

## Reporte: Práctica 2

Barrera Pérez Carlos Tonatihu  
Profesor: Saucedo Delgado Rafael Norman  
Compiladores  
Grupo: 3CM6

2 de septiembre de 2017

# Índice

1. Introducción	2
2. Desarrollo	3
3. Resultados	5
4. Conclusiones	7
Referencias	7

# 1. Introducción

Existen diferentes formas para poder construir un autómata finito no determinista a partir de una expresión regular, en esta práctica se utilizó la construcción de Thomson [1] con este algoritmo cada vez que se representa algún símbolo, una unión, concatenación, cerradura de Kleene o positiva se crean dos nuevos estados. El algoritmo consiste en lo siguiente:

- Para  $\epsilon$  se construyen dos estados y la transición entre ellos se etiqueta con épsilon como en la figura 1

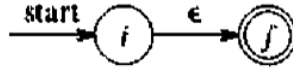


Figura 1: Épsilon representado en un autómata

- Para algún símbolo del alfabeto se realiza lo mismo que para  $\epsilon$  pero esta vez se etiqueta con el respectivo símbolo como el ejemplo de la figura 2.

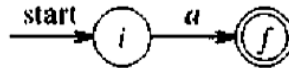


Figura 2: Representación de la transición de un estado a otro a través del símbolo  $a$ .

- La unión de dos expresiones regulares  $s$  y  $t$  se representa a través del uso de transiciones épsilon como se muestra en la figura 3.

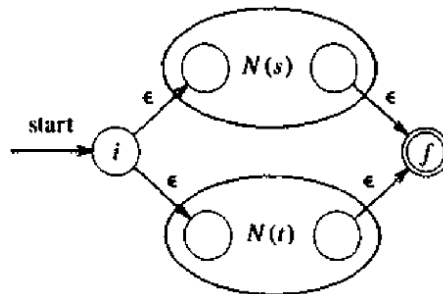


Figura 3: Unión de dos expresiones regulares  $(s|t)$  representadas en un AFN- $\epsilon$ .

- La concatenación de dos expresiones regulares  $s$  y  $t$  se realiza tomando el estado inicial de la transición  $t$  y convirtiéndolo en el estado final de la transición producida por la expresión regular  $s$  como en la figura 4.

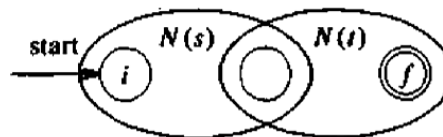


Figura 4: Concatenación  $(st)$  mediante la fusión de un estado final y uno inicial respectivamente.

- La cerradura de Kleene de una expresión regular  $s^*$  se realiza usando transiciones épsilon entre los estados que componen esta expresión. Esto se muestra en la figura 5.

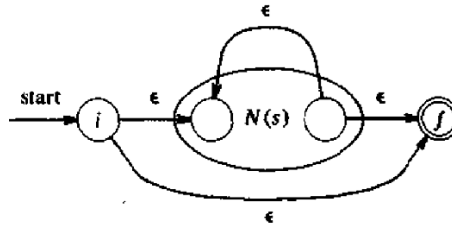


Figura 5: Cerradura ( $s^*$ ).

- La cerradura positiva es similar a la de Kleene pero esta no realiza una transición del estado inicial al final.

Finalmente, para la evaluación de la expresión regular primero la pasamos a postfijo con el algoritmo shunting yard [2] que se utiliza para la evaluación de expresiones matemáticas pero adaptándolo a expresiones regulares.

## 2. Desarrollo

La siguiente clase contiene todos los métodos necesarios para obtener un AFN a través de una expresión regular los métodos mas importantes de esta clase son:

- *convertir\_postfijo*. Este método genera la expresión en postfijo mediante el uso del algoritmo shunting yard [2] por lo que utilizamos una pila y una cola, en la pila se almacenan los paréntesis y operadores que irán saliendo de esta pila y se pondrán en la cola dependiendo de la lectura de carácter que se haga. Por otro lado, en la cola se guardara la salida en postfijo que después sera utilizada por el método generar autómata
- *generar\_automata*. Este método toma la salida del método anterior y haciendo uso de las construcciones de Thomson obtiene el autómata. Para hacer esto utiliza una pila donde iremos guardando las transiciones que se generen y después sacándolas cuando sea necesario volver a utilizarlas y formar una nueva transición, del mismo modo guardara las transiciones resultantes, el alfabeto y los estados inicial y final en una instancia de la clase *AFN* para después ser usada en la evaluación de cadenas.

---

```

1 from automata.automatas import Transicion
2 from automata.automatas import AFN
3
4 class Analizador:
5     def __init__(self):
6         self.salida_postfijo = list()
7         self.transiciones = list()
8         self.pila_transiciones = []
9         self.POSITIVA = 0
10        self.KLEENE = 1
11        self.AFN = AFN()
12
13    def mostrar_expresion_postfijo(self):
14        print(self.salida_postfijo)
15
16    def mostrar_automata(self):
17        print('Inicial: %s Finales: %s' % (self.AFN.estado_inicial, self.AFN.estados_finales))
18        print("Transiciones:")

```

```

19     for t in self.AFN.transiciones:
20     print(t)
21
22 def convertir_postfijo(self, cadena):
23     pila = []
24     punto = False
25     for c in cadena:
26         if c == '(':
27             if punto:
28                 while len(pila) > 0 and (pila[-1] == '+' or pila[-1] == '*'):
29                     self.salida_postfijo.append(pila.pop())
30                 pila.append('.')
31                 punto = False
32             pila.append(c)
33         elif c == ')':
34             punto = True
35             while len(pila) > 0 and pila[-1] != '(':
36                 self.salida_postfijo.append(pila.pop())
37             try:
38                 pila.pop()
39             except IndexError as e:
40                 raise e
41         elif c == '+' or c == '*':
42             self.salida_postfijo.append(c)
43         elif c == '|':
44             while len(pila) > 0 and (pila[-1] == '+' or pila[-1] == '*' or pila[-1] == '.'):
45                 self.salida_postfijo.append(pila.pop())
46             pila.append(c)
47             punto = False
48         else:
49             if punto:
50                 while len(pila) > 0 and (pila[-1] == '+' or pila[-1] == '*'):
51                     self.salida_postfijo.append(pila.pop())
52                 pila.append('.')
53                 punto = True
54             self.salida_postfijo.append(c)
55
56     while len(pila) > 0:
57         self.salida_postfijo.append(pila.pop())
58
59 def generar_automata(self):
60     inicial = 1
61     final = 2
62     self.AFN.alfabeto.add('e')
63     for c in self.salida_postfijo:
64         if c == '*':
65             print('CERRADURA KLEENE')
66             s = self.pila_transiciones.pop()
67             self.cerradura(s, self.KLEENE)
68         elif c == '+':
69             print('CERRADURA DE POSITIVA')
70             s = self.pila_transiciones.pop()
71             self.cerradura(s, self.POSITIVA)
72         elif c == '|':
73             print("UNION")
74             s = self.pila_transiciones.pop()
75             t = self.pila_transiciones.pop()
76             i1 = Transicion(s[1] + 1, s[0], 'e')
77             i2 = Transicion(s[1] + 1, t[0], 'e')

```

```

78     f1 = Transicion(s[1], s[1] + 2, 'e')
79     f2 = Transicion(t[1], s[1] + 2, 'e')
80     self.pila_transiciones.append([s[1] + 1, s[1] + 2])
81     self.transiciones.extend((i1, i2, f1, f2))
82 elif c == '.':
83     print('CONCATENACION')
84     t = self.pila_transiciones.pop()
85     s = self.pila_transiciones.pop()
86     for aux in self.transiciones:
87         if aux.siguiente == s[1]:
88             aux.siguiente = t[0]
89     self.pila_transiciones.append([s[0], t[1]])
90 else:
91     print('ESTADO')
92     if self.pila_transiciones.__len__() > 0:
93         inicial = self.pila_transiciones[-1][1] + 1
94         final = inicial + 1
95         transicion = Transicion(inicial, final, c)
96         self.pila_transiciones.append([inicial, final])
97         self.transiciones.append(transicion)
98         self.AFN.alfabeto.add(c)
99
100 self.AFN.agregar_inicial(self.pila_transiciones[0][0])
101 self.AFN.agregar_finales({self.pila_transiciones[0][1]})
102 self.AFN.transiciones = self.transiciones
103
104 def cerradura(self, s, tipo):
105     i1 = Transicion(s[1] + 1, s[0], 'e')
106     s1 = Transicion(s[1], s[0], 'e')
107     s2 = Transicion(s[1], s[1] + 2, 'e')
108     self.pila_transiciones.append([s[1] + 1, s[1] + 2])
109     self.transiciones.extend((i1, s1, s2))
110
111 if tipo == self.KLEENE:
112     i2 = Transicion(s[1] + 1, s[1] + 2, 'e')
113     self.transiciones.append(i2)

```

---

El resto de métodos solo son auxiliares y sirven para mostrar el estado del autómata o para imprimir la cadena en postfijo.

### 3. Resultados

A continuación se presenta el resultado de la implementación de la clase *Analizador* en donde se convierte una expresión regular a un AFN para después evaluar algunas cadenas y ver si son validas. Las pruebas se realizaron con el siguiente código, el cual ocupa la clase *AFN* que se construye gracias a la clase *Analizador* para poder llevar a cabo la evaluación de algunas cadenas.

---

```

1 def correr_analizador():
2     print("Generando el automata de la expresion: (a|b)*abb ...")
3     analizador = Analizador()
4     # Transformamos la expresion a postfijo
5     analizador.convertir_postfijo('(a|b)*abb')
6     analizador.mostrar_expresion_postfijo()
7     # Creamos el AFN a partir de la expresion en postfijo
8     analizador.generar_automata()
9     analizador.mostrar_automata()
10    # Probamos nuestro automata con algunas cadenas
11    print("Pruebas sobre el automata generado...")

```

```

12 automataAFN = analizador.AFN
13 for n in range(5):
14     cadena = input("-Ingresa una cadena: ")
15     print("La cadena es:")
16     if automataAFN.evaluar_cadena(cadena):
17         print("Valida")
18     else:
19         print("No valida")

```

---

```

tona@anti: ~/Documents/quinto/compiladores/Practicas
File Edit View Search Terminal Help
+ Practicas git:(master) x python iniciar.py
Generando el automata de la expresion: (a|b)*abb ...
['a', 'b', '|', '*', 'a', 'b', 'b', '.', '.', '.']
ESTADO
ESTADO
UNION
CERRADURA KLEENE
ESTADO
ESTADO
ESTADO
CONCATENACION
CONCATENACION
CONCATENACION
Inicial: 7 Finales: {14}
Transiciones:
1->2: a
3->4: b
5->3: e
5->1: e
4->6: e
2->6: e
7->5: e
6->5: e
6->9: e
7->9: e
9->11: a
11->13: b
13->14: b
Pruebas sobre el automata generado...
-Ingresa una cadena: ba

```

Figura 6: Transformando la expresión regular  $(a|b)^*abb$  a su respectivo autómata

Como se puede observar en la figura 6 la expresión regular se pasa a notación posfija y después se puede observar como se realiza la creación de las transiciones, lo siguiente que aparece es mostrar los estados inicial y final y finalmente se muestran las transiciones resultantes.

Después se realizaron pruebas sobre el autómata que se genero de forma similar a la práctica anterior.

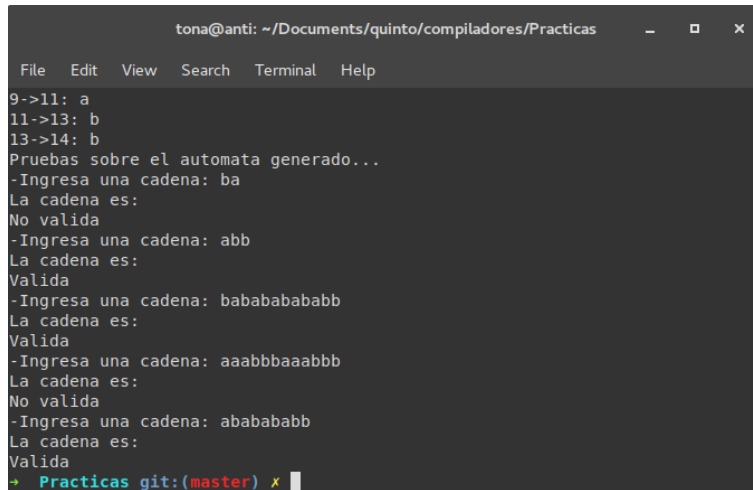
A screenshot of a terminal window titled 'tona@anti: ~/Documents/quinto/compiladores/Practicas'. The terminal shows a series of commands and outputs for testing a generated automaton. The commands are: '9->11: a', '11->13: b', and '13->14: b'. This is followed by the text 'Pruebas sobre el automata generado...'. Then, several test cases are shown: '-Ingresa una cadena: ba' with output 'La cadena es: No valida'; '-Ingresa una cadena: abb' with output 'La cadena es: Valida'; '-Ingresa una cadena: bababababbb' with output 'La cadena es: Valida'; '-Ingresa una cadena: aaabbbbaabbb' with output 'La cadena es: No valida'; and '-Ingresa una cadena: ababababb' with output 'La cadena es: Valida'. The terminal ends with a prompt 'Practicas git:(master) x'.

Figura 7: Pruebas sobre el autómata generado.

La generación del autómata se realizó de forma correcta (figura 7), ya que la evaluación de las cadenas fue satisfactoria esto quiere decir que las transiciones que se muestran en la figura 6 no tuvieron errores.

## 4. Conclusiones

La elaboración de esta práctica fue un poco más sencilla que la anterior debido a que solo consistió en implementar el algoritmo de Thomson y con ello poder construir cualquier AFN, la parte complicada de modelar este algoritmo fue la concatenación ya que requirió unos pasos extra a diferencia del resto de construcciones.

Y como se pudo observar en las pruebas la conversión de expresión regular a autómata se realiza correctamente. Esto demuestra la versatilidad que tienen las expresiones regulares y los autómatas además de que será de mucha utilidad cuando realicemos nuestro analizador léxico que su principal tarea es revisar el vocabulario de nuestro compilador.

## Referencias

- [1] V. A. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1st ed., 1986.
- [2] P. Oser, “The shunting yard algorithm.” <http://www.oxfordmathcenter.com/drupal17/node/628>. Consultado: 2017-09-2.