Practica 6

Alumno: *Ángel López Manríquez*
Tema: *Automatas de pila*
Fecha: *17 de junio de 2019*
Grupo: *2CV1*

M. en C. Luz María Sánchez García

Instituto Politecnico Nacional
Escuela Superior de Cómputo

Índice

Automatas de pila

Lopez Manriquez Angel 2CV1

Mayo 2018

1. Introducción

En esta práctica se implementará el Autómata Finito con Pila Determinista (AFPD), con el objetivo de verificar si determinadas cadenas pertenecen al lenguaje generado por la GLC que representa al autómata.

1.1. Definición de AFPD

El AFPD es el modelo de autómata requerido para aceptar los lenguajes libres de contexto, que a diferencia de un AFD normal, solo acepta lenguajes regulares.

Un AFPD es una tupla de 7 elementos: $M = (Q, q_0, F, \Sigma, \Gamma, z_0, \delta)$, donde:

- Q es el conjunto finito de estados.
- $q_0 \in Q$ es el estado inicial.
- $F \subseteq Q$ es el conjunto no vacío de estados finales o de aceptación.
- Σ es el alfabeto finito de entrada, también llamado alfabeto de cinta.
- Γ es el alfabeto finito de pila.
- $z_0 \in \Gamma$ es el símbolo inicial de la pila o el marcador de fondo, donde $z_0 \notin \Sigma$.
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*$ es la función de transición del autómata. Es de la forma $\delta(q, a, s) = (q', \alpha)$.

1.2. Descripción

Un AFPD procesa cadenas sobre una cinta de entrada (la cadena de entrada) justo como en un AFD, pero hay una cinta adicional (la pila) que es utilizada por el autómata como lugar de almacenamiento. En un momento determinado estando en un estado q, se escanea un símbolo a sobre la cinta de entrada y el símbolo s en el tope de la pila. De esta forma la transición $\delta(q, a, s) = (q', \alpha)$ representa un paso computacional: pasamos al estado q', extraemos el caracter del tope de la pila e insertamos la cadena α en el tope de la pila, caracter por caracter.

Recordemos que la pila es una estructura de datos del tipo LIFO (Last In First Out), por lo que en todo momento el autómata solo tiene acceso al símbolo que está en el tope de la pila, y el contenido de la pila siempre se lee de arriba (tope) hacia abajo (fondo). También, por definición de δ, nunca podemos realizar alguna transición si la pila está vacía.

En caso de que alguna transición $\delta(q, a, s)$ no exista o no esté definida, se aborta el procesamiento de la cadena de entrada, tal y como ocurría en el AFD.

Algunos casos especiales de transiciones son:

- $\delta(q, a, s) = (q', s)$: el contenido de la pila no se altera, pues luego de borrar el símbolo s lo volvemos a insertar.
- $\delta(q, a, s) = (q', \varepsilon)$: se borra el símbolo del tope de la pila y en el siguiente paso se escanea el nuevo tope de la pila, que es el símbolo colocado justo debajo de s.
- $\delta(q, \varepsilon, s) = (q', \alpha)$: esta es una ε-transición o transición espontánea, en donde no procesamos el símbolo actual de la cadena de entrada pero en la pila borramos s e insertamos la nueva cadena α. De esta forma podemos modificar el contenido de la pila sin consumir símbolos de la cadena de entrada. Notemos que para que el autómata sea determinista, no podemos tener al mismo tiempo las transiciones $\delta(q, a, s)$ y $\delta(q, \varepsilon, s)$.

1.3. Configuración del autómata

La terna $(q, au, s\beta)$, donde $q \in Q$, $a \in \Sigma$, $u \in \Sigma^*$, $s \in \Gamma$ y $\beta \in \Gamma^*$, representa la **configuración instantánea** del autómata. Nos dice que el autómata está en el estado q, au es la parte no procesada de la cadena de entrada y estamos escaneando el símbolo a; además de que s es el símbolo colocado en el tope de la pila y $s\beta$ es el contenido total de la pila.

De esta forma, la notación $(q, au, s\beta) \vdash (q', u, \alpha\beta)$ nos indica los pasos que hace el autómata en cada escaneo, es decir, que estando en el estado q leyendo el caracter a de la cadena de entrada y s en el tope de la pila, hemos avanzado en la cadena de entrada ahora estando en el estado q' y reemplazando s por α en la pila. Esta notación es equivalente a decir que existe una transición $\delta(q, a, s) = (q', \alpha)$.

Y así, la notación $(q, u, \beta) \implies {}^*(q', v, \alpha)$ significa que el autómata pasa de la configuración instantánea (q, u, β) a (q', v, α) en cero o más pasos.

De forma natural, tenemos estos dos casos particulares de configuración instantánea:

- **Configuración inicial:** Es (q_0, w, z_0), donde $w \in \Sigma^*$ es la cadena de entrada. Cuando comenzamos a procesarla, el contenido de la pila es z_0 y estamos en el estado q_0.
- **Configuración de aceptación:** Es (p, ε, β), donde $p \in F$ y $\beta \in \Gamma^*$. Quiere decir que, para que una cadena sea aceptada, debe ser procesada completamente y el autómata debe terminar en un estado de aceptación.

Por último, definiremos al **lenguaje aceptado** por el AFPD M como $L(M) = \{w \in \Sigma^* \mid \exists (q_0, w, z_0) \implies {}^*(p, \varepsilon, \beta), p \in F\}$. Es decir, la cadena w es aceptada si se puede ir desde la configuración inicial hasta una configuración de aceptación, en cero o más pasos.

2. Diseño de la solución

Para saber si la palabra es aceptada por el automata de pila se procede a realizar el siguiente algoritmo

Algorithm 1 procesar palabra

```
function PARSE(w : string )
    q ← q_0
    for c ∈ w do
        if stack ≠ ∅ then
            (q, to_push) ← δ(q, c, stack.top)
            if stack = ∅ then
                return false
            end if
            if to_push = ϵ then
                continue
            end if
            stack.push(to_push)
        end if
    end for
    return q ∈ F
end function
```

3. Implementación

stack.py

```python
__author__ = 'Ang3l Lopez Manriquez'

class Stack(list):
    """ Tipo abstracto de dato de tipo LIFO. Hereda de list.
    """

    def __init__(self):
        """ Inicializamos el constructor de la superclase.
        """

        super().__init__()

    def push(self, e):
        self.append(e)

    def pop(self):
        if self: # si tenemos elementos
            return super().pop()
```

```python
        raise Exception("Pila vacia")

    def top(self):
        if self:
            return self[len(self) - 1]
        raise Exception("Pila vacia")

    def size(self):
        return len(self)

    def empty(self):
        return self.size() == 0
```

DPDA.py

```python
__author__ = 'Angel Lopez Manriquez'

import pdb
from stack import Stack

class DPDA:

    """ Implementacion de un automata de pila. """

    def __init__(self, states, q0, final, sigma, gamma, z0, epsilon = chr(949),
        empty_stack_as_valid = True, from_left = True):
        """ Para inicializar el automata de pila determinista se requiere de:

                states[int]: el numero de estados (q0, q1, ..., qn)
                q0[int]: el id del estado inicial
                final[set]: conjunto de estados finales del automata
                sigma[set]: conjunto de variables de entrada
                gamma[set]: conjunto de variables de la pila
                z0[str]: caracter que se apilara al inicializar el automata
                epsilon[str]: caracter que representa a la cadena vacia epsilon, por
                    defecto se pone la letra minuscula epsilon (el 949 es el valor es ASCII
                    de este)
                empty_stack_as_valid[bool]: si este valor es True definiremos al automata tal
                    que si en algun punto la pila es vacia la cadena es aceptada por el
                    lenguaje generado por este
                from_left[bool]: si es True, la cadena retornada por delta sera apilada
                    empezando por el caracter de la izquierda
        """

        self.states = states
```

```
        self.q0 = q0
        self.final = final
        self.sigma = sigma
        self.gamma = gamma
        self.z0 = z0
        self.epsilon = epsilon

        self.table = dict() # variable en la cual se guardan los valores al ejecutar add_transition

        self.stack = Stack() # pila del programa
        self.stack.push(z0) # apilamos el caracter z0

    def delta(self, state, sigma, gamma):
        """ La funcion de transicion delta se define como:
            delta: state x Sigma x Gamma -> Q x Gamma ** *

        Argumentos:
            state {int} -- Estado del automata.
            sigma {str} -- Simbolo de entrada.
            gamma {str} -- Simbolo de la pila.
        """
        # si obtenemos un caracter invalido, retornamos una bina
        if self.stack.empty() or gamma != self.stack.top() or (state, sigma, gamma) not in self.table:
            return None, None
        self.stack.pop()
        return self.table[(state, sigma, gamma)] # retornamos una cadena

    def parse(self, word):
        """ Metodo que determina si una cadena es valida por el automata de pila.

        Arguments:
            word {str} -- Posible palabra aceptada.

        Returns:
            bool -- True si pertenece, False en caso contrario.
        """

        state = self.q0
        stack = self.stack
        for char in word:
            if stack: # si la pila no esta vacia
                print('\ndomain', state, char, stack.top())
                state, to_push = self.delta(state, char, stack.top())
                print('codomain', state, to_push)
                if state == None: # No hay valor correspondiente para la letra actual
                    return False
                if to_push == self.epsilon: # no apilaremos nada en este paso
                    print('stack', stack)
```

```
                    continue
                to_push = list(to_push)
                if from_left:
                    to_push.reverse()
                while to_push: # mientras la pila tenga elementos
                    stack.push(to_push.pop())
                print('stack', stack)
            else:
                return self.empty_stack_as_valid
        return state in self.final # el estado pertenece a self.final ?

    def add_transition(self, origin, end):
        """ Agregamos una transision.

        Arguments:
            origin {int} -- Estado de origen
            end {int} -- Estado de destino

        Raises:
            Exception -- Se lanzara una excepcion en caso de se quiera agregar una
            transicion invalida.
        """

        state = origin[0]
        sigma = origin[1]
        gamma = origin[2]

        state_out = end[0]
        gamma_out = end[1]

        if state in range(0, self.states) and sigma in self.sigma and gamma in self.gamma:
            self.table[(state, sigma, gamma)] = (state_out, gamma_out)
        else:
            raise Exception("Valores invalidos.")
```

main.py

```
from DPDA import DPDA


def add_transitions():
    a = DPDA(3, 0, {2}, set('abc'), set('ABCDZ'), 'Z')
    a.add_transition((0, 'a', 'Z'), (0, 'ZC'))
    a.add_transition((0, 'b', 'Z'), (0, 'ZD'))
    a.add_transition((0, 'a', 'C'), (0, 'CA'))
    a.add_transition((0, 'a', 'D'), (0, 'DA'))
    a.add_transition((0, 'a', 'A'), (0, 'AA'))
    a.add_transition((0, 'a', 'B'), (0, 'BA'))
```

```
    a.add_transition((0, 'b', 'C'), (0, 'CB'))
    a.add_transition((0, 'b', 'D'), (0, 'DB'))
    a.add_transition((0, 'b', 'A'), (0, 'AB'))
    a.add_transition((0, 'b', 'B'), (0, 'BB'))
    a.add_transition((0, 'c', 'C'), (1, 'C'))
    a.add_transition((0, 'c', 'D'), (1, 'D'))
    a.add_transition((0, 'c', 'A'), (1, 'A'))
    a.add_transition((0, 'c', 'B'), (1, 'B'))
    a.add_transition((0, 'c', 'Z'), (2, 'Z'))
    a.add_transition((1, 'a', 'A'), (1, chr(949)))
    a.add_transition((1, 'b', 'B'), (1, chr(949)))
    a.add_transition((1, 'a', 'C'), (2, chr(949)))
    a.add_transition((1, 'b', 'D'), (2, chr(949)))
    return a

a = add_transitions()
print(a.parse('abacaba'))
print(a.parse('baacaab'))
print(a.parse('baca'))
```

4. Funcionamiento

El autómata de pila determinístico descrito en el archivo `in1.txt` es:

Aqui se muestra el procedimiento para un automata particular para realizar pruebas:

$Q = \{q_0, q_1, q_2\}$

$q_0 = q_0$

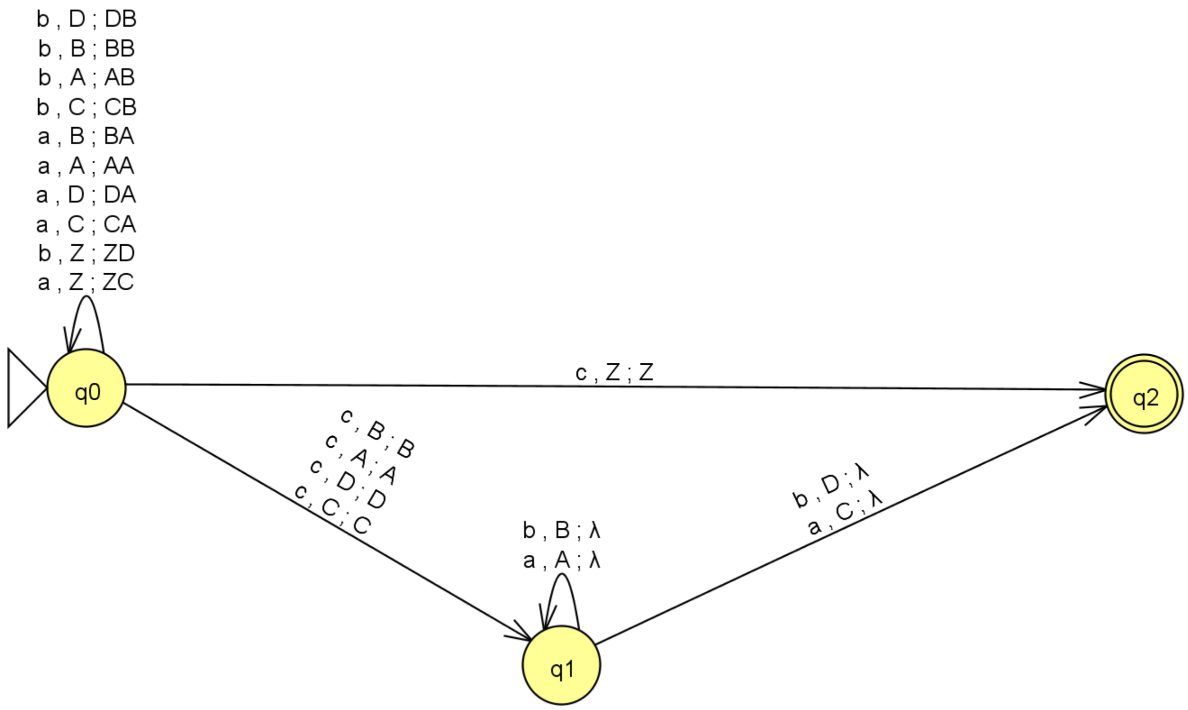
$F = \{q_2\}$

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

$\Gamma = \{Z, A, B, C, D\}$

$z_0 = Z$

$$\delta(q_0, a, Z) = (q_0, ZC)$$
$$\delta(q_0, b, Z) = (q_0, ZD)$$
$$\delta(q_0, a, C) = (q_0, CA)$$
$$\delta(q_0, a, D) = (q_0, DA)$$
$$\delta(q_0, a, A) = (q_0, AA)$$
$$\delta(q_0, a, B) = (q_0, BA)$$
$$\delta(q_0, b, C) = (q_0, CB)$$
$$\delta(q_0, b, D) = (q_0, DB)$$
$$\delta(q_0, b, A) = (q_0, AB)$$
$$\delta(q_0, b, B) = (q_0, BB)$$
$$\delta(q_0, c, C) = (q_1, C)$$
$$\delta(q_0, c, D) = (q_1, D)$$
$$\delta(q_0, c, A) = (q_1, A)$$
$$\delta(q_0, c, B) = (q_1, B)$$
$$\delta(q_0, c, Z) = (q_2, Z)$$
$$\delta(q_1, a, A) = (q_1, \varepsilon)$$
$$\delta(q_1, b, B) = (q_1, \varepsilon)$$
$$\delta(q_1, a, C) = (q_2, \varepsilon)$$
$$\delta(q_1, b, D) = (q_2, \varepsilon)$$



El lenguaje que genera es: $L = \{wcw^{-1} \mid w \in (a|b)^*\}$, es decir, todas las cadenas que son palíndromos con una c enmedio.

4.1. prueba 1

con $w = abacaba$ tenemos

```
C:\Users\ANGEL\Documents\codes\theory-of-computation\practice6>python main.py


domain 0 a Z
codomain 0 ZC
stack ['Z', 'C']

domain 0 b C
codomain 0 CB
stack ['Z', 'C', 'B']

domain 0 a B
codomain 0 BA
stack ['Z', 'C', 'B', 'A']

domain 0 c A
codomain 1 A
stack ['Z', 'C', 'B', 'A']

domain 1 a A
codomain 1 ε
stack ['Z', 'C', 'B']

domain 1 b B
codomain 1 ε
stack ['Z', 'C']

domain 1 a C
codomain 2 ε
stack ['Z']
True
```

vemos que $w \in L(A)$.

4.2. prueba 2

con $w = baacaab$ tenemos

```
domain 0 b Z
codomain 0 ZD
stack ['Z', 'D']

domain 0 a D
codomain 0 DA
stack ['Z', 'D', 'A']

domain 0 a A
codomain 0 AA
stack ['Z', 'D', 'A', 'A']

domain 0 c A
codomain 1 A
stack ['Z', 'D', 'A', 'A']

domain 1 a A
codomain 1 ε
stack ['Z', 'D', 'A']

domain 1 a A
codomain 1 ε
stack ['Z', 'D']

domain 1 b D
codomain 2 ε
stack ['Z']
True
```

vemos que $w \in L(A)$.

4.3. prueba 3

con $w = baca$ tenemos

```
domain 0 b Z
codomain 0 ZD
stack ['Z', 'D']

domain 0 a D
codomain 0 DA
stack ['Z', 'D', 'A']

domain 0 c A
codomain 1 A
stack ['Z', 'D', 'A']

domain 1 a A
codomain 1 ε
stack ['Z', 'D']
False
```

vemos que $w \notin L(A)$.

5. Conclusiones

En esta practica se aprecia una de las aplicaciones de las estructuras de datos pues el uso de la pila y el grafo conforman este automata de pila y nos brinda un algoritmo mas para determinar pertenencias de palabras a lenguajes generados por el mismo.