Práctica 3

IPN Escuela Superior de Cómputo
Ing. En Sis. Computacionales

Compiladores 3CV5
M.C Saucedo Delgado Rafael Norman

2014090186
Elizalde Díaz Roberto Carlos

Introducción

En este documento se encontrará la descripción para el desarrollo de la práctica número 3 que consta de programar el algoritmo 3.20 del libro de Compiladores, principios, técnicas y herramientas del autor Alfred V. Aho.

Este algoritmo es utilizado para realizar la construcción de subconjuntos que pertenezcan a un autómata finito determinista, a partir de un autómata finito determinista.

Para la realización de esta práctica se utilizó el lenguaje Java en su versión libre (openjdk) 11.0.9.1 y Netbeans 11 como IDE, corriendo en el sistema operativo Ubuntu 20.04.1 LTS.

Desarrollo

Para el desarrollo de esta práctica no se utilizó metodología específica, sólo fue un análisis corto de la implementación del algoritmo ya descrito en el libro.

Esta práctica implementó clases de las prácticas pasadas tales como:

- Automata: Clase en donde se describe la estructura y funcionamiento de un autómata estándar.
- AFN y AFD: Subclases de la clase Automata
- Transicion: Clase que describe a una transición, esta incluye un estado de inicio, final y en símbolo por el cual un estado pasa al otro.

La clase específica para esta práctica es la clase AFNConvertidor y funciones principales son:

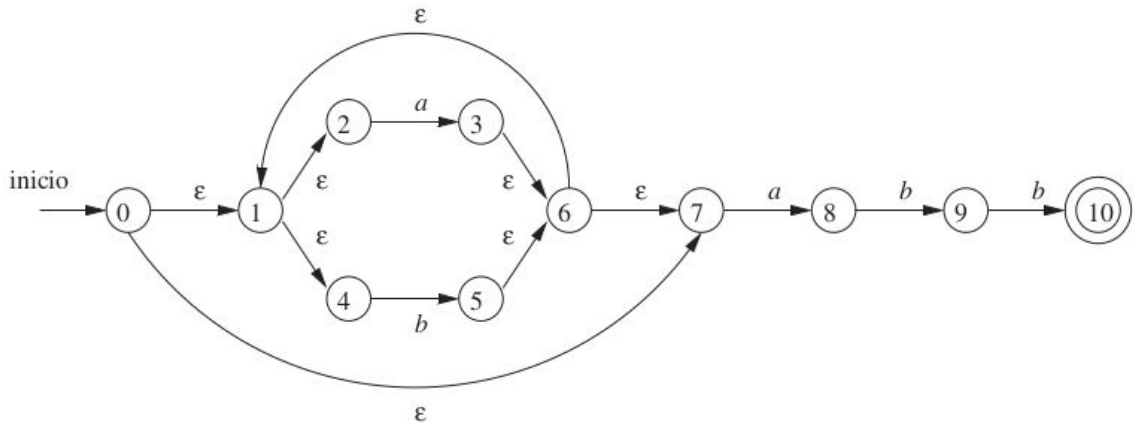
- ***private List<Integer> eCerraduraT(List<Integer> T):*** Método que calcula la operación e-cerradura(T), donde T es una lista de enteros para representar el conjunto de estados T.
- ***private List<Integer> eCerraduraS(int s, List<Integer> estadosCerraduraS):*** Método que calcula la operación e-Cerradura(s) donde s representa el estado del que se le quiere aplicar la operación cerradura y estadosCerraduraS es una lista de estados resultantes al aplicar la operación.
- ***private List<Integer> moverTa(List<Integer> T, char a):*** Método que calcula la operación mover(T,a) donde T es una lista de los estados a los que se requiere comprobar si es posible pasar a otro estado mediante el símbolo 'a'.
- private void construccionSubconjunto(): Este es el método que engloba todo el algoritmo de construcción de subconjuntos.

Otro métodos utilizados para complementar los métodos anteriores son:

- private List<Integer> eliminarRepeticiones(List<Integer> estados)
- private boolean hayEstadoSinMarcar()
- private List<Character> obtenerSimbolosEntrada()
- private List<Character> eliminarRepeticionesSimbolos(List<Character> simbolos)
- private boolean estaEnLista(List<Character> ls, char c)
- private boolean estaEnDestados(List<Integer> U)
- private void dtrans(List<Integer> T, char a, List<Integer> U)
- public void printAFD()

Pruebas

Para la realización de las pruebas se implementó el AFN que se usa como ejemplo ilustrativo en de la página 155 del libro ya mencionado.



Para implementar un AFN se hace una instancia de la clase con el mismo nombre, se agregan las transiciones y se establece estado final e inicial de la siguiente manera:

```
AFN temp = new AFN();
temp.agregar_transicion(0, 1, 'E');
temp.agregar_transicion(1, 2, 'E');
temp.agregar_transicion(1, 4, 'E');
temp.agregar_transicion(2, 3, 'a');
temp.agregar_transicion(4, 5, 'b');
temp.agregar_transicion(3, 6, 'E');
temp.agregar_transicion(5, 6, 'E');
temp.agregar_transicion(6, 7, 'E');
temp.agregar_transicion(7, 8, 'a');
temp.agregar_transicion(8, 9, 'b');
temp.agregar_transicion(9, 10, 'b');
temp.agregar_transicion(6, 1, 'E');
temp.agregar_transicion(0, 7, 'E');

temp.establecer_inicial(0);
temp.establecer_final(10, true);
```

Posteriormente al constructor se le pasa como argumento el objeto AFN llamado temp al instanciar un objeto de la clase AFNConvertidor:

```
AFNConvertidor ctemp = new AFNConvertidor(temp);
```

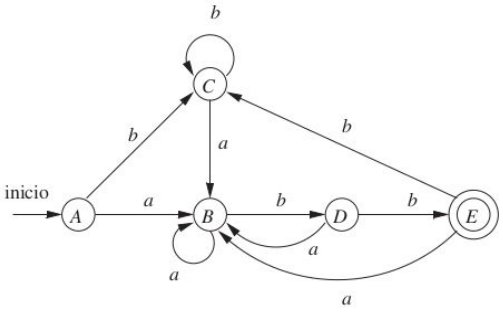
Para poder obtener el AFD equivalente se manda a llamar al método obtenerAFD() del objeto ctemp:

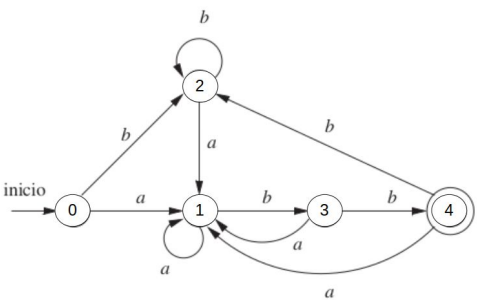
AFD afnTemp = ctemp.obtenerAFD();

Este método retornará un objeto AFD que será el equivalente al AFN temp. Finalmente para corroborar que se obtuvo en AFD correctamente podemos llamar la método printAFD() de la clase AFNConvertidor imprimiendo así en consola la siguiente salida:

```
0->1,a
0->2,b
2->1,a
2->2,b
1->1,a
1->3,b
3->1,a
3->4,b
4->1,a
4->2,b
0: [0, 1, 2, 4, 7]
1: [3, 6, 7, 1, 2, 4, 8]
2: [5, 6, 7, 1, 2, 4]
3: [5, 6, 7, 1, 2, 4, 9]
4: [5, 6, 7, 1, 2, 4, 10]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Vemos que las primeras líneas representan las transiciones que el AFD contiene y las últimas representan los elementos de los conjuntos asociados a un estado. Para comprobar el autómata podemos seguir dibujar cada una de las transiciones y comparar:





Conclusiones

Esta fue sin lugar a duda la práctica más sencillas de todas las que he realizado en esta materia, se nota claramente la diferencia de velocidad entre programar un algoritmos ya diseñado que diseñarlo uno mismo, y a la vez la que más entusiasmo para realizar pues el hecho de incluir el tema de conjuntos se me hizo interesante, además recuerdo que en la asignatura de Teoría Computacional tuve muchos problemas con las AFN pues la manera de plantearlos para su programación era sumamente compleja, bueno creo que es imposible si es que no se hace un procedimiento como este para llevarlo a un AFD, algo que también me causó confusión en Teoría Computacional.

Referencias

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compiladores: principios, técnicas y herramientas, Segunda Edición.
- Documentación ArrayList Java 8 https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
- Videos de clase