Tecnológico de Monterrey

Tecnológico de Monterrey - Campus Monterrey

Escuela de ingeniería y ciencias

Departamento regional de ciencias de la computación

Unidad de formación TC2037.601 - Implementación de métodos computacionales

Docente: Dr. Jesús Guillermo Falcón Cardona

Actividad integradora 1 - Lenguajes regulares

4/05/2025

Grupo 601

Daniela Herrera García A00838702

Emily Joanne Castillo Martínez A00839559

Introducción

En esta actividad integradora, se ha desarrollado una aplicación en Python que, al recibir como entrada una expresión regular y un alfabeto, es capaz de construir y mostrar gráficamente el autómata finito determinista correspondiente. Se nos dieron especificaciones, tales como el formato específico y los operadores aceptados, los cuales serían: * (estrella de Kleene), | (unión), y . (concatenación). Cuando se ingresa esta expresión regular a través de una interfaz de usuario, se visualiza claramente el DFA, comprobando si una palabra pertenece o no al lenguaje representado por la expresión que se ingresó. La situación consiste en implementar algoritmos de conversión para cambiar una expresión regular a un autómata finito no determinista (NFA) y luego, un NFA a un autómata finito determinista (DFA), usando técnicas como el algoritmo de Thompson y el de subconjuntos para realizar conversiones. Al tener desarrollada la aplicación, se espera que la interfaz permita al usuario escribir una palabra. Después, al procesarla, el DFA debe mostrar gráficamente el recorrido de estados paso a paso, finalizando con un mensaje que indique si la palabra fue aceptada.

Construcción del autómata finito no determinista

La primera fase de la realización de la actividad integradora constó en realizar un programa que nos permitiera transformar una expresión regular normal en un autómata finito no determinista, mostrando entonces todas las posibles secuencias que se pueden formar con esa expresión. Se utilizó la función infix2postfix, que convierte la expresión de su forma original (en forma infija) a una forma postfija. Esto con el objetivo de hacer el análisis más fácil, pues los símbolos o dígitos aparecen primero y luego los operadores. Se hace uso también de un diccionario para poder saber la prioridad de operadores, y también se emplea una pila para ir armando la salida.

Luego, la función postRe2NFA toma esa expresión postfija y va construyendo un autómata poco a poco, usando el método de Thompson. Cada simbolo crea un pequeño autómata, y los operadores como “ * ”, “ | ”, “ . “ y “ () ” combinan cada pieza según las reglas que se especifiquen. Cada una de esas “combinaciones” se guarda como una lista de diccionarios que indican los estados y las transiciones entre ellos. Al final, la salida es dicho diccionario con la información del NFA: sus estados, su estado inicial, el final, y el alfabeto que se usó.

Código:

```python
import re
from graphviz import Digraph


def infix2postfix(regex):
  postFix = []
  stack = []
  precedence = {
    '^' : 5,
    '/': 4,
    '*': 4,
    '.': 3,
    '+': 2,
    '-': 2,
    '(': 1
  }

  associativity = {
    '^': 'RL',  # Derecha a izquierda
    '*': 'LR',   # Izquierda a derecha
    '/': 'LR',
    '+': 'LR',
    '-': 'LR',
    '.': 'LR'
  }

  for ele in regex:
    if ele == ' ':
      continue

    if ele not in "()^/*+-.":
      postFix.append(ele)
    elif ele == '(':
      stack.append(ele)
    elif ele == ')':
      fin = ''
      while fin != '(':
        fin = stack.pop()
        if fin != '(':
          postFix.append(fin)
    elif len(stack) > 0:
      if precedence[ele] > precedence[stack[-1]]:
        stack.append(ele)
```

```python
      elif precedence[ele] < precedence[stack[-1]]: #current element precedence is lower than last element in stack precedence
        while len(stack) > 0 and precedence[ele] < precedence[stack[-1]]:
            lastEle  =  stack.pop()
            postFix.append(lastEle)
        stack.append(ele)

      elif precedence[ele]  ==  precedence[stack[-1]]:
        if associativity[ele] == 'RL':
          stack.append(ele)
        elif associativity[ele] == 'LR':
          sameLastAsso  =  stack.pop()
          postFix.append(sameLastAsso)
          stack.append(ele)
    else:
      stack.append(ele)

  #pop out last stack elements
  while stack:
    postFix.append(stack.pop())

  postFix = ''.join(postFix)
  noBlanks = postFix.replace(" ", "")
  print(noBlanks)
  return noBlanks


def postRe2NFA(postfix):
  #waits for postfix re
  regex = ''.join(postfix)

  #abecedario, takes only the elements it sees in the expression
  keys = list(set(re.sub('[^A-Za-z0-9]+', '', regex)+'e'))

  s = []
  stack = []
  start = 0
  end = 1

  #counter for states
  counter = -1

```

```
    #classifier for the different states we will have
    c1 = 0
    c2 = 0


   #lets say key is = AB*CD*+
    for i in regex:
        if i in keys:
            counter = counter+1;c1 = counter
            counter = counter+1;c2 = counter
            s.append({});s.append({}) #creates a dictionary for each state, so that each can have a transition
            stack.append([c1,c2])
            s[c1][i] = c2 #ex; s = [{1 : {'A' : 2}},{}]; es decir, 1---A--->2
        elif i == '*':
            r1,r2 = stack.pop()
            counter = counter+1;c1 = counter
            counter = counter+1;c2 = counter
            s.append({});s.append({})
            stack.append([c1,c2])
            s[r2]['e'] = [r1,c2]
            s[c1]['e'] = [r1,c2]
            if start == r1:start = c1
            if end == r2:end = c2
        elif i == '+':
          r1, r2 = stack.pop()
          counter += 1; c1 = counter
          counter += 1; c2 = counter
          s.append({}); s.append({})
          stack.append([c1, c2])
          s[r2]['e'] = [r1, c2]
          s[c1]['e'] = [r1]
        elif i == '.':
            #get last states in stack and unite them
            nfa21, nfa22 = stack.pop()
            nfa11, nfa12 = stack.pop()
            stack.append([nfa11, nfa22])
            s[nfa12]['e'] = nfa21
            if start == nfa21 : start = nfa11
            if end == nfa12 : end = nfa22
        elif i == '|':
            counter = counter+1
            c1 = counter
            counter = counter+1
            c2 = counter
```

```
            s.append({})
            s.append({})
            r11,r12 = stack.pop()
            r21,r22 = stack.pop()
            stack.append([c1,c2])
            s[c1]['e'] = [r21,r11]
            s[r12]['e'] = c2
            s[r22]['e'] = c2
            if start == r11 or start == r21:start = c1
            if end == r22 or end == r12:end = c2
    print (keys)
    print (s)

    return {
      "states": s,
      "start": start,
      "end": end,
      "alphabet": keys
    }

def main():
    # Example expression that should result in a DFA with more than two states
    # This expression includes concatenation, union, and Kleene star
    expression = 'A.B|C*'
    result = infix2postfix(expression)
    nfa = postRe2NFA(result)
    print(nfa)


if __name__ == '__main__':
    main()
```

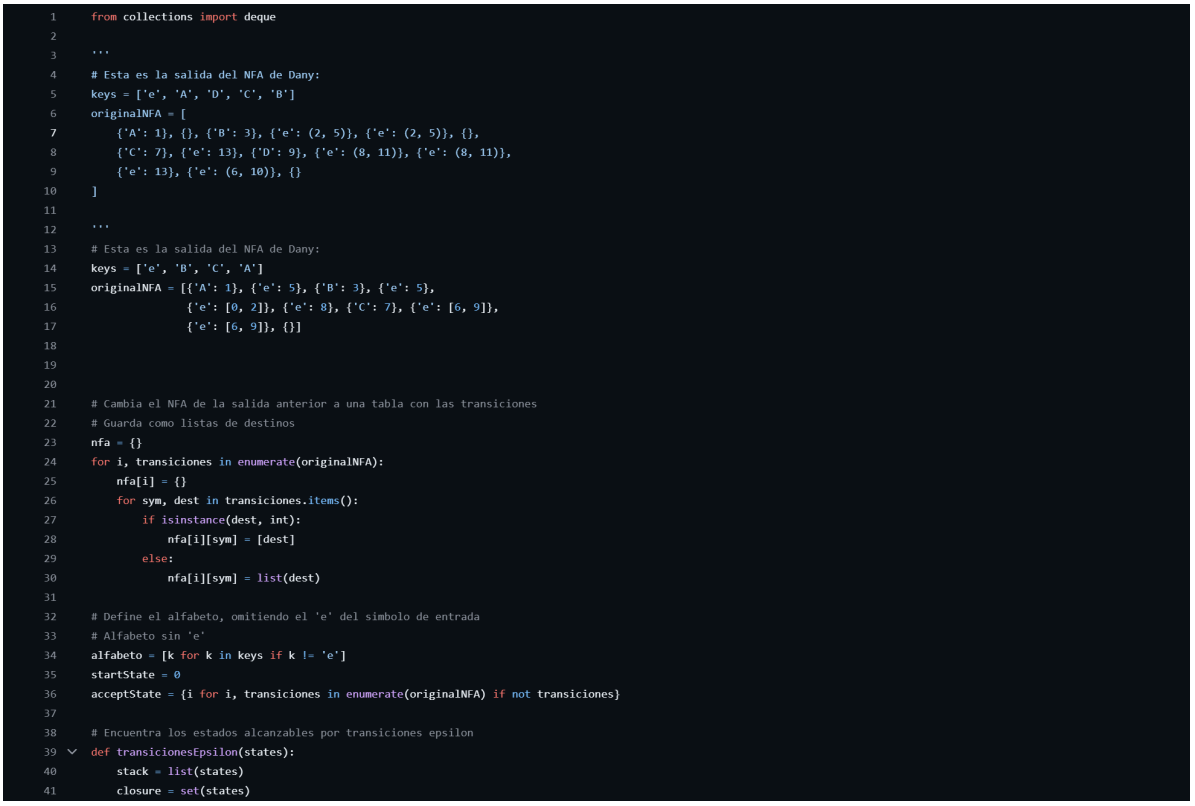
```
Run    Thompson ×

C:\Users\emily\PycharmProjects\PythonProject\.venv\Scripts\python.exe C:\Users\emily\PycharmProjects\PythonProject\Thompson.py
['B', 'e', 'C', 'D', 'A']
[{'A': 1}, {}, {'B': 3}, {'e': (2, 5)}, {'e': (2, 5)}, {}, {'C': 7}, {'e': 13}, {'D': 9}, {'e': (8, 11)}, {'e': (8, 11)}, {'e': 13}, {'e': (6, 10)}, {}]

Process finished with exit code 0
```

Construcción del autómata finito determinista

La segunda fase de la actividad integradora consistió en convertir el autómata finito no determinista en un autómata finito determinista utilizando el algoritmo de subconjuntos. Este algoritmo integra las transiciones épsilon, representadas mediante una lista de diccionarios. Para mantener una continuidad, se tomó como entrada la salida del programa anterior, entonces se tuvo que transformar la representación del NFA original a un formato uniforme, haciendo que cada transición tenga una lista de estados de destino. Luego, se eliminan las transiciones epsilon ('e') del NFA al calcular encontrar los estados alcanzables por transiciones epsilon desde un conjunto dado. Para esto, se usa un bucle while para que se repita hasta determinar todos los nuevos estados del DFA y se construyan sus transiciones.

Después de construir todos los posibles estados del DFA, se identifican los estados finales considerando los estados que incluyen al menos uno de los estados finales del NFA original, como aprendimos en la conversión en clase. Finalmente, el código imprime el conjunto de estados del DFA, sus transiciones, el estado inicial y los estados finales. De esta manera, pudimos transformar el NFA en un DFA mediante el algoritmo de conversión visto en clase, el cual sería el algoritmo de subconjuntos.

Código:



```python
from collections import deque

'''
# Esta es la salida del NFA de Dany:
keys = ['e', 'A', 'D', 'C', 'B']
originalNFA = [
    {'A': 1}, {}, {'B': 3}, {'e': (2, 5)}, {'e': (2, 5)}, {},
    {'C': 7}, {'e': 13}, {'D': 9}, {'e': (8, 11)}, {'e': (8, 11)},
    {'e': 13}, {'e': (6, 10)}, {}
]

'''
# Esta es la salida del NFA de Dany:
keys = ['e', 'B', 'C', 'A']
originalNFA = [{'A': 1}, {'e': 5}, {'B': 3}, {'e': 5},
               {'e': [0, 2]}, {'e': 8}, {'C': 7}, {'e': [6, 9]},
               {'e': [6, 9]}, {}]



# Cambia el NFA de la salida anterior a una tabla con las transiciones
# Guarda como listas de destinos
nfa = {}
for i, transiciones in enumerate(originalNFA):
    nfa[i] = {}
    for sym, dest in transiciones.items():
        if isinstance(dest, int):
            nfa[i][sym] = [dest]
        else:
            nfa[i][sym] = list(dest)

# Define el alfabeto, omitiendo el 'e' del simbolo de entrada
# Alfabeto sin 'e'
alfabeto = [k for k in keys if k != 'e']
startState = 0
acceptState = {i for i, transiciones in enumerate(originalNFA) if not transiciones}

# Encuentra los estados alcanzables por transiciones epsilon
def transicionesEpsilon(states):
    stack = list(states)
    closure = set(states)
```

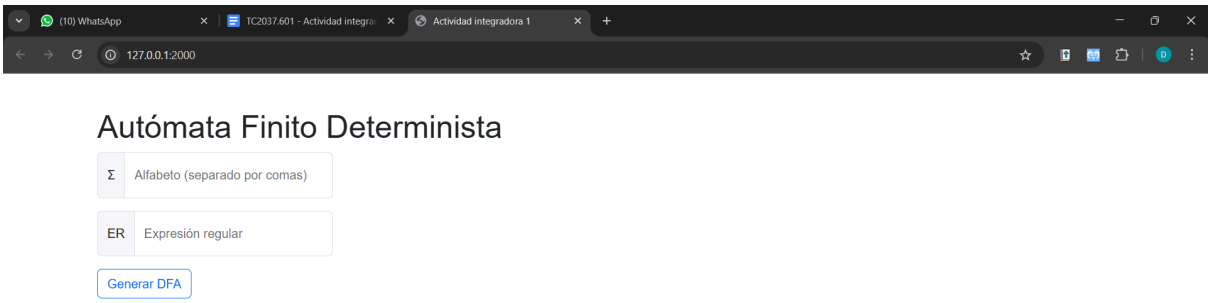
```python
        # Recorre los destinos epsilon
        while stack:
            state = stack.pop()
            for nextState in nfa.get(state, {}).get('e', []):
                if nextState not in closure:
                    closure.add(nextState)
                    stack.append(nextState)
        return closure

    # Encuentra los estados alcanzables por simbolos especificos (que nos sean epsilon)
    def transicionesSimbolos(states, symbol):
        result = set()
        for state in states:
            for dest in nfa.get(state, {}).get(symbol, []):
                result.add(dest)
        return result

    # Define variables para empezar la conversion
    dfaStates = []
    dfaTransitions = {}
    dfaFinalState = []
    visto = set()

    # Usar para el estado inicial la funcion transicionesEpsilon
    start = frozenset(transicionesEpsilon([startState]))
    queue = deque([start])
    visto.add(start)

    # Aplica el algoritmo de subconjuntos
    while queue:
        current = queue.popleft()
        dfaStates.append(current)

        for symbol in alfabeto: # Se intenta mover con cada simbolo del alfabeto
            movimiento = transicionesSimbolos(current, symbol)
            estadosAlcanzables = transicionesEpsilon(movimiento) # aplica transicionesEpsilon despues de cada movimiento
            conjuntoEstados = frozenset(estadosAlcanzables)

            dfaTransitions[(current, symbol)] = conjuntoEstados # Conjunto de transiciones para el DFA

            # Agregar conjuntoEstados a la fila si no se ha visto, para procesar despues
            if conjuntoEstados not in visto:
                visto.add(conjuntoEstados)
                queue.append(conjuntoEstados)
```

```python
    # Toma como estado final del AFD cualquier conjunto de estados del DFA que contenga el estado final del AFN
    for state in dfaStates:
        if any(s in acceptState for s in state):
            dfaFinalState.append(state)


    # Imprime resultados
    imprimeEstados = []
    for i in range(len(dfaStates)):
        imprimeEstados.append("Q" + str(i) + ":" + str(sorted(dfaStates[i])))
    print("Estados del AFD: " + " , ".join(imprimeEstados))

    imprimeTransiciones = []
    for key in dfaTransitions:
        src = sorted(list(key[0]))
        sym = key[1]
        dest = sorted(dfaTransitions[key])
        imprimeTransiciones.append(str(src) + " --" + str(sym) + "--> " + str(dest))
    print("Transiciones del AFD: " + " , ".join(imprimeTransiciones))

    print("Estado inicial del AFD: " + str(sorted(start)))

    imprimeEstadosFinales = [str(sorted(state)) for state in dfaFinalState]
    print("Estados finales del AFD: " + " , ".join(imprimeEstadosFinales))
```

```
Run  NFAtoDFA
C:\Users\emily\PycharmProjects\PythonProject\.venv\Scripts\python.exe C:\Users\emily\PycharmProjects\PythonProject\NFAtoDFA.py
Estados del AFD: Q0:[0] , Q1:[] , Q2:[1, 5, 6, 8, 9] , Q3:[6, 7, 9]
Transiciones del AFD: [0] --B--> [] , [0] --C--> [] , [0] --A--> [1, 5, 6, 8, 9] , [] --B--> [] , [] --C--> [] , [] --A--> [] , [1, 5, 6, 8, 9] --B--> [] , [1, 5, 6, 8, 9] --C-
Estado inicial del AFD: [0]
Estados finales del AFD: [1, 5, 6, 8, 9] , [6, 7, 9]

Process finished with exit code 0
```

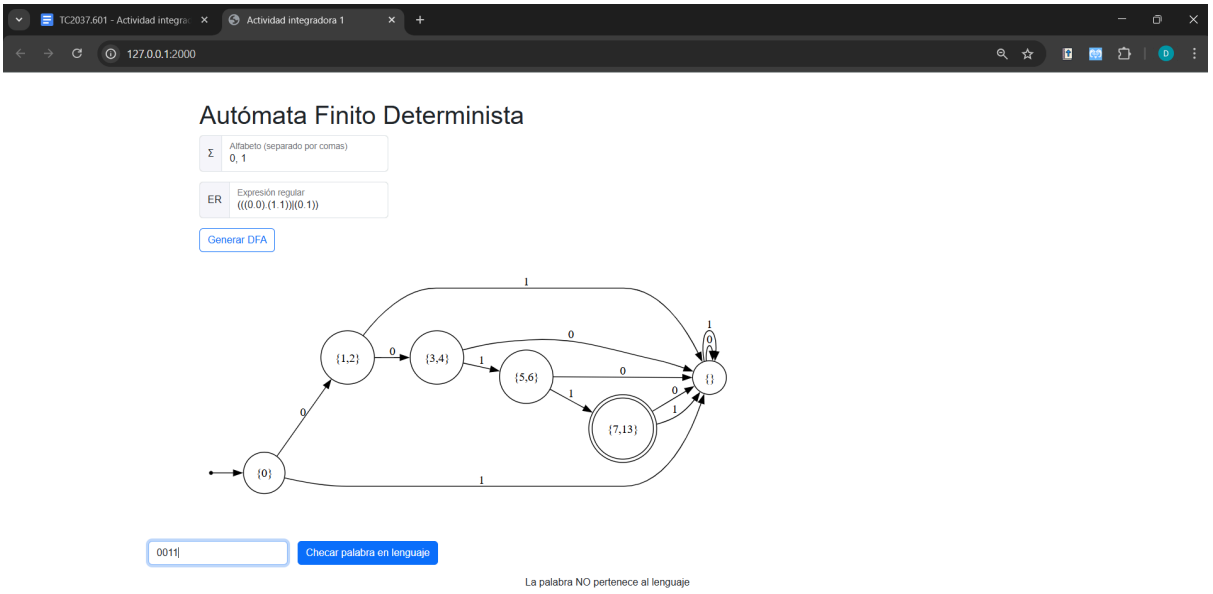
Presentación visual

Con el autómata finito determinista creado en la fase anterior, el siguiente paso consistió en representar este diagrama de transición gráficamente en una UI; para lograrlo, desarrollamos una aplicación web que integraba diferentes tecnologías. En la app, creada con HTML, el usuario ingresa un alfabeto y la expresión regular para generar el DFA. Utilizando un API, la información se envía mediante JavaScript para que el backend en Python procese los datos. Es ahí donde se juntan los pasos anteriores: La expresión regular se convierte a notación postfija, se genera un autómata finito no determinista (NFA) a través del uso del algoritmo de Thompson, y luego se transforma en un DFA utilizando el algoritmo de subconjuntos. Finalmente, usando la librería Graphviz, se genera una imagen que es devuelta al navegador para ser mostrada al usuario.



Interfaz por default

Al mostrar el DFA en la pantalla, la interfaz también despliega una segunda parte, en la cual le permite al usuario introducir una palabra. El sistema toma el input del usuario y la revisa con el DFA generado previamente, determinando si pertenece o no al lenguaje que fue definido por la expresión regular que insertó al inicio. El resultado es un mensaje el cual indica si la palabra pertenece al lenguaje. Esto permite al usuario construir y observar de manera clara el autómata, además de probarlo directamente con la UI.



Interfaz en uso

Pruebas

Finalmente, se realizaron distintas pruebas ingresando un alfabeto, una expresión regular y palabras para verificar si fueron aceptadas por el DFA generado. En los primeros csos, se introdujeron palabras que sí cumplían con el lenguaje y el sistema lo confirmó correctamente. En una prueba, se ingresó una cadena que no seguí la estructura de la expresión regular, y el sistema respondió adecuadamente al indicar que la palabra no pertenece al lenguaje, mostrando que esa palabra no fue reconocida por el autómata. Estas pruebas permitieron validar que la conversión de la expresión regular al DFA y el proceso de evaluar las palabras funcionara correctamente y de manera confiable. A continuación el video realizando las pruebas:

[Haz clic aquí para ver el video]

Enlace a repositorio

https://github.com/DanHeGa/M-todos

Para ejecutar la página final se requiere correr API.py