

# Universidad de Guadalajara

Centro Universitario de Ciencias Exactas e Ingenierías

DIVISIÓN DE TECNOLOGÍAS PARA LA INTEGRACIÓN  
CIBER-HUMANA

Departamento de Ciencias Computacionales



Seminario de Solución de Problemas de Traductores de  
Lenguajes II  
Sección: D06

Subproducto integrador 1. Analizador Léxico

Presenta:

215638591 - Avila Guerrero Ismael Guadalupe

215769734 - Hernandez Chavez Isaac Alberto

219747646 - Martinez Zepeda Diego Aalberto

Profesor: Roberto Patiño Ruiz

Fecha de entrega: 23 de septiembre de 2023

## ÍNDICE

<b>OBJETIVO</b>	<b>2</b>
<b>DESARROLLO</b>	<b>3</b>
Requerimientos Funcionales	3
Requerimientos no Funcionales	5
Gramática	7
Autómata	8
Interfaz Gráfica	9
Código Fuente	10
Pruebas de Ejecución	18
<b>CONCLUSIONES PERSONALES</b>	<b>19</b>
<b>REFERENCIAS BIBLIOGRÁFICAS</b>	<b>20</b>

## **OBJETIVO**

Aplicar los fundamentos básicos de los compiladores, para desarrollar un analizador léxico capaz de leer un flujo de caracteres de entrada (símbolos léxicos) y transformarlo en una secuencia de componentes léxicos que posteriormente utilizará el analizador sintáctico.

## DESARROLLO

Para el desarrollo del analizador léxico se tomaron en cuenta los siguientes requerimientos funcionales y no funcionales:

### Requerimientos Funcionales

<b>Número de requerimiento</b>	1		
<b>Nombre del requerimiento</b>	Permitir la entrada de caracteres para dar el proceso de análisis		
<b>Tipo</b>	<input checked="" type="checkbox"/> Requerimiento		<input type="checkbox"/> Restricción
<b>Prioridad del requerimiento</b>	<input checked="" type="checkbox"/> Alta/Eseñcial	<input type="checkbox"/> Media/Deseado	<input type="checkbox"/> Baja/Opcional

<b>Número de requerimiento</b>	2		
<b>Nombre del requerimiento</b>	Eliminar los blancos o comentarios		
<b>Tipo</b>	<input checked="" type="checkbox"/> Requerimiento		<input type="checkbox"/> Restricción
<b>Prioridad del requerimiento</b>	<input checked="" type="checkbox"/> Alta/Eseñcial	<input type="checkbox"/> Media/Deseado	<input type="checkbox"/> Baja/Opcional

<b>Número de requerimiento</b>	3		
<b>Nombre del requerimiento</b>	Hacer el proceso de verificación del análisis léxico con la cadena ingresada por el usuario		
<b>Tipo</b>	<input checked="" type="checkbox"/> Requerimiento		<input type="checkbox"/> Restricción
<b>Prioridad del requerimiento</b>	<input checked="" type="checkbox"/> Alta/Eseñcial	<input type="checkbox"/> Media/Deseado	<input type="checkbox"/> Baja/Opcional

<b>Número de requerimiento</b>	4		
<b>Nombre del requerimiento</b>	Mostrar en pantalla los token encontrados en la cadena ingresada por el usuario		
<b>Tipo</b>	<input checked="" type="checkbox"/> Requerimiento		<input type="checkbox"/> Restricción
<b>Prioridad del requerimiento</b>	<input checked="" type="checkbox"/> Alta/Eseñcial	<input type="checkbox"/> Media/Deseado	<input type="checkbox"/> Baja/Opcional

## Requerimientos no Funcionales

<b>Número de requerimiento</b>	1		
<b>Nombre del requerimiento</b>	Mostrar en pantalla la ejecución del programa exitosamente		
<b>Tipo</b>	<input checked="" type="checkbox"/> Requerimiento		<input type="checkbox"/> Restricción
<b>Prioridad del requerimiento</b>	<input checked="" type="checkbox"/> Alta/Esencial	<input type="checkbox"/> Media/Deseado	<input type="checkbox"/> Baja/Opcional

<b>Número de requerimiento</b>	2		
<b>Nombre del requerimiento</b>	Usar una interfaz gráfica que permita al usuario hacer uso de la herramienta sin conocerla previamente		
<b>Tipo</b>	<input checked="" type="checkbox"/> Requerimiento		<input type="checkbox"/> Restricción
<b>Prioridad del requerimiento</b>	<input checked="" type="checkbox"/> Alta/Esencial	<input type="checkbox"/> Media/Deseado	<input type="checkbox"/> Baja/Opcional

<b>Número de requerimiento</b>	3		
<b>Nombre del requerimiento</b>	Usar una interfaz gráfica que permita al usuario sentir que hay una linealidad en los pasos a seguir para usar la herramienta		

<b>Tipo</b>	<input checked="" type="checkbox"/> Requerimiento		<input type="checkbox"/> Restricción
<b>Prioridad del requerimiento</b>	<input type="checkbox"/> Alta/Eencial	<input checked="" type="checkbox"/> Media/Deseado	<input type="checkbox"/> Baja/Opcional

## Gramática

La gramática utilizada para reconocer los símbolos es la siguiente:

Token	Tipo	Lexema
Identificador	0	
Número	1	
Punto y coma	2	;
Coma	3	,
Paréntesis Izq	4	(
Paréntesis Der	5	)
Llave Izq	6	{
Llave Der	7	}

Token	Tipo	Lexema
Signo de Pesos	8	\$
Operador Relacional	9	!=, ==, <, <=, >, >=
Asignación	10	=
Operador Lógico	11	, &&
Operador Suma	12	+, -
Operador Multiplicación	13	*, /
Tipo de Dato	14	int, float
Palabra Reservada	15	if, else, while, return

Imagen 1. Gramática.



# Autómata

El autómata que utilizamos para el desarrollo del analizador léxico es el siguiente:

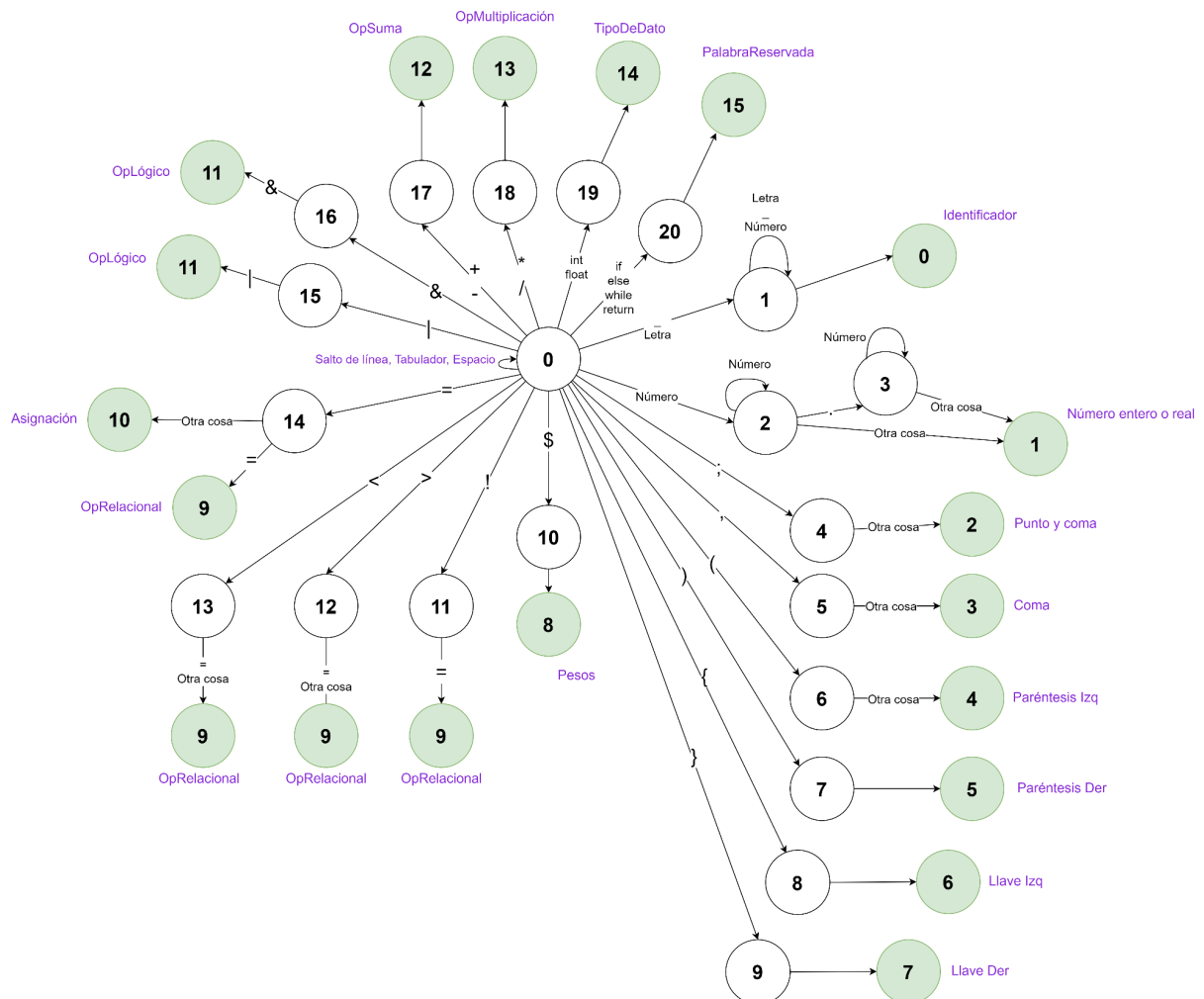


Imagen 2. Autómata.

## Interfaz Gráfica

El analizador léxico utilizamos el lenguaje de programación C#, debido a que es un lenguaje de alto nivel, es un lenguaje orientado a objetos y cuenta con una sintaxis muy familiar para nosotros, ya que es muy similar a lenguajes como C++ y Java.

Otra de las razones para utilizar este lenguaje es que el framework de .NET tiene soporte para los Windows Forms, que son elementos de interfaz gráfica para crear aplicaciones de escritorio nativas para Windows.

La interfaz gráfica desarrollada está pensada para ser muy intuitiva para el usuario. Cuenta con un input en el que se puede ingresar texto, el cual debe ser código fuente. El botón con la leyenda “Analizador Léxico” toma el texto ingresado, lo analiza y crea una lista de componentes léxicos que se muestra en el listado de la parte inferior.

El diseño implementado es el siguiente:

[illegible]

Imagen 3. Interfaz gráfica de usuario.

## Código Fuente

El analizador léxico se dividió en varias clases para facilitar el desarrollo. La primera de ellas es la clase `LexicalComponent`, que guarda las propiedades de los componentes léxicos reconocidos por el analizador.

La clase cuenta con un constructor que recibe como parámetro los valores para inicializar el componente léxico. Además, tiene implementadas las propiedades para acceder y modificar los atributos de cada objeto de esta clase.

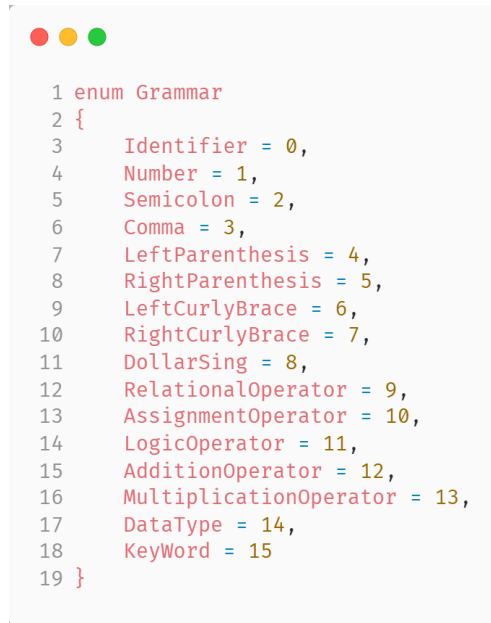
El código de esta clase es el siguiente:

A screenshot of a code editor window with a light gray background. At the top left, there are three colored window control buttons: red, yellow, and green. The code is written in C# and is for a class named `LexicalComponent`. It includes three private fields: `string lexeme`, `string token`, and `int number`. There are two public constructors: a parameterless one and one that takes the three fields as arguments. Additionally, there are three public properties: `Lexeme`, `Token`, and `Numbe` (note the typo), each with a `get` and `set` accessor. The code is line-numbered from 1 to 33.

```
1  internal class LexicalComponent
2  {
3      string lexeme;
4      string token;
5      int number;
6
7      public LexicalComponent() { }
8
9      public LexicalComponent(string lexeme, string token, int number)
10     {
11         this.lexeme = lexeme;
12         this.token = token;
13         this.number = number;
14     }
15
16     public string Lexeme
17     {
18         get { return lexeme; }
19         set { lexeme = value; }
20     }
21
22     public string Token
23     {
24         get { return token; }
25         set { token = value; }
26     }
27
28     public int Numbe
29     {
30         get { return number; }
31         set { number = value; }
32     }
33 }
```

Imagen 4. Código de la clase `LexicalComponent`.

El analizador léxico se implementó en una clase llamada `Analyzers`. La gramática se cargó en un enum para facilitar su uso a lo largo del análisis.

A screenshot of a code editor window with a light gray background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in a C#-like syntax, defining an enum named 'Grammar'. The enum contains 15 members, each with a name and a numerical value from 0 to 15. The code is as follows:

```
1 enum Grammar
2 {
3     Identifier = 0,
4     Number = 1,
5     Semicolon = 2,
6     Comma = 3,
7     LeftParenthesis = 4,
8     RightParenthesis = 5,
9     LeftCurlyBrace = 6,
10    RightCurlyBrace = 7,
11    DollarSing = 8,
12    RelationalOperator = 9,
13    AssignmentOperator = 10,
14    LogicOperator = 11,
15    AdditionOperator = 12,
16    MultiplicationOperator = 13,
17    DataType = 14,
18    KeyWord = 15
19 }
```

Imagen 5. Gramática cargada en un enum.

El método `LexicalAnalyze` es un método estático que recibe como parámetros un `string` y la lista de elementos léxicos. El `string` es una línea de entrada proporcionada por el usuario para el analizador.

El analizador se implementó con una robusta estructura de `switch-case`. El analizador recorre carácter por carácter el `string` recibido. Se inicializan una serie de variables que son usadas para guardar el estado actual, el lexema que se va construyendo y el token correspondiente al lexema.

Conforme se va reconociendo un lexema, se agrega a la lista de elementos léxicos y se reinician las variables antes mencionadas. El código fuente del analizador léxico se detalla a continuación.

El case 0 es el más extenso de todos, ya que es donde inicia el reconocimiento de los lexemas (ver [Imagen 2](#)).

```

1 case 0:
2 if (Char.IsLetter(line[i]) || line[i] == '_') {
3     state = 1;
4     lexeme += line[i];
5     token = Grammar.Identifier.ToString();
6
7     if (!Char.IsDigit(line[i + 1]) && !Char.IsLetter(line[i + 1]) && line[i + 1] != '_') {
8         if (lexeme == "int" || lexeme == "float")
9             elements.Add(new LexicalComponent(lexeme, Grammar.DataType.ToString(), (int)Grammar.DataType));
10
11         else if (lexeme == "if" || lexeme == "else" || lexeme == "while" || lexeme == "return")
12             elements.Add(new LexicalComponent(lexeme, Grammar.Keyword.ToString(), (int)Grammar.Keyword));
13
14         else
15             elements.Add(new LexicalComponent(lexeme, token, (int)Grammar.Identifier));
16
17         state = 0; lexeme = ""; token = "";
18     }
19
20     break;
21 }
22 else if (Char.IsDigit(line[i])) {
23     state = 2;
24     lexeme += line[i];
25     token = Grammar.Number.ToString();
26     break;
27 }
28 else if (line[i] == ';') {
29     lexeme += line[i];
30     elements.Add(new LexicalComponent(lexeme, Grammar.Semicolon.ToString(), (int)Grammar.Semicolon));
31     state = 0; lexeme = ""; token = "";
32     break;
33 }
34 else if (line[i] == ',') {
35     lexeme += line[i];
36     elements.Add(new LexicalComponent(lexeme, Grammar.Comma.ToString(), (int)Grammar.Comma));
37     state = 0; lexeme = ""; token = "";
38     break;
39 }
40 else if (line[i] == '(') {
41     lexeme += line[i];
42     elements.Add(new LexicalComponent(lexeme, Grammar.LeftParenthesis.ToString(), (int)Grammar.LeftParenthesis));
43     state = 0; lexeme = ""; token = "";
44     break;
45 }
46 else if (line[i] == ')') {
47     lexeme += line[i];
48     elements.Add(new LexicalComponent(lexeme, Grammar.RightParenthesis.ToString(), (int)Grammar.RightParenthesis));
49     state = 0; lexeme = ""; token = "";
50     break;
51 }
52 else if (line[i] == '{') {
53     lexeme += line[i];
54     elements.Add(new LexicalComponent(lexeme, Grammar.LeftCurlyBrace.ToString(), (int)Grammar.LeftCurlyBrace));
55     state = 0; lexeme = ""; token = "";
56     break;
57 }
58 else if (line[i] == '}') {
59     lexeme += line[i];
60     elements.Add(new LexicalComponent(lexeme, Grammar.RightCurlyBrace.ToString(), (int)Grammar.RightCurlyBrace));
61     state = 0; lexeme = ""; token = "";
62     break;
63 }
64 else if (line[i] == '$') {
65     lexeme += line[i];
66     elements.Add(new LexicalComponent(lexeme, Grammar.DollarSing.ToString(), (int)Grammar.DollarSing));
67     state = 0; lexeme = ""; token = "";
68     break;
69 }
70 else if (line[i] == '!') {
71     state = 11;
72     lexeme += line[i];
73     token = Grammar.RelationalOperator.ToString();
74     break;
75 }
76 else if (line[i] == '>') {
77     state = 12;
78     lexeme += line[i];
79     token = Grammar.RelationalOperator.ToString();
80     break;
81 }
82 else if (line[i] == '<') {
83     state = 13;
84     lexeme += line[i];
85     token = Grammar.RelationalOperator.ToString();
86     break;
87 }
88 else if (line[i] == '=') {
89     state = 14;
90     lexeme += line[i];
91     break;
92 }
93 else if (line[i] == '|') {
94     state = 15;
95     lexeme += line[i];
96     token = Grammar.LogicOperator.ToString();
97     break;
98 }
99 else if (line[i] == '&') {
100     state = 16;
101     lexeme += line[i];
102     token = Grammar.LogicOperator.ToString();
103     break;
104 }
105 else if (line[i] == '+' || line[i] == '-') {
106     lexeme += line[i];
107     elements.Add(new LexicalComponent(lexeme, Grammar.AdditionOperator.ToString(), (int)Grammar.AdditionOperator));
108     state = 0; lexeme = ""; token = "";
109     break;
110 }
111 else if (line[i] == '*' || line[i] == '/') {
112     lexeme += line[i];
113     elements.Add(new LexicalComponent(lexeme, Grammar.MultiplicationOperator.ToString(), (int)Grammar.MultiplicationOperator));
114     state = 0; lexeme = ""; token = "";
115     break;
116 }
117 break;

```

Imagen 6. Case 0 del analizador léxico.

El case 1 se encarga de reconocer palabras. Aquí mismo se evalúa si el lexema corresponde a un tipo de dato, a una palabra reservada o es simplemente un identificador.



```
1 case 1:
2     if (Char.IsDigit(line[i]) || Char.IsLetter(line[i]) || line[i] == '_') {
3         state = 1;
4         lexeme += line[i];
5         token = Grammar.Identifier.ToString();
6
7         if (!Char.IsDigit(line[i+1]) && !Char.IsLetter(line[i+1]) && line[i+1] != '_') {
8             if (lexeme == "int" || lexeme == "float")
9                 elements.Add(new LexicalComponent(lexeme, Grammar.DataType.ToString(), (int)Grammar.DataType));
10
11             else if (lexeme == "if" || lexeme == "else" || lexeme == "while" || lexeme == "return")
12                 elements.Add(new LexicalComponent(lexeme, Grammar.KeyWord.ToString(), (int)Grammar.KeyWord));
13
14             else
15                 elements.Add(new LexicalComponent(lexeme, token, (int)Grammar.Identifier));
16
17             state = 0; lexeme = ""; token = "";
18         }
19     }
20 break;
```

Imagen 7. Case 1 del analizador léxico. Reconocimiento de identificadores, tipos de datos y palabras reservadas.

El case 2 y 3 corresponden a números. En estos dos estados se reconocen tanto números enteros como reales.

```

1
2 case 2:
3     if (Char.IsDigit(line[i])) {
4         state = 2;
5         lexeme += line[i];
6         token = Grammar.Number.ToString();
7     }
8     else if (line[i] == '.') {
9         state = 3;
10        lexeme += line[i];
11        token = Grammar.Number.ToString();
12    }
13    else {
14        elements.Add(new LexicalComponent(lexeme, token, (int)Grammar.Number));
15        state = 0; lexeme = ""; token = "";
16    }
17 break;
18
19 case 3:
20     if (char.IsDigit(line[i])) {
21         state = 3;
22         lexeme += line[i];
23         token = Grammar.Number.ToString();
24     }
25     else {
26        elements.Add(new LexicalComponent(lexeme, token, (int)Grammar.Number));
27        state = 0; lexeme = ""; token = "";
28    }
29 break;

```

Imagen 8.Case 2 y 3. Reconocimiento de números reales y enteros.

Los case del 11 al 16 se encarga de reconocer operadores relacionales, ya que los operadores relaciones se suelen componer de dos caracteres.

```

1 case 11:
2     if (line[i] == '=') {
3         lexeme += line[i];
4         elements.Add(new LexicalComponent(lexeme, token, (int)Grammar.RelationalOperator));
5         state = 0; lexeme = ""; token = "";
6     }
7     break;
8
9 case 12:
10    if (line[i] == '=') {
11        lexeme += line[i];
12        elements.Add(new LexicalComponent(lexeme, token, (int)Grammar.RelationalOperator));
13        state = 0; lexeme = ""; token = "";
14    }
15    break;
16
17 case 13:
18    if (line[i] == '=') {
19        lexeme += line[i];
20        elements.Add(new LexicalComponent(lexeme, token, (int)Grammar.RelationalOperator));
21        state = 0; lexeme = ""; token = "";
22    }
23    break;
24
25 case 14:
26    if (line[i] == '=') {
27        lexeme += line[i];
28        token = Grammar.RelationalOperator.ToString();
29        elements.Add(new LexicalComponent(lexeme, token, (int)Grammar.RelationalOperator));
30        state = 0; lexeme = ""; token = "";
31    }
32    else {
33        token = Grammar.AssignmentOperator.ToString();
34        elements.Add(new LexicalComponent(lexeme, token, (int)Grammar.AssignmentOperator));
35    }
36    state = 0; lexeme = ""; token = "";
37    break;
38
39 case 15:
40    if (line[i] == '|') {
41        lexeme += line[i];
42        elements.Add(new LexicalComponent(lexeme, token, (int)Grammar.LogicOperator));
43        state = 0; lexeme = ""; token = "";
44    }
45    break;
46
47 case 16:
48    if (line[i] == '|') {
49        lexeme += line[i];
50        elements.Add(new LexicalComponent(lexeme, token, (int)Grammar.LogicOperator));
51        state = 0; lexeme = ""; token = "";
52    }
53    break;

```

Imagen 9. Casos del 11 al 16. Reconocimiento de operadores relacionales.

El reconocimiento de los demás operadores incluidos en la gramática se hace desde el mismo caso 0.



```

1 else if (line[i] == ';') {
2     lexeme += line[i];
3     elements.Add(new LexicalComponent(lexeme, Grammar.Semicolon.ToString(), (int)Grammar.Semicolon));
4     state = 0; lexeme = ""; token = "";
5     break;
6 }
7 else if (line[i] == ',') {
8     lexeme += line[i];
9     elements.Add(new LexicalComponent(lexeme, Grammar.Comma.ToString(), (int)Grammar.Comma));
10    state = 0; lexeme = ""; token = "";
11    break;
12 }
13 else if (line[i] == '(') {
14     lexeme += line[i];
15     elements.Add(new LexicalComponent(lexeme, Grammar.LeftParenthesis.ToString(), (int)Grammar.LeftParenthesis));
16     state = 0; lexeme = ""; token = "";
17     break;
18 }
19 else if (line[i] == ')') {
20     lexeme += line[i];
21     elements.Add(new LexicalComponent(lexeme, Grammar.RightParenthesis.ToString(), (int)Grammar.RightParenthesis));
22     state = 0; lexeme = ""; token = "";
23     break;
24 }
25 else if (line[i] == '{') {
26     lexeme += line[i];
27     elements.Add(new LexicalComponent(lexeme, Grammar.LeftCurlyBrace.ToString(), (int)Grammar.LeftCurlyBrace));
28     state = 0; lexeme = ""; token = "";
29     break;
30 }
31 else if (line[i] == '}') {
32     lexeme += line[i];
33     elements.Add(new LexicalComponent(lexeme, Grammar.RightCurlyBrace.ToString(), (int)Grammar.RightCurlyBrace));
34     state = 0; lexeme = ""; token = "";
35     break;
36 }
37 else if (line[i] == '$') {
38     lexeme += line[i];
39     elements.Add(new LexicalComponent(lexeme, Grammar.DollarSing.ToString(), (int)Grammar.DollarSing));
40     state = 0; lexeme = ""; token = "";
41     break;
42 }
43 else if (line[i] == '!') {
44     state = 11;
45     lexeme += line[i];
46     token = Grammar.RelationalOperator.ToString();
47     break;
48 }
49 else if (line[i] == '>') {
50     state = 12;
51     lexeme += line[i];
52     token = Grammar.RelationalOperator.ToString();
53     break;
54 }
55 else if (line[i] == '<') {
56     state = 13;
57     lexeme += line[i];
58     token = Grammar.RelationalOperator.ToString();
59     break;
60 }
61 else if (line[i] == '=') {
62     state = 14;
63     lexeme += line[i];
64     break;
65 }
66 else if (line[i] == '|') {
67     state = 15;
68     lexeme += line[i];
69     token = Grammar.LogicOperator.ToString();
70     break;
71 }
72 else if (line[i] == '&') {
73     state = 16;
74     lexeme += line[i];
75     token = Grammar.LogicOperator.ToString();
76     break;
77 }
78 else if (line[i] == '+' || line[i] == '-') {
79     lexeme += line[i];
80     elements.Add(new LexicalComponent(lexeme, Grammar.AdditionOperator.ToString(), (int)Grammar.AdditionOperator));
81     state = 0; lexeme = ""; token = "";
82     break;
83 }
84 else if (line[i] == '*' || line[i] == '/') {
85     lexeme += line[i];
86     elements.Add(new LexicalComponent(lexeme, Grammar.MultiplicationOperator.ToString(), (int)Grammar.MultiplicationOperator));
87     state = 0; lexeme = ""; token = "";
88     break;
89 }

```

Imagen 10. Reconocimiento de los demás operadores en el case 0.

La última clase utilizada es la propia del Windows Form, la cual inicializa la interfaz gráfica. Además, en el constructo de la clase se hace la instancia de la lista de componentes léxicos. El método `buttonLexicalAnalyzer_Click` guarda en una lista de strings el texto ingresado por el usuario, a cada línea se le agrega el operador “\$” que indica el final de una línea.

Cada una de las líneas se pasa al analizador léxico, y guarda en la lista de componentes los tokens que va reconociendo.

Finalmente, los componentes léxicos reconocidos se muestran en la tabla que está en la parte inferior de la interfaz.



```
1 public partial class Form1 : Form
2 {
3     List<LexicalComponent> lexicalComponents;
4
5     public Form1()
6     {
7         InitializeComponent();
8
9         lexicalComponents = new List<LexicalComponent>();
10    }
11
12    private void buttonLexicalAnalyzer_Click(object sender, EventArgs e)
13    {
14        List<string> lines = new List<String>();
15
16        foreach (string line in textBoxInput.Lines)
17            lines.Add(line + "$");
18
19        foreach (string line in lines)
20            Analyzers.LexicalAnalyze(line, lexicalComponents);
21
22        foreach (LexicalComponent component in lexicalComponents)
23            listView1.Items.Add(new ListViewItem(new String[] { component.Lexeme, component.Token, component.Number.ToString() }));
24    }
25 }
26 }
```

Imagen 11. Código de la clase de la interfaz gráfica.

## Pruebas de Ejecución

Al ejecutar la aplicación, ingresar texto en el input señalado, y presionar el botón “Analizador Léxico”, se ejecuta este análisis y se muestran los componentes léxicos del texto ingresado.

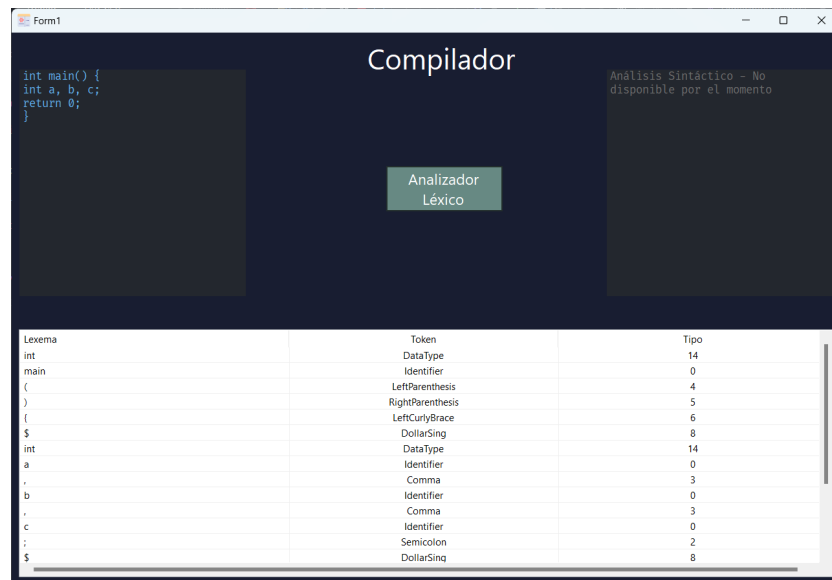


Imagen 12. Prueba de ejecución 1. Reconocimiento exitoso de componentes léxicos.

Aquí se muestra otro ejemplo de ejecución, en la que se reconocen exitosamente los componentes léxicos del texto ingresado.

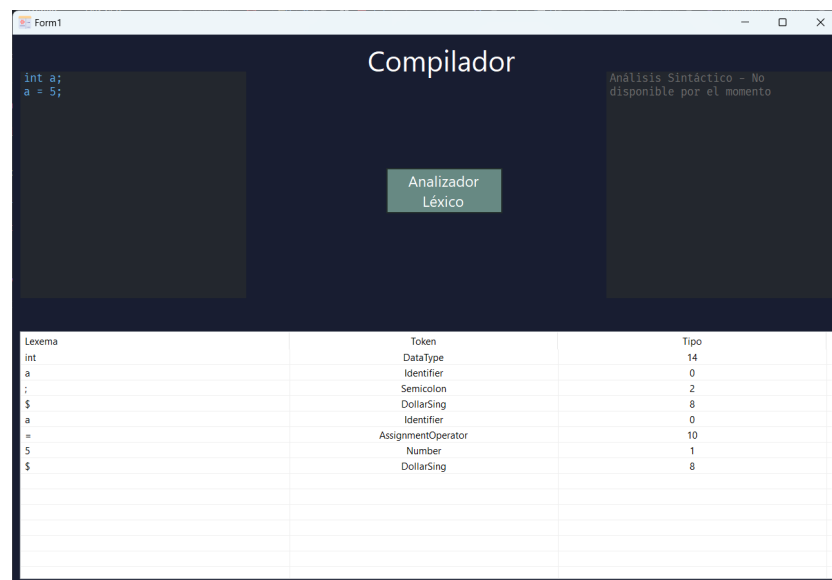


Imagen 13. Prueba de ejecución 2. Reconocimiento exitoso de componentes léxicos.

## **CONCLUSIONES PERSONALES**

### **Diego Alberto Martinez Zepeda:**

El trabajo llevado a cabo y presentado en este documento, permite concluir los conocimientos adquiridos y necesarios para desarrollar un analizador léxico donde se contempla la entrada en tiempo real de un usuario externo, haciendo uso de otras tecnologías y no solo de las que permiten el procesamiento, logramos desarrollar además una interfaz amigable al usuario.

### **Isaac Alberto Hernandez Chavez:**

Para llevar a cabo la actividad de desarrollar un analizador léxico como parte de la aplicación de fundamentos básicos en compiladores, fue necesario el comprender como funciona la lectura de flujo en un flujo de caracteres. demostramos que este es un proceso esencial para la correcta programación de programas informáticos, es bueno comprender algo tan común para nosotros como lo es un compilador.

### **Ismael Guadalupe Avila Guerrero**

Este trabajo nos permitió comprender lo complejo que es el diseño y desarrollo de compiladores. La principal complejidad fue el diseño del autómata, ya que es muy fácil perderse en la gran cantidad de estados que se llegan a tener. Una vez terminado el diseño, la implementación en código fue más sencilla. La importancia de este analizador radica en que el reconocimiento de los componentes léxicos se realice de manera correcta, ya que de ello dependen las demás etapas del proceso de compilación.

## REFERENCIAS BIBLIOGRÁFICAS