Procesadores de Lenguaje

ETSI
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA
uhu.es

Trabajo final

Generador de Autómatas Finitos Deterministas

Autor: Daniel Linfon Ye Liu

Curso 2023/2024

Índice

Introducción

El propósito de este proyecto es crear manualmente una aplicación que pueda generar un Autómata Finito Determinista (AFD) a partir de la descripción de una expresión regular. Este proceso se llevará a cabo utilizando el algoritmo de expresiones regulares punteadas.

Analizador Léxico

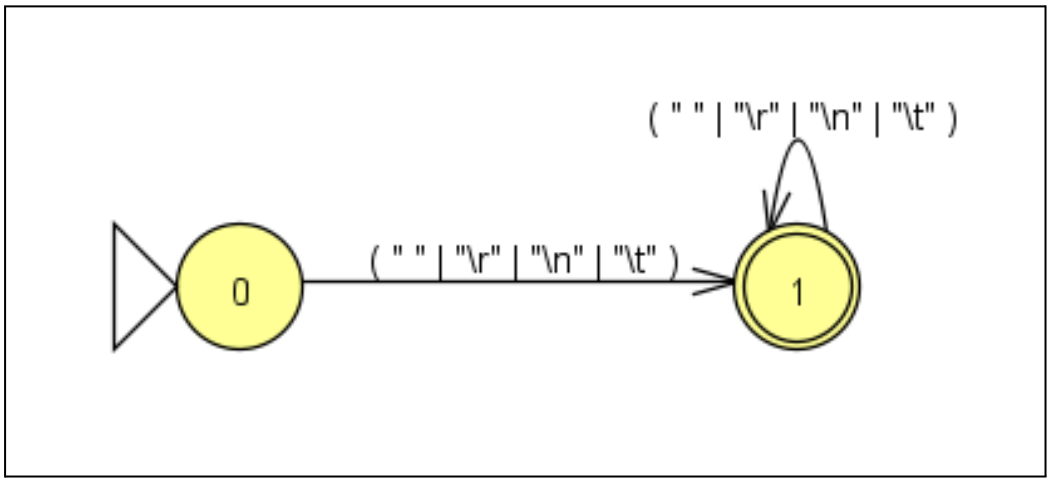
La especificación léxica de las descripciones de Expresiones Regulares están formadas por las siguientes categorías:

Especificación	Expresión regular
Blanco	(" " | "\r" | "\n" | "\t")
Comentario	"/*" (("*")* ~["*", "/"] | "/")* ("*")+ "/"
ID	["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])*
SYMBOL	" ' " (~[" ' ","\\","\n","\r"]) | ("\\" ["n","t","b","r","f","\\","'","\""]) " ' "
EQ	"::="
OR	"|"
SEMICOLON	";"
LPAREN	"("
RPAREN	")"
STAR	"*"
PLUS	"+"
HOOK	"?"

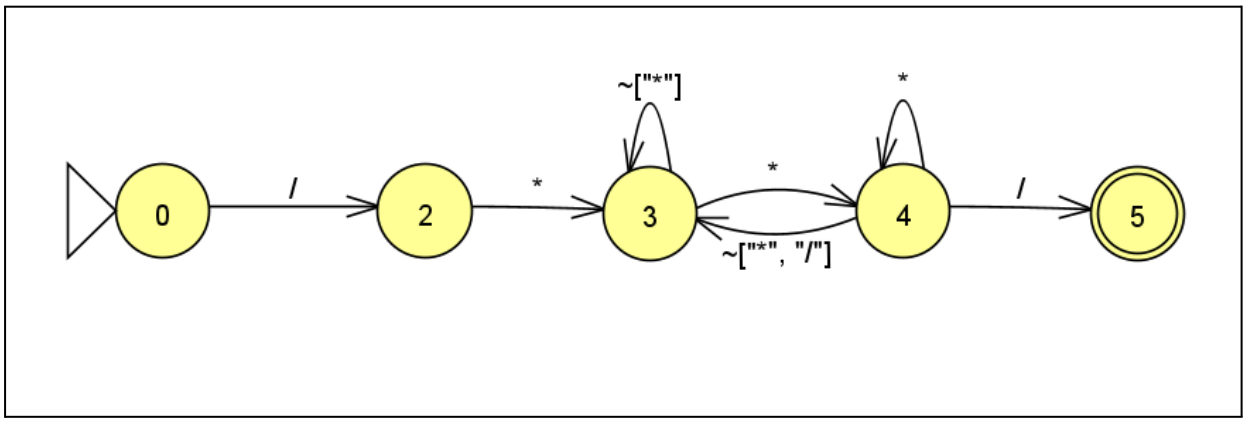
A continuación, hemos diseñado el Autómata Finito Determinista en el que se basa el analizador léxico.

AFD del analizador léxico

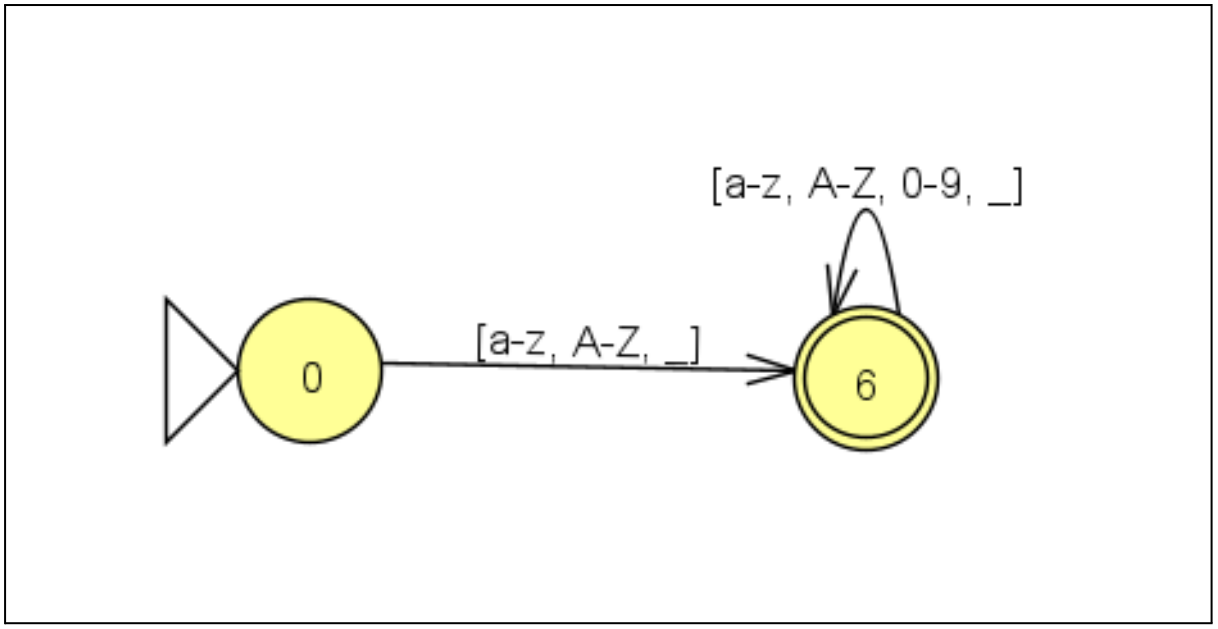
- Blanco



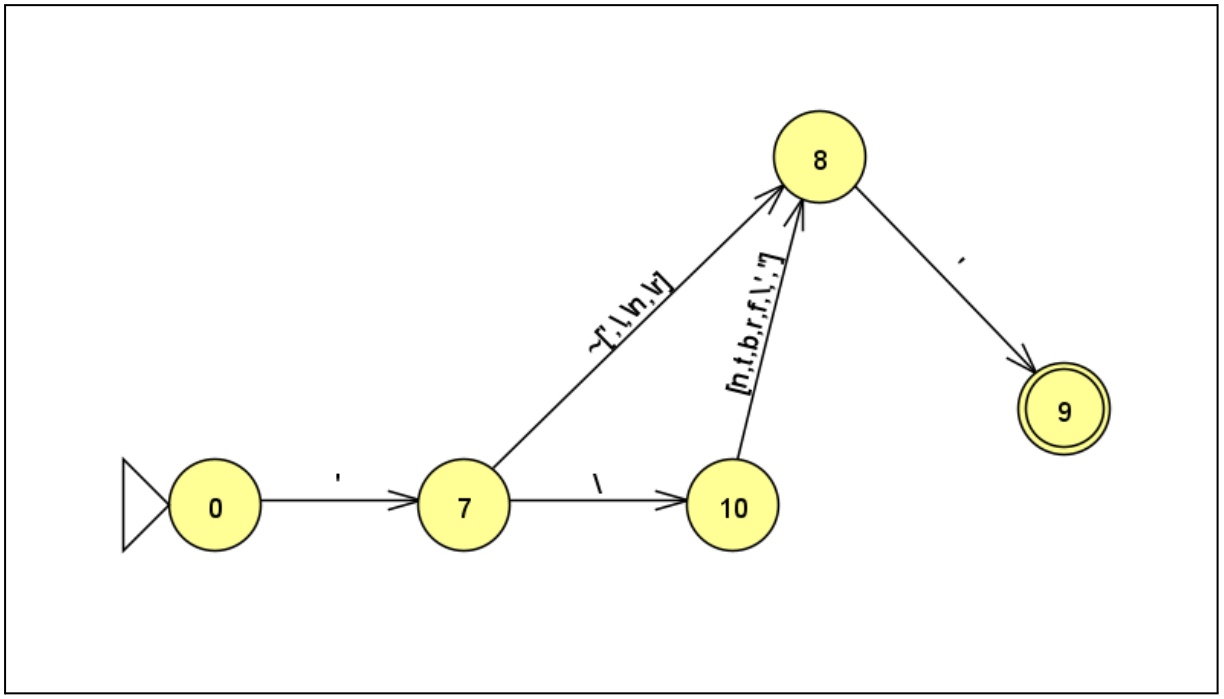
- Comentario



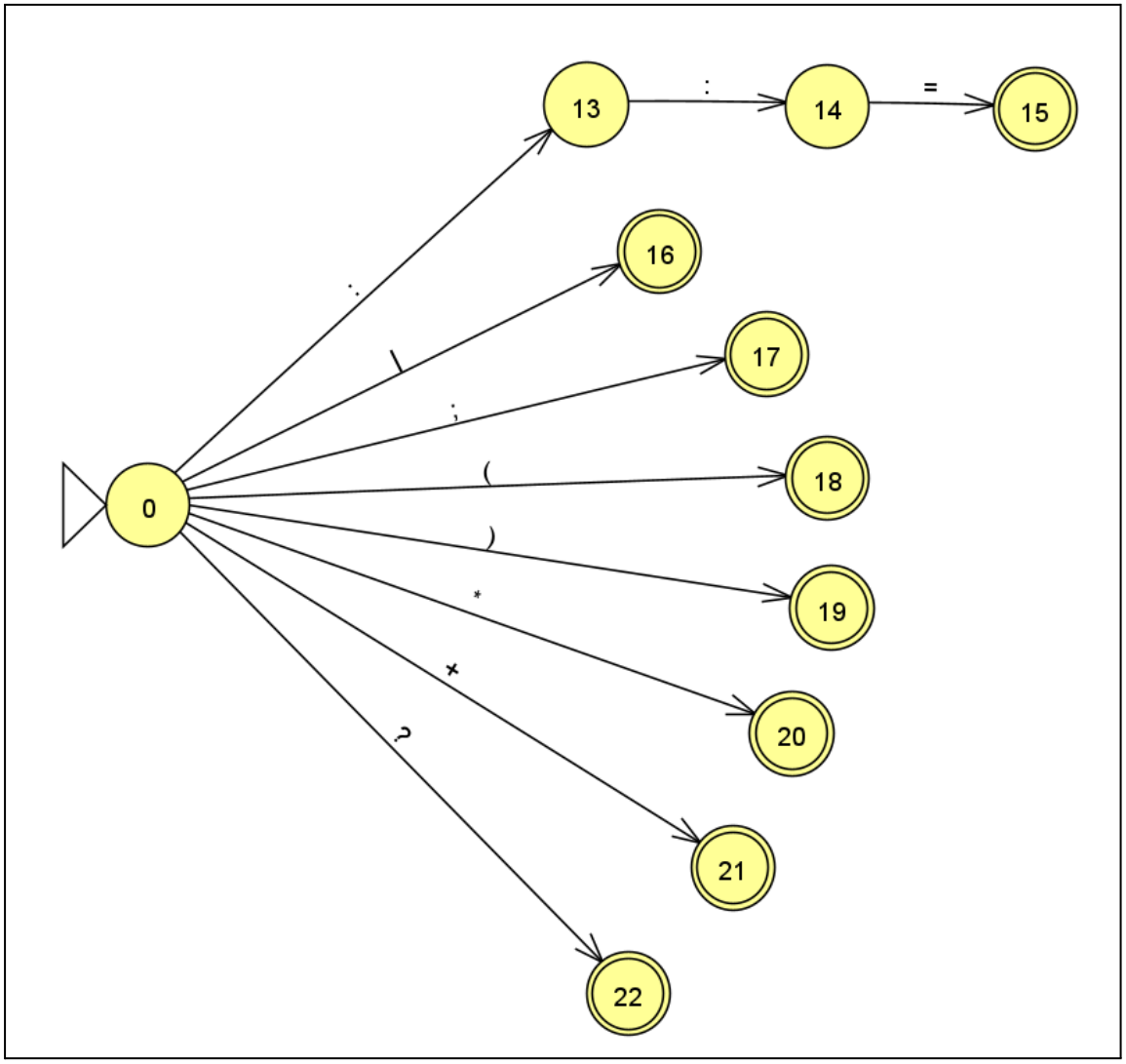
- Identificador



- SYMBOL

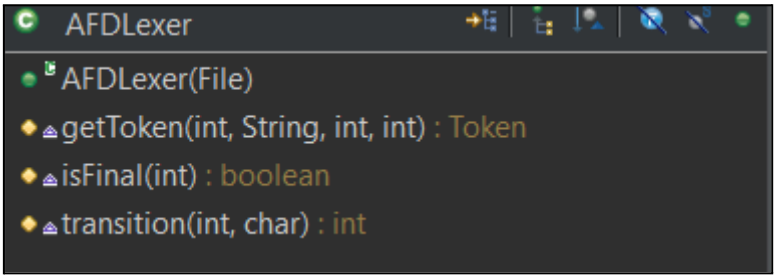


- EQ, OR, SEMICOLON, LPAREN, RPAREN, STAR, PLUS, HOOK



Código de la clase “AFDLexer” que desarrolla el analizador léxico.

Está formado por los siguientes métodos:



- **`protected boolean isFinal(int state):`** Nos sirve para verificar si un estado es final (true) o no (false).

```
protected boolean isFinal(int state)
{
    if(state <=0 || state > 22) return false;
    switch(state)
    {
        case 1:
        case 5:
        case 6:
        case 9:
        case 15:
        case 16:
        case 17:
        case 18:
        case 19:
        case 20:
        case 21:
        case 22:
            return true;
        default:
            return false;
    }
}
```

- **`protected Token getToken(int state, String lexeme, int row, int column):`** Genera el componente léxico correspondiente al estado final y al lexema encontrado.

```
protected Token getToken(int state, String lexeme, int row, int column)
{
    switch(state)
    {
        case 1: return null;
        case 5: return null;
        case 6: return new Token(IDENTIFIER, lexeme, row, column);
        case 9: return new Token(SYMBOL, lexeme, row, column);
        case 15: return new Token(EQ, lexeme, row, column);
        case 16: return new Token(OR, lexeme, row, column);
        case 17: return new Token(SEMICOLON, lexeme, row, column);
        case 18: return new Token(LPAREN, lexeme, row, column);
        case 19: return new Token(RPAREN, lexeme, row, column);
        case 20: return new Token(STAR, lexeme, row, column);
        case 21: return new Token(PLUS, lexeme, row, column);
        case 22: return new Token(HOOK, lexeme, row, column);
        default: return null;
    }
}
```

- **`protected int transition(int state, char symbol):`** Contiene las transiciones posibles desde cada estado del AFD.

```
protected int transition(int state, char symbol)
{
    switch(state)
    {
        case 0:

            //Blanco
            if(symbol == ' ' || symbol == '\t' || symbol == '\r' || symbol == '\n') return 1;

            //Comentario
            else if(symbol == '/') return 2;

            //-----------------Identificador----------------------
            else if(symbol >= 'a' && symbol <= 'z') return 6;
            else if(symbol >= 'A' && symbol <= 'Z') return 6;
            else if(symbol == '_') return 6;
            //----------------------------------------------------

            else if(symbol == '\'') return 7; //Symbol
            else if(symbol == ':') return 13; //EQ
            else if(symbol == '|') return 16;//OR
            else if(symbol == ';') return 17;//SEMICOLON
            else if(symbol == '(') return 18;//LPAREN
            else if(symbol == ')') return 19;//RPAREN
            else if(symbol == '*') return 20;//STAR
            else if(symbol == '+') return 21;//PLUS
            else if(symbol == '?') return 22;//HOOK
            else return -1;
```

Transiciones que parten desde el estado inicial

Analizador Sintáctico

Gramática BNF

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**)?

Transformándolo a gramática BNF:

- Fichero -> **ID EQ** Expr **SEMICOLON**
- Expr -> Option OrOption
- OrOption- > λ
- OrOption-> **OR** Option OrOption
- Option -> Base BasePositive
- BasePositive-> λ
- BasePositive-> Base BasePositive
- Base -> **SYMBOL**
- Base -> **LPAREN** Expr **RPAREN** Oper
- Oper -> StarPlusHook
- StarPlusHook-> λ
- StarPlusHook-> **STAR**
- StarPlusHook-> **PLUS**
- StarPlusHook-> **HOOK**

Conjunto de predicciones

Para comprobar que cumple con la propiedad LL(1), calculamos el conjunto de predicciones:

Reglas	Primeros	Siguientes	Predicción
Fichero -> **ID EQ** Expr **SEMICOLON**	ID	$	ID
Expr -> Option OrOption	SYMBOL, LPAREN	SEMICOLON, RPAREN	SYMBOL, LPAREN
OrOption- > λ	λ	SEMICOLON, RPAREN	SEMICOLON, RPAREN
OrOption-> OR Option OrOption	OR		OR
Option -> Base BasePositive	SYMBOL, LPAREN	SEMICOLON, RPAREN, OR	SYMBOL, LPAREN
BasePositive-> λ	λ	SEMICOLON, RPAREN, OR, SYMBOL, LPAREN	SEMICOLON, RPAREN, OR, SYMBOL, LPAREN
BasePositive-> Base BasePositive	LPAREN, SYMBOL		LPAREN, SYMBOL
Base -> **SYMBOL**	SYMBOL	SEMICOLON, RPAREN, OR, SYMBOL, LPAREN	SYMBOL
Base -> **LPAREN** Expr **RPAREN** Oper	LPAREN		LPAREN
Oper -> StarPlusHook	STAR, PLUS, HOOK	SEMICOLON, RPAREN, OR, SYMBOL, LPAREN	STAR,PLUS,HOOK
StarPlusHook-> λ	λ	SEMICOLON, RPAREN, OR, SYMBOL, LPAREN	SEMICOLON, RPAREN, OR, SYMBOL, LPAREN
StarPlusHook-> **STAR**	STAR		STAR
StarPlusHook-> **PLUS**	PLUS		PLUS
StarPlusHook-> **HOOK**	HOOK		HOOK

Al ser los valores disjuntos, cumple con la propiedad LL(1).

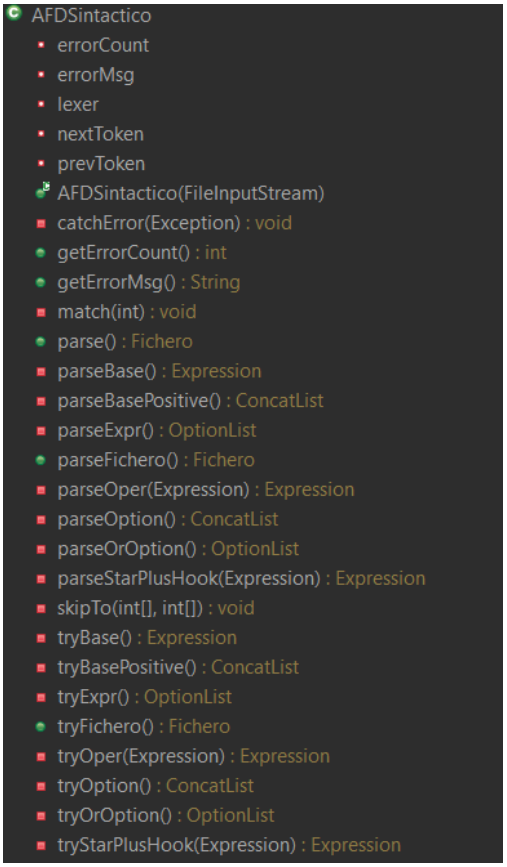
Código de la clase “AFDParser” que desarrolla el analizador sintáctico/semántico.

A partir de la gramática LL(1) expresada en notación BNF y de los conjuntos de predicción calculados para cada regla de la gramática se ha desarrollado un analizador sintáctico descendente recursivo.

Para ello, se ha creado la clase `AFDParser` que contiene principalmente dos miembros privados: el analizador léxico (*lexer*), utilizado para obtener el flujo de tokens de entrada, y el token de preanálisis (*nextToken*), que es el que determina la regla a ejecutar en el análisis descendente.

La implementación del analizador sintáctico descendente recursivo consiste en crear un nuevo método asociado a cada símbolo no terminal de la gramática, de manera que a cada símbolo *A* se le va a asociar los métodos *parseA()* y *tryA()*. El método *tryA()* envuelve la llamada al método *parseA()* en un bloque *try-catch*, de manera que si se produce una excepción sintáctica en *parseA()* se captura el error y se sincroniza el analizador.

Los métodos que contiene la clase son los siguientes:



El código de estas funciones es similar para todos los símbolos por lo que solo se muestra algunos ejemplos:

```
public Fichero tryFichero() {
    int[] lsync = {};
    int[] rsync = { EOF };
    Fichero f = null;
    try {
        f = parseFichero();
    } catch (Exception ex) {
        catchError(ex);
        skipTo(lsync, rsync);
    }
    return f;
}
```

```
public Fichero parseFichero() throws SintaxException {
    int[] expected = { IDENTIFIER };
    OptionList s = null;
    Fichero f = null;
    switch (nextToken.getKind()) {
    case IDENTIFIER:
        String id = nextToken.getLexeme();
        match(IDENTIFIER);
        match(EQ);
        s = tryExpr();
        match(SEMICOLON);
        f = new Fichero(id, s);
        break;
    default:
        throw new SintaxException(nextToken, expected);
    }
    return f;
}
```

```
private ConcatList tryOption() {
    int[] lsync = {};
    int[] rsync = { SEMICOLON, RPAREN, OR };
    ConcatList s = null;
    try {
        s = parseOption();
    } catch (Exception ex) {
        catchError(ex);
        skipTo(lsync, rsync);
    }
    return s;
}
```

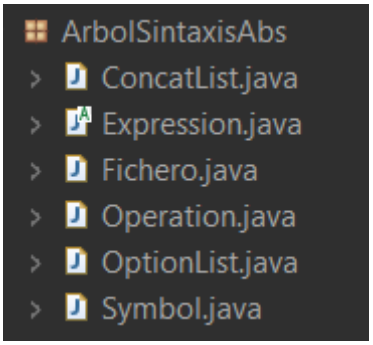
```
private ConcatList parseOption() throws SintaxException {
    int[] expected = { SYMBOL, LPAREN };
    ConcatList s = null;
    switch (nextToken.getKind()) {
    case SYMBOL:
    case LPAREN:
        Expression base_s = tryBase();
        ConcatList BasePos_s = tryBasePositive();
        BasePos_s.concat(base_s);
        s = BasePos_s;
        break;
    default:
        throw new SintaxException(nextToken, expected);
    }
    return s;
}
```

El reconocimiento de los tokens se representa mediante llamadas al método *match()*, que verifica que el token de preanálisis corresponde al token esperado y lo consume solicitando al analizador léxico el siguiente token de entrada.

Para realizar el análisis semántico ha sido necesario modificar los métodos asociados a cada símbolo para añadir las verificaciones y acciones semánticas y para devolver los datos correspondientes.

Árbol de sintaxis abstracta

Las clases utilizadas para desarrollar esta estructura de datos son:



- **`Expression`**: Se trata de una clase abstracta que se utiliza como superclase de todas las clases que describen expresiones.

- **`ConcatList`**: Es la clase que describe una concatenación de expresiones " a b c "

```java
public class ConcatList extends Expression {

    private ArrayList<Expression> list;

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

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

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

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

}
```

- **`OptionList`**: Es la clase que describe una lista de opciones " a | b | c ".

```java
public class OptionList extends Expression {

    private ArrayList<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 ArrayList<Expression> getList() {
        return list;
    }
}
```

- **`Symbol`**: Clase que describe un símbolo del lenguaje.

```java
public class Symbol extends Expression{
    private char symbol;

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

    public char getSymbol() {
        return symbol;
    }

    public void setSymbol(char symbol) {
        this.symbol = symbol;
    }
}
```

- **`Operation`**: Clase que describe una operación de clausura " (a)* ", clausura positiva " (a)+ ", o una opcionalidad " (a)? "

```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) {
        this.operator = op;
        this.operand = exp;
    }

    public int getOperator() {
        return operator;
    }

    public Expression getOperand() {
        return operand;
    }

    public static String operator(int i) {
        switch (i) {
        case 1: return "STAR";
        case 2: return "PLUS";
        case 3: return "HOOK";
        default:
            return "";
        }
    }
```

- **`Fichero`**: Es la clase que contiene toda la información recopilada de un archivo fuente. Contiene un id (nombre de la expresión) y la expresión regular.

```java
public class Fichero {

    private String id;
    private Expression exp;

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

    public String getId() {
        return id;
    }

    public Expression getExp() {
        return exp;
    }

}
```

Cálculo del AFD

Para describir el AFD hemos desarrollado las siguientes clases:

- **Estado**. Representa cada estado en el AFD. Tiene un atributo id que corresponde al número del estado y un atributo booleano que indica si es un estado final o no.

```java
public class Estado {

    private int id;
    private boolean esFinal;

    public Estado(int id, boolean esFinal) {
        this.id = id;
        this.esFinal = esFinal;
    }
    public int getId() {
        return id;
    }
    public boolean isFinal() {
        return esFinal;
    }
}
```

- **Transición**. Representa una transición en el AFD. Contiene dos atributos: símbolo, que es el carácter que provoca la transición, y destino, que es el identificador del estado al que se transita.

```java
public class Transicion {

    private char simbolo;
    private int destino;

    public Transicion(char simbolo, int destino) {
        this.simbolo = simbolo;
        this.destino = destino;
    }
    public char getSimbolo() {
        return simbolo;
    }
    public int getDestino() {
        return destino;
    }


}
```

- **Fila**. Representa una fila de la tabla de transiciones. Contiene los siguientes atributos:
    - estado: un objeto de la clase Estado que indica el estado actual de la fila.
    - posPuntos: una lista de enteros que representa las posiciones de los puntos en la expresión regular.
    - transiciones: una lista de objetos de la clase Transicion que contiene las transiciones desde el estado actual a otros estados.

```java
public class Fila {

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

    public Fila(Estado estado, ArrayList<Integer> posPuntos, ArrayList<Transicion> transiciones) {
        this.estado = estado;
        this.posPuntos = posPuntos;
        this.transiciones = transiciones;
    }

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

    public Estado getEstado() {
        return estado;
    }

    public ArrayList<Integer> getPosPuntos() {
        return posPuntos;
    }

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

La clase Main es el punto de entrada de la aplicación. Esta clase se encarga de leer una expresión regular desde un archivo, procesarla para generar AFD, y luego generar el fichero ".java".

```
public class Main {

    static int PosPuntoFinal;

    public static void main(String[] args) throws IOException {

        File file = new File("src/prueba.txt");
        FileInputStream fis = new FileInputStream(file);
        AFDParser parser = new AFDParser(fis);
        Fichero f = parser.parse();
        Expression e = f.getExp();
        GenerarFichero gf = new GenerarFichero(f.getId());

        ArrayList<Character> listaSymbol = new ArrayList<>();
        buscarSymbol(e, listaSymbol);

        ArrayList<Integer> posPuntos = new ArrayList<Integer>();
        AtomicInteger SymbolVisitado = new AtomicInteger(-1);
        AlgoritmoER_AFD(e, -1, posPuntos, 1, SymbolVisitado);

        PosPuntoFinal = listaSymbol.size();

        Estado inicial = new Estado(0, posPuntos.contains(PosPuntoFinal));
        ArrayList<Transicion> transicionesE0 = new ArrayList<>();
        Fila f0 = new Fila(inicial, posPuntos, transicionesE0);

        ArrayList<Fila> conjuntoFilas = new ArrayList<Fila>();
        conjuntoFilas.add(f0);
```

```
    int contadorFilas = 0;
    while (contadorFilas < conjuntoFilas.size()) {

        Fila fila = conjuntoFilas.get(contadorFilas);
        generaTransiciones(e, fila, conjuntoFilas, listaSymbol);
        contadorFilas++;
    }

    for (Fila F : conjuntoFilas) {
        System.out.print("Estado: " + F.getEstado().getId());
        if (F.getEstado().isFinal())
            System.out.print("*");
        for (Transicion trans : F.getTransiciones()) {
            System.out.print("\t" + trans.getSimbolo() + "-->" + trans.getDestino() + "    ");
        }
        System.out.println();
    }

    gf.generar(conjuntoFilas);

}
```

En primer lugar, leemos el fichero de entrada y parseamos el contenido de este para obtener la expresión regular a analizar. Luego llamamos a la función buscarSimbolos para obtener un array con todos los símbolos en la expresión regular.

```java
private static void buscarSimbolos(Expression e, ArrayList<Character> listaSymbol) {

    if (e instanceof ConcatList) {

        for (Expression exp : ((ConcatList) e).getList()) {
            buscarSimbolos(exp, listaSymbol);
        }

    } else if (e instanceof OptionList) {

        for (Expression exp : ((OptionList) e).getList()) {
            buscarSimbolos(exp, listaSymbol);
        }

    } else if (e instanceof Operation) {

        buscarSimbolos(((Operation) e).getOperand(), listaSymbol);

    } else if (e instanceof Symbol) {

        listaSymbol.add(((Symbol) e).getSymbol());
    }

}
```

A continuación, llamamos al método AlgoritmoER_AFD que aplica el algoritmo visto en la asignatura para puntear. Procesa diferentes tipos de subexpresiones, ajustando el punteado según las reglas y acumulando las posiciones de los puntos en una lista. Es una función recursiva que vamos controlando a partir de unas señales definidas.

```java
// Tipo punteada:

static final int SEGUIR_PUNTEANDO = 1;
static final int BUSCAR_PUNTEADO = 2;
static final int CONCATLIST_COMPLETO = 3;
static final int NO_PUNTEAR_MAS = -1;

private static void AlgoritmoER_AFD(Expression e, int posPuntoIni, ArrayList<Integer> posPuntos, int tipoPunteada, AtomicInteger SymbolVisitado) {

    if (e instanceof OptionList) {
        Expression exp = ((OptionList) e).getList().get(0);
        AlgoritmoER_AFD(exp, posPuntoIni, posPuntos, tipoPunteada, SymbolVisitado);

    } else if (e instanceof ConcatList) {
        int puntear = tipoPunteada;
        int i = 1;
        for (Expression exp : ((ConcatList) e).getList()) {
            if (i == ((ConcatList) e).getList().size()) {
                puntear = Puntear(exp, posPuntoIni, posPuntos, puntear, SymbolVisitado);

                if (puntear == CONCATLIST_COMPLETO)
                    puntear = SEGUIR_PUNTEANDO;
                if (puntear == SEGUIR_PUNTEANDO)
                    posPuntos.add(PosPuntoFinal);

            } else {
                puntear = Puntear(exp, posPuntoIni, posPuntos, puntear, SymbolVisitado);
                if (puntear == NO_PUNTEAR_MAS)
                    break;
                if (puntear == CONCATLIST_COMPLETO)
                    puntear = SEGUIR_PUNTEANDO;
            }
            i++;
        }
    }

}
```

Dichas señales las utilizamos con una función auxiliar Puntear:

```
private static int Puntear(Expression e, int posPuntoIni, ArrayList<Integer> posPuntos, int tipoPunteada, AtomicInteger SymbolVisitado) {

    if (e instanceof ConcatList) {
        if (tipoPunteada == SEGUIR_PUNTEANDO) {
            int puntear = tipoPunteada;
            int i = 1;
            for (Expression exp : ((ConcatList) e).getList()) {
                puntear = Puntear(exp, posPuntoIni, posPuntos, puntear, SymbolVisitado);
                if (puntear == NO_PUNTEAR_MAS) {
                    SymbolVisitado.set(SymbolVisitado.get() + (((ConcatList) e).getList().size() - i));
                    break;
                }
                i++;
            }

            return NO_PUNTEAR_MAS;
        } else if (tipoPunteada == BUSCAR_PUNTEADO) {

            int puntear = tipoPunteada;
            int i = 1;
            for (Expression exp : ((ConcatList) e).getList()) {
                if (i == ((ConcatList) e).getList().size()) {
                    puntear = Puntear(exp, posPuntoIni, posPuntos, puntear, SymbolVisitado);
                    if (puntear == SEGUIR_PUNTEANDO)
                        return CONCATLIST_COMPLETO;
                } else {
                    puntear = Puntear(exp, posPuntoIni, posPuntos, puntear, SymbolVisitado);
                    if (puntear == NO_PUNTEAR_MAS) {
                        SymbolVisitado.set(SymbolVisitado.get() + (((ConcatList) e).getList().size() - i));
                        return NO_PUNTEAR_MAS;
                    }
                }
                i++;
            }

            return puntear;
        }
```

```
} else if (e instanceof OptionList) {

    if (tipoPunteada == SEGUIR_PUNTEANDO) {
        for (Expression exp : ((OptionList) e).getList())
            Puntear(exp, posPuntoIni, posPuntos, tipoPunteada, SymbolVisitado);

        return NO_PUNTEAR_MAS;
    } else if (tipoPunteada == BUSCAR_PUNTEADO) {
        int puntear = tipoPunteada;
        for (Expression exp : ((OptionList) e).getList()) {
            puntear = Puntear(exp, posPuntoIni, posPuntos, puntear, SymbolVisitado);
            if (puntear == NO_PUNTEAR_MAS)
                return NO_PUNTEAR_MAS;
            if (puntear == CONCATLIST_COMPLETO)
                return CONCATLIST_COMPLETO;

        }
        return puntear;
    }

} else if (e instanceof Operation) {

    int tipo = ((Operation) e).getOperator();
    Expression exp = ((Operation) e).getOperand();

    if (tipoPunteada == SEGUIR_PUNTEANDO) {
        if (Operation.operator(tipo) == "STAR") {
            Puntear(exp, posPuntoIni, posPuntos, tipoPunteada, SymbolVisitado);
            return SEGUIR_PUNTEANDO;
        }
        if (Operation.operator(tipo) == "PLUS") {
            Puntear(exp, posPuntoIni, posPuntos, tipoPunteada, SymbolVisitado);
            return NO_PUNTEAR_MAS;
        }
```

```
        } else if (tipoPunteada == BUSCAR_PUNTEADO) {
            if (Operation.operator(tipo) == "STAR" || Operation.operator(tipo) == "PLUS") {
                AtomicInteger aux = new AtomicInteger(SymbolVisitado.get());
                int puntear = Puntear(exp, posPuntoIni, posPuntos, tipoPunteada, SymbolVisitado);
                if (puntear == CONCATLIST_COMPLETO) {
                    Puntear(exp, posPuntoIni, posPuntos, SEGUIR_PUNTEANDO, aux);
                    SymbolVisitado.set(aux.get());
                    return SEGUIR_PUNTEANDO;
                }
                if (puntear != BUSCAR_PUNTEADO)
                    return NO_PUNTEAR_MAS;
                else
                    return puntear;
            }
        }

    } else if (e instanceof Symbol) {

        SymbolVisitado.incrementAndGet();

        if (tipoPunteada == BUSCAR_PUNTEADO) {
            if (SymbolVisitado.get() < posPuntoIni) {
                return BUSCAR_PUNTEADO;
            }
            if (SymbolVisitado.get() == posPuntoIni) {
                return SEGUIR_PUNTEANDO;
            }
        }
        if (tipoPunteada == SEGUIR_PUNTEANDO) {
            posPuntos.add(SymbolVisitado.get());
            return NO_PUNTEAR_MAS;
        }
    }

    return 0;
}
```

Tras la ejecución de AlgoritmoER_AFD, obtenemos las posiciones de los puntos del estado 0, por tanto, ya somos capaces de generar el estado inicial. A partir de este estado, desarrollamos la tabla de transiciones con la función generarTransiciones.

Dicha función utiliza una lógica similar a la anterior y se va aplicando por cada fila generada.

```java
private static void generaTransiciones(Expression e, Fila fila, ArrayList<Fila> conjuntoFilas,
        ArrayList<Character> listaSymbol) {

    ArrayList<Integer> posPuntos = fila.getPosPuntos();
    ArrayList<ArrayList<Integer>> posPuntos_de_cada_posPunto = new ArrayList<ArrayList<Integer>>();

    for (Integer pos : posPuntos) {
        if (pos == PosPuntoFinal)
            continue;
        ArrayList<Integer> posP = new ArrayList<Integer>();
        AtomicInteger SymbolVisitado = new AtomicInteger(-1);
        AlgoritmoER_AFD(e, pos, posP, 2, SymbolVisitado);

        posPuntos_de_cada_posPunto.add(posP);
    }

    if (posPuntos_de_cada_posPunto.isEmpty())
        return;

    ArrayList<Character> ListaSymbolTransiciones = ListaSymbolPosPuntos(listaSymbol, posPuntos);

    if (ListaSymbolTransiciones.size() > 1) {

        ArrayList<Character> ListaSymbolUnique = CharUnicos(ListaSymbolTransiciones);
        ArrayList<ArrayList<Object>> PosSymbol = new ArrayList<ArrayList<Object>>();

        for (int i = 0; i < ListaSymbolUnique.size(); i++) {
            PosSymbol.add(new ArrayList<Object>());
            int index = PosSymbol.size() - 1;
            PosSymbol.get(index).add(ListaSymbolUnique.get(i));
            for (int j = 0; j < ListaSymbolTransiciones.size(); j++) {
                Character c = ListaSymbolTransiciones.get(j);
                if (c == ListaSymbolUnique.get(i)) {
                    PosSymbol.get(index).add(j);
                }
            }
            if (PosSymbol.get(index).size() == 2) {
                PosSymbol.remove(index);
            }
        }
```

```java
    for (int i = 0; i < ListaSymbolTransiciones.size(); i++) {
        Character c = ListaSymbolTransiciones.get(i);
        ArrayList<Object> listaRepetido = comprobarRepetido(PosSymbol, c);
        ArrayList<Integer> ListaElementos = null;
        if (listaRepetido != null) {
            if (listaRepetido.size() > 1) {
                int posLista1 = (Integer) listaRepetido.get(1);
                for (int j = 2; j < listaRepetido.size(); j++) {
                    int posLista2 = (Integer) listaRepetido.get(j);
                    posPuntos_de_cada_posPunto.get(posLista1).addAll(posPuntos_de_cada_posPunto.get(posLista2));
                }
                listaRepetido.clear();
                listaRepetido.add(c);
                ListaElementos = IntUnicos(posPuntos_de_cada_posPunto.get(posLista1));
            } else
                continue;
        } else {
            ListaElementos = posPuntos_de_cada_posPunto.get(i);
        }

        int valorTransicion = perteneceEstado(ListaElementos, conjuntoFilas);
        Transicion Trans = new Transicion(c, valorTransicion);
        fila.anadirTransicion(Trans);

        if (valorTransicion == conjuntoFilas.size()) {
            Estado estado = new Estado(valorTransicion, ListaElementos.contains(PosPuntoFinal));
            ArrayList<Transicion> transicionesE = new ArrayList<>();
            Fila F = new Fila(estado, ListaElementos, transicionesE);
            conjuntoFilas.add(F);
        }

    }
} else {
```

```
        ArrayList<Integer> ListaElementos = posPuntos_de_cada_posPunto.get(0);
        Character c = ListaSymbolTransiciones.get(0);

        int valorTransicion = perteneceEstado(ListaElementos, conjuntoFilas);
        Transicion Trans = new Transicion(c, valorTransicion);
        fila.anadirTransicion(Trans);

        if (valorTransicion == conjuntoFilas.size()) {
            Estado estado = new Estado(valorTransicion, ListaElementos.contains(PosPuntoFinal));
            ArrayList<Transicion> transicionesE = new ArrayList<>();
            Fila F = new Fila(estado, ListaElementos, transicionesE);
            conjuntoFilas.add(F);
        }
    }

}
```

Métodos Auxiliares

- perteneceEstado: Verifica si un conjunto de posiciones ya pertenece a algún estado.
- posPuntosIguales: Compara dos listas de posiciones para ver si son iguales.
- comprobarRepetido: Comprueba si un símbolo se repite en una lista.
- CharUnicos y IntUnicos: Eliminan duplicados en listas de caracteres y enteros, respectivamente.
- ListaSymbolPosPuntos: Genera una lista de símbolos correspondientes a las posiciones de puntos dados.

Finalmente, nos devuelve un array con todas las filas de la tabla, las cuales contienen los estados con sus correspondientes transiciones.

Generar fichero “.java”

Una vez calculado el AFD, se ha creado la clase `GeneradorFichero` para el generar el código que cree el fichero ".java":

```java
public class GenerarFichero {

    private String id;

    public GenerarFichero(String id) {
        this.id = id;
    }

    public void generar(ArrayList<Fila> conjuntoFilas) {
        String contenido = "public class " + id + "{\n\n" + "\tpublic int transition(int state, char symbol) {\n"
                + " \t\tswitch(state) {\n";

        for (Fila f : conjuntoFilas) {
            contenido += "\t\t\tcase " + f.getEstado().getId() + ": \n";
            for (Transicion trans : f.getTransiciones()) {
                contenido += "\t\t\t\tif(symbol == '" + trans.getSimbolo() + "') return " + trans.getDestino() + ";\n";
            }
            contenido += "\t\t\t\treturn -1; \n";
        }

        contenido += "\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 (Fila f : conjuntoFilas) {
            contenido += "\t\t\tcase " + f.getEstado().getId() + ": return " + f.getEstado().isFinal() + ";\n";
        }
        contenido += "\t\t\tdefault: return false;\n" + "\t\t}\n" + "\t}\n" + "}";

        try (BufferedWriter writer = new BufferedWriter(new FileWriter(id+".java"))) {
            writer.write(contenido);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

Se trata de ir rellenando una cadena String con el código que describe el AFD. Para ello, recibe un id, que será el nombre de la clase, y un array con la información de los estados y sus transiciones. Como resultado nos genera un fichero como el siguiente:

```
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;
        }
    }
}
```

Pruebas de funcionamiento

Para probar el algoritmo hemos creado 3 ficheros de prueba:

- `prueba1.txt`

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

Estado	Elementos	Transiciones
0	· b a (((a)* o | b))* (a)+ b	b → 1
1	b · a (((a)* o | b))* (a)+ b	a → 2
2	b a ((· a)* o | b))* (a)+ b b a (((a)* · o | b))* (a)+ b b a (((a)* o | · b))* (a)+ b b a (((a)* o | b))* (· a)+ b	a → 3 o → 2 b → 2
3	b a ((· a)* o | b))* (a)+ b b a (((a)* · o | b))* (a)+ b b a (((a)* o | b))* (· a)+ b b a (((a)* o | b))* (a)+ · b	a → 3 o → 2 b → 4
4	b a (((a)* o | b))* (a)+ b ·	

Salida por consola:

```
Comment ::= b a ((a )STAR o  | b )STAR (a )PLUS b
Estado: 0       b-->1
Estado: 1       a-->2
Estado: 2       a-->3           o-->2           b-->2
Estado: 3       a-->3           o-->2           b-->4
Estado: 4*
```

Además genera el fichero `Comment.java` mostrado anteriormente.

- `prueba2.txt`

```
PruebaDos ::= 'l' 'l' ( 'o' | 'g' 'o' )* 'g' 'g' ;
```

Estado	Elementos	Transiciones
0	· l l (o | g o)* g g	l → 1
1	l · l (o | g o)* g g	l → 2
2	l l (· o | g o)* g g l l (o | · g o)* g g l l (o | g o)* · g g	o → 2 g → 3
3	l l (o | g · o)* g g l l (o | g o)* g · g	o → 2 g → 4
* 4	l l (o | g o)* g g ·	

Salida por consola:

```
PruebaDos ::= l l (o  | g o )STAR g g
Estado: 0       l-->1
Estado: 1       l-->2
Estado: 2       o-->2          g-->3
Estado: 3       o-->2          g-->4
Estado: 4*
```

Fichero `PruebaDos.java` generado:

```java
public class PruebaDos{

    public int transition(int state, char symbol) {
        switch(state) {
            case 0:
                if(symbol == 'l') return 1;
                return -1;
            case 1:
                if(symbol == 'l') return 2;
                return -1;
            case 2:
                if(symbol == 'o') return 2;
                if(symbol == 'g') return 3;
                return -1;
            case 3:
                if(symbol == 'o') return 2;
                if(symbol == 'g') 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;
        }
    }
}
```

- `prueba3.txt`

```
PruebaTres ::= ('b' ('a' 'b')*) ('o' 'b' ('a' 'b')*)* ;
```

Estado	Elementos	Transiciones
0	(· b (a b)*) (o b (a b)*)*	b → 1
1 *	(b (· a b)*) (o b (a b)*)* (b (a b)*) (· o b (a b)*)* (b (a b)*) (o b (a b)*)* ·	a → 2 o → 3
2	(b (a · b)*) (o b (a b)*)*	b → 1
3	(b (a b)*) (o · b (a b)*)*	b → 4
4 *	(b (a b)*) (o b (· a b)*)* (b (a b)*) (· o b (a b)*)* (b (a b)*) (o b (a b)*)* ·	a → 5 o → 3
5	(b (a b)*) (o b (a · b)*)*	b → 4

Salida por consola:

```
PruebaTres ::= b (a b )STAR  (o b (a b )STAR )STAR
Estado: 0       b-->1
Estado: 1*      a-->2           o-->3
Estado: 2       b-->1
Estado: 3       b-->4
Estado: 4*      o-->3           a-->5
Estado: 5       b-->4
```

Fichero `PruebaTres.java` generado:

```java
public class PruebaTres{

    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;
                if(symbol == 'o') return 3;
                return -1;
            case 2:
                if(symbol == 'b') return 1;
                return -1;
            case 3:
                if(symbol == 'b') return 4;
                return -1;
            case 4:
                if(symbol == 'o') return 3;
                if(symbol == 'a') return 5;
                return -1;
            case 5:
                if(symbol == 'b') return 4;
                return -1;
            default:
                return -1;
        }
    }

    public boolean isFinal(int state) {
        switch(state) {
            case 0: return false;
            case 1: return true;
            case 2: return false;
            case 3: return false;
            case 4: return true;
            case 5: return false;
            default: return false;
        }
    }
```