Reporte: Práctica 3

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

2 de octubre de 2017

Índice

1. Introducción

El transformar un autómata no determinista a uno determinista surge de la necesidad de modelar dicho autómata computacionalmente. Esto debido a que la implementación de un AFN debido a los múltiples movimientos que podría realizar en un estado con un sólo símbolo

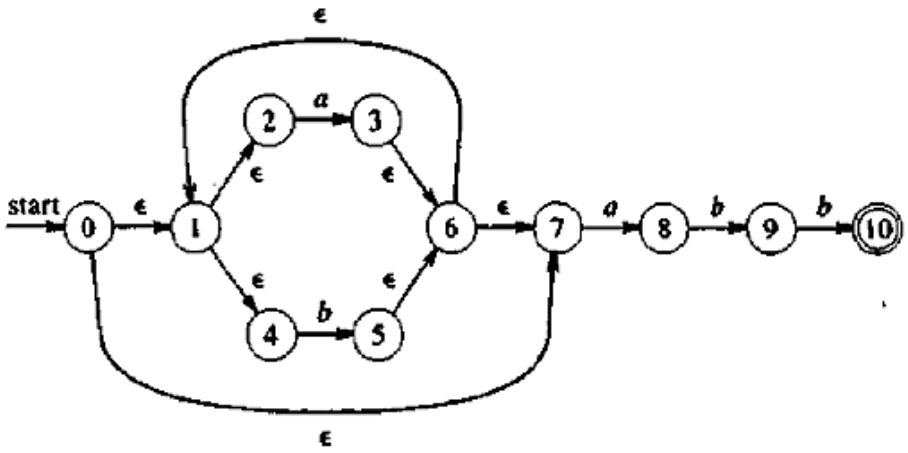

Figura 1: AFN que reconoce el lenguaje $(a|b)^*abb$.

Es por esto que se utiliza el siguiente algoritmo (construcciones de subconjuntos) para pasar de un AFN a un AFD [1]. Este algoritmo se basa en que cada estado en el AFD es igual a un subconjunto de estados del AFN. Para construir estos subconjuntos se utilizan las siguientes funciones.

- *cerradura-*$\epsilon(T)$. Esta función recibe un conjunto de estados T del AFN y retorna el conjunto de estados que se pueden obtener al hacer transiciones épsilon. Por ejemplo, aplicar esta función sobre el estado inicial del autómata de la figura 1 produciría lo siguiente: cerradura-$\epsilon(\{0\}) = \{0, 1, 2, 4, 7\}$.
- $mover(T, a)$. Regresa el conjunto de estados que se pueden obtener al realizar transiciones con el símbolo a sobre el conjunto de estados T del AFN. Un ejemplo de esto es que si se aplica esta función sobre el conjunto del ejemplo anterior se produciría lo siguiente: $\text{mover}(\{0, 1, 2, 4, 7\}, a) = \{3, 8\}$.

Este conjunto de funciones se ejecutaran las veces que sean necesarias hasta que se obtenga el nuevo conjunto de estados del AFD. Esto se ve de forma más clara en el siguiente pseudocódigo.

Algorithm 1 Obtención de subconjuntos

```
Agregar cerradura-ϵ(Inicial del AFN) a E
for cada e en E do
  for cada simbolo en el alfabeto del AFN do
    Agregar cerradura-ϵ(mover(e, s))
    Agregar transición e → cerradura-ϵ(mover(e, s)) con símbolo s
  end for
end for
El inicial del AFD es cerradura-ϵ(Inicial del AFN)
Los finales del AFD contienen finales del AFN
```

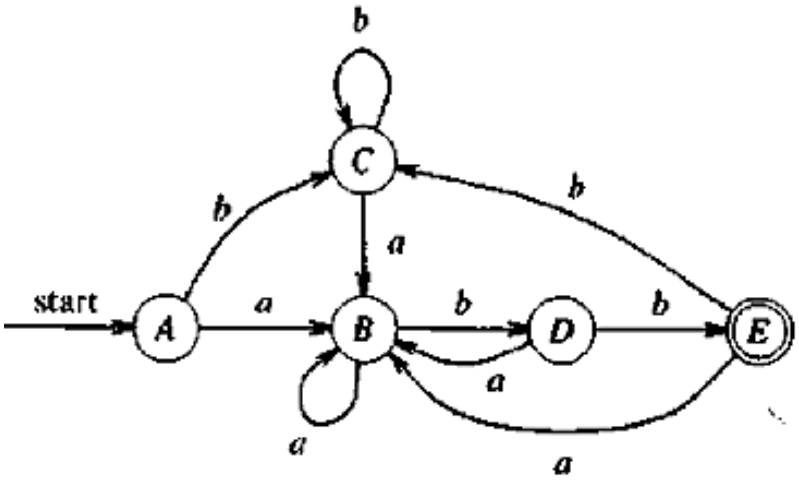

Figura 2: AFD que reconoce el mismo lenguaje que el AFN de la figura 1.

2. Desarrollo

La siguiente clase contiene los métodos necesarios para transformar un AFN en un AFD basados en el algoritmo anterior.

```
from automata.automatas import AFD

# Clase para poder etiquetar los nuesvos estados de manera mas facil
class Subconjunto:
  def __init__(self, estados, etiqueta):
    self.estados = estados
    self.etiqueta = etiqueta


class Transformacion:
  def __init__(self, AFN):
    self.lista = list()
    self.AFN = AFN
    self.AFD = AFD()
    self.etiqueta = 'A' # Para etiquetar los estados AFD

  def convertir_automata(self):
    self.AFD.alfabeto = self.AFN.alfabeto
    self.AFD.alfabeto.remove('e')
    estado = self.cerradura_epsilon(self.AFN.estado_inicial)
    actual = Subconjunto(estado, self.etiqueta)
    self.lista.append(actual)
    pendientes = list()
    pendientes.append(actual)
    agregar = True
    self.etiqueta = chr(ord(self.etiqueta) + 1)
    nuevo = None
    while len(pendientes) > 0:
      actual = pendientes.pop()
      for simbolo in self.AFD.alfabeto:
        estados = self.cerradura_epsilon(self.mover(actual.estados, simbolo))
```

```
        agregar = True
        for i in self.lista:
          if i.estados == estados:
            agregar = False
            nuevo = i
            break
        if agregar:
          nuevo = Subconjunto(estados, self.etiqueta)
          pendientes.append(nuevo)
          self.lista.append(nuevo)
          self.etiqueta = chr(ord(self.etiqueta) + 1)
        print("Transicion %s -> %s con: %s" % (actual.estados, nuevo.estados, simbolo))
        self.AFD.agregar_transicion(actual.etiqueta, nuevo.etiqueta, simbolo)

    # Obtenemos los estados finales e inicial
    for elemento in self.lista:
      if self.AFN.estado_inicial in elemento.estados:
        self.AFD.estado_inicial = elemento.etiqueta
      for final in self.AFN.estados_finales:
        if final in elemento.estados:
          self.AFD.estados_finales.add(elemento.etiqueta)

  # recibe un solo elemento o un conjunto de estados
  def cerradura_epsilon(self, estados):
    estados_epsilon = set()
    aux = list()
    if type(estados) is set:
      aux.extend(estados)
    else:
      aux.append(estados)
    for estado in aux:
      for t in self.AFN.transiciones:
        if t.caracter == 'e' and t.actual == estado and (t.siguiente not in aux):
          aux.append(t.siguiente)

    estados_epsilon.update(aux)
    return estados_epsilon

  # Obtiene los siguientes estados con base en un simbolo y un conjunto de entrada
  def mover(self, estados, simbolo):
    estados_aux = set()
    for estado in estados:
      for transicion in self.AFN.transiciones:
        if simbolo != 'e' and estado == transicion.actual and transicion.caracter == simbolo:
          estados_aux.add(transicion.siguiente)
    return estados_aux
```

3. Resultados

Para realizar las pruebas se utilizo la siguiente función. En donde lo primero que se hace es crear un AFN basado en las clases creadas en prácticas anteriores para después transformarlo y realizar pruebas con diversas cadenas sobre este nuevo autómata determinista

```
def transformar_automata(self):
    # Creacion de un AFN
  print('Transformando AFN a AFD')
  automata = AFN()
```

```
#Se establece su alfabeto
automata.agregar_alfabeto({'a', 'b'})
# Agregamos los estados que tendra dicho automata
automata.agregar_estado(0)
automata.agregar_estado(1)
automata.agregar_estado(2)
automata.agregar_estado(3)
automata.agregar_estado(4)
automata.agregar_estado(5)
automata.agregar_estado(6)
automata.agregar_estado(7)
automata.agregar_estado(8)
automata.agregar_estado(9)
automata.agregar_estado(10)
# Se indican los estados finales y el inicial
automata.agregar_inicial(0)
automata.agregar_finales({10})
# Por ultimo agregamos las transiciones necesarias
automata.agregar_transicion(0, 1, 'e')
automata.agregar_transicion(1, 2, 'e')
automata.agregar_transicion(1, 4, 'e')
automata.agregar_transicion(0, 7, 'e')
automata.agregar_transicion(2, 3, 'a')

automata.agregar_transicion(4, 5, 'b')
automata.agregar_transicion(3, 6, 'e')
automata.agregar_transicion(5, 6, 'e')
automata.agregar_transicion(6, 1, 'e')
automata.agregar_transicion(6, 7, 'e')
automata.agregar_transicion(7, 8, 'a')
automata.agregar_transicion(8, 9, 'b')
automata.agregar_transicion(9, 10, 'b')

#Creamos un objeto de nuestra clase Transformacion y le
# indicamos el AFN con el que trabajara
transformer = Transformacion(automata)
print('Transformando el automata')
# A nuestra instancia de Transformacion le decimos que trasforme el automata
transformer.convertir_automata()
print()
# Por ultimo solo mostramos los componentes del AFD final
print('-Automata deterministico final:')
print("Estado inicial %s" % transformer.AFD.estado_inicial)
print("Estados finales %s" % transformer.AFD.estados_finales)
print("Transiciones")
for t in transformer.AFD.transiciones:
  print(t)

# Realizamos pruebas con este AFD, mediante el ingreso
# de cadenas en consola para su evaluacion
print('Evaluacion de cadenas:')
for n in range(5):
  cadena = input("-Ingresa una cadena: ")
  print("La cadena es:")
  if transformer.AFD.evaluar_cadena(cadena):
    print("Valida")
  else:
    print("No valida")
```

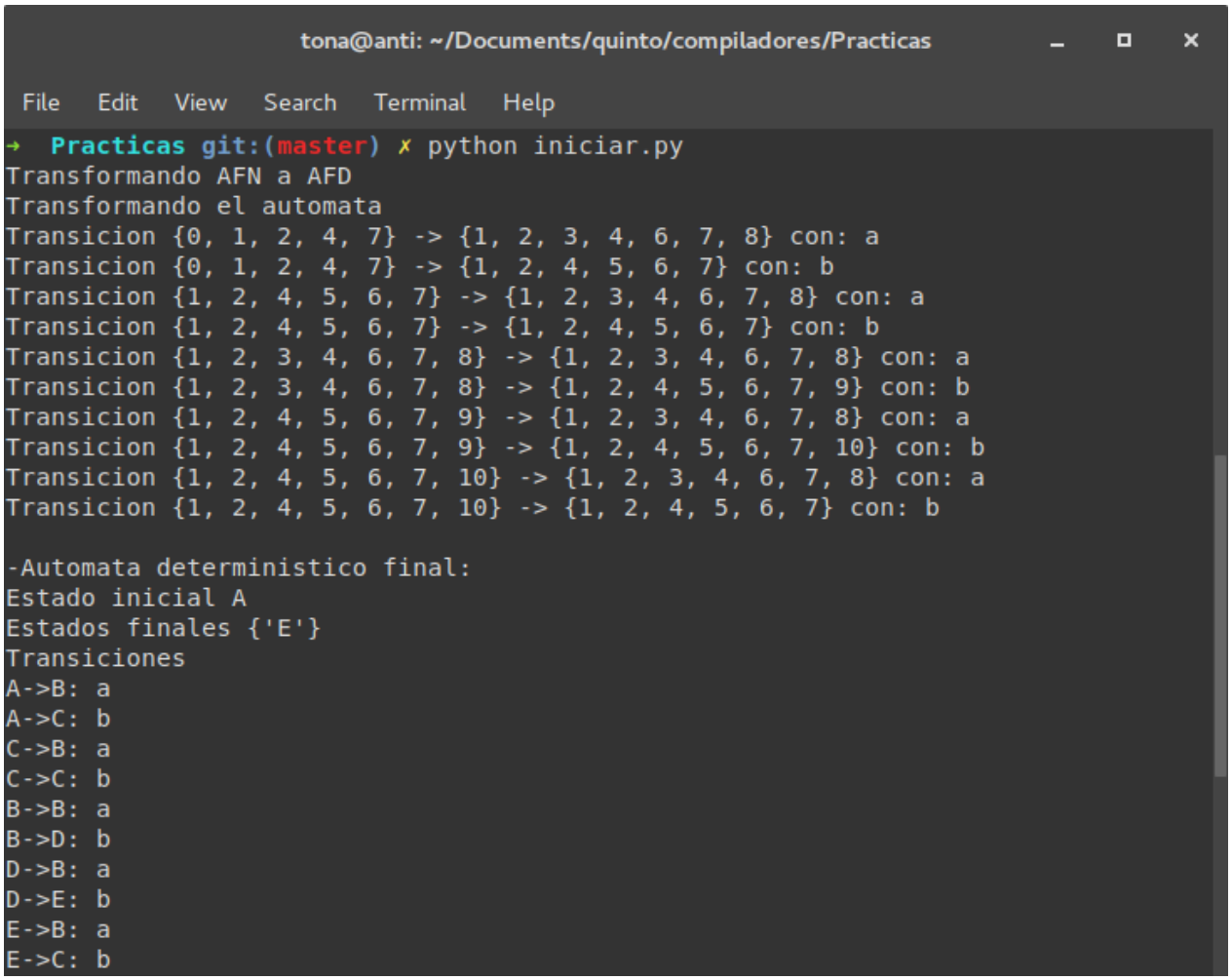

Figura 3: Estados como subconjuntos y estados re-etiquetados en el autómata determinista final.

En la imagen 3 lo primero que se puede es el como se generan los estados de nuestro autómata determinista, los cuales son conjuntos de estados del AFN, lo siguiente es imprimir los componentes de nuestro nuevo AFD, comenzamos por el estado final, después los estados finales y finalmente imprimimos las transiciones mostradas al inicio pero ahora reetiquetadas.

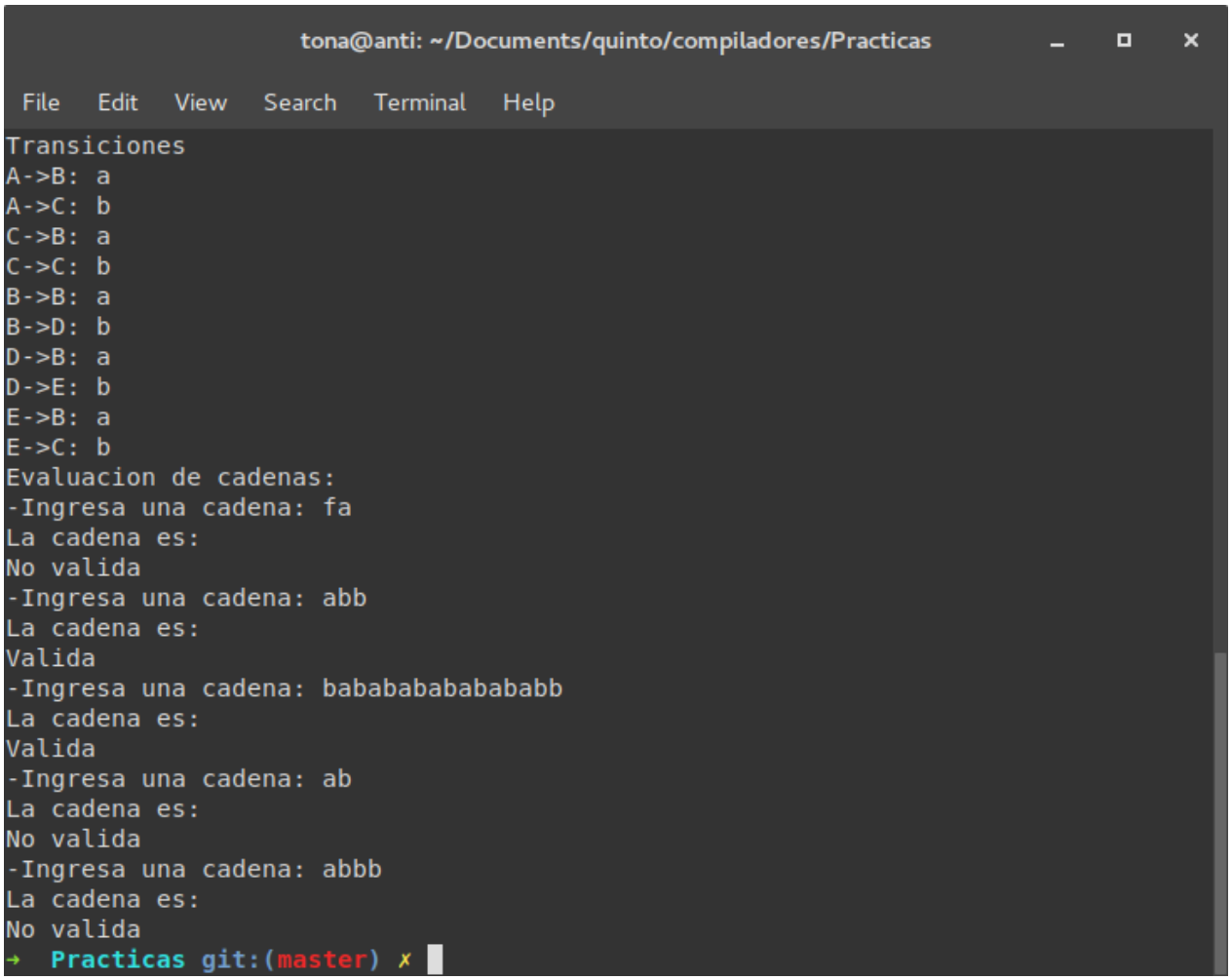

Figura 4: Pruebas realizadas sobre el autómata que se obtuvo a través de la transportación

Lo primero que se observa en la figura 4 es las transiciones que se obtuvieron después del proceso de transformación, lo siguiente es la evaluación de un conjunto de cadenas ingresadas por consola, es en esta parte en la que se puede observar que la transformación se realizó correctamente debido a que la evaluación de las cadenas es correcta.

4. Conclusiones

El desarrollo de esta práctica fue sencillo debido a que al tener los algoritmos ya definidos [1] solo se tuvo que modelar dichos algoritmos en un lenguaje orientado a objetos en este caso Python y después combinar esta práctica con la primer práctica realizada para la evaluación de cadenas.

Además, la realización de esta practica y las anteriores permitió entender la importancia que tienen las expresiones regulares y los autómatas en el análisis léxico de un compilador. Es por esto que para este punto la elaboración de un analizador léxico personal es posible.

Referencias

[1] V. A. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1st ed., 1986.