Instituto Politécnico Nacional
“Escuela Superior de Computo”
Ingeniería en Sistemas Computacionales

PRÁCTICA 1

“GENERADOR DE AUTÓMATAS FINITOS”

UNIDAD DE APRENDIZAJE:

“COMPILADORES”

PROFESOR:

HERNANDEZ OLVERA LUIS ENRIQUE

ALUMNO:

LOPEZ HERNANDEZ DAVID

NO. BOLETA:

2018631531

GRUPO

3CV7

1/11/2020

Contenido

Objetivo de la práctica:

El estudiante pondrá en práctica sus conocimientos de los temas vistos en clase de autómatas finitos.

Instrucciones

La especificación de la práctica se dio en clase virtual, junto a estas instrucciones se anexarán 3 archivos que corresponden a 3 autómatas finitos los cuales deberán utilizar para probar su programa y documentar en el reporte los resultados para cada uno

Valores de entrada

- AF1.txt
  - aaab
  - abababab
- AF2.txt
  - ab
  - aaab
  - abab
- AF3.txt
  - a.aa
  - +a.a
  - +aa.aa

Introducción

QUE ES UN AUTOMATA FINITO

Un autómata finito (AF) o máquina de estado finito es un modelo computacional que realiza cómputos en forma automática sobre una entrada para producir una salida.

Este modelo está conformado por un alfabeto, un conjunto de estados finito, una función de transición, un estado inicial y un conjunto de estados finales. Su funcionamiento se basa en una función de transición, que recibe a partir de un estado inicial una cadena de caracteres pertenecientes al alfabeto (la entrada), y que va leyendo dicha cadena a medida que el autómata se desplaza de un estado a otro, para finalmente detenerse en un estado final o de aceptación, que representa la salida.

La finalidad de los autómatas finitos es la de reconocer lenguajes regulares, que corresponden a los lenguajes formales más simples según la Jerarquía de Chomsky.

AUTÓMATAS FINITOS Y EXPRESIONES REGULARES

Los lenguajes descritos por expresiones regulares son los lenguajes reconocidos por los autómatas finitos. Existe un algoritmo para convertir una expresión regular en el autómata finito no determinístico correspondiente. El algoritmo construye a partir de la expresión regular un autómata con transiciones vacías, es decir un autómata que contiene arcos rotulados con ε. Luego este autómata con transiciones vacías se puede convertir en un autómata finito sin transiciones vacías que reconoce el mismo lenguaje.

- Dada una expresión regular existe un autómata finito capaz de reconocer el lenguaje que ésta define.

- Recíprocamente, dado un autómata finito, se puede expresar mediante una expresión regular del lenguaje que reconoce.

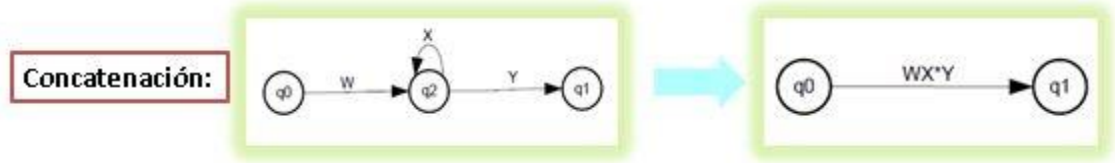
Conversión de un AFD en una expresión regular mediante la eliminación de estados

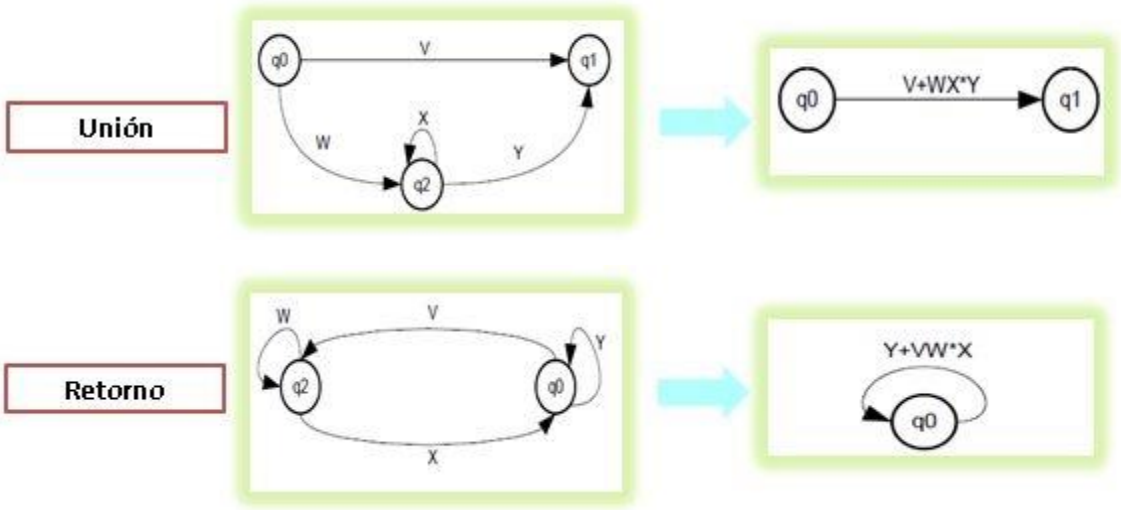
En este texto vamos a ver uno de los métodos que se usan para transformar autómatas finitos deterministas en expresiones regulares, el método de eliminación de estados.

Cuando tenemos un autómata finito, determinista o no determinista, podemos considerar que los símbolos que componen a sus transiciones son expresiones regulares. Cuando eliminamos un estado, tenemos que reemplazar todos los caminos que pasaban a través de él como transiciones

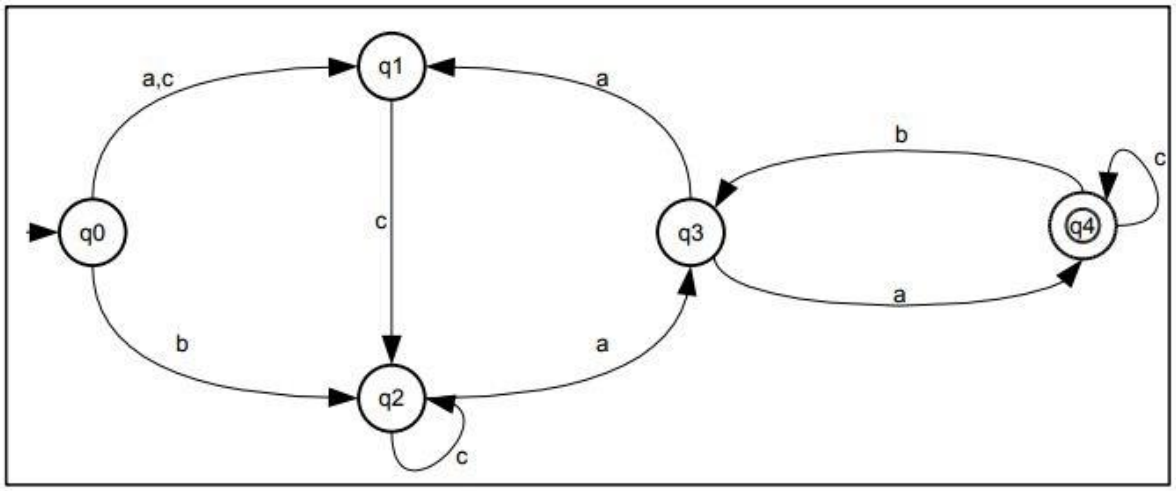
directas que ahora se realizan con el ingreso de expresiones regulares, en vez de con símbolos.

Los casos bases son los siguientes:



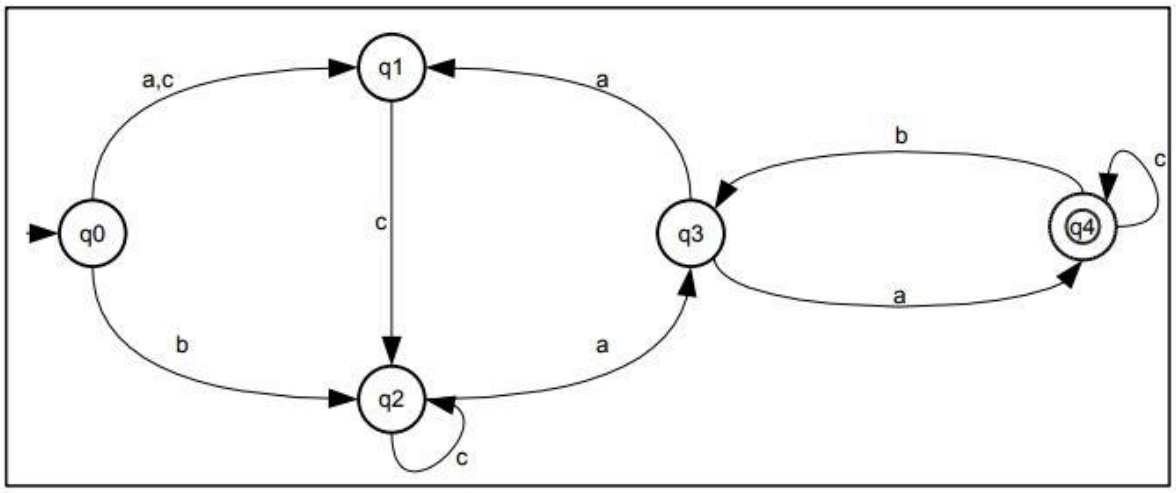


Ejemplo



1. Transformar todas aquellas transiciones que contemplan más de un símbolo en expresiones regulares del tipo “R+P.

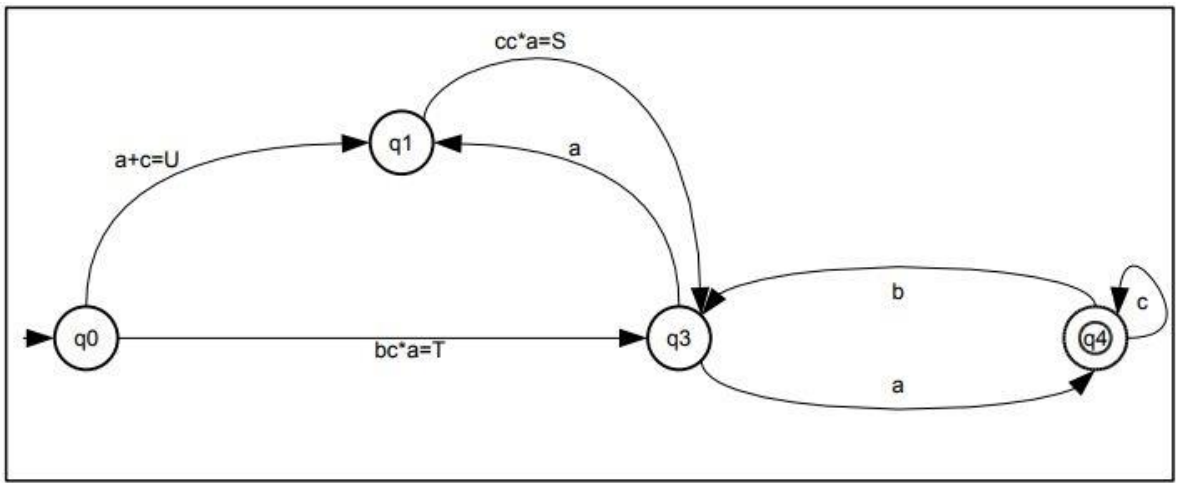
Por lo tanto, la transición q_0 a q_1, queda como “a+c”. Todas las demás permanecen iguales, por ser símbolos.



Se procede a eliminar el estado q_2. Verificar todos los caminos que atraviesan q_2:

- q_0 puede llegar a q_3 pasando por q_2.
- q_1 puede llegar a q_3 pasando por q_2.

Al **eliminar es estado q2** las transiciones del AFD quedan:



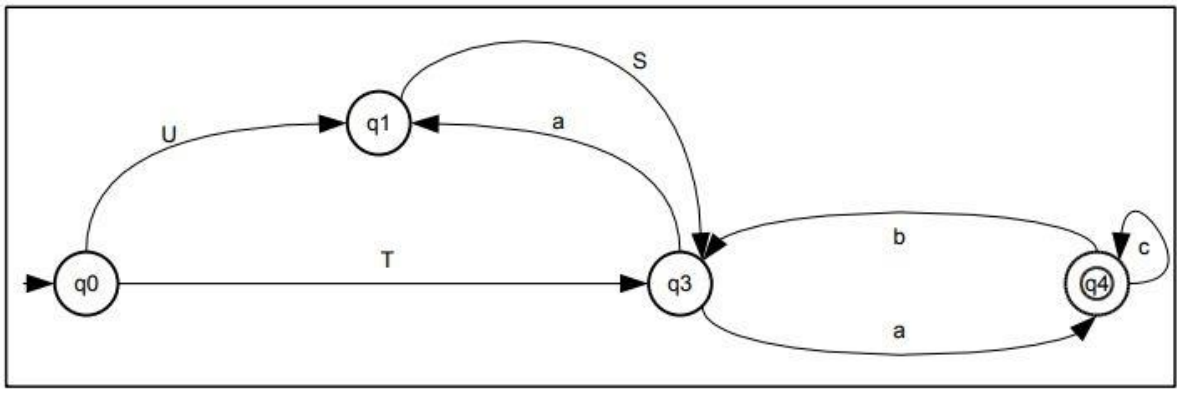
1) Para llegar a q_3, q_0 primero debe pasar por q_2 usando el símbolo “b”, luego puede pasar repetidas veces por q_2 (o ninguna vez) usando cero, uno o varios símbolos “c” y, por último, pasa de q_2 a q_3 con el símbolo “a”. Esto queda expresado con la expresión regular T = bc*a

2) Para pasar de q_1 a q_3, primero se debe llegar a q_2 con una “c”. Después de esto, se puede volver repetidas veces a q_2, usando el símbolo “c”.

Se puede pasar cero, una o muchas veces por q_2 de esta forma.

Como último paso, se debe llegar desde q_2 a q_3 con una “a”. Al concatenar estos tres caminos se obtiene:

S = cc*a



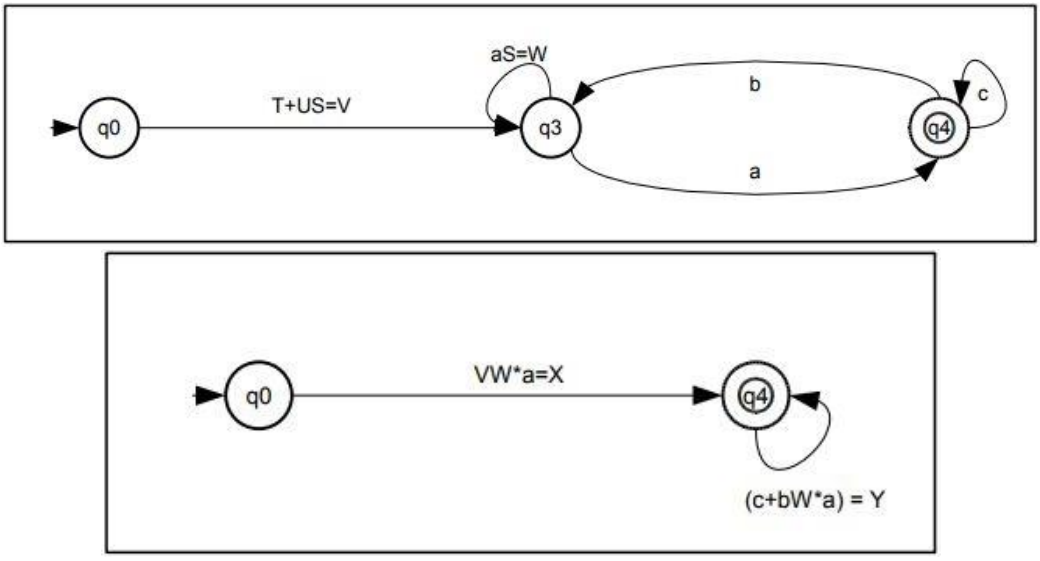
Al hacer las simplificaciones correspondientes, sólo se dejan las expresiones regulares resumidas por las letras mayúsculas (incluyendo a+c = U), y el autómata queda:

Eliminar q_1. Las transiciones afectadas son:

1) De q_0 a q_3, que puede hacerlo a través de q_1 o directamente, usando T. La expresión regular que va de q_0 a q_3 a través de q_1 es US y la expresión que va directamente de q_0 a q3 es T. Como puede ser una o la otra, como resultado queda la siguiente expresión: V = T + US

2) De q_3 a q_3, a través de q_1. Esto es: Primero llega a q_1 con una “a”, y luego pasa de q_1 a q_3, usando S. La expresión resultante es:

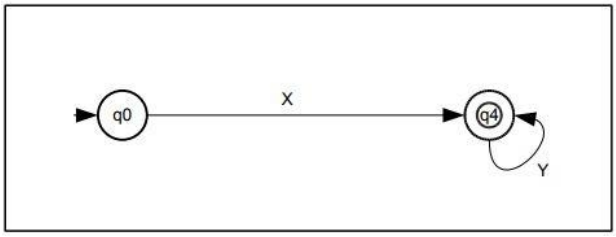
W = aS



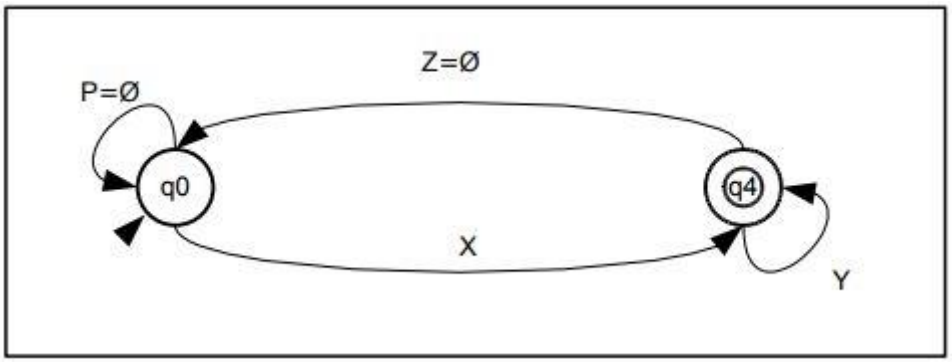
Eliminar q_3. Las transiciones afectadas son:

1) De q_0 a q_4, pasando por q_3: Primero va de q_0 a q_3 usando la expresión V. Luego de q_3 a q_3 repetidas veces, usando W. Por último, pasa de q_3 a q_4 usando una "a". La expresión regular queda : X = V.W*.a

2) De q_4 a q_4, pasando por q_3 o directamente por q4 con una "c". Para pasar a través de q_3, primero va de q_4 a q_3 con una "b". Luego de q_3 a q_3 usando W. Y, por último, pasa de q_3 a q_4 con una "a". Como puede ir desde q_4 a q_4 usando la "c" o través de q_3, la expresión regular que queda es: Y = (c + bW*a)



Al dejar las Expresiones Regulares de forma resumida, el último autómata queda:



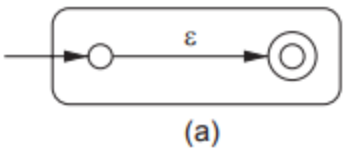
Completando las transiciones vacías (de q_4 a q_1 y de q_1 a q_1) con las expresiones regulares vacías Z y P

De esta forma, se puede generar la expresión regular final a partir de la fórmula

(P + XY*Z)*XY*

Expresión Final = (P + XY*Z)*XY* = (Ø* + XY*Ø)* XY* = (ε+Ø)*XY* = ε*XY* = XY*

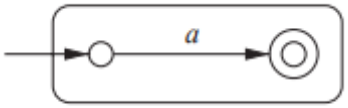
Conversión de una expresión regular a un autómata



En la parte (a) vemos cómo se maneja la expresión ε. Puede verse fácilmente que el lenguaje del autómata es {ε}, ya que el único camino desde el estado inicial a un estado de aceptación está etiquetado con ε

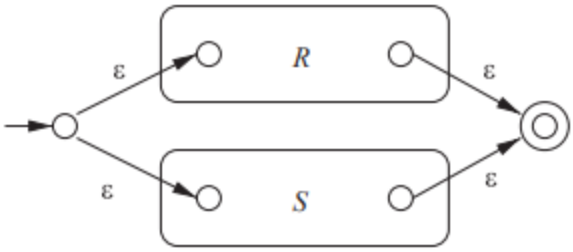
(b)

La parte (b) muestra la construcción de /0. Claramente, no existen caminos desde el estado inicial al de aceptación, por lo que /0 es el lenguaje de este autómata.



(c)

La parte (c) proporciona el autómata que reconoce una expresión regular a. Evidentemente, el lenguaje de este autómata consta de una cadena a, que es también L(a).

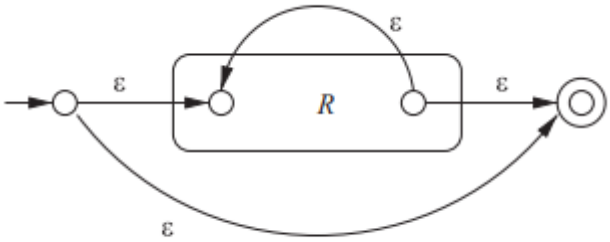


(a)

La expresión es de la forma R + S para dos expresiones R y S más pequeñas. Por tanto, el lenguaje del autómata de la (a) es L(R) ∪ L(S).

(b)

La expresión es de la forma RS para expresiones R y S más pequeñas. Por tanto, los caminos en el autómata de la (b) son todos y sólo los etiquetados con cadenas pertenecientes a L(R)L(S).



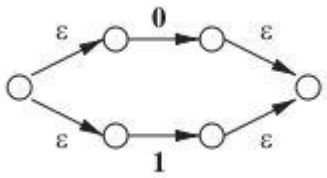
(c)

La expresión es de la forma R∗ para una expresión R más pequeña. Dicho camino acepta ε, que pertenece a L(R∗) sin importar qué expresión sea R.

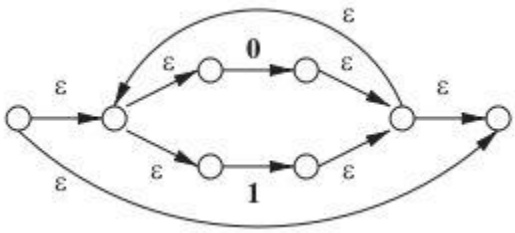
Ejemplo

Deseamos convertir la expresión regular (0+ 1)∗1(0 + 1) en un AFN-ε.

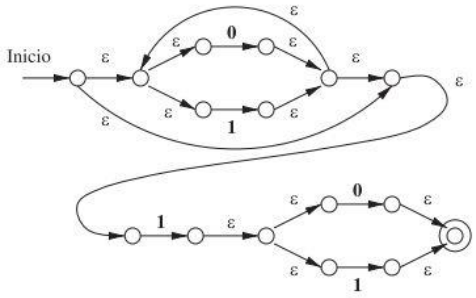
El **primer paso** consiste en construir un autómata para 0 + 1.



Luego Construiremos (0+1)* a partir de (0+1), Colocando una transición vacia en el nodo final de 0+1 devolviendome al inicio para las veces que se repita el (0+1), luego colocaremos un nodo anterior al nodo inicial de (0+1) y añadiremos una transición vacia entre este nodo y un nodo adelante del nodo final para cuando (0+1)^0



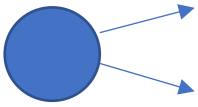
Finalmente colocaremos una transición 1 que conecte (0+1)* con (0+1) dando como resultado



Desarrollo

PROBLEMAS A LOS QUE ME ENFRENTE:

Durante el desarrollo de esta práctica, no me enfrente problemas tan complicados ya que en la unidad de aprendizaje de teoría computacional realice una práctica bastante similar, a diferencia que en esta materia resolví el problema con una estructura de datos en c, fue un desarrollo bastante tardado ya que el profe quería algo más visual, y esta fue mi solución. Lo que hice fue crear algoritmos que recorrieran la estructura de datos completa (lo cual consumió bastante tiempo de desarrollo), pero en si no represento una dificultad tan alta. Debo agregar que el alfabeto de esta práctica estaba reducido solo a 2 cosas en el alfabeto y era un autómata determinista.



En el caso para esta práctica use las funciones delta, para realizar los recorridos, me auxilie de 4 listas

```
#5 tupla de componentes de un AF
Estados = []    //En este se guardaban los estados que existian
Alfabeto = []   //Aquí los componentes del alfabeto
EI = ""         //El estado inicial
EF = []         //Los estados finales , era una lista
TablaTrans= []  //Las funciones delta en formato de arreglo
```

Use un auxiliar llamado estado actual

```
EA = str(EI)#ESTADO ACTUAL
```

Tomaba el estado inicial y lo almacenaba en estado actual, en seguida manda a llamar a una función recursiva que es la siguiente:

```
def ProcesaCadena(Cadena,Estados,Alfabeto,EI,EF,TablaT,EA,Camino,I
ndiceCadena):
    global CaminosValidos
    if IndiceCadena>(len(Cadena)-1):
        if EA in EF:
            CaminosValidos.append(Camino[::])
        return
    else:
        c=Cadena[IndiceCadena]
        for f in TablaT:
            if c == f[1] and EA == f[0]:
                Camino[IndiceCadena]=f[0]+"-"+f[1]+"->"+f[2]
```

```
                ProcesaCadena(Cadena,Estados,Alfabeto,EI,EF,TablaT
,f[2],Camino,IndiceCadena+1)
```

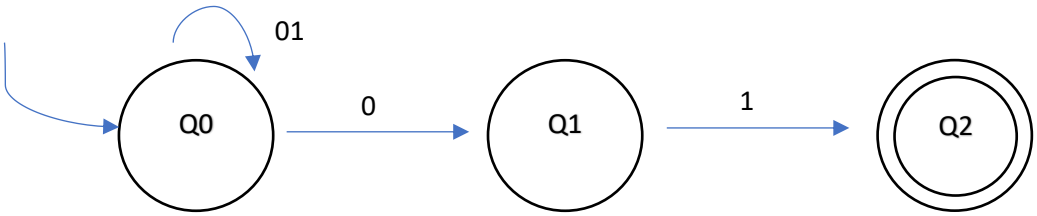
Esta función recorre todos los caminos posibles para la cadena que está en lectura y se llama a si misma aumentando el índice de la lectura de la cadena que se está analizando, llamándose a si misma de forma recursiva, si encuentra un camino valido almacena una copia en la lista caminos válidos.

Crea un tipo árbol de la función, ejemplo:

Cadena = 0 1 0 1

Y si existen ramificaciones explora todas las posibilidades.

Un ejemplo pondremos el autómata que termina en 01:



Entonces va analizando todas las ramificaciones

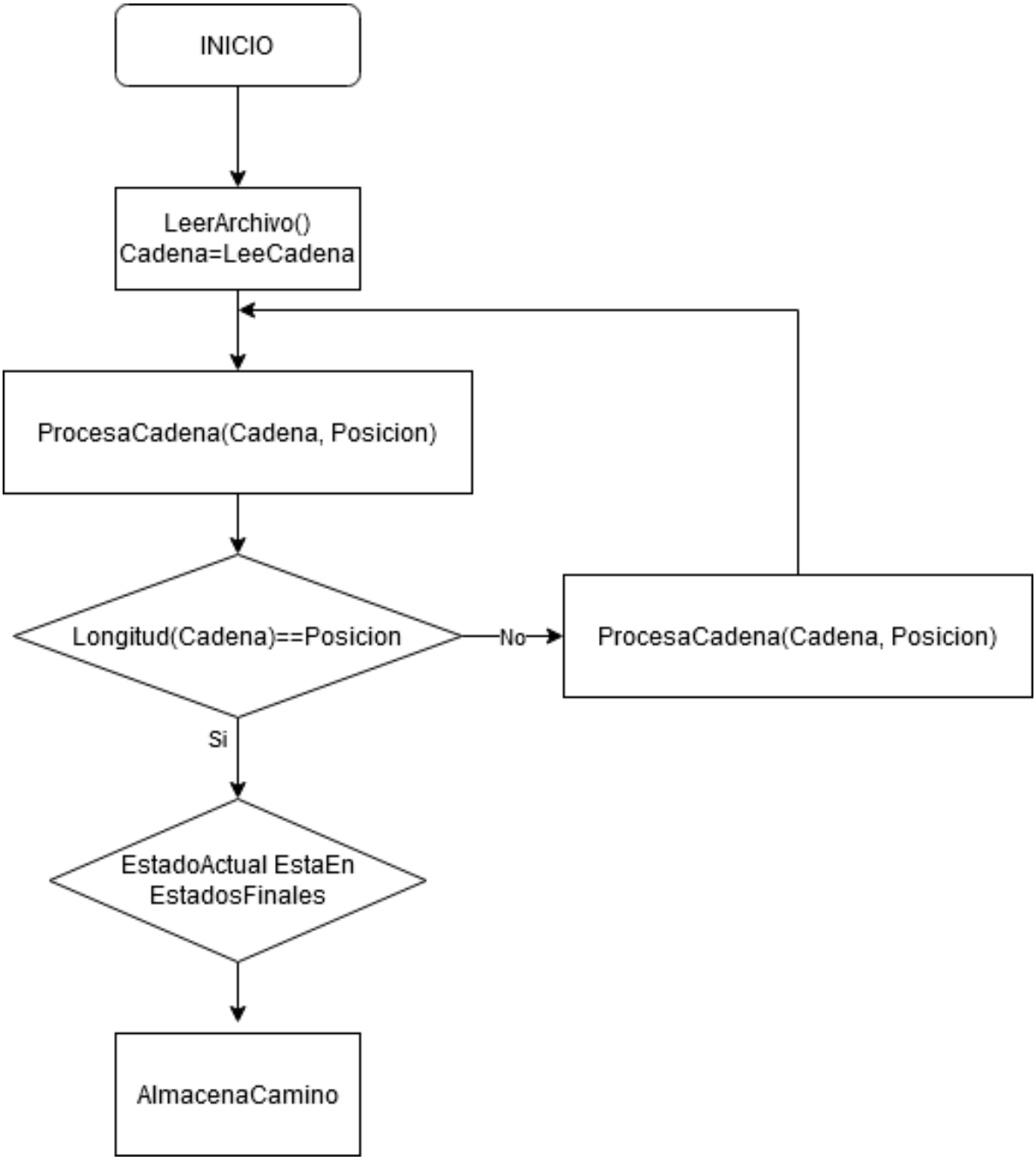
0 1 0 1

[Q0-0->Q1], [Q1-1->Q2] SIN CAMINOS

[Q0-0->Q0], [Q0-1->Q0], [Q0-0->Q0], [Q0-1->Q0] NO ES ESTADO DE ACEPTACION

[Q0-0->Q0], [Q0-1->Q0], [Q0-0->Q1], [Q1-1->Q2] ACEPTADA

DIAGRAMA DE FLUJO



CODIGO

```
def LeeArchivo(Estados,Alfabeto,EI,EF,TablaTrans,nombreArchivo):
    with open(nombreArchivo) as fname:
        LineasArchivo = [i.strip('\n') for i in fname.readlines()]
        #Separa el archivo de entrada por lineas
        Estados=LineasArchivo[0].split(',')#Estados
        Alfabeto=LineasArchivo[1].split(',')#Alfabeto
        EI=str(LineasArchivo[2])#Estado Inicial
        EF=LineasArchivo[3].split(',')#Estados Finales
        for x in range(4, len(LineasArchivo), 1):#Transiciones
            temp=[LineasArchivo[x]]
            TablaTrans.append(temp[0].split(','))
    return Estados,Alfabeto,EI,EF,TablaTrans

def ProcesaCadena(Cadena,Estados,Alfabeto,EI,EF,TablaT,EA,Camino,I
ndiceCadena):
    global CaminosValidos
    if IndiceCadena>(len(Cadena)-1):
        if EA in EF:
            CaminosValidos.append(Camino[::])
        return
    else:
        c=Cadena[IndiceCadena]
        for f in TablaT:
            if c == f[1] and EA == f[0]:
                Camino[IndiceCadena]=f[0]+"-"+f[1]+"->"+f[2]
                ProcesaCadena(Cadena,Estados,Alfabeto,EI,EF,TablaT
,f[2],Camino,IndiceCadena+1)

print(">>>>>>>>>>>>>>>>>>>Programa AFN<<<<<<<<<<<<<<<<<<")#Titulo
#5 tupla de componentes de un AF
Estados = []
Alfabeto = []
EI = ""
EF = []
TablaTrans= []
CaminosValidos=[]#Almacenara los caminos
#Leer Archivo
nombreArchivo="AF.txt"
```

```
Estados,Alfabeto,EI,EF,TablaTrans=LeeArchivo(Estados,Alfabeto,EI,E
F,TablaTrans,nombreArchivo)#Manda a leer el archivo
EA = str(EI)#ESTADO ACTUAL
print("Estados:"+str(Estados))
print("Alfabeto:"+str(Alfabeto))
print("Estado inicial:"+EI)
print("Estados Finales:"+str(EF))
print("Tabla de transiciones")
for x in TablaTrans:
    print("\t"+str(x))
#Lectura de cadena
print("Introduce una cadena")
Cadena=input()
print("Cadena:"+Cadena)
Camino=[None]*len(Cadena)
ProcesaCadena(Cadena,Estados,Alfabeto,EI,EF,TablaTrans,EA,Camino,0
)
#FINAL ACEPTACION O RECHAZO DE CADENA
if len(CaminosValidos)>0:
    print(">>>>Cadena Aceptada<<<<")
    print("Caminos validos para la cadena :"+Cadena)
    for x in CaminosValidos:
        print(x)
else:
    print(">>>>Cadena Rechzada<<<<")
```

VALORES DE ENTRADA

- AF1.txt

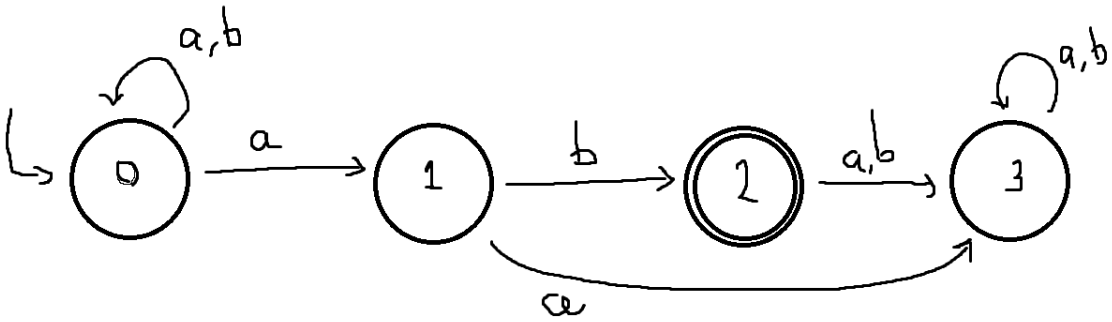


Imagen AF1

- aaab ✓ 1

```
>>>>>>>>>>>>>>>>>>>Programa AFN<<<<<<<<<<<<<<<<<<<
Estados:['0', '1', '2', '3']
Alfabeto:['a', 'b']
Estado inicial:0
Estados Finales:['2']
Tabla de transiciones
        ['0', 'a', '0']
        ['0', 'b', '0']
        ['0', 'a', '1']
        ['1', 'b', '2']
        ['1', 'a', '3']
        ['2', 'a', '3']
        ['2', 'b', '3']
        ['3', 'a', '3']
        ['3', 'b', '3']
Introduce una cadena
aaab
Cadena:aaab
>>>>Cadena Aceptada<<<<
Caminos validos para la cadena :aaab
['0-a->0', '0-a->0', '0-a->1', '1-b->2']
```

Captura: AF1.txt

Descripción:
Podemos observar el autómata en la imagen **AF1** y al cargar el autómata nos dice los la 5 tupla que lo componen, después nos pide una cadena la cual procesara nuestra función mencionada anteriormente **procesaCadena().** La cual calcula todos los caminos posibles para la cadena y nos almacena los caminos válidos.

- ababab ✓ 1

```
>>>>>>>>>>>>>>>>>>>Programa AFN<<<<<<<<<<<<<<<<<<<
Estados:['0', '1', '2', '3']
Alfabeto:['a', 'b']
Estado inicial:0
Estados Finales:['2']
Tabla de transiciones
        ['0', 'a', '0']
        ['0', 'b', '0']
        ['0', 'a', '1']
        ['1', 'b', '2']
        ['1', 'a', '3']
        ['2', 'a', '3']
        ['2', 'b', '3']
        ['3', 'a', '3']
        ['3', 'b', '3']
Introduce una cadena
ababab
Cadena:ababab
>>>>Cadena Aceptada<<<<
Caminos validos para la cadena :ababab
['0-a->0', '0-b->0', '0-a->0', '0-b->0', '0-a->1', '1-b->2']
```

Captura: AF1.txt

Descripción:
Podemos observar el autómata en la imagen **AF1** y al cargar el autómata nos dice los la 5 tupla que lo componen, después nos pide una cadena la cual procesara nuestra función mencionada anteriormente **procesaCadena().** La cual calcula todos los caminos posibles para la cadena y nos almacena los caminos validos.

- AF2.txt

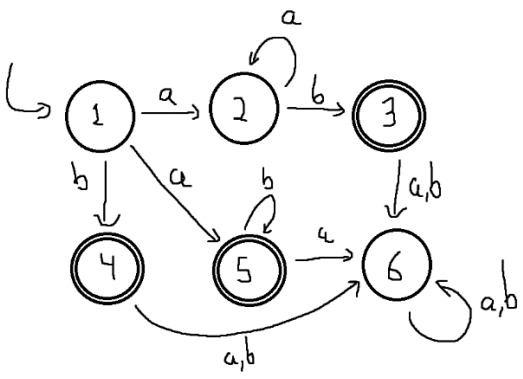
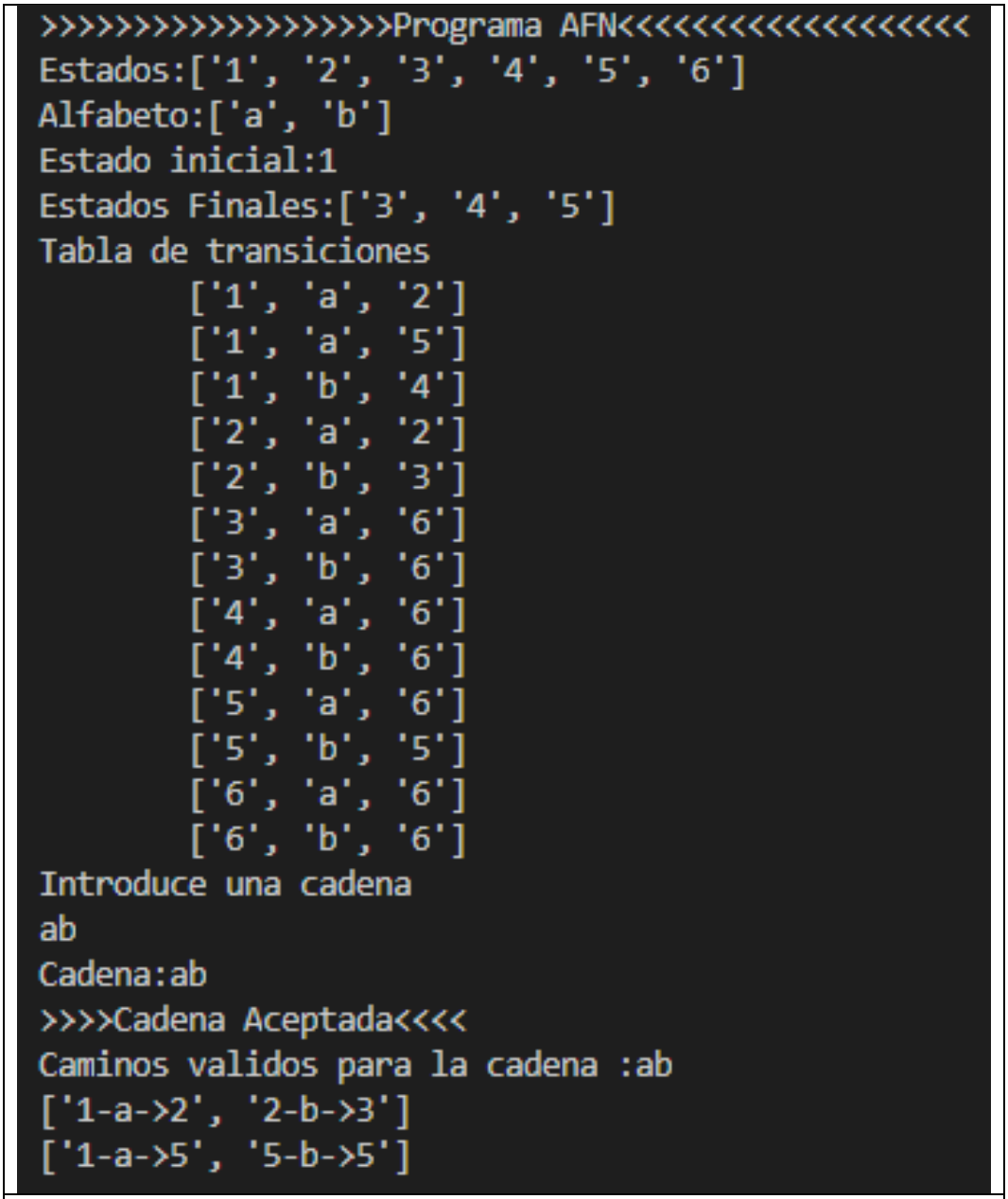


Imagen AF2

- ab ✓ 2



Captura: AF2.txt

Descripción:
Podemos observar el autómata en la imagen **AF2** y al cargar el autómata nos dice los la 5 tupla que lo componen, después nos pide una cadena la cual procesara nuestra función mencionada anteriormente **procesaCadena().** La cual calcula todos los caminos posibles para la cadena y nos almacena los caminos válidos.
En este caso podemos observar que solo tiene dos caminos validos que terminan en los dos estados de acepción 5 Y 3 .

- aaab ✓ 1

```
>>>>>>>>>>>>>>>>>>>Programa AFN<<<<<<<<<<<<<<<<<<<
Estados:['1', '2', '3', '4', '5', '6']
Alfabeto:['a', 'b']
Estado inicial:1
Estados Finales:['3', '4', '5']
Tabla de transiciones
        ['1', 'a', '2']
        ['1', 'a', '5']
        ['1', 'b', '4']
        ['2', 'a', '2']
        ['2', 'b', '3']
        ['3', 'a', '6']
        ['3', 'b', '6']
        ['4', 'a', '6']
        ['4', 'b', '6']
        ['5', 'a', '6']
        ['5', 'b', '5']
        ['6', 'a', '6']
        ['6', 'b', '6']
Introduce una cadena
aaab
Cadena:aaab
>>>>Cadena Aceptada<<<<
Caminos validos para la cadena :aaab
['1-a->2', '2-a->2', '2-a->2', '2-b->3']
```

Captura: AF2.txt

Descripción:
Podemos observar el autómata en la imagen **AF2** y al cargar el autómata nos dice los la 5 tupla que lo componen, después nos pide una cadena la cual procesara nuestra función mencionada anteriormente **procesaCadena().** La cual calcula todos los caminos posibles para la cadena y nos almacena los caminos válidos.

- abab x

```
>>>>>>>>>>>>>>>>>>>>Programa AFN<<<<<<<<<<<<<<<<<<<<
Estados:['1', '2', '3', '4', '5', '6']
Alfabeto:['a', 'b']
Estado inicial:1
Estados Finales:['3', '4', '5']
Tabla de transiciones
        ['1', 'a', '2']
        ['1', 'a', '5']
        ['1', 'b', '4']
        ['2', 'a', '2']
        ['2', 'b', '3']
        ['3', 'a', '6']
        ['3', 'b', '6']
        ['4', 'a', '6']
        ['4', 'b', '6']
        ['5', 'a', '6']
        ['5', 'b', '5']
        ['6', 'a', '6']
        ['6', 'b', '6']
Introduce una cadena
abab
Cadena:abab
>>>>Cadena Rechzada<<<<
```

Captura: AF2.txt

Descripción:
Podemos observar el autómata en la imagen **AF2** y al cargar el autómata nos dice los la 5 tupla que lo componen, después nos pide una cadena la cual procesara nuestra función mencionada anteriormente **procesaCadena().** La cual calcula todos los caminos posibles para la cadena y nos almacena los caminos válidos.
En este caso no encontró un camino valido para la cadena abab y lo podemos observar viendo el autómata en la imagen **AF2.**

- AF3.txt

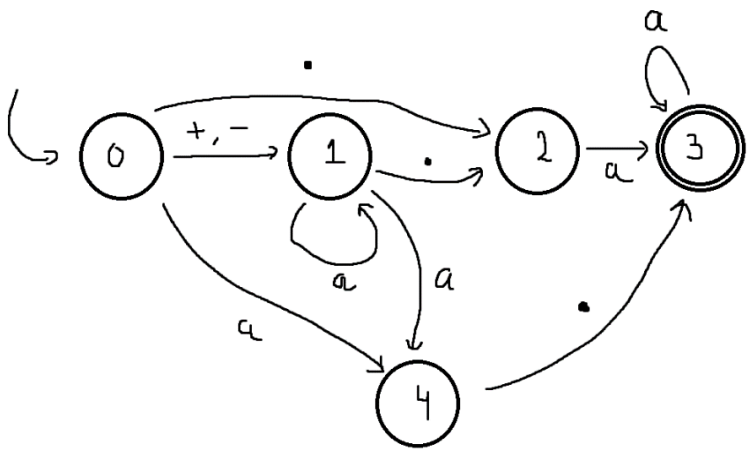


Imagen AF2

- a.aa ✓ 1

```
>>>>>>>>>>>>>>>>>>>Programa AFN<<<<<<<<<<<<<<<<<<<
Estados:['0', '1', '2', '3', '4']
Alfabeto:['a', '+', '-', '.']
Estado inicial:0
Estados Finales:['3']
Tabla de transiciones
       ['0', '+', '1']
       ['0', '-', '1']
       ['0', '.', '2']
       ['0', 'a', '4']
       ['1', '.', '2']
       ['1', 'a', '1']
       ['1', 'a', '4']
       ['2', 'a', '3']
       ['3', 'a', '3']
       ['4', '.', '3']
Introduce una cadena
a.aa
Cadena:a.aa
>>>>Cadena Aceptada<<<<
Caminos validos para la cadena :a.aa
['0-a->4', '4-.->3', '3-a->3', '3-a->3']
```

Captura: AF3.txt

Descripción:
Podemos observar el autómata en la imagen **AF3** y al cargar el autómata nos dice los la 5 tupla que lo componen, después nos pide una cadena la cual procesara nuestra función mencionada anteriormente **procesaCadena().** La cual calcula todos los caminos posibles para la cadena y nos almacena los caminos válidos.

- +a.a ✓ 2

```
>>>>>>>>>>>>>>>>>>>Programa AFN<<<<<<<<<<<<<<<<<<<
Estados:['0', '1', '2', '3', '4']
Alfabeto:['a', '+', '-', '.']
Estado inicial:0
Estados Finales:['3']
Tabla de transiciones
        ['0', '+', '1']
        ['0', '-', '1']
        ['0', '.', '2']
        ['0', 'a', '4']
        ['1', '.', '2']
        ['1', 'a', '1']
        ['1', 'a', '4']
        ['2', 'a', '3']
        ['3', 'a', '3']
        ['4', '.', '3']
Introduce una cadena
+a.a
Cadena:+a.a
>>>>Cadena Aceptada<<<<
Caminos validos para la cadena :+a.a
['0-+->1', '1-a->1', '1-.->2', '2-a->3']
['0-+->1', '1-a->4', '4-.->3', '3-a->3']
```

Captura: AF3.txt

Descripción:
Podemos observar el autómata en la imagen **AF3** y al cargar el autómata nos dice los la 5 tupla que lo componen, después nos pide una cadena la cual procesara nuestra función mencionada anteriormente **procesaCadena().** La cual calcula todos los caminos posibles para la cadena y nos almacena los caminos válidos.

- +aa.aa ✓ 2

```
>>>>>>>>>>>>>>>>>>>Programa AFN<<<<<<<<<<<<<<<<<<<
Estados:['0', '1', '2', '3', '4']
Alfabeto:['a', '+', '-', '.']
Estado inicial:0
Estados Finales:['3']
Tabla de transiciones
        ['0', '+', '1']
        ['0', '-', '1']
        ['0', '.', '2']
        ['0', 'a', '4']
        ['1', '.', '2']
        ['1', 'a', '1']
        ['1', 'a', '4']
        ['2', 'a', '3']
        ['3', 'a', '3']
        ['4', '.', '3']
Introduce una cadena
+aa.aa
Cadena:+aa.aa
>>>>Cadena Aceptada<<<<
Caminos validos para la cadena :+aa.aa
['0-+->1', '1-a->1', '1-a->1', '1-.->2', '2-a->3', '3-a->3']
['0-+->1', '1-a->1', '1-a->4', '4-.->3', '3-a->3', '3-a->3']
```

Captura: AF3.txt

Descripción:
Podemos observar el autómata en la imagen **AF3** y al cargar el autómata nos dice los la 5 tupla que lo componen, después nos pide una cadena la cual procesara nuestra función mencionada anteriormente **procesaCadena().** La cual calcula todos los caminos posibles para la cadena y nos almacena los caminos válidos.

Conclusión

Fue una práctica con una implementación algo sencilla, aunque presente sus correspondientes inconvenientes a la hora de implementarlo, pero nada muy complejo. Me base en las funciones delta para procesar la cadena.

Como mencione anteriormente en la asignatura de teoría computacional programe un procesador de cadenas para autómatas finitos los cuales solo tuvieran un Alfabeto de 0 y 1, pero lo implemente en C con estructuras de datos, ya que me parecía una buena práctica, para no olvidar lo que vi en la respectiva materia. En esa práctica presente dificultades un poco más complejas ya que el profe quería una representación más visual de cómo se ejecuta el análisis de la cadena y también me dio bastante práctica, por lo que al enfrentar este problema no fue tan complicado.

Algo muy importante es el lenguaje de programación que use, en este caso fue Python ya que la facilidad para procesar archivos de texto se reduce demasiado. Y así poder dividir el archivo de texto de una forma más sencilla. Otra cosa que me auxilio demasiado fueron las listas ya que, si las usamos de Buena manera, se puede sacar mucho provecho a esto.

Un problema al que me pude enfrentar durante esta práctica es las direcciones de memoria con Python ya que al asignar mi lista a la lista de caminos vacíos siempre me dejaba todas igual ya que se pasan por referencia de memoria, entonces para solucionar esto use la siguiente notación

```
CaminosValidos.append(Camino[::])
```

Lo que se puede observar en amarrillo es lo que menciona, esto crea una copia de la lista en memoria, así podemos evitar problemas al agregar la lista a los caminos válidos.

Finalmente, quiero agregar que en general disfruto las practicas que representan retos a la hora de la implementación, ya que nos pone en un ambiente bastante interesante de como propondremos la solución.

Referencias

[1]2020. [Online]. Available: https://www.citethisforme.com/es/cite/sources/websiteautociteconfirm. [Accessed: 02- Nov- 2020]

[2]E. Viso Gurovich, *Introduccion a la Teoria de la Computacion*. UNAM, 2008, 2008, p. 304.

[3]P. Flores Suárez, *Compiladores: principios, técnicas y herramientas*, 1st ed. Pearson Educación, 1998, 2020, p. 820.