Trabajo Práctico Especial
Diseño de Compiladores

Grupo 18

Adán Matías Diaz Graziano

Esteban Boroni

Temas Asignados:
1 6 7 10 11 13 19 22 23 27 28

Analizador Léxico

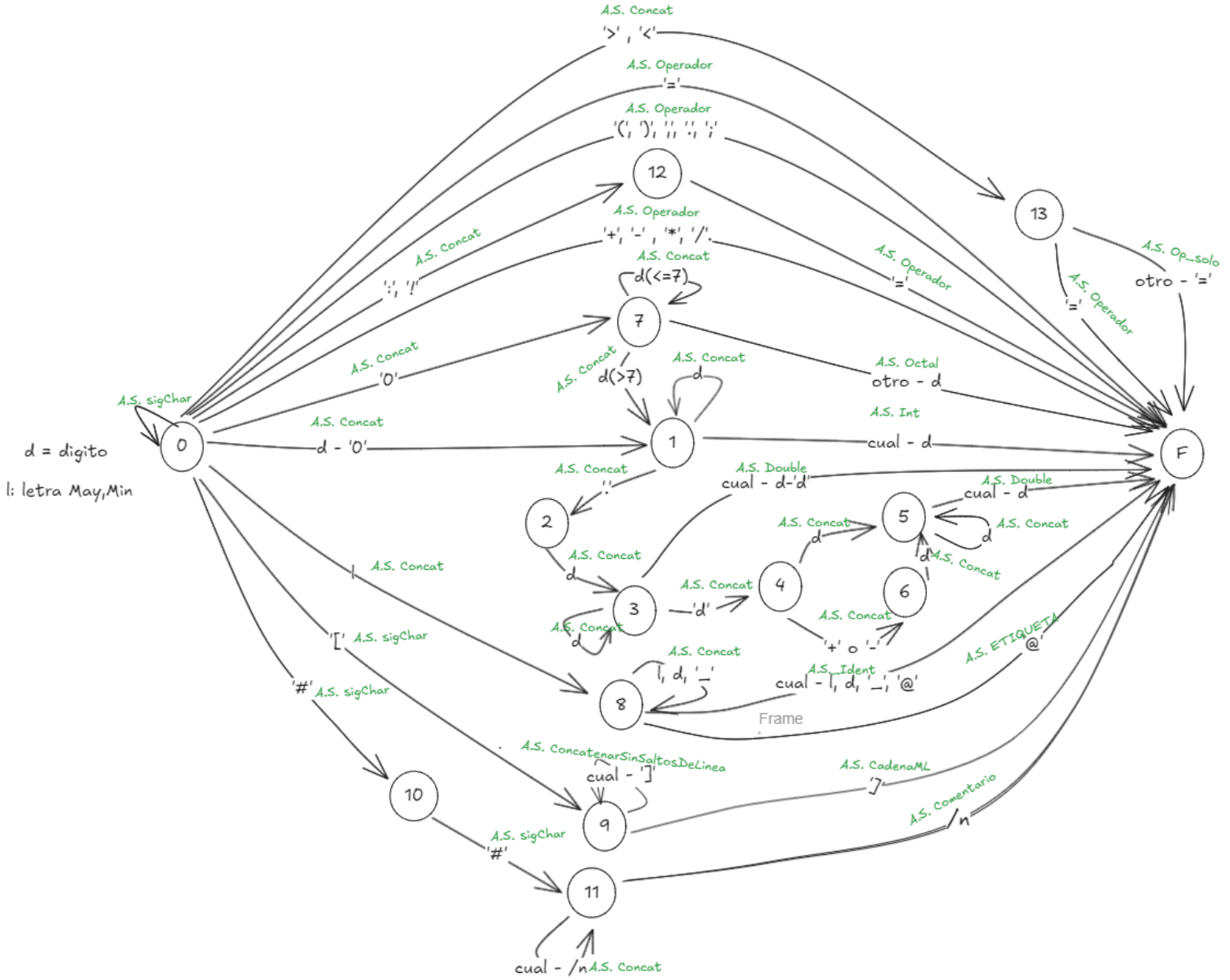
Decisiones de Diseño e Implementación

Nuestra implementación está basada principalmente en una clase Analizador Léxico que devuelve tokens cada vez que logra generar uno, aislando la identificación y creación de estos a las Acciones Semánticas.
De esta manera, en el Analizador Léxico únicamente nos encargamos de identificar en qué estado estamos, a qué estado vamos y qué Acción Semántica hay que ejecutar según el caracter leído.
Por otra parte, en las Acciones Semánticas nos encargamos de saber que hay que hacer con el caracter e identificamos si, junto a los caracteres previos, se generó un TOKEN; si hay que seguir leyendo o si es un ERROR.
En el caso de que el TOKEN identificado no esté en la Tabla de Palabras Reservadas, cada vez que se identifica uno lo va a cargar en la Tabla de Símbolos. En nuestro caso la tabla es un map<> con el token como key y una clase Símbolo con toda su información.
Para la Tabla de Palabras Reservadas utilizamos un Map<String, short> con su TOKEN junto al valor definido por el YACC. De esta manera, mediante la el nombre del TOKEN devolvemos el numero que lo representa en el YACC.
Símbolo es una clase en la cual cada tipo de token puede almacenar la información que necesita para la etapa de Análisis Sintáctico. Por ejemplo, le dimos un gran uso a la hora de agregar nuevamente a la Tabla de Símbolos las constantes negativas con el signo '-', ya que, si no teníamos un contador de referencia, las demás instancias de esa misma constante no iban a tener su entrada a la Tabla de Símbolos, o peor, si existía una constante igual a ella pero positiva nunca más iba a estar cargada.
Por último, la clase CargadorDeMatriz se encarga de cargar la matriz de Acciones Semánticas, la matriz de Estados y cargar la Tabla de Palabras Reservadas.

Para un mejor seguimiento, adjuntamos el diagrama de clases correspondiente a esta etapa del compilador.

URL al UML para mayor legibilidad: Diagrama_UML_Compilador.png

Diagrama de Transición de Estados



Matriz de Transición de Estados

URL a hoja de cálculo de google para mayor legibilidad: Matriz Estados

	0	1-7	8-9	L - 'd'	'd'	','	'+' '-'	'*' '/'	'_'	'@'	'['	']'	'#'	'/n'	':' '!'	'='	'(' ')' ';' '{' '}'	'>' '<'	'.'	OTRO
0	7/AS.Concat	1/AS.Concat	1/AS.Concat	8/AS.Concat	8/AS.Concat	F/AS.Operador	F/AS.Operador	F/AS.Operador	0/AS.ERROR	0/AS.ERROR	9/AS.SigChar	0/AS.ERROR	10/AS.SigChar	0/sigChar	12/AS.Concat	F/AS.Operador	F/AS.Operador	13/AS.Concat	F/AS.Operador	0/AS_SIgChar
1	1/AS.Concat	1/AS.Concat	1/AS.Concat	F/AS.Int	F/AS.Int	F/AS.Int	F/AS.Int	F/AS.Int	F/AS.Int	F/AS.Int	F/AS.Int	F/AS.Int	F/AS.Int	F/AS.Int	F/AS.Int	F/AS.Int	F/AS.Int	F/AS.Int	2/AS.Concat	F/AS.Int
2	3/AS.Concat	3/AS.Concat	3/AS.Concat	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR
3	3/AS.Concat	3/AS.Concat	3/AS.Concat	F/AS.Double	4/AS.Concat	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double
4	5/AS.Concat	5/AS.Concat	5/AS.Concat	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	6/AS.Concat	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR
5	5/AS.Concat	5/AS.Concat	5/AS.Concat	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double	F/AS.Double
6	5/AS.Concat	5/AS.Concat	5/AS.Concat	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR
7	7/AS.Concat	7/AS.Concat	1/AS.Concat	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal	F/AS.Octal
8	8/AS.Concat	8/AS.Concat	8/AS.Concat	8/AS.Concat	8/AS.Concat	F/AS.Ident	F/AS.Ident	F/AS.Ident	8/AS.Concat	F/AS.Etiqueta	F/AS.Ident	F/AS.Ident	F/AS.Ident	F/AS.Ident	F/AS.Ident	F/AS.Ident	F/AS.Ident	F/AS.Ident	F/AS.Ident	F/AS.Ident
9	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	F/AS.CadenaML	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin	9/AS.ConcatSalLin
10	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	11/AS.SigChar	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR
11	11/AS.Concat	11/AS.Concat	11/AS.Concat	11/AS.Concat	11/AS.Concat	11/AS.Concat	11/AS.Concat	11/AS.Concat	11/AS.Concat	11/AS.Concat	11/AS.Concat	11/AS.Concat	11/AS.Concat	F/AS.Comentario	11/AS.Concat	11/AS.Concat	11/AS.Concat	11/AS.Concat	11/AS.Concat	11/AS.Concat
12	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	F/AS.Operador	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR	0/AS.ERROR
13	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	F/AS.Operador	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo	0/AS_Op_solo

Mecanismo de Implementación de Matrices

Para la implementación de las Matrices se crearon las siguientes instancias en la clase AnalizadorLexico.

Tabla de Simbolos

```
public static Map<String, Simbolo> TablaDeSimbolos = new
HashMap<>();
```

Para la TablaDeSimbolos se utilizó un Map<String , Simbolo> ya que nuestra idea fue a partir del TOKEN obtenido por la Accion Semantica generar una entrada en esta tabla únicamente si no era una palabra reservada. Esto nos llevó a que en las Acciones Semánticas también se envíe justo al TOKEN el LEXEMA para poder acceder a la ubicación en la Tabla Símbolos de dicho TOKEN.
Este LEXEMA es una constante del Analizador Lexico que cuando se envía el TOKEN se verifica si es diferente a null, en caso de que lo sea, se utiliza su valor para asignarse a la variable `yylval` perteneciente al YACC.

Tabla de Palabras Reservadas

```
private static final TablaPalabrasReservadas
PalabrasReservadas = new
TablaPalabrasReservadas("resources\\PalabrasReservadas.txt");
```

Para la TablaPalabrasReservadas se creó una clase ya que creíamos en un principio que íbamos a necesitar agregar más funcionalidad pero, al avanzar en el desarrollo, se fue sacando funcionalidad y nos dimos cuenta que era innecesaria la clase. Por lo tanto, su único fin es almacenar una entrada por cada palabra reservada existente y devolver el número que representa a cada token.

```
   public TablaPalabrasReservadas(String ruta) {
     PALABRA_NO_RESERVADA= -1;
     ARCHIVO_PALABRAS_RESERVADAS=ruta;

palabras_reservadas=CargadorDeMatriz.CrearMapDeArch(ARCHIVO_PA
LABRAS_RESERVADAS);
   }
```

```
   public static Map<String, Short> CrearMapDeArch(String
path) {
     Map<String, Short> map = new HashMap<>();
       try (BufferedReader br = new BufferedReader(new
FileReader(path))) {
           String linea;
           while ((linea = br.readLine()) != null) {
               String key = linea.trim();
```

```
                short Num = Short.parseShort(br.readLine());
                map.put(key, Num);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return map;
    }
```

Matriz de Acciones Semanticas

```
private static final AccionSemantica[][] accionesSemanticas =
CargadorDeMatriz.CargarMatrizAS(ARCH_MATRIZ_ACCIONES,
CANTIDAD_ESTADOS, CANTIDAD_CARACTERES);
```

Para la Acciones Semánticas se utilizó una Matriz de AccionesSemanticas y dos constantes que indican el tamaño de la Matriz. Esta se cargó con la clase `CargadorDeMatriz`, que lee un archivo txt y por cada linea de texto crea una instancia de la Accion Semantica que corresponde.

```
  public static AccionSemantica[][] CargarMatrizAS(String
path, int filas, int columnas){
        AccionSemantica[][] matriz = new
AccionSemantica[filas][columnas];
        try (BufferedReader br = new BufferedReader(new
FileReader(path))) {
            String nombreAccion;
            int fila = 0, columna = 0;
            while ((nombreAccion = br.readLine()) != null) {
                AccionSemantica accion =
crearAcSem(nombreAccion);
                matriz[fila][columna] = accion;  // Asigna la
acción a la matriz
                columna++;
                if (columna == columnas) {
                    columna = 0;
                    fila++;
                }
                if (fila == filas) {
                    break;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return matriz;
```

```
    }
    private static AccionSemantica crearAcSem(String
accion_semantica) {
      switch (accion_semantica) {
        case "AS_CadenaML":
            return new AS_CadenaML();
        case "AS_Comentario":
            return new AS_Comentario();
        case "AS_Concatenas":
            return new AS_Concatenas();
        case "AS_ConcatenasSinSaltoDeLinea":
            return new AS_ConcatenasSinSaltoDeLinea();
        case "AS_Double":
            return new AS_Double();
        case "AS_ERROR":
            return new AS_ERROR();
        case "AS_ETIQUETA":
            return new AS_ETIQUETA();
        case "AS_Identificador":
            return new AS_Identificador();
        case "AS_Int":
            return new AS_Int();
        case "AS_Octal":
            return new AS_Octal();
        case "AS_Operador":
            return new AS_Operador();
        case "AS_SIgChar":
            return new AS_SIgChar();
        case "AS_Op_solo":
            return new AS_Op_solo();
        default:
           System.out.println("'" + accion_semantica + "'" + "
da null ");
            return null;
    }
      }
```

Matriz de Transición de Estados

```
private static final int[][] transicion_estados =
CargadorDeMatriz.CargarMatrizEstados(ARCH_MATRIZ_ESTADOS,
CANTIDAD_ESTADOS, CANTIDAD_CARACTERES);
```

Para esta se realizó un proceso similar a la de Acciones Semánticas pero con números enteros.

```
   public static int[][] CargarMatrizEstados(String
nombreArchivo, int filas, int columnas) {
       int[][] matriz = new int[filas][columnas];
       try (BufferedReader br = new BufferedReader(new
FileReader(nombreArchivo))) {
           String linea;
           int fila = 0, columna = 0;
           while ((linea = br.readLine()) != null) {
               matriz[fila][columna] =
Integer.parseInt(linea.trim());
               columna++;
               if (columna == columnas) {
                   columna = 0;
                   fila++;
               }
               // Si ya se han leído todas las filas y
columnas, salir
               if (fila == filas) {
                   break;
               }
           }
       } catch (IOException e) {
           e.printStackTrace();
       }
       return matriz;
   }
```

Acciones Semánticas

AccionSemantica

```
TOKEN_ACTIVO = -1;
ERROR = 280;
public abstract Short ejecutar(char, Reader, token, PalabrasReservadas, TablaDeSimbolos);

public void cargarSalida(String in) {
		try {
			AnalizadorLexico.salida.newLine();  // Agregar un salto de lÃnea
			AnalizadorLexico.salida.write(" "+in+" ");
		} catch (IOException e) {
			e.printStackTrace();
		}
}
```

Es la clase abstracta de la que heredan todas las acciones semánticas.
Posee un método “Ejecutar” el cual devolverá en cada caso el token reconocido por cada acción semántica particular. Este recibirá la tabla de símbolos, la tabla de palabras reservadas, el token stringbuilder en el cual se va leyendo y anidando el lexema, el Reader que se encarga de la lectura del código y un caracter que se usa en algunas acciones semánticas para retornar el código ASCII de algunos caracteres individuales.

El método cargar salida se usa para escribir la salida en un archivo txt como lo indica el enunciado.

Además del método, todas poseen un atributo cuyo valor indica que el token leído es un error y otro atributo que indica que aún no se finalizó de identificar el token y está aún en proceso.

AS_CadenaML

```
public Short ejecutar(char car, Reader lector, StringBuilder token, TablaPalabrasReservadas PalabrasReservadas, Map<String,
Simbolo> TablaDeSimbolos) {
                if (!TablaDeSimbolos.containsKey(token.toString())) { //Si no contiene el lexema en la TS
                        CREA SIMBOLO;
                        AGREGA A LA TABLA;
                        PASA ID AL PARSER;
                }
                PRINT;
                cargarSalida("Cadena multilinea " + token.toString());
                 AnalizadorLexico.token_actual.setLength(0); //VACIAMOS EL BUFFER
                 return PalabrasReservadas.obtenerIdentificador("CADENAMULTILINEA"); //devolvemos el token correspondiente
 };
```

Se utiliza para identificar únicamente a las cadenas multilínea.

La acción chequea que no exista una key en la tabla de símbolos con el lexema actual del token. Si no existe dicha key, se crea un símbolo con el lexema, se lo agrega a la tabla y se le envía el token al parser mediante su cargado en el atributo ‘lexema’ de la clase AnalizadorLéxico.

Posterior al chequeo, se imprime el valor obtenido, se vacía el buffer para leer un nuevo token desde cero y se retorna el número que identifica a la cadena multilínea en la tabla de palabras reservadas.

AS_Comentario

Únicamente se imprime el comentario, se vacía el buffer y se devuelve que el token está activo.

No es necesario guardar nada por ser un comentario.

AS_Concatenas

Únicamente concatena el último carácter leído al token que va construyendo el lexema. Devuelve el token activo.

AS_ConcatenasSinSaltoDeLinea

```
public Short ejecutar(char car, Reader lector, StringBuilder token, TablaPalabrasReservadas
PalabrasReservadas, Map<String, Simbolo> TablaDeSimbolos) {
    //Si es un salto de linea pasa al siguiente.
        if(car!=AnalizadorLexico.SALTO_DE_LINEA) {
        token.append(car);  //Concatena si no es salto de línea
    }
    return TOKEN_ACTIVO;
 };
```

Revisa que el carácter leído no sea un salto de línea (\n) y si no lo es entonces concatena el carácter al token. De esta manera no tenemos en cuenta los saltos de línea para la formación de la cadena multilinea.
Luego se devuelve el token activo.

AS_Double

```
public Short ejecutar(char, Reader, token, PalabrasReservadas, TablaDeSimbolos){

    String tokenString = token.toString().replace('d', 'E');
    try {
            // Convertir el token a un valor double
            Double tokenDouble = Double.parseDouble(tokenString);
            // Verificar si el valor cumple con la condicion
            if ((tokenDouble < 1.7976931348623157E+308)
                    || (tokenDouble > 2.2250738585072014E-308)
                            || (tokenDouble == cero)) {
                    if(no está en la TS){
                            Agrega a la TS
                            Le paso el ID al Parser
                    else { incremento el contador de referencias si existe en la TS }
            else{
                    Si no está en rango retorna error y repite el último char}
    catch {
            cargarSalida(Error: número invalido)}
    repite ultimo char;
    vacía buffer;
    retorna ID asociado a CTE;
}
```

Primero convertimos el lexema del token siendo leído a Double dado que leemos siempre un String. En esta conversión reemplazamos la ‘d’ que indica el exponente a una ‘E’ que es la manera que usa Java para indicar el exponente y así poder comparar correctamente.
Esta conversión de la ‘d’ a la ‘E’ fue una sugerencia del Chat GPT.

Esta acción semántica (y las acciones que detectan enteros y octales) posee un método “cumple”, el cual se encarga de verificar que la constante leída esté dentro del rango definido.

Si el Double leído está dentro del rango, se chequea si hay ya una entrada con el mismo valor en la tabla de símbolos. Si no la hay se crea el símbolo, se agrega y se pasa la ID al parser.
Si ya hay una entrada con un mismo lexema entonces se incrementa el contador de referencia que posee la constante.

Si la constante no está dentro del rango entonces se devuelve que es un error y se imprime por pantalla.

Utilizamos el atributo SEREPITE del analizador léxico para indicar que se vuelva a leer el último carácter leído y no se descarte en el proceso ocasionando algún error en la lectura del próximo token.
Si todo fue satisfactorio se devuelve el identificador de una constante.

AS_ERROR

Únicamente vacía el buffer con la lectura hasta el momento ya que es un error, se notifica y se devuelve el atributo error.

AS_ETIQUETA

Similar a las anteriores, si no existe una entrada en la tabla de símbolos, crea el símbolo, lo agrega y pasa el ID al parser.
Luego se imprime, se vacía el buffer con el lexema actual para leer un nuevo token y se devuelve el valor asociado al token Etiqueta.

AS_Identificador

```
public class AS_Identificador extends AccionSemantica {
        public Short ejecutar(char car, Reader lector, StringBuilder token, TablaPalabrasReservadas
PalabrasReservadas, Map<String, Simbolo> TablaDeSimbolos) {
            // En caso de superar los 15 caracteres se trunca
           String tokenString = truncar(token.toString());
            short salida;
            // verifica que no sea una palabra reservada
            if(PalabrasReservadas.obtenerIdentificador(tokenString)== -1){ //Si NO es palabra
reservada
                  //Si no lo es ve si ya existe en la tabla de simbolos
                  if(!TablaDeSimbolos.containsKey(tokenString)){
                        Simbolo simb = new Simbolo();
                        simb.setId(tokenString);
                        TablaDeSimbolos.put(tokenString,simb);
                        AnalizadorLexico.Lexema = tokenString;  //LE PASO EL ID A LA TABLA DE
SIMBOLOS AL PARSER.
                  }
                  //si ya existe solo la devuelve
                  salida=PalabrasReservadas.obtenerIdentificador("ID");
            }else {
                  salida=PalabrasReservadas.obtenerIdentificador(tokenString);
```

```
            }
                  cargarSalida("Identificador "+tokenString);
            //Se repite el ultimo caracter leido
             AnalizadorLexico.SEREPITE=true;
             AnalizadorLexico.token_actual.setLength(0); //VACIAMOS EL BUFFER YA QUE SE ESPERA UN
NUEVO TOKEN
             return salida; //devolvemos el token correspondiente
         }
            private String truncar(String tokenString) {
                  String token =tokenString;
                  if(token.length() > 15) {
                        token=token.substring(0, 15);
                        System.out.println("\u001B[31m"+"Linea "+ AnalizadorLexico.saltoDeLinea +"
WARNING: El identificador '"
                                                            + tokenString + "' fue truncado a '"+token+
"' y este podria reconocerse como palabra reservada."+"\u001B[0m");
                  }
                  return token;
            }

      }
```

Esta acción semántica tiene un método que chequea la cantidad de caracteres del token y lo trunca en caso de superar los 15 y notifica.

El método “ejecutar” chequea que el lexema no corresponda a alguna de las palabras reservadas de la lista y si no es el caso, hace el chequeo de la tabla de símbolos y lo agrega.
Al final retrocedemos y hacemos que se vuelva a leer el último caracter previamente leído, se vacía el buffer y se retorna el token correspondiente, ya sea un ID común o una palabra reservada.

AS_Int

```
public Short ejecutar(char car, Reader lector, StringBuilder token, TablaPalabrasReservadas PalabrasReservadas, Map<String,
Simbolo> TablaDeSimbolos) {
    long tokenlong = Long.valueOf(token.toString());
    if(cumple(tokenlong)) {
      if(!TablaDeSimbolos.containsKey(token.toString())){
              Agrego a la TS
              Paso la ID al Parser
      }else {TablaDeSimbolos.get(token.toString()).incrementarContDeRef();}
      AnalizadorLexico.Lexema = token.toString();  //LE PASO EL ID A LA TABLA DE SIMBOLOS AL PARSER.
    }else {
      System.out.println("Error lexico en la linea " + AnalizadorLexico.saltoDeLinea+" : Constante entera fuera de rango ");
              cargarSalida("Error lexico en la linea " + AnalizadorLexico.saltoDeLinea+" : Constante entera fuera de rango ");
      AnalizadorLexico.SEREPITE=true;
       return ERROR;
    }
    System.out.println("Constante entera "+tokenlong);
    cargarSalida("Constante entera "+tokenlong);
    SEREPITE=true;
   VACIAMOS EL BUFFER YA QUE SE ESPERA UN NUEVO TOKEN
    //retornar el key
    return PalabrasReservadas.obtenerIdentificador("CTE");
 };
```

Como se mencionó anteriormente, todas las acciones semánticas referidas a constantes poseen un método de chequeo de rangos.

El método “ejecutar” convierte el lexema de String a Long, usamos Long específicamente para permitir mayor cantidad de valores aunque luego no estén dentro del rango.
Si está dentro de los rangos se hace el chequeo de la tabla de símbolos como en las demás acciones semánticas. En caso de ya existir el lexema en la tabla entonces se incrementa el contador de referencias de dicho símbolo.

En el final, hacemos releer el último caracter leído, vaciamos el buffer y devolvemos el identificador del token.

AS_Octal

```
if(!TablaDeSimbolos.containsKey(token.toString())){
                        Simbolo simb = new Simbolo(8);
                        simb.setEntero(tokenint);
                        TablaDeSimbolos.put(token.toString(),simb);
                        AnalizadorLexico.Lexema = token.toString();//LE PASO
EL ID A LA TABLA DE SIMBOLOS AL PARSER.
                    }
```

Es muy similar a la anterior, la principal diferencia es que realizamos la conversión de String a Integer pero especificamos en la conversión que la base es 8.
Otra diferencia referido a esto es, en caso de agregar el lexema a la tabla de símbolos, especificar la base 8 en el constructor para poder diferenciarlo de un entero normal.

El chequeo de rangos es igual al de un entero ya que al realizar la conversión de Octal a Entero entonces los rangos serán iguales.

AS_Op_solo

```
public Short ejecutar(char, Reader, token, PalabrasReservadas,TablaDeSimbolos) {
        SEREPITE=true;
           switch (token.toString()) {
                case ">":
                        AnalizadorLexico.token_actual.setLength(0); //VACIAMOS EL BUFFER
                        cargarSalida(">");
                        return (short)'>';
                case "<":
                        AnalizadorLexico.token_actual.setLength(0); //VACIAMOS EL BUFFER
                        cargarSalida(" <");
                        return (short)'<';
                default:
                        cargarSalida(token.toString());
                        AnalizadorLexico.token_actual.setLength(0); //VACIAMOS EL BUFFER
                        return Short.valueOf(token.toString());
        }}
```

Esta acción semántica indica que hay que releer el último caracter para evitar errores y luego se hace un case para verificar si se trata de un ‘<’, ‘>’ u otro por default.
Sea cual sea el caso, se vacía el buffer con el token, se carga la salida y se retorna el código ASCII correspondiente al caracter leído mediante su conversión de Char a Short/Integer.

ACLARACION: Tuvimos que dividir las Acciones Semanticas asi porque los operadores simples se reconocen como caracter y los compuestos (pj: :=) como String.

AS_Operador

En esta se concatena el caracter leído al token actual, se carga en la salida y luego se realiza un case similar al de la acción semántica anterior.

AS_SigChar

Solamente retorna que se trata de un token activo. Sirve para descartar caracteres como el espacio en blanco y que no se carguen en el token.

Analizador Sintáctico

Gramática Generada

En primera instancia creamos una gramática simple y general en la cual abarcamos toda la funcionalidad del compilador sin realizar testeos sobre esta.
Al momento de ejecutarla mediante el YACC, nos alertó de una enorme cantidad de conflictos shift/reduce, reduce/reduce y varias reglas sin reducir.
Para poder realizar una comprobación de errores y conflictos fue necesario generar el autómata en cada compilación de la gramática y se debió realizar un proceso incremental para localizar los errores.
La ambigüedad de la gramática fue corregida en primera instancia, previo a la implementación de reglas que identifiquen errores, las cuales volvieron a generar los conflictos mencionados anteriormente.

Para solucionar muchos de estos conflictos fue necesario realizar un manejo prudencial de ciertos tokens como los ' **;** ' para que no se solapen y también que sirviera para que no haya conflictos en las reglas.

También se agregaron reglas a No Terminales que quizá parezcan redundantes pero posibilitaron que no se generen conflictos shift/reduce y reduce/reduce por lo tanto fueron de gran utilidad para nosotros.

Para su funcionamiento , en Gramatica.y colocamos esta funcion para el retorno del token junto al lexema.

```
int yylex() {
    int tokenSalida = AnalizadorLexico.getToken();
    yylval = new ParserVal(AnalizadorLexico.Lexema);
    if(tokenSalida==0) {
        return
AnalizadorLexico.siguienteLectura(AnalizadorLexico.archivo_original,'
');
    }
    return tokenSalida;
}
```

Adicional a esto, para el funcionamiento de algunas reglas explicadas abajo se utilizaron los siguientes métodos y variables. Estas utilizan métodos de la clase símbolo para manipular la TABLA DE SIMBOLOS.

```
private static boolean RETORNO = false;

private static void cambioCTENegativa(String key) {
    String keyNeg = "-" + key;
    if (!AnalizadorLexico.TablaDeSimbolos.containsKey(keyNeg)) {
```

```
        AnalizadorLexico.TablaDeSimbolos.put(keyNeg,
AnalizadorLexico.TablaDeSimbolos.get(key).getCopiaNeg());
    }
    AnalizadorLexico.TablaDeSimbolos.get(keyNeg).incrementarContDeRef();
    System.out.println("Linea " + AnalizadorLexico.saltoDeLinea +  " se
reconocio token negativo ");
    // es ultimo ya decrementa
    if (AnalizadorLexico.TablaDeSimbolos.get(key).esUltimo()) {
        AnalizadorLexico.TablaDeSimbolos.remove(key);
    }
}
private static boolean estaRango(String key) {
    if (AnalizadorLexico.TablaDeSimbolos.get(key).esEntero()) {
        if (!AnalizadorLexico.TablaDeSimbolos.get(key).enRangoPositivo(key))
{
                if (AnalizadorLexico.TablaDeSimbolos.get(key).esUltimo()) {
                    AnalizadorLexico.TablaDeSimbolos.remove(key);
                }
            yyerror("\u001B[31m"+"Linea " +"Linea " +
AnalizadorLexico.saltoDeLinea + " Error: " +key +" fuera de
rango."+"\u001B[0m");
            return false;
        }
    }
    return true;
}
```

Lista de No Terminales

En esta sección detallaremos cada No Terminales

Cadena

```
cadena   : CADENAMULTILINEA {System.out.println("Linea " +
AnalizadorLexico.saltoDeLinea + ": cadena multilinea ");}
;
```

Este No Terminal se reduce mediante el procesamiento del token **CADENAMULTILINEA**.

En primera instancia habíamos considerado la siguiente regla '[' CADENAMULTILINEA ']' hasta que notamos que el propio Analizador Léxico devolvería el token CADENAMULTILINEA y por lo tanto esto sería erróneo de esta manera y pudimos corregirlo.

CTE_con_sig

```
CTE_con_sig : CTE {if(estaRango($1.sval)) { $$.sval = $1.sval; } }
            | '-' CTE { cambioCTENegativa($2.sval); $$.sval = "-" +
$2.sval;}
            | ERROR
            | '-' ERROR
;
```

Este No Terminal es reducido mediante 4 posibles reglas.

- **CTE:** Reconoce el token CTE y a partir de aquí se realiza nuevamente el chequeo del rango de este. Esto debido a que si la constante era 32768 y resultó ser positiva, entonces se excedió del rango y debe ser eliminada de la tabla de símbolos.

  Esta funcion “estaRango” chequea en primera instancia que se trate de un valor Entero (Integer u Octal) ya que en estos dos se da la particularidad de que si el valor negativo es el mínimo posible, al considerarlo positivo se iría de rango. En Double los límites son iguales pero con signo opuesto así que no ocurriría.
  Si se trata de un Entero, se comprueba que esté dentro del rango (ya sea Integer u Octal) y si no lo está se llama a la función “esUltimo” que, en caso de ser positiva, se elimina la key de la tabla de símbolos. Si es negativa, no se elimina la entrada a la tabla y únicamente se decrementa el contador de referencia de dicha constante.
  Esta implementación es necesaria ya que múltiples apariciones del mismo lexema de una constante en el código estarán asociados al mismo símbolo de la TS.

```
private static boolean estaRango(String key) {
    if (AnalizadorLexico.TablaDeSimbolos.get(key).esEntero()) {
        if (!AnalizadorLexico.TablaDeSimbolos.get(key).enRangoPositivo(key)) {
            if (AnalizadorLexico.TablaDeSimbolos.get(key).esUltimo()) {
                AnalizadorLexico.TablaDeSimbolos.remove(key);
            }
            yyerror("Imprime el error"
            return false;
        }
    }
    return true;
}
```

```
public boolean esUltimo() {
    if(this.contadorDeReferencias==1) {
        return true;
    }else {
        this.contadorDeReferencias--;
        return false;
    }
}
```

- **'-' CTE:** Si se reconoce que se trata de una constante negativa, se concatena el signo negativo al String de la constante. Luego se chequea si ya existe una entrada en la TS con el mismo lexema. Si existe, se incrementa el contador de referencias. Si no existe entonces la clase Símbolo crea una copia negativa de sí mismo y se agrega a la TS.
  Nuevamente ante una posible eliminación de un elemento en la TS, se verifica que

sea el último y sino únicamente se decrementa el contador de referencias.

```
public boolean enRangoPositivo(String id) {
    int auxint;
    double mayor = 32767;
    if (this.base == 8) {
        auxint = Integer.parseInt(id, 10);
    }else {auxint=this.entero;}
    if(auxint<= mayor) {
        return true;
    }
    return false;
}
public void decrementarContDeRef() {this.contadorDeReferencias--;}
public void incrementarContDeRef() {this.contadorDeReferencias++;}

public Simbolo getCopiaNeg() {
    if(this.esEntero()) {
        return new Simbolo(this.entero*-1,this.doub,this.base);
    }
    return new Simbolo(this.entero,this.doub*-1.0,this.base);
}
```

```
private static void cambioCTENegativa(String key) {
    String keyNeg = "-" + key;
    if (!AnalizadorLexico.TablaDeSimbolos.containsKey(keyNeg)) {
        AnalizadorLexico.TablaDeSimbolos.put(keyNeg,
AnalizadorLexico.TablaDeSimbolos.get(key).getCopiaNeg());
    }

AnalizadorLexico.TablaDeSimbolos.get(keyNeg).incrementarContDeRef(
);
    System.out.println("Linea " + AnalizadorLexico.saltoDeLinea +
" se reconocio token negativo ");
    // es ultimo ya decrementa
    if (AnalizadorLexico.TablaDeSimbolos.get(key).esUltimo()) {
        AnalizadorLexico.TablaDeSimbolos.remove(key);
    }
}
```

- Las otras dos reglas las usamos para acaparar posibles errores.

ID_Simple

```
ID_simple : ID
;
```

El ID_simple es únicamente un ID.

variable_simple

```
variable_simple : ID_simple
```

```
;
```

La variable_simple es un ID_simple, se implementó esto que pareciera ser redundante e innecesario pero al hacerlo se solucionaron muchos conflictos shift/reduce y reduce/reduce. Por lo tanto en nuestra gramática es muy necesario.

variables

```
variables   : variables ',' variable_simple
            | variable_simple
;
```

Las variables puede ser una sola variable únicamente o una lista de variables separadas por coma. La lista de variables es usada únicamente en la declaración de variable por si se quiere declarar más de una.
De momento no pudimos implementar el error de falta de la coma entre las variables.

factor

```
factor  : variable_simple
        | CTE_con_sig
        | invocacion
        | variable_simple '{' CTE '}'
;
```

El factor puede ser una constante positiva o negativa, una variable, una invocación a una función o la posición de un arreglo. Para la posición del arreglo se tiene en cuenta el token CTE dado que sería erróneo intentar acceder con un índice negativo.

termino

```
termino : termino '*' factor
        | termino '/' factor
        | factor
;
```

El término es simplemente un factor o una multiplicación/división entre términos y factores. Con esta recursividad podemos concatenar muchas multiplicaciones/divisiones.

expresion_arit

```
expresion_arit  : expresion_arit '+' termino
                | expresion_arit '-' termino
                | termino
                | error {System.out.println("La expresion está mal
escrita);}
;
```

La expresión aritmética funciona de la misma manera que un término pero ahora con la suma y la resta. De esta manera conseguimos precedencia entre estos sin usar comandos como %left.
Además usamos el token error para considerar casos mal escritos y no falle la compilación.

list_expre

```
list_expre  : list_expre ',' expresion_arit
            | expresion_arit
;
```

Las lista de expresiones concatenan expresiones aritméticas separadas por coma y también considera el caso en el que no haya más de una expresión. En dicho caso, la lista de expresiones será una expresión aritmética individual.

invocacion

```
invocacion  : ID_simple '(' expresion_arit ')'
            | ID_simple '(' tipo_primitivo '(' expresion_arit ')' ')'
            | ID_simple '(' ')' {System.out.println("Error: Falta el
parametro real en la invocacion a funcion);}
;
```

La invocación es simplemente un ID_simple (nombre de la función que se está invocando), seguida de una expresión aritmética encerrada entre paréntesis.
Nosotros tenemos como tema particular la conversión en invocación a función. Para poder implementarlo no pudimos simplemente anteponer un “tipo” a la expresión aritmética ya que, como se verá más adelante en la declaración de un tipo, puede tratarse de un ID el tipo y causar muchos conflictos.
Para poder contemplar la conversión, sin causar conflictos ni errores en la gramática, tomamos la consideración que únicamente se podrá convertir a tipos primitivos (INTEGER y DOUBLE) y de esa manera funciona sin errores. Según nuestra gramática, nuestro compilador no permitirá una conversión a un tipo definido por el usuario.
Por último, tenemos una regla para contemplar el caso en que falte el parámetro real en la invocación y así permitir la compilación del error.

asignacion

```
asignacion  : variable_simple ASIGNACION expresion_arit
            | variable_simple '{' CTE '}' ASIGNACION expresion_arit
;
```

Una asignación tiene dos posibles reglas. En cualquiera de las dos, del lado izquierdo de la asignación debe haber una variable_simple y del lado derecho una expresión aritmética.

La diferencia es que una contempla el uso de un arreglo, teniendo el nombre de este, precedido por el índice entre llaves (se usaron llaves para evitar conflictos con la cadena multilinea). Nuevamente el índice es solamente una constante positiva.

outf_rule

```
outf_rule     : OUTF '(' expresion_arit ')'
              | OUTF '(' ')' {System.out.println("Error");}
              | OUTF '(' cadena ')'
              | OUTF '(' sentencias ')' {System.out.println("Error");}
;
```

La sentencia OUTF solo puede permitir cadenas multilíneas o expresiones aritméticas.
Si recibe un parámetro vacío se lo considera como regla pero imprimiremos el error correspondiente.
Tenemos en cuenta como error que reciba por parámetro un conjunto de sentencias. Si aceptamos sentencias entonces acapararemos cualquier otra cosa que no sea una expresión aritmética o una cadena y así contemplaremos todos los casos posibles.

sentencia_ejecutable

```
sentencia_ejecutable     : asignacion
                         | sentencia_IF
                         | sentencia_WHILE
                         | sentencia_goto
                         | ETIQUETA
                         | outf_rule
                         | retorno {RETORNO = true;}
;
```

Si se lee un retorno, este pone en TRUE una variable global declarada en la Gramatica.y que es utilizada para saber si dentro de una funcion se llamo un retorno.

bloque_sentencia_simple

```
bloque_sentencia_simple: sentencia_ejecutable
;
```

Se crea este bloque de sentencias simple a partir de una sentencia ejecutable para formar parte de los cuerpos de los IF y WHILE.

bloque_sent_ejecutables

```
bloque_sent_ejecutables : bloque_sent_ejecutables ';'
bloque_sentencia_simple
                        | bloque_sentencia_simple
;
```

El bloque de sentencias ejecutables será una concatenación de los simples o únicamente uno de estos.

bloque_unidad_simple

```
bloque_unidad_simple:  bloque_sentencia_simple
;
```

El bloque de unidad simple es solamente un bloque de sentencia simple. Esto se usa para no tener que rodear de BEGIN END a una sola sentencia, como lo indica el enunciado.
De esta manera en el cuerpo de un IF o WHILE puede haber una sola sentencia y no llevará BEGIN END.

bloque_unidad_multiple

```
bloque_unidad_multiple  : BEGIN bloque_sent_ejecutables END
;
```

Como se detalló en el anterior caso, aquí buscamos justamente rodear de BEGIN END un conjunto de sentencias ejecutables.

bloque_unidad

```
bloque_unidad   : bloque_unidad_simple
                | bloque_unidad_multiple
;
```

El bloque unidad es el bloque principal utilizado en sentencias IF y WHILE. Se compone de un bloque de unidad simple (sin BEGIN END) y un bloque múltiple.

bloque_else_simple

```
bloque_else_simple: ELSE bloque_sentencia_simple
;
```

El bloque else antepone el token del mismo nombre a un bloque de sentencia simple. Se usará únicamente en el IF y se asegurará nuevamente que una única sentencia no tenga que estar rodeada de BEGIN END.

bloque_else_multiple

```
bloque_else_multiple:   ELSE BEGIN bloque_sent_ejecutables END
;
```

Lo mismo que se explicó con anterioridad y anteponiendo el token ELSE al resto de la regla.

bloque_else

```
bloque_else: bloque_else_simple
```

```
            | bloque_else_multiple
;
```

El bloque else es solamente una de las dos opciones, ya sea con una sola sentencia a ejecutar o con más de una posible sentencia a ejecutar.

comparador

```
comparador  : '>'
            | MAYORIGUAL
            | '<'
            | MENORIGUAL
            | '='
            | DISTINTO
;
```

El comparador es simplemente alguno de los comparadores, ya sea individuales o combinados.

condicion

```
condicion   : '(' '(' list_expre ')' comparador '(' list_expre ')' ')'
            | '(' expresion_arit comparador expresion_arit ')'
            | '(' list_expre ')' comparador '(' list_expre ')' ')' {Println("Error");}
            | '(' '(' list_expre ')' comparador '(' list_expre ')' {Println("Error");}
            |'(' list_expre ')' comparador '(' list_expre ')' {Println("Error");}
            |  expresion_arit comparador expresion_arit ')' {Println("Error");}
            | '(' expresion_arit comparador expresion_arit  {Println("Error");}
            | expresion_arit comparador expresion_arit  {Println("Error");}
;
```

En una condición tenemos como casos correctos la comparación de lista de expresiones entre paréntesis y la comparación de expresiones simples.

El resto de las reglas fueron agregadas para considerar los casos de paréntesis faltantes en alguna de las reglas correctas de arriba.

El único error que no pudo ser implementado sin causar conflictos es el faltante del comparador.

sentencia_IF

```
sentencia_IF: IF condicion THEN bloque_unidad ';' bloque_else ';' END_IF
            | IF condicion THEN bloque_unidad ';' END_IF
            | IF condicion THEN bloque_unidad ';' {Println("Error");}
            | IF condicion THEN bloque_unidad ';' bloque_else ';' {Println("Error");}
            | IF condicion THEN bloque_else ';' END_IF {Println("Error");}
```

```
        | IF condicion THEN END_IF {Println("Error");}
        | IF condicion THEN bloque_unidad ';' ELSE END_IF {Println("Error");}
;
```

La sentencia IF está compuesta por el token del mismo nombre, una condición de las previamente explicadas y luego el token THEN que anticipa a un bloque_unidad.
Dado nuestro manejo del **;** con las sentencias, es necesario agregar uno al finalizar el bloque unidad.
Posterior a eso puede venir un END_IF que indica que se puede reducir de manera correcta o esperar el bloque del else. En cualquiera de los casos la sentencia se cierra con el END_IF.

Luego de esto contemplamos todos los errores posibles, ya sean faltantes de bloques o faltantes del END_IF.

retorno

```
retorno : RET '(' expresion_arit ')'
;
```

El retorno es una sentencia simple que utiliza el token especial RET, seguido de una expresión entre paréntesis. Representa el retorno de las funciones.

cuerpo_funcion

```
cuerpo_funcion  : sentencias
;
```

El cuerpo de una función son sentencias de cualquier tipo (ejecutables o declarativas).

parametro

```
parametro   : tipo ID_simple
;
```

El parámetro en sí es un tipo (ID o tipo primitivo) seguido de un ID_simple que representa a una variable. Se usa de esta manera para corregir conflictos en la gramática.

parametros_parentesis

```
parametros_parentesis: '(' parametro ')'
                     | '(' ')' {Println("Error");}
                     | '(' error ')' {Println("Error");}
;
```

Este No Terminal se usa para encasillar con paréntesis a los parámetros y de esa manera también poder contemplar errores con el token error. Fue una solución encontrada ante la

intención de utilizar el token error y poseer los paréntesis dentro de la declaración de función.
Al bajar los paréntesis directamente con el parámetro pudimos corregir estos errores y asegurar el funcionamiento.

Uno de los errores es el caso de que se invoque a una función sin pasar parámetros reales. Esto surge a partir de un entendimiento nuestro del enunciado. Entendemos que las funciones solo deben tener un parámetro, ni más ni menos.

El token error se usa para encasillar posibles errores en el parámetro de una función.

declaracion_funciones

```
declaracion_funciones     : tipo FUN ID parametros_parentesis BEGIN
cuerpo_funcion END {if(RETORNO==false){System.out.println(Erro: Faltan
el RETORNO de al funcion);RETORNO=false;}
                        | tipo FUN parametros_parentesis BEGIN
cuerpo_funcion END
{if(RETORNO==false){System.out.println("\u001B[31m"+"Linea " +
AnalizadorLexico.saltoDeLinea + ": Erro: Faltan el RETORNO de al
funcion "+"\u001B[0m");RETORNO=false;}
System.out.println("\u001B[31m"+"Linea " +
AnalizadorLexico.saltoDeLinea + ": Erro: Faltan el nombre en la funcion
"+"\u001B[0m");}
;
```

A partir de la variable global RETORNO explicada anteriormente se chequea si se leyó un retorno dentro de la funcion, si no fue asi, imprime un error indicando este problema.
Como la segunda regla se contempla si existe nombre en la declaracion de la funcion.

tipo_primitivo

```
tipo_primitivo: INTEGER
                |DOUBLE
;
```

Se creó la separación de tipo_primitivo para poder utilizarlo en la invocación a función con conversión de tipo. De esta manera podemos separar lo que es un tipo definido por el usuario de un tipo primitivo y aplicarlos por separado.

tipo

```
tipo : ID_simple
       |tipo_primitivo
;
```

```
```

El tipo es simplemente un tipo primitivo o un tipo definido por el usuario (en nuestro caso subtipo o triple). La manera de identificar un tipo definido por el usuario es mediante un ID_simple.
Este No Terminal nos causaba muchos conflictos al intentar usarlo para la conversión. Los conflictos eran por poseer en la regla ID_simple.

declaracion_variable

```
declaracion_variable    : tipo variables  {System.out.println("Linea "
+ AnalizadorLexico.saltoDeLinea + " declaracion de variables ");}
;
```

La declaración de variables es solamente un tipo seguido de una lista de variables. Esta lista de variables puede ser una variable sola.

sentencia_declarativa

```
sentencia_declarativa   : declaracion_variable
                        | declaracion_funciones
                        | declaracion_subtipo
;
```

Una sentencia declarativa son 3 posibles reglas. Ya sea el declarar una variable, una función o un subtipo.

sentencia

```
sentencia   : sentencia_declarativa
            | sentencia_ejecutable
;
```

Una sentencia puede ser tanto declarativa como ejecutable.

sentencias

```
sentencias  : sentencias sentencia ';'
            | sentencia ';'
            | sentencia {Error: Falta ';' al final de la sentencia}
;
```

Aquí en las sentencias concatenamos multiples sentencias, separadas por **;** .
Al final de cada sentencia agregamos un **;** y será un error si no ocurre esto.

programa

```
programa     : ID_simple BEGIN sentencias END
             | BEGIN sentencias END {Falta el nombre del programa}
             | ID_simple BEGIN sentencias {Falta el delimitador END}
             | ID_simple sentencias END {Falta el delimitador BEGIN}
             | ID_simple sentencias {Faltan los delimitadores del}
;
```

Un programa tiene un ID_simple (nombre del programa) seguido de un conjunto de sentencias delimitadas por un BEGIN END. Será error casos con faltantes de estos.

Temas Particulares

Subtipos y Tripe

```
declaracion_subtipo : TYPEDEF ID_simple ASIGNACION tipo '{' CTE_con_sig
','  CTE_con_sig '}'
                    | TYPEDEF ID_simple ASIGNACION tipo CTE_con_sig ','
CTE_con_sig '}'
                    | TYPEDEF ID_simple ASIGNACION tipo '{' CTE_con_sig
',' CTE_con_sig
                    | TYPEDEF ID_simple ASIGNACION tipo CTE_con_sig ','
CTE_con_sig
                    | TYPEDEF TRIPLE '<' tipo '>' ID_simple
                    | TYPEDEF ID_simple ASIGNACION tipo '{' ','
CTE_con_sig '}'
                    | TYPEDEF ID_simple ASIGNACION tipo '{' CTE_con_sig
'}'
                    | TYPEDEF ID_simple ASIGNACION tipo '{' CTE_con_sig
',' '}'
                    | TYPEDEF ID_simple ASIGNACION tipo '{' '}'
                    |TYPEDEF ASIGNACION tipo '{' CTE_con_sig ','
CTE_con_sig '}'
                    | TYPEDEF ID_simple ASIGNACION '{' CTE_con_sig ','
CTE_con_sig '}'
                    | TYPEDEF '<' tipo '>' ID_simple
                    | TYPEDEF TRIPLE tipo '>' ID_simple
                    | TYPEDEF TRIPLE '<' tipo ID_simple
                    | TYPEDEF TRIPLE tipo ID_simple
                    | TYPEDEF TRIPLE '<' tipo '>' error
;
```

En estas reglas se declaran los subtipos y los tipo triples.

En las demás reglas se consideran los posibles errores, tales como:

- Faltan el '{' , '}' o ambos en el rango
- Falta rango inferior, superior o ambos.
- Falta de nombre en el tipo definido
- Falta el tipo base en la declaracion de subtipo
- Falta de la palabra reservada TRIPLE
- Falta del '<','>' o ambos en TRIPLE
- Falta identificador al final de la declaracion

WHILE

```
sentencia_WHILE : WHILE condicion bloque_unidad
                | WHILE condicion error {println(Error);}
;
```

El While se compone del token del mismo nombre, una condición a cumplir y un bloque unidad. De esta manera el bloque puede ser una sola sentencia o muchas sentencias delimitadas por BEGIN END.
La faltante del bloque unidad será un error y por eso usamos este token en la otra regla.

Pattern Matching

```
condicion   : '(' '(' list_expre ')' comparador '(' list_expre ')' ')'
{"Condicion con lista de expresiones ");}
            | '(' list_expre ')' comparador '(' list_expre ')' ')'
{"Error : Falta el '(' en la condicion "}
            | '(' '(' list_expre ')' comparador '(' list_expre ')'
{Falta el ')' en la condicion}
            |'(' list_expre ')' comparador '(' list_expre ')'{Faltan
los parentesis en la condicion "}
```

El pattern matching fue considerado dentro de la condición con la comparación de las listas de expresiones.
Faltantes en los paréntesis son consideradas reglas posibles a reducir pero declaradas como error.

GOTO

```
sentencia_goto  : GOTO ETIQUETA
                | GOTO error  {Error: Falta la etiqueta en GOTO "}
;
```

La sentencia GOTO es el token del mismo nombre, seguido del token ETIQUETA devuelto por el analizador léxico.
La faltante de la etiqueta es un error entonces para poder realizarlo tuvimos que usar el token provisto por la herramienta.

Conversiones en Parámetros

```
invocacion  : ID_simple '(' expresion_arit ')'
            | ID_simple '(' tipo_primitivo '(' expresion_arit ')' ')'
            | ID_simple '(' ')' {System.out.println("Error: Falta el
parametro real en la invocacion a funcion);}
;
```

Para poder contemplar la conversión, sin causar conflictos ni errores en la gramática, tomamos la consideración que únicamente se podrá convertir a tipos primitivos (INTEGER y DOUBLE) y de esa manera funciona sin errores. Según nuestra gramática, nuestro compilador no permitirá una conversión a un tipo definido por el usuario.

Errores considerados

Léxicos

- Constantes enteras, octales y flotantes fuera de rango.
    - En este caso en las enteras únicamente se verificó que sean menor que el valor impuesto por el rango negativo, ya que de esta manera no perdemos la constante negativa y en la parte sintáctica verificamos que si es positivo, debe verificarse el rango nuevamente. En el caso de ser negativa, se decrementa el contador de referencia de la tabla de símbolos o se borra la entrada si es el unico y se vuelve a cargar pero con la nueva key siendo ahora negativa. De esta manera evitamos que la misma constante o una de signo opuesto deje de tener referencia en la tabla de símbolos al borrarla.
    - Para los Octales lo pasamos a entero y realizamos el mismo calculo.
    - Para el flotante como no hay diferencia entre los rangos negativos y positivos se chequea solamente en esta etapa.
- Cuando un estado del autómata recibe un caracter q no coincide con ninguno de los arcos de salida, lo devuelve como un token ERROR identificado por el YACC como 280.
- Un caracter inválido de entrada es devuelto con el TOKEN ERROR tambien.

Sintácticos

- Falta de nombre de programa.
- Falta de delimitador de programa.
- Falta de “;” al final de las sentencias.
- Falta de nombre en función.
- Falta de sentencia RET en función.
- Falta de nombre de parámetro formal en declaración de función.
- Falta de tipo de parámetro formal en declaración de función.
- Cantidad errónea de parámetros en declaración o invocación de función.
- Falta parámetro en sentencia OUTF. Parámetro incorrecto en sentencia OUTF.
- Falta de paréntesis en condición de selecciones e iteraciones.
- Falta de cuerpo en iteraciones..
- Falta de END_IF.
- Falta de contenido en bloque THEN/ELSE.
- Falta de operando en expresión.
- Falta de operador en expresión.
- Tema 11: Falta de [] en rango. Falta de rango. Falta nombre del tipo definido. Falta el tipo base.
- Tema 19: Falta de paréntesis externos.
- Tema 22: Falta triple. Falta <>. Falta identificador al final de la declaración.
- Tema 23: Falta etiqueta.

Consideraciones, Inconvenientes Encontrados y Solucionados

- Los **resultados** se imprimen por pantalla y solo se cargan en un archivo txt los léxicos, el sintáctico se nos complicó porque nos manejabamos con prints y no los almacenabamos en ningun lado.
- No tuvimos en cuenta la declaración de la misma CTE y esto nos generó que al borrar en el Sintáctico la variable fuera de rango también se borre la referencia a todas las que tenían el mismo nombre. Para solucionarlo agregamos en Símbolo un contador de referencia que si se identificaba una CTE negativa en el Sintáctico solo borra su anterior referencia (la positiva) si ella es la única, sino decrementa su contador de referencias.
- Para verificar si está en rango le damos importancia únicamente a los enteros ya que con los double el rango superior e inferior son iguales.
- No sabíamos la existencia del autómata de pila así que la gramática al inicio tuvo muchos problemas. Se soluciona reiniciando la gramática, ir probando de a pocas reglas iniciando por las declaraciones y asignaciones y por último, añadiendo múltiples reglas para generar precedencia.
- Al finalizar el compilador nos dimos cuenta que la clase Símbolo que hicimos es muy similar a ParserVal y pudimos utilizarla directamente a ella, agregándole funcionalidades extra.
- En principio al crear el WHILE para el bloque THEN y ELSE utilizabamos sentencias, esto generaba conflicto al identificar los bloque de sentencias del THEN y ELSE. Para solucionarlo bajamos los tokes THEN y ELSE para los bloques de sentencia creando reglas nuevas.
- Al principio no encontramos la forma de realizar conversiones a tipos creados por el usuario porque nos generaba conflicto con otras reglas que utilizaban el token ID. (pj: una variables declarada de un tipo triple y realizar la conversión a triple). Nos dimos cuenta que no tenía mucho sentido realizar esa conversión entonces, para solucionarlo hicimos que las conversiones explícitas realizadas en invocaciones a función sólo serán de tipos primitivos (INTEGER,DOUBLE,OCTAL) y no se podrá realizar con subtipos declarados por el usuario.
- Entendemos según el enunciado que las funciones solo pueden tener un único parámetro.

Casos de prueba

Explicacion del codigo	Codigo.

CASO DE PRUEBA 1:
Sin errores.
	`prog`
	`BEGIN`
	`integer varx, vary, varx;`
	`varx := 327;`
	`varx := 2.3d+34;`
	`tipo_abc vary;`
	`integer FUN juancito ( double a ) BEGIN`
	`WHILE (2768 > -5.8)OUTF(1);`
	`WHILE ((a,3+4)=(b,35))BEGIN OUTF(1);`
	`OUTF([Hola Mundo]); goto afuera@ END;`
	`a:=funcion(b);`
	`a:=funcion(integer (a+b));`
	`a:=4+5;`
	`IF((a,3+4)=(b,35))THEN A:=1;END_IF;`
	`IF(-32768 > 5.8)THEN A:=1;ELSE`
	`B:=0;END_IF;`
	`RET (a);`
	`END;`
	`TYPEDEF TRIPLE < integer > tint;`
	`TYPEDEF flotadito := integer {-6, -8};`
	`flotadito a;`
	`tint b;`
	`b{1} := 8;`
	`a := b{1};`
	`RET (0);`
	`afuera@;`
	`a := 1 +2;`
	`END`

CASO DE PRUEBA 2:
● Falta de nombre de programa.
● CTEs fuera y dentro de rango
● Falta de “;” al final de las sentencias.
● Falta de nombre en función.
● Falta de sentencia RET en función.
● Falta de nombre de parámetro formal en declaración de función.
● Falta de tipo de parámetro formal en declaración de función.
● Cantidad errónea de parámetros en declaración o invocación de función.
● Falta parámetro en sentencia OUTF. Parámetro incorrecto en sentencia OUTF.
	`BEGIN`
	`adan := 20`
	`integer adancsadjesaea, boro32, andr_ea;`
	`boro := 32767;`
	`boro := -32768;`
	`boro := 32768;`
	`boro := -32769;`
	`boro := 3.0d-5;`
	`boro := -2.3d+34;`
	`boro := 2.3d+34;`
	`adan boro;`
	`integer FUN () BEGIN`
	`WHILE 32768 > -5.8 OUTF(1);`
	`WHILE ((a,3+4)=(b,35))BEGIN OUTF();`
	`OUTF([Hola Mundo]); goto afuera@ END;`
	`WHILE ((a,3+4)=(b,35)) ;`
	`a:=funcion();`
	`a:=funcion(integer (a+b));`
	`a:=4+5;`
	`IF((a,3+4)=(b,35))THEN A:=1;END_IF;`
	`IF -32768 > 5.8 THEN A:=1;ELSE`
	`B:=0;END_IF;`
	`IF((a,3+4)=(b,35))THEN A:=1;;`
	`IF((a,3+4)=(b,35))THEN A:=1; ELSE`
	`END_IF;`
	`END;`
	`TYPEDEF TRIPLE < integer > tint;`
	`TYPEDEF flotadito := integer {-6, -8};`
	`flotadito a;`
	`tint b;`
	`b{1} := 8;`

• Falta de paréntesis en condición de selecciones e iteraciones. • Falta de cuerpo en iteraciones.. • Falta de END_IF. //Hay que poner ; • Falta de contenido en bloque THEN/ELSE.	a := b{1}; RET (0); afuera@; a := 1 +2; END

CASO DE PRUEBA 3:
• Falta de delimitador de programa.
• Falta de operando en expresión.
• Falta de operador en expresión.
• Tema 11: Falta de [] en rango. Falta de rango. Falta nombre del tipo definido. Falta el tipo base.
• Tema 19: Falta de paréntesis externos.
• Tema 22: Falta triple. Falta <>. Falta identificador al final de la declaración.
• Tema 23: Falta etiqueta.
	prog
	integer adanc, boro, andrea;
	VARX := 1 5;
	VARX := 1 + ;
	VARX := + 5 ;
	TYPEDEF flotadito := integer {-6, -8};
	TYPEDEF flotadito := integer -6, -8;
	TYPEDEF flotadito := integer {-6,};
	TYPEDEF flotadito := integer {, -8};
	TYPEDEF flotadito := integer {};
	TYPEDEF := integer {-6, -8};
	TYPEDEF flotadito := {-6, -8};
	IF(a,3+4)=(b,35)THEN A:=1;END_IF;
	IF((a,3+4)=(b,35)THEN A:=1;END_IF;
	IF(a,3+4)=(b,35))THEN A:=1;END_IF;
	TYPEDEF < integer > tint;
	TYPEDEF TRIPLE integer tint;
	TYPEDEF TRIPLE < integer > ;
	goto ;
	END

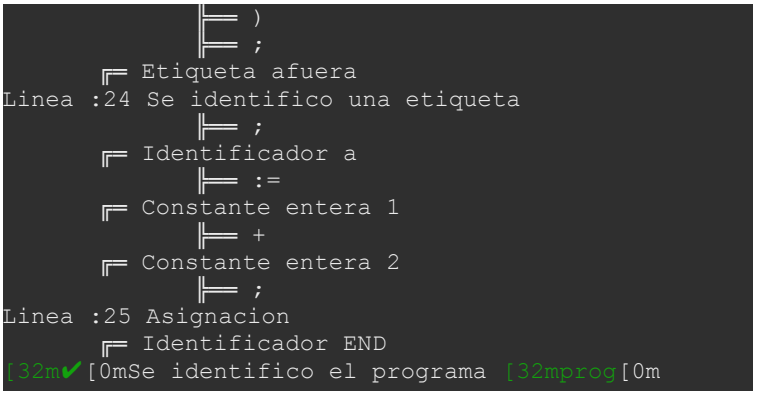
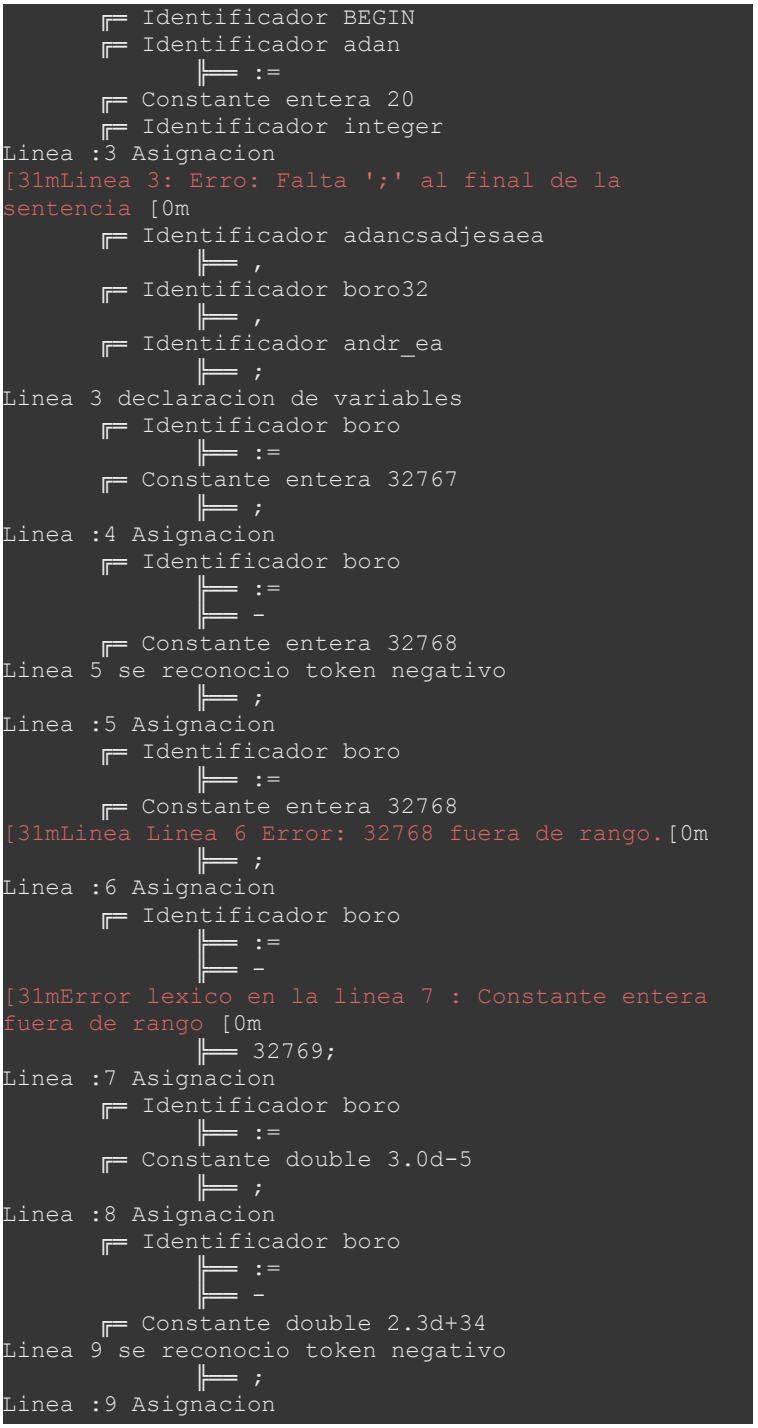
Resultados

Codigos	Resultados.
CASO DE PRUEBA 1: Sin errores.	╔═ Identificador prog ╔═ Identificador BEGIN ╔═ Identificador integer ╔═ Identificador varx ╠═ , ╔═ Identificador vary ╠═ , ╔═ Identificador varx ╠═ ; Linea 3 declaracion de variables ╔═ Identificador varx ╠═ := ╔═ Constante entera 327 ╠═ ; Linea :4 Asignacion ╔═ Identificador varx ╠═ := ╔═ Constante double 2.3d+34 ╠═ ;

```
Linea :5 Asignacion
       ╔═ Identificador tipo_abc
       ╔═ Identificador vary
              ╠══ ;
Linea 6 declaracion de variables
       ╔═ Identificador integer
       ╔═ Identificador FUN
       ╔═ Identificador juancito
              ╠══ (
       ╔═ Identificador double
       ╔═ Identificador a
              ╠══ )
       ╔═ Identificador BEGIN
       ╔═ Identificador WHILE
              ╠══ (
       ╔═ Constante entera 2768
              ╠═══ >
              ╠═══ -
       ╔═ Constante double 5.8
Linea 8 se reconocio token negativo
              ╠══ )
Linea 8: Condicion
       ╔═ Identificador OUTF
              ╠══ (
       ╔═ Constante entera 1
              ╠══ )
Linea :8 Se reconocio OUTF de Expresion Aritmetica
Linea 8: Se identifico un WHILE
              ╠══ ;
       ╔═ Identificador WHILE
              ╠═══ (
              ╠═══ (
       ╔═ Identificador a
              ╠══ ,
       ╔═ Constante entera 3
              ╠══ +
       ╔═ Constante entera 4
              ╠═══ )
              ╠═══ =
              ╠═══ (
       ╔═ Identificador b
              ╠══ ,
       ╔═ Constante entera 35
              ╠═══ )
              ╠═══ )
Linea 9: Condicion con lista de expresiones
       ╔═ Identificador BEGIN
       ╔═ Identificador OUTF
              ╠══ (
       ╔═ Constante entera 1
              ╠══ )
Linea :9 Se reconocio OUTF de Expresion Aritmetica
              ╠══ ;
       ╔═ Identificador OUTF
              ╠══ (
       ╔═ Cadena multilinea Hola Mundo
Linea 9: cadena multilinea
              ╠══ )
Linea :9 Se reconocio OUTF de cadena de caracteres
              ╠══ ;
       ╔═ Identificador goto
       ╔═ Etiqueta afuera
Linea 9: Sentencia GOTO
       ╔═ Identificador END
Linea 9: Se identifico un WHILE
              ╠══ ;
       ╔═ Identificador a
              ╠══ :=
```

```
      ╒═ Identificador funcion
            ╞══ (
      ╒═ Identificador b
            ╞══ )
Linea :10 Invocacion a funcion
            ╞══ ;
Linea :10 Asignacion
      ╒═ Identificador a
            ╞══ :=
      ╒═ Identificador funcion
            ╞══ (
      ╒═ Identificador integer
            ╞══ (
      ╒═ Identificador a
            ╞══ +
      ╒═ Identificador b
            ╞══ )
            ╞══ )
Linea :11 Invocacion con conversion
            ╞══ ;
Linea :11 Asignacion
      ╒═ Identificador a
            ╞══ :=
      ╒═ Constante entera 4
            ╞══ +
      ╒═ Constante entera 5
            ╞══ ;
Linea :12 Asignacion
      ╒═ Identificador IF
            ╞══ (
            ╞══ (
      ╒═ Identificador a
            ╞══ ,
      ╒═ Constante entera 3
            ╞══ +
      ╒═ Constante entera 4
            ╞══ )
            ╞══ =
            ╞══ (
      ╒═ Identificador b
            ╞══ ,
      ╒═ Constante entera 35
            ╞══ )
            ╞══ )
Linea 13: Condicion con lista de expresiones
      ╒═ Identificador THEN
      ╒═ Identificador A
            ╞══ :=
      ╒═ Constante entera 1
            ╞══ ;
Linea :13 Asignacion
      ╒═ Identificador END_IF
Linea 13: Sentencia IF
            ╞══ ;
      ╒═ Identificador IF
            ╞══ (
            ╞══ -
      ╒═ Constante entera 32768
Linea 14 se reconocio token negativo
            ╞══ >
      ╒═ Constante double 5.8
            ╞══ )
Linea 14: Condicion
      ╒═ Identificador THEN
      ╒═ Identificador A
            ╞══ :=
      ╒═ Constante entera 1
            ╞══ ;
```

```
Linea :14 Asignacion
       ╔═ Identificador ELSE
       ╔═ Identificador B
              ╠══ :=
       ╔═ Constante entera 0
              ╠══ ;
Linea :14 Asignacion
       ╔═ Identificador END_IF
Linea 14: Sentencia IF
              ╠══ ;
       ╔═ Identificador RET
              ╠══ (
       ╔═ Identificador a
              ╠══ )
              ╠══ ;
       ╔═ Identificador END
Linea 16 declaracion de Funcion
              ╠══ ;
       ╔═ Identificador TYPEDEF
       ╔═ Identificador TRIPLE
              ╠══ <
       ╔═ Identificador integer
              ╠══ >
       ╔═ Identificador tint
Linea 17 declaracion de Triple
              ╠══ ;
       ╔═ Identificador TYPEDEF
       ╔═ Identificador flotadito
              ╠══ :=
       ╔═ Identificador integer
              ╠══ {
              ╠══ -
       ╔═ Constante entera 6
Linea 18 se reconocio token negativo
              ╠══ ,
              ╠══ -
       ╔═ Constante entera 8
Linea 18 se reconocio token negativo
              ╠══ }
Linea 18 declaracion de Subtipo
              ╠══ ;
       ╔═ Identificador flotadito
       ╔═ Identificador a
              ╠══ ;
Linea 19 declaracion de variables
       ╔═ Identificador tint
       ╔═ Identificador b
              ╠══ ;
Linea 20 declaracion de variables
       ╔═ Identificador b
              ╠══ {
       ╔═ Constante entera 1
              ╠══ }
              ╠══ :=
       ╔═ Constante entera 8
              ╠══ ;
Linea :21 Asignacion a arreglo
       ╔═ Identificador a
              ╠══ :=
       ╔═ Identificador b
              ╠══ {
       ╔═ Constante entera 1
              ╠══ }
              ╠══ ;
Linea :22 Asignacion
       ╔═ Identificador RET
              ╠══ (
       ╔═ Constante entera 0
```

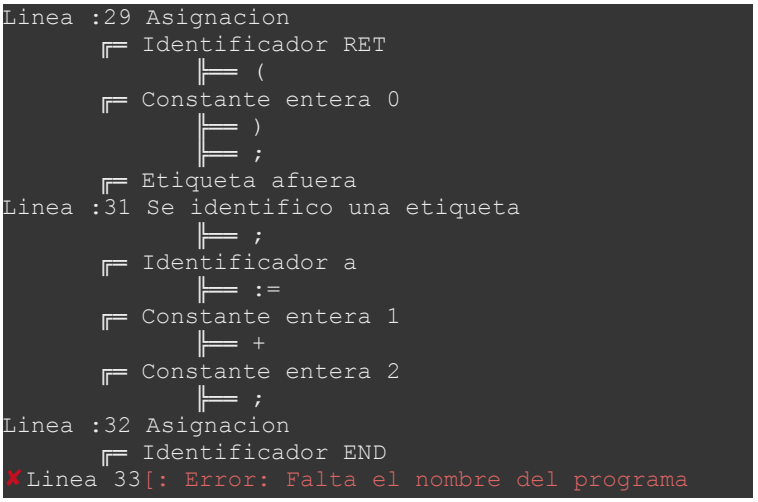
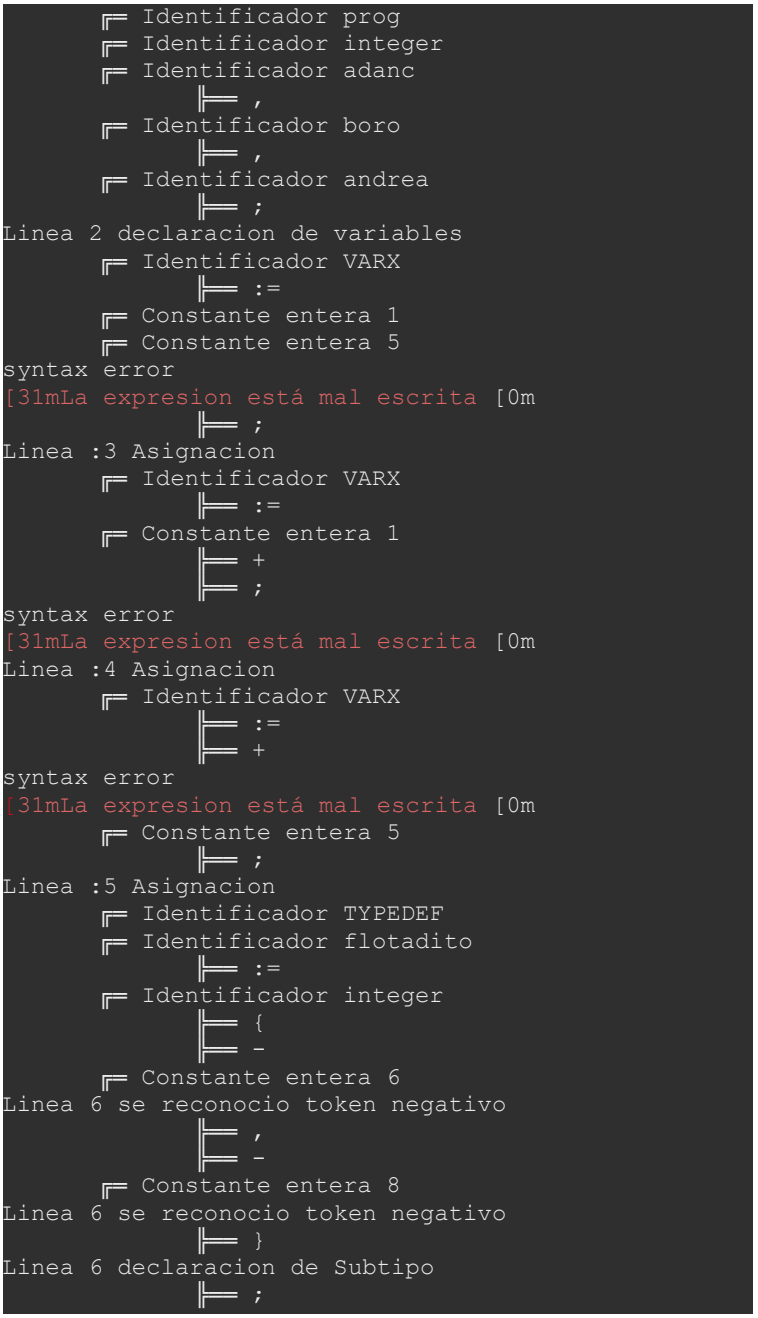
	
CASO DE PRUEBA 2: • Falta de nombre de programa. • CTEs fuera y dentro de rango • Falta de “;” al final de las sentencias. • Falta de nombre en función. • Falta de sentencia RET en función. • Falta de nombre de parámetro formal en declaración de función. • Falta de tipo de parámetro formal en declaración de función. • Cantidad errónea de parámetros en declaración o invocación de función. • Falta parámetro en sentencia OUTF. Parámetro incorrecto en sentencia OUTF. • Falta de paréntesis en condición de selecciones e iteraciones. • Falta de cuerpo en iteraciones.. • Falta de END_IF. //Hay que poner ; • Falta de contenido en bloque THEN/ELSE.	

```
       ╔═ Identificador boro
              ╠══ :=
       ╔═ Constante double 2.3d+34
              ╠══ ;
Linea :10 Asignacion
       ╔═ Identificador adan
       ╔═ Identificador boro
              ╠══ ;
Linea 11 declaracion de variables
       ╔═ Identificador integer
       ╔═ Identificador FUN
              ╠═╦ (
              ╠═╩ )
[31mLinea 12 Error: Falta el parametro en la
funcion [0m
       ╔═ Identificador BEGIN
       ╔═ Identificador WHILE
       ╔═ Constante entera 32768
[31mLinea Linea 13 Error: 32768 fuera de rango.[0m
              ╠═╦ >
              ╠═╩ -
       ╔═ Constante double 5.8
Linea 13 se reconocio token negativo
       ╔═ Identificador OUTF
[31mLinea 13Error : Faltan los parentesis en la
condicion [0m
              ╠══ (
       ╔═ Constante entera 1
              ╠══ )
Linea :13 Se reconocio OUTF de Expresion Aritmetica
Linea 13: Se identifico un WHILE
              ╠══ ;
       ╔═ Identificador WHILE
              ╠═╦ (
              ╠═╩ (
       ╔═ Identificador a
              ╠══ ,
       ╔═ Constante entera 3
              ╠══ +
       ╔═ Constante entera 4
              ╠═╦ )
              ╠═╬ =
              ╠═╩ (
       ╔═ Identificador b
              ╠══ ,
       ╔═ Constante entera 35
              ╠═╦ )
              ╠═╩ )
Linea 14: Condicion con lista de expresiones
       ╔═ Identificador BEGIN
       ╔═ Identificador OUTF
              ╠═╦ (
              ╠═╩ )
[31mLinea :14 Error : Falta el parametro del OUTF
[0m
              ╠══ ;
       ╔═ Identificador OUTF
              ╠══ (
       ╔═ Cadena multilinea Hola Mundo
Linea 14: cadena multilinea
              ╠══ )
Linea :14 Se reconocio OUTF de cadena de caracteres
              ╠══ ;
       ╔═ Identificador goto
       ╔═ Etiqueta afuera
Linea 14: Sentencia GOTO
       ╔═ Identificador END
Linea 14: Se identifico un WHILE
```

```
                ╠══ ;
        ╒═ Identificador WHILE
                ╠══ (
                ╠══ (
        ╒═ Identificador a
                ╠══ ,
        ╒═ Constante entera 3
                ╠══ +
        ╒═ Constante entera 4
                ╠══ )
                ╠══ =
                ╠══ (
        ╒═ Identificador b
                ╠══ ,
        ╒═ Constante entera 35
                ╠══ )
                ╠══ )
Linea 15: Condicion con lista de expresiones
                ╠══ ;
syntax error
[31mLinea 15 Error: falta el cuerpo del WHILE [0m
        ╒═ Identificador a
                ╠══ :=
        ╒═ Identificador funcion
                ╠══ (
                ╠══ )
[31mLinea :16 Error:  faltan los parametros reales en la invocacion[0m
                ╠══ ;
Linea :16 Asignacion
        ╒═ Identificador a
                ╠══ :=
        ╒═ Identificador funcion
                ╠══ (
        ╒═ Identificador integer
                ╠══ (
        ╒═ Identificador a
                ╠══ +
        ╒═ Identificador b
                ╠══ )
                ╠══ )
Linea :17 Invocacion con conversion
                ╠══ ;
Linea :17 Asignacion
        ╒═ Identificador a
                ╠══ :=
        ╒═ Constante entera 4
                ╠══ +
        ╒═ Constante entera 5
                ╠══ ;
Linea :18 Asignacion
        ╒═ Identificador IF
                ╠══ (
                ╠══ (
        ╒═ Identificador a
                ╠══ ,
        ╒═ Constante entera 3
                ╠══ +
        ╒═ Constante entera 4
                ╠══ )
                ╠══ =
                ╠══ (
        ╒═ Identificador b
                ╠══ ,
        ╒═ Constante entera 35
                ╠══ )
                ╠══ )
Linea 19: Condicion con lista de expresiones
```

```
      ╔═ Identificador THEN
      ╔═ Identificador A
            ╠══ :=
      ╔═ Constante entera 1
            ╠══ ;
Linea :19 Asignacion
      ╔═ Identificador END_IF
Linea 19: Sentencia IF
            ╠══ ;
      ╔═ Identificador IF
            ╠══ -
      ╔═ Constante entera 32768
Linea 20 se reconocio token negativo
            ╠══ >
      ╔═ Constante double 5.8
      ╔═ Identificador THEN
[31mLinea 20Error : Faltan los parentesis en la
condicion [0m
      ╔═ Identificador A
            ╠══ :=
      ╔═ Constante entera 1
            ╠══ ;
Linea :20 Asignacion
      ╔═ Identificador ELSE
      ╔═ Identificador B
            ╠══ :=
      ╔═ Constante entera 0
            ╠══ ;
Linea :20 Asignacion
      ╔═ Identificador END_IF
Linea 20: Sentencia IF
            ╠══ ;
      ╔═ Identificador IF
            ╠══╦ (
            ╠══╗ (
      ╔═ Identificador a
            ╠══ ,
      ╔═ Constante entera 3
            ╠══ +
      ╔═ Constante entera 4
            ╠══╝ )
            ╠═══ =
            ╠══╗ (
      ╔═ Identificador b
            ╠══ ,
      ╔═ Constante entera 35
            ╠══╝ )
            ╠══ )
Linea 21: Condicion con lista de expresiones
      ╔═ Identificador THEN
      ╔═ Identificador A
            ╠══ :=
      ╔═ Constante entera 1
            ╠══ ;
Linea :21 Asignacion
            ╠══ ;
[31mLinea 21: Error : Falta el END_IF en IF  [0m
      ╔═ Identificador IF
            ╠══╦ (
            ╠══╗ (
      ╔═ Identificador a
            ╠══ ,
      ╔═ Constante entera 3
            ╠══ +
      ╔═ Constante entera 4
            ╠══╝ )
            ╠═══ =
            ╠══ (
```

```
       ╒═ Identificador b
              ╞══ ,
       ╒═ Constante entera 35
              ╞══ )
              ╞══ )
Linea 22: Condicion con lista de expresiones
       ╒═ Identificador THEN
       ╒═ Identificador A
              ╞══ :=
       ╒═ Constante entera 1
              ╞══ ;
Linea :22 Asignacion
       ╒═ Identificador ELSE
       ╒═ Identificador END_IF
[31mLinea 22 Error : falta cuerpo en el ELSE [0m
              ╞══ ;
       ╒═ Identificador END
[31mLinea 23: Erro: Faltan el RETORNO de al funcion
[0m
[31mLinea 23: Erro: Faltan el nombre en la funcion
[0m
              ╞══ ;
       ╒═ Identificador TYPEDEF
       ╒═ Identificador TRIPLE
              ╞══ <
       ╒═ Identificador integer
              ╞══ >
       ╒═ Identificador tint
Linea 24 declaracion de Triple
              ╞══ ;
       ╒═ Identificador TYPEDEF
       ╒═ Identificador flotadito
              ╞══ :=
       ╒═ Identificador integer
              ╞══ {
              ╞══ -
       ╒═ Constante entera 6
Linea 25 se reconocio token negativo
              ╞══ ,
              ╞══ -
       ╒═ Constante entera 8
Linea 25 se reconocio token negativo
              ╞══ }
Linea 25 declaracion de Subtipo
              ╞══ ;
       ╒═ Identificador flotadito
       ╒═ Identificador a
              ╞══ ;
Linea 26 declaracion de variables
       ╒═ Identificador tint
       ╒═ Identificador b
              ╞══ ;
Linea 27 declaracion de variables
       ╒═ Identificador b
              ╞══ {
       ╒═ Constante entera 1
              ╞══ }
              ╞══ :=
       ╒═ Constante entera 8
              ╞══ ;
Linea :28 Asignacion a arreglo
       ╒═ Identificador a
              ╞══ :=
       ╒═ Identificador b
              ╞══ {
       ╒═ Constante entera 1
              ╞══ }
              ╞══ ;
```

	
CASO DE PRUEBA 3: ● Falta de delimitador de programa. ● Falta de operando en expresión. ● Falta de operador en expresión. ● Tema 11: Falta de [] en rango. Falta de rango. Falta nombre del tipo definido. Falta el tipo base. ● Tema 19: Falta de paréntesis externos. ● Tema 22: Falta triple. Falta <>. Falta identificador al final de la declaración. ● Tema 23: Falta etiqueta.	

```
      ╔═ Identificador TYPEDEF
      ╔═ Identificador flotadito
            ╠══ :=
      ╔═ Identificador integer
            ╠══ -
      ╔═ Constante entera 6
Linea 7 se reconocio token negativo
            ╠══ ,
            ╠══ -
      ╔═ Constante entera 8
Linea 7 se reconocio token negativo
            ╠══ ;
[31mLinea 7: Error: Faltan ambos '{' '}' en el
rango [0m
      ╔═ Identificador TYPEDEF
      ╔═ Identificador flotadito
            ╠══ :=
      ╔═ Identificador integer
            ╠══ {
            ╠══ -
      ╔═ Constante entera 6
Linea 8 se reconocio token negativo
            ╠══ ,
            ╠══ }
[31mLinea 8: Error: Falta rango superior [0m
            ╠══ ;
      ╔═ Identificador TYPEDEF
      ╔═ Identificador flotadito
            ╠══ :=
      ╔═ Identificador integer
            ╠══ {
            ╠══ ,
            ╠══ -
      ╔═ Constante entera 8
Linea 9 se reconocio token negativo
            ╠══ }
[31mLinea 9: Error: Falta rango inferior [0m
            ╠══ ;
      ╔═ Identificador TYPEDEF
      ╔═ Identificador flotadito
            ╠══ :=
      ╔═ Identificador integer
            ╠══ {
            ╠══ }
[31mLinea 10: Error: Faltan ambos rangos [0m
            ╠══ ;
      ╔═ Identificador TYPEDEF
            ╠══ :=
      ╔═ Identificador integer
            ╠══ {
            ╠══ -
      ╔═ Constante entera 6
Linea 11 se reconocio token negativo
            ╠══ ,
            ╠══ -
      ╔═ Constante entera 8
Linea 11 se reconocio token negativo
            ╠══ }
[31mLinea 11: Error: Falta de nombre en el tipo
definido [0m
            ╠══ ;
      ╔═ Identificador TYPEDEF
      ╔═ Identificador flotadito
            ╠══ :=
            ╠══ {
            ╠══ -
      ╔═ Constante entera 6
Linea 12 se reconocio token negativo
```

```
                ╠══ ,
                ╠══ -
        ╒═ Constante entera 8
Linea 12 se reconocio token negativo
                ╠══ }
[31mLinea 12: Error: Falta el tipo base en la
declaracion de subtipo [0m
                ╠══ ;
        ╒═ Identificador IF
                ╠══ (
        ╒═ Identificador a
                ╠══ ,
        ╒═ Constante entera 3
                ╠══ +
        ╒═ Constante entera 4
                ╠══ )
                ╠══ =
                ╠══ (
        ╒═ Identificador b
                ╠══ ,
        ╒═ Constante entera 35
                ╠══ )
        ╒═ Identificador THEN
[31mLinea 13Error : Faltan los parentesis en la
condicion [0m
        ╒═ Identificador A
                ╠══ :=
        ╒═ Constante entera 1
                ╠══ ;
Linea :13 Asignacion
        ╒═ Identificador END_IF
Linea 13: Sentencia IF
                ╠══ ;
        ╒═ Identificador IF
                ╠══ (
                ╠══ (
        ╒═ Identificador a
                ╠══ ,
        ╒═ Constante entera 3
                ╠══ +
        ╒═ Constante entera 4
                ╠══ )
                ╠══ =
                ╠══ (
        ╒═ Identificador b
                ╠══ ,
        ╒═ Constante entera 35
                ╠══ )
        ╒═ Identificador THEN
[31mLinea 14Error : Falta el ')' en la condicion
[0m
        ╒═ Identificador A
                ╠══ :=
        ╒═ Constante entera 1
                ╠══ ;
Linea :14 Asignacion
        ╒═ Identificador END_IF
Linea 14: Sentencia IF
                ╠══ ;
        ╒═ Identificador IF
                ╠══ (
        ╒═ Identificador a
                ╠══ ,
        ╒═ Constante entera 3
                ╠══ +
        ╒═ Constante entera 4
                ╠══ )
                ╠══ =
```

```
                ╠══ (
        ╒═ Identificador b
                ╠══ ,
        ╒═ Constante entera 35
                ╠══ )
                ╘══ )
[31mLinea 15Error : Falta el '(' en la condicion
[0m
        ╒═ Identificador THEN
        ╒═ Identificador A
                ╠══ :=
        ╒═ Constante entera 1
                ╠══ ;
Linea :15 Asignacion
        ╒═ Identificador END_IF
Linea 15: Sentencia IF
                ╠══ ;
        ╒═ Identificador TYPEDEF
                ╠══ <
        ╒═ Identificador integer
                ╠══ >
        ╒═ Identificador tint
[31mLinea 16: Error: Falta de la palabra reservada
TRIPLE [0m
                ╠══ ;
        ╒═ Identificador TYPEDEF
        ╒═ Identificador TRIPLE
        ╒═ Identificador integer
        ╒═ Identificador tint
[31mLinea 17: Error: Faltan ambos '<>' en TRIPLE[0m
                ╠══ ;
        ╒═ Identificador TYPEDEF
        ╒═ Identificador TRIPLE
                ╠══ <
        ╒═ Identificador integer
                ╠══ >
                ╘══ ;
syntax error
[31mLinea 18: Error: Falta identificador al final
de la declaracion[0m
        ╒═ Identificador goto
                ╠══ ;
[31mLinea :19 Error: Falta la etiqueta en GOTO [0m
        ╒═ Identificador END
[31m✘[0mLinea 20[31m: Error: Falta el delimitador
BEGIN [0m
```

Errores pendientes

Léxico

Intento de incluir en el nombre de un id un carácter que no sea letra, dígito o "_".

Esto ocurre porque al leer este caracter invalido devuelve un ERROR (token 280) el Analizador Léxico entonces no lo considera una variable, pero como ya está esperando una variable en la asignación se rompe y no sabemos cómo resolverlo.

Sintácticos

- Falta de “,” en declaración de variables.

Para esto tenemos

```
sentencia    : sentencia_declarativa
             | sentencia_ejecutable
;
sentencias   : sentencias sentencia ';'
             | sentencia ';'
             | sentencia {System.out.println("\u001B[31m"+"Linea " +
AnalizadorLexico.saltoDeLinea + ": Erro: Falta ';' al final de la
sentencia "+"\u001B[0m");}
;
sentencia_declarativa    : declaracion_variable
                         | declaracion_funciones
                         | declaracion_subtipo
;

declaracion_variable     : tipo variables  {System.out.println("Linea "
+ AnalizadorLexico.saltoDeLinea + " declaracion de variables ");}
;

variables    : variables ',' variable_simple
             | variable_simple
;
```

creemos que el problema viene de que colocamos el ‘;’ recien en sentencias y eso genera que al hacer la siguiente regla

```
variables    : variables ',' variable_simple
             | variable_simple
            | variables variable_simple {print (“ERROR, SE ESPERA ,”);};
```

lee “ integer a,b c; “ y al terminar de leer b espera o un ‘;’ o ‘,’. Al llegar otro ID no sabe si es una declaración de una variable considerando a b como subtipo o si es parte de la

declaración generando shift/reduce. Esto se solucionaria bajando el ‘**;**’ a las variables pero también nos provoca problemas en las sentencias como IF o WHILE que solamente funcionarian si colocamos ‘; ;’.

- Falta de comparador en comparación.

```
condicion   : '(' '(' list_expre ')' comparador '(' list_expre ')' ')'
{System.out.println("Linea " + AnalizadorLexico.saltoDeLinea + ":
Condicion con lista de expresiones ");}
            | '(' list_expre ')' comparador '(' list_expre ')' ')'
{System.out.println("\u001B[31m"+"Linea " +
AnalizadorLexico.saltoDeLinea + "Error : Falta el '(' en la condicion
"+"\u001B[0m");}
            | '(' '(' list_expre ')' comparador '(' list_expre ')'
{System.out.println("\u001B[31m"+"Linea " +
AnalizadorLexico.saltoDeLinea + "Error : Falta el ')' en la condicion
"+"\u001B[0m");}
            |'(' list_expre ')' comparador '(' list_expre ')'
{System.out.println("\u001B[31m"+"Linea " +
AnalizadorLexico.saltoDeLinea + "Error : Faltan los parentesis en la
condicion "+"\u001B[0m");}
            | '(' expresion_arit comparador expresion_arit ')'
{System.out.println("Linea " + AnalizadorLexico.saltoDeLinea + ":
Condicion");}
            |  expresion_arit comparador expresion_arit ')'
{System.out.println("\u001B[31m"+"Linea " +
AnalizadorLexico.saltoDeLinea + "Error : Falta el '(' en la condicion
"+"\u001B[0m");}
            | '(' expresion_arit comparador expresion_arit
{System.out.println("\u001B[31m"+"Linea " +
AnalizadorLexico.saltoDeLinea + "Error : Falta el ')' en la condicion
"+"\u001B[0m");}
            | expresion_arit comparador expresion_arit
{System.out.println("\u001B[31m"+"Linea " +
AnalizadorLexico.saltoDeLinea + "Error : Faltan los parentesis en la
condicion "+"\u001B[0m");}
;
```

Este en un principio lo solucionamos al poner una regla con el comparador faltante, pero a último momento dejó de funcionar y no sabes porque entonces la sacamos.

- No devuelve el código sintáctico en un archivo.txt

Nos dimos cuenta tarde que el código sintáctico se debía devolver en un txt y como estábamos imprimiendo por pantalla no llegamos con el tiempo para realizar ese cambio. El del codigo léxico si llegamos.