	INSTITUTO TECNOLÓGICO DE CELAYA
PROGRA	**AUTOR: JUANA RUBI RAMIREZ GARCIA** **VALLE RODRIGUEZ JULIO CESAR**

CARRERA	NOMBRE DE LA ASIGNATURA
INGENIERÍA EN SISTEMAS COMPUTACIONALES	LENGUAJES Y AUTÓMATAS II

PROGRAMA No.	NOMBRE DEL PROGRAMA
2	PROTOTIPO DE ANALIZADOR SINTÁCTICO

1 INTRODUCCIÓN

Un analizador sintáctico es una de las partes de un compilador, es la fase que sigue una vez que se cuenta con un analizador léxico bien estructurado. Como se sabe, el analizador léxico crea tokens, estos tokens son procesados por el analizador sintáctico para poder construir la estructura de los datos, para ayudarnos a saber si el programa está sintácticamente correcto.

2 OBJETIVO (COMPETENCIA)

Realizar la segunda etapa de nuestro compilador, hacer el analizador sintáctico, desarrollando un programa en el cual se utilizan los tokens ya generados por el analizador léxico y comprobando que estos tokens están en el orden correcto, definido por el lenguaje definido previamente.

3 MARCO TEÓRICO

Conceptos

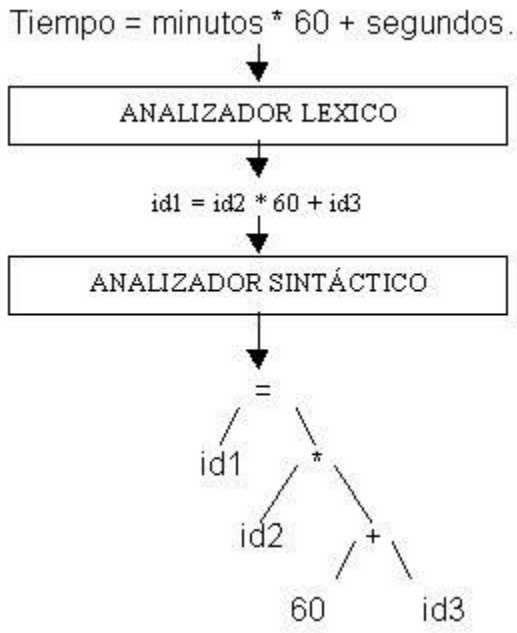
- **ANALIZADOR SINTÁCTICO**

Un analizador sintáctico es la segunda fase de un compilador que transforma su entrada a un árbol de derivación.

Con el analizador léxico creamos los tokens desde una secuencia de caracteres de entrada y son estos tokens los que son procesados por el analizador sintáctico.

Los lenguajes más conocidos por los analizadores sintácticos son los lenguajes libres de contexto. Todo analizador sintáctico que reconoce un lenguaje libre de contexto es equivalente en capacidad computacional a un autómata de pila.

Los estudios de los analizadores sintácticos se remontan a los años 1970, ya que en estos años se comenzaron a estudiar extensivamente, encontrando Numerosos patrones entre ellos y así permitiendo generar programas de analizadores sintácticos a partir de un lenguaje por ejemplo: yacc, GNU bison y javaCC.



Manejo de errores sintácticos

Considerar desde el principio el manejo de errores puede simplificar la estructura de un compilador y mejorar su respuesta a los errores.

Los errores en la programación pueden ser de los siguientes tipos:

• **Sintácticos,** por una expresión aritmética o paréntesis no equilibrados.

El manejo de errores de sintaxis es el más complicado desde el punto de vista de la creación de compiladores. Nos interesa que cuando el compilador encuentre un error, se recupere y siga buscando errores. Por lo tanto el manejador de errores de un analizador sintáctico debe tener como objetivos:

- Indicar los errores de forma clara y precisa. Aclarar el tipo de error y su localización.
- Recuperarse del error, para poder seguir examinando la entrada.
- No ralentizar significativamente la compilación.

Un buen compilador debe hacerse siempre teniendo también en mente los errores que se pueden producir; con ello se consigue:

- Simplificar la estructura del compilador.
- Mejorar la respuesta ante los errores

Cada lenguaje de programación debe tener reglas que describen la estructura sintáctica de un programa. Se puede describir la sintaxis de las construcciones de los lenguajes con ayuda de las gramáticas de contexto libre.

Las gramáticas nos ofrecen ciertas ventajas:

- Son especificaciones sintácticas y precisas de lenguajes de programación
- A partir de una gramática se puede generar automáticamente un analizador sintáctico
- Proporciona una estructura a un lenguaje de programación, siendo más fácil generar código y detectar errores

¿Qué es un analizador sintáctico?
Es la fase del analizador que se encarga de comprobar el orden en que el analizador léxico le va entregando los tokens sea válido. Si esto es así, significa que la sucesión de símbolos que representan dichos tokens puede ser generada por la gramática correspondiente al lenguaje del código fuente.

Si el programa es válido, suministra el árbol sintáctico que lo reconoce. En pocas palabras, la salida de un analizador sintáctico es alguna representación del árbol sintáctico que reconoce la secuencia de tokens suministrada por el analizador léxico.

Funciones analizador sintáctico
- Accede a la tabla de símbolos
- Comprueba si el programa es sintácticamente correcto

- Genera las estructuras de datos (árboles sintácticos) que representan el programa
- Reaccionar frente a los errores e intentar acotar la propagación de los errores

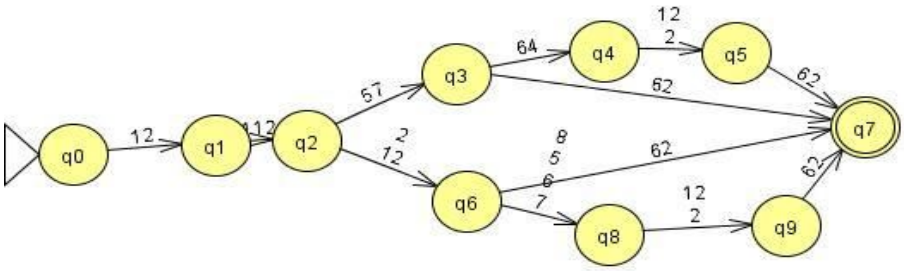
Para poder realizar este analizador se diseñó la siguiente gramática.

<bloquecodigo>	<lineacodigo>
<lineacodigo>	<lineacodigo> <linea> | <linea>
<linea>	<crearvariable> | <cambiarvalor> | <ciclocondicion> | <funcion>
<funcion>	<imprimir>
<imprimir>	<IMPRIME> (<CADENA> | <NOMBREVARIABLE>);
<valor>	<NUMERO> | <CADENA>
<crearvariable>	<tipodato> <NOMBREVARIABLE>;
<tipodato>	<T_ENTERO> | <T_CADENA>
<oprcomun>	<valor> <tipoopr> <valor> | <valor> <tipoopr> <NOMBREVARIABLE> | <NOMBREVARIABLE> <tipoopr> <valor> | <NOMBREVARIABLE> <tipoopr> <NOMBREVARIABLE>
<tipoopr>	<SUMA> | <RESTA> | <MULTIPLICACION> | <DIVISION>
<cambiarvalor>	<NOMBREVARIABLE> <ASIGNADOR> <cambvalor>;
<cambvalor>	<valor> | <oprcomun> | <NOMBREVARIABLE>
<ciclocondicion>	<condicionsi> | <ciclowhile>
<condicionsi>	<condicionsi> | <condicionsi><condicionsino>
<condicionsi>	<SI>(<condicion>) { <bloquecodigo> }

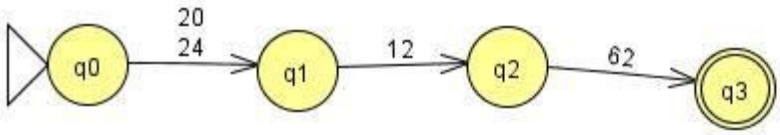
<condicion>	(<valor> <condicional> <valor>) | <valor> <condicional> <NOMBREVARIABLE> | <NOMBREVARIABLE> <condicional> <valor> | <NOMBREVARIABLE><condicional><NOMBREVARIABLE>
<condicional>	<MAYOR> | <MENOR> | <IGUAL> | <MAYORIGUAL> | <MENORIGUAL>
<condicionsino>	<SINO> {<bloquecodigo>}
<NUMERO>	2
<CADENA>	<CADENAT> . <CADENAT> | <CADENAT> . <valor> | <CADENAT>
<NUMERO>	2
<IMPRIME>	32
<ASIGNADOR>	112
<NOMBREVARIABLE>	12
<T_ENTERO>	24
<T_CADENA>	20
<SUMA>	5
<RESTA>	6
<MULTIPLICACION>	7
<DIVISION>	8
<MAYOR>	107
<MENOR>	105
<IGUAL>	111
<MAYORIGUAL>	110

<MENORIGUAL>	109
<MIENTRAS>	16
<SINO>	18
<SI>	17
<CADENAT>	57
(	58
)	59
;	62
{	60
}	61

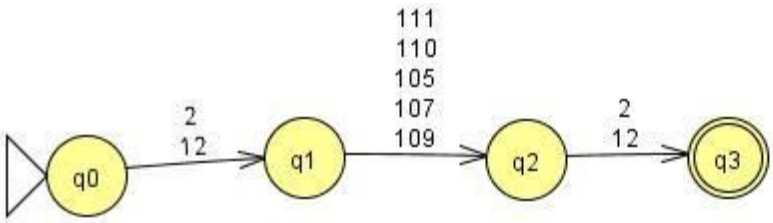
Producción <asignar var>



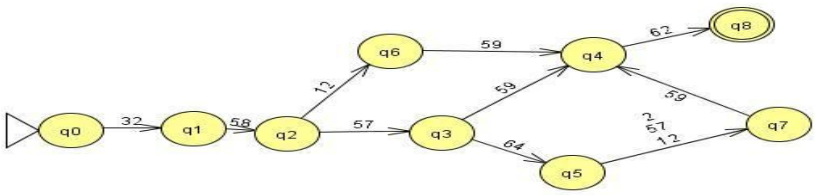
Producción <crear variables>



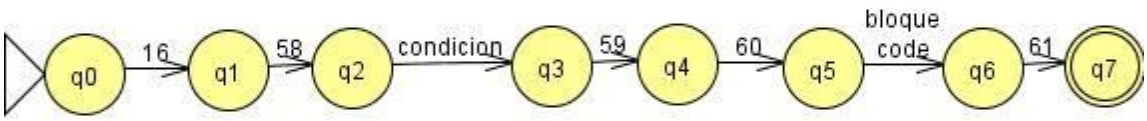
Producción <condición>



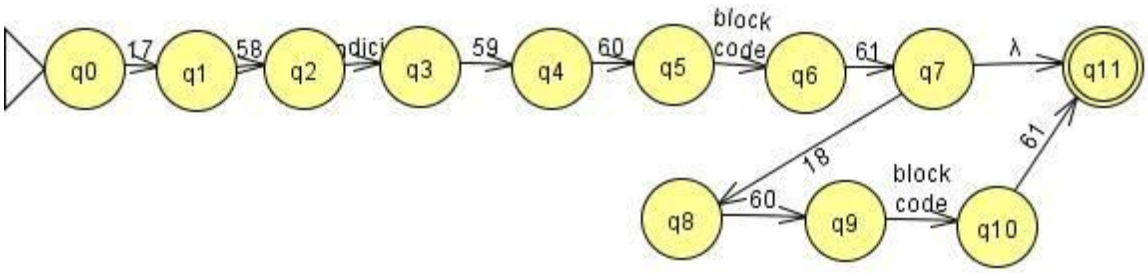
Producción <imprimir>

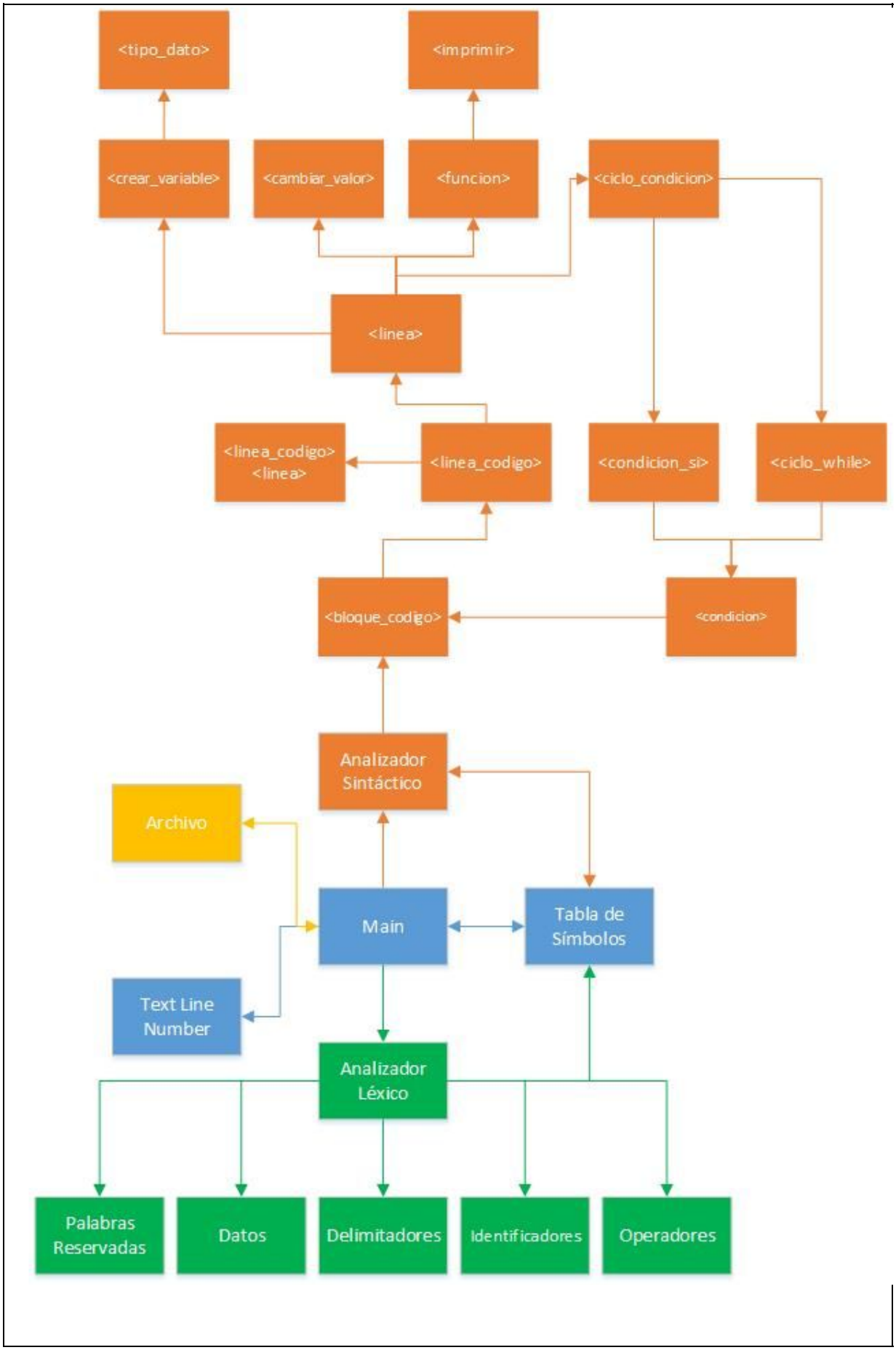


Producción <mientras>



Producción <si-sino>



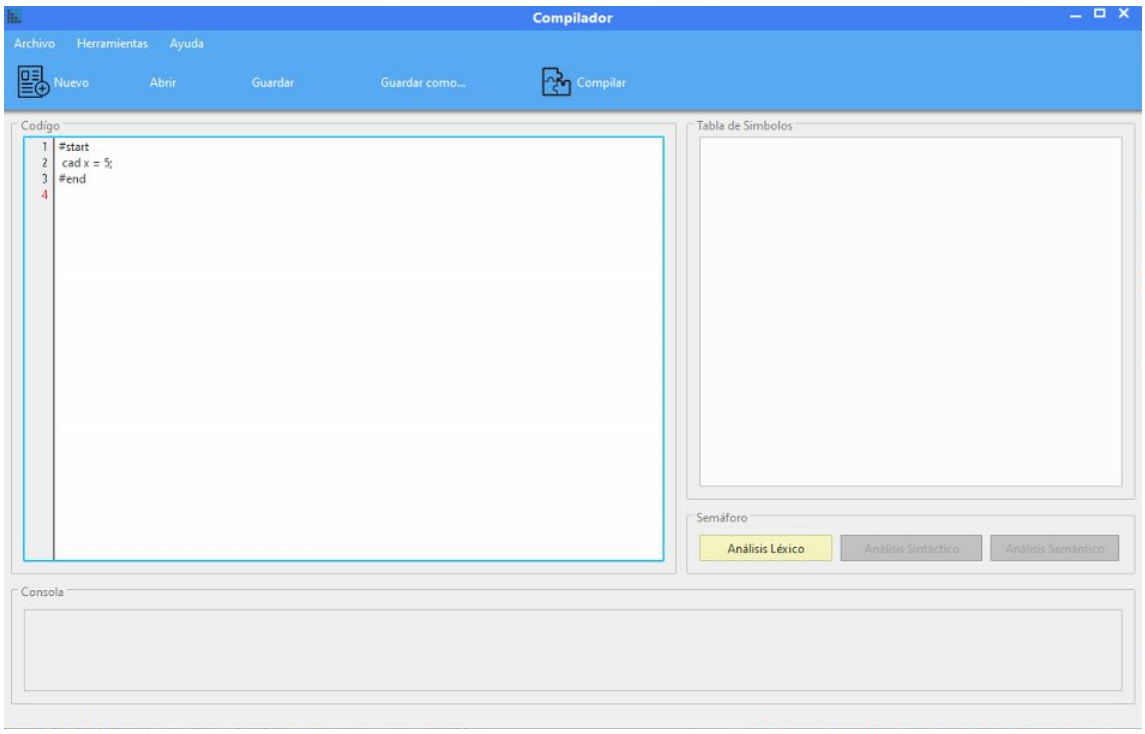

<tipo_dato>
<imprimir>
<crear_variable>
<cambiar_valor>
<funcion>
<ciclo_condicion>
<linea>
<linea_codigo>
<linea>
<linea_codigo>
<condicion_si>
<ciclo_while>
<bloque_codigo>
<condicion>
Analizador Sintáctico
Archivo
Main
Tabla de Símbolos
Text Line Number
Analizador Léxico
Palabras Reservadas
Datos
Delimitadores
Identificadores
Operadores

4	REQUISITOS BÁSICOS

- Tener conocimientos básicos sobre compiladores y como se conforma, enfatizando en saber que es un analizador sintáctico y su estructura.
- Contar con un analizador léxico completo en cuanto a manejo de la tabla de símbolos y de manejo errores.
- Manejo de estructuras de datos en java.

5	DESARROLLO

Para el desarrollo del analizador sintáctico primero se hizo la implementación de un semáforo para el análisis léxico y sintáctico, los botones del análisis sintáctico se encuentran deshabilitados hasta que el análisis léxico haya terminado de manera correcta.



Una vez que termina el análisi léxico correctamente habilita el botón par el análisis sintáctico el cual llama a la clase AnalisisSintactico del paquete compilador.sintactico este se distribuye de esta manera para mantener una modularidad dentro del proyecto.

El método una vez ejecutado llama a la clase BloqueLinea para llamar al primer

bloque de líneas que conforman a la clase principal

```
private void a_anaSintactico(){
      BloqueCodigo o_bloqueCodigo = new BloqueCodigo( a_TablaDeSimbolos ,
a_codFuente );
       a_consola=o_bloqueCodigo.m_getConsola();
       a_TablaDeSimbolos=o_bloqueCodigo.a_TablaDeSimbolos;
   }
```

Este llama a su vez a la linea o lineas de código que pueden componer el código fuente declarado en nuestras gramáticas dependientes del contexto desarrolladas para el análizador sintáctico.

```
private void m_tipoLinea(String p_codFuente){
       String v_codFuente=p_codFuente;
       if(!"".equals(v_codFuente)){
                              CreaVariable                    o_creaVariable=new
CreaVariable(a_TablaDeSimbolos,v_codFuente,a_Linea);
          a_consola+=o_creaVariable.m_getConsola();
          a_TablaDeSimbolos=o_creaVariable.m_getTabla();
          a_codFuente=o_creaVariable.m_getCodigoFuente();
          if(o_creaVariable.m_getCodigoFuente()!=v_codFuente){
             a_codFuente=o_creaVariable.m_getCodigoFuente();
          }
       }
   }
```

Para este caso de estudio en particular se llama la instrucción para crear una variable y llama a su vez la clase encargada de gestionar el desarrollo de una variable.

```
private void m_cadQ4(String p_codFuente){
       String v_codFuente=p_codFuente;
       boolean v_bandera=true;
       while(!"".equals(v_codFuente)&&v_bandera){
          if(v_codFuente.charAt(0)==32){
             v_bandera=true;
             v_codFuente=v_codFuente.substring(1,v_codFuente.length());
          }else{
             if(v_codFuente.charAt(0)==10){
                v_bandera=true;
                a_Linea++;
                v_codFuente=v_codFuente.substring(1,v_codFuente.length());
             }else{
                if(v_codFuente.charAt(0)==' '){
                   v_bandera=true;
                   v_codFuente=v_codFuente.substring(1,v_codFuente.length());
                }else{
```

```
                    v_bandera=false;
                }
            }
        }
    }
    if(!"".equals(v_codFuente)){
                                                Variable o_variable = new
Variable(a_TablaDeSimbolos,v_codFuente,a_Linea,m_buscaTipoVariable("cad")
);
        a_TablaDeSimbolos=o_variable.m_getTabla();
        a_consola+=o_variable.m_getConsola();
        a_codFuente=o_variable.m_getCodigoFuente();
    }else{
        a_consola+="Error [210]: Error no se declaro ninguna variable\n";
        a_consola+="Error en la linea: "+a_Linea+"\n";
    }
  }
```

Una vez que detectó el tipo de variable que se va a declarar el analizador se dispone a comprobar la formación de su sintaxis determinando que lo que se declaró en la tabla de símbolos es una variable.

```
private void m_variableQ3(String p_codFuente,int p_index){
    String v_codFuente=p_codFuente;
    boolean v_bandera=true;
    while(!"".equals(v_codFuente)&&v_bandera){
        if(v_codFuente.charAt(0)==32){
            v_bandera=true;
            v_codFuente=v_codFuente.substring(1,v_codFuente.length());
        }else{
            if(v_codFuente.charAt(0)==10){
                v_bandera=true;
                a_Linea++;
                v_codFuente=v_codFuente.substring(1,v_codFuente.length());
            }else{
                if(v_codFuente.charAt(0)==' '){
                    v_bandera=true;
                    v_codFuente=v_codFuente.substring(1,v_codFuente.length());
                }else{
                    v_bandera=false;
                }
            }
        }
    }
    if(!"".equals(v_codFuente)){
        m_variableQ4(v_codFuente.substring(1,v_codFuente.length()),p_index);
                    Valor o_Valor=new Valor(a_TablaDeSimbolos, p_codFuente,
a_Linea,m_buscaVariable(a_codFuente.substring(0,p_index)));
```

```
            a_TablaDeSimbolos=o_Valor.m_getTabla();
            a_consola+=o_Valor.m_getConsola();
            a_codFuente=o_Valor.m_getCodigoFuente();
        }else{
            a_consola+="Error [215]: Error no se declaro ningun valor\n";
            a_consola+="Error en la linea: "+a_Linea+"\n";
        }
    }
```

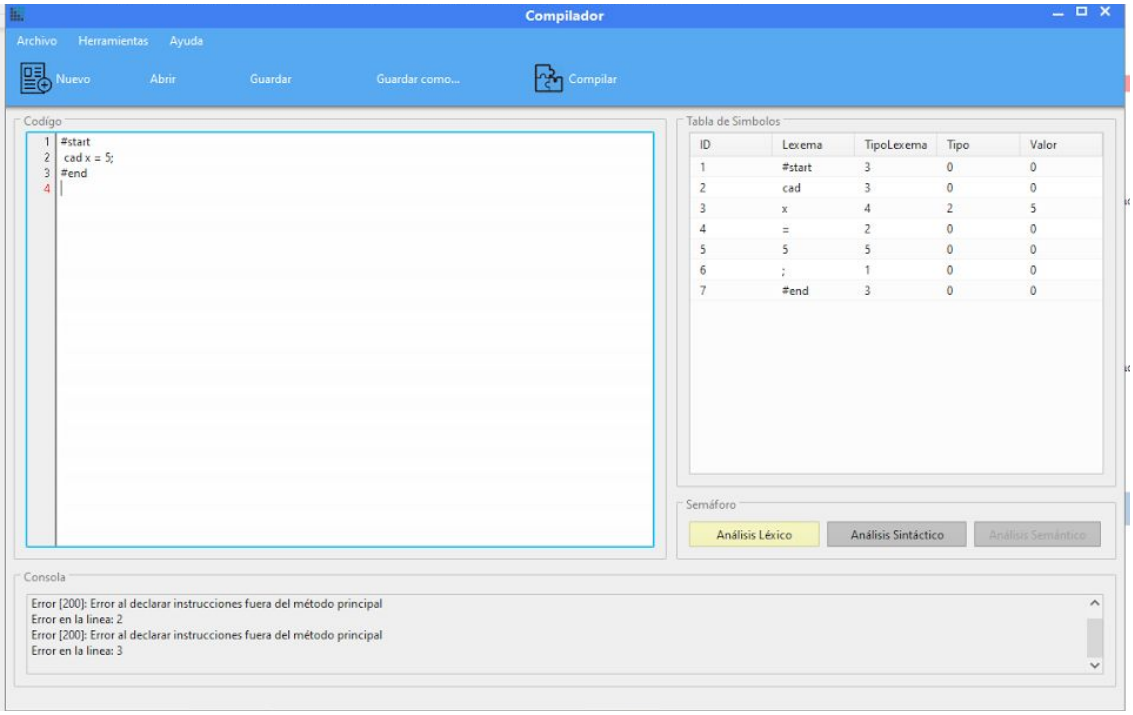
Si se declaró correctamente la variable en la tabla de símbolos y la sintaxis es adecuada, el analizador compara si es que se asignó un valor o solo se termino la sentencia con un ‘;’.

```
private boolean m_entQ2(String p_codFuente,int p_index){
        String v_codFuente=p_codFuente;
        boolean v_bandera=true;
        while(!"".equals(v_codFuente)&&v_bandera){
            if(v_codFuente.charAt(0)==32){
                v_bandera=true;
                v_codFuente=v_codFuente.substring(1,v_codFuente.length());
            }else{
                if(v_codFuente.charAt(0)==10){
                    v_bandera=true;
                    a_Linea++;
                    v_codFuente=v_codFuente.substring(1,v_codFuente.length());
                }else{
                    if(v_codFuente.charAt(0)==' '){
                        v_bandera=true;
                        v_codFuente=v_codFuente.substring(1,v_codFuente.length());
                    }else{
                        v_bandera=false;
                    }
                }
            }
        }
        if(!"".equals(v_codFuente)){
            if(v_codFuente.charAt(0)==';'){
                m_entQ3(v_codFuente.substring(1,v_codFuente.length()),p_index);
            }else{
                m_entQ3(v_codFuente.substring(1,v_codFuente.length()),p_index);
                a_consola+="Error [210]: Error falta ';'\n";
                a_consola+="Error en la linea: "+a_Linea+"\n";
            }
        }else{
            a_consola+="Error [210]: Error falta ';'\n";
            a_consola+="Error en la linea: "+a_Linea+"\n";
        }
        return true;
```

```
    }
```

En caso de encontrar un valor asignado este comprueba inmediatamente que la sentencia se termine con un ‘;’ de lo contrario llama a la pila de errores. Si no llama los métodos de asignación de valores para ingresar en la tabla de símbolos el valor que le pertenece al lexema correspondiente.

```
public  ArrayList <Token> m_getTabla(){
        return a_TablaDeSimbolos;
    }
```



6 ANEXOS

Bitácora

Tiempo	Problema	Solución
15/11/16	Al momento de implementar el semáforo dentro del compilador, los procesos de análisis se ejecutaban uno tras otro.	Se agregaron banderas para el control de los procesos de análisis, de tal manera que no se puedan ejecutar de forma consecutiva, sino a través del semáforo implementado.
16/11/16	El analizador sintáctico no tenía la capacidad de reponerse de	Se creó un método para descartar al lexema que

	un error y esto provocaba que el analizador colapsara.	estuviera provocando errores y continuar con el proceso de análisis.
17/11/16	El analizador no estaba asignando el tipo ni el valor de los lexemas que estaban siendo declarados.	Se mandó un parámetro para revisar el id del lexema al cual se le iba a asignar el valor.

Errores

Número	Descripción
140	Sin: Variable mal creada, no se respeta el orden de creación.
150	Sin: Asignación mal a una variable.
155	Sin: Asignación mal a variable, paréntesis mal estructurados con número negativo.
160	Sin: Error en la función imprime, mal estructurada.
170	Sin: Llave sin cerrar.
180	Sin: Error sobre la estructura de la función SI.
190	Sin: Error en la condición de la función SI.
200	Sin: Error en la estructura la función SINO
210	Sin: Uso de SINO sin antes haber declarado un SI.
220	Sin: No se declaró el método inicio.
225	Sin: Error en la estructura del método inicio.

Versiones

Versión	Detalles	Fecha
1.1	Implementación de analizador léxico.	14/10/16
1.2	Optimización de analizador léxico aplicando autómatas para el reconocimiento y clasificación de tokens.	28/10/16
1.3	Implementación del analizador sintáctico. División del analizador en varios autómatas en lugar de solo uno. Optimización de los autómatas sintácticos.	18/11/16

Casos de uso sintácticos

1.- Una variable al momento de ser declarada debe seguir el orden estricto de la gramática, que es “<tipodato> <NOMBREVARIABLE>;”, por lo que si se trata de asignar un valor al momento de declarar o no se sigue la estructura para declarar el compilador marca el error 140 que es un aviso de variable mal declarada.
Ejemplo.

```
ent x;  <--- correcto.
ent x = 5; <--- incorrecto.
```

2.- La asignación de números negativos a variables numéricas debe ser por medio de paréntesis, si se abre el paréntesis pero no se cierra se marcará como error 155 indicando una mala asignación de número negativo, si no se abre pero se cierra solo se marcara error 150 de asignación incorrecta de valor. Se puede asignar valores positivos dentro de los paréntesis.

```
ent x;
ent y;
x = (-1);
y = (1);
```

3.- La función *imprime* debe seguir la estructura declarada en la gramática del lenguaje de manera que cualquier falla en esta estructura, como puede ser la

ausencia de paréntesis o de argumento, se marca con el error 160, el cual denota un error en la función imprime. La estructura debe ser: “<IMPRIME> (<CADENA> | <NOMBREVARIABLE>);”.

```
imprime (abc;
imprime abc);
imprime ();
```

4.- Una condición debe estar conformada de dos valores y un signo comparativo siguiendo la estructura que la gramática indica, de tal manera que sólo se pueden utilizar valores numéricos y/o identificadores. Si la estructura es incorrecta se muestra al usuario el error 170 que denota la estructura incorrecta en la condición y la línea aproximada en la que ésta se encuentra.
Ejemplo.

```
mientras ( x < 55) ←--- correcto.
si ( 55 < x )      ←--- correcto.
mientras (x >= )   ←--- incorrecto.
si ( == x)         ←--- incorrecto.
```

5. La función *si* se ve obligada a seguir la estructura “<SI>(<condicion>) { <bloquecodigo> }” señalada en la definición de la gramática, de no seguir esta estructura el compilador lo da a conocer al usuario por medio del error 200 mencionando también la línea donde se da el error. Ejemplo.

```
si ( x <= 10) {
//bloque de código//
}
```

6.- La función *sino* se rige por la definición de la gramática “<SINO> {<bloquecodigo>}” por lo que si falta un llave en su estructura o por ejemplo se reemplaza una llave por un paréntesis se marca con el error 220 sobre una mal estructura de la función.
Ejemplo.

```
sino { //bloque de código// }  ←--- correcto
sino ( //bloque de código// )  ←--- incorrecto
sino //bloque de código//      ←--- incorrecto
```

7.- La función *sino* no puede ser usada sin anteriormente haber declarado una función *si*, por lo que si se trata de realizar esta acción, se le notifica al usuario por medio del error 225 mencionando la línea donde se encuentra esta estructura *sino.*
Ejemplo.

```
imprime (“hola”);
sino{
  //bloque de código//
}
```

8.- La declaración de variables debe hacerse siguiendo la estructura que la gramática indica, “<tipodato> <NOMBREVARIABLE>;”, de no seguir este orden se toma como si se deseara asignar un valor a una variable y será marcado con

el error 150.
Ejemplo.

```
ent x;      ←--- correcto
cad hola;   ←--- correcto
x ent;      ←--- incorrecto
hola cad;   ←--- incorrecto
```

9. En la asignación de valores a variables se pueden realizar operaciones aritméticas pero no más de una, por lo tanto como máximo deben ser dos operandos y un operador aritmético.
Ejemplo.

```
ent x;
ent y;
x = 5 * 2;
y = 2 * x;
```

11.- Antes de comenzar a codificar se debe declarar la etiqueta o palabra clave *inicio* y posteriormente realizar el bloque de código que se desee, por lo que la palabra inicio será el primer token en la tabla de símbolos, de no ser así se marca el error 230 al usuario.
Ejemplo.

```
#start
{
   ent x;
   x=5;
}
```

12.- Para la etiqueta *#start* se debe cumplir el hecho de abrir y cerrar llaves, según lo marca la producción gramatical “#start { <bloquecodigo>}”, de no ser así el usuario recibe como retroalimentación el mensaje del error 240 donde se declara que la estructura del método está equivoca.
Ejemplo.

```
#start
{


}
```

13. En la función imprime se pueden imprimir variables con la condición de que antes de éstas debe haber una cadena y de preferencia no vacía y concatenar por medio del símbolo “.”, por el contrario se marca el error 160.
Ejemplo.

```
ent x;
x = 5;
cad abc;
abc = “hola”;
imprime (“hola ”.x);
imprime (“valor ”.abc);
```

7	REFERENCIAS

- Aho A.V (2008). Compiladores. Principios, técnicas y herramientas. Ed. 2, México: Pearson Addison Whesley.
- http://www.lcc.uma.es/~galvez/ftp/tci/tictema3.pdf
- http://www.oocities.org/mx/amigos2365/anteproyecto.html