Universidad de San Carlos de Guatemala

Centro Universitario de Occidente -CUNOC-

Lenguajes Formales y de Programación “A”

Ing. Bryan Monzón

Manual técnico proyecto 1

Estudiante	**Carné**
Ronald Renán Samayoa Martínez	202031046

Quetzaltenango 9 de abril de 2025

INTRODUCCIÓN

El siguiente manual se ha desarrollado con la finalidad de dar a conocer la información necesaria para realizar mantenimiento, ejecución y exploración del código del proyecto 1, la cual consta de diferentes paquetes e interfaces gráficas para la ejecución de un analizador léxico para una gramática de Chomsky de tipo 3.

El manual ofrece la información necesaria de ¿cómo está realizado el código? para que una persona (Desarrollador en el lenguaje C# con .NET) que desee comprender y posteriormente editar el código lo haga de una manera apropiada, dando a conocer la estructura del desarrollo del aplicativo.

El manual técnico hace referencia a la información necesaria con el fin de orientar al personal en la concepción, planteamiento, análisis, programación e instalación del sistema. Es de notar que la redacción propia del manual técnico está orientada a personal con conocimientos en sistemas y tecnologías de informática y conocimientos de programación sobre el lenguaje C#con .NET. Para su desarrollo se ha decidido utilizar programación modular y la estructura modelo, vista y controlador (MVC). También se ha utilizado la herramienta WPF (Windows Presentation Foundation) para crear una interfaz gráfica más amigable con el usuario.

DESARROLLO

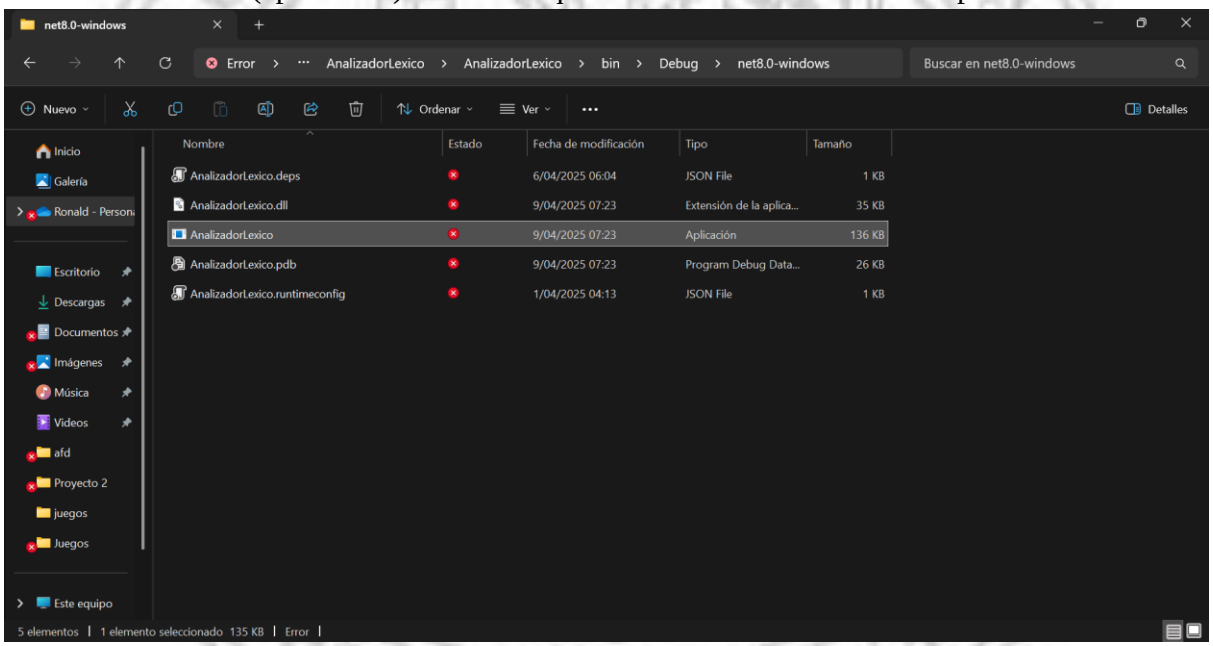
Para el desarrollo del código usando el lenguaje C# (C sharp), utilizando Microsoft Visual Studio en su versión 2022. Siendo compilado mediante el entorno de ejecución .NET en su version 8.0. Además de hacer uso de la tecnología de Visual Studio llamada WPF (Windows Presentation Foundation).

Prerrequisitos

- Mínimo 4 GB de memoria RAM
- Sistema operativo, preferentemente Linux o Windows (preferiblemente)
- Instalación de una versión de Microsoft Visual Studio, en especial sus componentes, específicamente para el desarrollo de C#

Utilización del programa

Para acceder al programa (mediante un archivo ejecutable .exe) se debe ubicar en la ruta AnalizadorLexico\AnalizadorLexico\bin\Debug\net8.0-windows y hacer doble click en AnalizadorLexico (aplicación). Esto hará que inmediatamente se abra la aplicación.



ALGORITMOS DE MODELAMIENTO

Para el desarrollo del código del proyecto 1, se ha hecho el uso del patrón de arquitectura Modelo Vista Controlador (MVC), al igual que conceptos básicos de programación modular. El archivo principal de arranque (App.xaml) se encuentra enraizados en la carpeta de origen (AnalizadorLexico) y también se realizaron las carpetas:

- Analizador
- Modelos
- Vistas

Desarrollo de algoritmos

A continuación, se presentan los algoritmos desarrollados para la creación de las carpetas, incluyendo sus archivos.

Clase App.xaml

Propósito: Inicializar la aplicación, para este caso, inicia directamente en la ventana Window1.xaml. Su ubicación es enraizada a la creación del proyecto.

```
<Application x:Class="AnalizadorLexico.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:AnalizadorLexico"
             StartupUri="Vistas/Window1.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Carpeta Analizador

Contiene la lógica principal del analizador léxico, responsable de identificar y clasificar los distintos tokens del código fuente.

Clase AnalizadorLexico.cs

Propósito: Ejecuta el análisis léxico sobre la entrada y retorna una lista de tokens reconocidos.

- Constructor AnalizadorLexico: Inicializa la lista de autómatas y configura el estado inicial (posición, línea, columna)
- Analizar(): Le permite al analizador saber si este autómata puede encargarse del carácter actual, y si ningún autómata lo reconoce llama a AutomataError
- ObtenerTokens(): Devuelve la lista de tokens generada

```
public AnalizadorLexico(string codigoFuente)
{
    automatas = new List<IAutomata>
    {
        new AutomataCaracterEspecial(),
        new AutomataComentarioLinea(),
        new AutomataComentarioBloque(),
        new AutomataLiteral(),
        new AutomataPalabraReservada(),
        new AutomataIdentificador(),
        new AutomataOperadorRelacional(),
        new AutomataOperadorLogico(),
        new AutomataOperadorAritmetico(),
        new AutomataOperadorAsignacion(),
        new AutomataNumero(),
        new AutomataSignosAgrupacion(),
        new AutomataSignosPuntuacion(),
        new AutomataError()
    };

    tokens = new List<Token>();
    entrada = codigoFuente;
    posicion = 0;
    linea = 1;
    columna = 1;
}
```

Clase IAutomata

Propósito: Es una interfaz que define el contrato base para todos los autómatas del analizador léxico. Cualquier clase que implemente esta interfaz debe proporcionar dos funcionalidades clave:

- Reconocer: Intenta reconocer un token válido a partir de la posición actual en la cadena de entrada

- PuedeAnalizar: Le permite al analizador saber si este autómata puede encargarse del carácter actual

```
using AnalizadorLexico.Modelos;

namespace AnalizadorLexico.Analizador
{
    16 referencias
    public interface IAutomata
    {
        16 referencias
        Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna);
        18 referencias
        bool PuedeAnalizar(char actual);
    }
}
```

Subcarpeta Automatas

Clase AutomataCaracterEspecial.cs

Propósito: Este autómata se encarga de reconocer y manejar los caracteres especiales que no son relevantes para el análisis léxico pero que deben ser procesados para mantener un seguimiento preciso de la posición en el código. Sus funciones principales son: Identificar espacios en blanco ( ), saltos de línea (\n), retornos de carro (\r), tabulaciones (\t) y saltos de página (\f).

```
public bool PuedeAnalizar(char actual)
{
    return actual == ' ' || actual == '\n' || actual == '\r' || actual == '\t' || actual == '\f';
}

2 referencias
public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
{
    if (posicion >= entrada.Length)
        return null;

    char actual = entrada[posicion];

    if (actual == '\n')
    {
        linea++;
        columna = 1;
    }
    else
    {
        columna++;
    }

    posicion++;

    // No se retorna token porque debe ser ignorado
    return new Token(TipoToken.CaracterEspecial, actual.ToString(), linea, columna);
}
```

Clase AutomataComentarioBloque.cs

Propósito: Este autómata se encarga de identificar y procesar comentarios de bloque en el código fuente (delimitados por /* y */). Sus funciones principales son **r**econocer el inicio de un posible comentario de bloque cuando encuentra el carácter '/', verificar su sintaxis y analizar todo el contenido del comentario hasta encontrar el cierre correspondiente (*/).

```
public bool PuedeAnalizar(char actual)
{
    return actual == '/';
}

2 referencias
public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
{
    int inicio = posicion;
    int columnaInicio = columna;

    if (posicion + 1 >= entrada.Length || entrada[posicion + 1] != '*')
        return null;

    // Avanza por '/*'
    Avanzar(entrada, ref posicion, ref linea, ref columna);
    Avanzar(entrada, ref posicion, ref linea, ref columna);

    while (posicion + 1 < entrada.Length)
    {
        if (entrada[posicion] == '*' && entrada[posicion + 1] == '/')
        {
            Avanzar(entrada, ref posicion, ref linea, ref columna);
            Avanzar(entrada, ref posicion, ref linea, ref columna);
            string lexema = entrada.Substring(inicio, posicion - inicio);
            return new Token(TipoToken.ComentarioBloque, lexema, linea, columnaInicio);
        }
```

Clase AutomataComentarioLinea.cs

Propósito: Este autómata se encarga de reconocer y procesar comentarios de línea en el código fuente (los que comienzan con # y terminan con salto de línea). Opera de la siguiente manera: Identifica el carácter '#' como inicio de comentario mediante el

método PuedeAnalizar(), una vez detectado el #, recorre todos los caracteres siguientes hasta encontrar un salto de línea ('\n') o llegar al final del archivo.

```
public class AutomataComentarioLinea : IAutomata
{
    2 referencias
    public bool PuedeAnalizar(char actual)
    {
        return actual == '#';
    }

    2 referencias
    public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
    {
        int inicio = posicion;
        int columnaInicio = columna;

        while (posicion < entrada.Length && entrada[posicion] != '\n')
        {
            Avanzar(entrada, ref posicion, ref linea, ref columna);
        }

        string lexema = entrada.Substring(inicio, posicion - inicio);
        return new Token(TipoToken.ComentarioLinea, lexema, linea, columnaInicio);
    }
```

Clase AutomataError.cs

Propósito: Este autómata funciona como mecanismo de "último recurso" para manejar caracteres no reconocidos en el código fuente. Su funcionalidad es estar dispuesto a analizar cualquier carácter, lo que lo convierte en el último autómata en la cadena de procesamiento.

```
public class AutomataError : IAutomata
{
    2 referencias
    public bool PuedeAnalizar(char actual)
    {
        // Siempre puede analizar, es el "último recurso"
        return true;
    }
    3 referencias
    public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
    {
        if (posicion >= entrada.Length)
            return null;

        int columnaInicio = columna;
        char simbolo = entrada[posicion];
        Avanzar(entrada, ref posicion, ref linea, ref columna);

        return new Token(TipoToken.Error, simbolo.ToString(), linea, columnaInicio);
    }
```

Clase AutomataIdentificador.cs

Propósito: Este autómata se especializa en reconocer identificadores que siguen un formato específico en el código fuente. Su operación se compone de: identifica el carácter '$' como señal de inicio y rechazar todo aquel que no inicie con él. Y permite combinaciones con letras (mayúsculas y minúsculas), números y caracteres especiales.

```
public bool PuedeAnalizar(char actual)
{
    return actual == '$';
}

2 referencias
public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
{
    int inicio = posicion;
    int columnaInicio = columna;

    if (entrada[posicion] != '$')
        return null;

    Avanzar(entrada, ref posicion, ref linea, ref columna); // Salta el '$'

    while (posicion < entrada.Length &&
          (EsLetra(entrada[posicion]) || EsNumero(entrada[posicion]) ||
          entrada[posicion] == '_' || entrada[posicion] == '-'))
    {
        Avanzar(entrada, ref posicion, ref linea, ref columna);
    }

    string lexema = entrada.Substring(inicio, posicion - inicio);
    return new Token(TipoToken.Identificador, lexema, linea, columnaInicio);
}
```

```
private bool EsLetra(char c) =>
    (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');

1 referencia
private bool EsNumero(char c) =>
    c >= '0' && c <= '9';
```

Clase AutomataLiteral.cs

Propósito: Este autómata está diseñado para reconocer literales de cadena (strings) en el código fuente, tanto con comillas dobles (") como simples ('). Captura todo el contenido entre las comillas de apertura y cierre. Maneja ambos tipos de comillas (consistentes: si empieza con ", debe terminar con ") y permite cualquier carácter dentro de la cadena excepto saltos de línea.

```
public bool PuedeAnalizar(char actual)
{
    return actual == '"' || actual == '\'';
}

2 referencias
public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
{
    if (posicion >= entrada.Length)
        return null;

    int inicio = posicion;
    int columnaInicio = columna;
    char comillaApertura = entrada[posicion];

    // Asegurarse que empieza con comilla válida
    if (comillaApertura != '"' && comillaApertura != '\'')
        return null;

    Avanzar(entrada, ref posicion, ref linea, ref columna); // Consumimos la comilla inicial

    while (posicion < entrada.Length)
    {
        char actual = entrada[posicion];

        if (actual == '\n')
        {
```

```
            // No se permite salto de línea en cadena
            return new Token(TipoToken.Error, entrada.Substring(inicio, posicion - inicio), linea, columna
        }

        if (actual == comillaApertura)
        {
            // Fin de la cadena
            Avanzar(entrada, ref posicion, ref linea, ref columna); // Consumimos la comilla de cierre
            string lexema = entrada.Substring(inicio, posicion - inicio);
            return new Token(TipoToken.Literal, lexema, linea, columnaInicio);
        }

        Avanzar(entrada, ref posicion, ref linea, ref columna);
    }

    // Si llegamos al final sin encontrar comilla de cierre
    string errorLexema = entrada.Substring(inicio, posicion - inicio);
    return new Token(TipoToken.Error, errorLexema, linea, columnaInicio);
}
```

Clase AutomataNumero.cs

Propósito: Este autómata está diseñado para reconocer y validar números en el código fuente, tanto enteros como decimales, con las siguientes características: Reconoce números cuando encuentra dígitos del 0 al 9 con o sin signo positivo o negativo, no permite ceros a la izquierda, y para los números decimales: detecta el punto decimal como separador y valida que tenga al menos un dígito antes y después del punto.

```
public bool PuedeAnalizar(char actual)
{
    return actual == '+' || actual == '-' || (actual >= '0' && actual <= '9');
}

2 referencias
public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
{
    int inicio = posicion;
    int columnaInicio = columna;
    bool esDecimal = false;

    // Paso 1: signo opcional (solo si va seguido de dígito)
    if (entrada[posicion] == '+' || entrada[posicion] == '-')
    {
        if (posicion + 1 >= entrada.Length || !EsNumero(entrada[posicion + 1]))
        {
            return null; // No es número, dejar c
        }
        Avanzar(entrada, ref posicion, ref linea, ref columna);
        if (posicion >= entrada.Length) return null;
    }
```

bool AutomataNumero.EsNumero(char c)

```
    else
    {
        // Primer dígito distinto de cero, avanzar por los enteros
        while (posicion < entrada.Length && EsNumero(entrada[posicion]))
            Avanzar(entrada, ref posicion, ref linea, ref columna);

        // Si hay punto → es decimal
        if (posicion < entrada.Length && entrada[posicion] == '.')
        {
            esDecimal = true;
            Avanzar(entrada, ref posicion, ref linea, ref columna);

            if (posicion >= entrada.Length || !EsNumero(entrada[posicion]))
                return null;

            while (posicion < entrada.Length && EsNumero(entrada[posicion]))
                Avanzar(entrada, ref posicion, ref linea, ref columna);
        }
    }

    string lexema = entrada.Substring(inicio, posicion - inicio);
    return new Token(esDecimal ? TipoToken.NumeroDecimal : TipoToken.NumeroEntero, lexema, linea, columnaInicio);
}

8 referencias
private bool EsNumero(char c)
{
    return c >= '0' && c <= '9';
}
```

Clase AutomataOperadorAritmetico.cs

Propósito: Este autómata se encarga de reconocer los operadores aritméticos básicos en el código fuente. Su funcionamiento se caracteriza por una detección simple y eficiente de los caracteres + (suma), - (resta), * (multiplicación), / (división), ^ (potencia). Y procesa un carácter a la vez.

```
public bool PuedeAnalizar(char actual)
{
    return actual == '+' || actual == '-' || actual == '*' || actual == '/' || actual == '^';
}

2 referencias
public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
{
    int inicio = posicion;
    int columnaInicio = columna;

    if (posicion >= entrada.Length)
        return null;

    char actual = entrada[posicion];

    if (PuedeAnalizar(actual))
    {
        Avanzar(entrada, ref posicion, ref linea, ref columna);
        string lexema = entrada.Substring(inicio, 1);
        return new Token(TipoToken.OperadorAritmetico, lexema, linea, columnaInicio);
    }

    return null;
}
```

Clase AutomataOperadorAsignacion.cs

Propósito: Este autómata está diseñado específicamente para reconocer el operador de asignación simple en el código fuente. Su comportamiento se caracteriza por identificar exclusivamente el carácter '=' mediante el método de análisis.

```
public bool PuedeAnalizar(char actual)
{
    return actual == '=';
}

2 referencias
public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
{
    int inicio = posicion;
    int columnaInicio = columna;

    if (posicion < entrada.Length && entrada[posicion] == '=')
    {
        Avanzar(entrada, ref posicion, ref linea, ref columna);
        return new Token(TipoToken.OperadorAsignacion, "=", linea, columnaInicio);
    }

    return null;
}
```

Clase AutomataOperadorLogico.cs

Propósito: Este autómata está diseñado para reconocer operadores lógicos en el código fuente, tanto en formato de símbolos como de palabras reservadas. Su funcionamiento se caracteriza por identificar los caracteres iniciales de operadores lógicos (&, |, A, O) mediante el método de análisis y actuar como puerta de entrada para cuatro tipos de operadores lógicos diferentes.

```
public bool PuedeAnalizar(char actual)
{
    return actual == '&' || actual == '|' || actual == 'A' || actual == 'O';
}

2 referencias
public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
{
    int estado = 0;
    int inicio = posicion;
    int columnaInicio = columna;
    string lexema = "";

    while (posicion < entrada.Length)
    {
        char actual = entrada[posicion];

        switch (estado)
        {
            case 0:
                if (actual == '&') { estado = 1; lexema += actual; }
                else if (actual == '|') { estado = 3; lexema += actual; }
                else if (actual == 'A') { estad                          }
                else if (actual == 'O') { estad                          }
                else return null;
                break;
```

(variable local) int estado

```
    // &&
    case 1:
        if (actual == '&') { estado = 2; lexema += actual; }
        else return null;
        break;

    case 2:
        Avanzar(entrada, ref posicion, ref linea, ref columna);
        return new Token(TipoToken.OperadorLogico, "&&", linea, columnaInicio);

    // ||
    case 3:
        if (actual == '|') { estado = 4; lexema += actual; }
        else return null;
        break;

    case 4:
        Avanzar(entrada, ref posicion, ref linea, ref columna);
        return new Token(TipoToken.OperadorLogico, "||", linea, columnaInicio);

    // AND
    case 5:
        if (actual == 'N') { estado = 6; lexema += actual; }
        else return null;
        break;

    case 6:
        if (actual == 'D') { estado = 7; lexema += actual; }
        else return null;
```

```
            case 7:
                Avanzar(entrada, ref posicion, ref linea, ref columna);
                return new Token(TipoToken.OperadorLogico, "AND", linea, columnaInicio);

            // OR
            case 8:
                if (actual == 'R') { estado = 9; lexema += actual; }
                else return null;
                break;

            case 9:
                Avanzar(entrada, ref posicion, ref linea, ref columna);
                return new Token(TipoToken.OperadorLogico, "OR", linea, columnaInicio);
        }

        Avanzar(entrada, ref posicion, ref linea, ref columna);
    }

    return null;
}
```

Clase AutomataOperadorRelacional.cs

Propósito: Este autómata se especializa en reconocer operadores relacionales que comparan valores en el código fuente. Su operación se caracteriza por identificar los caracteres '<' y '>' mediante el método de análisis y el reconocimiento de sus dos variantes, simples ('<' y '>') y dobles ('<=' y '>=').

```
public bool PuedeAnalizar(char actual)
{
    return actual == '>' || actual == '<';
}
```

```
public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
{
    if (posicion >= entrada.Length)
        return null;

    int inicio = posicion;
    int columnaInicio = columna;

    char actual = entrada[posicion];

    if (actual == '>' || actual == '<')
    {
        Avanzar(entrada, ref posicion, ref linea, ref columna);

        if (posicion < entrada.Length && entrada[posicion] == '=')
        {
            Avanzar(entrada, ref posicion, ref linea, ref columna);
            string lexema = entrada.
            return new Token(TipoTok  (parámetro) ref int posicion  xema, linea, columnaInicio);
        }

        string simple = entrada.Substring(inicio, 1);
        return new Token(TipoToken.OperadorRelacional, simple, linea, columnaInicio);
    }

    return null;
}
```

Clase AutomataPalabraReservada.cs

Propósito: Este autómata se encarga de identificar palabras reservadas del lenguaje de programación. Su funcionamiento se caracteriza por reconocer cualquier carácter alfabético (a-z, A-Z) como posible inicio de palabra reservada. También maneja un conjunto predefinido de 25 palabras reservadas comunes como estructuras de control, modificadores de acceso, tipos de datos, o palabras especiales; distinguiendo entre mayúsculas y minúsculas.

```
private readonly string[] palabrasReservadas = new string[]
{
    "if", "class", "for", "then", "else", "public", "private", "package",
    "import", "static", "void", "int", "true", "false", "extends", "short",
    "boolean", "float", "interface", "final", "protected", "return", "while",
    "case", "implements"
};

2 referencias
public bool PuedeAnalizar(char actual)
{
    return EsLetra(actual); // Comienzan con letra obligatoriamente
}
```

```
public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
{
    int inicio = posicion;
    int columnaInicio = columna;

    while (posicion < entrada.Length && EsLetra(entrada[posicion]))
    {
        Avanzar(entrada, ref posicion, ref linea, ref columna);
    }

    string lexema = entrada.Substring(inicio, posicion - inicio);

    // Validar si es palabra reservada exacta
    foreach (var palabra in palabrasReservadas)
    {
        if (lexema == palabra)
        {
            return new Toke                                  a, lexema, linea, columnaInicio);
        }
    }

    // No es palabra reservada
    posicion = inicio;
    columna = columnaInicio;
    return null;
}
2 referencias
private bool EsLetra(char c) =>
    (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');
```

(variable local) string? palabra

"palabra" no es NULL aquí.

Clase AutomataSignosAgrupacion.cs

Propósito: Este autómata se especializa en reconocer los signos de agrupación utilizados en el código fuente. Su operación se caracteriza por identifica 6 símbolos de agrupación: Paréntesis: (y), llaves: { y }, corchetes: [y]. Reconociendo cada símbolo individualmente, aunque no maneja pares combinados.

```
public bool PuedeAnalizar(char actual)
{
    return actual == '(' || actual == ')' ||
           actual == '{' || actual == '}' ||
           actual == '[' || actual == ']';
}

2 referencias
public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
{
    int columnaInicio = columna;

    if (posicion < entrada.Length)
    {
        char actual = entrada[posicion];

        if (PuedeAnalizar(actual))
        {
            Avanzar(entrada, ref posicion, ref linea, ref columna);
            return new Token(TipoToken.SignoAgrupacion, actual.ToString(), linea, columnaInicio);
        }
    }

    return null;
}
```

Clase AutomataSignosPuntuacion.cs

Propósito: Este autómata está diseñado para reconocer signos de puntuación utilizados en el código fuente. Su operación se compone de identifica 4 signos de puntuación: Punto (.), coma (,), dos puntos (:), y punto y coma (;).

```
public bool PuedeAnalizar(char actual)
{
    return actual == '.' || actual == ',' || actual == ':' || actual == ';';
}

2 referencias
public Token? Reconocer(string entrada, ref int posicion, ref int linea, ref int columna)
{
    int columnaInicio = columna;

    if (posicion < entrada.Length)
    {
        char actual = entrada[posicion];

        if (PuedeAnalizar(actual))
        {
            Avanzar(entrada, ref posicion, ref linea, ref columna);
            return new Token(TipoToken.SignoPuntuacion, actual.ToString(), linea, columnaInicio);
        }
    }

    return null;
}
```

Carpeta Modelos

Define las estructuras base utilizadas por el analizador.

Clase TipoToken.cs

Propósito: Define un enum con las categorías de tokens reconocidos.

```
namespace AnalizadorLexico.Modelos
{
    27 referencias
    public enum TipoToken
    {
        CaracterEspecial, // Espacio, \n, \t, \r, \f
        ComentarioLinea,  // Empieza con #
        ComentarioBloque, // /* ... */ con saltos de línea
        NumeroDecimal, // Decimales con o sin signo (+ o -)
        NumeroEntero, // Enteros con o sin signo (+ o -)
        Identificador, // Variables que empiezan con $
        Literal, // Cadenas de texto entre " o '
        OperadorAsignacion, // =
        OperadorAritmetico, // +, -, *, /, ^
        OperadorLogico, // &&, ||, AND, OR
        OperadorRelacional, // <, >, <=, >=
        PalabraReservada, // if, else, while, return...
        SignoAgrupacion, // ( ) { } [ ]
        SignoPuntuacion, // ; , : .
        Error, // Cualquier otro símbolo no reconocido
        FinDeLinea // Para manejo explícito de \n
    }
}
```

Clase Token.cs

Propósito: Representa un token reconocido, incluyendo información como:

- Valor
- Tipo (según TipoToken)
- Línea y columna donde fue encontrado

```
namespace AnalizadorLexico.Modelos
{
    45 referencias
    public class Token
    {
        4 referencias
        public TipoToken Tipo { get; }
        2 referencias
        public string Valor { get; }
        2 referencias
        public int Linea { get; }  // Nueva propiedad para rastrear línea
        2 referencias
        public int Columna { get; } // Nueva propiedad para rastrear columna

        22 referencias
        public Token(TipoToken tipo, string valor, int linea, int columna)
        {
            Tipo = tipo;
            Valor = valor;
            Linea = linea;
            Columna = columna;
        }

        0 referencias
        public override string ToString()
        {
            return $"[Linea{Linea},Columna{Columna}] {Tipo}: {Valor}";
        }
    }
}
```

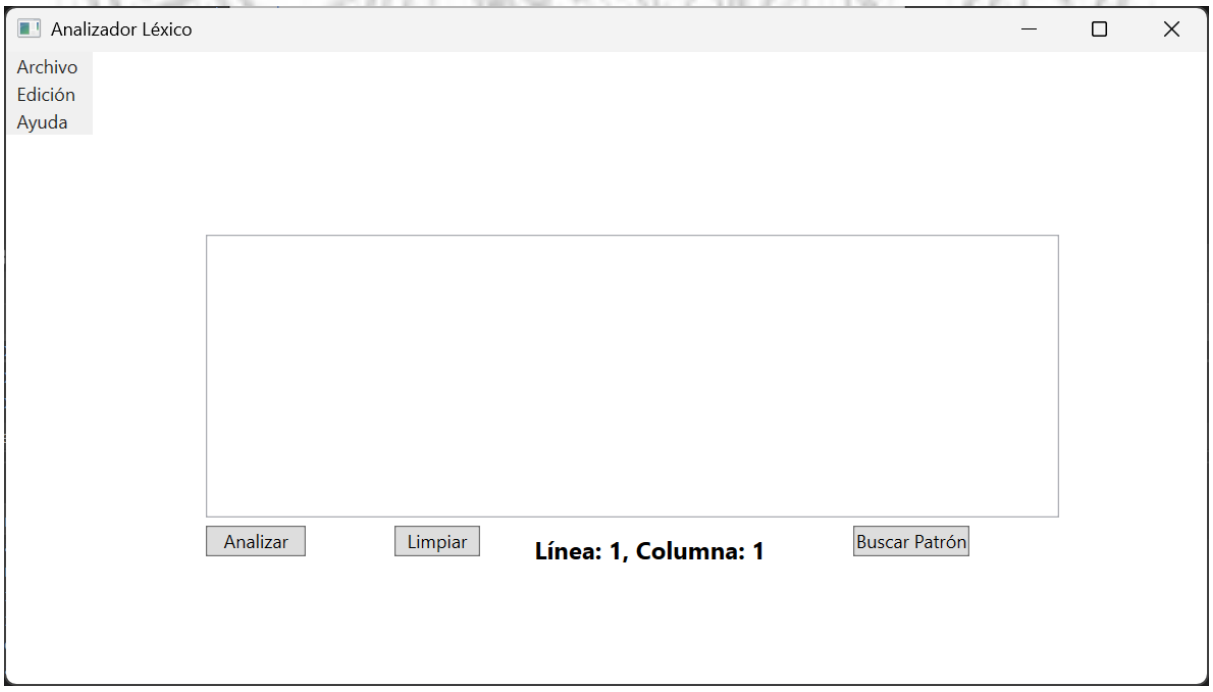
Carpeta Vistas

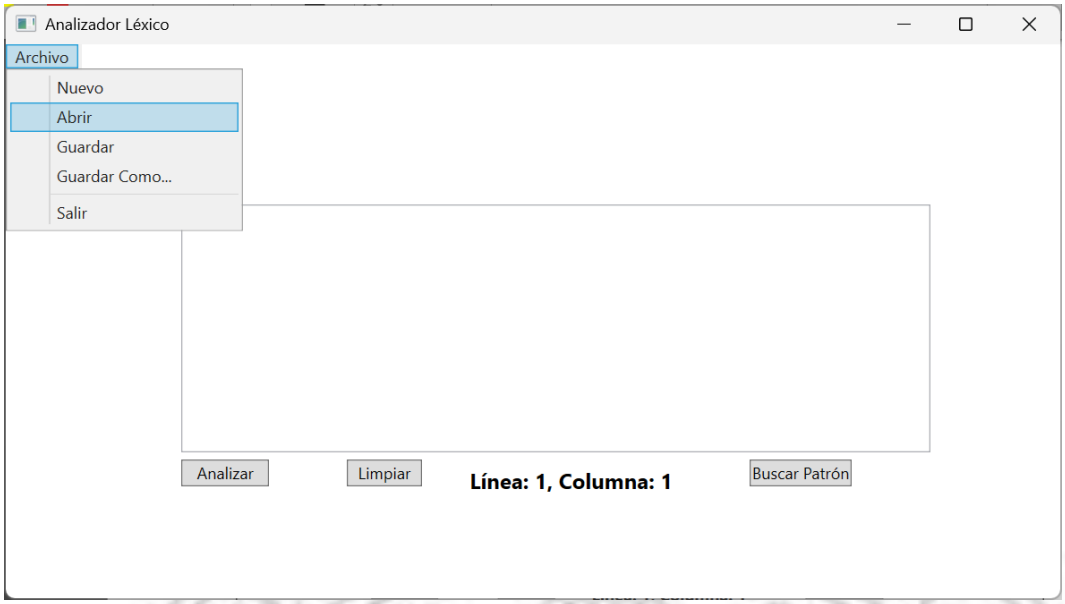
Clase Window1.xaml

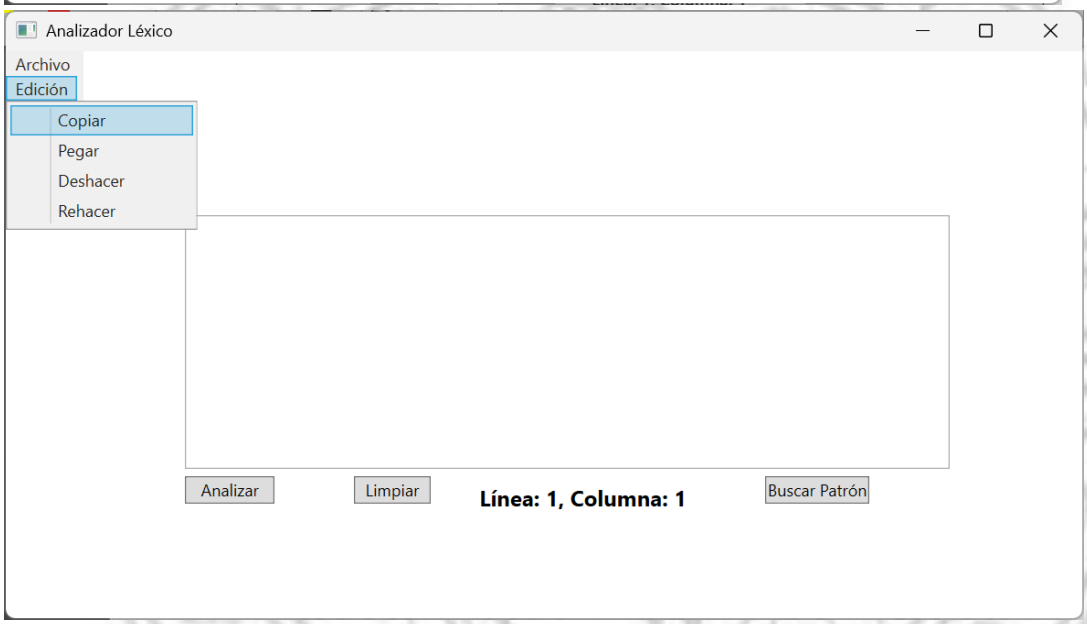
Propósito: Define los estilos visuales de la interfaz del mini analizador lexico. Su funcion principal es mejorar la apariencia de la aplicacion y hacerla más intuitiva para el usuario.

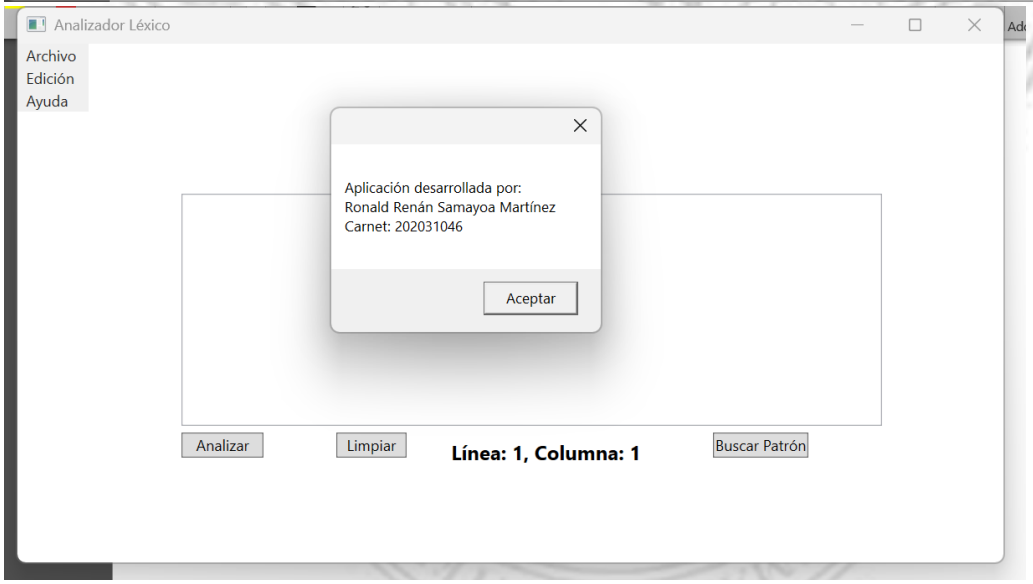
Clase Window1.xaml.cs

Es la ventana principal, la cual permite realizar el análisis léxico a través de un cuadro de texto, posee pestañas que permiten iniciar nuevos análisis, guardar, editar, y posee información del desarrollador del sistema.




Analizador Léxico
Archivo
Nuevo
Abrir
Guardar
Guardar Como...
Salir
Analizar
Limpiar
Línea: 1, Columna: 1
Buscar Patrón


Analizador Léxico
Archivo
Edición
Copiar
Pegar
Deshacer
Rehacer
Analizar
Limpiar
Línea: 1, Columna: 1
Buscar Patrón


Analizador Léxico
Archivo
Edición
Ayuda
Aplicación desarrollada por:
Ronald Renán Samayoa Martínez
Carnet: 202031046
Aceptar
Analizar
Limpiar
Línea: 1, Columna: 1
Buscar Patrón

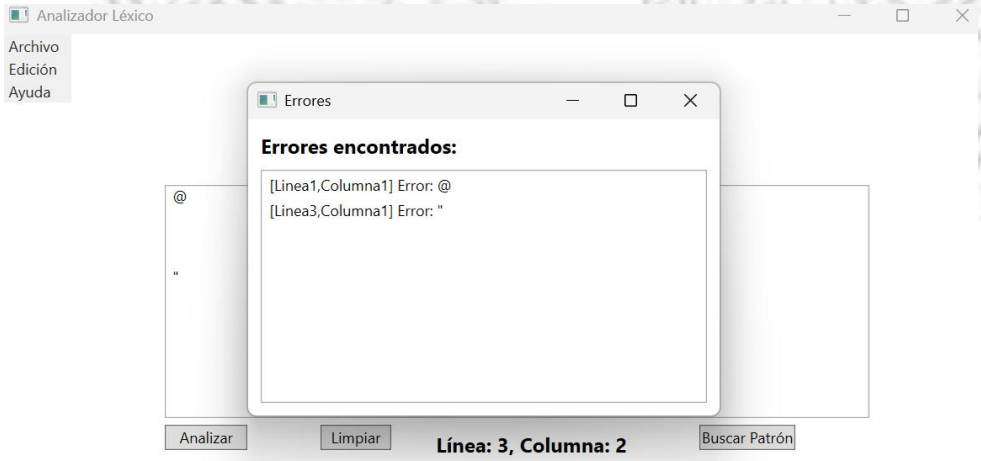
Window1.xaml.cs

Propósito: Esta clase representa la ventana principal de la aplicación y gestiona toda la interacción del usuario con el editor de código y el análisis léxico. Sus componentes clave son:

1. **Gestión de Archivos:**
    - Nuevo_Click(): Crea un nuevo documento, con confirmación si hay cambios sin guardar
    - Abrir_Click(): Permite abrir archivos de texto (.txt) mostrando advertencia si hay cambios pendientes
    - Guardar_Click()/GuardarComo_Click(): Guarda el contenido en la ruta actual o abre diálogo para nueva ruta
    - GuardarArchivo(): Escribe el contenido del editor en el archivo especificado
2. **Control de Edición:**
    - Copiar_Click()/Pegar_Click(): Funciones básicas de portapapeles
    - Deshacer_Click()/Rehacer_Click(): Manejo del historial de edición
    - Limpiar_Click(): Borra todo el contenido del editor
    - txtCodigoFuente_TextChanged: Detecta cambios para marcar el documento como no guardado
3. **Análisis Léxico:**
    - btnAnalizar_Click(): Ejecuta el análisis en segundo plano y muestra resultados:
        - Abre ventana de errores si se detectan
        - Muestra ventana de tokens si el análisis es exitoso
    - Usa Task.Run para evitar bloqueos de la interfaz durante el análisis

Window2.xaml

Propósito: Es la ventana de visualización de errores dentro del texto ingresado



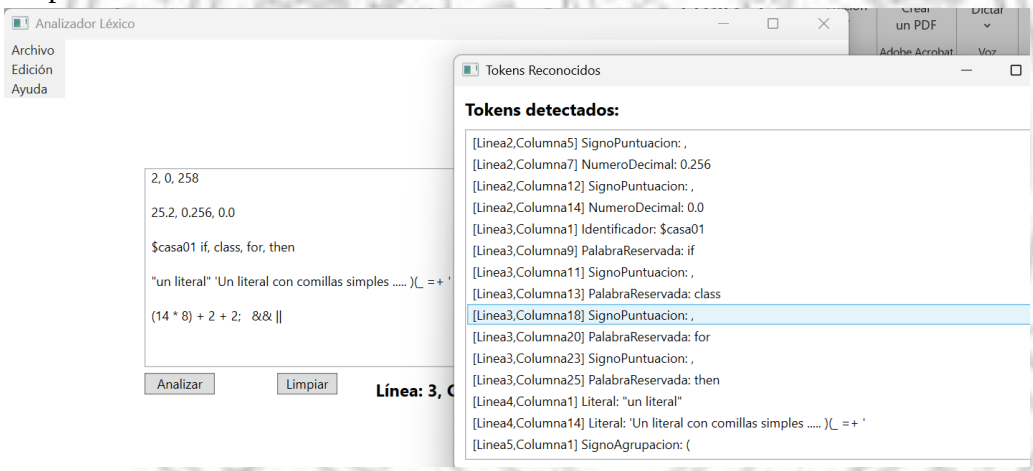
Window2.xaml.cs

Propósito: Esta clase representa una ventana especializada para mostrar los errores detectados durante el análisis léxico. Su diseño y funcionalidad son:

1. **Propósito Principal:**
    - Visualizar la lista de errores léxicos encontrados durante el análisis del código fuente
    - Presentar información clara y estructurada sobre cada error detectado
2. **Componentes Clave:**
    - lstErrores: ListView que muestra la colección de tokens de error
    - Constructor que recibe e inicializa la lista de errores
3. **Flujo de Operación:**
    - Recibe una lista de tokens de tipo Error al crearse
    - Asigna esta lista como fuente de datos al ListView (ItemsSource)
    - Muestra cada error con su información asociada
4. **Estructura de Datos:**
    - Trabaja con objetos Token que deben tener:
        - TipoToken = Error
        - Valor = Texto del lexema problemático
        - Linea/Columna = Ubicación del error en el código

Window3.xaml

Propósito: Es la ventana de visualización de tokens reconocidos dentro del texto ingresado



Window3.xaml.cs

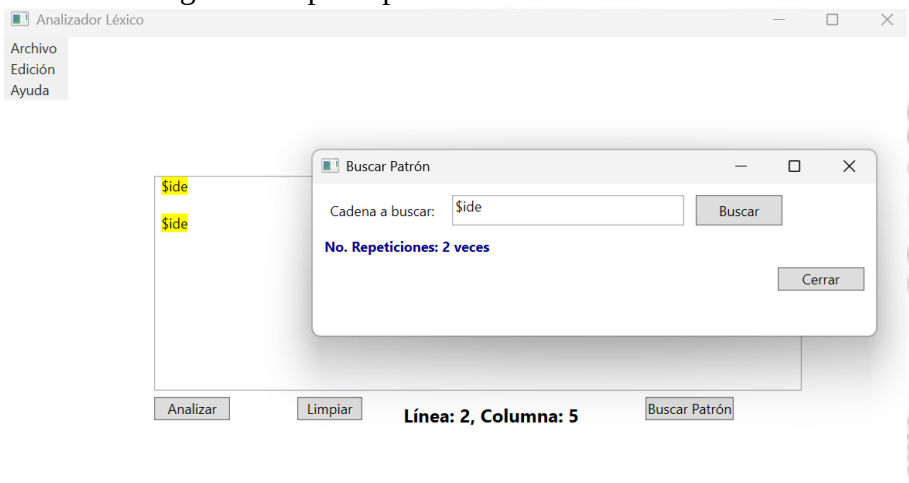
Propósito: Esta clase representa una ventana especializada para mostrar los tokens detectados durante el análisis léxico. Su diseño y funcionalidad son:

1. **Propósito Principal:**
    - Visualizar la lista de tokens encontrados durante el análisis del código fuente
    - Presentar información clara y estructurada sobre cada token detectado
2. **Componentes Clave:**
    - lstTokens: ListView que muestra la colección de tokens reconocidos
    - Constructor que recibe e inicializa la lista de tokens
3. **Flujo de Operación:**
    - Recibe una lista de tokens según su tipo al crearse
    - Asigna esta lista como fuente de datos al ListView (ItemsSource)
    - Muestra cada token con su información asociada

4. **Estructura de Datos:**
    - Trabaja con objetos Token que deben tener:
        - TipoToken = Error
        - Valor = Texto del lexema
        - Linea/Columna = Ubicación del token en el código

Window4.xaml

Propósito: Esta clase implementa una ventana de búsqueda de patrones de texto dentro del editor de código fuente principal



Window4.xaml.cs

Propósito principal: Permitir búsqueda y resaltado de patrones de texto en el editor de código y mostrar estadísticas de coincidencias encontradas

1. **Componentes Clave:**
    - editor: Referencia al RichTextBox principal (inyectado en el constructor)
    - txtPatron: Campo de texto para ingresar el patrón a buscar
    - txtResultado: Área para mostrar resultados numéricos
    - Botones de acción (Buscar, Cerrar)
2. **Funcionalidades de Búsqueda:**
    - Validación de patrones no vacíos (ignora espacios/tabulaciones)
    - Resaltado visual (amarillo) de todas las coincidencias
    - Conteo preciso de repeticiones del patrón
    - Distinción entre mayúsculas/minúsculas (comparación cultural exacta)
3. **Validaciones**:
    - Rechaza patrones vacíos o que solo contengan espacios
    - Muestra mensaje informativo cuando el patrón es inválido
4. **Interacción con el usuario**:
    - Distinción entre mayúsculas/minúsculas (comparación cultural exacta)
    - Muestra resultados en formato claro (ej: "3 veces")
    - Proporciona feedback visual inmediato
    - Permite múltiples búsquedas sin cerrar la ventana