendoza Jaimes

Compiladores

Profesor: Rafael Norman Saucedo Delgado

Grupo: 3CM6

Índice

1. Introducción

Los autómatas finitos pueden ser definidos formalmente como una tupla (Q, Σ, δ, F), de donde:

- Q es un conjunto finito de estados.
- Σ es un alfabeto.
- δ es su función de transición.
- F es un conjunto finito de estados finales tales que $F \subseteq Q$.

Tienen multiples aplicaciones dentro de la teoría de la computación tales como: el análisis de textos, software para el diseño de circuitos electrónicos, verificación de sistemas de todo tipo con un número finito de estados, compiladores, etc [1]. Para nuestros fines, nos enfocaremos brevemente en esta última aplicación.

La primera etapa de un compilador es el llamado *analizador léxico*, el cual se encarga de descomponer el texto ingresado en unidades lógicas que esten dentro del lenguaje que modele el compilador. Los automatas son el corazón de esta etapa de compilación gracias a su capacidad de modelar lenguajes formales.

En esta práctica, se realizará la implementación de las clases AFD y AFN en algún lenguaje de programación orientado a objetos. Teniendo como finalidad acentar las bases para la creación de un analizador léxcio.

2. Desarrollo

2.1. Descripción de la problema

Se deben implementar las clases Afn y Afd. Deben de permitir crear cualquier tipo de autómata y evaluar alguna cadena ingresada por el usuario.

2.2. Código

main.py

```
from afnd import Afn, Afd

class Main(object):

    def iniciar(self):
        automata = Afn()

        automata.anadirEstado()
        automata.anadirEstado()
        automata.anadirEstado()
        automata.anadirEstado()
        automata.anadirEstado()
        automata.anadirEstado()

        automata.anadirFinal(6)

        automata.anadirTransicion(1, 2, ['1'])
        automata.anadirTransicion(2, 1, ['1'])
        automata.anadirTransicion(2, 3, ['0'])
        automata.anadirTransicion(3, 2, ['0'])
        automata.anadirTransicion(3, 4, ['1'])
        automata.anadirTransicion(4, 3, ['1'])
        automata.anadirTransicion(4, 1, ['0'])
        automata.anadirTransicion(1, 4, ['0'])
        automata.anadirTransicion(4, 5)
        automata.anadirTransicion(5, 6, ['j'])

        for x in automata.crearTablaEstados():
            print(x)

        automata.dibujarAutomata()
        print('====')
        while True:
            print(automata.evaluarCadena(input("Ingresa una cadena: ")))
            if input("Quieres ingresar otra?: s/n    ") != 's':
                break


main = Main()
main.iniciar()
```

afnd.py

```
from estado import Estado, Transicion
import networkx as nx
import matplotlib.pyplot as plt

class Afn(object):

    def __init__(self):
        self.estados = []
        self.tablaEstados = []
        self.contadorEstados = 0
        self.inicial = 0
        self.historialCondiciones = set()
        self.err = 0

    def anadirEstado(self):
        self.contadorEstados += 1

        if self.inicial == 0:
            self.estados.append(Estado(self.contadorEstados, True))
            self.inicial = self.contadorEstados
        else:
            self.estados.append(Estado(self.contadorEstados, False))

        return self.contadorEstados

    def anadirTransicion(self, estado, siguiente, condiciones=[]):
        if estado > self.contadorEstados or estado < 1:
            return -1

        if siguiente > self.contadorEstados or estado < 1:
            return -1

        for x in condiciones:
            self.historialCondiciones.add(x)

        if len(condiciones) > 0:
            transicion = Transicion(siguiente, condiciones)
            self.estados[estado-1].anadirTransicion(transicion)
            for condicion in condiciones:
                self.historialCondiciones.add(condicion)
        else:
            transicion = Transicion(siguiente)
            self.estados[estado-1].anadirTransicion(transicion)

        return 0

    def anadirFinal(self, estado):
        if estado > self.contadorEstados or estado < 1:
            return -1

        self.estados[estado-1].volverFinal()
        return 0
```

```
53
54      def cambiarInicial(self, estado):
55          if estado > self.contadorEstados or estado < 1:
56              return -1
57
58          self.estados[self.inicial-1].volverInicial()
59          self.estados[estado-1].volverInicial()
60          self.inicial = estado
61          return 0
62
63      def dibujarAutomata(self):
64          G = nx.DiGraph()
65          etiqueta = {}
66          for estado in self.estados:
67              if estado.inicial:
68                  print('SOY EL INICIAL:', estado.nombre)
69              if estado.final:
70                  print('SOY EL FINAL:', estado.nombre)
71              for transicion in estado.transiciones:
72                  G.add_edge(estado.nombre, transicion.siguiente)
73                  etiqueta[estado.nombre, transicion.siguiente] =
                        transicion.condiciones
74
75          pos=nx.spring_layout(G)
76
77          nx.draw_networkx_nodes(G, pos, node_size=500, node_color="blue")
78          nx.draw_networkx_edges(G, pos,width=2, alpha=0.5, edge_color='
                black')
79          nx.draw_networkx_labels(G, pos, font_size=5, font_family='sans-
                serif')
80
81          nx.draw_networkx_edge_labels(G, pos, etiqueta, label_pos=0.3,
                with_labels = True)
82
83          plt.show();
84
85      def manejarEpsilon(self, nombreEstado):
86          epsilons = self.estados[nombreEstado - 1].obtenerNumEpsilons()
87          aux = [nombreEstado]
88
89          if len(epsilons) == len(self.estados[nombreEstado - 1].
                transiciones) and len(self.estados[nombreEstado - 1].
                transiciones) > 0:
90              aux.pop()
91          if len(epsilons) == 0:
92              return aux
93
94          for e in epsilons:
95              aux += self.manejarEpsilon(e)
96
97          return aux
98
99
100     def obtenerTransiciones(self, nombreEstado):
```

```
101         aux = dict()
102         aux[' '] = []
103         for transicion in self.estados[nombreEstado - 1].transiciones:
104             for condicion in transicion.condiciones:
105                 if condicion not in aux:
106                     aux[condicion] = []
107 
108                 aux[condicion].append(transicion.siguiente)
109 
110             if len(transicion.condiciones) == 0:
111                 aux[' '] += self.manejarEpsilon(transicion.siguiente)
112 
113         return aux
114 
115 
116     def crearTablaEstados(self):
117         if len(self.estados) == 0:
118             return -1
119 
120         aux = []
121         cont = 0
122         for estado in self.estados:
123             aux.append([])
124             for condicion, etds in self.obtenerTransiciones(estado.
                    nombre).items():
125                 if len(etds) > 0:
126                     aux[cont].append([condicion, etds])
127             cont += 1
128 
129         self.tablaEstados = aux
130         return self.tablaEstados
131 
132 
133     def validarCadena(self, cadena):
134         if type(cadena) is not str:
135             return False
136         if len(self.tablaEstados) == 0:
137             print(cadena)
138         if type(self.inicial) is not int:
139             return False
140 
141         return True
142 
143 
144     def evaluarEpsilon(self, estados, caracter):
145         temp = []
146         for e in estados:
147             for x in self.tablaEstados[e-1]:
148                 if x[0] == caracter:
149                     temp += x[1]
150                 if x[0] == ' ':
151                     temp += self.evaluarEpsilon(x[1], caracter)
152         return temp
153 
```

```
154
155     def validacionFinal(self, estados):
156         temp = []
157         for e in estados:
158             epsilons = self.estados[e-1].obtenerNumEpsilons()
159             if len(epsilons) == 0:
160                 temp += [e]
161             else:
162                 temp += self.validacionFinal(epsilons)
163         return temp
164
165
166     def evaluarCadena(self, cadena):
167         if not self.validarCadena(cadena):
168             return False
169
170         if len(cadena) == 0:
171             cadena = ' '
172
173         estds = [self.inicial]
174         temp = []
175         temp2 = []
176
177         for caracter in cadena:
178             temp = []
179             for e in estds:
180                 for x in self.tablaEstados[e-1]:
181                     if x[0] == caracter:
182                         temp += x[1]
183                     if x[0] == ' ':
184                         temp += self.evaluarEpsilon(x[1], caracter)
185             estds = temp
186
187         estds = self.validacionFinal(estds)
188
189         for e in estds:
190             if self.estados[e-1].final:
191                 return True
192
193         return False
194
195
196 class Afd(Afn):
197
198     def anadirTransicion(self, estado, siguiente, condiciones=[]):
199         if len(condiciones) > 1:
200             return -1
201
202         if estado > self.contadorEstados or estado < 1:
203             return -1
204
205         if siguiente > self.contadorEstados or estado < 1:
206             return -1
207
```

```
208             if len(condiciones) > 0:
209                 transicion = Transicion(siguiente, condiciones)
210                 self.estados[estado-1].anadirTransicion(transicion)
211             else:
212                 transicion = Transicion(siguiente)
213                 self.estados[estado-1].anadirTransicion(transicion)
214 
215             return 0
```

estado.py

```
1  class Transicion(object):
2      def __init__(self, siguiente, condiciones=[]):
3          self.condiciones = condiciones
4          self.siguiente = siguiente
5  
6  
7  class Estado(object):
8      def __init__(self, nombre, inicial=False, final=False):
9          self.nombre = nombre
10         self.inicial = inicial
11         self.final = final
12         self.transiciones = []
13 
14     def anadirTransicion(self, transicion):
15         self.transiciones.append(transicion)
16         return 0;
17 
18     def obtenerNumEpsilons(self):
19         cont = []
20         for t in self.transiciones:
21             if len(t.condiciones) == 0:
22                 cont.append(t.siguiente)
23         return cont
24 
25     def volverFinal(self):
26         self.final = not self.final
27         return 0;
28 
29     def volverInicial(self):
30         self.inicial = not self.inicial
31         return 0;
```

3. Resultados

En esta sección se presentarán capturas de pantalla de los resultados arrojados por el código mostrado anteriormente. En la figura 1 se muestra el autómata que se generó.

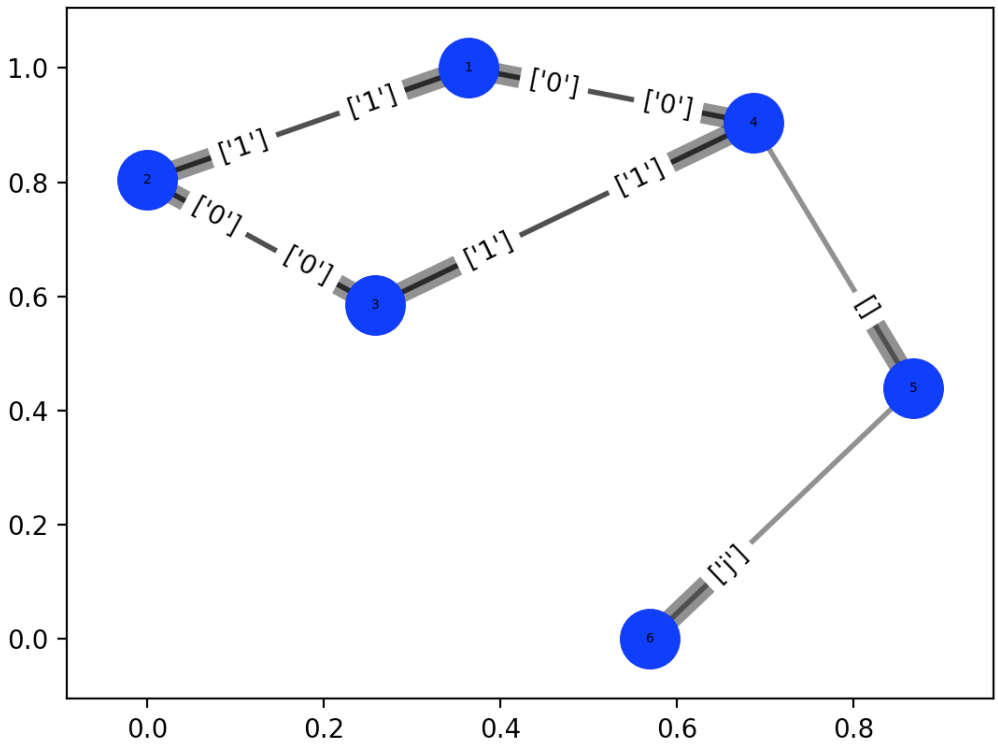

Figura 1: El autómata generado.

En la figura 2 se muestra el resultado de varias cadenas ingresadas al autómata generado y como retorna True si pertenecen al lenguaje modelado o False de otro modo.

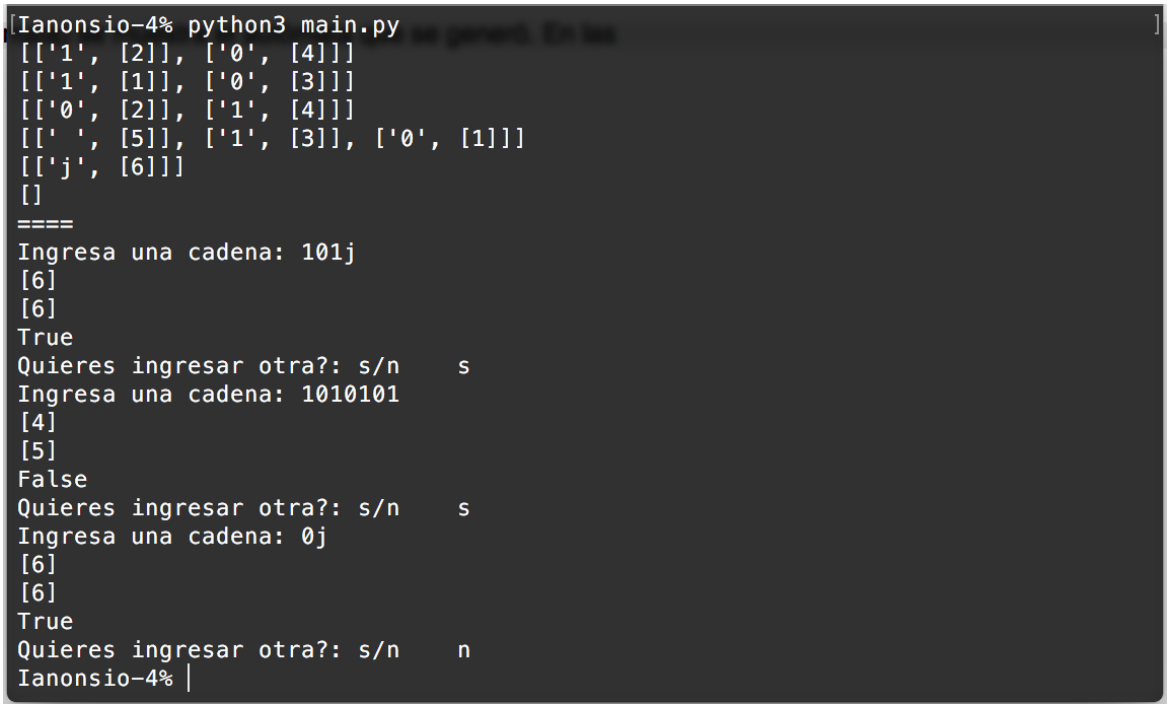

Figura 2: El resultado de varias evaluaciones.

4. Conclusiones

Los autómatas finitos son una manera de modelar un lenguaje formal. Se encargan de evaluar una cadena y decirnos si pertenece o no a algún lenguaje. Existen dos tipos de autómatas finitos: deterministas y no deterministas.

El estudio de los autómtas es importante pues con ellos es posible hacer busquedas en grandes cantidades de información de una manera optima, pero también tienen otras aplicaciones más concretas. En el caso de los compiladores, componen al analizador léxico.

Finalmente, en esta práctica, se recordaron los conceptos principales acerca de los autómatas finitos y se permitio crear la base para lo que será la construcción de la primera etapa de un compilador.

5. Referencias

[1] *Introduction to Automata Theory, Lenguajes and Computation*. 2001, ch. 2.