

Practica 3

Alumno: Ángel López Manríquez

Tema: Automatas Finitos

Fecha: 17 de junio de 2019

Grupo: 2CV1

M. EN C. LUZ MARÍA SÁNCHEZ GARCÍA

Instituto Politecnico Nacional
Escuela Superior de Cómputo

Índice

1. Introducción	2
2. Planteamiento del problema	3
3. Diseño de la solución	3
3.1. Código	4
4. Funcionamiento	8
5. Conclusiones	8

Automatas finitos con transiciones vacias

Lopez Manriquez Angel 2CV1

Marzo 2018

1. Introducción

En esta práctica se implementará un Autómata Finito Determinista (AFD) en `Kotlin`, con el objetivo de que podamos verificar si diversas cadenas pertenecen al lenguaje generado por dicho AFND- ϵ .

Un **autómata finito** es un modelo matemático de una máquina que acepta cadenas de un lenguaje definido sobre un alfabeto Σ . Consiste en un conjunto finito de estados y un conjunto de transiciones entre esos estados, que dependen de los símbolos de la cadena de entrada. Decimos que el autómata finito *acepta* una cadena w si la secuencia de transiciones correspondientes a los símbolos de w nos lleva desde el **estado inicial** a uno de los estados finales.

Para definirlo formalmente, decimos que un autómata finito es una quintupla $A = (Q, \Sigma, \delta, e_0, F)$, donde:

- Q es el conjunto finito de estados.
- Σ es el alfabeto o conjunto finito de símbolos de entrada.
- δ es la función de transición de estados, que se define como:
 - Si el autómata es *determinista*, $\delta : Q \times \Sigma \rightarrow Q$, es decir, recibe un estado de origen y un elemento del alfabeto mediante el cual se moverá, para devolver el estado de destino.
 - Si el autómata es *no determinista*, $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$, es decir, recibe un estado de origen y un elemento del alfabeto mediante el cual se moverá, para devolver un conjunto de estados posibles de destino.
- e_0 es el estado inicial, donde $e_0 \in Q$.
- F es el conjunto de estados finales o de aceptación, donde $F \subseteq Q$.

Para que un autómata lea y valide una cadena w , vamos a definir algunos conceptos:

- **Configuración de un autómata finito:** es un par ordenado de la forma (q_i, W) , donde $q_i \in Q$ es el estado actual y W la cadena que queda por leer en ese instante.
- **Configuración inicial:** es el par (e_0, w) .
- **Configuración final:** es el par (f_i, ε) , donde $f_i \in F$ y ε representa que ya no queda nada por leer.
- **Movimiento de un autómata finito:** es la transición entre dos configuraciones, y se representa por $(q_i, cW) \rightarrow (\delta(q_i, c), W)$, es decir, leemos el carácter $c \in \Sigma$ del principio de la cadena actual estando en el estado $q_i \in Q$, y nos movemos a donde la transición indicada por ese estado y ese carácter (si existe) nos lleve, eliminando el carácter c de la cadena por leer.

Por lo tanto, decimos que la cadena w es aceptada por el autómata finito si y solo si terminamos en alguno de los estados finales, es decir, existe una secuencia de movimientos tal que $(e_0, w) \rightarrow \dots \rightarrow (f_i, \varepsilon)$, donde $f_i \in F$.

Para representar gráficamente un AFD, usamos un grafo dirigido, también llamado diagrama de transición de estados. Cada nodo del grafo corresponde a un estado y las aristas corresponden a las transiciones. El estado inicial se indica mediante una flecha que no tiene nodo origen, y los estados finales con un círculo doble. Si existe una transición del estado q_i al estado q_j mediante el carácter c (es decir, $\delta(q_i, c) = q_j$), se traza una arista que vaya de q_i a q_j y se etiqueta con el carácter c .

2. Planteamiento del problema

De la práctica anterior, se implementó un breve programa que determina si una línea de cabecera del lenguaje C para incluir una biblioteca estaba bien escrita con una expresión regular, esta estaba dada por

```
#include(\s?)<((([a-z])|([A-Z])|(\d))+).h>
```

donde s representa a un espacio y $d = [0-9]$. En la práctica actual se tiene que implementar un autómata (a escoger) que acepte esta expresión regular, aquí se escogió un AFD con transiciones epsilon, directamente de la expresión regular se hizo un AFN- ϵ por Thompson.

3. Diseño de la solución

Para implementar el autómata crearemos dos clases `State` y `NonDeterministicAutomata`, estarán descritas a continuación.

Las variables del objeto `State`

- `Int id`, Un identificador del estado.
- `HashMap<String, HashSet<Int>delta`, Una tabla hash, que representara la funcion de transicion δ .

Las variables del objeto `NonDeterministicAutomata` mas representativas son

- `HashSet<Int>initial`, Conjunto que indicará conjunto de estados iniciales.
- `HashSet<int>finalStates`, Conjunto que almacenará los estados finales.
- `HashMap<Int, State>states`, Tabla que asocia un `id` con su respectivo objeto `State`
- `HashMap<Int, ArrayList<State epsilonClosure`, Tabla que asocia a un identificador de un estado con los estados alcanzables mediante transiciones vacias.

3.1. Codigo

`Hi.txt`

```

1 fun initAutomata(): NonDeterministicAutomata {
2     val auto = NonDeterministicAutomata(hashSetOf(0), hashSetOf(28))
3     auto.add(0, hashMapOf("#" to hashSetOf(1)))
4     auto.add(1, hashMapOf("i" to hashSetOf(2)))
5     auto.add(2, hashMapOf("n" to hashSetOf(3)))
6     auto.add(3, hashMapOf("c" to hashSetOf(4)))
7     auto.add(4, hashMapOf("l" to hashSetOf(5)))
8     auto.add(5, hashMapOf("u" to hashSetOf(6)))
9     auto.add(6, hashMapOf("d" to hashSetOf(7)))
10    auto.add(7, hashMapOf("e" to hashSetOf(8)))
11    auto.add(8, hashMapOf("^$" to hashSetOf(9, 11))) // ^$ is a regex for an empty
    ↳ string
12    auto.add(9, hashMapOf(" " to hashSetOf(10)))
13    auto.add(10, hashMapOf("^$" to hashSetOf(11)))
14    auto.add(11, hashMapOf("<" to hashSetOf(12)))
15    auto.add(12, hashMapOf("^$" to hashSetOf(13, 17, 21))) // empty range
16    auto.add(13, hashMapOf("^$" to hashSetOf(14)))
17    auto.add(14, hashMapOf("[a-z]" to hashSetOf(15)))
18    auto.add(15, hashMapOf("^$" to hashSetOf(16)))
19    auto.add(16, hashMapOf("^$" to hashSetOf(25)))
20    auto.add(17, hashMapOf("^$" to hashSetOf(18)))
21    auto.add(18, hashMapOf("[A-Z]" to hashSetOf(19)))
22    auto.add(19, hashMapOf("^$" to hashSetOf(20)))
23    auto.add(20, hashMapOf("^$" to hashSetOf(25)))
24    auto.add(21, hashMapOf("^$" to hashSetOf(22)))
25    auto.add(22, hashMapOf("[0-9]" to hashSetOf(23)))

```

```

26     auto.add(23, hashMapOf("^$" to hashSetOf(24)))
27     auto.add(24, hashMapOf("^$" to hashSetOf(25)))
28     auto.add(25, hashMapOf("^$" to hashSetOf(12),
29                          "." to hashSetOf(26)))
30     auto.add(26, hashMapOf("h" to hashSetOf(27)))
31     auto.add(27, hashMapOf(">" to hashSetOf(28)))
32     auto.add(28, hashMapOf(" " to hashSetOf())) // empty range
33     return auto
34 }
35
36 fun main(args: Array<String>) {
37     val auto = initAutomata()
38     println("Introduzca las palabras a validar")
39     while (true) {
40         val str = readLine()
41         if (auto.match(str!!)) println("La palabra $str es aceptada")
42         else println("La palabra $str no es aceptada")
43     }
44 }

```

State.kt

```

1  // object state used in automaton
2  class State (val id: Int,
3              var _delta: HashMap<String, HashSet<Int>> = HashMap()) {
4
5      override fun toString(): String { // method called when trying to print this
6          ↪ object
7          var str = String()
8          for ((key, value) in _delta) str += " Id: $id Regex: $key Codomain: $value "
9          return str
10     }
11
12     fun isDefined(input: String): Boolean { // exists a value for delta distinct than
13         ↪ null ?
14         return delta(input) != null
15     }
16
17     // yeah, i know there exist getters n setters for kotlin, but they aren't
18     // useful for me this time...
19     fun add(regex: String, codomain: HashSet<Int>) {
20         _delta[regex] = codomain
21     }
22
23     fun numberOfFunctions(): Int { // number of domains defined for this state
24         return _delta.size
25     }
26 }

```

```

25     fun delta(char: String): HashSet<Int>? { // we return null if there's no value for
        ↪ input
26         for (key in _delta.keys)
27             if (char.matches(key))
28                 return _delta[key]
29         return null
30     }
31
32     fun String.matches(regex: String): Boolean { // extension function, same behavior a
        ↪ matches method in java for String
33         val regularExpr = Regex(regex)
34         return regularExpr.matches(this)
35     }
36
37 }

```

NonDeterministicAutomata.kt

```

1  /* Formally an automata A is defined as (Q, Sigma, delta, q0, F)
2     In implementation we're only interested q0 and F in the constructor.
3     Delta it'll be defined by a method.
4  */
5
6  class NonDeterministicAutomata(val initial: HashSet<Int>, val finalStates:
    ↪ HashSet<Int>) {
7
8      private var currentStates = ArrayList<State>() // used in match function
9      private var epsilonClosure = HashMap<Int, ArrayList<State>>()
10     private var statesWithSkippedEpsilon = HashMap<Int, List<State>>>() // we skip most
        ↪ epsilon transition
11     private var initializeOnce = false // used for initialize the last two variables
        ↪ once
12     val states = HashMap<Int, State>()
13
14     fun <T> ArrayList<T>.replace (old: T, newValues: Collection<T>) { // not used
        ↪ function
15         val i = this.indexOf(old)
16         this.remove(old)
17         this.addAll(i, newValues)
18     }
19
20     override fun toString(): String {
21         var str = String()
22         for (value in states.values) str += "${value}"
23         return str
24     }
25
26     fun add(index: Int, values: HashMap<String, HashSet<Int>>>) {
27         states[index] = State(index, values)

```

```

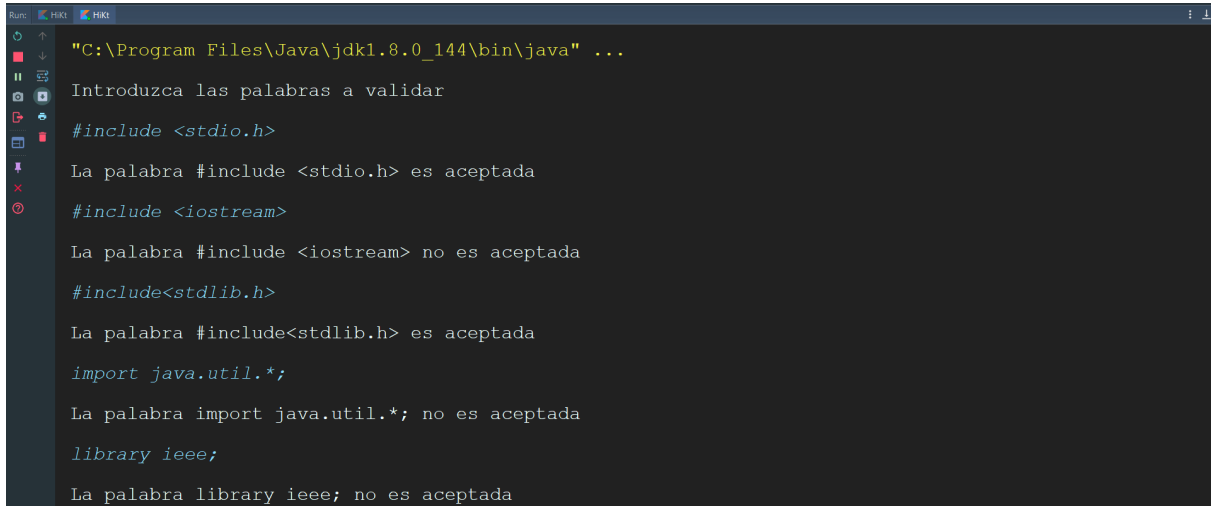
28     }
29
30     private fun setCurrentStates(ints: HashSet<Int>) {
31         states.forEach({ (id, state) -> if (id in ints) currentStates.add(state) })
32     }
33
34     fun match(word: String): Boolean {
35         setCurrentStates(initial)
36         if (!initializeOnce) { initEpsilonClosure(); initStateWithSkippedEpsilon();
37             ↪ initializeOnce = true }
38         word.forEach { char -> // it's actually a Char, not String type
39             var statesBuff = ArrayList<State>()
40             currentStates.forEach {
41                 ↪ statesBuff.addAll(statesWithSkippedEpsilon[it.id]!!.asIterable()) }
42             currentStates = ArrayList(statesBuff.distinct())
43             statesBuff.clear()
44             currentStates.forEach {
45                 val others = it.delta(char.toString())
46                 if (others != null) others.forEach { id -> statesBuff.add(states[id]!!)
47                     ↪ }
48             }
49             if (statesBuff.isEmpty()) return false // we can't keep going with word
50                 ↪ given
51             currentStates = ArrayList(statesBuff.distinct())
52         }
53         return currentStates.any { it.id in finalStates }
54     }
55
56     // helper function for initEpsilonClosure
57     private fun createClosure(id: Int, buffer: ArrayList<State>) {
58         buffer.add(states[id]!!)
59         if (states[id]!!.isDefined("")) {
60             states[id]!!.delta("")!!.forEach { createClosure(it, buffer) }
61         }
62     }
63
64     private fun initEpsilonClosure() {
65         states.keys.forEach {
66             var tmp = ArrayList<State>()
67             createClosure(it, tmp)
68             epsilonClosure[it] = tmp
69         }
70     }
71
72     private fun initStateWithSkippedEpsilon() { // initEpsilon should have been
73         ↪ executed before
74         epsilonClosure.forEach { (id, list) ->
75             statesWithSkippedEpsilon[id] = list.filter { !("^$" in it._delta.keys) ||
76                 ↪ (it.numberOfFunctions() > 1) }
77         }

```



```
72     }  
73  
74 }
```

4. Funcionamiento



```
"C:\Program Files\Java\jdk1.8.0_144\bin\java" ...  
Introduzca las palabras a validar  
#include <stdio.h>  
La palabra #include <stdio.h> es aceptada  
#include <iostream>  
La palabra #include <iostream> no es aceptada  
#include<stdlib.h>  
La palabra #include<stdlib.h> es aceptada  
import java.util.*;  
La palabra import java.util.*; no es aceptada  
library ieee;  
La palabra library ieee; no es aceptada
```

5. Conclusiones

Con esta práctica comprendí mejor la relación que existe entre una expresión regular y su autómata finito determinista equivalente. Con los AFD, es posible codificar desde cero un programa que verifique si una cadena pertenece al lenguaje generado por la expresión regular sin usar bibliotecas externas, usando estructuras de datos muy sencillas como arreglos, cadenas, conjuntos y tablas hash. Sin embargo, el AFD puede crecer demasiado con expresiones regulares que no son tan complejas, y representarlo gráficamente resulta complicado. También refuerza un poco los conceptos de Tail Recursion para obtener las clausuras epsilon.