INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

GARZÓN DOMÍNGUEZ GERARDO ISMAEL

3CV18

COMPILADORES

PRÁCTICA 1:
AFN-E, Autómata finito no determinista con transiciones epsilon y manejo de errores.

INTRODUCCIÓN

Si examinamos los autómatas finitos deterministas, notamos que la característica en común que tienen, es que, la función de transición, δ, define una sola transición para cada par estado-símbolo. En tal caso, la función δ es llamada una función total, y es por ello que estos autómatas son deterministas. [1]

Para los autómatas finitos no deterministas, se agrega la característica de que, para cada estado y símbolo, puede haber más de una transición definida. Formalmente, esto se puede lograr definiendo la función de transición tal que para un par estado-símbolo, esta tenga un rango el cual es un conjunto de estados. [1]

Un autómata finito no determinista esta definido mediante la 5-tupla: $M = (Q, \Sigma, \delta, q_0, F)$. [1]

Dónde:
Q : El conjunto finito de estados internos del autómata.
Σ : El alfabeto de entrada del autómata, toda cadena de entrada debe estar construida sobre este. Para que pueda ser procesada.
δ : La función de transición: $\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$. Una función que mapea un par (q_i, a), $q_i \in Q, a \in \Sigma \cup \{\lambda\}$ a un subconjunto del conjunto potencia de Q.
$q_0 \in Q$: Un elemento perteneciente al conjunto de estados del autómata, se considera como el estado inicial del autómata, es decir desde este estado comenzaría su funcionamiento al procesar una cadena.
$F \subseteq Q$: Subconjunto del conjunto de estados internos del autómata. Este subconjunto representa los estados finales válidos del autómata, es decir que las cadenas que lleven al autómata a este estado como paso final, serán aceptadas. [1]

De manera similar al AFD, un AFN, se encontrará inicialmente en el estado q_0, a partir de este estado, leerá un símbolo de la cadena y se tomará el resultado de la función δ. Dado que el resultado es un subconjunto de 2^Q, se tomará cada estado del subconjunto como una opción para moverse al siguiente estado, para el cual se repite el proceso (recursividad). Nótese también, que dada la definición de la función δ, se puede aceptar la cadena vacía como argumento para δ. De esta manera el autómata podrá moverse de un estado a otro sin la necesidad de consumir un símbolo de la cadena. [1]

Ahora se mostrará un poco más de teoría para expandir el entendimiento de los AFN-E.
Como ya se mencionó, un autómata finito no determinista puede tener como alfabeto, $\Sigma \cup \epsilon$, para que un autómata sea un AFN-E (Autómata finito no determinista epsilon), su función de transiciones debe tener definidas transiciones con pares $(q_i, \epsilon), q_i \in Q$ [2]. La siguiente definición informal ayuda a comprender como es que los AFN-E pueden atravesar caminos con transiciones ϵ.
La cerradura epsilon de un estado: $ECLOSE(q)$ se obtiene siguiendo todas las transiciones de q etiquetadas como ϵ, en otras palabras esto es, $\delta(q, \epsilon)$ al llegar a un nuevo estado p, se aplica nuevamente la definición $ECLOSE(p, \epsilon)$, y así sucesivamente, de esa manera se puede notar la naturaleza recursiva de la cerradura epsilon. [2] Esta definición es importante para la implementación de autómatas finitos no deterministas epsilon, ya que en teoría, una cerradura epsilon se podría dar en

cualquier punto del procesamiento de una cadena, siempre y cuando $\delta(q_i, \epsilon) \neq \phi$. De manera concisa, esta definición permitirá a un autómata generar caminos de transiciones vacías.

DESARROLLO E IMPLEMENTACIÓN

En esta sección se mostrarán los problemas a resolver y cuales fueron las soluciones propuestas respecto a la implementación de esta práctica.

En esta práctica se implementará un programa que sea capaz de generar autómatas a partir de una definición en un archivo de texto y procesar cadenas de texto para mostrar si estas pertenecen al lenguaje descrito por el autómata. Los autómatas generados tendrán un par de características adicionales a los AFN, esto es, podrán recuperarse del estado de error al que se entra cuando algún símbolo que no pertenece al alfabeto del autómata, y serán capaces de realizar transiciones ϵ, o sea, transiciones mediante cadenas vacías, o sin consumir símbolos de la cadena de entrada.
Se usará el paradigma orientado a objetos mediante el lenguaje *python* para implementar los requerimientos de esta práctica.

La primera parte de la implementación consiste crear una manera de procesar archivos de texto con una estructura predeterminada, que es la siguiente:
[linea 1] <conjunto de estados Q del autómata separados por comas>
[linea 2] <alfabeto del autómata Σ, elementos separados por comas>
[linea 3] <estado inicial del autómata $q_0 \in Q$>
[linea 4] <estados de aceptación del autómata $F \subseteq Q$, separados por comas>
[linea 5 …] <definición de la función de transiciones a_i, b_i, c_i> $i = 5, ...n$, donde n es el último renglón no vacío del archivo.
donde cada elemento $a_i \in Q, b_i \in \Sigma \cup \epsilon, c_i \in Q$, o sea, a_i es un estado del autómata, b_i es un símbolo del alfabeto del autómata o la cadena vacía ϵ, que en este caso será denotada con el símbolo especial "E" (E mayúscula), y c_i un estado perteneciente al conjunto de estados del autómata.

Se podrán instanciar autómatas a partir de la clase *automaton* (en código), esta clase tendrá un constructor para generar la definición del autómata a partir de la ruta de archivo de texto.
Para inicializar un autómata a partir de un archivo de texto, se usa un constructor que recibe como argumentos la ruta del archivo, y el *token* que separá a elementos listados, en este caso, es la coma.
Dado el formato de archivo, el procedimiento general a seguir para configurar un autómata es el siguiente:

1. Cargar el archivo a memoria si se pudo abrir sin problemas
2. Separar el archivo por lineas, esto no requiere mayor esfuerzo, ya que *python* tiene funciones incluidas para realizar esto
3. Para las lineas 1,2,4, separar los elementos por comas y almacenarlos en 3 listas distintas
4. Generar los conjuntos correspondientes en base a las 3 listas generadas, estos conjuntos son atributos del objeto autómata, y representan a los elementos Q, Σ, F de la 5-tupla del autómata
5. La linea 3 corresponde al estado de inicio del autómata, esta debe encontrarse dentro de Q, también se guarda como atributo del autómata y se comprueba si $q_0 \in Q$
6. Desde la linea 5 hasta la ultima linea no vacía, considerando que cada linea tiene 3 elementos a_i, b_i, c_i
    1. Generar una llave, la cual es un par ordenado $k = (a_i, b_i)$

2. Agregar la llave a un diccionario, de tal manera que $\Delta[k] = c_i$
3. De manera adicional, se realiza una comprobación para conocer la validez de la definición de la tabla de transiciones, esto es, saber si $a_i \in Q, b_i \in \Sigma \cup \epsilon, c_i \in Q$, si en alguna linea no se cumple, se imprime un mensaje de advertencia.

Tras el procedimiento anterior, y si el archivo está definido de manera correcta, se obtiene un objeto de tipo *automaton*, este objeto tendrá un método visible para el usuario final (el programador), con el cual se podrá comprobar la validez de una cadena con la definición del autómata, nótese que el manejo de errores es muy limitado por ahora.
Cabe mencionar que gracias a que *python 3* y versiones posteriores proveen gran variedad de estructuras de datos nativas, por lo cual se facilitó la decisión de que estructuras usar para cada elemento del autómata, así como la implementación en general.

Ahora se discute la segunda parte de la implementación de este programa, en particular el algoritmo para atravesar todos los caminos posibles del autómata dada una cadena de entrada.
La solución propuesta para este problema es una función recursiva con 3 casos límite:

1. La posición j en la que se encuentra el autómata sobre la cadena es menor que la longitud de la cadena, esto es, $j < |S|$ y el símbolo actual $S[j] \in \Sigma$
2. La posición j en la que se encuentra el autómata sobre la cadena es menor que la longitud de la cadena, o sea, $j < |S|$ y el símbolo actual $S[j] \notin \Sigma$
3. La posición j en la que se encuentra el autómata es igual que la longitud de la cadena, esto es, $j = |S|$, este es simplemente el caso especial cuando se consumieron todos los símbolos de la cadena.

Cada uno de estos casos cubre otras partes del problema:
El caso 1 sucede cuando $j < |S|$ y $S[j] \in \Sigma$, o sea, que aquí se realizarán las transiciones ϵ, si es que el estado en el que se encuentra el autómata la acepta, si no, se realizarán transiciones consumiendo símbolos de la cadena de entrada.
El caso 2 cubre $j < |S|$ y $S[j] \notin \Sigma$, esto es, cuando un símbolo no pertenece al alfabeto del autómata, esto es a lo que conocemos como completar el autómata para que se pueda recuperar de estados de error.
El caso 3 sucede cuando se llego al final de la cadena, o sea, se consumieron todos los símbolos de la cadena, sin embargo, se da la posibilidad de que el estado en el que el autómata quedó, acepte transiciones ϵ, en este caso se cubren el resto de transiciones ϵ para el estado actual, si es que existen.

Los caminos se guardarán en una lista que será tratada como un *stack*, esto es, tendrá longitud variable, y solo se podrá agregar y eliminar elementos al final del mismo. En este caso se eligió un *stack* ya que no es posible conocer el tamaño máximo del arreglo que sería capaz de guardar todos los caminos, esto debido a las transiciones ϵ, es decir, cuando se tiene un AFN sin transiciones ϵ, se conoce que el tamaño máximo que un camino puede tomar es la longitud de la cadena de entrada.

A continuación se mostrará la función que implementa lo que se describió en los párrafos anteriores

```
def __traverse_path__(self, curr_state, sequence, pos):
    if pos < len(sequence) and sequence[pos] in self.__alphabet__:

        key = (curr_state, epsilon)
        m = self.__Delta__(key)
        if m != [None]:
            for i in m:
                self.__ctrl_path__(sequence, pos, i, True)
                self.__traverse_path__(i, sequence, pos)
                self.__pathstack__.pop()

        key = (curr_state, sequence[pos])
        if key in self.__delta_map__:
            for i in self.__Delta__(key):
                self.__ctrl_path__(sequence, pos, i, False)
                self.__traverse_path__(i, sequence, pos+1)
                self.__pathstack__.pop()

    elif pos < len(sequence) and sequence[pos] not in self.__alphabet__:
        self.__ctrl_path__(sequence, pos, None, False)
        self.__traverse_path__(curr_state, sequence, pos+1)
    else:
        key = (curr_state, epsilon)
        if key in self.__delta_map__:
            for i in self.__Delta__(key):
                self.__ctrl_path__(sequence, pos, i, True)
                self.__traverse_path__(i, sequence, len(sequence))
                self.__pathstack__.pop()
```

Figura 1: Función para atravesar todos los caminos posibles de un autómata dada una cadena de entrada

Ahora se comentarán un par de detalles adicionales para que se tenga un entendimiento de esta función.

Como se nota, se genera una llave (q_i, ϵ), para comprobar si el estado actual acepta transiciones vacías, en caso afirmativo, se añadirá un estado al *pathstack* mediante la función *__ctrl_path__*, y la función se llamará a sí misma pero sin consumir el símbolo actual, para comprobar si el nuevo estado acepta transiciones vacías, en caso negativo, se intentará generar una llave $(q_i, S[j])$ y se comprobará si este es un par válido para la función delta, en caso afirmativo la función se llama a sí misma, pero esta vez sí consume un símbolo de la cadena.

En la segunda rama general de la función (elif …), se cubrirá el en el que el símbolo de entrada actual no pertenezca al alfabeto del autómata, la función *__ctrl_path__* se encargará de registrar los símbolos no válidos, además, la función se llamará a sí misma en esta rama mientras los símbolos actuales sean inválidos, básicamente se “comerá” las subcadenas inválidas.

La última rama general de la función (else) , cubrirá el caso en el que se hayan consumido todos los símbolos de la cadena, pero comprobará nuevamente si el estado actual acepta transiciones vacías, en caso afirmativo, la función se llamará a sí misma en esta rama hasta que llegue a un estado que no acepte transiciones vacías.

Cabe mencionar que la función *__ctrl_path__* se encargará de identificar cuando el autómata llegue a un estado de aceptación, y añadirá el *pathstack* actual a una lista de caminos para la cadena de entrada.

Otro detalle, es que dado que la función *__ctrl_path__*, añade elementos al *pathstack*, la manera de mantener el *pathstack* con los elementos y longitud correcta, es eliminando elementos del mismo una vez que se llegue al fondo de la recursividad, esto sucede cuando se consumieron todos los elementos de una cadena y se realizaron las transiciones vacías faltantes, entonces, cuando se llega a la última llamada de la función *__traverse_path__*, esta comienza a regresar y a ejecutar la siguiente línea de código, que para 2 de las ramas generales de la función (if, else) es método *pop* para el *pathstack.* Nótese que la segunda rama general de la función (elif) no realiza este paso, esto se debe a que para ese caso, la función *__ctrl_path__* no agrega ningún elemento al *pathstack.*

Un detalle interesante al momento de implementar esta práctica es que, si bien *python* acepta ciertas abstracciones orientadas a objetos, su implementación del paradigma orientado a objetos no sigue estrictamente otras implementaciones populares, como java o c++, por ejemplo, en *python* no existen realmente los métodos privados, más bien se sigue una práctica común de agregar “__” como prefijo a atributos que no necesitan ser visibles al exterior, este prefijo, le indica al compilador que debe aplicar

una técnica conocida como *name mangling*, esto es, alterar el nombre del atributo para que no pueda ser accedido desde el exterior del objeto, de hecho, el nombre final del atributo es sustituido por _<nombre de clase>__<nombre de atributo>. [3]

A continuación se mostrará ejemplos del funcionamiento del programa con unas cadenas de entrada predefinidas.

```
>> ba
Cadena "ba" es valida
Info: [['0', '8', '9', '10', '11']] Sin errores encontrados
```

Figura 2: Autómata generado con el archivo "Thompson1.txt" probado con la cadena "ba"

```
>> ba1a
Cadena "ba1a" es valida
Info: [['0', '1', '2', '3', '4', '5', '6', '7', '10', '11']] Simbolos no permitidos: {'1'}
```

Figura 3: Autómata generado con el archivo "Thompson1.txt" probado con la cadena "ba1a"

```
>> abbzbaa
Cadena "abbzbaa" es valida
Info: [['0', '1', '2', '5', '6', '2', '3', '4', '3', '4', '3', '4', '5', '6', '7', '10', '11']] Simbolos no permitidos: {'z'}
```

Figura 4: Autómata generado con el archivo "Thompson1.txt" probado con la cadena "abbzbaa"

```
>> ab@abaa
Cadena "ab@abaa" es valida
Info: [['0', '1', '2', '5', '6', '2', '3', '4', '5', '6', '2', '3', '4', '5', '6', '7', '10', '11']] Simbolos no permitidos: {'@'}
```

Figura 5: Autómata generado con el archivo "Thompson1.txt" probado con la cadena "ab@abaa"

Como se puede notar en las figuras 3, 4 y 5, el programa imprime una advertencia con los símbolos inválidos para el autómata.

La siguiente tabla muestra la función de transiciones para el autómata del archivo “Thompson1.txt”

Q	a	b	ϵ
→ 0	ϕ	ϕ	{1,8}
1	ϕ	ϕ	{2,7}
2	ϕ	ϕ	{3,5}
3	ϕ	{4}	ϕ
4	ϕ	ϕ	{5,3}
5	{6}	ϕ	ϕ
6	ϕ	ϕ	{7,2}
7	ϕ	ϕ	{10}
8	ϕ	{9}	ϕ
9	ϕ	ϕ	{10}
10	{11}	ϕ	ϕ
11 *	ϕ	ϕ	ϕ

Figura 6: Tabulación de la función de transiciones del autómata definido en "Thompson1.txt"

A partir de esta tabla se puede generar el autómata en su forma de grafo dirigido:

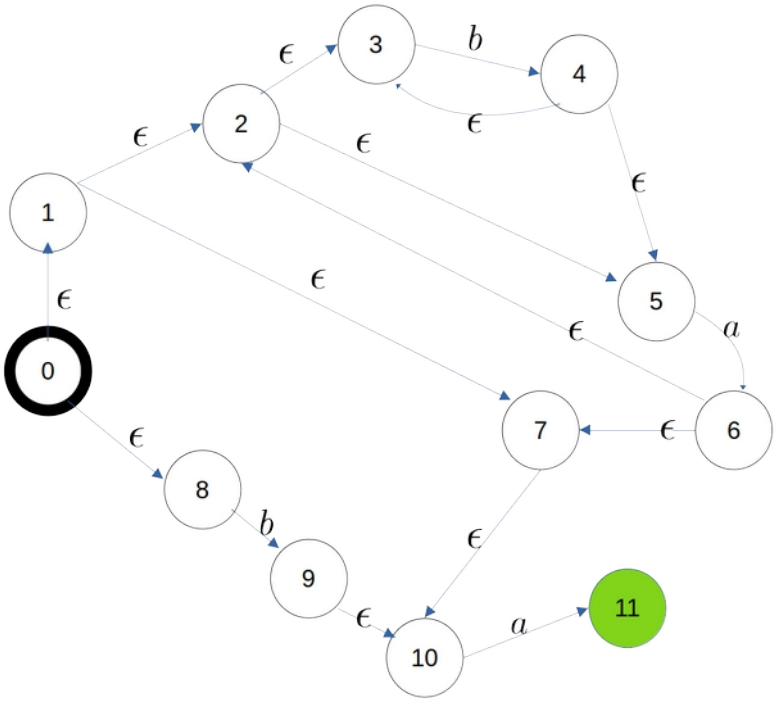

Figura 7: Grafo del autómata del archivo "Thompson1.txt", el circulo con contorno grueso es el estado de inicio, y el circulo verde el estado de aceptación

Si analizamos el camino de validación para la cadena “ba”, notamos que el único que existe es, 0,8,9,10,11. Ya que, si se observa, el otro camino que consume una “b” al inicio es 0,1,2,3,5. Pero en el siguiente paso hacia 6, es necesario consumir una a, lo cual impide la validación de la cadena, ya que si se siguiese el resto del camino, 0,1,2,3,4,5,6,7,10,11. Se necesitaría de una “a” extra al final de la cadena.

A continuación se muestra el resultado de la validación de la cadena “ba”, si la función de transiciones incluyera el siguiente mapeo: $\delta(3, \epsilon) = 7$

```
>> ba
Cadena "ba" es valida
Info: [['0', '1', '2', '3', '4', '3', '7', '10', '11'], ['0', '8', '9', '10', '11']] Sin errores encontrados
```

Figura 8: Validación de la cadena “ba” si se incluye el mapeo $\delta(3, \epsilon) = 7$

Ahora se mostrará la experimentación con el segundo autómata, dado por el archivo “Thompson2.txt”.

Primero se muestran 2 representaciones del autómata, mediante una tabulación de su función de transiciones y mediante un grafo dirigido:

Q	a	b	c	ϵ
→ 0	ϕ	ϕ	ϕ	{1,8}
1	ϕ	ϕ	{2}	ϕ
2	ϕ	ϕ	ϕ	{3,5}
3	{4}	ϕ	ϕ	ϕ
4	ϕ	ϕ	ϕ	{7}
5	ϕ	{6}	ϕ	ϕ
6	ϕ	ϕ	ϕ	{5,7}
7	ϕ	ϕ	ϕ	{12}
8	ϕ	ϕ	ϕ	{9,11}
9	{10}	ϕ	ϕ	ϕ
10	ϕ	ϕ	ϕ	{9,11}
11	ϕ	ϕ	ϕ	{12}
12*	ϕ	ϕ	ϕ	ϕ

Figura 9: Tabulación de la función de transiciones del autómata definido en "Thompson2.txt"

En forma de grafo:

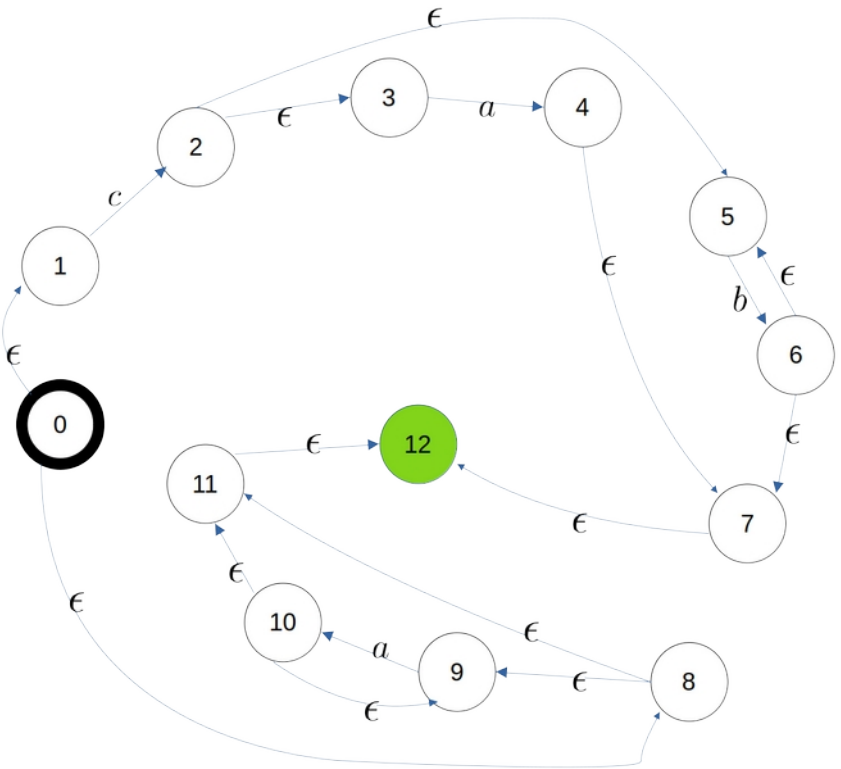

Figura 10: Grafo del autómata del archivo "Thompson2.txt", el circulo con contorno grueso es el estado de inicio, y el circulo verde el estado de aceptación

Dadas estas figuras, será más fácil visualizar el análisis que se dará a continuación sobre las cadenas prueba que se usaron con este autómata.

```
>> aaaa
Cadena "aaaa" es valida
Info: [['0', '8', '9', '10', '9', '10', '9', '10', '9', '10', '11', '12']] Sin errores encontrados
```

Figura 11: Autómata generado con el archivo "Thompson2.txt" probado con la cadena "aaaa"

Nótese que el camino no podría tomar los pasos 0,1,2,… Ya que esto requeriría que la cadena contuviese el símbolo “c”. En este caso toma una transición vacía hacia 8, y se cicla entre 9,10 dependiendo de la cantidad de “a” que contenga la cadena, observe que el camino de 10 a 9 es una transición vacía, y esto es lo que permite realizar el ciclado mencionado anteriormente.

```
>> ab@ab
Cadena "ab@ab" es invalida para el automata
```

Figura 12: Autómata generado con el archivo "Thompson2.txt" probado con la cadena "ab@ab"

Esta cadena no es valida, ya que el único camino que permite comenzar con una “a” no contiene transiciones con “B”.

```
>> cbqbb
Cadena "cbqbb" es valida
Info: [['0', '1', '2', '5', '6', '5', '6', '5', '6', '7', '12']] Simbolos no permitidos: {'q'}
```

Figura 13: Autómata generado con el archivo "Thompson2.txt" probado con la cadena "cbqbb"

En este caso, si se ignora la “q”, la cadena “cbbb” es parte del lenguaje descrito por el autómata, y se puede notar que este es uno de los casos en los que se tienen que realizar transiciones vacías después de haber consumido todos los símbolos de la cadena.

```
>> cb_a
Cadena "cb_a" es invalida para el automata
```

Figura 14: Autómata generado con el archivo "Thompson2.txt" probado con la cadena "cb_a"

Esta cadena sería válida si no tuviera el símbolo “a” al final, esto es ya que desde el ciclo que se genera entre los estados 5,6. No hay ninguna transición con “a”, teniendo en cuenta que “_” se ignora.

CONCLUSIÓN

En este práctica pude expandir el conocimiento previo de autómatas. En general, creo que ya comienzo a visualizar más la utilidad que tienen para expresiones regulares, en especial si tenemos en cuenta que esta implementación permite al autómata realizar transiciones vacías y recuperarse de errores, esto es útil, ya que tal vez más adelante sea posible generar algún programa que valide expresiones regulares y genere los autómatas correspondientes, esto con el hecho de que el programa, por ahora, acepta definiciones de autómatas generados mediante la construcción de Thompson.
Por el lado programático, cabe mencionar que no suelo usar mucho python, y a pesar de que no esté totalmente convencido de mi implementación, pude notar una gran diferencia entre escribir programas similares en C y python, especialmente respecto al procesamiento de archivos y las estructuras de datos, ya que, python ofrece una variedad de estructuras de datos de las cuales es fácil elegir cual usar para cada situación, sin embargo también se pueden implementar estas estructuras de datos en C, y probablemente lo único que haría falta para facilitar la escritura de estos programas en C, sería implementar también alguna interfaz de objetos genéricos usando estructuras en vez del clásico void*. Sin embargo, respecto a la parte del paradigma orientado a objetos, mientras que se puede simular en C, usar un lenguaje que tiene soporte para esto facilita un poco pensar respecto al problema, creo que esto tiene que ver más con la sintaxis, semántica y abstracciones que ofrecen los lenguajes con soporte para POO, por ejemplo meth(obj, a,b) sería algo equivalente a obj.meth(a, b).

REFERENCIAS

[1] P. Linz, An Introduction to Formal Languages and Automata, 6ta ed. Davis, California: Jones & Bartlett, 2017, pp. 51-52.

[2] J. Hopcroft, Introduction to Automata Theory, Languages, and Computation, 3ra de. Ithaca New York: Prentice Hall, 2006, p 75.

[3] Classes - Python 3.9.7 documentation, Python Software Foundation, 01 Oct, 2021. Accedido: 01 Oct, 2021. [Online]. Disponible: https://docs.python.org/3/tutorial/classes.html.