Instituto Tecnológico y de Estudios Superiores de Monterrey

Jesús Ramírez Delgado

A01274723

Implementacion de Metidos Computacionales

Actividad Integradora 2: Validación de entradas usando autómatas.

Para esta actividad usando Racket se debía hacer un programa desarrollando una función que reciba como entrada la descripción formal de un autómata de estado finito y una lista de entradas a validar y la función debe retornar una lista indicando si acepta las entradas a evaluar.

Un ejemplo utilizado en la resolución de esta situación problema fue el siguiente autómata:

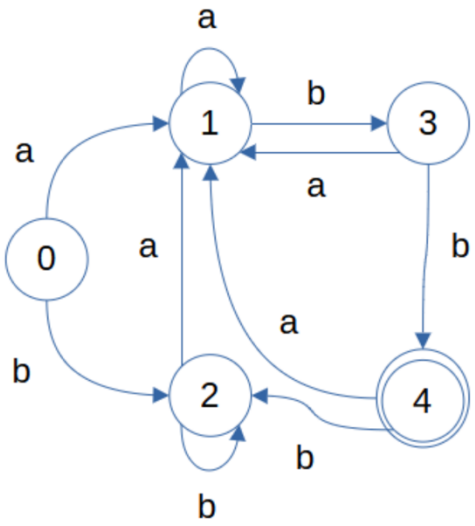


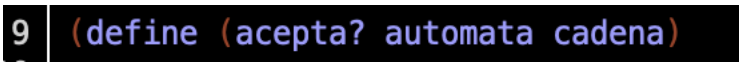
Ilustración 1

El cual se desarrolla con la siguiente lista profunda en Racket:

((0 1 2 3 4) (a b) ((0 a 1) (0 b 2) (1 a 1) (1 b 3) (2 b 2) (2 a 1) (3 a 1) (3 b 4) (4 a 1) (4 b 2)) 0 (4)). La lista profunda para describir las entradas sería: ((a b a b a a b b) (a a a a a a b) (a b a b a b b) (a b))
Y la salida esperada debería ser: (#t #f #t #f).

El código implementado fue el siguiente, desglosado en partes:

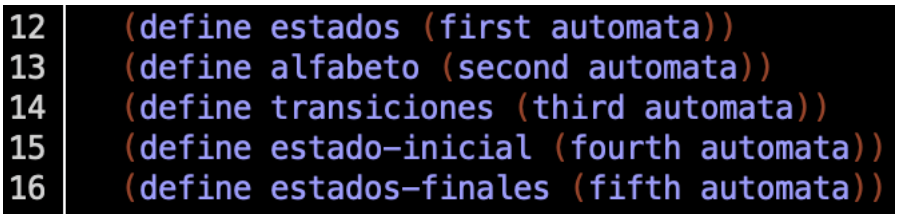
1. Primero creamos una función que determina si una cadena de entrada es acpetada por el autómata.



```
9  (define (acepta? automata cadena)
```

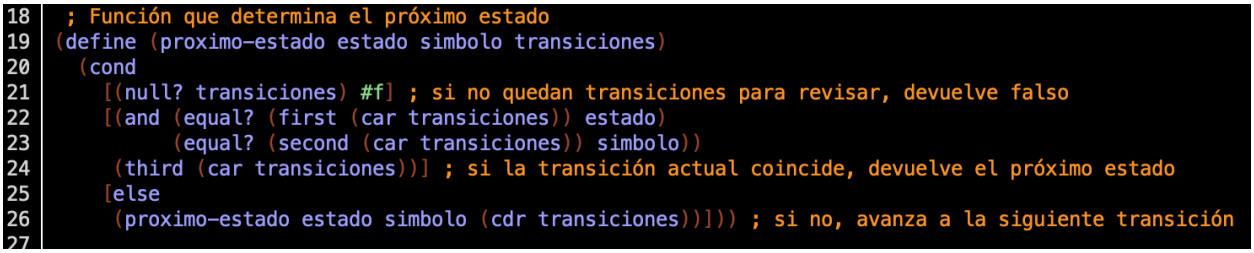
La complejidad de esta función esta ligada a la función **procesa-cadena** por lo que tiene una complejidad de O(m*n).

2. Ahora definimos varias funciones para extraer el contenido/componentes del automata.



```
12    (define estados (first automata))
13    (define alfabeto (second automata))
14    (define transiciones (third automata))
15    (define estado-inicial (fourth automata))
16    (define estados-finales (fifth automata))
```

3. Después se definido una función que determina el siguiente estado.



```
18  ; Función que determina el próximo estado
19 (define (proximo-estado estado simbolo transiciones)
20   (cond
21     [(null? transiciones) #f] ; si no quedan transiciones para revisar, devuelve falso
22     [(and (equal? (first (car transiciones)) estado)
23           (equal? (second (car transiciones)) simbolo))
24      (third (car transiciones))] ; si la transición actual coincide, devuelve el próximo estado
25     [else
26      (proximo-estado estado simbolo (cdr transiciones))])) ; si no, avanza a la siguiente transición
27
```

Esta función es lineal por lo que se puede decir que la complejidad de esta es O(n).

4. El siguiente paso es definir una función que ayude a procesar la cadena de entrada.

```
  ; Función auxiliar para procesar la cadena de entrada
  (define (procesa-cadena estado cadena)
    (if (null? cadena)
    (not (eq? #f (member estado estados-finales)))
        (let ([prox-estado (proximo-estado estado (car cadena) transiciones)])
          (if prox-estado
              (procesa-cadena prox-estado (cdr cadena))
              #f))))

  ; Llamar a la función auxiliar con el estado inicial y la cadena de entrada
  (procesa-cadena estado-inicial cadena))
```

¿Como se mencionó previamente en el caso de la función acepta?**,** la complejidad de esta función es igual de O(m*n) ya que se extraen los componentes y luego se llama a la función.

5. Finalmente, la última función **valida-cadenas** para procesar varias cadenas de entrada.

```
(define (valida-cadenas automata cadenas)
  (map (lambda (cadena) (acepta? automata cadena)) cadenas))
```

En este caso la complejidad es de O(k*m*n), esto debido a que hace una llamada a **acepta?** que tiene una complejidad de O(m*n) y k es la cantidad de cadenas en la lista.

Solamente definimos el autómata y ejecutamos las funciones.

```
; Ejemplo de uso
(define automata '((0 1 2 3 4)
                   (a b)
                   ((0 a 1) (0 b 2) (1 a 1) (1 b 3) (2 b 2) (2 a 1) (3 a 1) (3 b 4) (4 a 1) (4 b 2))
                   0
                   (4)))

(define automata2 '((0 1 2)
                          (0 1)
                          ((0 0 1) (0 1 0) (1 0 1) (1 1 2) (2 0 1) (2 1 0))
                          0
                          (2)))


(define entradas '((a b a b a a b b)
                   (a a a a a a b)
                   (a b a b a b b)
                   (a b)
                   (a b a b a a b b)))

(define entradas2 '((1 0 1)
                    (1 0 0 1)
                    (1 1 1)
                    (0 0 0 0)))


(display (valida-cadenas automata entradas))

(display (valida-cadenas automata2 entradas2))
```

Los resultados de salida fueron:

Con los resultados de la ejecución podemos corroborar que el código funciona correctamente y funciona con otros autómatas. En cuanto a la complejidad, puede ser que no sean los mas eficientes, pero en términos de practicidad son fáciles de programar. Se podría mejorar su eficiencia optimizando los algoritmos, utilizando alguna alternativa de estructura de datos o haciendo programación paralela.

Esta actividad integradora en lo personal me gustó mucho sobre todo por el uso de Racket, que es un lenguaje que a pesar de que no se utiliza mucho en la industria es algo diferente al paradigma al que estoy más acostumbrado y me enseña sus bondades de los lenguajes de programación funcionales y en este caso de un dialecto de Lisp.