



Autómata Pushdown Determinista (APD)

Tarea – Informe final

Asignatura: INFO 139 – Teoría de Autómatas

Profesora: María Eliana de la Maza W.

Fecha: 12 de junio de 2025

Integrantes del grupo: Francisco Hernández, Ángel Leal, Matías Toledo

Contenido

| | |
|--|----|
| Introducción..... | 3 |
| Desarrollo | 4 |
| Estructuras de datos utilizadas | 4 |
| Métodos y Algoritmos | 6 |
| Especificaciones del programa..... | 10 |
| Nombre del programa..... | 10 |
| Lenguaje de implementación..... | 10 |
| Requerimientos de hardware | 10 |
| Indicaciones ejecución del programa..... | 11 |
| Ejemplo de funcionamiento | 11 |
| Conclusiones | 13 |
| Bibliografía | 14 |

Introducción

Un Autómata Pushdown (APD) es un modelo teórico de computación que extiende las capacidades de un autómata finito mediante la incorporación de una pila (Academy, 2023a). Un APD tiene como objetivo, en la teoría de complejidad computacional, estudiar qué clase de lenguajes puede reconocer para comprender lenguajes de libre contexto (LLC), los cuales son generados mediante una gramática del mismo tipo (GLC) (Academy, 2023b). Básicamente, este modelo permite estudiar qué tipos de lenguajes pueden ser procesados por máquinas mediante el uso de memoria adicional.

Según Hopcroft et al. (2007), la teoría de autómatas analiza modelos abstractos de cálculo y su capacidad para procesar y reconocer diferentes tipos de lenguajes. En este contexto, el APD representa una mejora sobre el autómata finito no determinista (AFND), al permitir transiciones nulas (ϵ -transiciones) y la manipulación de una pila, donde es posible leer, insertar o eliminar el tope de esta.

La definición formal de un autómata pushdown es la siguiente:

$$M_1 = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

Donde:

- Q : Conjunto de estados finitos
- Σ : Alfabeto de entrada
- Γ : Alfabeto del stack
- δ : Función transición
- $q_0 \in Q$: Estado inicial
- $Z_0 \in \Gamma$: Símbolo inicial del stack o pila
- $F \subseteq Q$: Conjunto de estados finales

El comportamiento de un APD es clave en compiladores para analizar estructuras anidadas mediante una pila, aplicación en evaluación de expresiones, resolución de problemas recursivos, validación de protocolos, procesamiento de lenguaje natural y verificación de software (GeeksforGeeks, 2024).

El presente informe tiene como objetivo relacionar, realizar y documentar el proceso del desarrollo de un programa capaz de simular el comportamiento de un Autómata Pushdown Determinista (APD), reconociendo si las palabras de entrada pertenecen al lenguaje definido por sus transiciones. Además, se exige, para el programa, una implementación eficiente, con un diseño claro y auto explicativo.

Desarrollo

El desarrollo del programa se llevó a cabo durante distintas etapas, entre las cuales está el desarrollo de una interfaz con Tkinter (librería de Python), desarrollo de funciones para recopilación de datos y simulación del autómata. El programa va de la mano con una interfaz de fácil entendimiento y con ciertas instrucciones en sí misma, pero que no son necesarias para simular un APD.

Estructuras de datos utilizadas

1. Estado inicial y estado final

Los estados inicial y final se almacenan como strings, ya que los estados son representados simbólicamente y no a través de operaciones. Esta elección permite una manipulación sencilla en caso de querer hacer modificaciones de estado final o inicial. Estos serán usados para comprobar la validez de las transiciones ingresadas del APD.



```
1 estado_inicial = "q0"
2 estado_final = "qf"
```

Figura 1: Ejemplo de código en Python para estados iniciales y finales.
Fuente: Elaboración propia.

2. Stack del APD

El stack del APD es gestionado mediante un arreglo dinámico, representado en Python como una lista. Esta estructura se trata como una pila de tipo LIFO (*Last In, First Out*), es decir, el último elemento insertado en esta lista será el primero en salir. Cabe mencionar también que los símbolos que serán guardados en el stack son tipo char (strings de largo 1 en Python).



```
1 stack_ejemplo = ["R", "A", "A", "A"]
```

Figura 2: Ejemplo de código en Python para el stack, cabe destacar que el elemento más a la derecha de la lista corresponde al último ingresado. Fuente: Elaboración propia.

3. Transiciones del APD

Las transiciones se almacenan como un diccionario de tuplas, donde la clave del diccionario será una tupla de la función de transición formada por el estado actual, palabra de entrada y el tope del stack. El valor asociado a cada clave es otra tupla que indica el nuevo estado y la operación sobre el stack, donde los símbolos a apilar se representan como una lista de caracteres. Cabe destacar que todos los valores de las tuplas serán strings y, la palabra vacía, será un string de largo cero.

Esta decisión permite agilizar la búsqueda de cada transición sin necesidad de recorrer un arreglo para encontrarla y, de esta forma, garantizar la eficiencia y optimización del rendimiento del programa.



```
1 transiciones = {
2     ("q0", "a", "R"): ("q0", ["A", "A", "R"]),
3     ("q0", "a", "A"): ("q0", ["A", "A", "A"]),
4     ("q0", "b", "A"): ("q1", [""]),
5     ("q1", "b", "A"): ("q1", [""]),
6     ("q1", "", "R"): ("q1", [""])
7 }
8
```

Figura 3: Ejemplo de código en Python para almacenamiento de transiciones.

Fuente: Elaboración propia.

4. Palabras de entrada

Las palabras de entrada se almacenan en una lista de listas que contienen a los caracteres de las entradas, es decir, cada símbolo es un elemento dentro de la lista interna almacenado como char (strings de largo 1 en Python). Esta elección ayuda a realizar una única ejecución para evaluar las palabras, ya que cada APD ingresado debe analizar más de una sola entrada.



```
1 palabras_entrada = [
2     ["a", "a", "b"],
3     ["a", "b"],
4     ["c", "b", "c"]
5 ]
```

Figura 4: Ejemplo de código en Python para almacenamiento de palabras de entrada.

Fuente: Elaboración propia.

De esta manera, la solución propuesta entrega una base sólida para el cumplimiento de la tarea asignada, además de claridad y eficiencia.

Para finalizar se crea un ejecutable con la versión final del programa para facilitar su uso en otros equipos de forma autónoma y sencilla.

Métodos y Algoritmos

1. Recolección de entradas

Haciendo uso de los métodos de Tkinter (librería de Python usado para programar la interfaz gráfica), tenemos acceso a métodos de creación de ventanas, botones, menús, etc. Uno de estos es el método “Entry” (Figura 4), utilizado para definir casillas de entrada, donde el usuario puede ingresar una cadena de texto cualquiera, la que después es recolectada a través de un evento, para el caso de la Figura 4, este evento “<Return>” corresponde a cuando el usuario presiona la tecla “Enter” de su computador.



Figura 5: Ejemplo de definición de una casilla de entrada para las palabras de entrada, a la cual se le define sus dimensiones ("width"), ubicación ("grid") y acción a realizar ("bind"). Fuente: Elaboración propia.

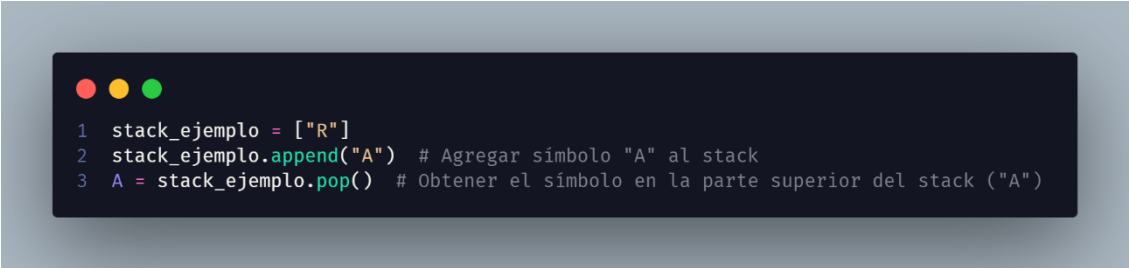
Luego de ingresados estos datos de entrada en las casillas correspondiente, usamos otro método de Tkinter llamado “.get()” (Figura 5), donde “palabras_entradas” es un arreglo global que guarda las casillas de entrada definidas en la Figura 4, y “palabras_entrada” es un arreglo global que guardará, también en un arreglo global, las palabras de entrada ingresadas por el usuario, para su posterior evaluación.



Figura 6: Ejemplo de recolección de palabras de entrada con el método “.get()”, las cuales son guardadas en un arreglo global para su posterior evaluación. Fuente: Elaboración propia.

2. Manejo del Stack

Para manejar el stack del APD (Figura 2), usaremos los métodos de listas de Python que nos permiten manejar esta estructura de datos, como si fuera un stack (estructura tipo LIFO), usando los métodos “.pop()”, el cual saca el último elemento de la lista, y el método “.append(X)”, método que agrega un elemento “X” a la lista, al final de esta (última posición).

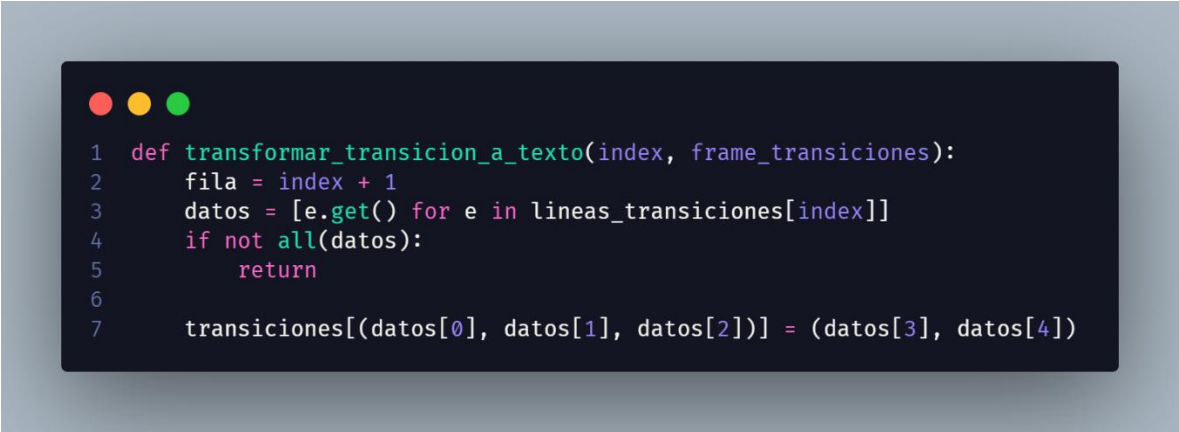


```
1 stack_ejemplo = ["R"]
2 stack_ejemplo.append("A") # Agregar símbolo "A" al stack
3 A = stack_ejemplo.pop() # Obtener el símbolo en la parte superior del stack ("A")
```

Figura 7: Código utilizado para convertir las transiciones del APD, ingresadas por el usuario, en un diccionario de transiciones. Fuente: Elaboración propia.

3. Diccionario Transiciones

Como se define en la sección anterior (Figura 3), las transiciones del APD son guardadas en un diccionario de tuplas, tanto para la llave como para el valor. Gracias a esto, y como se observa en la Figura 8, con los datos ingresados en las casillas de entrada, se generan y guardan estas transiciones ingresadas por el usuario, en un diccionario global llamado “transiciones”.



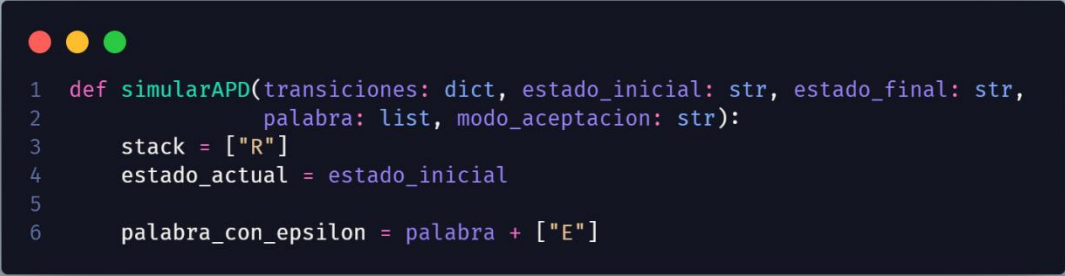
```
1 def transformar_transicion_a_texto(index, frame_transiciones):
2     fila = index + 1
3     datos = [e.get() for e in lineas_transiciones[index]]
4     if not all(datos):
5         return
6
7     transiciones[(datos[0], datos[1], datos[2])] = (datos[3], datos[4])
```

Figura 8: Código utilizado para convertir las transiciones del APD, ingresadas por el usuario, en un diccionario de transiciones. Fuente: Elaboración propia.

4. Simulación APD

En esta sección se describe, en tres partes, el funcionamiento de la función “simularAPD ()”, la cual es responsable de tomar el diccionario de transiciones, el estado inicial y final ingresados (Strings), la palabra de entrada (Lista de caracteres) a evaluar y el modo de aceptación, con el fin de evaluar si el APD determinista descrito acepta o rechaza la palabra de entrada evaluada.

Para empezar, y como se observa en la Figura 9, se inicializa el stack que el APD determinista usará, con un símbolo inicial “R” (línea 2 Figura 9), la variable “estado_actual” se inicializa con el valor del estado inicial ingresado (línea 3 Figura 9), y a la palabra a evaluar se le agrega un carácter “E” al final (línea 5 Figura 9), con el fin de fácilmente determinar el fin de la palabra.



```
1 def simularAPD(transiciones: dict, estado_inicial: str, estado_final: str,
2               palabra: list, modo_aceptacion: str):
3     stack = ["R"]
4     estado_actual = estado_inicial
5
6     palabra_con_epsilon = palabra + ["E"]
```

Figura 9: Inicio de la función “simularAPD()”, donde se inicializan las variables necesarias para la simulación del APD determinista ingresado por el usuario. Fuente: Elaboración propia.

Tras la inicialización de las variables, se procede a recorrer la palabra de entrada, carácter por carácter, utilizando un ciclo de iteración “for”. Para cada carácter, primero, se obtiene (si se puede) el símbolo al tope del stack (línea 3 Figura 10), con el que se arma la clave que se ingresa al diccionario para obtener el resultado de la transición (línea 6 Figura 10). Si esta clave no se encuentra en el diccionario, significa que la transición no es válida, por lo tanto, el APD rechaza la palabra (líneas 8 a 12 Figura 10).



```
1 for simbolo in palabra_con_epsilon:
2     if stack:
3         tope_pila = stack[-1]
4     else:
5         tope_pila = None
6     clave = (estado_actual, simbolo, tope_pila)
7
8     if clave not in transiciones:
9         # Si es epsilon y no hay transición, continuamos (no es error)
10        if simbolo == "E":
11            break
12        return False # No hay transición válida para simbolo normal
13
14    nuevo_estado, accion_pila = transiciones[clave]
15    estado_actual = nuevo_estado
16
17    if stack:
18        stack.pop()
19
20    for simbolo_pila in reversed(accion_pila):
21        if simbolo_pila != "E" and simbolo_pila != "":
22            stack.append(simbolo_pila)
```

Figura 10: Ciclo de iteración de la palabra de entrada realizado por la función “simularAPD()”, donde se evalúa, carácter a carácter, las transiciones del APD determinista descrito por el usuario. Fuente: Elaboración propia.

Luego, si esta transición es válida (existe en el APD), se guarda el nuevo estado y las acciones a realizar sobre la pila (líneas 14 y 15 Figura 10). Posteriormente, se itera sobre la variable “accion_pila” (obtenida de la transición válida), para agregar estos caracteres al stack (filas 17 a 22 Figura 10).

Finalmente, y dependiendo del método de aceptación del APD determinista ingresado por el usuario (si acepta por estado final o stack vacío). En el caso de que el APD determinista acepte por estado final, se compara la variable “estado_actual” con el estado final ingresado por el usuario (línea 3 Figura 11). Por otra parte, si el APD determinista acepta por stack vacío, se revisa si el stack está vacío o no (línea 5 Figura 11). Cabe mencionar que la función retorna un booleano (verdadero o falso) dependiendo de esta evolución mencionada, aceptando o rechazando la palabra de entrada.



```
1 # Verificación de aceptación
2     if modo_aceptacion == "estado_final":
3         return estado_actual == estado_final
4     elif modo_aceptacion == "stack_vacio":
5         return len(stack) == 0
6     return False
```

Figura 11: Final de la función “simularAPD()”, donde se evalúa, dependiendo del modo de aceptación de palabras por parte del APD determinista (estado final o stack vacío), si se acepta o rechaza la expresión. Fuente: Elaboración propia.

Especificaciones del programa

Nombre del programa

El nombre asignado al programa es: *Simulador de Autómata Pushdown Determinista (APD)*.

Lenguaje de implementación

El lenguaje de programación seleccionado para implementar el simulador de APD es Python. Esta elección se debe a la amplia disponibilidad de bibliotecas y herramientas para la gestión de estructura de datos, soporte de interfaces gráficas, como Tkinter, y la creación de un ejecutable del programa. Además, facilita el desarrollo colaborativo, legibilidad de código y mantenimiento de este. Se posee una base fuerte para el uso del lenguaje de programación seleccionado.

Requerimientos de hardware

El programa no necesita de un hardware potente. Sus requerimientos mínimos serían los siguientes:

- Cualquier procesador de 32 o 64 bits, con velocidad de 1GHz o superior ya que la aplicación es principalmente de interfaz gráfica sin cálculos intensivos.
- Mínimo 512 MB de RAM, aunque se recomienda 1 GB RAM o más. Se llegó a esta decisión ya que el programa maneja estructuras de datos simples (listas, diccionarios) para las transiciones y palabras.
- Un almacenamiento mínimo de 16 GB. Aunque el simulador no pesa mucho (menos de 1 MB), se recomienda tener espacio libre de sobra para evitar problemas de rendimiento.
- Cualquier sistema operativo es compatible con el programa.

Este programa funcionaría perfectamente incluso en hardware muy antiguo, ya que las operaciones son principalmente de interfaz gráfica y lógica de autómatas finitos.

Indicaciones ejecución del programa

Al seleccionar el archivo ejecutable se abrirá el programa mostrando la interfaz gráfica con 3 secciones.

- La sección izquierda permite definir las transiciones del APD simulado, siguiendo el formato: δ (estado actual, símbolo a leer, tope del stack) = (nuevo estado, nuevo tope). Las transiciones serán enumeradas y se pueden agregar cuantas se necesiten, para definir las correctamente se pulsa la tecla ENTER. En el caso de algún error a la hora de definir una transición ya guardada existe la opción para editar transiciones.
- La sección del medio corresponde a la declaración del estado inicial como a la forma como el autómata acepta por stack vacío o por estado final, de ser este último se solicitará ingresar el estado final del autómata.
- Finalmente, la sección derecha permite registrar la lista de palabras que se desean verificar a través del autómata, enumerándose y añadiéndose a medida que se registran con la tecla ENTER.

Luego de definir toda la entrada del autómata para ejecutar la salida se selecciona el botón de “Simular”. Para la salida, a la derecha de las palabras de entrada se mostrarán con un tick o palomilla (“✓”) las palabras que sean aceptadas por el APD, mientras que para las palabras rechazadas aparecerá una equis (“X”).

Ejemplo de funcionamiento

Para denotar un ejemplo de funcionamiento se eligió el siguiente lenguaje: Sea $L = \{a^n b^n \mid n \geq 0\}$. (Palabras con cierta cantidad de ‘a’ seguida por la misma cantidad de ‘b’). A continuación, se muestra paso a paso como ingresar el lenguaje al simulador:

1. Al abrir el programa se despliega la siguiente ventana:

Simulador de Autómata Pushdown Determinista (APD)

$\Sigma = \{ \}$

Transiciones del APD

δ (q, símbolo, tope de la pila) = (q', acción de la pila)

1. δ ([] , [] , []) = ([] , [])

2. δ ([] , [] , []) = ([] , [])

Eliminar

Agregar transición Editar transiciones

Estados

Estado inicial: []

Aceptación por:

☒ Estado final

☐ Stack vacío

Estado final: []

Palabras de entrada

1. []

2. []

Eliminar

Agregar palabra Editar palabras

Condiciones de uso

El símbolo Epsilon está representado por una E.

El estado inicial debe incluirse en la transición uno.

El estado q' es el estado al que q pasará.

Para una palabra aceptada se mostrará: ✓, mientras que para una palabra rechazada se mostrará: X.

Simular Limpiar Salir

En esta ventana se muestran las secciones de transición, estados, palabras de entrada y condiciones de uso. Estas secciones muestran claramente con que se tiene que rellenar cada entrada. Para las transiciones hay un símbolo sigma que muestra el lenguaje del APD ingresado.

2. Se definen las transiciones del APD:

$\Sigma = \{a, b\}$

Transiciones del APD

$\delta (q, \text{símbolo, tope de la pila}) = (q', \text{acción de la pila})$

| | | | |
|----|----------------------|---------------|----------|
| 1. | $\delta (q_0, a, R)$ | $= (q_0, AR)$ | |
| 2. | $\delta (q_0, a, A)$ | $= (q_0, AA)$ | Eliminar |
| 3. | $\delta (q_0, b, A)$ | $= (q_1, E)$ | Eliminar |
| 4. | $\delta (q_1, b, A)$ | $= (q_1, E)$ | Eliminar |
| 5. | $\delta (q_1, E, R)$ | $= (q_2, R)$ | Eliminar |

Agregar transición Editar transiciones

Se rellena cada campo escribiendo las transiciones en el formato correspondiente. El botón “Agregar transición” agrega una línea para poner más transiciones y el “Eliminar” borra la línea de transición. Mientras que el botón editar no realiza nada hasta que no se haya dado a “Simular”.

3. Se establece el estado inicial y la condición de aceptación del autómata, la opción para definir el estado final se bloquea si se escoge aceptar por stack vacío:

Estados

Estado inicial:

Aceptación por:

☒ Estado final

☐ Stack vacío

Estado final:

4. Se escribe la lista de palabras que se desean verificar en el simulador:

Palabras de entrada

| | | |
|----|------------------------------------|----------|
| 1. | <input type="text" value="aabb"/> | |
| 2. | <input type="text" value="ab"/> | Eliminar |
| 3. | <input type="text" value="abc"/> | Eliminar |
| 4. | <input type="text" value="aab"/> | Eliminar |
| 5. | <input type="text" value="abbbb"/> | Eliminar |

Agregar palabra Editar palabras

5. Una vez rellenados los campos requeridos se presiona el botón “Simular” y se mostrará la siguiente pantalla:

Simulador de Autómata Pushdown Determinista (APD)

$\Sigma = \{a, b\}$

Transiciones del APD

$\delta(q, \text{símbolo, tope de la pila}) = (q', \text{acción de la pila})$

1. $\delta(q_0, a, R) = (q_0, AR)$
2. $\delta(q_0, a, A) = (q_0, AA)$
3. $\delta(q_0, b, A) = (q_1, E)$
4. $\delta(q_1, b, A) = (q_1, E)$
5. $\delta(q_1, E, R) = (q_2, R)$

Eliminar

Agregar transición Editar transiciones

Estados

Estado inicial:

Aceptación por:

☒ Estado final

☐ Stack vacío

Estado final:

Palabras de entrada

1. aabb ✓
2. ab ✓
3. abc ✗
4. aab ✗
5. abbbb ✗

Eliminar

Agregar palabra Editar palabras

Condiciones de uso

El símbolo Epsilon está representado por una E.

El estado inicial debe incluirse en la transición uno.

El estado q' es el estado al que q pasará.

Para una palabra aceptada se mostrará: ✓, mientras que para una palabra rechazada se mostrará: ✗

Simular Limpiar Salir

Por último, explicando los demás botones, para eliminar palabras y/o transiciones basta con presionar el botón “Eliminar” y estas desaparecerán, el botón de “Limpiar” borra todos los campos rellenos y los botones de editar funcionan una vez esté hecha la simulación para activar los campos y volver a rellenarlos para corregir algún error.

Conclusiones

Durante el desarrollo de este simulador de Autómatas Pushdown Deterministas (APD), fue posible consolidar conceptos teóricos vistos en clases, mediante su implementación práctica mediante distintas estructuras de datos y lógica de programación. A continuación, se presentan algunas reflexiones sobre este trabajo:

- **Desafíos del desarrollo:** Una de las principales dificultades del desarrollo de esta aplicación fue simular el funcionamiento del APD de forma precisa y eficiente. Fue necesario tener especial cuidado al definir el flujo de las transiciones, la validación del estado actual y del elemento al tope del stack. Por otra parte, el uso de la librería Tkinter de Python, para la programación de la interfaz gráfica, presentó ciertos retos en cuanto al manejo de eventos de edición de las entradas dadas por el usuario.
- **Ventajas del uso de estructuras clave-valor:** El uso de diccionarios en Python para representar las transiciones del autómata pushdown determinista ingresado por el usuario, permitió una manipulación eficiente de los datos. Esta estructura clave-valor facilitó la búsqueda y redujo la complejidad del algoritmo al evitar iteraciones innecesarias para validar cada posible transición ante las variables entregadas.
- **Otros lenguajes de programación:** Si bien Python facilitó la implementación de este simulador, debido a su sencilla sintaxis, una implementación en otros lenguajes como C++ no hubiera sido mucho más complejo, hubiera requerido una gestión más explícita de memoria, probablemente con un mejor rendimiento. El uso de estructuras de datos

en C++, como mapa y vector, serían alternativas eficientes al diccionario y a las listas de Python.

- **Posibles mejoras futuras:** Una mejora inmediata sería extender la funcionalidad del simulador, para que también se puedan simular Autómatas Pushdown No Deterministas (APND), lo cual implicaría una revisión no solo del algoritmo de simulación del APD, sino que también de cómo se toman los datos (ya que una transición puede llevar a varios resultados). Esto permitiría modelar una gama más amplia de lenguajes de libre contexto.
Otra mejora puede ser permitir la visualización de las descripciones instantáneas al momento de simular una palabra en el APD determinista descrito por el usuario, con el fin de entender de mejor manera como el APD acepta o rechaza la palabra de entrada.

A modo de resumen, este trabajo no solo permite consolidar conocimientos teóricos sobre autómatas pushdown deterministas, sino también enfrentarse a los desafíos propios de un software con fin educativo, equilibrando calidad, funcionalidad y eficiencia.

Bibliografía

- Academy, E. (2023a, August 2). *¿Cómo funciona un autómata pushdown al reconocer una cadena de terminales?* - Academia EITCA. EITCA Academy. <https://es.eitca.org/la-seguridad-cibern%C3%A9tica/eitc-es-fundamentos-de-la-teor%C3%ADa-de-la-complejidad-computacional-cctf/aut%C3%B3mata-pushdown/equivalencia-de-cfgs-y-pdas/examen-revisi%C3%B3n-equivalencia-de-cfgs-y-pdas/%C2%BFC%C3%B3mo-funciona-un-aut%C3%B3mata-pushdown-para-reconocer-una-cadena-de-terminales%3F/>
- Academy, E. (2023b, August 2). *¿Cuál es el propósito de un autómata pushdown (PDA) en la teoría de la complejidad computacional y la ciberseguridad?* - Academia EITCA. EITCA Academy. <https://es.eitca.org/la-seguridad-cibern%C3%A9tica/eitc-es-fundamentos-de-la-teor%C3%ADa-de-la-complejidad-computacional-cctf/aut%C3%B3mata-pushdown/pdas-pushdown-aut%C3%B3matas/examen-revisi%C3%B3n-pdas-pushdown-aut%C3%B3matas/%C2%BFCu%C3%A1l-es-el-prop%C3%B3sito-de-un-aut%C3%B3mata-pushdown-pda-en-la-teor%C3%ADa-de-la-complejidad-computacional-y-la-ciberseguridad%3F/>
- GeeksforGeeks. (2025, May 1). *Applications of various Automata*. GeeksforGeeks. <https://www.geeksforgeeks.org/applications-of-various-automata/>
- Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2007). *Introduction to Automata Theory, Languages, and Computation*. Pearson.