

Reporte Práctica: 1.- Clases AFN y AFD

Alumno: Monroy Martos Elioth

Profesor: Saucedo Delgado Rafael Norman

Materia: Compiladores

Grupo: 3CM6

23 de septiembre de 2017

Índice

1. Introducción	3
2. Desarrollo	4
3. Resultados	12
4. Conclusiones	16
Referencias	16

1. Introducción

Los autómatas son programas comúnmente utilizados para reconocer lenguajes regulares (indica si una palabra pertenece o no a un lenguaje). Estos están compuestos de un conjunto de estados terminales, no terminales, un alfabeto, un estado inicial y una función de transferencia la cual podría considerarse como el control con la cual el autómata sabe como desplazarse dentro de sus estados.

Para la labor de reconocer lenguajes regulares, se pueden usar los automatas finitos deterministas (AFD) y los automatas finitos no deterministas (AFND), siendo la principal diferencia entre ellos la cantidad de estados en los que se puede encontrar un autómata en un determinado momento. Para los AFD solo es posible encontrarse en un estado en todo momento mientras que para los AFND es posible estar en más de un estado en un determinado tiempo.

Esto le permite a los AFND ser más eficientes en el reconocimiento de lenguajes regulares, sin embargo, los AFD son más eficientes en tiempo de procesamiento, ya que los AFND para ser implementados requieren el uso de funciones recursivas o de hilos, mientras que los AFD pueden ser implementados mucho más fácilmente.

Al final, el autómata entrega como resultado sólo dos posibles estados; Aceptado o no aceptado, esto refiriéndose a la cadena que recibió de entrada.

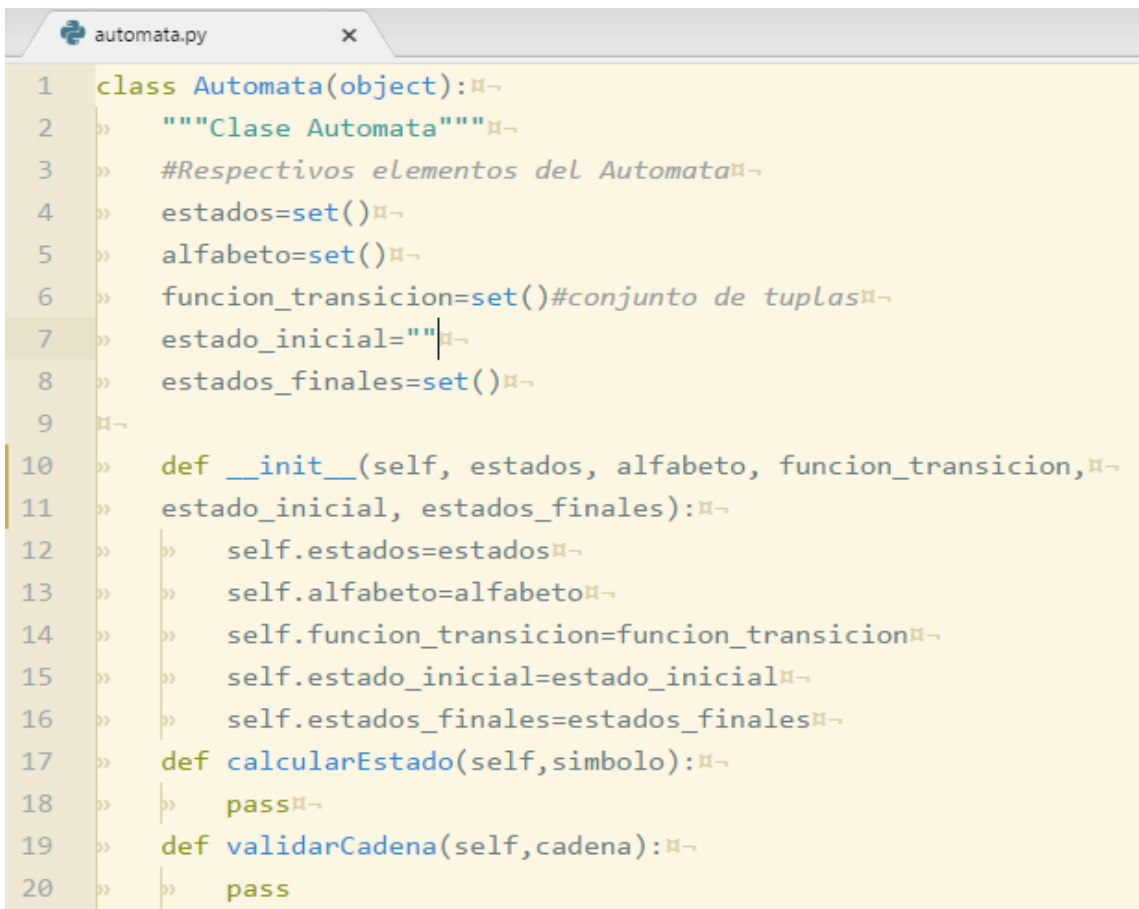
El siguiente trabajo, consiste en la implementación de una clase AFD y otra AFND (en el lenguaje Python) las cuales sean capaces de realizar la validación de cadenas que sean validas para cierto autómata, el cual será cargado mediante un archivo, el cual contendrá los 5 componentes que todo autómata debe tener mencionados arriba.

2. Desarrollo

El desarrollo de la práctica consistió en lo siguiente:

Clases AFD y AFN Ambas clases heredan de un clase padre llamada *Automata*, la cual define los elementos que todo autómata debe tener (Alfabeto, Estados, Función de Transición, Estado Inicial y Estados Finales) y define algunos métodos para que cada tipo de autómata implemente (calcularEstado y validarCadena).

El código de esta clase padre (automata.py) es el siguiente:



```
1 class Automata(object):  
2     """Clase Automata"""  
3     #Respectivos elementos del Automata  
4     estados=set()  
5     alfabeto=set()  
6     funcion_transicion=set()#conjunto de tuplas  
7     estado_inicial=""  
8     estados_finales=set()  
9  
10    def __init__(self, estados, alfabeto, funcion_transicion,  
11    estado_inicial, estados_finales):  
12        self.estados=estados  
13        self.alfabeto=alfabeto  
14        self.funcion_transicion=funcion_transicion  
15        self.estado_inicial=estado_inicial  
16        self.estados_finales=estados_finales  
17    def calcularEstado(self,simbolo):  
18        pass  
19    def validarCadena(self,cadena):  
20        pass
```

Figura 1: automata.py

A continuación se muestra el código de la clase AFD la cual actúa como un autómata finito determinista heredando de la clase autómata:

```

1  from automata import Automata
2  class AFD(Automata):
3      """Automata Finito Determinista"""
4      estado_actual=""
5
6      #Constructor de La Clase AFD
7      def __init__(self, estados, alfabeto, funcion_transicion, estado_inicial, estados_finales):
8          #Llamamos al constructor de la clase padre
9          Automata.__init__(self, estados, alfabeto,
10             funcion_transicion, estado_inicial, estados_finales)
11      #Método que permite calcula el estado actual del automata
12
13      def calcularEstado(self, simbolo):
14          #Variable auxiliar que nos permite saber si el estado actual esta
15          #en alguna función de transición
16          aux=0
17          #Itero en cada una de las funciones de transición
18          for transicion in sorted(self.funcion_transicion):
19              #print(transicion)
20              #Si fue encontrado el estado actual en alguna función de transición entonces:
21              if transicion[0]==self.estado_actual:
22                  #Ahora evaluo si el simbolo corresponde a esa transición
23                  if transicion[1]==simbolo:
24                      #La bandera aux toma el valor de uno
25                      aux=1
26                      #En caso afirmativo, actualizo el estado actual del automata
27                      #Y termino el ciclo, ya que solo puede haber una función de transición que
28                      #cumpla con esas características (en un AFD)
29                      self.estado_actual=transicion[2]
30                      break
31              #Si aux es cero, significa que ninguna función de transición uso ese simbolo y por lo tanto
32              #la cadena es invalida
33              if aux==0:

```

Figura 2: afd.py (1)

```

33     »     » if aux==0:↵
34     »     »     self.estado_actual="muerto"↵
35     » ↵
36     »     #Método que permite evaluar una cadena con el automata↵
37     »     def validarCadena(self,cadena):↵
38     »     »     #Inicializo el estado actual del automata al estado inicial↵
39     »     »     self.estado_actual=self.estado_inicial↵
40     »     »     #Itero por cada simbolo en la cadena↵
41     »     »     for simbolo in cadena:↵
42     »     »     »     #print(simbolo)↵
43     »     »     »     #Si el simbolo se encuentra en el alfabeto, entonces procedo a evaluarlo↵
44     »     »     »     if simbolo in self.alfabeto:↵
45     »     »     »     »     #print(self.estado_actual)↵
46     »     »     »     »     #Calculo el estado actual del automata a partir del simbolo ingresado↵
47     »     »     »     »     self.calcularEstado(simbolo)↵
48     »     »     »     #Si el simbolo no pertenece al alfabeto entonces la cadena no es valida↵
49     »     »     »     else:↵
50     »     »     »     »     self.estado_actual="muerto"↵
51     »     »     #Si el estado actual del automata esta entre los estados finales entonces↵
52     »     »     #La cadena es valida↵
53     »     »     if self.estado_actual in self.estados_finales:↵
54     »     »     »     return "Cadena Valida"↵
55     »     »     #En caso contrario, la cadena es invalida↵
56     »     »     else:↵
57     »     »     »     return "Cadena Invalida"↵

```

Figura 3: afd.py (2)

A continuación se muestra el código de la clase AFND la cual actúa como un autómata finito no determinista (el cual permite las transiciones epsilon), al igual que la clase anterior, hereda de la clase autómata:

```

1  from automata import Automata
2  class AFND(Automata):
3      """Automata Finito Determinista"""
4      estado_actual=list()
5      estado_actual_aux=list()
6      #Constructor de La Clase AFD
7
8      def __init__(self, estados, alfabeto, funcion_transicion, estado_inicial, estados_finales):
9          #Llamamos al constructor de la clase padre
10         Automata.__init__(self, estados, alfabeto, funcion_transicion, estado_inicial, estados_finales)
11
12         #Método que permite calcula el estado actual del automata
13     def calcularEstado(self, simbolo):
14         #Variable auxiliar que nos permite saber si el estado actual esta
15         #en alguna función de transición
16         recursiva=0
17         aux=0
18         #Itero en cada una de las funciones de transición
19         continuar=0
20         for transicion in sorted(self.funcion_transicion):
21             if recursiva==1:
22                 aux=1
23                 break
24             #print(transicion)
25             #Si fue encontrado el estado actual en alguna función de transición entonces:
26             #print("Estado actual antes if:",self.estado_actual)
27             if transicion[0] in self.estado_actual:
28                 #Ahora evaluo si el simbolo corresponde a esa transición
29                 #print("Simbolo",simbolo)
30                 if transicion[1]==simbolo:
31                     #La bandera aux toma el valor de uno
32                     aux=1
33                     #En caso afirmativo, actualizo el estado actual del automata

```

Figura 4: afnd.py (1)

```

33  »  »  »  »  »  #En caso afirmativo, actualizo el estado actual del automata
34  »  »  »  »  »  #Y termino el ciclo, ya que solo puede haber una función de transición que
35  »  »  »  »  »  #cumpla con esas características (en un AFD)
36  »  »  »  »  »  #print("El simbolo pertenece a la transición")
37  »  »  »  »  »  self.estado_actual_aux.append(transicion[2])
38  »  »  »  »  »  #print(self.estado_actual_aux)
39  »  »  »  »  »  if transicion[1]=="$":
40  »  »  »  »  »  aux=1
41  »  »  »  »  »  continuar=1
42  »  »  »  »  »  self.estado_actual_aux.append(transicion[2])
43  »  »  »  »  »  #print("Estado actual auxiliar:",self.estado_actual_aux)
44  »  »  if continuar==1:
45  »  »  »  self.estado_actual.clear()
46  »  »  »  self.estado_actual=list(self.estado_actual_aux)
47  »  »  »  self.estado_actual_aux.clear()
48  »  »  »  self.calcularEstado(simbolo)
49  »  »  »  recursiva=1
50  »  »  #Si aux es cero, significa que ninguna función de transición uso ese simbolo y por lo tanto
51  »  »  #La cadena es invalida
52  »  »  if recursiva==0:
53  »  »  »  self.estado_actual.clear()
54  »  »  »  self.estado_actual=list(self.estado_actual_aux)
55  »  »  »  self.estado_actual_aux.clear()
56  »  »  if aux==0:
57  »  »  »  self.estado_actual.append("muerto")
58  »
59  »  #Método que permite evaluar una cadena con el automata
60  »  def validarCadena(self,cadena):
61  »  »  #Inicializo el estado actual del automata al estado inicial

```

Figura 5: afnd.py (2)


```

61 » » #Inicializo el estado actual del automata al estado inicial↵
62 » » self.estado_actual.append(self.estado_inicial)↵
63 » » #Itero por cada simbolo en la cadena↵
64 » » for simbolo in cadena:↵
65 » »     » #print(simbolo)↵
66 » »     » #Si el simbolo se encuentra en el alfabeto, entonces procedo a evaluarlo↵
67 » »     » if simbolo in self.alfabeto:↵
68 » »     »     » #print(self.estado_actual)↵
69 » »     »     » #Calculo el estado actual del automata a partir del simbolo ingresado↵
70 » »     »     » self.calcularEstado(simbolo)↵
71 » »     »     » #Si el simbolo no pertenece al alfabeto entonces la cadena no es valida↵
72 » »     »     » else:↵
73 » »     »         self.estado_actual.append("muerto")↵
74 » »     » #Si el estado actual del automata esta entre los estados finales entonces↵
75 » »     » #la cadena es valida↵
76 » »     » #print("Estados al final:",self.estado_actual)↵
77 » »     » for estado in self.estados_finales:↵
78 » »     »     » if estado in self.estado_actual:↵
79 » »     »         » return "Cadena Valida"↵
80 » »     » return "Cadena Invalida"↵

```

Figura 6: afnd.py (3)

Archivos auxiliares Como se mencionó anteriormente, la información necesaria para la creación de los autómatas es ingresada mediante un archivo el cual es ingresado al momento de ejecutar el programa. El archivo contiene un estructura como la que se indica a continuación:

1. Primer línea: El conjunto de estados del autómata, separados por comas.
2. Segunda línea: El alfabeto de entrada, igualmente cada símbolo separado por comas.
3. Tercer línea: función de transición de la siguiente manera: (estado,símbolo,estadoResultado) separadas por guiones.
4. Cuarta línea: Estado inicial.
5. Quinta línea: Estados finales o de aceptación separados por comas.

Por ende, fue necesario la creación de una clase extra la cual se encargará de manejar el archivo para mandarle la información correspondiente al autómata. El código de este manejador (archivo.py) es el siguiente:

```

1 class ArchivoAutomata(object):
2     """Clase para leer el archivo de entrada para el automata"""
3     lineas=""
4     def __init__(self, archivo):
5         aux=open(archivo,"r")
6         self.lineas=aux.readlines()
7     def obtenerEstados(self):
8         return set((self.lineas[0].replace("\n","")).split(","))
9     def obtenerAlfabeto(self):
10        return set((self.lineas[1].replace("\n","")).split(","))
11    def obtenerFuncionTransicion(self):
12        transiciones=set()
13        lista=list()#Pueden ser modificadas, por lo cual necesitamos una tupla
14        aux=(self.lineas[2].replace("\n","")).split("-")
15        for tran in aux:
16            tran=tran.replace("(","").replace(")","")
17            for elemento in tran.split(","):
18                lista.append(elemento)
19                transiciones.add(tuple(lista))
20            lista=list()
21        return transiciones
22    def obtenerEstadoInicial(self):
23        return (self.lineas[3].replace("\n",""))
24    def obtenerEstadosFinales(self):
25        return set((self.lineas[4].replace("\n","")).split(","))

```

Figura 7: archivo.py

3. Resultados

Para comprobar la funcionalidad del programa, primero probaré ingresando un AFD y posteriormente un AFND

AFD El AFD que creará nuestra clase AFD es el siguiente:

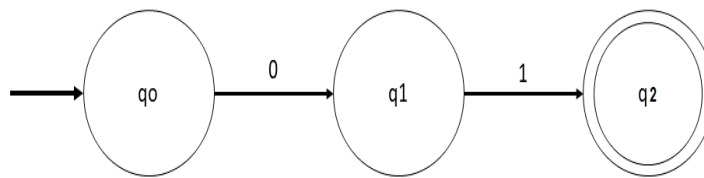


Figura 8: AFD de Prueba

Este autómata solo acepta la cadena '01'. Cualquier otra cadena sería inválida.

El archivo de entrada para la creación de este autómata, luce de la siguiente manera:

1	<code>q0,q1,q2</code>
2	<code>0,1</code>
3	<code>(q0,0,q1)-(q1,1,q2)</code>
4	<code>q0</code>
5	<code>q2</code>

Figura 9: Archivo de entrada para el AFD

Al ejecutar el programa e ingresar el archivo, y posteriormente las cadenas, obtenemos las salidas esperadas.

```
C:\> Símbolo del sistema

C:\Users\ELITH\Documents\GitHub\Compilers\1>python main.py
Por favor indique el tipo de automata a crear:
<1>AFD <2>AFND
1
Por favor ingrese el nombre del archivo de entrada:
Ejem: entrada.txt
ejemplo.txt
El automata se construira con base al archivo: ejemplo.txt
<> Automata Finito Determinista Creado <>

Por favor ingrese una cadena para validarla:
01
Cadena Valida
Desea probar otra cadena?
<1>Si <2>No:
1
Por favor ingrese una cadena para validarla:
10
Cadena Invalida
Desea probar otra cadena?
<1>Si <2>No:
2
Desea Continuar?:
<1>Si <2>No:
2
C:\Users\ELITH\Documents\GitHub\Compilers\1>_
```

Figura 10: Ejecución y pruebas del AFD

AFND El AFND que creará nuestra clase AFND es el siguiente:

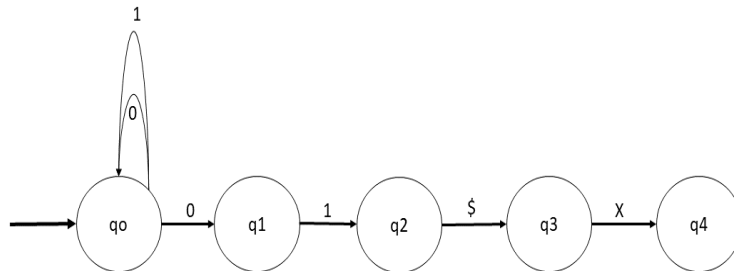


Figura 11: AFND de Prueba

Cabe señalar, que las transiciones epsilon son representadas por el símbolo '\$'.

Este autómata acepta la cadena '001x' por ejemplo, en general acepta cualquier cadena la cual empiece con cualquier cantidad de '0' o '1' pero que siempre al final lleve la siguiente combinación '1x'. Cualquier otra cadena sería invalida, por ejemplo, cadena '00x1'.

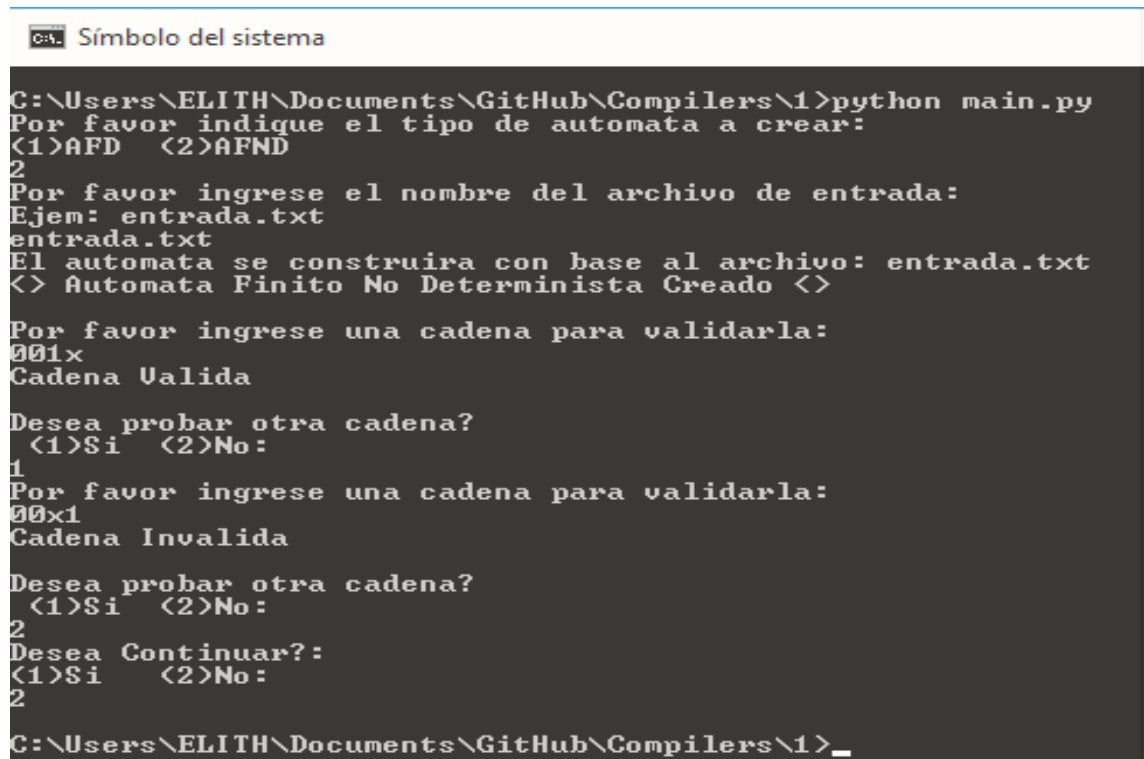
El archivo de entrada para la creación de este autómata, luce de la siguiente manera:

```

1 q0,q1,q2,q3,q4
2 0,1,$,x
3 (q0,0,q0)-(q0,1,q0)-(q0,0,q1)-(q1,1,q2)-(q2,$,q3)-(q3,x,q4)
4 q0
5 q4
  
```

Figura 12: Archivo de entrada para el AFND

Al ejecutar el programa e ingresar el archivo, y posteriormente las cadenas, obtenemos las salidas esperadas.



```
C:\Users\ELITH\Documents\GitHub\Compilers\1>python main.py
Por favor indique el tipo de automata a crear:
<1>AFD <2>AFND
2
Por favor ingrese el nombre del archivo de entrada:
Ejem: entrada.txt
entrada.txt
El automata se construira con base al archivo: entrada.txt
<> Automata Finito No Determinista Creado <>

Por favor ingrese una cadena para validarla:
001x
Cadena Valida

Desea probar otra cadena?
<1>Si <2>No:
1
Por favor ingrese una cadena para validarla:
00x1
Cadena Invalida

Desea probar otra cadena?
<1>Si <2>No:
2
Desea Continuar?:
<1>Si <2>No:
2
C:\Users\ELITH\Documents\GitHub\Compilers\1>_
```

Figura 13: Ejecución y pruebas del AFND

4. Conclusiones

El uso de autómatas finitos deterministas y no deterministas nos permite validar que el orden en cadenas de entrada sea correcto. Esto tiene diversas aplicaciones, desde muy sencillas (como validar el formato de un CURP) hasta aplicaciones más complejas como realizar parte del análisis léxico de un lenguaje de entrada en un compilador.

Los cuales nos ayudan a saber si una entrada pertenece a nuestro lenguaje o no, siendo de gran utilidad al momento de validar cadenas a partir de una expresión regular.

Referencias

- [1] V. A. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1st ed., 1986.
- [2] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd ed., 2001.