LyCenciados Corruptos

Universidad Nacional Mayor de San Marcos
Facultad de Ingeniería de Sistemas e Informática

Curso: Lenguajes y Compiladores
Tema: Automata a Pila
Docente: Gelber Christian Uscuchagua Flores

2025

Grupo 01:

- Bayes Enriquez Eva María Florisa
- Gonzales Mendieta Claudio Camilo
- Melendez Blas Jhair Roussell
- Pacotaype Chuchon, Diego Alonzo
- Torres Tineo Cristhian Anthony

Problemática

Los autómatas finitos deterministas (AFD) son adecuados para reconocer lenguajes regulares, pero presentan limitaciones al enfrentarse a lenguajes libres de contexto (LLC). Esta restricción se debe a que los AFD carecen de una memoria adicional que les permita manejar estructuras más complejas, como las que se encuentran en los LLC. Por lo tanto, es necesario utilizar modelos más potentes, como los autómatas de pila, para reconocer estos lenguajes (Cueva Lovelle, J. M. , 2001)

Además, es importante destacar que un reconocedor diseñado para un LLC no necesariamente puede reconocer un lenguaje regular, ya que los LLC abarcan una clase más amplia de lenguajes que incluye a los lenguajes regulares como un subconjunto. Esto implica que las capacidades de reconocimiento de un autómata para LLC son más generales y no se limitan a las estructuras más simples de los lenguajes regulares (Formella, A. , 2007).

Por lo tanto, al diseñar reconocedores para lenguajes formales, es crucial seleccionar el modelo de autómata adecuado que corresponda al tipo de lenguaje que se desea reconocer, asegurando así una interpretación precisa y eficiente.

Solución

Para abordar el problema de reconocer lenguajes libres de contexto (LLC), que no pueden ser procesados por autómatas finitos deterministas (AFD) debido a su falta de memoria, se recurre al Autómata a Pila (AP) como solución. El AP introduce una estructura de memoria en forma de pila, permitiendo el manejo de dependencias y estructuras recursivas presentes en los LLC. Esta capacidad lo convierte en un modelo computacional más potente que los AFD, aunque aún limitado en comparación con los autómatas de Turing (Rincón, L. , s.f.).

Formalmente, un AP se define como una 6-tupla $(Q, \Sigma, \Gamma, \delta, q_0, F)$, donde:

Q: Conjunto finito de estados.

Σ: Alfabeto de entrada.

Γ: Alfabeto de la pila.

δ: Función de transición que define las reglas de movimiento basadas en el estado actual, el símbolo de entrada y el contenido de la pila.

q_0: Estado inicial.

F: Conjunto de estados de aceptación.

A diferencia de los autómatas finitos, los AP pueden empujar (apilar) o desapilar símbolos de una pila, lo que les permite recordar información de la entrada durante el análisis. Esto es clave en la identificación de estructuras anidadas, como los paréntesis balanceados, la correspondencia entre abertura y cierre en expresiones matemáticas, y la validación de lenguajes con relaciones específicas, como $a^n b^n c^n$.

El mecanismo de transición del AP sigue reglas específicas que dictan el comportamiento del sistema. Cada regla tiene la forma (estado actual, símbolo de entrada, símbolo en la cima de la pila) → (nuevo estado, acción en la pila). Las acciones en la pila pueden ser empujar un símbolo, desapilar o no hacer nada, permitiendo gestionar la memoria de manera eficiente.

En el contexto del problema planteado, donde un lenguaje libre de contexto requiere ser reconocido, la implementación de un AP es la solución adecuada. Su capacidad de manejar información adicional mediante la pila lo convierte en el modelo ideal para procesar gramáticas libres de contexto y validar cadenas que siguen estructuras específicas. Además, su uso es fundamental en el desarrollo de análisis sintácticos en compiladores, donde los LLC se emplean para definir la estructura de los lenguajes de programación (Milesi Vidal, A. , 2019).

Objetivo

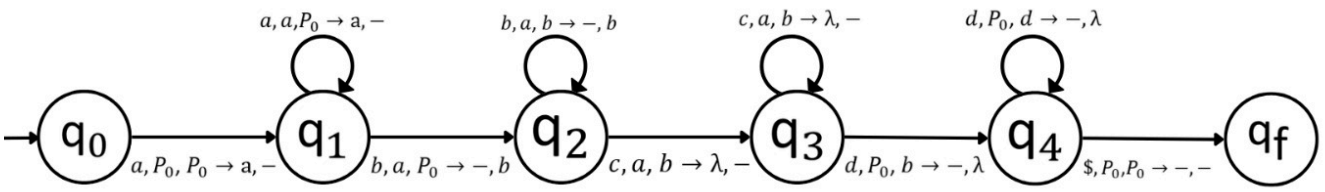
Proporcionar una comprensión detallada de los autómatas de pila, abarcando desde sus fundamentos teóricos hasta su aplicación práctica. Para ello, se explicarán conceptos clave relacionados con estos autómatas, se resolverá un ejercicio asignado que incluye su representación mediante grafos, y se desarrollará su correspondiente implementación en código.

Desarrollo

En esta sección, profundizaremos en la representación gráfica y la implementación en código de los autómatas de pila. Comenzaremos analizando su estructura mediante grafos, los cuales ilustran cómo el autómata cambia de estado en respuesta a diferentes símbolos de entrada y operaciones en la pila. Posteriormente, presentaremos una implementación práctica en lenguaje Python, que permitirá visualizar el funcionamiento de estos autómatas en un entorno de programación real.

Representación en Grafos

9. $L = \{ a^m b^n c^m d^n / \text{ m>=1, n>=1} \}$



Representación en Reglas de Transición

(q0, a, P0, P0) → (q1, a, -)
(q1, a, a, P0) → (q1, a, -)
(q1, b, a, P0) → (q2, -, b)
(q2, b, a, b) → (q2, -, b)
(q2, c, a, b) → (q3, λ, -)
(q3, c, a, b) → (q3, λ, -)
(q3, d, P0, b) → (q4, -, λ)
(q4, d, P0, b) → (q4, -, λ)
(q4, $, P0, P0) → (qF, -, -)

Código en Python

El código del ejercicio se encuentra presente en el siguiente repositorio: https://github.com/CTDreamer/AutomataPila.git

Conclusiones

El desarrollo de este trabajo permitió comprender en profundidad el funcionamiento de los autómatas a pila (AP) y su importancia en el reconocimiento de lenguajes libres de contexto (LLC). Se pudo evidenciar que los autómatas finitos deterministas (AFD) no son suficientes para procesar estructuras recursivas, lo que hace que los AP sean una solución más adecuada gracias a su capacidad de manejar memoria en forma de pila.

Mediante la implementación de un autómata a pila en Python, se logró verificar la correcta aceptación de cadenas pertenecientes a un lenguaje específico. Se utilizaron dos pilas para modelar las transiciones necesarias, permitiendo gestionar la correspondencia entre los símbolos de entrada de manera eficiente. Esto permitió analizar el comportamiento del

autómata y visualizar cómo las operaciones de apilado y desapilado determinan la validez de una cadena.

Además, la representación de las transiciones en grafos facilitó la interpretación del comportamiento del autómata, proporcionando una visión más clara de los estados y las reglas que rigen su funcionamiento. Este enfoque gráfico complementa la implementación en código, asegurando una comprensión integral del modelo computacional utilizado.

Referencias

1. Cueva Lovelle, J. M. (2001). Lenguajes, gramáticas y autómatas (2ª ed.). Universidad de Oviedo. Recuperado de https://reflection.uniovi.es/ortin/publications/automata.pdf
2. Formella, A. (2007). Teoría de autómatas y lenguajes formales. Universidad de Vigo. Recuperado de https://formella.webs.uvigo.es/doc/talf07/talf.pdf
3. Milesi Vidal, A. (2019). Teoría de autómatas (Trabajo final de grado). Universitat de Barcelona. Recuperado de https://diposit.ub.edu/dspace/bitstream/2445/150797/2/150797.pdf
4. Rincón, L. (s.f.). Autómatas de pila y lenguajes independientes del contexto. Recuperado de https://libroweb.alfaomega.com.mx/book/681/free/ovas_statics/cap9/Automatas%20de%20Pila%20y%20Lenguajes%20independientes%20del%20contexto.%20Rincon%2C%20Luis.pdf
5. Uscuchagua G. (2025). Notas Magistrales

Anexos

A1. Código Desarrollado por LyCenciados Corruptos

```
# Grupo: LyCenciados Corruptos
# Este código implementa un Autómata a Pila (AP) que verifica si una
cadena
# pertenece a un lenguaje específico basado en las reglas de apilado y
desapilado.
# Utiliza dos pilas para controlar la correspondencia entre los
símbolos.

class AutomataPila:
    def __init__(self):
        self.pila_1 = []  # Pila para manejar los símbolos 'a' y 'c'
        self.pila_2 = []  # Pila para manejar los símbolos 'b' y 'd'

    def procesar_cadena(self, cadena):
        for simbolo in cadena:
            if simbolo == 'a':
                self.pila_1.append('a')  # Apilar 'a' en pila_1
            elif simbolo == 'b':
                self.pila_2.append('b')  # Apilar 'b' en pila_2
            elif simbolo == 'c':
                if self.pila_1:
                    self.pila_1.pop()  # Desapilar 'a' si existe en
pila_1
                else:
                    return False  # Error: no hay 'a' para desapilar
            elif simbolo == 'd':
                if self.pila_2:
                    self.pila_2.pop()  # Desapilar 'b' si existe en
pila_2
                else:
                    return False  # Error: no hay 'b' para desapilar
            else:
                return False  # Error: símbolo no válido

        # Acepta la cadena si ambas pilas están vacías al finalizar
        return len(self.pila_1) == 0 and len(self.pila_2) == 0

if __name__ == "__main__":
    automata = AutomataPila()
```

```
    cadena = input("Ingrese una cadena: ")  # Solicita cadena desde la
terminal
    resultado = automata.procesar_cadena(cadena)
    print("Cadena pertenece al lenguaje" if resultado else "Cadena
rechazada, no pertenece al lenguaje")
```