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 = False
        self.err = 0

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

        if not self.inicial:
            self.estados.append(Estado(self.contadorEstados, True))
            self.inicial = self.contadorEstados
        else:
            self.estados.append(Estado(self.contadorEstados, False))

        return 0

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

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

        if len(condiciones) > 0:
            transicion = Transicion(siguiente, condiciones)
            self.estados[estado-1].anadirTransicion(transicion)
        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

    def dibujarAutomata(self):
        G = nx.DiGraph()
        etiqueta = {}
        for estado in self.estados:
            for transicion in estado.transiciones:
```

```
53                  G.add_edge(estado.nombre, transicion.siguiente)
54                  etiqueta[estado.nombre, transicion.siguiente] =
                        transicion.condiciones
55
56          pos=nx.spring_layout(G)
57
58          nx.draw_networkx_nodes(G, pos, node_size=500, node_color="blue")
59          nx.draw_networkx_edges(G, pos,width=2, alpha=0.5, edge_color='
                black')
60          nx.draw_networkx_labels(G, pos, font_size=5, font_family='sans-
                serif')
61
62          nx.draw_networkx_edge_labels(G, pos, etiqueta, label_pos=0.3,
                with_labels = True)
63
64          plt.show();
65
66      def manejarEpsilon(self, nombreEstado):
67          epsilons = self.estados[nombreEstado - 1].obtenerNumEpsilons()
68          aux = [nombreEstado]
69
70          if len(epsilons) == len(self.estados[nombreEstado - 1].
                transiciones) and len(self.estados[nombreEstado - 1].
                transiciones) > 0:
71              aux.pop()
72          if len(epsilons) == 0:
73              return aux
74
75          for e in epsilons:
76              aux += self.manejarEpsilon(e)
77
78          return aux
79
80
81      def obtenerTransiciones(self, nombreEstado):
82          aux = dict()
83          aux[' '] = []
84          for transicion in self.estados[nombreEstado - 1].transiciones:
85              for condicion in transicion.condiciones:
86                  if condicion not in aux:
87                      aux[condicion] = []
88
89                  aux[condicion].append(transicion.siguiente)
90
91              if len(transicion.condiciones) == 0:
92                  aux[' '] += self.manejarEpsilon(transicion.siguiente)
93
94          return aux
95
96
97      def crearTablaEstados(self):
98          if len(self.estados) == 0:
99              return -1
100
```

```
101         aux = []
102         cont = 0
103         for estado in self.estados:
104             aux.append([])
105             for condicion, etds in self.obtenerTransiciones(estado.
                    nombre).items():
106                 if len(etds) > 0:
107                     aux[cont].append([condicion, etds])
108             cont += 1
109 
110         self.tablaEstados = aux
111         return self.tablaEstados
112 
113     def validarCadena(self, cadena):
114         if len(cadena) == 0:
115             return False
116         if type(cadena) is not str:
117             return False
118         if len(self.tablaEstados) == 0:
119             print(cadena)
120         if type(self.inicial) is not int:
121             return False
122 
123         return True
124 
125     def evaluarEpsilon(self, estados, caracter):
126         temp = []
127         for e in estados:
128             for x in self.tablaEstados[e-1]:
129                 if x[0] == caracter:
130                     temp += x[1]
131                 if x[0] == ' ':
132                     temp += self.evaluarEpsilon(x[1], caracter)
133         return temp
134 
135     def validacionFinal(self, estados):
136         temp = []
137         for e in estados:
138             epsilons = self.estados[e-1].obtenerNumEpsilons()
139             if len(epsilons) == 0:
140                 temp += [e]
141             else:
142                 temp += self.validacionFinal(epsilons)
143         return temp
144 
145     def evaluarCadena(self, cadena):
146         if not self.validarCadena(cadena):
147             return False
148 
149         estds = [self.inicial]
150         temp = []
151         temp2 = []
152 
153         for caracter in cadena:
```

```
154             temp = []
155             for e in estds:
156                 for x in self.tablaEstados[e-1]:
157                     if x[0] == caracter:
158                         temp += x[1]
159                     if x[0] == ' ':
160                         temp += self.evaluarEpsilon(x[1], caracter)
161             estds = temp
162 
163         print(estds)
164         estds = self.validacionFinal(estds)
165         print(estds)
166 
167         for e in estds:
168             if self.estados[e-1].final:
169                 return True
170 
171         return False
172 
173 
174 class Afd(Afn):
175 
176     def anadirTransicion(self, estado, siguiente, condiciones=[]):
177         if len(condiciones) > 1:
178             return -1
179 
180         if estado > self.contadorEstados or estado < 1:
181             return -1
182 
183         if siguiente > self.contadorEstados or estado < 1:
184             return -1
185 
186         if len(condiciones) > 0:
187             transicion = Transicion(siguiente, condiciones)
188             self.estados[estado-1].anadirTransicion(transicion)
189         else:
190             transicion = Transicion(siguiente)
191             self.estados[estado-1].anadirTransicion(transicion)
192 
193         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.

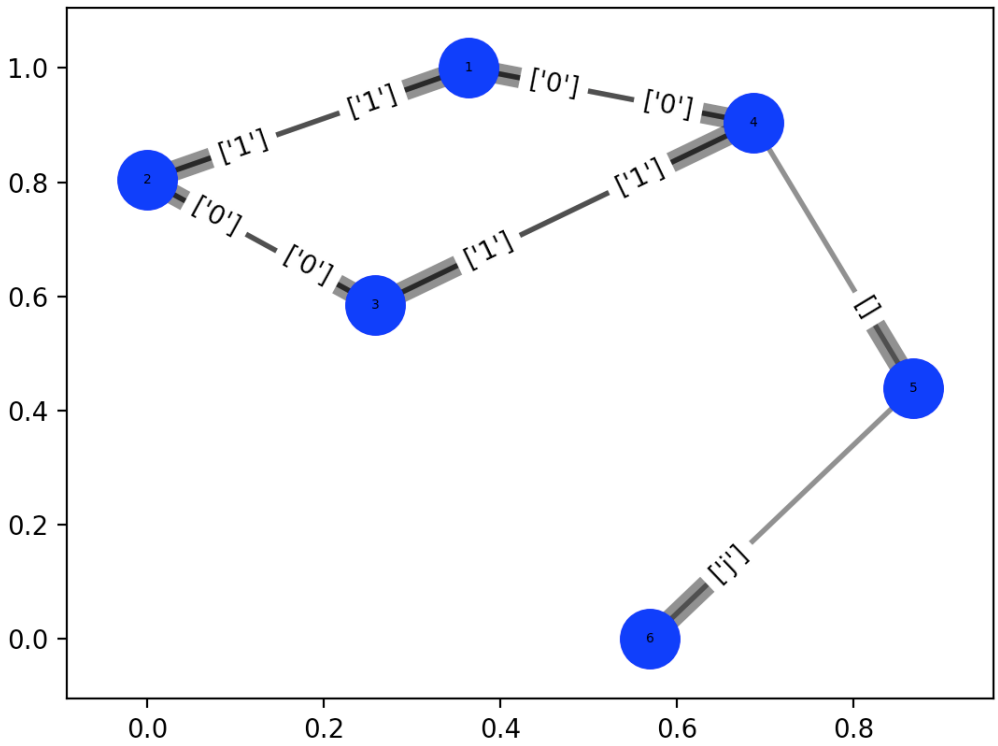

Figura 1: El autómata generado.

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.

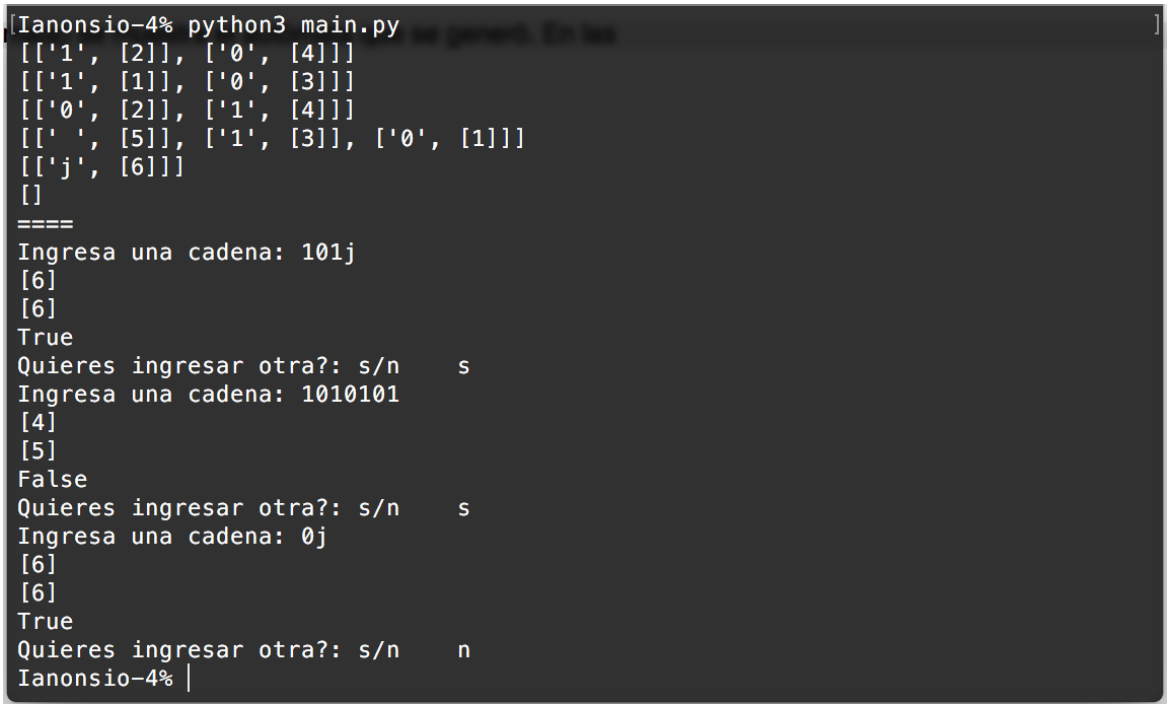

Figura 2: El resultado de varias evaluaciones.

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.