Toral Maldonado Rosa Guadalupe
2153045948
Compiladores

Analizador sintáctico

TAREA No. Y

COMPILADOR DESCENDENTE RECURSIVO, PREDICTIVO A MANO

Como habrá podido constatar, el código del **Parser** (Analizador Sintáctico) visto en clase, servirá como punto de partida para obtener la versión final del Compilador hecho a mano (tradicionalmente, sin herramientas); del cual tendrá que corregir ciertos detalles y/o completarlo. Proporcione lo necesario en cada uno de los siguientes temas:

1.- Análisis Sintáctico:

1.1.- Proporcione la **Gramática tipo 2** que define al Analizador Sintáctico, incluyendo las reglas de producción finales obtenidas en el curso.
1.2.- Incluya el análisis sintáctico del enunciado **para**, el cual se dejó en la tarea X. 2
1.3.- Complete el enunciado **si – entonces – otro**, como se dejó en la tarea X. 2
1.4.- Incluya el enunciado **imprime** conforme se pidió en la tarea X. 2
1.5.- Encierre en un marco **rojo** dentro del listado fuente la sección de código incluída y/o modificada para cada punto anterior.

2.- Manejo de Errores:

Conforme a las estrategias vistas en clase para el manejo de errores en un compilador, proporcione lo siguiente:

2.1.- Modifique el código de tal manera que implemente una de las siguientes técnicas:
a).- Sin Recuperación: Reporte del error.
b).- Con Recuperación: Modo de pánico.
c).- Con Recuperación: A nivel de frase.
2.2.- Encierre en un recuadro **azul** en su listado fuente, la sección de código incluída y/o modificada para implementar el inciso anterior.
2.3.- Describa en la documetación de su sistema cualquier tipo de detalle que considere conveniente.
2.4.- Los errore encontrados deberán reportarse en el archivo **<programa fuente>.err** , ejemplo:

```
alfa := beta * gama ;
        ^
```

ERROR: Se esperaba "="

(Implemente "manejo de errores sin recuperación, reporte del error")

2.5.- NOTA: Recuerde que siempre se debe concluír la compilación a pesar de encontrar un error.

3.- Generación de Código:

Proporcione lo siguiente:
3.1.- El **código intermedio para una máquina abstracta de pila**, el cual será el código objeto a generarse. Deberá guardarlo en un archivo con el siguiente nombre:
<programa fuente>.obj
3.2.-Incluya los **diagramas de transición modificados** de todas las reglas de producción de la Gramática tipo 2, la cual define al Parser del lenguaje **uami**, las llamadas respectivas a la rutina **emite** (la cual genera el código objeto), así como aquellas **variables auxiliares** que utilice en la generación del código.
3.3.- En su listado fuente encierre en un recuadro **negro** las invocaciones a la rutina **emite** y a las **variables auxiliares** que utilice.

4.- General:

4.1.- Proporcione un disco flexible con el código (programa) fuente de su compilador del lenguaje uami (todos los módulos que lo conforman, como lo son: lexico, parser, error, emisión de código objeto, etc.)

4.2.- Entregue un listado del programa fuente.
4.3.- Entregue una impresión del archivo que contiene el **código objeto** (*.obj) que su compilador genere para el programa fuente de prueba dado en clase.
4.4.- Entregue una impresión del archivo **<programa fuente>.err** generado.
4.5.- Entregue el enunciado de esta tarea.

Toral Maldonado Rosa Guadalupe
2153045948
Compiladores

Puntos 1.1 a 1.4 Gramática regular con las modificaciones hechas.

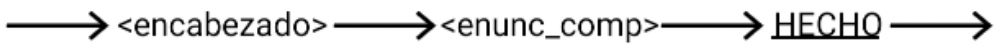
G_2 = (**T**, **V**, **P**, **S**)
Donde:

T = { HECHO, PROGRAMA, ID, ;, COMIENZA, TERMINA, ASIGNACION, SI, ENTONCES, OTRO, MIENTRAS, HAZ, IMPRIME, (, CADENA,), REPITE, HASTA, PARA, A, RELOP, ADDOP, MULOP, NUM_ENT }

V = { <programa>, <encabezado>, <encabezado>, <enunc_comp>, <enunciado>, <asignación>, <enunc_condicional>, <enunc_mientras>, <enunc_impresión>, < enunc_repite>, <enunc_para>, <expresión>, < expr_simple>, <término>, < factor> }

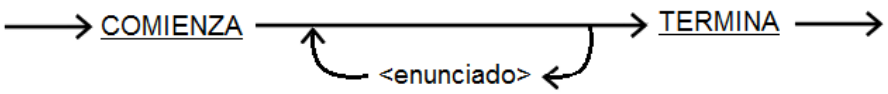
S = { <programa> }

P = { <programa> :

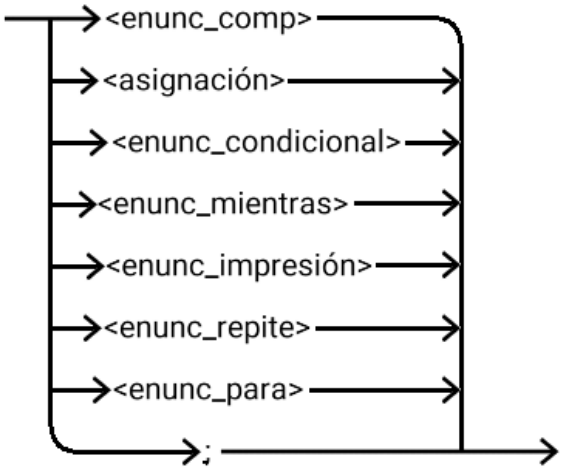


<encabezado>:

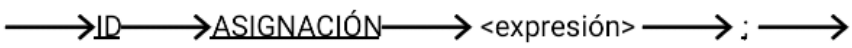
<enunc_comp>:



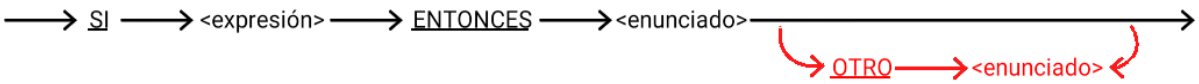
<enunciado>:



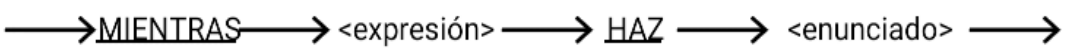
<asignación>:



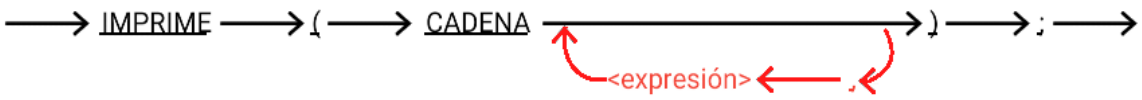
<enunc_condicional>:



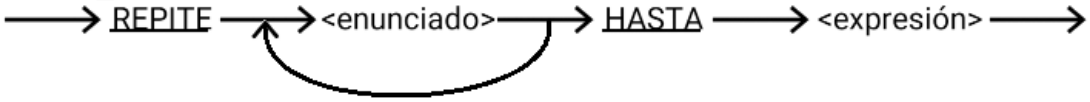
<enunc_mientras>:



<enunc_impresión>:

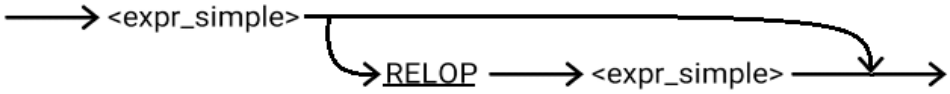


<enunc_repite>:

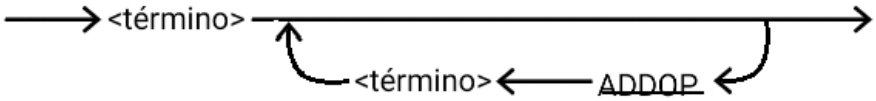


<enunc_para>:

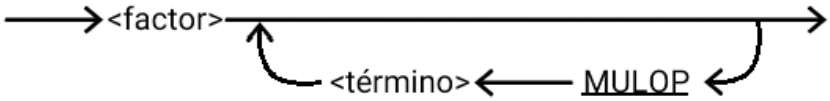
<expresión>:



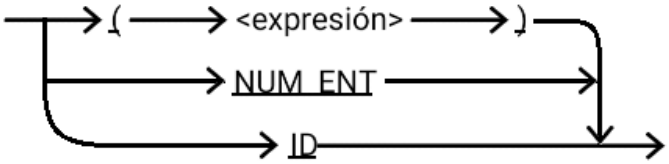
< expr_simple>:



<término>:



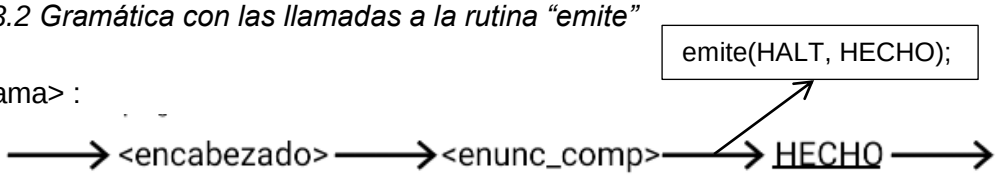
< factor>:



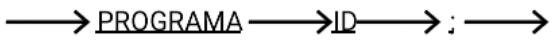
}

Punto 3.2 Gramática con las llamadas a la rutina "emite"

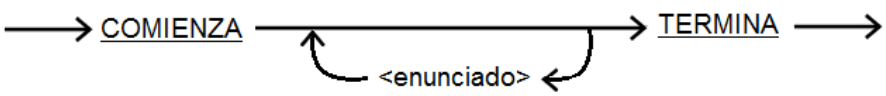
<programa> :



<encabezado>:

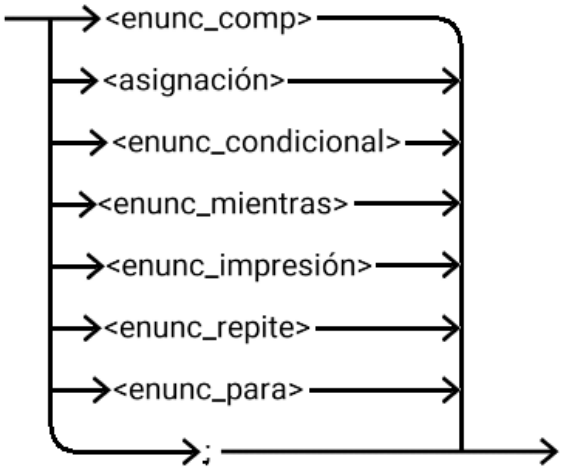


<enunc_comp>:



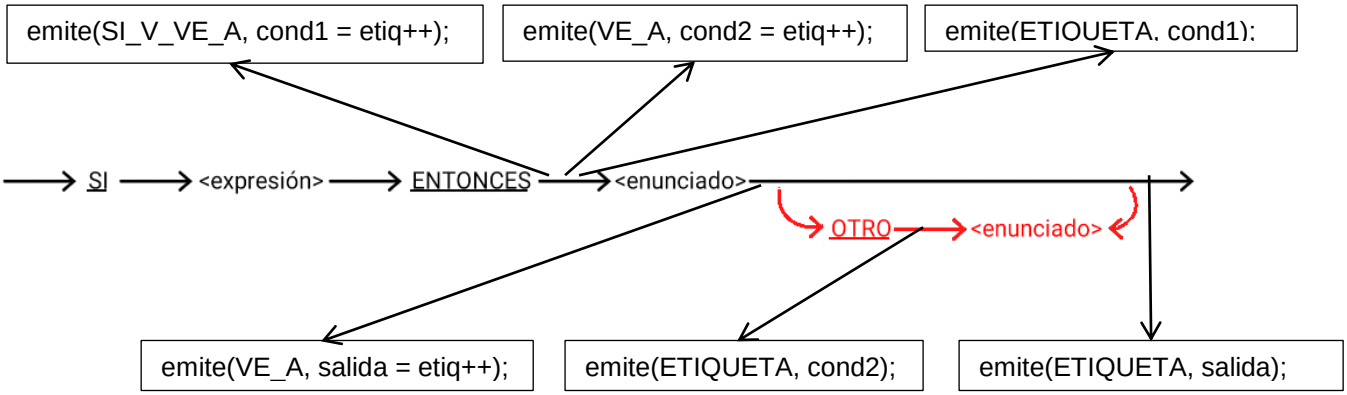
Toral Maldonado Rosa Guadalupe
2153045948
Compiladores

<enunciado>:

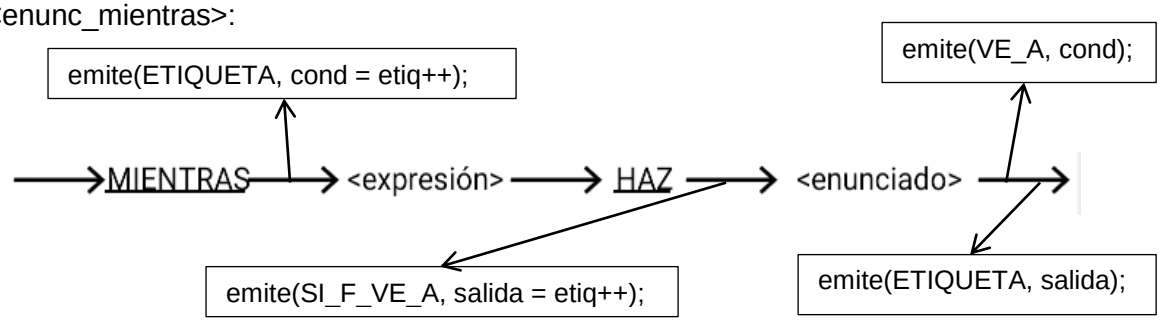


<asignación>:

<enunc_condicional>:

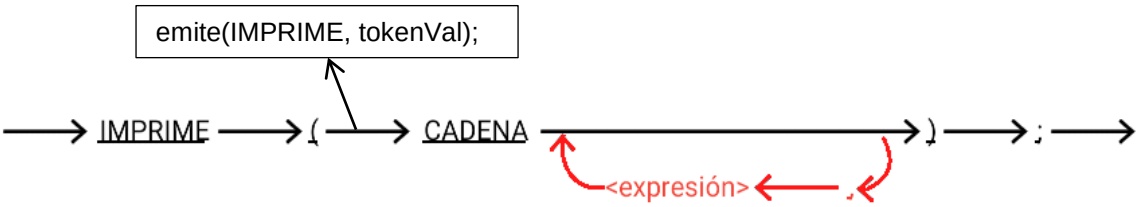


<enunc_mientras>:

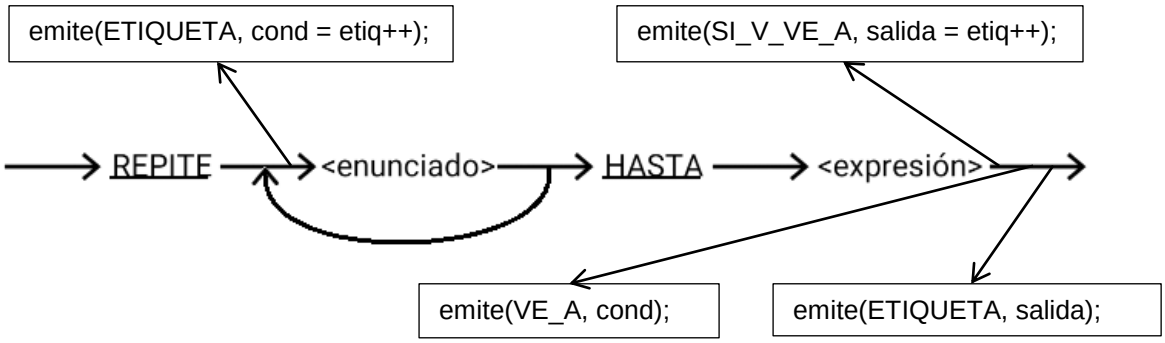


Toral Maldonado Rosa Guadalupe
2153045948
Compiladores

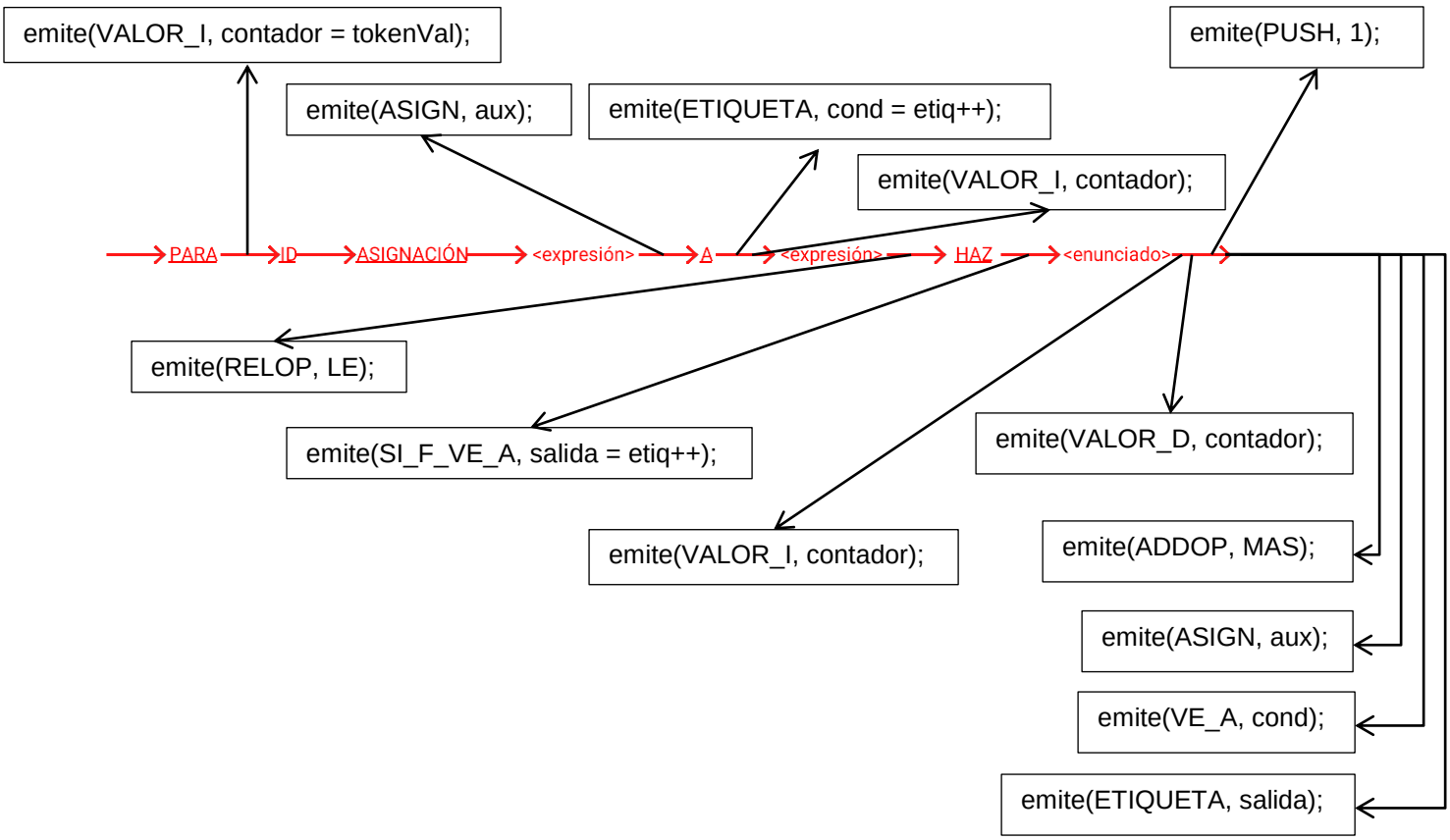
<enunc_impresión>:



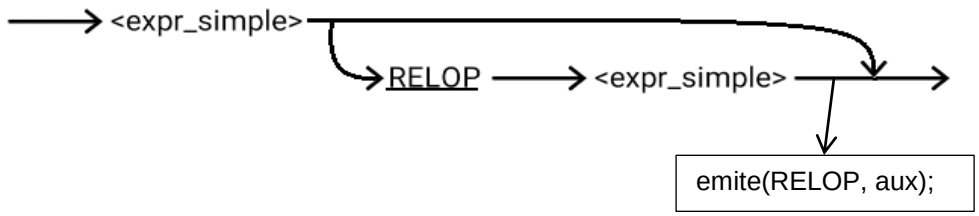
<enunc_repite>:



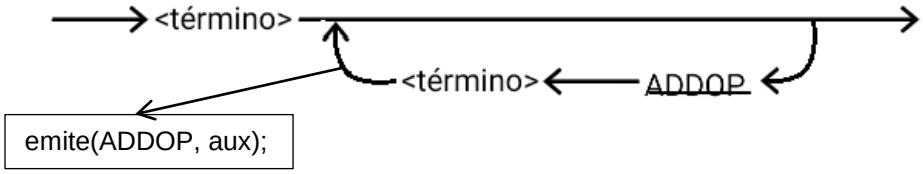
<enunc_para>:



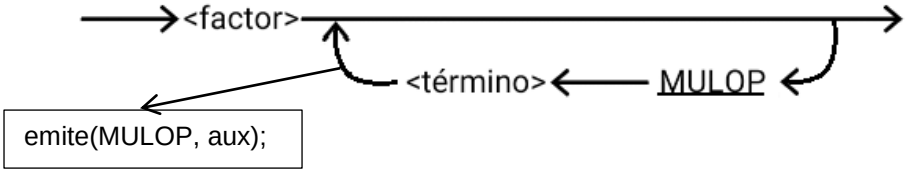
<expresión>:



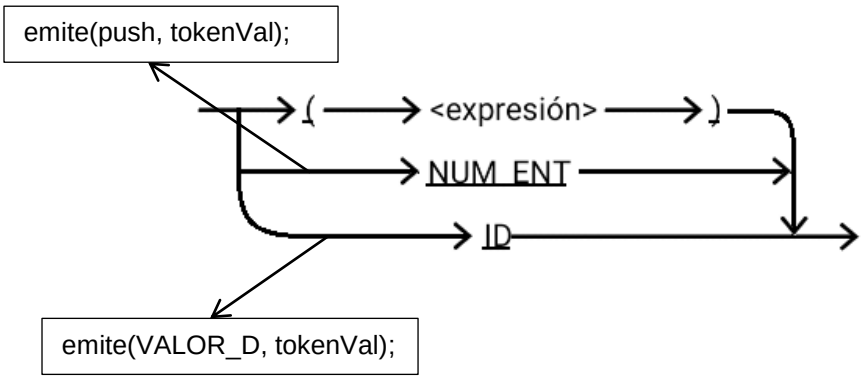
< expr_simple>:



<término>:



< factor>:



Puntos 1.5, 2 y 3.3 Analizador sintáctico (con indicaciones a las modificaciones hechas y llamadas a la rutina emite) y manejador de errores.

Clase AnalizadorSintactico

```
import java.io.IOException;

/**
* Toral Maldonado Rosa Guadalupe 2153045948 Compiladores Analizador Sintactico
* (parser)
*
*/
public class AnalizadorSintactico {

	private int preanalisis;
	private Alex lexer;
	private ManejadorDeErrores manejador;
	private Tabla_de_Simbolos tabla;
	private int etiq = 0;

	/** Constructor de la clase AnalizadorSintactico que inicializa al buffer de
	 * entrada y la tabla de simbolos (los cuales son usados por el Alex)
	 * @param buffer
	 * @param tabla
	 */
	public AnalizadorSintactico(BufferDeEntrada buffer, Tabla_de_Simbolos tabla) {
		this.tabla = tabla;
		lexer = new Alex(buffer, this.tabla);
		manejador = new ManejadorDeErrores(tabla, lexer);
	}

	/** Metodo que compara si lo que contiene preanalisis es igual a lo que la
	 * regla dice que valla despues.
	 * @param seEspera
	 */
	private void parea(int seEspera) {
		if (preanalisis == seEspera)
			preanalisis = lexer.alex();
		else {
			manejador.error(lexer.getNoLinea(), seEspera);
			preanalisis = lexer.alex();
		}
	}

	/** Metodo que crea el codigo
	 * @param demonico
	 * @param valor
	 */
	private void emite(int demonico, int valor) {
		try {
			switch (demonico) {
				case Constantes.PUSH: Principal.bwObj.write("push " + valor +
"\r\n");
					break;
			case Constantes.VALOR_I:	Principal.bwObj.write("valor_i " +
tabla.indicaValor(valor) + "\r\n");
					break;
			case Constantes.VALOR_D:	Principal.bwObj.write("valor_d " +
tabla.indicaValor(valor) + "\r\n");
					break;
			case Constantes.ASIGN:		Principal.bwObj.write(":=\r\n");
```

```
                        break;
                    case Constantes.ETIQUETA:    Principal.bwObj.write("etiqueta " + valor +
"\r\n");
                        break;
                    case Constantes.VE_A: Principal.bwObj.write("ve_a " + valor + "\r\n");
                        break;
                    case Constantes.SI_V_VE_A:   Principal.bwObj.write("si_v_ve_a " + valor +
"\r\n");
                        break;
                    case Constantes.SI_F_VE_A:   Principal.bwObj.write("si_f_ve_a " + valor +
"\r\n");
                        break;
                    // Caso de que sean operadores
                    case Constantes.ADDOP:
                    case Constantes.RELOP:
                    case Constantes.MULOP:
                        switch (valor) {
                        case Constantes.LT:   Principal.bwObj.write("<\r\n");
                            break;
                        case Constantes.LE:   Principal.bwObj.write("<=\r\n");
                            break;
                        case Constantes.EQ:   Principal.bwObj.write("==\r\n");
                            break;
                        case Constantes.GE:   Principal.bwObj.write(">=\r\n");
                            break;
                        case Constantes.GT:   Principal.bwObj.write(">" + "\r\n");
                            break;
                        case Constantes.NE:   Principal.bwObj.write("<>\r\n");
                            break;
                        case Constantes.MAS:   Principal.bwObj.write("+\r\n");
                            break;
                        case Constantes.MENOS:   Principal.bwObj.write("-\r\n");
                            break;
                        case Constantes.MULT:   Principal.bwObj.write("*\r\n");
                            break;
                        case Constantes.DIV:   Principal.bwObj.write("div\r\n");
                            break;
                        case Constantes.MODULO:   Principal.bwObj.write("mod\r\n");
                            break;
                        case Constantes.AND:  Principal.bwObj.write("and\r\n");
                            break;
                        case Constantes.OR:   Principal.bwObj.write("or\r\n");
                            break;
                        default:   Principal.bwObj.write("OPERADOR " + demonico + " " +
valor + "\r\n");
                            break;
                        }
                        break;
                    case Constantes.LEE:   Principal.bwObj.write("lee\r\n");
                        break;
                    case Constantes.ESCRIBE:   Principal.bwObj.write("escribe\r\n");
                        break;
                    case Constantes.IMPRIME:   Principal.bwObj.write("imprime " + valor +
"\r\n");
                        break;
                    case Constantes.HALT:   Principal.bwObj.write("halt\r\n");
                        break;
```

```
                        case Constantes.DUMP:   Principal.bwObj.write("dump\r\n");
                                break;
                        default:   Principal.bwObj.write("token " + demonico + "tokenval " + valor
+ "\r\n");
                                break;
                        }
                } catch (IOException err) {
                        System.out.println("No se puede escribir");
                }
        }

        /** Procedimiento correspondiente a la variable <programa>
         */
        public void programa() {
                preanalisis = lexer.alex();
                encabezado();
                enunc_comp();
                emite(Constantes.HALT, Constantes.HECHO);
                parea(Constantes.HECHO);
                System.out.println("Programa Terminado");
        }

        /** Procedimiento correspondiente a la variable <encabezado>
         */
        private void encabezado() {
                parea(Constantes.PROGRAMA);
                parea(Constantes.ID);
                parea(';');
        }

        /** Procedimiento correspondiente a la variable <enunc comp>
         */
        private void enunc_comp() {
                parea(Constantes.COMIENZA);
                while (preanalisis != Constantes.TERMINA) {
                        if (preanalisis == Constantes.HECHO)
                                break;
                        enunciado();
                }
                parea(Constantes.TERMINA);

        }

        /** Procedimiento correspondiente a la variable <enunciado>
         */
        private void enunciado() {
                switch (preanalisis) {
                case Constantes.COMIENZA: enunc_comp();
                        break;
                case Constantes.ID:   asignacion();
                        break;
                case Constantes.SI:   enunc_condicional();
                        break;
                case Constantes.MIENTRAS:   enunc_mientras();
                        break;
                case Constantes.IMPRIME:   enunc_impresion();
                        break;
```

```
                case Constantes.REPITE:    enunc_repite();
                        break;
                case Constantes.PARA:    enunc_para();
                        break;
                case ';':    parea(';');
                        break;
                default:  manejador.error(lexer.getNoLinea(), preanalisis); //preanalisis =
lexer.alex();
                        break;
                }
        }

        /** Procedimiento correspondiente a la variable <asignacion>
         */
        private void asignacion() {
                emite(Constantes.VALOR_I, lexer.getTokenVal());
                parea(Constantes.ID);
                int aux = lexer.getTokenVal();
                parea(Constantes.ASIGNACION);
                expresion();
                parea(';');
                emite(Constantes.ASIGN, aux);
        }

        /** Procedimiento correspondiente a la variable <enunc condicional>
         */
        private void enunc_condicional() {
                int cond1 = 0, cond2 = 0, salida = 0;
                parea(Constantes.SI);
                expresion();
                parea(Constantes.ENTONCES);
                emite(Constantes.SI_V_VE_A, cond1 = etiq++);
                emite(Constantes.VE_A, cond2 = etiq++);
                emite(Constantes.ETIQUETA, cond1);
                enunciado();
                emite(Constantes.VE_A, salida = etiq++);
                if (preanalisis == Constantes.OTRO) {
                        parea(Constantes.OTRO);
                        emite(Constantes.ETIQUETA, cond2);
                        enunciado();
                }
                emite(Constantes.ETIQUETA, salida);

        }

        /** Procedimiento correspondiente a la variable <enunc mientras>
         */
        private void enunc_mientras() {
                int cond = 0, salida = 0;
                parea(Constantes.MIENTRAS);
                emite(Constantes.ETIQUETA, cond = etiq++);
                expresion();
                parea(Constantes.HAZ);
                emite(Constantes.SI_F_VE_A, salida = etiq++);
                enunciado();
                emite(Constantes.VE_A, cond);
                emite(Constantes.ETIQUETA, salida);
```

```
}

/** Procedimiento correspondiente a la variable <enunc impresion>
 */
private void enunc_impresion() {
        parea(Constantes.IMPRIME);
        parea('(');
        emite(Constantes.IMPRIME, lexer.getTokenVal());
        parea(Constantes.CADENA);
        while (preanalisis == ',') {
                if(preanalisis == Constantes.HECHO)
                        break;
                parea(',');
                expresion();
        }
        parea(')');
        parea(';');
}

/** Procedimiento correspondiente a la variable <enunc repite>
 */
private void enunc_repite() {
        int cond = 0, salida = 0;
        parea(Constantes.REPITE);
        do {
                if (preanalisis == Constantes.HECHO)
                        break;
                emite(Constantes.ETIQUETA, cond = etiq++);
                enunciado();
        } while (preanalisis != Constantes.HASTA);
        parea(Constantes.HASTA);
        expresion();
        emite(Constantes.SI_V_VE_A, salida = etiq++);
        emite(Constantes.VE_A, cond);
        emite(Constantes.ETIQUETA, salida);
        parea(';');
}

/** Procedimiento correspondiente a la variable <enunc para>
 */
private void enunc_para() {
        int contador = 0, cond = 0, salida = 0, aux = 0;
        parea(Constantes.PARA);
                // Asignacion
        emite(Constantes.VALOR_I, contador = lexer.getTokenVal());
        parea(Constantes.ID);
        aux = lexer.getTokenVal();
        parea(Constantes.ASIGNACION);
        expresion();
        emite(Constantes.ASIGN, aux);
        parea(Constantes.A);
        emite(Constantes.ETIQUETA, cond = etiq++);
                // Condicion
        emite(Constantes.VALOR_I, contador);
        expresion();
        emite(Constantes.RELOP, Constantes.LE);
        parea(Constantes.HAZ);
        
```

```
        emite(Constantes.SI_F_VE_A, salida = etiq++);
        enunciado();
                // Incremento
        emite(Constantes.VALOR_I, contador);
        emite(Constantes.VALOR_D, contador);
        emite(Constantes.PUSH, 1);
        emite(Constantes.ADDOP, Constantes.MAS);
        emite(Constantes.ASIGN, aux);
        emite(Constantes.VE_A, cond);
        emite(Constantes.ETIQUETA, salida);
}

/** Procedimiento correspondiente a la variable <expresion>
 */
private void expresion() {
        expr_simple();
        int aux = lexer.getTokenVal();
        if (preanalisis == Constantes.RELOP) {
                parea(Constantes.RELOP);
                expr_simple();
                emite(Constantes.RELOP, aux);
        }

}

/** Procedimiento correspondiente a la variable <expr simple>
 */
private void expr_simple() {
        termino();
        int aux = lexer.getTokenVal();
        while (preanalisis == Constantes.ADDOP) {
                if (preanalisis == Constantes.HECHO)
                        break;
                parea(Constantes.ADDOP);
                termino();
                emite(Constantes.ADDOP, aux);
        }

}

/** Procedimiento correspondiente a la variable <termino>
 */
private void termino() {
        factor();
        int aux = lexer.getTokenVal();
        while (preanalisis == Constantes.MULOP) {
                if(preanalisis == Constantes.HECHO)
                        break;
                parea(Constantes.MULOP);
                termino();
                emite(Constantes.MULOP, aux);
        }
}

/** Procedimiento correspondiente a la variable <factor>
 */
private void factor() {
        switch (preanalisis) {
```

```
                case '(': {
                        parea('(');
                        expresion();
                        parea(')');
                        break;
                }
                case Constantes.NUM_ENT: {
                        emite(Constantes.PUSH, lexer.getTokenVal());
                        parea(Constantes.NUM_ENT);
                        break;
                }
                case Constantes.ID: {
                        emite(Constantes.VALOR_D, lexer.getTokenVal());
                        parea(Constantes.ID);
                        break;
                }
                default: manejador.error(lexer.getNoLinea(), preanalisis); //preanalisis =
lexer.alex();
                        break;
                }
        }
}
```

Clase ManejadorDeErrores

```
import java.io.IOException;
/** COMPILADORES
* Toral Maldonado Rosa Guadalupe.
* 2153045948
*/
public class ManejadorDeErrores {

        // Instancias de la tabla de Simbolos y el Buffer de entrada
        Tabla_de_Simbolos tabla;
        Alex lexer;

        /** Constructor de la clase ManejadorDeErrores
         * @param tabla
         * @param lexer
         */
        public ManejadorDeErrores(Tabla_de_Simbolos tabla, Alex lexer) {
                this.tabla = tabla;
                this.lexer = lexer;
        }

        /** Metodo que imprime el error encontrado y el numero de linea donde se encuantra
         * @param noLinea
         * @param seEspera
         */
        public void error(int noLinea, int seEspera) {
                try{
                switch(seEspera){
                case '(':       Principal.bwError.write("Error " + noLinea + ":" + ". Se esperaba: "
+ (char)seEspera + "\r\n");
                        break;
                case ')':       Principal.bwError.write("Error " + noLinea + ":" + ". Se esperaba: "
+ (char)seEspera + "\r\n");
                        break;
```

```
                case ',':      Principal.bwError.write("Error " + noLinea + ":" + ". Se esperaba: "
+ (char)seEspera + "\r\n");
                        break;
                case ';':      Principal.bwError.write("Error " + noLinea + ":" + ". Se esperaba: "
+ (char)seEspera + "\r\n");
                        break;
                case '"':   Principal.bwError.write("Error " + noLinea + ":" + ". Se esperaba: " +
(char)seEspera + "\r\n");
                        break;
                case Constantes.COMIENZA:    Principal.bwError.write("Error " + noLinea + ":" + ".
Se esperaba: " + Constantes.regresaValor(seEspera) + "\r\n");
                        break;
                case Constantes.TERMINA:     Principal.bwError.write("Error " + noLinea + ":" + ".
Se esperaba: " + Constantes.regresaValor(seEspera) + "\r\n");
                        break;
                case Constantes.A:      Principal.bwError.write("Error " + noLinea + ":" + ". Se
esperaba: " + Constantes.regresaValor(seEspera) + "\r\n");
                        break;
                case Constantes.NUM_ENT:     Principal.bwError.write("Error " + noLinea + ":" + ".
Se esperaba: " + Constantes.regresaValor(seEspera) + "\r\n");
                        break;
                case Constantes.ASIGNACION:    Principal.bwError.write("Error " + noLinea + ":" +
". Se esperaba: '=' " + "\r\n");
                        break;
                case Constantes.CADENA:    Principal.bwError.write("Error " + noLinea + ":" + ". Se
esperaba: " + Constantes.regresaValor(seEspera) + "\r\n");
                        break;
                case Constantes.ID:     Principal.bwError.write("Error " + noLinea + ":" + ". Se
esperaba: " + Constantes.regresaValor(seEspera) + "\r\n");
                        break;
                case Constantes.PROGRAMA:    Principal.bwError.write("Error " + noLinea + ":" + ".
Se esperaba: " + Constantes.regresaValor(seEspera) + "\r\n");
                        break;
                case Constantes.ENTONCES:    Principal.bwError.write("Error " + noLinea + ":" + ".
Se esperaba: " + Constantes.regresaValor(seEspera) + "\r\n");
                        break;
                case Constantes.OTRO:    Principal.bwError.write("Error " + noLinea + ":" + ". Se
esperaba: " + Constantes.regresaValor(seEspera) + "\r\n");
                        break;
                case Constantes.HAZ:     Principal.bwError.write("Error " + noLinea + ":" + ". Se
esperaba: " + Constantes.regresaValor(seEspera) + "\r\n");
                        break;
                case Constantes.IMPRIME:    Principal.bwError.write("Error " + noLinea + ":" + ".
Se esperaba: " + Constantes.regresaValor(seEspera) + "\r\n");
                        break;
                case Constantes.HASTA:    Principal.bwError.write("Error " + noLinea + ":" + ". Se
esperaba: " + Constantes.regresaValor(seEspera) + "\r\n");
                        break;
                default: Principal.bwError.write("Error " + seEspera + "\r\n");
                        break;
                }
                }catch(IOException err){
                        System.out.println("Error al escribir en el archivo");
                }
        }
}
```

Otras clases utilizadas

_____Clase Alex_____

```
/**
 * COMPILADORES Toral Maldonado Rosa Guadalupe. 2153045948
 */
public class Alex {

	// Se crea una instancia del buffer de entrada. Se declara estática para que
	// sea solamente un buffer
	private BufferDeEntrada buffer;
	private Tabla_de_Simbolos tabla;
	private int num_linea = 1;
	private int tokenVal;
	private String valor = "";
	private int posicion = 0;

	/** Constructor de la clase Alex
	 * @param buffer
	 * @param tabla
	 */
	public Alex(BufferDeEntrada buffer, Tabla_de_Simbolos tabla) {
		this.buffer = buffer;
		this.tabla = tabla;
	}

	/** Metodo que regresa el tokenVal Regresa un entero con eel valor del
	 * tokenVal
	 */
	public int getTokenVal() {
		return tokenVal;
	}

	/** Metodo que regresa el numero de lineas leido Regresa un entero con el
	 * valor del numero de lineas contado
	 */
	public int getNoLinea() {
		return num_linea;
	}

	/** Funcion alex. Es el metodo que se manda a llamar para analizar un archivo
	 * Regresa un entero con el tipo de la palabra encontrada
	 */
	public int alex() {
		// Verifica, si el buffer de entrada esta vacio o el contenido ya fue
		// analizado se vuelve a cargar
		if (buffer.esVacio() == 0)
			buffer.cargaBE();
		char c = buffer.lee();

		// Verifica, si es un delimitador o comentario <eb>
		while (esMetapalabraOComentario(c) == true) {
			// Verifica, si llega una llave que abre es un comentario. Lee todo
			// lo que le sigue hasta que encuentre un salto de linea
			if (c == '{')
				while (c != Constantes.EOS && c != '\n')
					c = buffer.lee();
			// Verifica, si es un salto de linea aumenta el numero de lineas
			// analizado
			if (c == '\n')
				num_linea++;
			c = buffer.lee();
		}

		// Verifica si es un numero entero <N>
```

```
		if (Character.isDigit(c)) {
			return esN(c);
		}

		// Verifica si es un id <id>
		if (Character.isLetter(c))
			return esID(c);

		// Verifica si es una cadena <cad>
		if (c == '"') {
			return esCadena(c);
		}

		// Verifica si es un logico <logic ampeer pleca>
		if (c == '&' || c == '|') {
			return esLogic_amper_pleca(c);
		}

		// Verifica si es un relacional <relac admin asig>
		if (c == '>' || c == '!' || c == '<' || c == '=') {
			return esRelac_admin_asig(c);
		}

		// Verifica si es un aritmetico <arit>
		if (c == '+' || c == '*' || c == '-' || c == '/' || c == '%') {
			return esAritmetico(c);
		}

		// Verifica si es el fin de archivo <>
		if (c == Constantes.EOS) {
			tokenVal = Constantes.NINGUNO;
			return Constantes.HECHO;
		}

		// Si no pertenece a ninguno de los anteriores se regresa el codigo
		// ASCII del caracter
		tokenVal = Constantes.NINGUNO;
		return c;
	}

	/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
	 * METODOS MODULARES USADOS PARA VERIFICAR LA REGLA DE PRODUCCION A USAR   *
	 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
	 */

	/** Metodo que indica si un caracter es una metapalabra. Regla de produccion
	 * <eb> Regresa true si c es una metapalabra, false de lo contrario
	 */
	private boolean esMetapalabraOComentario(char c) {
		if (c == ' ' || c == '\t' || c == '\n' || c == '{')
			return true;
		return false;
	}

	/** Metodo que identifica ID's. Regla de produccion <id>
	 */
	private int esID(char c) {
		valor = "";
		while (Character.isLetterOrDigit(c) || c == '_') {
			valor = valor + c;
			c = buffer.lee();
		}
		// Deslee al ultimo valor valido
		buffer.deslee();
		// Se busca en la tabla de simbolos
		posicion = tabla.buscaValor(valor);
		// Si regresa un -1, entonces el elmento no exiiste en la tabla de
		// simbolos
		// Se agrega
```

```
		if (posicion == -1)
			posicion = tabla.inserta(valor, Constantes.ID);
		tokenVal = posicion;
		return tabla.indicaTipo(posicion);
	}

	/** Metodo que indentifica numeros enteros numeros enteros. Regla de
	 * produccion <N>
	 */
	private int esN(char c) {
		valor = "";
		while (Character.isDigit(c)) {
			valor = valor + c;
			c = buffer.lee();
		}
		// Deslee al ultimo valor valido
		buffer.deslee();
		tokenVal = Integer.parseInt(valor);
		return Constantes.NUM_ENT;
	}

	/** Metodo que identifica si hay simbolos relacionales, signos de admiracion
	 * o de asignacion. Regla de produccion <relac admir asig>. Regresa true si
	 * pertenece al grupo
	 */
	private int esRelac_admin_asig(char c) {
		// Se guarda el valor actual del caracter
		char aux = c;
		if (c == '>' || c == '!' || c == '=') {
			if (c == '>') {
				c = buffer.lee();
				if (c == '=') {
					// Lo concatena al caracter anterior para que sea un mayor o
					// igual que
					valor = valor + c;
					tokenVal = Constantes.GE;
				} else {

					// Deslee alultimo valor valido
					buffer.deslee();
					c = aux;
					tokenVal = Constantes.GT;
				}
				return Constantes.RELOP;
			}
			if (c == '!') {
				c = buffer.lee();
				if (c == '=') {
					// Lo concatena al caracter anterior para que sea un
					// distinto
					valor = valor + c;
					tokenVal = Constantes.NE;
					return Constantes.RELOP;
				}
				// Deslee al ultimo valor valido
				buffer.deslee();
				// Se carga el valor inicial antes de analizar la palabra
				c = aux;
				tokenVal = Constantes.NINGUNO;
				// Como el signo de admiracion no esta contemplado dentro de
				// las constantes globales, el tokenVal va a ser
				// NINGUNO y el tipo va a ser su equivalente en ASCII
			}
			if (c == '=') {
				c = buffer.lee();
				if (c == '=') {
					// Lo concatena al caracter anterior para que sea un igual a
					// (doble)
					valor = valor + c;
```

```
                        tokenVal = Constantes.EQ;
                        return Constantes.RELOP;
                }
                // Deslee al ultimo valor valido
                buffer.deslee();
                // Se carga el valor inicial antes de analizar la palabra
                c = aux;
                tokenVal = Constantes.NINGUNO;
                return Constantes.ASIGNACION;
            }
        }
        if (c == '<') {
            c = buffer.lee();
            // Lo concatena al caracter anterior para que sea un menor o igual
            // que, o un distinto de
            valor = valor + c;
            if (c == '=')
                tokenVal = Constantes.LE;
            if (c == '>')
                tokenVal = Constantes.NE;
            if (c != '>' && c != '=')
                tokenVal = Constantes.LT;
            return Constantes.RELOP;
        }
        return c;
    }

    /** Metodo que identifica si hay simbolos logicos, ampersands (&) o plecas
     * (|) solos. Regla de produccion <logic amper pleca>. Rgresa true si
     * pertenece al grupo
     */
    private int esLogic_amper_pleca(char c) {
        // Se guarda el valor actual de c
        char aux = c;
        if (c == '&') {
            c = buffer.lee();
            if (c == '&') {
                // Lo concatena al caracter anterior para que sea un AND
                valor = valor + c;
                tokenVal = Constantes.AND;
                return Constantes.MULOP;
                // Como el & no está definido dentro de las palabras
                // reservadas usadas, entonces el tokenVal va a ser igual a
                // NINGUNO y el tipo va a ser el codigo ASCII de &
            }
            // Deslee al ultimo valor valido
            buffer.deslee();
            // Se carga el valor inicial de c antes de analizar la variable
            c = aux;
        }
        if (c == '|') {
            c = buffer.lee();
            if (c == '|') {
                // Lo concatena al caracter anterior para que sea un OR
                valor = valor + c;
                tokenVal = Constantes.OR;
                return Constantes.ADDOP;
                // Como el | no está definido dentro de las palabras
                // reservadas usadas, entonces el tokenVal va a ser igual a
                // NINGUNO y el tipo va a regresar el codigo ASCII de |
            }
            // Deslee al ultimo valor valido
            buffer.deslee();
            // Se carga el valor inicial de c antes de analizar la variable
            c = aux;
        }
        tokenVal = Constantes.NINGUNO;
        return c;
    }
```

```
	/** Metodo que identifica si hay cadenas. Regla de producion <cad>
	 */
	private int esCadena(char c) {
		valor = "";
		// Concatena las primeras comillas
		valor = valor + c;
		c = buffer.lee();
		while (c != '"') {
			if (c == Constantes.EOS)
				break;
			else {
				valor = valor + c;
				c = buffer.lee();
			}
		}
		if (c == Constantes.EOS)
			return Constantes.HECHO;
		else {
			// Concatena la ultimas comillas
			valor = valor + c;
			// Se mete a la tabal de simbolos
			posicion = tabla.buscaValor(valor);
			if (posicion == -1)
				posicion = tabla.inserta(valor, Constantes.CADENA);
			tokenVal = posicion;
			return tabla.indicaTipo(posicion);
		}
	}

	/** Metodo que analiza si el caracter leido es un operador aritmetico Regresa
	 * un entero con el tipo de la palabra encontrada
	 */
	private int esAritmetico(char c) {
		// Si c es un + o un -, regresa el tipo ADDOP
		if (c == '+' || c == '-') {
			if (c == '-')
				tokenVal = Constantes.MENOS;
			if (c == '+')
				tokenVal = Constantes.MAS;
			return Constantes.ADDOP;
		} else {
			// Si c es un *, o / o % regresa el tipo MULOP
			if (c == '*')
				tokenVal = Constantes.MULT;
			if (c == '/')
				tokenVal = Constantes.DIV;
			if (c == '%')
				tokenVal = Constantes.MODULO;
			return Constantes.MULOP;
		}
	}
}
```

Toral Maldonado Rosa Guadalupe
2153045948
Compiladores

Clase BufferDeEntrada

```java
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

import javax.swing.JOptionPane;
/** COMPILADORES
* Toral Maldonado Rosa Guadalupe.
* 2153045948
*/
public class BufferDeEntrada {

	// Define el tamaño del arreglo
	private static final int TAM_BUFFER = 200;

	// Arreglo tipo cola que simula al buffer de entrada.
	private char[] buffer = new char[TAM_BUFFER];

	// Reader que sirve para leer el archivo fuente.
	private BufferedReader b;

	// Apuntador al frente de la cola (buffer)
	private int fren = -1;

	// Apuntador al posterior de la cola (buffer)
	private int post = -1;

	/** Constructor de la clse BufferDeEntrada
	 */
	public BufferDeEntrada() {

	}

	/** Metodo que abre el archivo fuente
	 *  @param f
	 * @return 0 si se cargo con exito, 1 de lo contrario
	 */
	public Integer identificaF(String f) {
		// Se controla en caso de error
		try {
			// Se lee el archivo fuente
			b = new BufferedReader(new FileReader(f));
			// Se atrapa la excepcion en caso de que no lea el archivo y regresa
			// 1
		} catch (FileNotFoundException e) {
			return 1;
		}
		// Regresa 0 si se cargo con exito
		return 0;
	}

	/** Metodo que carga el buffer con el siguiente contenido del archivo fuente
	 */
	public void cargaBE() {
		// Variable del tipo entero que guarda el codigo ASCII del siguiente
		// caracter encontrado en el archivo
```

```
		int siguiente = 0;

		// El frente y el posterior deben reiniciarse cuando el buffer es
		// llenado nuevamente
		post = -1;
		fren = -1;

		// Se controla en caso de error
		try {
			// Se guardan los primeros 200 caracteres en el buffer
			while ((post < (TAM_BUFFER - 1)) && ((siguiente = b.read()) != -1)) {
				// Se eliminan los retornos de carro (\r) y se dejan solamente
				// los saltos de linea (\n)
				if (siguiente != '\r') {
					post++;
					buffer[post] = (char) siguiente;
				}
			}

			// Se atrapa la excepcion en caso de que no se pueda seguir leyendo
			// el contenido del archivo. Lanza un mensaje de error.
		} catch (IOException e) {
			JOptionPane.showMessageDialog(null, "No se puede leer el contenido del
archivo");
		}
	}

	/** Metodo que lee el siguiente caracter de la entrada
	 * @return el caracter leido
	 * @throws Exception
	 */
	public Character lee() {
		// Se verifica, si está vacío que no regrese nada, y el frente lo
		// reinicie a -1
		if (esVacio() == 0 || fren == (TAM_BUFFER - 1)) {
			// Si el frente esta en la última psición es porque ya se trabajo
			// con todo el contenido
			if (fren == (TAM_BUFFER - 1))
				// Se vuelve a cargar el Buffer de Entrada
				cargaBE();
		}

		// Si el frente es mayor al posterior, quiere decir que ya no hay más
		// caracteres que leer: llego al fin de archivo
		if (fren > post)
			return Constantes.EOS;

		// Incrementa el frente y regresa a lo que apunta fren
		fren++;
		return buffer[fren];
	}

	/** Metodo que deslee el último caracter encontrado
	 * @param f
	 */
	public void deslee() {
		// Verifica, si el frente está en la posición 0 o -1, manda un mensaje
		// de error
```

```
			if (fren == -1)
				System.out.println("No se puede hacer la operacion");
			// De otra manera decrementa el frente regresandolo a la posición
			// anterior
			else {
				fren--;
			}
		}

		/** Metodo que indica si el buffer esta lleno
		 * @return 0 si esta lleno o 1 de lo contrario
		 */
		public Integer estaLleno() {
			// Verifica, si el posterior esta situado antes de la ultima posicion,
			// entonces el buffer esta lleno y regresa 0
			if (post == (TAM_BUFFER - 1)) {
				return 0;
			}
			// Regresa 1, en caso de que no este lleno
			return 1;
		}

		/**Metodo que indica si el buffer esta vacio
		 * @return 0 si esta vacio o 0 de lo contrario
		 */
		public Integer esVacio() {
			// Verifica, si el frente y el posterior no han sido movidos de su
			// posicion inicial, el buffer esta vacio y regresa 0
			if ((fren == -1 && post == -1))
				return 0;
			// Regresa 1, en caso de que no este vacio
			return 1;
		}
}
```

_____Clase Tabla_de_Simbolos_____

```
/**
* Toral Maldonado Rosqa Guadalupe
* 2153045948
* Compiladores
*/

import java.util.LinkedList;

public class Tabla_de_Simbolos {

		// Se crea una clase Simbolos que contenga como parametros el valor y el
		// tipo de la tabla de simbolos
		private class Simbolo {

			// Atributos de la clase Simbolos
			private int tipo;
			private final String valor;

			/** Constructor de la clase Simbolo
			 * @param tipo
			 * @param valor
			 */
```

```
	Simbolo(int tipo, String valor) {
		this.tipo = tipo;
		this.valor = valor;
	}

	/** Método que regresa los datos guardados en tipo y valor
	 */
	@Override
	public String toString() {
		return "[" + valor + ", " + tipo + "]";
	}
}

// Se crea un LinkedList Para simular la tabla
private final LinkedList<Simbolo> tablaSimb;

/** Constructor de la clase Tabla_De_Simbolos que inicializa la tabla
 * Reemplaza al metodo inicializa.
 */
public Tabla_de_Simbolos() {
	// Se incia la tabla de simbolos
	tablaSimb = new LinkedList<>();

	// Se llena con las palabras reservadas
	Simbolo palabrasReserv = new Simbolo(Constantes.PROGRAMA, "programa");
	tablaSimb.add(palabrasReserv);
	palabrasReserv = new Simbolo(Constantes.COMIENZA, "comienza");
	tablaSimb.add(palabrasReserv);
	palabrasReserv = new Simbolo(Constantes.TERMINA, "termina");
	tablaSimb.add(palabrasReserv);
	palabrasReserv = new Simbolo(Constantes.SI, "si");
	tablaSimb.add(palabrasReserv);
	palabrasReserv = new Simbolo(Constantes.ENTONCES, "entonces");
	tablaSimb.add(palabrasReserv);
	palabrasReserv = new Simbolo(Constantes.OTRO, "otro");
	tablaSimb.add(palabrasReserv);
	palabrasReserv = new Simbolo(Constantes.MIENTRAS, "mientras");
	tablaSimb.add(palabrasReserv);
	palabrasReserv = new Simbolo(Constantes.HAZ, "haz");
	tablaSimb.add(palabrasReserv);
	palabrasReserv = new Simbolo(Constantes.IMPRIME, "imprime");
	tablaSimb.add(palabrasReserv);
	palabrasReserv = new Simbolo(Constantes.REPITE, "repite");
	tablaSimb.add(palabrasReserv);
	palabrasReserv = new Simbolo(Constantes.PARA, "para");
	tablaSimb.add(palabrasReserv);
	palabrasReserv = new Simbolo(Constantes.HASTA, "hasta");
	tablaSimb.add(palabrasReserv);
	palabrasReserv = new Simbolo(Constantes.A, "a");
	tablaSimb.add(palabrasReserv);
}

/** Metodo que busca un valor en la lista
 *  @param v
 * @return la posicion del valor en la lista
 */
public Integer buscaValor(String v) {
```

```
		// indice que sirve para recorrer el arreglo
		int i = 0;
		// Se recorre el arreglo, si el valor contenido de la tabla es igual al
		// que esta buscando, deja de recorrer el arreglo
		for (i = 0; i < tablaSimb.size(); i++)
			if (tablaSimb.get(i).valor.compareTo(v) == 0)
				break;

		// Verifica si el indice es igual al tamaño del arreglo, regresa un -1
		if (i == tablaSimb.size())
			return -1;
		return i;
	}

	/** Metodo que inserta un nuevo simbolo a la tabla
	 * @param v
	 * @param t
	 * @return la posicion en donde se inserto el nuevo elemento
	 */
	public Integer inserta(String v, int t) {
		// Se crea un nuevo simbolo y se agrega a la tabla de simbolos
		Simbolo simbol = new Simbolo(t, v);
		tablaSimb.add(simbol);

		// Se regresa la posicion donde se inserto. Se usa el tamaño de la
		// lista, ya que se inserta en la última posicion
		return tablaSimb.size() - 1;
	}

	/** Metodo que permite cambiar el tipo de un simbolo
	 * @param pos
	 * @param n_t
	 */
	public void fijaTipo(int pos, int n_t) {
		// Verifica, si la el entero "pos" se sale de la ultima posicion dentro
		// de la tabla de simbolos manda un error
		if (pos >= tablaSimb.size())
			System.out.println("Error");

		// Se cambia el tipo en la posición "pos" en la tabla de simbolos
		else
			tablaSimb.get(pos).tipo = n_t;
	}

	/** Metodo que regresa el tipo de una posicion dada en la tabla
	 * @param pos
	 * @return el tipo gurdado en la posicion indicada
	 */
	public Integer indicaTipo(int pos) {
		// Verifica, si el entero "pos" se sale del arreglo, regresa un -1
		if (pos >= tablaSimb.size() || pos < 0)
			return -1;

		// Regresa el tipo indicado en esa posicion
		return tablaSimb.get(pos).tipo;
	}

	/** Metodo que regresa el valor de una posicion en la tabla
```

```
         * @param pos
         * @return el valor guardado en la posicion indicada
         */
        public String indicaValor(int pos) {
                // Verifica, si el entero "pos" se sale del arreglo, regresa la cadena
                // vacia
                if (pos >= tablaSimb.size() || pos < 0)
                        return "";

                // Regresa el valor indicado en esa posicion
                return tablaSimb.get(pos).valor;
        }

        /** Metodo que imprime la tabla
         */
        public void imrpimeTS() {
                // Imprime el contenido de la tabla de simbolos
                System.out.println("Contenido de la tabla de simbolos: \n");
                for (int i = 0; i < tablaSimb.size(); i++)
                        System.out.println(tablaSimb.get(i).toString());
        }
}
```

_____Clase Constantes_____

```
/**
* COMPILADORES Toral Maldonado Rosa Guadalupe. 2153045948
*/
public class Constantes {

        public static final int NINGUNO = -1;

        public static final char EOS = '\0';

        // Tokens lexicográficos
        public static final int PROGRAMA = 256;
        public static final int COMIENZA = 257;
        public static final int TERMINA = 258;
        public static final int HECHO = 259;
        public static final int ASIGNACION = 260;
        public static final int ID = 261;
        public static final int NUM_ENT = 262;
        public static final int CADENA = 263;
        public static final int SI = 264;
        public static final int ENTONCES = 265;
        public static final int OTRO = 266;
        public static final int MIENTRAS = 267;
        public static final int HAZ = 268;
        public static final int IMPRIME = 269;
        public static final int DUMP = 270;
        public static final int HALT = 271;
        public static final int REPITE = 272;
        public static final int HASTA = 273;
        public static final int PARA = 274;
        public static final int A = 275;
```

```
/** Metodo que indica qué tpo de palabra reservada es el numero que se le pasa por
ro
 * @param constante
 * @return una cadena con el valor correspondiente al numero recibido
 */
public static String regresaValor(int constante) {
        switch (constante) {
        case NINGUNO:     return "NINGUNO";
        case PROGRAMA:          return "PROGRAMA";
        case COMIENZA:          return "COMIENZA";
        case TERMINA:     return "TERMINA";
        case HECHO:       return "HECHO";
        case ASIGNACION:   return "ASIGNACION";
        case ID:                return "ID";
        case NUM_ENT:     return "NUM_ENT";
        case CADENA:            return "CADENA";
        case SI:                return "SI";
        case ENTONCES:     return "ENTONCES";
        case OTRO:              return "OTRO";
        case MIENTRAS:     return "MIENTRAS";
        case HAZ:               return "HAZ";
        case IMPRIME:      return "IMPRIME";
        case DUMP:              return "DUMP";
        case HALT:        return "HALT";
        case REPITE:      return "REPITE";
        case HASTA:       return "HASTA";
        case PARA:        return "PARA";
        case A:                 return "A";
```

```
// Símbolos utilizados por el emisor
public static final int VALOR_I = 901;
public static final int VALOR_D = 902;
public static final int PUSH = 903;
public static final int ASIGN = 904;
public static final int LEE = 905;
public static final int ESCRIBE = 906;
public static final int ETIQUETA = 908;
public static final int VE_A = 909;
public static final int SI_V_VE_A = 910;
public static final int SI_F_VE_A = 911;
```

```
// Tokens lexicográficos y valores para operadores
public static final int RELOP = 300;
public static final int LT = 301;
public static final int LE = 302;
public static final int EQ = 303;
public static final int GE = 304;
public static final int GT = 305;
public static final int NE = 306;
public static final int ADDOP = 400;
public static final int MAS = 401;
public static final int MENOS = 402;
public static final int OR = 403;
public static final int MULOP = 500;
public static final int MULT = 501;
public static final int DIV = 502;
public static final int MODULO = 503;
public static final int AND = 504;
```

```
case A:             return "A";
case RELOP:     return "RELOP";
case LT:            return "LT";
case LE:            return "LE";
case EQ:            return "EQ";
case GE:            return "GE";
case GT:            return "GT";
case NE:            return "NE";
case ADDOP:     return "ADDOP";
case MAS:           return "MAS";
case MENOS:     return "MENOS";
```

```
case OR:            return "OR";
case MULOP:     return "MULOP";
case MULT:      return "MULT";
case DIV:           return "DIV";
case MODULO:    return "MODULO";
case AND:           return "AND";
}
return "";
    }
}
```

Clase Principal

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

import javax.swing.JOptionPane;

/** COMPILADORES
* Toral Maldonado Rosa Guadalupe.
* 2153045948
*/
public class Principal {
        // Variables que permiten manejar archivos
        private static File archivoError, archivoObj;
        public static BufferedWriter bwError, bwObj;

        public static void main(String[] args){
                Tabla_de_Simbolos tabla = new Tabla_de_Simbolos();
                BufferDeEntrada buffer = new BufferDeEntrada();
                String archivo = "prueba.txt";
                buffer.identificaF(archivo);
```

```
			creaArchivos(archivo);
			AnalizadorSintactico parser = new AnalizadorSintactico(buffer, tabla);
			parser.programa();
			try {
				bwError.close();
				bwObj.close();
				JOptionPane.showMessageDialog(null, "Archivos creados");
			} catch (IOException e) {
				System.out.println("No se pudieron cerrar los archivos");
			}
		}

		/** Metodo que permite crear los archivos
		 */
		private static void creaArchivos(String direccion) {
			String nombre = "";
			for (int i = 0; i < direccion.length() - 4; i++)
				nombre = nombre + direccion.charAt(i);
			archivoError = new File(nombre + ".err");
			archivoObj = new File(nombre + ".obj");
			try {
				bwError = new BufferedWriter(new FileWriter(archivoError));
				bwObj = new BufferedWriter(new FileWriter(archivoObj));
			} catch (IOException e) {
				System.out.println("No se puede escribir en los archivos");
			}
		}
}
```