PROCESADORES DEL LENGUAJE

PRACTICA 1

* Equivalencias de ER a JFLAPS

|  |  |
| --- | --- |
| ER | JFLAPS |
| () | () (no hay cambios) |
| \* (Cierre de Kleene) | \* (no hay cambios) |
| + | (no existe equivalente directo) |
| | (xor) | + |
| [] | (no existe equivalente directo) |
| ? | (no existe equivalente directo) |
| (no existe equivalente directo) | ! (elemento vacío épsilon) |

|  |  |
| --- | --- |
| ER | JFLAP |
| a|b|c | a+b+c |
| abc | abc |
| a?bc | (!+a)bc |
| a+bc | a(!+a)\*bc |
| (abc)\* | (abc)\* |
| [abcd] | (a+b+c+d) |
| [a-z] | (a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z) |

* ER a tratar:

a?bc(de)\*

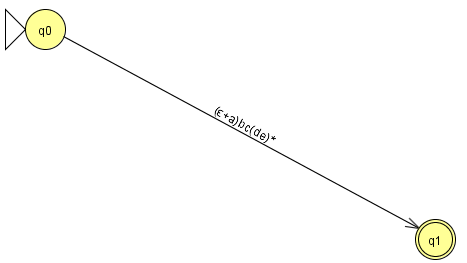
* Transformación a ER JFLAPS:

(!+a)bc(de)\*

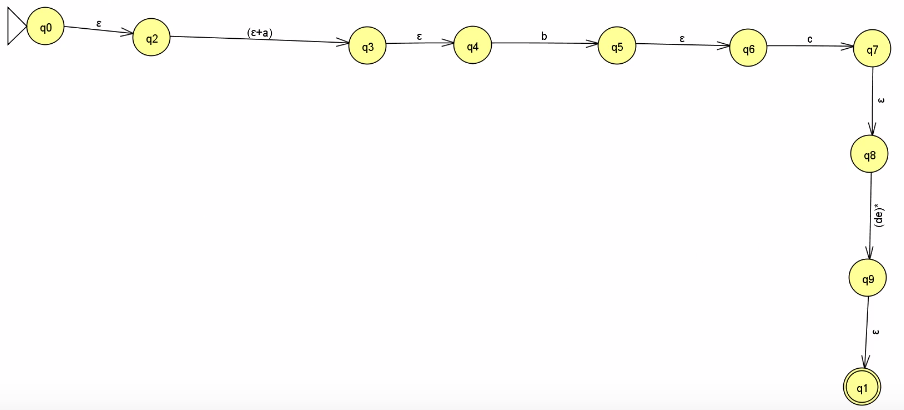
* Lexemas candidatos para probar el patrón:
* abcdedede
* bc
* bcdedebc

Con la herramienta JFLAP descargada y una vez puestas las preferencias (épsilon como elemento vacío) vamos al apartado “Regular Expression” donde añadiremos nuestra expresión regular y después pulsaremos sobre “Convert to NFA” para convertir nuestra ER en un autómata finito no determinista (NFA, Non-Deterministic Finite Automata).

Un Autómata Finito No Determinista (AFND) es una construcción matemática, que es un grafo con arcos dirigidos, me permite moverme entre estados. Una expresión regular lo programaría con un montón de if, los autómatas sirven para ahorrar líneas de código y que usando la herramienta matemática AFND nos solucionará este problema de hacer tantos ifs.

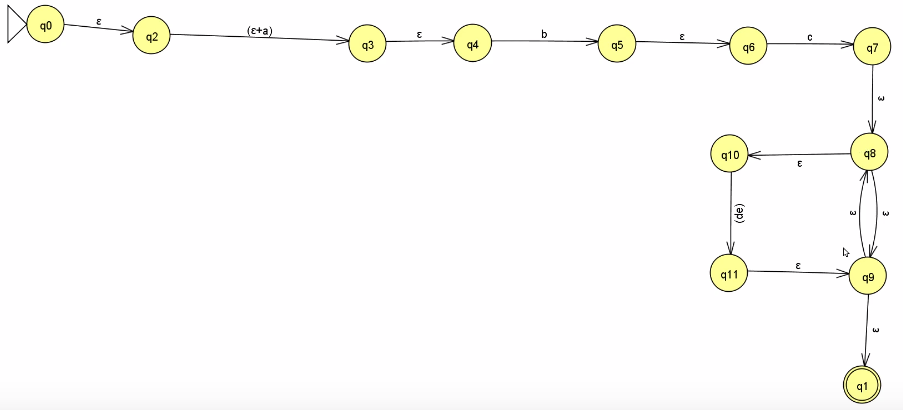


Una vez convertida la ER a un AFND, vemos que el autómata consta de dos estados de los cuales uno es el estado inicial Q0 y el estado final Q1. Esta es la versión comprimida del autómata, pero, hay que seguir desarrollándolo en sus pasos más básicos que es ir analizando las operaciones y añadiendo nuevos estados que me permitan ir paso a paso comprobando la ER que hemos añadido y que queremos analizar. Pulsamos en “Do Step”.



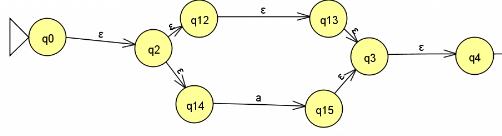
Ahora tengo cosas separadas entre estados, pasando de un estado a otro con épsilon. La razón para que el sistema haga esto es una razón organizativa, hace esto para separar los distintos trozos de la expresión

Seguimos desarrollando la ER:

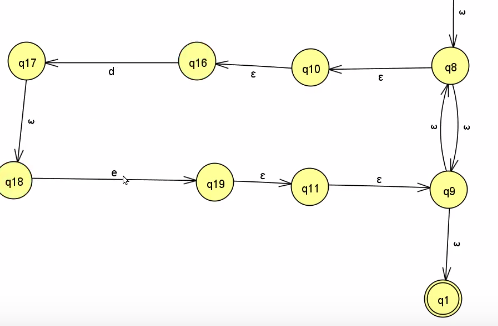


Seguimos desarrollando:

Al estar eligiendo entre una cosa y otra (entre épsilon o a). El sistema poco a poco nos va dando las distintas opciones sobre qué hacer.



En “de” no hay que elegir entre d ó e. Tenemos en cuenta que en un analizador se lee la entrada carácter a carácter y no de dos en dos, por tanto, hay que separar la entrada. Este será el siguiente paso, seguimos desarrollando:

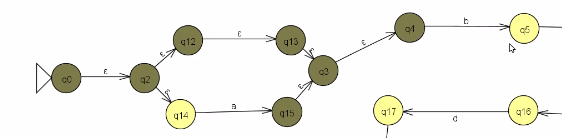


Como hemos dicho antes, los épsilon nos lo pone para separar los caracteres. Ya hemos terminado el desarrollo y tenemos el AFND desarrollado.

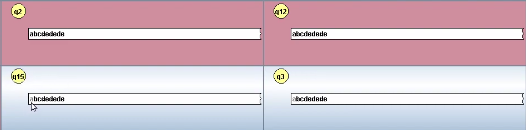
* Le damos a “Export” y nos aparece una nueva ventana con el mismo AFND desarrollado por tanto ahora probaremos la ER con algunos lexemas de prueba para ello pulsamos en “Input” y “Step by State”.

Le damos abcdedede para que compruebe. El analizador dice “Yo podría arrancar el proceso de análisis de esta cadena en cualquiera de estos estados (en los sombreados)”

Una vez que le damos a “Step” para comprobar se iluminan en rojo algunos estados que quiere decir “Si yo le hubiese pasado esa cadena de entrada y hubiese intentado procesar el primer carácter, el a, en estos estados no habría podido.







Estando en Q15, Q4 y Q3 he sido capaz de procesar la a, por ello no sale en rojo. Así con todas las letras.

Una vez terminado, sale el estado Q1 en verde, que querrá decir que ya hemos terminado correctamente. Si pinchamos en el cuadrito en verde de Q1 y pinchamos en “Traceback” sale una pestañita con la información resumida. Q14 🡪 Q4 🡪 Q6 🡪 Q16 🡪 Q18 🡪 Q 16 🡪 Q18 🡪 Q16 🡪 Q17

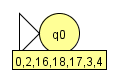
* ¿Qué pasaría si yo intentase hacer todas las opciones posibles?

“Input” 🡪 “Step by state”

El AFND tarda mucho porque es un “no determinista” lo que quiere decir que puede tomar múltiples opciones

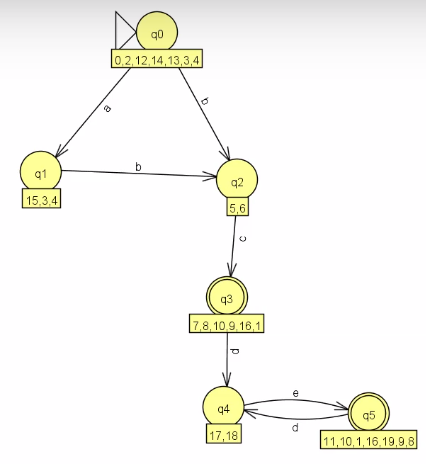
* Convirtamos el AFND a un AFD para que todo esté más simplificado

Convert NFA to DFA.



Aparece Q0 y unos números debajo. Q0 es un estado del AFD y los números son los estados que son equivalentes a Q0, los estados a partir de Q0 con los que puedo llegar con épsilon, eso se denomina **cierre**.

Pulsamos en “Complete”



Este es el grafo del autómata finito determinista. Es más sencillo y tiene menos estados, estados iniciales solo hay uno obligatoriamente y agrupa a varios estados que tenía en el AFND.

Hay varios estados finales porque puedo acabar en el estado Q3 ya que el de no es necesario, es válido un lexema sin “de”.

En un AFD no hay épsilon y todas las transiciones son más directas.

* Pulsamos en “Complete”

Se abre una nueva ventana

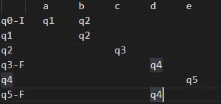
“Input” 🡪 “Step by State”

Nos damos cuenta de que es mucho más fácil y rápido porque no hay tantos caminos.

* Finalmente quiere comprobar la ER en un programa Java

Una manera muy sencilla de transformar un AFD en algo fácil de gestionar es con una matriz. Una máquina de estados que se basa en una matriz.

Como columnas tenemos todos los caracteres validos del alfabeto sin épsilon (ya que es un autómata determinista) y en las filas tenemos los estados del AFD marcando cual es el estado final y cuáles son los finales.



LS02 Implementación de Autómatas Finitos Deterministas y Máquinas de Estado

Una estructura que tendrá: máquina de estados y la funcionalidad que queremos tener (por ejemplo, función 1 que comprueba si una cadena es válida o función 2 que calcula cadenas nuevas para una ER dada).

Una máquina de estados tendrá que tener algún tipo de interfaz que le permita conectarse de alguna manera con la funcionalidad del programa que queramos hacer. Es posible que tengamos que hacer varias interfaces depende de la funcionalidad. En la máquina de estados tendremos:

* Un autómata finito determinista que tendrá los n elementos del alfabeto, una lista que contiene los distintos estados, aparte tendremos cual es el estado inicial y cuales son los estados finales. Como bloque central tendremos una matriz de cambio de estados que estará dividida en tantas columnas como elementos tenga el alfabeto y tantas filas como estados tenga. Con esto es con lo que tiene que trabajar la máquina de estados.
* Lógica de control que hará será hablar con el autómata para realizar las distintas operaciones y esa lógica de control al mismo tiempo tendrá una zona de memoria donde se guarde el estado actual de la máquina.

Entonces, ¿qué diferencia hay entre la máquina de estados y el autómata finito determinista? El autómata finito determinista es el mapa de como navego yo dentro de esa lógica y la máquina de estados es en el estado actual en el que estoy trabajando para ese autómata y por donde he ido pasando antes si es que lo necesitase.

**Autómata Finito Determinista**

Como estructuras de datos va a tener en nuestro caso estados y alfabeto, que eso pueden ser listas de elementos.

El alfabeto debe corresponderse con una lista de elementos de tipo *char List <character>*.

Al igual que el alfabeto consiste en una lista de caracteres, lo normal es que los estados sean una lista de estados que llamaremos como Q0, Q1, Q2, etc. Por tanto, los estados se pueden corresponder como enteros *List <Integer>*.

Aparte de esto, tendremos un estado inicial de tipo *Integer* y estados finales que como pueden ser varios será una lista de enteros *List <Integer>*.

Estableceremos nuestra matriz como un HashMap clave valor tal que:

*HashMap <Integer, HashMap <Character, Integer>>*

Estas estructuras de datos a su vez necesitarán tienen que tener algún tipo de contrapartida, es decir, vamos a necesitar funciones para poder operar con el autómata. Nos podemos encontrar con distintas operaciones como, por ejemplo:

* cargarAlfabeto() introducimos los datos del alfabeto a nuestra variable List <Character> haciendo Alfabeto.add(new Character(‘a’)) n veces, tantas como necesite.
* cargarEstados() introducimos los estados . Cada vez que quiera meter un autómata nuevo a mi máquina de estados probablemente tendré que cambiar este método.
* establecerQi() nos pasan por parámetro el estado inicial y hago *EstadoInicial=p*
* establecerQf() tenemos que añadir todos los estados finales, *EstadosFinales.add (new Integer(7))* pondré tantos como estados finales tenga. Si cambio de autómata finito tendré que cambiar esta función.
* Para inicializar la matriz necesitamos dos fases:

1. Inicializamos el primer HashMap que tenga tantos estados como habíamos inicializado en la función *cargarEstados()*
2. Inicializamos el resto de HashMap

* Void inicializaMatriz() introducimos en la matriz cada estado existente con un nuevo HashMap <Character, Integer> ())
* Para recibir los datos y completar la segunda tabla de mi matriz tendré una función que será *void cargarMatriz()* que hará *Matriz.get(Estado).add(caracter, estado\_de\_salto); tendré tantas como necesite.* Por ejemplo: *Matriz.get(0).add(‘a’,3);* que significa que si estoy en el estado 0 y me entra una ‘a’ que me vaya al estado 3.

En esta función es donde realmente cargamos la información de la matriz.

* Integer getSiguienteEstado (estado, caracter) retornará de la matriz el siguiente estado al que iría desde un estado y un caracter dados. Si me devuelve un valor nulo significa que la transición que estoy haciendo no existe. Está pensado para que sea la máquina de estado la que lo llama y forma parte de la interfaz de control donde estarán hablando la máquina de estados y el autómata.

Es posible que necesite funcionalidades extras como por ejemplo una función extra booleana que me devuelva true si un estado es final o false si no lo es.

También una función que devuelva cual es el estado inicial, normalmente es el 0 pero puede ser otro.

**Máquina de estados**

Lo que la máquina de estados necesita es almacenar cual es el estado inicial y una lógica de control. Esa lógica de control será poder coger los datos de la máquina, etc.

Estructuras de datos:

EstadoActual que será un Integer

Tendremos los métodos:

* Inicializar() donde empezaremos a trabajar cogiendo el estado actual a partir del autómata finito determinista AFD.getEstadoInicial(); El AFD es una clase porque tiene entidad propia.
* Acepta(caracter)
* Bool isFinal()
* bool compruebaCadena (cadena)

El constructor de nuestra máquina de estados recibirá por el constructor al AFD para poder trabajar con él.