ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA

PROCESADORES DE LENGUAJE

GENERADOR DE ANALIZADORES SINTÁCTICOS ASCENDENTES

Juan A. Contreras Fernández

Capítulo 1: Analizador Léxico.

1.1 Categorías léxicas:

A partir de la expresión regular de cada uno de los tokens que podremos reconocer crearemos el AFD de cada uno de ellos.

1.1.1 Carácter en Blanco:

Estado 1	(." "|"\r"|"\n"|"\t") (" "|."\r"|"\n"|"\t") (" "|"\r"|."\n"|"\t") (" "|"\r"|"\n"|."\t")	
Estado 1(" ")	(" "|"\r"|"\n"|"\t").	Estado 2*
Estado 1("\")	(" "|"\.r"|"\n"|"\t") (" "|"\r"|"\.n"|"\t") (" "|"\r"|"\n"|"\.t")	Estado 3
Estado 3("r")	(" "|"\r"|"\n"|"\t").	Estado 2*
Estado 3("n")	(" "|"\r"|"\n"|"\t").	Estado 2*
Estado 3("t")	(" "|"\r"|"\n"|"\t").	Estado 2*

1.1.2 Comentario:

Estado 1	"./*" (("*")* ~["*", "/"] | "/")* ("*")+ "/"	
Estado 1("/")	"/.*" (("*")* ~["*", "/"] | "/")* ("*")+ "/"	➔ Estado 2
Estado 2("*")	"/*" (.("*")* ~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* .~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | ."/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | "/")* .("*")+ "/"	➔ Estado 3
Estado 3("/")	"/*" (.("*")* ~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* .~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | ."/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | "/")* .("*")+ "/"	➔ Estado 3
Estado 3("*")	"/*" (**.**("*")* ~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* **.**~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | **.**"/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | "/")* **.**("*")+ "/" "/*" (("*")* ~["*", "/"] | "/")* ("*")+ **.**"/"	➔ Estado 4
Estado 3("~["*", "/"]")	"/*" (.("*")* ~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* .~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | ."/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | "/")* .("*")+ "/"	➔ Estado 3
Estado 4("/")	"/*" (.("*")* ~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* **.**~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] |. "/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | "/")* .("*")+ "/" "/*" (("*")* ~["*", "/"] | "/")* ("*")+ "/".	➔ Estado 5*
Estado 4("*")	"/*" (.("*")* ~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* **.**~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | ."/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | "/")* .("*")+ "/" "/*" (("*")* ~["*", "/"] | "/")* ("*")+ ."/"	➔ Estado 4
Estado 4(~["*", "/"])	"/*" (.("*")* ~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | ."/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | "/")* .("*")+ "/"	➔ Estado 3
Estado 5("*")	"/*" (.("*")* ~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | ."/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | "/")* .("*")+ "/"	➔ Estado 3
Estado 5("/")	"/*" (.("*")* ~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | ."/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | "/")* .("*")+ "/"	➔ Estado 3

Estado 5(~["*", "/"])	"/*" (.("*")* ~["*", "/"] | "/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | ."/")* ("*")+ "/" "/*" (("*")* ~["*", "/"] | "/")* .("*")+ "/"	➔ Estado 3

1.1.3 NOTERMINAL:

E1	.["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])*	
E1()	["_","a"-"z","A"-"z"] .(["_","a"-"z","A"-"Z","0"-"9"])* ["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])*.	➔ Estado 2*
E2()	["_","a"-"z","A"-"z"] .(["_","a"-"z","A"-"Z","0"-"9"])* ["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])*.	➔ Estado 2*

1.1.4 TERMINAL:

E1	".<" ["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])* ">"	
E1("<")	"<" .["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])* ">"	➔ E2
E2("nn")	"<" ["_","a"-"z","A"-"z"] .(["_","a"-"z","A"-"Z","0"-"9"])* ">" "<" ["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])* .">	➔ E3
E3("*")	"<" ["_","a"-"z","A"-"z"] .(["_","a"-"z","A"-"Z","0"-"9"])* ">" "<" ["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])* .">	➔ E3
E3(">")	"<" ["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])* ">".	➔ E4

1.1.5 EQ:

E1:	."::="	
E1(":")	":.:="	➔ E2
E2(":")	"::.="	➔ E3
E3("=")	":==".	➔ E4*

1.1.6 BAR:

E1	."|"	
E1("|")	"|".	➔ E2*

1.1.7 SEMICOLON:

E1	.";"	
E1(";")	";".	➔ E2*

1.2 Maquina discriminadora determinista:

A partir de los autómatas y expresiones que hemos desarrollado crearemos el autómata que reconocerá cualquier token de la gramática que estamos trabajando.

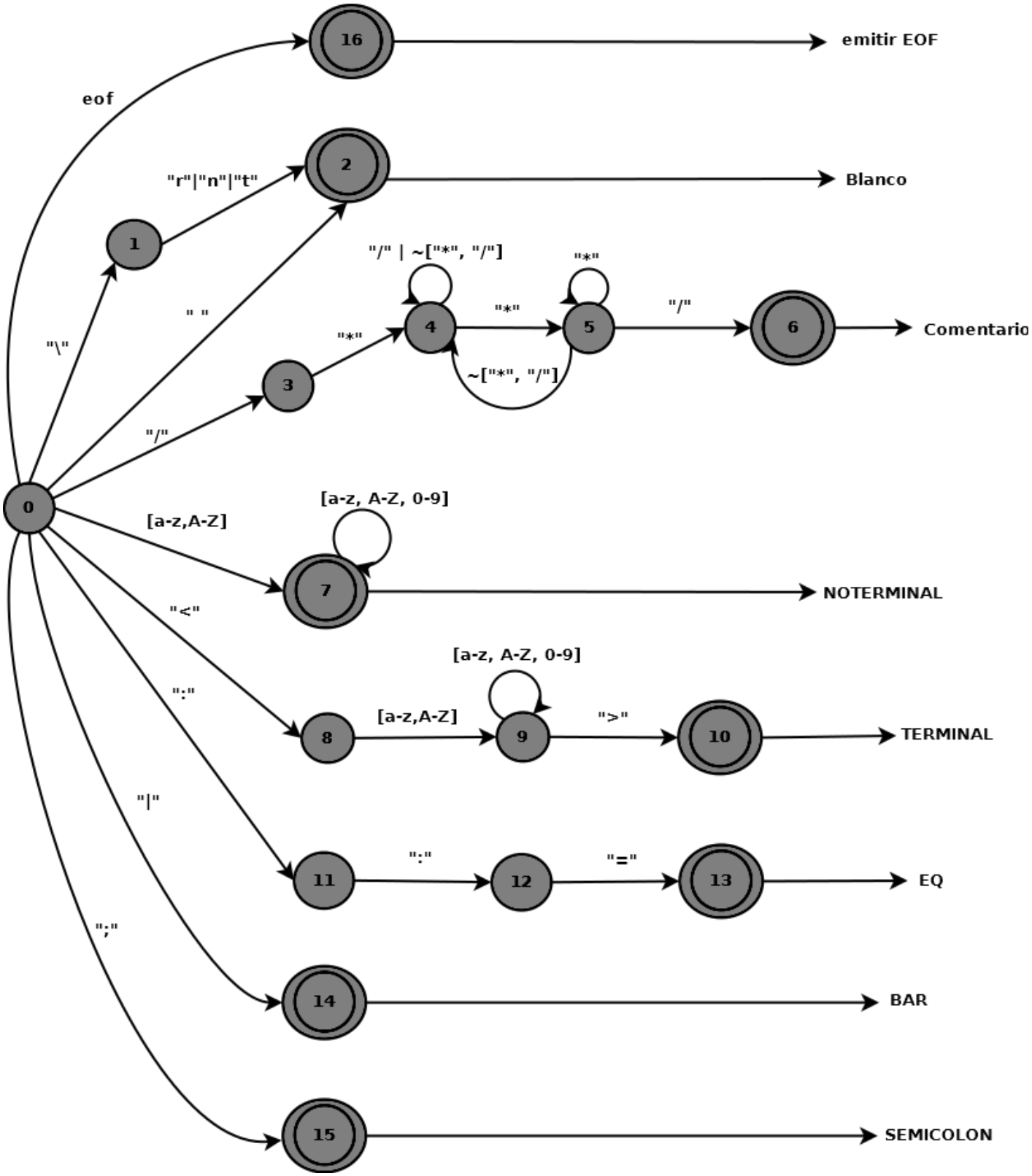
Paso 1: Ordenar las diferentes expresiones regulares.

Especificación	Expresión regular
blanco	(" " | "\r" | "\n" | "\t")
comentario	"/*" (("*")* ~["*", "/"] | "/")* ("*")+ "/"
NOTERMINAL	["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])*
TERMIINAL	"<" ["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])* ">"
EQ	"::="
BAR	"|"
SEMICOLON	";"

Paso 2: Unir todas las expresiones regulares.

Expresión regular
(" " / "\r" / "\n" / "\t") ┃ *(" " / "\r" / "\n" / "\t")* ┃ *["_","a"-"z","A"-"z"]* (["_","a"-"z","A"-"Z","0"-"9"])* ┃ "<" ["_","a"-"z","A"-"z"] (["_","a"-"z","A"-"Z","0"-"9"])* ">" ┃ "::=" ┃ "|" ┃ ";" ┃ eof

Paso 3: Autómata resultante.



1.3 Código de la clase que desarrolla el analizador léxico.

```
package Analizador_Lexico;

/**
 * Autor Francisco Jose Moreno Velo.
 * Modificacion: Juan Antonio Contreras Fernández.
 * Curso: 2015/2016.
 * Asignatura: Procesadores de lenguaje.
 *
 * Fichero: JLexer.java
 * Descripcion: clase principal del lexer implementado para el trabajo final.
 */


import java.io.*;

/**
 * Clase que implementa el analizador lexico de la practica final
 *
 * @author Juan A. Contreras Fernández
 *
 */
public class JLexer extends Lexer implements TokenConstants {

        /**
         * Transiciones del automata del analizador lexico
         *
         * @param state
         * @param symbol
         * @return Estado
         */
        protected int transition(int state, char symbol) {
                switch(state) {
                        case 0:
                                  if(symbol == ' ' || symbol == '\n') return 2;
                                  else if(symbol == '\t' || symbol == '\r') return 2;
                                  else if(symbol == '/') return 3;
                                else if(symbol >= 'a' && symbol <= 'z') return 7;
                                else if(symbol >= 'A' && symbol <= 'Z') return 7;
                                else if(symbol == '<') return 8;
                                else if(symbol == ':') return 11;
                                else if(symbol == '|') return 14;
                                else if(symbol == ';') return 15;
                        case 3:
                                if(symbol == '*') return 4;
                                  else return -1;
                          case 4:
                                  if(symbol == '*') return 5;
                                  else
                                      return 4;
                        case 5:
                                if(symbol == '/') return 6;
                                else if(symbol == '*') return 5;
                                  else
                                      return 4;
                        case 7:
                                if(symbol >= 'a' && symbol <= 'z') return 7;
                                else if(symbol >= 'A' && symbol <= 'Z') return 7;
                        case 8:
                                if(symbol >= 'a' && symbol <= 'z') return 9;
                                else if(symbol >= 'A' && symbol <= 'Z') return 9;
                                else return -1;
                        case 9:
                                  if(symbol >= 'a' && symbol <= 'z') return 9;
                                else if(symbol >= 'A' && symbol <= 'Z') return 9;
                                  else if(symbol >= '0' && symbol <= '9') return 9;
                                  else if(symbol == '>') return 10;
                                else return -1;
                        case 11:
                                if(symbol == ':') return 12;
                                else return -1;
                        case 12:
                                if(symbol >= '=') return 13;
                                else return -1;
```

```
                case 15:
                        if(symbol == '\'') return 16;
                        else return -1;
                  default:
                      return -1;
        }
}

/**
 * Verifica si un estado es final
 *
 * @param state Estado
 * @return true, si el estado es final
 */
protected boolean isFinal(int state) {
        if(state <=0 || state > 42) return false;
        switch(state) {
                case 0:
                case 1:
                case 3:
                case 4:
                case 5:
                case 8:
                case 9:
                case 11:
                case 12:
                        return false;
                default:
                        return true;
        }
}

/**
 * Genera el componente lexico correspondiente al estado final y
 * al lexema encontrado. Devuelve null si la accion asociada al
 * estado final es omitir (SKIP).
 *
 * @param state Estado final alcanzado
 * @param lexeme Lexema reconocido
 * @param row Fila de comienzo del lexema
 * @param column Columna de comienzo del lexema
 * @return Componente lexico correspondiente al estado final y al lexema
 */
protected Token getToken(int state, String lexeme, int row, int column) {
        switch(state) {
                case 2: return null;
                case 6: return null;
                case 7: return new Token(NOTERMINAL,lexeme, row, column);
                case 10: return new Token(TERMINAL, lexeme, row, column);
                case 13: return new Token(EQ, lexeme, row, column);
                case 14: return new Token(BAR, lexeme, row, column);
                case 15: return new Token(SEMICOLON, lexeme, row, column);
                default: return null;
        }
}

/**
 * Constructor de la clase
 * @param filename Nombre del fichero fuente
 * @throws IOException En caso de problemas con el flujo de entrada
 */
public JLexer(String filename) throws IOException {
        super(filename);
}

/**
 * Punto de entrada para ejecutar pruebas del analizador lexico
 * @param args
 */
public static void main(String[] args) {
         //Si no hay archivo termina
        if(args.length == 0) return;
        try {
                JLexer lexer = new JLexer(args[0]);
                Token tk;
                do {
```

```
                        tk = lexer.getNextToken();
                        System.out.println(tk.toString());
                    } while(tk.getKind() != Token.EOF);
                } catch(Exception ex) {
                    ex.printStackTrace();
                }
        }
}
```

En el método transición de la clase podemos observar las transiciones del autómata que reconoce las expresiones y los tokens.

Para crear el analizador léxico hemos usado el código de la practica 2 y modificado los archivos necesarios para la creación del nuevo analizador léxico.

Capítulo 2: Analizador Sintáctico/Semántico.

2.1 Transformación de gramática EBNB a BNF:

Mediante a las transformaciones explicadas en clase se ha desarrollado la siguiente gramática que usaremos para el analizador sintáctico.

EBNF	BNF
Gramática → (Definición)*	Gramatica → GramaticaP GramaticaP → Definición Gramática GramaticaP → Lambda
Definición → NOTERMINAL EQ ListaReglas SEMICOLON	Definición → NOTERMINAL EQ ListaReglas SEMICOLON
ListaReglas → Regla (BAR Regla)*	ListaReglas → Regla ListaReglasP ListaReglasP → BAR Regla ListaReglasP ListaReglaP → Lambda
Regla → (NOTERMINAL | TERMINAL)*	Regla → Símbolo ReglaP ReglaP → Símbolo ReglaP → Lambda
	Símbolo → NOTERMINAL Símbolo → TERMINAL

2.2 Conjunto de predicción:

Desarrollo de los conjuntos de predicción.

Regla	Primeros	Siguientes	Prediccion
Gramática → GramaticaP	NOTERMINAL, lambda	$	NOTERMINAL, $
GramaticaP → Definición GramaticaP	NOTERIMINAL	$	NOTERMINAL
GramaticaP → Lambda	Lambda		$
Definición → NOTERMINAL EQ ListaReglas SEMICOLON	NOTERMINAL	NOTERMINAL, $	NOTERMINAL
ListaReglas → Regla ListaReglasP	NOTERMINAL, TERMINAL, Lambda	SEMICOLON	NOTERMINAL, TERMINAL, SEMICOLON
ListaReglasP → BAR Regla ListaReglas	BAR	SEMICOLON	BAR
ListaReglaP → Lambda	Lambda		SEMICOLON
Regla → ReglaP	NOTERMINAL, TERMINAL, Lambda	BAR, SEMICOLON	NOTERMINAL, TERMINAL, BAR, SEMICOLON
ReglaP → Símbolo ReglaP	NOTERMINAL, TERMINAL, lambda	BAR, SEMICOLON	BAR, SEMICOLON
ReglaP → Lambda	Lambda		NOTERMINAL, TERMINAL
Símbolo → NOTERMINAL	NOTERMINAL	NOTERMINAL, TERMINAL, BAR, SEMICOLON	NOTERMINAL
Símbolo → TERMINAL	TERMINAL		TERMINAL

2.3 Código del analizador sintáctico.

A partir del analizador sintáctico usado en la práctica de la asignatura se ha modificado para el reconocimiento de la gramática BNF creada y para reconocer los conjuntos de la gramática, así como la creación posterior del árbol de sintaxis abstracta.

A continuación se incluye el código que reconocerá de manera recursiva la gramática dada al programa mediante un archivo y con la estructura dada, cada uno de los métodos analiza un tipo de símbolo de la gramática, creándose una estructura con el árbol del sintaxis abstracta que comentaremos en el siguiente capítulo, todo el proceso comienza con la llamada al método **parseGramatica()** que inicia el reconocimiento de esta.

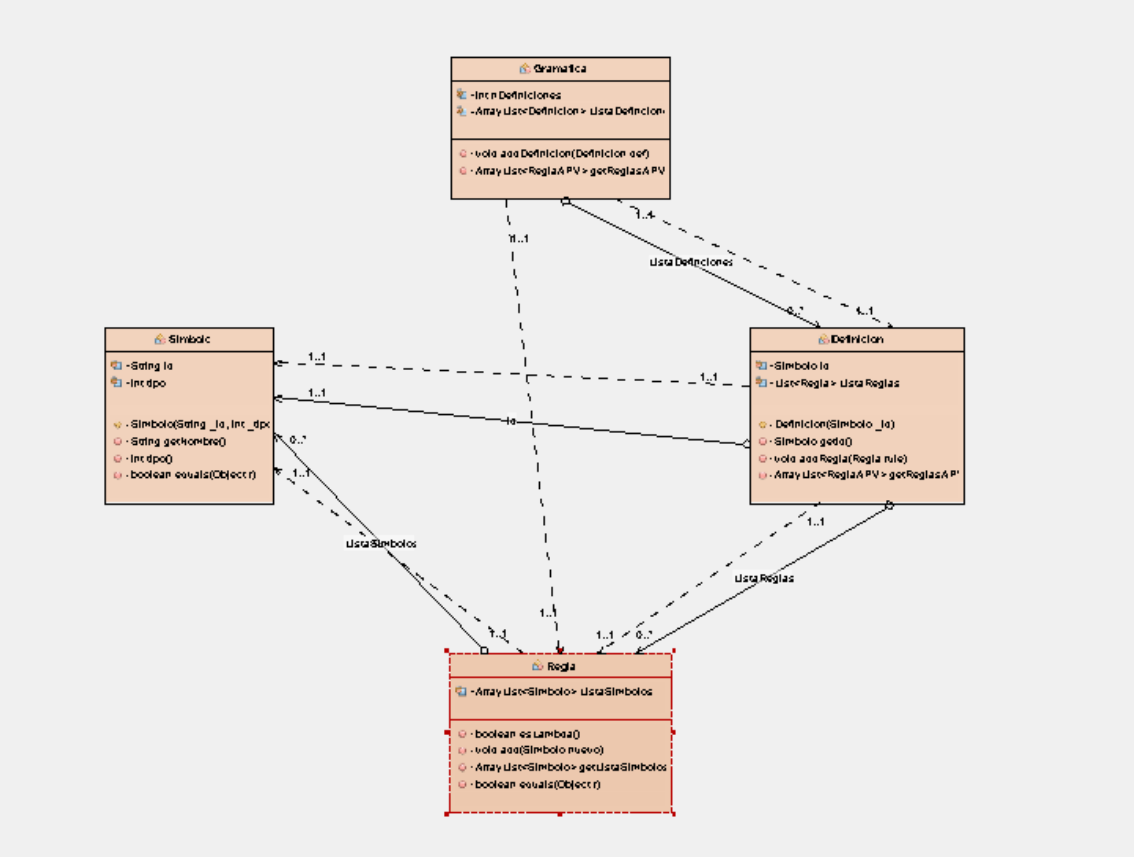
```
private Gramatica parseGramatica() throws SintaxException {
		int[] expected = { NOTERMINAL, EOF };
		 Gramatica g = new Gramatica();
		switch(nextToken.getKind()) {
			case NOTERMINAL://Simbolo inicial de gramatica
				parseGramaticaP(g);
				return g;
			case EOF://Gramatica vacia
				return g;
			default:
				throw new SintaxException(nextToken,expected);
		}
	}

	/**
	 * Analiza el simbolo <GramaticaP>
	 * @throws SintaxException
	 */
	private void parseGramaticaP(Gramatica g) throws SintaxException {
		int[] expected = { NOTERMINAL, EOF };
		switch(nextToken.getKind()) {
			case NOTERMINAL://Simbolo de definicion
				parseDefinicion(g);
				parseGramaticaP(g);//Siguiente simbolo no terminal
				break;
			case EOF://Gramatica terminada.
				break;
			default:
				throw new SintaxException(nextToken,expected);
		}
	}

	/**
	 * Analiza el simbolo <Definicion>
	 * @throws SintaxException
	 */
	private void parseDefinicion(Gramatica g) throws SintaxException {
		int[] expected = { NOTERMINAL };
		switch(nextToken.getKind()) {
			case NOTERMINAL:
				//Creamos una regla y la deficion que la contendra.
				Regla r = new Regla();
				Definicion d = new Definicion(new Simbolo(next-
Token.getLexeme(), 0));
				//Guardamos el simbolo Noterminal encontrado.
				Noterminales.remove(nextToken.getLexeme());
				Noterminales.add(nextToken.getLexeme());
				//Consumimos los caracteres
				match(NOTERMINAL);
				match(EQ);
				//Analizamos la lista de reglas de la definicion.
				parseListaReglas(d, r);
				match(SEMICOLON);
				g.addDefinicion(d);
				break;
			default:
```

```
                        throw new SintaxException(nextToken,expected);
                }
        }

        /**
         * Analiza el simbolo <ListaReglas>
         * @throws SintaxException
         */
        private void parseListaReglas(Definicion d, Regla r) throws SintaxException {
                int[] expected = { NOTERMINAL, TERMINAL, SEMICOLON };
                switch(nextToken.getKind()) {
                        case NOTERMINAL:
                          case TERMINAL:
                          case SEMICOLON:
                                  //Analizamos la regla
                                parseRegla(d, r);
                                  //Encadenamos con el resto de las reglas de la defini-
cion.
                                  parseListaReglasP(d, r);
                                break;
                        default:
                                throw new SintaxException(nextToken,expected);
                }
        }

        /**
         * Analiza el simbolo <ListaReglasP>
         * @throws SintaxException
         */
        private void parseListaReglasP(Definicion d, Regla r) throws SintaxException {
                int[] expected = { BAR, SEMICOLON};
                switch(nextToken.getKind()) {
                        case BAR:
                                  match(BAR);//Concatenacion de reglas
                                  d.addRegla(r);
                                  r = new Regla();
                                  parseRegla(d, r);
                                parseListaReglasP(d, r);
                                break;
                        case SEMICOLON://Final del conjunto de reglas de la definicion.
                                  d.addRegla(r);
                                  r = new Regla();
                                break;
                        default:
                                throw new SintaxException(nextToken,expected);
                }
        }

        /**
         * Analiza el simbolo <Regla>
         * @throws SintaxException
         */
        private void parseRegla(Definicion d, Regla r) throws SintaxException {
                int[] expected = { NOTERMINAL, TERMINAL, BAR, SEMICOLON };
                switch(nextToken.getKind()) {
                        case NOTERMINAL:
                        case TERMINAL:
                        case BAR:
                        case SEMICOLON:
                                parseReglaP(d, r);//Analizamos la regla
                                break;
                        default:
                                throw new SintaxException(nextToken,expected);
                }
        }

        /**
         * Analiza el simbolo <ReglaP>
         * @throws SintaxException
         */
        private void parseReglaP(Definicion d, Regla r) throws SintaxException {
                int[] expected = { NOTERMINAL, TERMINAL, BAR, SEMICOLON };
                switch(nextToken.getKind()) {
                        case NOTERMINAL:
                          case TERMINAL://Analizamos el simbolo encontrado y lo añádimos
                                parseSimbolo(d, r);
                                  parseReglaP(d, r);
```

```
                        break;
                    case BAR:
                    case SEMICOLON:
                            break;
                default:
                        throw new SintaxException(nextToken,expected);
        }
}

/**
 * Analiza el simbolo <Simbolo>
 * @throws SintaxException
 */
private void parseSimbolo(Definicion d, Regla r) throws SintaxException {
        int[] expected = { NOTERMINAL, TERMINAL };
        switch(nextToken.getKind()) {
                case NOTERMINAL://Añadimos simbolo noterminal a la regla
                        r.add(new Simbolo(nextToken.getLexeme(), 0));
                        match(NOTERMINAL);
                        break;
                case TERMINAL://Añadimos simbolo terminal a la regla.
                        String next = nextToken.getLexeme();
                        //Tratamiento para escritura de TokenConstants
                        next = next.replace('<', ' ');
                        next = next.replace('>', ' ');
                        Terminales.add(next);//Guardamos el simbolo
                        r.add(new Simbolo(next, 1));//Lo añadimos a la regla.
                        match(TERMINAL);
                        break;
                default:
                        throw new SintaxException(nextToken,expected);
        }
}
```

3.1 Diagrama UML.

Mediante un esquema UML podemos ver cómo están creadas y diseñadas las clases del árbol de sintaxis abstracta.



3.2 Código de las clases que componen el árbol de sintaxis abstracta.

Aquí podemos ver la implementación de la clase principal de la Gramática que englobara una lista de Definiciones, siendo definiciones otro de los objetos del árbol de sintaxis abstracta.

La gramática contara con una serie de atributos sintetizados que contendrá su lista de definiciones, a la hora de calcular la lista de reglas de la gramática para el autómata de prefijos viables el método **getReglasAPV ()** hace uso de los atributos sintetizados de las definiciones así como esta lo hará con las clases regla y símbolo, en ningún caso el árbol tiene atributos heredados o calculados a partir del conjunto superior del grafo.

```
public class Gramatica {

    private ArrayList<Definicion> ListaDefinciones = new ArrayList<Definicion>();

    /**
     * Añade una nueva definicion al conjunto de la gramatica.
     * @param def
     */
    public void addDefinicion(Definicion def)
    {
        ListaDefinciones.add(def);
    }

    /**
     * Devuelve una lista con todas las reglas preparadas para el automata
     * de prefijos viables.
     * @return
     */
    public ArrayList<ReglaAPV> getReglasAPV()
    {
        ArrayList<ReglaAPV> devolver = new ArrayList<ReglaAPV>();
        for(Definicion d: ListaDefinciones)
        {
            devolver.addAll(d.getReglasAPV());
        }
        return devolver;
    }

}
```

En siguiente código es el de la clase Definición que se identifica por un símbolo no terminal y que contiene a su vez un conjunto de reglas guardadas en forma de vector y cada regla a su vez tendrá un vector de símbolos terminales o no terminales.

```
public class Definicion {

    private final Simbolo id;
    private List<Regla> ListaReglas = new ArrayList<Regla>();

    /**
     * Constructor de la clase defincion id sera el simbolo inicial de
     * la definicion.
     *
     * @param _id
     */
    public Definicion(Simbolo _id)
    {
        id = _id;
    }

    /**
     * Devuelve el simbolo id de la definicion.
```

```
     * @return Simbolo
     */
    public Simbolo getid()
    {
        return id;
    }

    /**
     * Añade una nueva regla a la lista de la definicion.
     * @param rule
     */
    public void addRegla(Regla rule)
    {
        ListaReglas.add(rule);
    }

    /**
     * Devuelve el conjunto de reglas de la definicion completa en forma de reglas para
SLR.
     * @return ArrayList
     */
    public ArrayList<ReglaAPV> getReglasAPV()
    {
        ArrayList<ReglaAPV> Reglas = new ArrayList<>();
        for(Regla r:ListaReglas)
        {
            Reglas.add(new ReglaAPV(id, r.getListaSimbolos(), 0, 0));
        }
        return Reglas;
    }

}




public class Regla {

    //Conjunto de simbolos de la regla.
    private ArrayList<Simbolo> ListaSimbolos = new ArrayList<Simbolo>();

    /**
     * Permite saber si una regla es de tipo lambda.
     * @return
     */
    public boolean esLambda()
    {
        return ListaSimbolos.isEmpty();
    }

    /**
     * Añade un nuevo simbolo al conjunto de la regla.
     *
     * @param nuevo
     */
    public void add(Simbolo nuevo)
    {
        ListaSimbolos.add(nuevo);
    }

    /**
     * Devuelve la lista de simbolos de la regla.
     * @return ArrayList
     */
    public ArrayList<Simbolo> getListaSimbolos()
    {
        return ListaSimbolos;
    }

    /**
     * Sobrecarga del metodo equals para la comparacion de reglas entre si
     * @param rule
     * @return boolean
     */
    @Override
```

```
    public boolean equals(Object rule)
    {
        if(rule instanceof Regla)
        {
            Regla aux = (Regla)rule;

            if(aux.ListaSimbolos.size() != ListaSimbolos.size())
                    return false;
            for(int i = 0; i < ListaSimbolos.size(); i++)
            {
                if(!ListaSimbolos.get(i).equals(aux.ListaSimbolos.get(i)))
                    return false;
            }
        }
        return true;
    }
}



public class Simbolo {

    private final String id;
    private final int tipo;//0 noterminal 1 terminal

    /**
     * Constructor de la clase que asigna la cadena que da nombre al simbolo
     * y el tipo de este.
     * @param _id
     * @param _tipo
     */
    public Simbolo(String _id, int _tipo)
    {
        id = _id;
        tipo = _tipo;
    }

    /**
     * Devuelve un string con el identificador del simbolo.
     * @return
     */
    public String getNombre()
    {
        return id;
    }

    /**
     * Devuelve un int con el tipo del simbolo:
     * 0 =  no terminal.
     * 1 =  terminal.
     * @return int
     */
    public int tipo()
    {
        return tipo;
    }

    /**
     * Sobrecarga del metodo equals para la comparacion de simbolos de la
     * gramatica.
     * @param simbolo
     * @return boolean
     */
    @Override
    public boolean equals(Object simbolo)
    {
        if( simbolo instanceof Simbolo)
        {
            Simbolo aux = (Simbolo) simbolo;
            return id.equals(aux.id);
        }
        return false;
    }

}
```

Capítulo 4: Algoritmo SLR.

El algoritmo SLR se lleva a cabo en la clase APV que imlementa el autómata de prefijos viables, esta clase tiene los siguientes atributos.

```
private final Estado inicial;//Estado inicial del sistema
    private Estado actual;//Estado actual de generacion del sistema
    private final ArrayList<ReglaAPV> iniciales;//Lista de reglas del automata.
    private int nEstados = 0;//Numero de estados del automata.
    TablaDR table = null;
```

Cada uno de los atributos esta comentado para su posterior interpretación, en el caso de los atributos, los atributos de tipo estado están creados en una clase propia con los siguientes atributos:

```
private int nEstado = 0; //Indica el numero del estado.
    private ArrayList<Estado> Siguientes = new ArrayList<>(); //Estados siguientes del
estado actual.
    private final Simbolo transicion; //Transicion que lleva a este estado.
    private final ArrayList<ReglaAPV> Reglas; //Lista de reglas del actual estado.
    private ArrayList<Simbolo> proximos = new ArrayList<>(); //proximos simbolos que se
pueden analizar.
```

Cada atributo tiene un conjunto de estados siguientes a una transición propia que lo creo(El inicial tiene de transición la cadena vacía) y un conjunto de reglas punteadas que cambian en cada estado según el símbolo que se consume.

Además para simplificar el proceso al crearse un estado, este crea a su vez una lista con los símbolos que puede consumir en cada una de sus reglas pues será cada estado el encargado de generar sus sucesores o enlazar con los existentes en caso de un estado ya creado; de eso se encargara el método **generaSiguiente(),** que consume uno de los símbolos próximos y devuelve un estado alcanzado con este símbolo.

```
public Estado generaSiguiente(ArrayList<ReglaAPV> globales)
    {
        //Clonamos las reglas globales del APV
        ArrayList<ReglaAPV> reglasBase = (ArrayList<ReglaAPV>) globales.clone();

        //Si el estado dispone de simbolos para consumir.
        if(proximos.size() > 0)
        {
            //Cogemos el siguiente simbolo.
            Simbolo consumir = proximos.remove(0);
            ArrayList<ReglaAPV> nuevas = new ArrayList<>();

            //Para cada una de las reglas de este estado.
            for(ReglaAPV r:Reglas)
            {
                //Genera una transicion si es posible consumir dicho simbolo
                ReglaAPV siguiente = r.transicion(consumir);
                if(siguiente != null)
                {
                    //Si se consiguio generar regla se añade a las del nuevo estado.
                    nuevas.add(siguiente);
                }
            }

            /**
```

```
             * Tenemos que añadir las reglas de los simbolos no temrinales
             * que estan como proximos en el estado de la lista de reglas iniciales.
             */
            for(int i = 0; i < nuevas.size(); i++)//Para cada regla del estado
            {
                if(!nuevas.get(i).esFinal())
                {
                    if(nuevas.get(i).getSiguiente().tipo() == 0)
                    {
                        for(int j = 0; j < reglasBase.size(); j++)//Buscamos las reglas
producidas
                        {
                            if(nuevas.get(i).getSiguiente().equals(re-
glasBase.get(j).getInicial()))
                            {
                                nuevas.add(reglasBase.remove(j));
                                j--;//AL usar remove la lista reduce en 1 su capacidad y
necesitamos volver.
                            }
                        }
                    }
                }
            }
        //Si hemos generado alguna regla devolvemos un nuevo estado con estas.
        if(!nuevas.isEmpty())
            return new Estado(nuevas, consumir);
        else
            return null;
        }
    return null;
    }
```

Como podemos observar al crear un nuevo estado este tendrá una nueva serie de reglas punteadas con los cambios necesarios en estas según el símbolo consumido.

Con esto simplificamos y modularizamos el proceso del algoritmo SLR pues en la propia clase de autómata el proceso se encargara simplemente de enlazar los estados entre ellos, eliminar las posibles repeticiones de estado y generar posteriormente las tablas necesarias. El método **slr()** es el siguiente.

```
public void slr()
    {
        ArrayList<Estado> Pendiente = new ArrayList<>();
        ArrayList<Estado> Visitados = new ArrayList<>();

        Pendiente.add(inicial);
        //Mientras queden producciones en el estado.
        while(Pendiente.size() > 0)
        {
            actual = Pendiente.remove(0);
            System.out.println("Evaluando estado " + actual.getNestado() + "....");
            if(Visitados.indexOf(actual) == -1)
            {
                while(actual.getProducciones())
                {
                    //Generamos un nuevo estado
                    Estado aux = actual.generaSiguiente(iniciales);
                    Estado find = buscar(aux);
                    //Si el estado no existia lo añadimos a siguientes
                    if(find == null)
                    {
                        nEstados++;
                        aux.setNumero(nEstados);
                        System.out.println("E" + actual.getNestado() +":" + " Transicion
" +
                                aux.getTransicion().getNombre() + " Genera el estado "
                                                    + aux.getNestado() + ".");
```

```
                        actual.addSiguiente(aux);
                    }
                    //Si existe lo linkamos a siguiente
                    else
                    {
                        System.out.println("E"+ actual.getNestado() + ":" + " Transicion
" +
                                           find.getTransicion().getNombre() + " Enlaza
el estado " +
                                           find.getNestado() + " .");
                        actual.addSiguiente(find);
                    }
                }
                //Añadimos los siguientes generados.
                Pendiente.addAll(actual.getListaSiguientes());
                Visitados.add(actual);
                Pendiente.removeAll(Visitados);
            }
        }

    }
```

El método mostrara por pantalla el proceso de manera ordenada para poder seguir la correcta creación del autómata.

El método buscar, hace una búsqueda en profundidad por el autómata creado y en caso de existir el estado buscado devuelve una referencia a este para enlazarlo al estado que lo ha creado.

Para todo este proceso se usa el objeto de tipo ReglaAPV que es una versión avanzada de una regla gramática, cada uno de estos objetos tiene un símbolo inicial, una lista de símbolos que son la producción de este y un puntero que indica donde se encuentra el punto en la expresión punteada, sabiendo así cuál es el símbolo que va a consumirse.

Capítulo 5: Generación de las tablas de reducción y desplazamiento.

Con vistas a la creación del archivo “parser.java” que será creado posteriormente, la estructura de la tabla de reducción/desplazamiento será parecida.

En primer lugar cabe marcar que en la clase APV se añadieron dos métodos necesarios para la creación de las tablas, los métodos primeros y siguientes, que a partir de las reglas iniciales en forma de ReglaAPV genera una lista con los símbolos siguientes y primeros de un símbolo no terminal. Esto será necesario para incluir en la tabla las entradas de reducción por una regla propia. Aquí podemos ver los dos métodos implementados.

```
/**
     * Devuelve una lista con los primeros del simbolo s dado
     * @param Simbolo
     * @return ArrayList
     */
    public ArrayList<Simbolo> getPrimeros(Simbolo s)
    {
        ArrayList ret = new ArrayList<Simbolo>(), aux = new ArrayList<Simbolo>();
        boolean lambda = false;

        //Para cada regla del conjunto.
        for(ReglaAPV r : iniciales)
        {
            if(r.getInicial().equals(s) && !r.getInicial().equals(r.getSiguiente()))
            {
                if(r.getSiguiente().tipo()==0)//Si es no terminal llamada recursiva.
                {
                    //Buscamos los primeros del simbolo
                    aux = getPrimeros(r.getSiguiente());

                    //Mientras primero incluya lambda seguimos avanzando.
                    if(aux.indexOf(new Simbolo("Lambda",1)) != -1);
                    {
                        ReglaAPV rAux = r.transicion(r.getSiguiente());
                        if(rAux != null)
                        {
                            ArrayList<Simbolo> temp = getPrimeros(rAux.getSiguiente());
                            aux.removeAll(temp);
                            aux.addAll(temp);
                        }
                    }
                    ret.removeAll(aux);
                    ret.addAll(aux);
                }
                else
                    ret.add(r.getSiguiente());//Si es terminal añade a la lista.
            }
        }
        return ret;
    }

    /**
     * Devuelve una lista de simbolos siguientes al simbolo dado.
     * @param s
     * @return
     */
    public ArrayList<Simbolo> getSiguientes(Simbolo s)
    {
        ArrayList<Simbolo> ret = new ArrayList<>();
```

```
        //Si es la regla inicial se añade el dolar.
        if(iniciales.get(0).getInicial().equals(s))
            ret.add(new Simbolo("EOF",1));

        //Para cada regla del conjunto.
        for(ReglaAPV r: iniciales)
        {
            Simbolo aux = r.getSiguienteA(s);
            //Si existe un simbolo siguiente a "s" en la regla.
            if(!aux.getNombre().equals("vacio"))
            {
                if(aux.tipo() == 1)
                {
                    ret.remove(aux);//Eliminamos para no repetir.
                    ret.add(aux);
                }
                else
                {
                    //Si ha terminado la regla mis siguientes son sus siguientes.
                    if(aux.equals(r.getInicial()))
                    {
                        ArrayList<Simbolo> sigAux = getSiguientes(aux);
                        ret.removeAll(sigAux);
                        ret.addAll(sigAux);
                    }
                    //Si el simbolo es no terminal, añado sus primeros.
                    else
                    {
                        ArrayList<Simbolo> primAux = getPrimeros(aux);
                        //Si entre sus primeros existe un lambda añado sus siguientes.
                        if(primAux.indexOf(new Simbolo("Lambda",1)) != -1)
                        {
                            ArrayList<Simbolo> sigAux = getSiguientes(aux);
                            primAux.removeAll(sigAux);
                            primAux.addAll(sigAux);
                        }
                        ret.removeAll(primAux);
                        ret.addAll(primAux);
                    }
                }
            }
        }
        return ret;
    }
```

En cuanto a la tabla RD como nos referiremos a ella de aquí en adelante, se basa en 3 matrices de enteros distribuidas en acciones, reglas e ir a.

La tabla de “acciones” tendrá tantas columnas como símbolos terminales tenga la gramática y tantas filas como estados tenga el autómata, por eso la tabla será escrita desde el mismo autómata, pues este tiene control sobre los estados y sus transiciones. Las acciones será expresadas mediante enteros, un 0 será el elemento vacío, el número **Integer.MIN_VALUE** de java será el símbolo aceptar, un número entero positivo indica el desplazamiento de un estado a otro y por ultimo un numero entero negativo indica la reducción por la regla que tenga el mismo número positivo.

La tabla de “ir a” contiene tantas columnas como símbolos no terminales tenga la gramática y tantas filas como estados, en este caso su contenido es más simple, pues solo tendrá el número de estado a desplazar con cada símbolo cuando este sea necesario.

Por último la tabla de reglas contiene tantas filas como reglas y 2 columnas, en la primera columna tendrá el símbolo inicial de dicha regla y en la derecha el número de símbolos terminales o no terminales de la regla, como sabemos esto será necesario para la posterior reducción por una regla dada.

Aquí podemos ver el código de la clase de la tabla:

```
public class TablaDR
{
    /**
     * tabla de ir_a tiene tantas columnas como simbolos no terminales
y tantas
     * filas como estados.
     */
    private int[][] go_to;

    /**
     * Tabla de acciones, tiene tantas columnas como simbolos termina-
les y
     * tantas filas como estados generados.
     */
    private int[][] Action;

    /**
     * Tabla de reglas, tiene tantas columnas como reglas y dos culum-
nas una
     * indica las reglas y otra la cantidad de simbolos producidas por
esta.
     */
    private int[][] Rules;

    /**
     * Constructor de la clase, inicia las tablas a sus tamaños nece-
sarios.
    *
     * @param Terminales
     * @param Noterminales
     * @param reglas
     * @param nEstados
    */

    //Listas de simbolos terminales y no terminales.
    private final ArrayList<String> Term;
    private final ArrayList<String> nTerm;


    public TablaDR(ArrayList<String> Terminales, ArrayList<String>
Noterminales, ArrayList<ReglaAPV> reglas, int nEstados)
    {
        Term = Terminales;
        nTerm = Noterminales;

        go_to = new int[nEstados][Noterminales.size()];
        Action = new int[nEstados][Terminales.size()];
        Rules  = new int[reglas.size()+1][2];

        //Iniciamos ya las reglas.
```

```
        Rules[0][0] = 0;
        Rules[0][1] = 0;
        for(int i = 1; i < reglas.size()+1; i++)
        {
            Rules[i][0] = Noterminales.indexOf(reglas.get(i-1).getIni-
cial().getNombre());
            Rules[i][1] = reglas.get(i-1).getNprod();
        }
    }

    /**
     * Añade un nuevo valor a la transicion desde el estado dado con
el simbolo
     * dado al estado estipulado
     * @param origen
     * @param simbolo
     * @param destino
     */
    public void addGoTo(int origen, String simbolo, int destino)
    {
        go_to[origen][nTerm.indexOf(simbolo)] = destino;
    }

    /**
     * Añade una accion a la tabla de acciones.
     * -1 = aceptar; x = desplazar a x; 1x = reducir por x;
     * @param Estado
     * @param simbolo
     * @param accion
     */
    public void addAction(int Estado, String simbolo, int accion)
    {
        Action[Estado][Term.indexOf(simbolo)] = accion;
    }


    public int[][] getReglas()
    {
        return Rules;
    }

    public int[][] getAcciones()
    {
        return Action;
    }

    public int[][] getIra()
    {
        return go_to;
    }
}
```

La tabla será iniciada y creada desde la clase AVP con los siguientes métodos:

```
/**
     * Inicia la tabla con el tamaño designado por los simbolos y los estados.
     * @param Terminales
     * @param Noterminales
     */
    public void Iniciar_Tabla(ArrayList<String> Terminales, ArrayList<String> Notermina-
les)
    {
        table = new TablaDR(Terminales, Noterminales, iniciales, nEstados+1);
    }


    /**
     * Crea la tabla de desplazamiento/reduccion despues de haber creado el automata.
     * @return Tabla
     * @throws java.lang.Exception
     */
    public TablaDR Fill_Table() throws Exception
    {
        if(table == null)
            throw new Exception("Tabla no inicializada");

        ArrayList<Estado> Pendiente = new ArrayList<>();
        ArrayList<Estado> Visitado = new ArrayList<>();
        actual = inicial;
        Pendiente.add(actual);

        while(!Pendiente.isEmpty())
        {
            actual = Pendiente.remove(0);//Cogemos el siguiente
            //Primero evaluamos el estado actual.
            int tipo = actual.getTipoEstado();
            System.out.println("Evaluando estado " + actual.getNestado() +
":.............................");
            if(tipo == Integer.MIN_VALUE)
            {
                System.out.println("Estado con x->Expr., añado aceptar con $");
                table.addAction(actual.getNestado(), "EOF", Integer.MIN_VALUE);//Añadi-
mos accion aceptar.
            }
            if(tipo < 0 && tipo > Integer.MIN_VALUE)
            {
                //Calculamos los siguietes al numero de la regla.
                System.out.println("E"+actual.getNestado()+": Accion_reduccion por regla
"+ (-tipo) );
                ArrayList<Simbolo> siguientes = getSiguientes(iniciales.get(tipo*(-1)-
1).getInicial());
                for(int i = 0; i < siguientes.size(); i++)
                {
                    System.out.println("\t Simbolo: "+ siguientes.get(i).getNombre() + "
reducir por regla " +
                                       (-tipo));
                    table.addAction(actual.getNestado(), siguientes.get(i).getNombre(),
tipo);
                }
            }
            for(Estado e: actual.getListaSiguientes())
            {
                if(e.getTransicion().tipo() == 0)
                {
                    System.out.println("E" + actual.getNestado() + ": Ir_a " + e.getNes-
tado() + " con " +
                                       e.getTransicion().getNombre());
                    table.addGoTo(actual.getNestado(), e.getTransicion().getNombre(),
e.getNestado());
                }
                else
                {
                    System.out.println("E" + actual.getNestado() + ": Accion_desplazar_a
" +
                                       e.getNestado() + " con " + e.getTransi-
cion().getNombre());
```

```
                    table.addAction(actual.getNestado(), e.getTransicion().getNombre(),
e.getNestado());
                }
            }
            Visitado.add(actual);//Añadimos a visitados
            Pendiente.addAll(actual.getListaSiguientes());//Añadimos los siguientes
            //Evitamos la redundancia.
            Pendiente.removeAll(Visitado);
        }
        return table;
    }
```

El método **FillTable()** que será el encargado de rellenar la tabla no es mas que un recorrido en profundidad por el autómata que ira evaluando cada estado y las transiciones de sus siguientes para crear la fila de la tabla correspondiente al estado evaluado, cuando sea necesario por una reducción el estado llamara al método siguientes que le dirá en que métodos tiene que añadir el valor de reducción.

Capítulo 6: creadores de ficheros.

Mediante dos clases crearemos los ficheros necesarios para analizar la nueva gramática generada por el analizador, en primer lugar podemos ver el código de la clase que genera los ficheros **TokenConstants** y **SimbolConstants**, esta clase es bastante sencilla simplemente recorre con un **for** las listas ya creadas de los símbolos terminales y no terminales y los escribe en un fichero con la estructura deseada, los símbolos se crearon y guardaron en el momento del análisis semántico y sintáctico para tenerlos en orden de aparición.

```
public class Simbol_Writer
{
    private FileWriter tokenConstants = null;
    private FileWriter SymbolConstans = null;

    /**
    *  Constructor de la clase, enlaza los dos archivos y ademas es-
cribe las
    *  cabeceras de las clases.
    *
     * @throws java.io.IOException
    */
    public Simbol_Writer() throws IOException
    {
        tokenConstants = new FileWriter("TokenConstants.java");
        SymbolConstans = new FileWriter("SymbolConstants.java");

        tokenConstants.write("public interface TokenConstants {\n");
        tokenConstants.write("\n");

        SymbolConstans.write("public interface SymbolConstants {\n");
        SymbolConstans.write("\n");
    }

    /**
     * Escribe la lista de simbolos en los respectivos archivos con
sus valores
     * @param Terminal
     * @param Noterminales
     * @throws IOException
     */
    public void escribe(ArrayList<String> Terminal, ArrayList<String>
Noterminales) throws IOException
    {
        for(int i = 0; i < Terminal.size(); i++)
        {
            tokenConstants.write("public int " + Terminal.get(i) + " =
" + (i) + ";\n");
        }
        tokenConstants.write("}");
        tokenConstants.close();

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

```
            SymbolConstans.write("public int " + Noterminales.get(i) +
" = " + i + ";\n");
          }
          SymbolConstans.write("}");
          SymbolConstans.close();
     }

}
```

El archivo Parser.java será un poco más complicado por eso su clase es un poco más compleja pero no demasiado más, una vez la clase APV ha rellenado la tabla será devuelta al objeto de tipo **Parser_Writer** y este se dedicara a recorrerá por filas e ir escribiendo el archivo con la forma correcta, podemos ver el código de la clase aquí:

```
public class Parser_Writer
{
    private FileWriter Parser = null;

    public Parser_Writer() throws IOException
    {
        Parser = new FileWriter("Parser.java");

        Parser.write("public class Parser extends SLRParser implements TokenCon-
stants,\n");
        Parser.write("SymbolConstants {\n\n");

        Parser.write("\tpublic Parser() {\n");
        Parser.write("\t\tinitRules();\n");
        Parser.write("\t\tinitActionTable();\n");
        Parser.write("\t\tinitGotoTable();\n}\n\n");
    }

    public void Escribir_Tabla(TablaDR tabla, ArrayList<String> Terminales, Ar-
rayList<String> Noterminales) throws IOException
    {
        Escribir_Reglas(tabla.getReglas(), Noterminales);
        Escribir_Acciones(tabla.getAcciones(),Terminales);
        Escribir_Ir_a(tabla.getIra(), Noterminales);
        Parser.write("}");
        Parser.close();
    }

    private void Escribir_Reglas(int[][] Reglas, ArrayList<String> Noterminales) throws
IOException
    {
        Parser.write("\tprivate void initRules() {\n");
        Parser.write("\t\tint[][] initRule = {\n");

        Parser.write("\t\t\t{ 0, 0 },\n");

        for(int i = 1; i < Reglas.length; i++)
        {
            Parser.write("\t\t\t{ " + Noterminales.get(Reglas[i][0]) + ", " + Re-
glas[i][1] + " },\n");
        }

        Parser.write("\t\t};\n\n");
        Parser.write("\t\tthis.rule = initRule;\n");
        Parser.write("\t}\n\n");
    }

    private void Escribir_Acciones(int[][] Acciones, ArrayList<String> Terminales)
throws IOException
    {
        Parser.write("\tprivate void initActionTable() {\n");
        Parser.write("\t\tactionTable = new ActionElement[24][10];\n\n");

        for(int i = 0; i < Acciones.length; i++)
        {
```

```
            for(int j = 0; j < Acciones[i].length; j++)
            {
                //ActionELement aceptar.
                if(Acciones[i][j] == Integer.MIN_VALUE)
                    Parser.write("\t\t\tactionTable [" + i + "][" + Terminales.get(j)
                                    + "] = new ActionElement(ActionElement.ACCEPT,
0);\n");
                if(Acciones[i][j] < 0 && Acciones[i][j] > Integer.MIN_VALUE)
                    Parser.write("\t\t\tactionTable [" + i + "][" + Terminales.get(j)
                                    + "] = new ActionElement(ActionElement.REDUCE, "
+ -Acciones[i][j] + ");\n");
                if(Acciones[i][j] > 0)
                    Parser.write("\t\t\tactionTable [" + i + "][" + Terminales.get(j) +
                            "] = new ActionElement(ActionElement.SHIFT, " + Ac-
ciones[i][j] + ");\n");
            }
            Parser.write("\n");
        }
        Parser.write("\t}\n\n");
    }

    private void Escribir_Ir_a(int[][] Go_to, ArrayList<String> Noterminales) throws IO-
Exception
    {
        boolean escrito = false;

        Parser.write("\tprivate void initGotoTable() {\n");
        Parser.write("\t\tgotoTable = new int[24][5];\n\n");

        for(int i = 0; i < Go_to.length; i++)
        {
            for(int j = 0; j < Go_to[i].length; j++)
            {
                if(Go_to[i][j] > 0)
                {
                    Parser.write("\t\t\tgotoTable [" + i + "][" + Noterminales.get(j) +
                            "] = " + Go_to[i][j] + ";\n");
                    escrito = true;
                }
            }
            if(escrito)
            {
                Parser.write("\n");
                escrito = false;
            }
        }
        Parser.write("\t}\n");
    }
}
```

Capítulo 10: Pruebas de aplicación.

Una vez realizada la aplicación y procurado el correcto funcionamiento de cada una de sus partes se han realizado pruebas con la gramática que tenemos de ejemplo, produciendo los mismos resultados que han de esperarse de esta gramática.

El proyecto ha generado archivos idénticos a los dados para la comprobación de funcionamiento de la práctica.

Además se ha creado un javadoc del proyecto para facilitar su compresión y posible adaptación o ampliación, en la carpeta de entrega se incluye copia del proyecto, memoria, archivos creados y una copia del javadoc por si quiere analizarse de manera más sencilla.