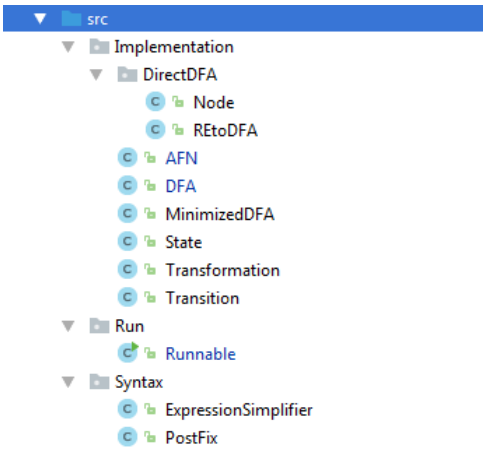
Diseño de Lenguajes de Programación
Proyecto No. 1

Gabriel Brolo Tobar, Carnet 15105

Diseño de la aplicación

La aplicación se encuentra dividida en tres directorios. El directorio 'Run' contiene la clase principal, 'Runnable' que se encarga de manejar las entradas del usuario y crear las llamadas a los autómatas. El directorio 'Syntax' contiene dos clases: 'PostFix' y 'ExpressionSimplifier'. La primera se encarga de convertir una expresión regular en formato infix a postfix, la segunda se encarga de realizar algunas abreviaturas. Dentro del directorio 'Implementation', se encuentran los archivos que realizan la mayor parte del trabajo.



Para construir un autómata, se decidió crear dos estructuras que se pudieran conectar entre sí, sin la necesidad de crear una estructura de Grafo. La primera de ellas es **State**. Esta representa un estado o nodo del autómata, por lo que tiene asociados un conjunto de estados previos y siguientes, además de tener un ID único y las propiedades de ser o no estado de aceptación.

```
public class State {
    private List<State> previousStates;
    private List<State> nextStates;
    private int stateId;

    private boolean isInitial;
    private boolean isFinal;
```

Para poder relacionar un estado con otro, se creó una estructura **Transition**, la cual tiene asociada un estado inicial, un estado final y un símbolo.

```
public class Transition {
    private State initialState;
    private State finalState;
    private String transitionSymbol;

    public Transition(String transitionSymbol) {
        this.transitionSymbol = transitionSymbol;
        this.initialState = new State(AFN.stateCount);
        this.finalState = new State(AFN.stateCount);

        /* add link to states */
        this.initialState.addNextState(this.finalState);
        this.finalState.addPreviousState(this.initialState);
    }
```

Con estas estructuras y las clases del directorio 'Syntax' ya es posible pensar en la construcción de un autómata. La clase **AFN** describe un autómata finito no determinista. En resumen, es un conjunto de listas de símbolos, listas de transiciones, estados finales, estados iniciales y estados en sí. Para crear un **AFN** solamente es necesaria una expresión regular, luego el objeto AFN se encarga de crear un objeto PostFix que traduce la expresión y la simplifica vía ExpressionSimplifier. Posteriormente se calcula la lista de símbolos, se realiza la construcción del NFA, se calcula la lista de estados y los estados finales e iniciales.

Es el método regExpToAFN el que se encarga de construir el NFA, utilizando un stack para ir guardando los estados iniciales y otro stack para guardar los estados finales. Además, los métodos unify, concatenate y kleene se encargan de crear las transiciones correspondientes según el operador actual en la expresión regular.

Ya con esta clase, es posible hablar sobre la transformación de NFA a **DFA**. La clase **Transformation** se encarga de esto. El resultado es un objeto **DFA**. Se calculan primero las cerraduras de los estados y se generan tablas de transiciones hechas con Hashmaps para guardar la información sobre qué subconjuntos llevan a otros con símbolos en específico. En total, se calcula una tabla de transiciones, una lista de estados, otra lista de estados con ID's y la lista de símbolos, parámetros que se envían al constructor de **DFA**, con el objetivo de obtener un DFA mediante la construcción de subconjuntos ya realizada en la clase Transformation. La clase DFA ya sólo se encarga de transcribir la información recibida para crear una lista de transiciones y de estados iniciales y finales.

Ya con el DFA, es posible minimizarlo. La clase **MinimizedDFA** se encarga de ello. Recibe un DFA y con él, utiliza el algoritmo de particiones para minimizarlo y generar una nueva tabla de transiciones con los nuevos estados. El método newPartition se encarga de crear particiones y el método minimize contiene la lógica para continuar generando particiones hasta que ya no sea necesario.

Para la generación directa del DFA, hemos de trasladarnos al directorio 'DirectDFA'. La estructura **Node** se emplea para crear los nodos del árbol sintáctico. Es la clase **REtoDFA** la que se encarga de generar el DFA directo. Primero extiende la expresión regular, luego encuentra las operaciones nullable, firstpos y lastpos para cada nodo del árbol sintáctico. Posteriormente calcula el followpos para los catnodes y kleene nodes y con esa información construye el DFA, con la ayuda del método buildDFA.

Discusión y ejemplos: un vistazo sincero del programa

Comenzamos diciendo que el programa no es 100% funcional. No es que todo esté malo, pero, algunas funciones del programa no se llevan a cabo de forma correcta con algunas expresiones regulares. Lo que sí funciona al 100% es la generación del NFA a partir de la expresión regular. Sin embargo, a continuación, explicamos a más detalle qué es lo que ocurre con este programa.

Ingreso de la expresión regular

El programa acepta una cadena que contenga (o no) los operadores esperados. Realiza además algunas simplificaciones, no todas las que existen, pero sí trata de hacer más amigable la expresión para poder reconocerla en formato postfix.

Generación de NFA

Esta generación funciona con el 100% de expresiones regulares que se han probado, i.e. las que se muestran aquí y otras probadas durante la realización del proyecto. Por el momento, no ha dejado de proporcionar un NFA correcto para alguna expresión regular. Al principio estaba mal hecho, pero se logró arreglar el manejo de los stacks para que se generara correctamente el NFA.

Conversión NFA – DFA

Esta conversión funciona para aquellas expresiones regulares en las que el procedimiento de encontrar la cerradura de un estado no produzca loops infinitos. El problema es precisamente ese. Cuando un NFA no tiene estados que generen loops a través de transiciones épsilon, la conversión funciona el 100% de las veces. Sin embargo, cuando el loop es de grado 1, es decir que el loop se mantiene solamente entre dos estados, el programa todavía genera correctamente la conversión. Ahora, cuando existe un loop que sucede a través de muchos estados, culpa de las transiciones épsilon, ahí si no funciona. El error es precisamente un StackOverflow, porque se queda infinitamente tratando de encontrar las cerraduras correspondientes.

Minimización de DFA

La minimización pareciera funcionar con expresiones simples, pero tiende a fallar en muchas ocasiones. También, como se verá en los ejemplos, pareciera que con caracteres extraños deja de funcionar correctamente (caracteres como '<', '%', etc.).

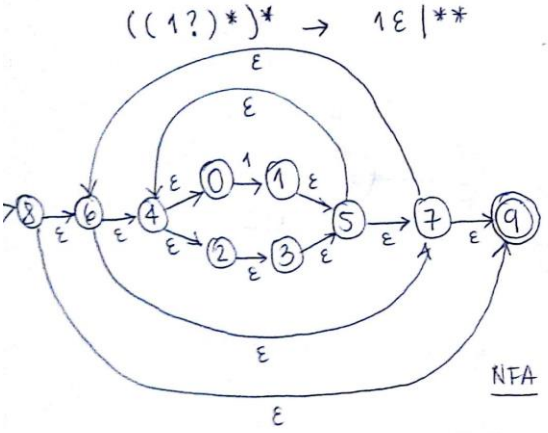
Generación directa.

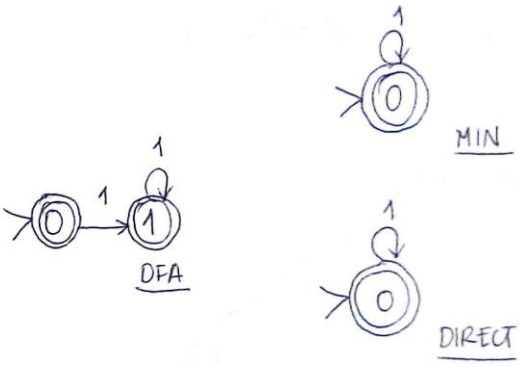
Esta funciona mejor que la minimización, ya que funciona con expresiones que fallan con la minimización. Sin embargo, en ciertas ocasiones falla. Curiosamente, a veces es a la inversa, cuando funciona la minimización, no funciona la generación, pero, es más funcional la generación directa si vemos la cantidad de casos en los que la minimización falla. Yo creería que es el cálculo del followpos el que falla en ocasiones, ya que las sesiones de debugging siempre salían bien para las otras operaciones.

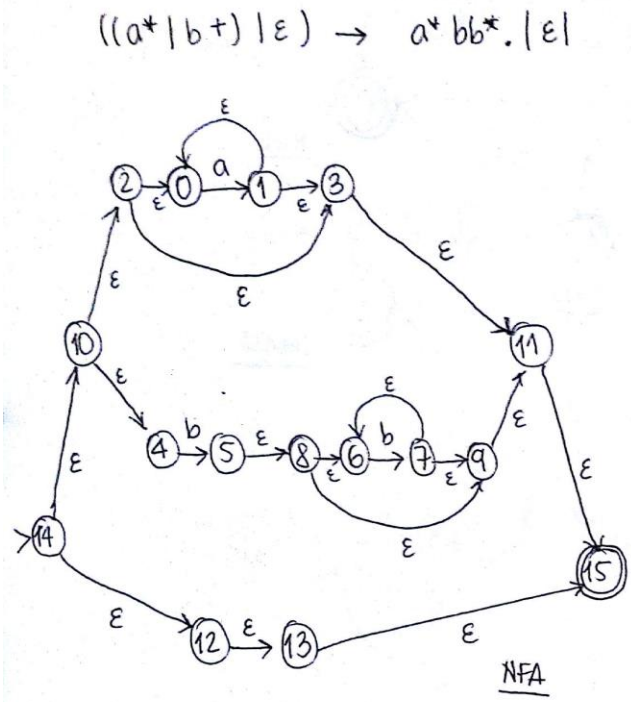
Simulación

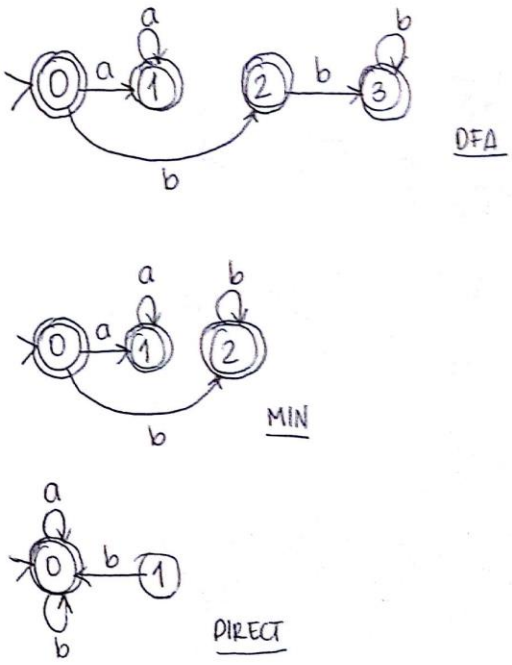
La simulación funciona correctamente, sin embargo, hay un caso curioso para la simulación de NFA. Piense en una cadena que es aceptada en ambos autómatas, luego agréguele un símbolo cualquiera que haga que ya no sea aceptada la cadena. El DFA le dirá que no es aceptada, en efecto, pero el NFA le seguirá diciendo que sí. La razón creo que es porque se calculan las cerraduras de los estados y se guarda una lista de estados y si esa lista tiene el estado de aceptación, se dice que la cadena se acepta, sin embargo, no se agregó una funcionalidad para detectar este tipo de situaciones, por lo que la lista de estados sigue teniendo el estado de aceptación, por lo que el NFA piensa que la cadena se acepta.

Ahora veremos algunos ejemplos. Se muestran algunos autómatas dibujados, con base en la información proveída por el programa. En el caso de los dibujos, los NFA y DFA están siempre correctos, lo que falla a veces son los DFA minimizados o directos.

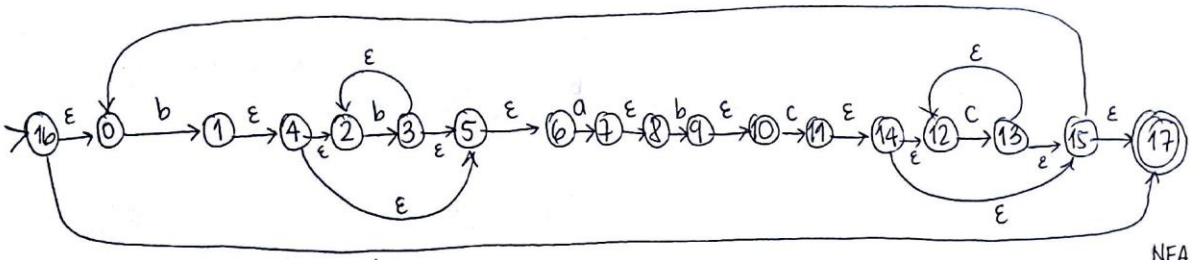




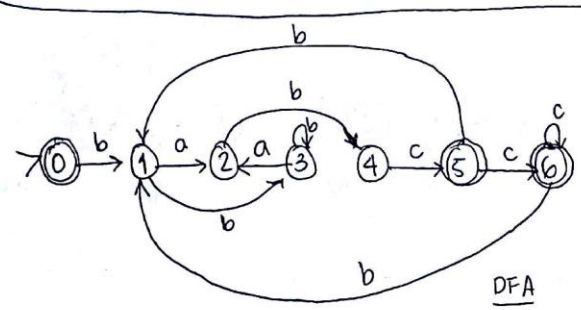




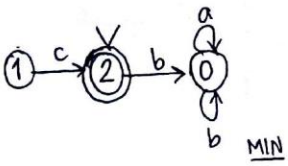
$(b^+abc^+)^* \rightarrow bb^*.a.b.c.c^*.^*$



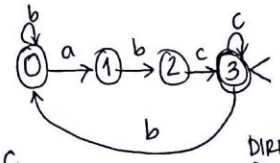
NFA



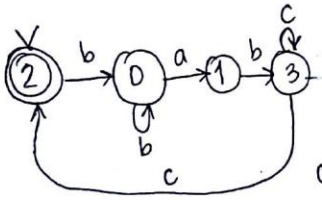
DFA



MIN

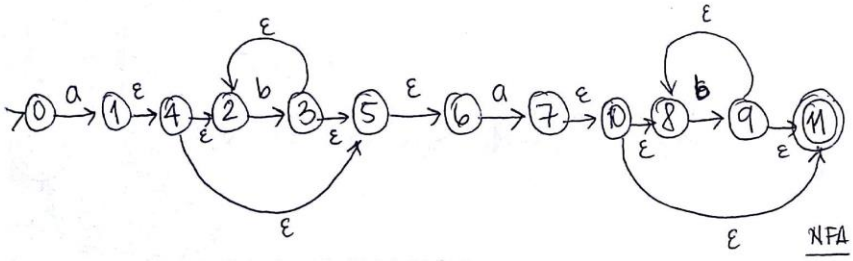


DIRECT DFA

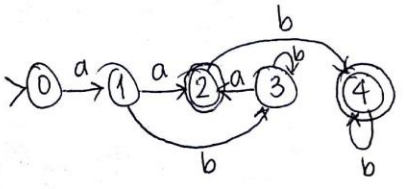


CORRECT

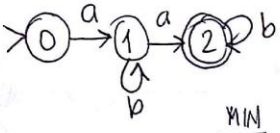
$ab^*ab^* \rightarrow ab^*.a.b^*.$



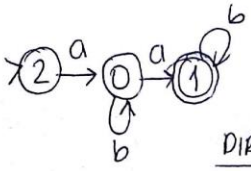
NFA



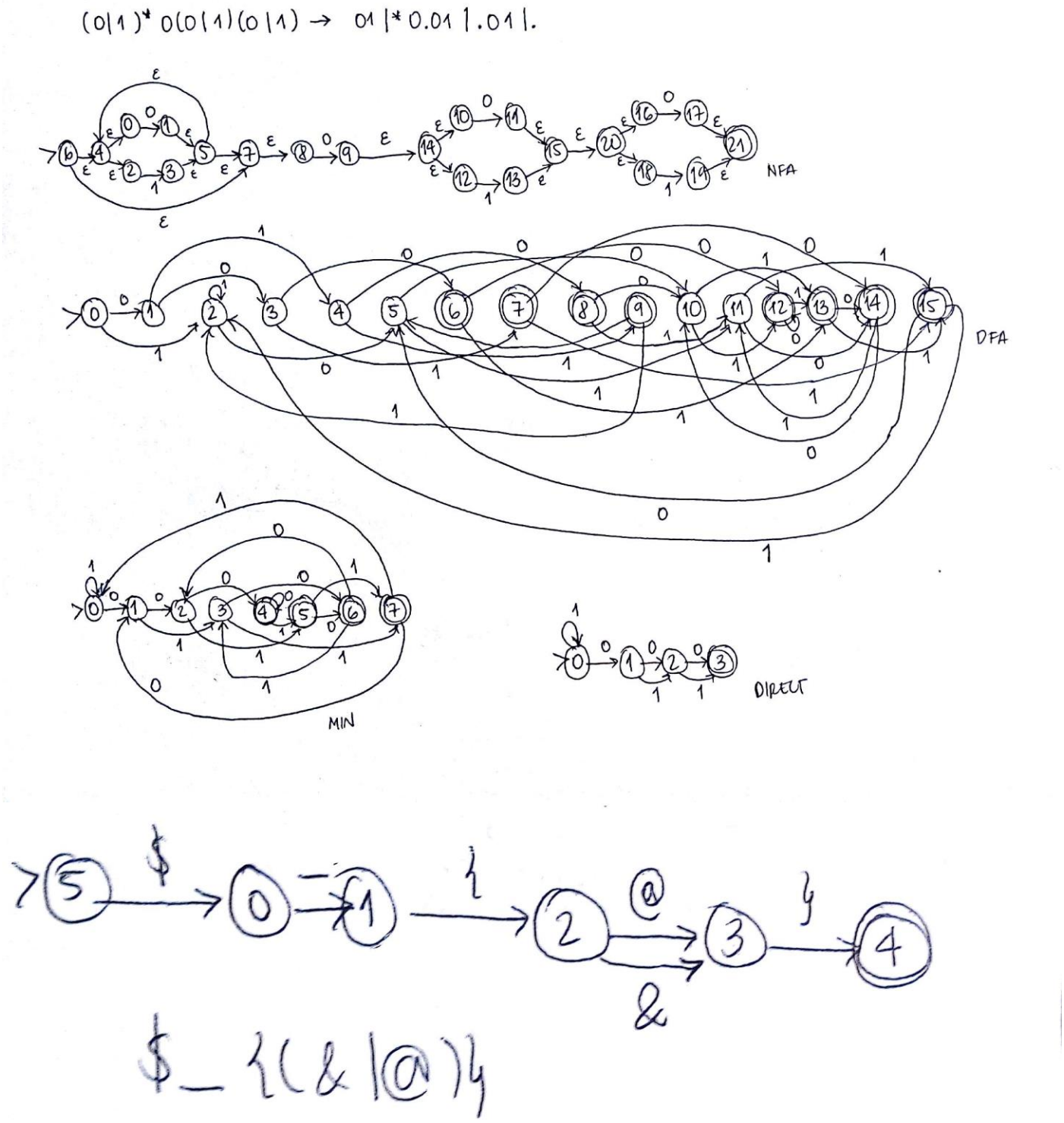
DFA



MIN



DIRECT



Casos de error

Expresiones como `((ε|0)1*)*` proporcionan el error de loop infinito. En el caso de `((a*|b+)|ε)` es la generación directa la que no funciona. Para `(b+abc+)*` no funciona correctamente ni la minimización ni la generación directa. En el caso de `(0|1)*0(0|1)(0|1)` no funciona la directa y en el último caso de la última figura mostrada, no funciona la minimización, debido a los símbolos raros que no se sabe a ciencia cierta el porqué del fallo.

Párrafo de reflexión

Hay muchas expresiones que funcionan bien, claro, la mayoría simples. Con expresiones un cacho complejas, funciona el NFA y DFA, pero las minimizaciones o generaciones directas fallan en ocasiones. Y claro, está el error de closure que hace que no siempre funcione el DFA. En resumen, hay cosas que fallan y cosas que funcionan según sea el caso. Me gustaría decir que siempre funciona, pero creo que son esos pequeños errores que no se corrigieron los que hacen que el programa falle en dichas ocasiones.

Puede visitar el repositorio del proyecto aquí:

`https://github.com/gbrolo/RE-to-NFA-DFA`