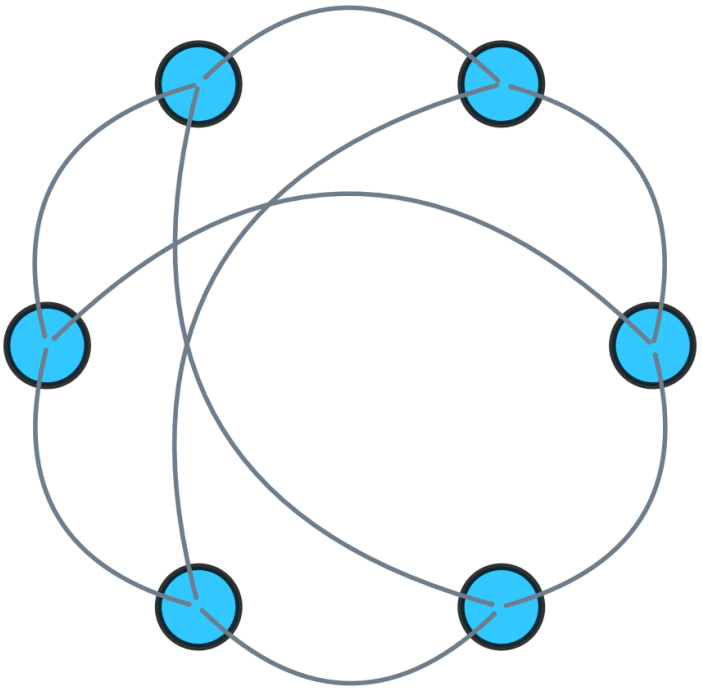
Procesadores de Lenguajes

Curso 2023/2024

Trabajo:

Generador de Autómatas Finitos Deterministas



Autor:

Miguel Quiroga Campos

Índice

Introducción

En el ámbito de la teoría de autómatas y lenguajes formales, los Autómatas Finitos Deterministas (AFD) desempeñan un papel crucial en la implementación de compiladores y analizadores léxicos. Este trabajo tiene como objetivo desarrollar una aplicación que, a partir de una descripción de una expresión regular, sea capaz de calcular el Autómata Finito Determinista asociado utilizando el algoritmo de expresiones regulares punteadas. Como resultado, la aplicación generará un fichero ".java" que implementará el AFD calculado.

El desarrollo de esta aplicación abarca varias etapas fundamentales. Primero, se debe crear un analizador léxico conforme a la especificación léxica proporcionada, utilizando las clases auxiliares aprendidas en la práctica 2 de la asignatura. Posteriormente, se generará una gramática LL(1) y se calcularán sus conjuntos de predicción para desarrollar el correspondiente analizador sintáctico.

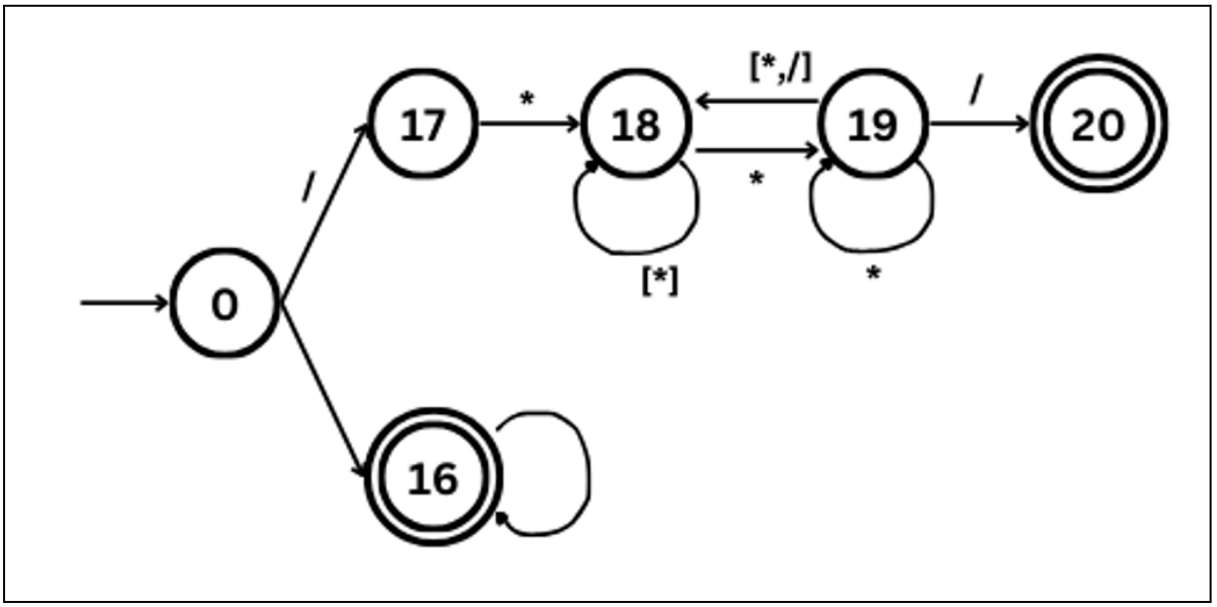
Una vez construido el analizador sintáctico, se definirán las clases necesarias para representar el Árbol de Sintaxis Abstracta (AST). Este analizador será ampliado para incluir un componente semántico que tome como entrada la descripción textual de una expresión regular y produzca una estructura de datos que represente dicha expresión.

El siguiente paso consiste en desarrollar las clases para representar los estados y transiciones de los autómatas basados en expresiones regulares punteadas y programar el algoritmo que calculará el AFD. Finalmente, se implementará un generador de código que creará el fichero ".java" correspondiente al AFD.

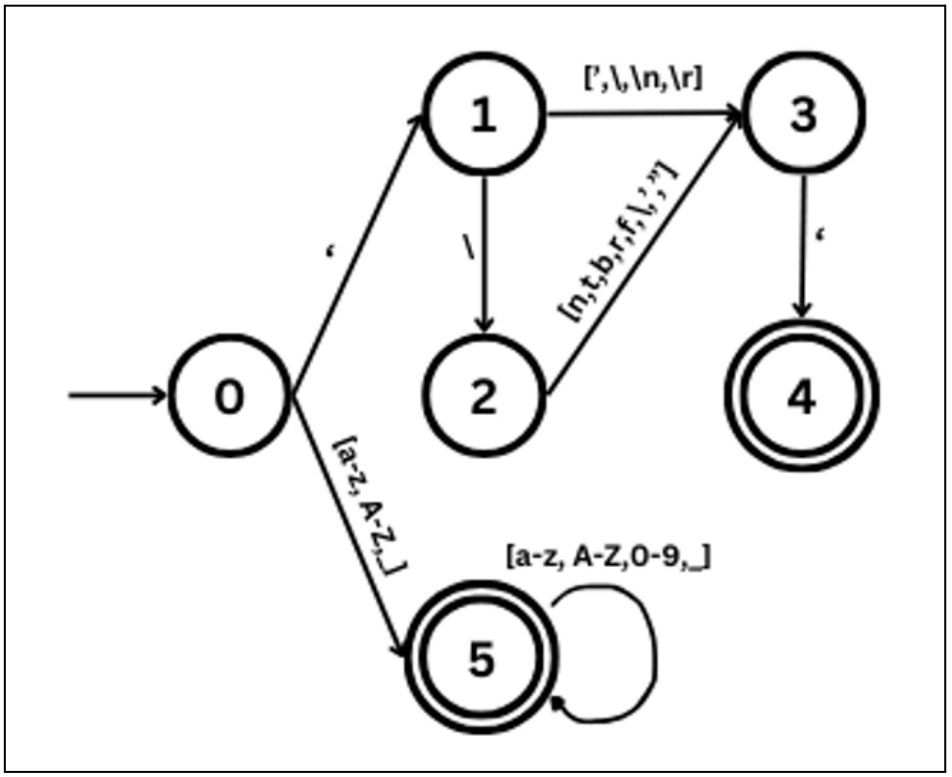
El trabajo concluirá con la realización de un conjunto de pruebas que validarán el correcto funcionamiento de la aplicación desarrollada.

AFD del analizador léxico

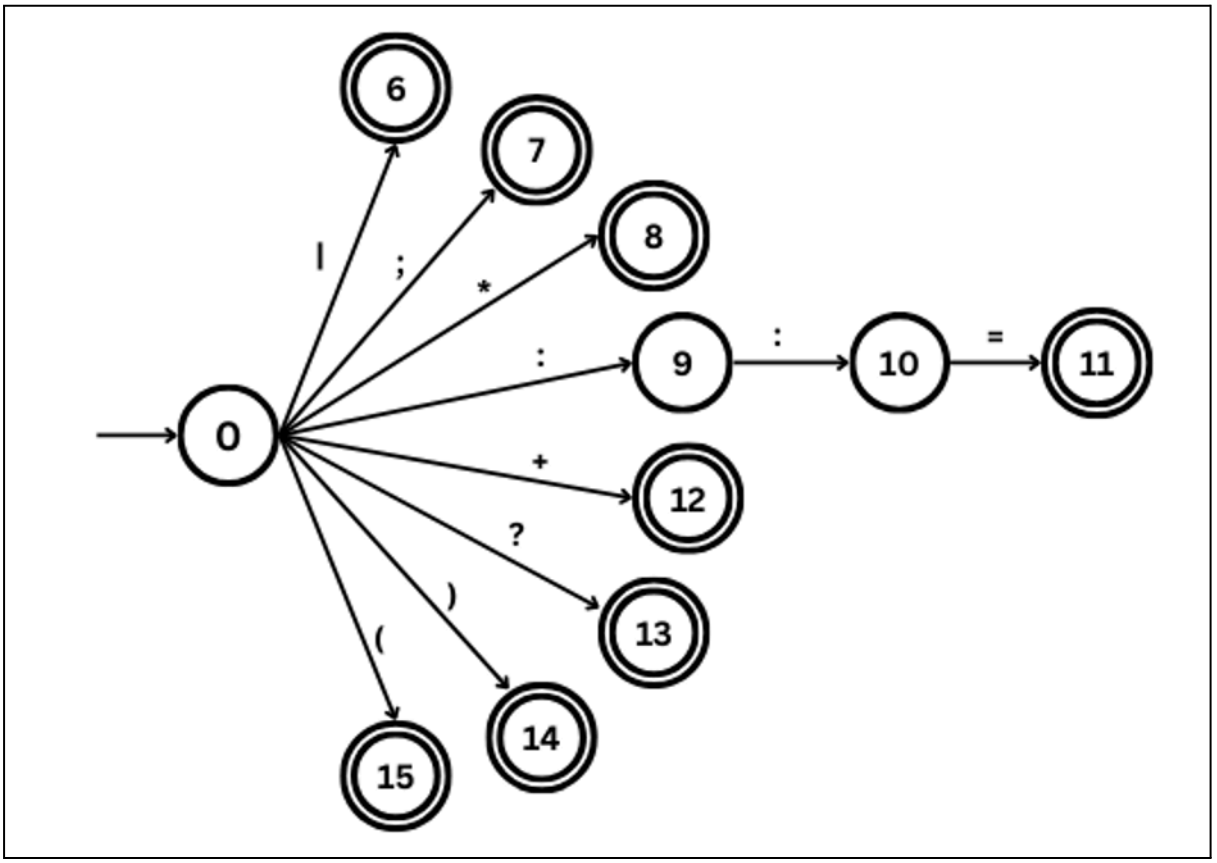
Blancos y comentarios



ID, SYMBOL



EQ, OR, SEMICOLON, RPAREN, LPAREN, STAR, PLUS HOOK



Clase que desarrolla el analizador léxico

Esta clase desarrolla el analizador léxico basado en una máquina discriminadora determinista, utilizando el AFD expuesto anteriormente.

Para implementar el analizador léxico ha sido necesario desarrollar los siguientes métodos: **transition(int,char)**, que contiene las transiciones del autómata finito determinista que describe el analizador léxico; **isFinal(int)**, que indica cuales son los estados finales del autómata; y **getToken(int, String, int, int)**, que genera el componente léxico asociado a cada estado final.

```
public class MyLexer extends Lexer implements TokenConstants {

    protected int transition(int state, char symbol) {
        switch (state) {
        case 0:
            if (symbol >= 'a' && symbol <= 'z')
                return 5;
            else if (symbol >= 'A' && symbol <= 'Z')
                return 5;
            else if (symbol == '_')
                return 5;
            else if (symbol == '\'')
                return 1;
            else if (symbol == '(')
                return 15;
            else if (symbol == ')')
                return 14;
            else if (symbol == ';')
                return 7;
            else if (symbol == '|')
                return 6;
            else if (symbol == '+')
                return 12;
            else if (symbol == '*')
                return 8;
            else if (symbol == '?')
                return 13;
            else if (symbol == ':')
                return 9;
            else if (symbol == ' ' || symbol == '\t')
                return 16;
            else if (symbol == '\r' || symbol == '\n')
                return 16;
            else if (symbol == '/')
                return 17;
            else
                return -1;
        case 1:
            if (symbol == '\\')
                return 2;
            else if (symbol != '\'' && symbol != '\n' && symbol != '\r')
```

```
            return 3;
        else
            return -1;
    case 2:
        if (symbol == 'n' || symbol == 't' || symbol == 'b')
            return 3;
        else if (symbol == 'r' || symbol == 'f' || symbol == '\\')
            return 3;
        else if (symbol == '\'' || symbol == '\"')
            return 3;
        else
            return -1;
    case 3:
        if (symbol == '\'')
            return 4;
        else
            return -1;
    case 5:
        if (symbol >= 'a' && symbol <= 'z')
            return 5;
        else if (symbol >= 'A' && symbol <= 'Z')
            return 5;
        else if (symbol >= '0' && symbol <= '9')
            return 5;
        else if (symbol == '_')
            return 5;
        else
            return -1;
    case 9:
        if (symbol == ':')
            return 10;
        else
            return -1;
    case 10:
        if (symbol == '=')
            return 11;
        else
            return -1;
    case 16:
        if (symbol == ' ' || symbol == '\t')
            return 16;
        else if (symbol == '\r' || symbol == '\n')
            return 16;
        else
            return -1;
```

```
    case 17:
        if (symbol == '*')
            return 18;
    case 18:
        if (symbol == '*')
            return 19;
        else
            return 18;
    case 19:
        if (symbol == '*')
            return 19;
        else if (symbol == '/')
            return 20;
        else
            return 18;
    default:
        return -1;
    }
}

protected boolean isFinal(int state) {
    if (state <= 0 || state > 20)
        return false;
    switch (state) {
    case 1:
    case 2:
    case 3:
    case 9:
    case 10:
    case 17:
    case 18:
    case 19:
        return false;
    default:
        return true;
    }
}
```

```
    protected Token getToken(int state, String lexeme, int row, int column) {
        switch (state) {
        case 4:
            return new Token(SYMBOL, lexeme, row, column);
        case 5:
            return new Token(ID, lexeme, row, column);
        case 6:
            return new Token(OR, lexeme, row, column);
        case 7:
            return new Token(SEMICOLON, lexeme, row, column);
        case 8:
            return new Token(STAR, lexeme, row, column);
        case 11:
            return new Token(EQ, lexeme, row, column);
        case 12:
            return new Token(PLUS, lexeme, row, column);
        case 13:
            return new Token(HOOK, lexeme, row, column);
        case 14:
            return new Token(RPAREN, lexeme, row, column);
        case 15:
            return new Token(LPAREN, lexeme, row, column);
        default:
            return null;
        }
    }

    public MyLexer(File file) throws IOException {
        super(file);
    }

    public MyLexer(FileInputStream fis) {
        super(fis);
    }
}
```

Gramática BNF del analizador sintáctico y sus conjuntos de predicción

---

La especificación sintáctica de las Expresiones Regulares es la siguiente:

- Fichero ::= **ID** **EQ** Expr **SEMICOLON**
- Expr ::= Option (**OR** Option)*
- Option ::= (Base)+
- Base ::= (**SYMBOL** | **LPAREN** Expr **RPAREN** Oper)
- Oper ::= (**STAR** | **PLUS** | **HOOK**)?

Un fichero de entrada contiene la definición de una única expresión regular. Esta definición comienza con un identificador (que representa el nombre que asignado a la expresión regular), seguida de un signo igual ("::=") y de la descripción de la expresión regular y termina en un punto y coma.

Las expresiones regulares están formadas por una serie de opciones (separadas por el símbolo "|"). Cada opción es una concatenación de expresiones básicas. Una expresión básica puede ser un símbolo (un literal de tipo char), un operador aplicado a una expresión regular (clausura, clausura positiva y opcionalidad) o una expresión regular entre paréntesis.

A partir de la gramática EBNF anterior, la gramática BNF obtenida, junto a sus conjuntos de predicción, es la siguiente:

REGLA	PREDICCIÓN
Fichero ::= **ID** **EQ** Expr **SEMICOLON**	**ID**
Expr ::= Option ExprClau	**SYMBOL, LPAREN**
ExprClau ::= **OR** Option ExprClau	**OR**
ExprClau ::= **lambda**	**SEMICOLON, RPAREN**

Option ::= Base OptionClauPos	**SYMBOL, LPAREN**
OptionClauPos ::= Base OptionClauPos	**SYMBOL, LPAREN**
OptionClauPos ::= **lambda**	**OR, SEMICOLON, RPAREN**
Base::= **SYMBOL**	**SYMBOL**
Base::= **LPAREN** Expr **RPAREN** Oper	**LPAREN**
Oper ::= **lambda**	**SEMICOLON, RPAREN, OR, SYMBOL, LPAREN**
Oper ::= **STAR**	**STAR**
Oper ::= **PLUS**	**PLUS**
Oper ::= **HOOK**	**HOOK**

REGLA	PRIMEROS	SIGUIENTES
Fichero	**ID**	**$**
Expr	**SYMBOL, LPAREN**	**SEMICOLON, RPAREN**
ExprClau	**OR, lambda**	**SEMICOLON, RPAREN**
Option	**SYMBOL, LPAREN**	**OR, SEMICOLON, RPAREN**
OptionClauPos	**SYMBOL, LPAREN, lambda**	**OR, SEMICOLON, RPAREN**
Base	**SYMBOL, LPAREN**	**SYMBOL, LPAREN, OR, SEMICOLON, RPAREN**
Oper	**STAR, PLUS, HOOK, lambda**	**SYMBOL, LPAREN, OR, SEMICOLON, RPAREN**

Clase que desarrolla el analizador sintáctico/semántico

A continuación se muestra la implementación del analizador sintáctico/semántico llamado **MyParser**, que procesa una secuencia de tokens generada por el analizador léxico (**MyLexer**).

La clase principal se inicializa con un flujo de entrada de archivo y tiene métodos para llevar a cabo el análisis, gestionar errores y sincronizar la cadena de tokens.
El análisis se estructura en varios métodos que representan las reglas gramaticales anteriores, teniendo cada una su correspondiente **try**[regla] y **parse**[regla], que devuelven los atributos sintetizados de la regla. Cada uno de estos métodos utiliza el método **match** para consumir tokens y **catchError** para manejar excepciones, manteniendo la cuenta y los mensajes de error. El método **skipTo** sincroniza los tokens en caso de error para retomar el análisis correctamente.

```java
package parser;

import java.io.FileInputStream;

public class MyParser implements TokenConstants {

    private MyLexer lexer; // Analizador lexico
    private Token nextToken; // Siguiente token de la cadena de entrada
    private Token prevToken; // Ultimo token analizado
    private int errorCount; // Contador de errores
    private String errorMsg; // Mensaje de errores

    public MyParser(FileInputStream fis)
    {
        this.lexer = new MyLexer(fis);
        this.prevToken = null;
        this.nextToken = lexer.getNextToken();
        this.errorCount = 0;
        this.errorMsg = "";
    }

    public int getErrorCount() // Obtiene el numero de errores del analisis
    {
        return this.errorCount;
    }

    public String getErrorMsg() // Obtiene el mensaje de error del analisis
    {
        return this.errorMsg;
    }

    private void catchError(Exception ex) // Almacena un error de analisis
    {
        this.errorCount++;
        this.errorMsg += ex.toString();
    }

    private void skipTo(int[] left, int[] right) { // Sincroniza la cadena de tokens
        boolean flag = false;
        if(prevToken.getKind() == EOF || nextToken.getKind() == EOF) flag = true;
        for(int i=0; i<left.length; i++) if(prevToken.getKind() == left[i]) flag = true;
        for(int i=0; i<right.length; i++) if(nextToken.getKind() == right[i]) flag = true;
```

```
        while(!flag)
        {
            prevToken = nextToken;
            nextToken = lexer.getNextToken();
            if(prevToken.getKind() == EOF || nextToken.getKind() == EOF) flag = true;
            for(int i=0; i<left.length; i++) if(prevToken.getKind() == left[i]) flag = true;
            for(int i=0; i<right.length; i++) if(nextToken.getKind() == right[i]) flag = true;
        }
    }

    private Token match(int kind) throws SintaxException { // Consume un token de la cadena de entrada

        if (nextToken.getKind() == kind) {
            prevToken = nextToken;
            nextToken = lexer.getNextToken();
            return prevToken;
        }
        else
            throw new SintaxException(nextToken, kind);
    }

    public Fichero parse() // Analiza un fichero de entrada
    {
        this.errorCount = 0;
        this.errorMsg = "";
        return tryFichero();
    }
```

```
public Fichero tryFichero() // Analiza el simbolo <Fichero>
{
    int[] lsync = { };
    int[] rsync = { EOF };

    Fichero fichero_s = null;

    try
    {
        fichero_s = parseFichero();
    }
    catch(Exception ex)
    {
        catchError(ex);
        skipTo(lsync,rsync);
    }

    return fichero_s;
}

private Fichero parseFichero() throws SintaxException {

    int[] expected = { ID };

    Fichero fichero_s = null;

    switch (nextToken.getKind()) {
    case ID:
        String id = nextToken.getLexeme();
        match(ID);
        match(EQ);
        OptionList expr_s = tryExpr();
        match(SEMICOLON);
        fichero_s = new Fichero(id, expr_s);
        break;
    default:
        throw new SintaxException(nextToken, expected);
    }

    return fichero_s;
}
```

```
public OptionList tryExpr() // Analiza el simbolo <Expr>
{
    int[] lsync = { };
    int[] rsync = { SEMICOLON, RPAREN };

    OptionList expr_s = null;

    try
    {
        expr_s = parseExpr();
    }
    catch(Exception ex)
    {
        catchError(ex);
        skipTo(lsync,rsync);
    }

    return expr_s;
}

private OptionList parseExpr() throws SintaxException {

    int[] expected = { SYMBOL, LPAREN };

    OptionList expr_s = null;

    switch (nextToken.getKind()) {
    case SYMBOL:
    case LPAREN:
        ConcatList option_s = tryOption();
        OptionList exprClau_s = tryExprClau();
        exprClau_s.addOption(option_s);
        expr_s = exprClau_s;
        break;
    default:
        throw new SintaxException(nextToken, expected);
    }

    return expr_s;
}
```

```java
public OptionList tryExprClau() // Analiza el simbolo <ExprClau>
{
    int[] lsync = { };
    int[] rsync = { SEMICOLON, RPAREN };

    OptionList exprClau_s = null;

    try
    {
        exprClau_s = parseExprClau();
    }
    catch(Exception ex)
    {
        catchError(ex);
        skipTo(lsync,rsync);
    }

    return exprClau_s;
}

private OptionList parseExprClau() throws SintaxException {

    int[] expected = { OR, SEMICOLON, RPAREN };

    OptionList exprClau_s = null;

    switch (nextToken.getKind()) {
    case OR:
        match(OR);
        ConcatList option_s = tryOption();
        OptionList exprClau1_s = tryExprClau();
        exprClau1_s.addOption(option_s);
        exprClau_s = exprClau1_s;
        break;
    case SEMICOLON:
    case RPAREN:
        exprClau_s = new OptionList();
        break;
    default:
        throw new SintaxException(nextToken, expected);
    }

    return exprClau_s;
}
```

```java
public ConcatList tryOption() // Analiza el simbolo <Option>
{
    int[] lsync = { };
    int[] rsync = { OR, SEMICOLON, RPAREN };

    ConcatList option_s = null;

    try
    {
        option_s = parseOption();
    }
    catch(Exception ex)
    {
        catchError(ex);
        skipTo(lsync,rsync);
    }

    return option_s;
}

private ConcatList parseOption() throws SintaxException {

    int[] expected = { SYMBOL, LPAREN };

    ConcatList option_s = null;

    switch (nextToken.getKind()) {
    case SYMBOL:
    case LPAREN:
        Expression base_s = tryBase();
        ConcatList optionClauPos_s = tryOptionClauPos();
        optionClauPos_s.concat(base_s);
        option_s = optionClauPos_s;
        break;
    default:
        throw new SintaxException(nextToken, expected);
    }

    return option_s;
}
```

```java
public ConcatList tryOptionClauPos() // Analiza el simbolo <OptionClauPos>
{
    int[] lsync = { };
    int[] rsync = { OR, SEMICOLON, RPAREN };

    ConcatList optionClauPos_s = null;

    try
    {
        optionClauPos_s = parseOptionClauPos();
    }
    catch(Exception ex)
    {
        catchError(ex);
        skipTo(lsync,rsync);
    }

    return optionClauPos_s;
}

private ConcatList parseOptionClauPos() throws SintaxException {

    int[] expected = { SYMBOL, LPAREN, OR, SEMICOLON, RPAREN };

    ConcatList optionClauPos_s = null;

    switch (nextToken.getKind()) {
    case SYMBOL:
    case LPAREN:
        Expression base_s = tryBase();
        ConcatList optionClauPos1_s = tryOptionClauPos();
        optionClauPos1_s.concat(base_s);
        optionClauPos_s = optionClauPos1_s;
        break;
    case SEMICOLON:
    case OR:
    case RPAREN:
        optionClauPos_s = new ConcatList();
        break;
    default:
        throw new SintaxException(nextToken, expected);
    }

    return optionClauPos_s;
}
```

```java
public Expression tryBase() // Analiza el simbolo <Base>
{
    int[] lsync = { };
    int[] rsync = { SYMBOL, LPAREN, OR, SEMICOLON, RPAREN };

    Expression base_s = null;

    try
    {
        base_s = parseBase();
    }
    catch(Exception ex)
    {
        catchError(ex);
        skipTo(lsync,rsync);
    }

    return base_s;
}

private Expression parseBase() throws SintaxException {

    int[] expected = { SYMBOL, LPAREN };

    Expression base_s = null;

    switch (nextToken.getKind()) {
    case SYMBOL:

        if(nextToken.getLexeme().length() > 1)
            base_s = new Symbol(nextToken.getLexeme().charAt(1));
        else
            base_s = new Symbol(nextToken.getLexeme().charAt(0));

        match(SYMBOL);
        break;
    case LPAREN:
        match(LPAREN);
        Expression expr_s = tryExpr();
        match(RPAREN);
        Expression oper_s = tryOper(expr_s);
        base_s = oper_s;
        break;
    default:
        throw new SintaxException(nextToken, expected);
    }

    return base_s;
}
```

```java
public Expression tryOper(Expression expr) // Analiza el simbolo <Oper>
{
    int[] lsync = { };
    int[] rsync = { SYMBOL, LPAREN, OR, SEMICOLON, RPAREN };

    Expression oper_s = null;

    try
    {
        oper_s = parseOper(expr);
    }
    catch(Exception ex)
    {
        catchError(ex);
        skipTo(lsync,rsync);
    }

    return oper_s;
}

private Expression parseOper(Expression expr) throws SintaxException {

    int[] expected = { STAR, PLUS, HOOK, SYMBOL, LPAREN, OR, SEMICOLON, RPAREN };

    Expression oper_s = null;

    switch (nextToken.getKind()) {
    case STAR:
        match(STAR);
        oper_s = new Operation(Operation.STAR, expr);
        break;
    case PLUS:
        match(PLUS);
        oper_s = new Operation(Operation.PLUS, expr);
        break;
    case HOOK:
        match(HOOK);
        oper_s = new Operation(Operation.HOOK, expr);
        break;
    case SYMBOL:
    case LPAREN:
    case OR:
    case SEMICOLON:
    case RPAREN:
        oper_s = expr;
        break;
    default:
        throw new SintaxException(nextToken, expected);
    }

    return oper_s;
}
```

Clases que desarrollan el Árbol de Sintaxis Abstracta

El Árbol de Sintaxis Abstracta (AST) es una representación crucial en el análisis de expresiones regulares, ya que permite estructurar y manipular de forma clara y eficiente las diferentes operaciones y componentes de una expresión regular. En este apartado, se presentan las clases desarrolladas para representar el AST de una expresión regular.

A continuación, se describe el código fuente de las clases que componen el AST. Estas clases incluyen una clase (Fichero) para la primera regla, que almacena toda la expresión regular, a la par que su identificador; y una clase abstracta base que define una expresión regular general, así como sus clases específicas para símbolos, listas de opciones, concatenaciones y operaciones como la clausura, la clausura positiva y la opcionalidad.

```java
public abstract class Expression {}
```

```java
public class Fichero {

    private String id;
    private Expression expr;

    public Fichero(String id, Expression expr) {
        this.id = id;
        this.expr = expr;
    }

    public String getId() {
        return id;
    }

    public Expression getExpr() {
        return expr;
    }
}
```

```java
// Clase que describe un lista de opciones " a | b | c "
public class OptionList extends Expression {

    private List<Expression> list;

    public OptionList() {
        list = new ArrayList<Expression>();
    }

    public OptionList(Expression exp) {
        list = new ArrayList<Expression>();
        list.add(exp);
    }

    public void addOption(Expression exp) {
        list.add(0, exp);
    }

    public List<Expression> getList() {
        return list;
    }
}
```

```java
//Clase que describe una concatenación de expresiones " a b c "
public class ConcatList extends Expression {

    private List<Expression> list;

    public ConcatList() {
        list = new ArrayList<Expression>();
    }

    public ConcatList(Expression exp) {
        list = new ArrayList<Expression>();
        list.add(exp);
    }

    public void concat(Expression exp) {
        list.add(0, exp);
    }

    public List<Expression> getList() {
        return list;
    }
}
```

```java
public class Operation extends Expression {

    public static final int STAR = 1;
    public static final int PLUS = 2;
    public static final int HOOK = 3;

    private int operator;
    private Expression operand;

    public Operation(int op, Expression exp) {

        operator = op;
        operand = exp;
    }

    public int getOperator() {
        return operator;
    }

    public Expression getOperand() {
        return operand;
    }

    public static String operatorToString(int operator) {

        switch (operator) {
        case 1:
            return "*";
        case 2:
            return "+";
        case 3:
            return "?";
        }

        return null;
    }

    @Override
    public String toString() {
        return operand.toString() + operatorToString(operator);
    }
}
```

```java
// Clase que describe un símbolo del lenguaje
public class Symbol extends Expression {

    private char symbol;

    public Symbol(char s) {
        symbol = s;
    }

    public char getSymbol() {
        return symbol;
    }

    @Override
    public String toString() {
        return Character.toString(symbol);
    }
}
```

Clases que desarrollan el algoritmo de cálculo del AFD

En el ámbito de la teoría de autómatas y lenguajes formales, la conversión de expresiones regulares en autómatas finitos deterministas es una técnica fundamental para el análisis y procesamiento de cadenas de texto.

A continuación se presenta la implementación en Java del algoritmo de expresiones regulares punteadas, que realiza dicha conversión, acompañado de una breve explicación de su funcionamiento. El algoritmo utiliza diversas estructuras de datos y métodos para llevar a cabo la transformación de una ER en un AFD.

```java
public class ExpPunteada {

    private Estado estado;
    private ArrayList<Transicion> transiciones;
    private ArrayList<Integer> posicionesPuntos;

    public ExpPunteada(Estado e, ArrayList<Integer> p, ArrayList<Transicion> t) {
        estado = e;
        posicionesPuntos = p;
        transiciones = t;
    }

    public void addTransicion(Transicion T) {
        transiciones.add(T);
    }

    public Estado getEstado() {
        return estado;
    }

    public ArrayList<Integer> getPosicionesPuntos() {
        return posicionesPuntos;
    }

    public ArrayList<Transicion> getTransiciones() {
        return transiciones;
    }
}
```

Esta clase auxiliar se utiliza para almacenar las expresiones regulares punteadas de forma cómoda y sencilla, para su posterior uso en el programa; y es esencial para entender el funcionamiento del algoritmo.

Cada expresión punteada es un estado, que tiene un conjunto de transiciones, y las posiciones de los puntos de las expresiones punteadas que contiene a su vez, que corresponden a cada uno de los literales de la expresión, de esta forma, acotamos el número de posiciones posibles del punto al número de literales de la expresión regular, ya que un punto sólo puede estar delante de un literal.

```java
ArrayList<Integer> posicionesPunto = new ArrayList<Integer>();
AtomicInteger literalAnalizado = new AtomicInteger(-1);
calcularExpPunteadas(ER, posicionesPunto, literalAnalizado, -1, 1);

ArrayList<ExpPunteada> expPunteadas = generarTablaEstadosTransiciones(posicionesPunto, ER);
```

Esta es la llamada al algoritmo que se realiza dentro del main(). Se inicializan las estructuras de datos necesarias para almacenar las posiciones de los puntos y los literales analizados. Posteriormente, se llama al método **calcularExpPunteadas** para identificar las posiciones punteadas en la expresión regular y, a continuación, se genera la tabla de estados y transiciones mediante el método **generarTablaEstadosTransiciones**.

```java
private static void calcularExpPunteadas(Expression exp, ArrayList<Integer> posicionesPunto,
        AtomicInteger literalAnalizado, int puntoInicial, int flag) {

    if (exp instanceof ConcatList) {

        List<Expression> concatList = ((ConcatList) exp).getList();
        int pos = 1;

        for (Expression e : concatList) {

            flag = actualizarPunteo(e, posicionesPunto, literalAnalizado, puntoInicial, flag);

            if (flag == 3) {
                flag = 1;
            }

            if (pos != concatList.size()) {

                if (flag == -1)
                    break;

            } else if (flag == 1)
                posicionesPunto.add(listaLiterales.size());

            pos++;
        }
    } else if (exp instanceof OptionList)
        calcularExpPunteadas(((OptionList) exp).getList().get(0), posicionesPunto, literalAnalizado, puntoInicial,
                flag);
}
```

Este método es responsable de calcular las posiciones punteadas en la expresión regular. Dependiendo del tipo de expresión (concatenación, opción, operación o símbolo), se procesan las subexpresiones y se actualizan las posiciones punteadas y el literal analizado. El comportamiento del método varía según el valor de la bandera flag, la cual indica el estado actual del proceso de punteado (-1, abortar punteado; 0, inicio; 1, seguir punteando; 2, buscar punteado; 3, concatenación completa).

```java
private static int actualizarPunteo(Expression exp, ArrayList<Integer> posicionesPunto,
        AtomicInteger literalAnalizado, int puntoInicial, int flag) {

    List<Expression> expList = null;

    if (exp instanceof OptionList) {

        expList = ((OptionList) exp).getList();

        switch (flag) {
        case 1:
            for (Expression e : expList) {
                actualizarPunteo(e, posicionesPunto, literalAnalizado, puntoInicial, flag);
            }

            return -1;
        case 2:
            for (Expression e : expList) {

                flag = actualizarPunteo(e, posicionesPunto, literalAnalizado, puntoInicial, flag);

                if (flag == -1 || flag == 3)
                    return flag;

            }

            return flag;
        }

    } else if (exp instanceof ConcatList) {

        expList = ((ConcatList) exp).getList();

        int pos = 1;

        switch (flag) {
        case 1:
            for (Expression e : expList) {

                flag = actualizarPunteo(e, posicionesPunto, literalAnalizado, puntoInicial, flag);

                if (flag == -1) {

                    literalAnalizado.set(literalAnalizado.get() + (expList.size() - pos));
                    break;
                }
                pos++;
            }

            return -1;
```

```
    case 2:
        for (Expression e : expList) {

            flag = actualizarPunteo(e, posicionesPunto, literalAnalizado, puntoInicial, flag);

            if (pos == expList.size()) {
                if (flag == 1)
                    return 3;
            } else if (flag == -1) {
                literalAnalizado.set(literalAnalizado.get() + (expList.size() - pos));
                return -1;
            }
            pos++;
        }

        return flag;
    }

} else if (exp instanceof Operation) {

    Expression operando = ((Operation) exp).getOperand();
    String operador = Operation.operatorToString(((Operation) exp).getOperator());

    switch (flag) {
    case 1:
        actualizarPunteo(operando, posicionesPunto, literalAnalizado, puntoInicial, flag);

        if (operador == "+")
            return -1;
        else if (operador == "*")
            return 1;

    case 2:
        if (operador == "+" || operador == "*") {

            AtomicInteger literalAnalizadoAux = new AtomicInteger(literalAnalizado.get());

            flag = actualizarPunteo(operando, posicionesPunto, literalAnalizado, puntoInicial, flag);

            if (flag == 3) {
                actualizarPunteo(operando, posicionesPunto, literalAnalizadoAux, puntoInicial, 1);
                literalAnalizado.set(literalAnalizadoAux.get());
                return 1;
            }

            if (flag == 2)
                return flag;
            else
                return -1;
        }
    }
```

```
    } else if (exp instanceof Symbol) {

        literalAnalizado.incrementAndGet();

        switch (flag) {
        case 0:
        case 1:
            posicionesPunto.add(literalAnalizado.get());
            return -1;
        case 2:
            if (literalAnalizado.get() == puntoInicial)
                return 1;
            else if (literalAnalizado.get() < puntoInicial)
                return 2;
        }
    }
    return 0;
}
```

El método **actualizarPunteo** actualiza las posiciones punteadas y el literal analizado según el tipo de expresión que se está procesando. Dependiendo del tipo de operador (concatenación, opción, operación) y del valor de la bandera flag, se ajustan las posiciones de los puntos y los literales, de acuerdo al algoritmo estudiado en el tema 2 de la asignatura.

```
private static ArrayList<ExpPunteada> generarTablaEstadosTransiciones(ArrayList<Integer> posicionesPunto,
        Expression expRegular) {

    Estado estadoIni = new Estado(0, posicionesPunto.contains(listaLiterales.size()));
    ArrayList<Transicion> transicionesEstadoIni = new ArrayList<>();

    ArrayList<ExpPunteada> expPunteadas = new ArrayList<ExpPunteada>();
    expPunteadas.add(new ExpPunteada(estadoIni, posicionesPunto, transicionesEstadoIni));

    for (int i = 0; i < expPunteadas.size(); i++) {

        addTransiciones(expPunteadas.get(i), expPunteadas, expRegular);

        if (expPunteadas.get(i).getEstado().esEstadoFinal())
            System.out.print("*");

        System.out.println("E" + expPunteadas.get(i).getEstado().getNumero() + ":");

        for (Transicion t : expPunteadas.get(i).getTransiciones()) {
            System.out.println(t.getSimbolo() + " -> " + t.getDestino());
        }

        System.out.println();
    }

    return expPunteadas;
}
```

Este método se encarga de generar la tabla de estados y transiciones del AFD a partir de las posiciones punteadas obtenidas en la etapa anterior. Se inicializa el estado inicial del AFD y se itera sobre las expresiones punteadas para añadir las transiciones correspondientes. Finalmente, se retorna una lista de **ExpPunteada** que representa el AFD.

```java
private static void addTransiciones(ExpPunteada expPunteada, ArrayList<ExpPunteada> expPunteadas,
        Expression expRegular) {

    ArrayList<ArrayList<Integer>> listaPosPuntos = new ArrayList<ArrayList<Integer>>();

    for (int punto : expPunteada.getPosicionesPuntos()) {

        if (punto == listaLiterales.size())
            continue;

        AtomicInteger literalAnalizado = new AtomicInteger(-1);
        ArrayList<Integer> posPuntos = new ArrayList<Integer>();

        calcularExpPunteadas(expRegular, posPuntos, literalAnalizado, punto, 2);

        listaPosPuntos.add(posPuntos);
    }

    if (!listaPosPuntos.isEmpty()) {

        ArrayList<Character> listaLiteralesPunteados = getListaLiteralesPunteados(
                expPunteada.getPosicionesPuntos());

        if (listaLiteralesPunteados.size() <= 1) {

            ArrayList<Integer> posPuntos = listaPosPuntos.get(0);
            int numEstado = estado(expPunteadas, posPuntos);

            char literalPunteado = listaLiteralesPunteados.get(0);
            Transicion t = new Transicion(literalPunteado, numEstado);

            expPunteada.addTransicion(t);

            if (numEstado == expPunteadas.size()) {

                Estado e = new Estado(numEstado, posPuntos.contains(listaLiterales.size()));
                expPunteadas.add(new ExpPunteada(e, posPuntos, new ArrayList<>()));
            }

        } else {

            Set<Character> set = new LinkedHashSet<>(listaLiteralesPunteados);
            ArrayList<Character> listaLiteralesNoRepetidos = new ArrayList<>(set);

            ArrayList<ArrayList<Object>> posLiterales = new ArrayList<ArrayList<Object>>();

            for (Character literalNoRep : listaLiteralesNoRepetidos) {

                posLiterales.add(new ArrayList<Object>());
                posLiterales.get(posLiterales.size() - 1).add(literalNoRep);
```

```
            for (int i = 0; i < listaLiteralesPunteados.size(); i++) {

                if (literalNoRep == listaLiteralesPunteados.get(i))
                    posLiterales.get(posLiterales.size() - 1).add(i);
            }

            if (posLiterales.get(posLiterales.size() - 1).size() == 2) {
                posLiterales.remove(posLiterales.size() - 1);
            }
        }

        for (int i = 0; i < listaLiteralesPunteados.size(); i++) {

            ArrayList<Integer> posPuntos = listaPosPuntos.get(i);
            ArrayList<Object> listaLiteralesRep = getListaLiteralesRep(listaLiteralesPunteados.get(i),
                    posLiterales);

            if (listaLiteralesRep != null) {

                if (listaLiteralesRep.size() <= 1)
                    continue;

                else {

                    int l1 = (Integer) listaLiteralesRep.get(1);

                    for (int j = 2; j < listaLiteralesRep.size(); j++) {

                        int l2 = (Integer) listaLiteralesRep.get(j);

                        listaPosPuntos.get(l1).addAll(listaPosPuntos.get(l2));
                    }

                    Set<Integer> set2 = new LinkedHashSet<>(listaPosPuntos.get(l1));
                    posPuntos = new ArrayList<>(set2);

                    listaLiteralesRep.clear();
                    listaLiteralesRep.add(listaLiteralesPunteados.get(i));
                }
            }

            int numEstado = estado(expPunteadas, posPuntos);
            Transicion t = new Transicion(listaLiteralesPunteados.get(i), numEstado);
            expPunteada.addTransicion(t);

            if (expPunteadas.size() == numEstado) {

                Estado e = new Estado(numEstado, posPuntos.contains(listaLiterales.size()));
                ExpPunteada exp = new ExpPunteada(e, posPuntos, new ArrayList<>());
                expPunteadas.add(exp);
            }
        }
```

Este método añade las transiciones entre los estados del AFD. Para cada posición punteada, se calcula el estado de destino y se añade la transición correspondiente. Si el estado de destino no existe, se crea un nuevo estado y se añade a la lista de expresiones punteadas.

```java
private static int estado(ArrayList<ExpPunteada> expPunteadas, ArrayList<Integer> posPuntos) {

    int estado = 0;

    for (ExpPunteada exp : expPunteadas) {

        Collections.sort(posPuntos);
        Collections.sort(exp.getPosicionesPuntos());

        if (Objects.equals(posPuntos, exp.getPosicionesPuntos()))
            return estado;

        estado++;
    }

    return estado;
}
```

El método **estado** verifica si un conjunto de posiciones punteadas ya corresponde a un estado existente en el AFD. Si es así, retorna el número del estado; de lo contrario, crea un nuevo estado y lo añade a la lista de expresiones punteadas.

```java
private static ArrayList<Object> getListaLiteralesRep(Character literal,
        ArrayList<ArrayList<Object>> posLiterales) {

    for (ArrayList<Object> literales : posLiterales) {

        if ((Character) literales.get(0) == literal)
            return literales;
    }

    return null;
}

private static ArrayList<Character> getListaLiteralesPunteados(ArrayList<Integer> posPuntos) {

    ArrayList<Character> listaLiteralesPunteados = new ArrayList<Character>();

    for (int i : posPuntos) {
        if (i != listaLiterales.size())
            listaLiteralesPunteados.add(listaLiterales.get(i));
    }

    return listaLiteralesPunteados;
}
```

Por último, estos métodos auxiliares se utilizan para gestionar las listas de literales y posiciones punteadas, evitando duplicados y facilitando la asociación entre literales y sus posiciones correspondientes.

Clases que describen el AFD

Para completar la explicación del algoritmo que convierte una expresión regular en un autómata finito determinista, es esencial comprender las clases que describen la estructura del AFD generado por el algoritmo anterior. Estas clases son fundamentales para la representación de los estados y transiciones del autómata, permitiendo su manipulación y análisis. A continuación, se presenta el código de las clases **Estado** y **Transicion**:

```java
public class Estado {

    private boolean estadoFinal;
    private int numero;

    public Estado(int n, boolean ef) {
        numero = n;
        estadoFinal = ef;
    }

    public boolean esEstadoFinal() {
        return estadoFinal;
    }

    public int getNumero() {
        return numero;
    }
}
```

```java
public class Transicion {

    private int destino;
    private char simbolo;

    public Transicion(char s, int d) {
        simbolo = s;
        destino = d;
    }

    public char getSimbolo() {
        return simbolo;
    }

    public int getDestino() {
        return destino;
    }
}
```

Método que genera el fichero ".java"

En la construcción de un autómata finito determinista a partir de una expresión regular, es crucial disponer de una representación programática que permita su uso práctico en diversas aplicaciones. Una forma efectiva de lograr esto es generar un archivo Java que contenga la implementación del AFD.

El siguiente método, **generarFichero**, es responsable de crear un archivo Java que define la clase del AFD, imitando la estructura de la clase de ejemplo. Este archivo incluye métodos para manejar las transiciones de estado y verificar si un estado es final. A continuación se presenta el código:

```
private static void generarFichero(String id, ArrayList<ExpPunteada> expPunteadas) throws IOException {

    String fichero = "public class " + id + "{\n\n" + "\tpublic int transition(int state, char symbol) {\n"
            + " \t\tswitch(state) {\n";

    for (ExpPunteada e : expPunteadas) {
        fichero += "\t\t\tcase " + e.getEstado().getNumero() + ": \n";

        for (Transicion t : e.getTransiciones()) {
            fichero += "\t\t\t\tif(symbol == '" + t.getSimbolo() + "') return " + t.getDestino() + ";\n";
        }

        fichero += "\t\t\t\treturn -1; \n";
    }

    fichero += "\t\t\tdefault:\n" + "\t\t\t\treturn -1;\n" + "\t\t}\n" + "\t}\n\n"
            + "\tpublic boolean isFinal(int state) {\n" + "\t\tswitch(state) {\n";

    for (ExpPunteada e : expPunteadas) {
        fichero += "\t\t\tcase " + e.getEstado().getNumero() + ": return " + e.getEstado().esEstadoFinal()
                + ";\n";
    }

    fichero += "\t\t\tdefault: return false;\n" + "\t\t}\n" + "\t}\n" + "}";

    BufferedWriter writer = new BufferedWriter(new FileWriter(id + ".java"));
    writer.write(fichero);
    writer.close();
}
```

Este método toma como parámetros el identificador de la regla **Fichero**, obtenido en el análisis semántico, y una lista de expresiones regulares punteadas. A continuación, crea el contenido del archivo recorriendo el conjunto de expresiones punteadas generado anteriormente, a partir de las cuales se obtienen fácilmente los estados y sus transiciones, además de poder saber cuáles son los estados finales.

Finalmente, escribe este contenido en un archivo con el nombre especificado, utilizando BufferedWriter.

Pruebas del funcionamiento

A continuación se mostrarán varias pruebas de funcionamiento distintas del programa, es decir, se le introducirán varias expresiones regulares distintas y se presentará el fichero ".java" generado.

Comment ::= 'b' 'a' (('a')* 'o' | 'b')* ('a')+ 'b' ;

```java
public class Comment {

    public int transition(int state, char symbol) {
        switch(state) {
            case 0:
                if(symbol == 'b') return 1;
                return -1;
            case 1:
                if(symbol == 'a') return 2;
                return -1;
            case 2:
                if(symbol == 'a') return 3;
                if(symbol == 'o') return 2;
                if(symbol == 'b') return 2;
                return -1;
            case 3:
                if(symbol == 'a') return 3;
                if(symbol == 'o') return 2;
                if(symbol == 'b') return 4;
                return -1;
            case 4:
                return -1;
            default:
                return -1;
        }
    }

    public boolean isFinal(int state) {
        switch(state) {
            case 0: return false;
            case 1: return false;
            case 2: return false;
            case 3: return false;
            case 4: return true;
            default: return false;
        }
    }
}
```

Expresion ::= 'b' 'a' (('b' | 'o')* 'a' ('a')* 'o')* ('b' | 'o')* 'a' ('a')* 'b' ;

```java
public class Expresion {

    public int transition(int state, char symbol) {
        switch(state) {
            case 0:
                if(symbol == 'b') return 1;
                return -1;
            case 1:
                if(symbol == 'a') return 2;
                return -1;
            case 2:
                if(symbol == 'b') return 2;
                if(symbol == 'o') return 2;
                if(symbol == 'a') return 3;
                return -1;
            case 3:
                if(symbol == 'a') return 3;
                if(symbol == 'o') return 2;
                if(symbol == 'b') return 4;
                return -1;
            case 4:
                return -1;
            default:
                return -1;
        }
    }

    public boolean isFinal(int state) {
        switch(state) {
            case 0: return false;
            case 1: return false;
            case 2: return false;
            case 3: return false;
            case 4: return true;
            default: return false;
        }
    }
}
```

(Ejercicio 2.13 Boletín)

Expresion2 ::= 'b' 'a' ('a' | 'b' | 'o')* 'a' 'b' ;

```java
public class Expresion2 {

    public int transition(int state, char symbol) {
        switch(state) {
            case 0:
                if(symbol == 'b') return 1;
                return -1;
            case 1:
                if(symbol == 'a') return 2;
                return -1;
            case 2:
                if(symbol == 'a') return 3;
                if(symbol == 'b') return 2;
                if(symbol == 'o') return 2;
                return -1;
            case 3:
                if(symbol == 'a') return 3;
                if(symbol == 'b') return 4;
                if(symbol == 'o') return 2;
                return -1;
            case 4:
                if(symbol == 'a') return 3;
                if(symbol == 'b') return 2;
                if(symbol == 'o') return 2;
                return -1;
            default:
                return -1;
        }
    }

    public boolean isFinal(int state) {
        switch(state) {
            case 0: return false;
            case 1: return false;
            case 2: return false;
            case 3: return false;
            case 4: return true;
            default: return false;
        }
    }
}
```

(Ejercicio 2.20 Boletín)

Como se puede comprobar en los boletines, los ficheros generados son correctos, por lo que podemos concluir que nuestro programa funciona perfectamente a la hora de generar un AFD a partir de una expresión regular cualquiera, mediante el algoritmo de expresiones regulares punteadas.