Práctica 2

Ian Mendoza Jaimes

Compiladores

Profesor: Rafael Norman Saucedo Delgado

Grupo: 3CM6

Índice

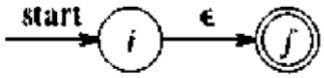
1. Introducción

Las expresiones regulares (ER) son un tipo de notación para definir lenguajes regulares. Estas pueden definir los mismos lenguajes que muchos tipos de autómatas, con la diferencia de que las ER nos ofrecen una descripción algebráica de los lenguajes y por ende, modelarlos u operar sobre ellos es más sencillo [2].

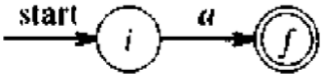
Las operaciones entre lenguajes que los operadores de las expresiones regulares representan son:

- La *unión* de dos lenguajes L y M, es denotada por $L \cup M$, es el conjunto de cadenas que se encuentran tanto en L como en M.
- La *concatenación* de los lenguajes L y M es el conjunto de cadenas que pueden estar formadas de tomar cualquier cadena de L y concatenarla con cualquier cadena de M.
- La *cerradura de Kleen* (también conocida como cerradura estrella) de el lenguaje L es denotada por L^* y representa el conjunto de las cadenas que se pueden formar de tomar cualquier número de cadenas de L, posiblemente con repeticiones. Más formalmente L^* es la unión infinita $\cup_{i \geq 0} L^i$ donde $L^0 = L$ y L^i para $i > 1$ es $LL \cdots L$.
- La *cerradura positiva* de el lenguaje L es denotada por L^+ y representa el mismo conjunto de la cerradura de Kleen, con excepción de que el caracter vacío ε no pertence a L^+.

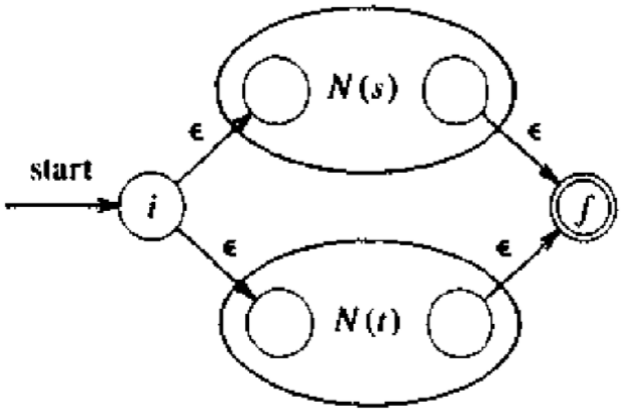
Debido a que las ER modelan los mismos lenguajes regulares que los autómatas, podemos convertir de una ER a un AFN. Un algoritmo que nos permite hacer esto, es el el *algoritmo de Thompson* [1]. El cual, convierte cada operador de las expresiones regulares en su correspondiente AFN. En la figura 1, podemos ver estas conversiones ilustradas.



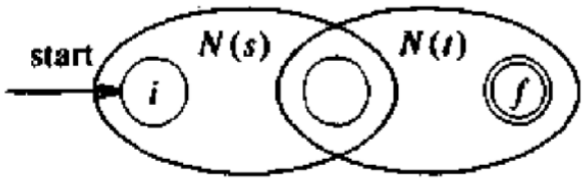
(a)



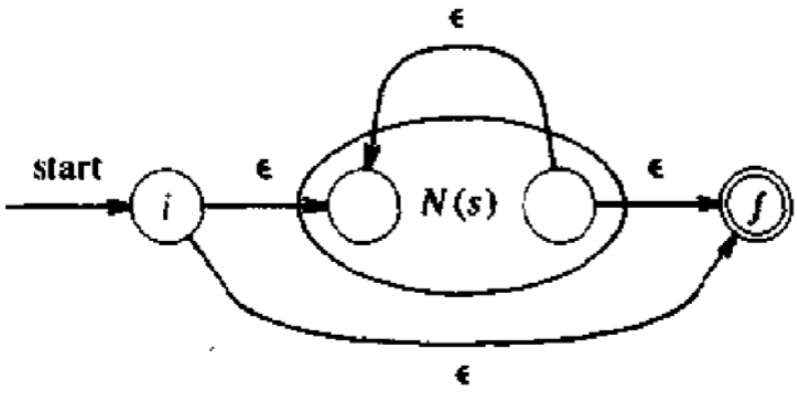
(b)



(c)



(d)



(e)

Figura 1: Conversiones de una ER a su respectivo AFN según el algoritmo de Thompson.

2. Desarrollo

2.1. Descripción de la problema

Se debe de implementar un analizador de expresiones regulares capaz de reconocer todas las operaciones clásicas, las cuales son: unión, concatenación, cerradura positiva y de Kleen. Con la expresión regular, se deberá crear su correspondiente $\epsilon - AFN$. La clase Afn es la misma que la práctica pasada y no se ha considerado incluir su código en este reporte.

2.2. Código

main.py

```
from analizador import Analizador

class Main(object):

    def iniciar(self):
        expresion = input("Ingresa una expresion: ")
        analizador = Analizador()

        analizador.analizar(expresion)
        print(analizador.lista)

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

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


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

analizador.py

```
from automatas.afnd import Afn

class Manejador_nodos(object):
    def __init__(self):
        self.inicio = None
        self.ultimo = None

class Analizador(object):
    def __init__(self):
        self.pila = []
```

```
11          self.lista = []
12          self.automata = None
13
14      def analizar(self, cadena):
15          if not self.validarCadena(cadena):
16              return -1
17          self.convertirPostorden(cadena)
18          self.crearAutomata()
19
20      def crearAutomata(self):
21          self.automata = Afn()
22          for elemento in self.lista:
23              if elemento == '+':
24                  auxiliar1 = self.pila.pop()
25                  auxiliar2 = self.pila.pop()
26                  manejador_nodos = Manejador_nodos()
27                  manejador_nodos.inicio = self.automata.anadirEstado()
28                  manejador_nodos.final = self.automata.anadirEstado()
29                  self.automata.anadirTransicion(manejador_nodos.inicio,
                        auxiliar1.inicio)
30                  self.automata.anadirTransicion(manejador_nodos.inicio,
                        auxiliar2.inicio)
31                  self.automata.anadirTransicion(auxiliar1.final,
                        manejador_nodos.final)
32                  self.automata.anadirTransicion(auxiliar2.final,
                        manejador_nodos.final)
33                  self.pila.append(manejador_nodos)
34
35              elif elemento == '.':
36                  auxiliar1 = self.pila.pop()
37                  auxiliar2 = self.pila.pop()
38                  manejador_nodos = Manejador_nodos()
39                  manejador_nodos.inicio = auxiliar2.inicio
40                  manejador_nodos.final = auxiliar1.final
41                  self.automata.anadirTransicion(auxiliar2.final,
                        auxiliar1.inicio)
42                  self.pila.append(manejador_nodos)
43
44              elif elemento == '^':
45                  auxiliar1 = self.pila.pop()
46                  auxiliar2 = self.pila.pop()
47                  manejador_nodos = Manejador_nodos()
48                  manejador_nodos.inicio = self.automata.anadirEstado()
49                  manejador_nodos.final = self.automata.anadirEstado()
50                  self.automata.anadirTransicion(manejador_nodos.inicio,
                        auxiliar2.inicio)
51                  self.automata.anadirTransicion(auxiliar2.final,
                        manejador_nodos.final)
52                  self.automata.anadirTransicion(auxiliar2.final,
                        auxiliar2.inicio)
53                  if auxiliar1.inicio == -1:
54                      self.automata.anadirTransicion(manejador_nodos.
                            inicio, manejador_nodos.final)
55                  self.pila.append(manejador_nodos)
```

```
56
57              elif elemento == '*':
58                  manejador_nodos = Manejador_nodos()
59                  manejador_nodos.inicio = -1
60                  manejador_nodos.final = -1
61                  self.pila.append(manejador_nodos)
62
63              elif elemento == '++':
64                  manejador_nodos = Manejador_nodos()
65                  manejador_nodos.inicio = -2
66                  manejador_nodos.final = -2
67                  self.pila.append(manejador_nodos)
68
69              elif elemento == 'E':
70                  manejador_nodos = Manejador_nodos()
71                  manejador_nodos.inicio = self.automata.anadirEstado()
72                  manejador_nodos.final = self.automata.anadirEstado()
73                  self.automata.anadirTransicion(manejador_nodos.inicio,
                        manejador_nodos.final)
74                  self.pila.append(manejador_nodos)
75
76              else:
77                  manejador_nodos = Manejador_nodos()
78                  manejador_nodos.inicio = self.automata.anadirEstado()
79                  manejador_nodos.final = self.automata.anadirEstado()
80                  self.automata.anadirTransicion(manejador_nodos.inicio,
                        manejador_nodos.final, [elemento])
81                  self.pila.append(manejador_nodos)
82
83          manejador_nodos = self.pila.pop()
84          self.automata.cambiarInicial(manejador_nodos.inicio)
85          self.automata.anadirFinal(manejador_nodos.final)
86
87
88      def convertirPostorden(self, cadena):
89          concatenacion = False
90          for caracter in cadena:
91              if caracter == '(':
92                  if concatenacion:
93                      self.pila.append('.')
94                  concatenacion = False
95                  self.pila.append(caracter)
96              elif caracter == '+':
97                  concatenacion = False
98                  if self.pilaTope() == '^':
99                      self.lista.append(caracter+caracter)
100                 elif self.pilaTope() == '.':
101                     while not self.pilaVacia():
102                         if self.pilaTope() != '(':
103                             self.lista.append(self.pila.pop())
104                         else:
105                             break
106                 self.pila.append(caracter)
107             else:
```

```
108                     self.pila.append(caracter)
109             elif caracter == '^':
110                 concatenacion = False
111                 self.pila.append(caracter)
112             elif caracter ==')':
113                 concatenacion = True
114                 while True:
115                     if self.pilaTope() == '(':
116                         self.pila.pop()
117                         break
118                     if not self.pilaVacia():
119                         self.lista.append(self.pila.pop())
120                     else:
121                         return None
122             else:
123                 if concatenacion:
124                     if self.pilaTope() == '^':
125                         while not self.pilaVacia():
126                             if self.pilaTope() != '(':
127                                 self.lista.append(self.pila.pop())
128                             else:
129                                 break
130                     self.pila.append('.')
131                 concatenacion = True
132                 self.lista.append(caracter)
133 
134         while not self.pilaVacia():
135             if self.pilaTope() != '(':
136                 self.lista.append(self.pila.pop())
137             else:
138                 return None
139 
140 
141 
142     def pilaVacia(self):
143         if len(self.pila) == 0:
144             return True
145         else:
146             return False
147 
148     def pilaTope(self):
149         if not self.pilaVacia():
150             return self.pila[len(self.pila)-1]
151         return None
152 
153     def validarCadena(self, cadena):
154         if type(cadena) is not str:
155             return False
156 
157         if len(cadena) == 0:
158             return False
159 
160         return True
```

3. Resultados

En esta sección se muestran los capturas de pantalla de los resultados del programa mostrado en la sección anterior. Se ingresaron tres diferentes expresiones regulares y se ingresaron diversas cadenas para realizar las pruebas.

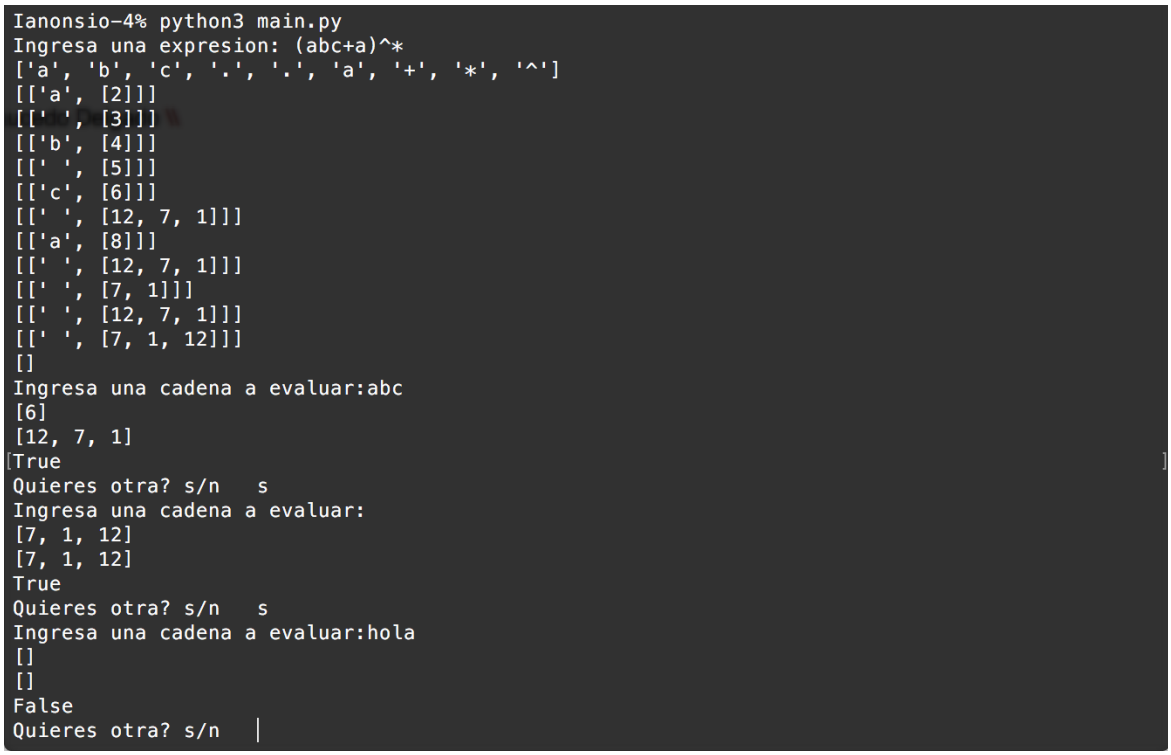


Figura 2: Cadenas siendo evaluadas para la expresión regular: $(abc + a)^*$.

```
[Ianonsio-4% python3 main.py
Ingresa una expresion: a+c+b
['a', 'c', 'b', '+', '+']
[['a', [2]]]
[[' ', [10]]]
[['c', [4]]]
[[' ', [10]]]
[['b', [6]]]
[[' ', [10]]]
[[' ', [5, 3]]]
[[' ', [10]]]
[[' ', [5, 3, 1]]]
[]
Ingresa una cadena a evaluar:a
[2]
[10]
True
Quieres otra? s/n   s
Ingresa una cadena a evaluar:b
[6]
[10]
True
Quieres otra? s/n   s
Ingresa una cadena a evaluar:f
[]
[]
False
Quieres otra? s/n
```

Figura 3: Cadenas siendo evaluadas para la expresión regular: $a + c + b$.

```
Ingresa una expresion: ab+(c+d)^*
['a', 'b', '.', 'c', 'd', '+', '*', '^', '+']
[['a', [2]]]
[[' ', [3]]]
[['b', [4]]]
[[' ', [14]]]
[['c', [6]]]
[[' ', [14, 7, 5]]]
[['d', [8]]]
[[' ', [14, 7, 5]]]
[[' ', [7, 5]]]
[[' ', [14, 7, 5]]]
[[' ', [7, 5, 14]]]
[[' ', [14]]]
[[' ', [7, 5, 14, 1]]]
[]
Ingresa una cadena a evaluar:
[7, 5, 14, 1]
[7, 5, 14, 1]
True
Quieres otra? s/n   s
Ingresa una cadena a evaluar:ab
[4]
[14]
True
Quieres otra? s/n   s
Ingresa una cadena a evaluar:cccccc
[6]
[14, 7, 5]
True
```

Figura 4: Cadenas siendo evaluadas para la expresión regular: $ab + (c + d)^*$.

4. Conclusiones

Las expresiones regulares, al igual que los autómatas nos sirven para modelar lenguajes formales. La ventaja que estas nos ofrecen sobre los autómatas, es que las expresiones regulares nos brindan una descripción algebráica del lenguaje.

Las expresiones regulares, pueden describir los mismos lenguajes que varios tipos de autómata. Gracias a esto, es posible pasar de una expresión regular a un autómata finito no determinista con transiciones epsilón y viceversa.

En esta práctica, realizamos el análisis de expresiones regulares y nos pudimos dar cuenta que estas, son una buena manera de describir el lenguaje de entrada de muchos tipos de aplicaciones que procesen texto, en este caso, un compilador.

5. Referencias

[1] *Compilers: principles, techniques and tools*. 2001, ch. 2.

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