

Tabla de símbolos en C# y su interacción con el analizador léxico.

FRANCISCO RÍOS ACOSTA
Instituto Tecnológico de la Laguna
Blvd. Revolución y calzada Cuauhtémoc s/n
Colonia centro
Torreón, Coah; México
Contacto : friosam@prodigy.net.mx

Resúmen. Se presenta la implementación de una tabla de símbolos en C#. Generalmente la explicación en clase de este tema, conlleva una buena cantidad de tiempo que podemos reducir, si empleamos un buen material donde las especificaciones de la tabla, estén previamente establecidas de manera clara. Un análisis de los atributos que almacena cada elemento de la tabla de símbolos, requiere de una definición clara acerca de lo que queremos almacenar y recuperar. Este trabajo propone una clase denominada *TablaSimbolos* con atributos y métodos que permitan interactuar con las 3 etapas de análisis de un compilador : análisis léxico, análisis sintáctico y análisis semántico. Como ejemplo, se construye una aplicación que analiza léxicamente una entrada usando un objeto oAnaLex perteneciente a la clase Lexico propuesta por R.A. Francisco, que interactúa con un objeto de la clase TablaSimbolos instalando identificadores, números y cadenas. La implementación de la tabla de símbolos se ha hecho de acuerdo a la teoría expresada en el libro del “dragón” Aho, Sethi y Ullman, así como del libro de Tremblay y Sorenson, ambos libros tratantes del tema de Compiladores.

1 Introducción.

Existen varias maneras de implementar una tabla de símbolos. En este trabajo trataremos una de ellas : abstraer la tabla de símbolos como un objeto. Nos basaremos en la teoría expuesta en el libro del “dragón” y del Tremblay para realizar la construcción de la tabla de símbolos. Estableceremos la propuesta de la clase *TablaSimbolos* de manera que podamos definir un objeto **oTablaSimb**, que interactúe en una aplicación Windows C# con un objeto **oAnaLex** –analizador léxico-. El análisis léxico lo haremos sobre sentencias de entrada cuya sintaxis está definida por una gramática de contexto libre no ambigua. Al término del trabajo tendremos construída una aplicación Windows C# que : inicialice la tabla de símbolos, que permita que el analizador léxico instale identificadores, números y cadenas en ella, y que visualice a los elementos de la tabla mostrando los atributos almacenados en la etapa de análisis léxico.

Los pasos que seguiremos en las secciones siguientes son :

- Modelo conceptual de una tabla de símbolos.
- Lenguaje de ejemplo y la gramática de contexto libre que denota su sintaxis.
- Análizador léxico para la gramática.
- Análisis de los atributos de los objetos que se almacenan en la tabla de símbolos.
- Clase *TablaSimbolos*.
- Instalación de identificadores, cadenas y números.
- Visualización de la tabla de símbolos.

2 Modelo conceptual de la tabla de símbolos.

Básicamente la tabla de símbolos es un arreglo de listas enlazadas. Inicialmente las listas enlazadas deberán estar vacías o sea que la referencia de su cabecera tiene el valor de null.

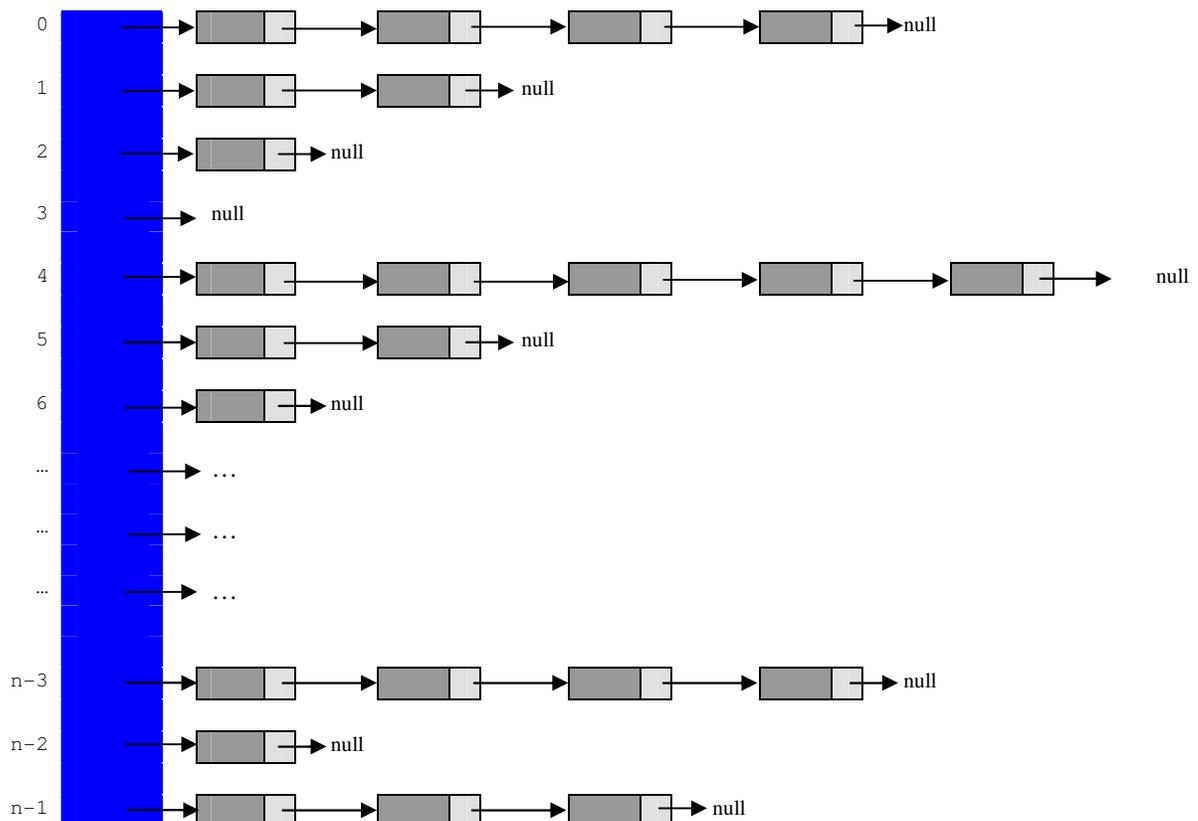


Fig. No. 2.1 Modelo conceptual de la tabla de símbolos para desmenuzamiento de n entradas.

Cada nodo en cada lista enlazada representa un símbolo instalado en la tabla por cualquier etapa de la fase de análisis, sea léxico, sintáctico o semántico. Esta fuera del alcance de estos apuntes, la fase de síntesis.

La inserción de los elementos en la tabla de símbolos requiere de definir una función de desmenuzamiento –hashing-. Cada símbolo antes de insertarlo en la lista enlazada correspondiente, debe ser procesado por esta función que será la encargada de decidir hacia que lista enlazada debe instalarse el símbolo en la entrada.

De acuerdo a lo anterior, si deseamos construir una tabla de símbolos sigamos las etapas que a continuación listamos :

- Definir la función de desmenuzamiento.
- Definir la clase Lista.
- Definir la clase Nodo.
- Definir la clase TipoElem.
- Definir la clase TablaSimbolos.

3 Lenguaje de ejemplo y su gramática de contexto libre que lo denota.

Durante algunos semestres hemos propuesto un lenguaje para el estudio de las diferentes fases de análisis, consistente de las instrucciones :

- Declaración de constantes
- Declaración de variables
- Asignación
- Entrada
- Salida

Las reglas de sintaxis para incluir las sentencias en este lenguaje de programación propuesto son :

- El cuerpo del programa es encerrado entre 2 sentencias : *inicio* y *fin*.
- La declaración de constantes debe ir en primer lugar antes de las declaraciones de variables, sentencias de asignación, de entrada y de salida. Puede no incluirse la declaración de constantes.
- La declaración de variables es la segunda en orden de inserción. Si existe una declaración de constantes, la declaración de variables debe incluirse después de la de constantes. Puede no existir una declaración de variables.
- Después de haber incluido la declaración de constantes y de variables, podemos incluir sentencias de asignación, de entrada y/o de salida.
- En sentencias de lectura o entrada de datos, sólo puede leerse un dato en la sentencia.
- En sentencias de visualización o de salida, pueden incluirse constantes, variables, pero no expresiones.
- En sentencias de asignación, sólo se permiten operaciones con valores numéricos. Los valores tipo cadena sólo se permite asignarlos no operarlos, es decir, los operadores +, -, *, y / no se usan para expresiones con datos cadena.

Ejemplos de código para este lenguaje los mostramos a continuación. Notemos que manejamos 3 tipos de datos :

- entero
- real
- cadena

Ejo. 1.

```
inicio
  const
    entero MAX=10;
    cadena MENSAJE="BIENVENIDOS AL SISTEMA";
    real PI=3.1416;

  visua "el valor de pi es = ",PI;
fin
```

Ejo. 2.

```
inicio
  var
    entero i,cont;
    cadena s1,s2,s3;
    real x,y,area,radio;
```

```

visua "teclea el valor de i : ";
leer i;
visua "teclea el valor del radio : ";
leer radio;
s1 = "hola";
i = i + 1;
fin

```

Ejo. 3.

```

inicio
    visua "HOLA MUNDO ";
fin

```

Ejo. 4.

```

inicio
    const
        entero MAX=10;
        cadena MENSAJE="BIENVENIDOS AL SISTEMA";
        real PI=3.1416;

    var
        entero i,cont;
        cadena s1,s2,s3;
        real x,y,area,radio;

    visua "teclea el valor de i : ";
    leer i;
    visua "teclea el valor del radio : ";
    leer radio;
    area = PI * radio * radio;
fin

```

La gramática propuesta para denotar a este lenguaje es :

```

P -> inicio C fin

C -> K S | V S | K V S | S

K -> const R

R -> R Y id = Z ; | Y id = Z ;

Y -> entero | cadena | real

Z -> num | cad

V -> var B

B -> B Y I ; | Y I ;

I -> I , id | id

S -> S A | S L | S O | A | L | O

A -> id = E ; | id = cad;

E -> E + T | E - T | T

T -> T * F | T / F | F

F -> id | num | ( E )

L -> leer id ;

O -> visua W ;

```

Tabla de símbolos en C# y su interacción con el analizador léxico.

W -> W , id | W , cad | W , num | id | num | cad

Usemos cualquiera de los 2 softwares didácticos RD-NRP o bien el RA-SLR para reconocer los ejemplos del 1 al 4. Antes deberemos teclear la gramática según se muestra en la figura #3.1 en el programa RD-NRP.

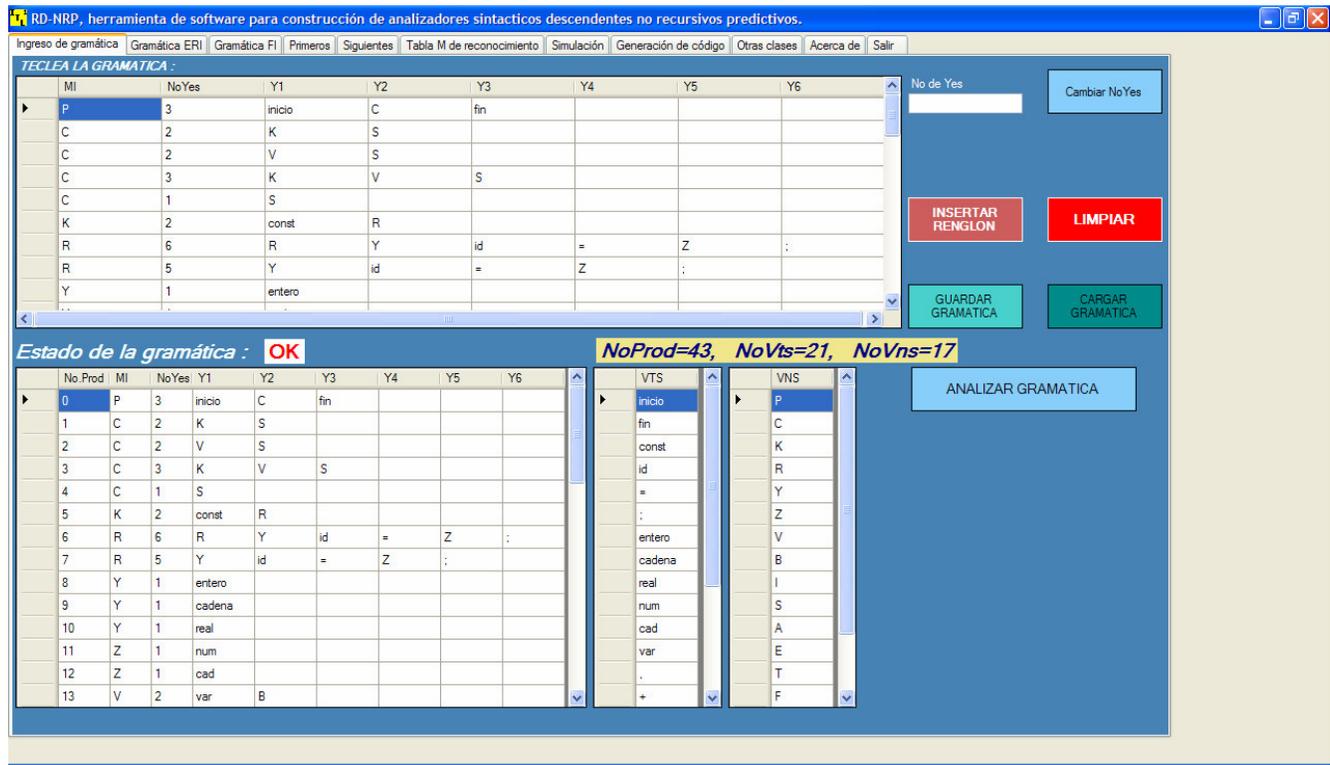


Fig. No. 3.1 Análisis de la gramática propuesta usando el RD-NRP.

La gramática transformada con E.R.I. y F.I. agrupada, obtenida con el RD-NRP es :

```

P -> inicio C fin
C -> K C' | V S | S
C' -> S | V S
K -> const R
R -> Y id = Z ; R'
R' -> Y id = Z ; R' | £
Y -> entero | cadena | real
Z -> num | cad
V -> var B
B -> Y I ; B'
B' -> Y I ; B' | £
I -> id I'
I' -> , id I' | £
S -> A S' | L S' | O S'
S' -> A S' | L S' | O S' | £
    
```

A -> **id = A'**

A' -> **E ; | cad ;**

E -> **T E'**

E' -> **+ T E' | - T E' | £**

T -> **F T'**

T' -> *** F T' | / F T' | £**

F -> **id | num | (E)**

L -> **leer id ;**

O -> **visua W ;**

W -> **id W' | num W' | cad W'**

W' -> **, W | £**

El ejemplo 3 correspondiente al famoso “HOLA MUNDO” simulándolo obtenemos lo que indica la figura #3.2.



Fig. No. 3.2 Reconocimiento de “HOLA MUNDO”, ejo. 3.

Podemos teclear otro ejemplo en la simulación, digamos que el ejo 2 el cual debemos teclear tal y como se indica. La figura #3.3 tiene el resultado como exitoso y la derivación a la izquierda -50 producciones usadas- del conjunto de sentencias.

```

inicio
var
    entero id,id;
    cadena id,id,id;
    real id,id,id,id;

visua cad;
leer id;
visua cad;
leer id;
id = cad;
id = id + num;
fin
    
```

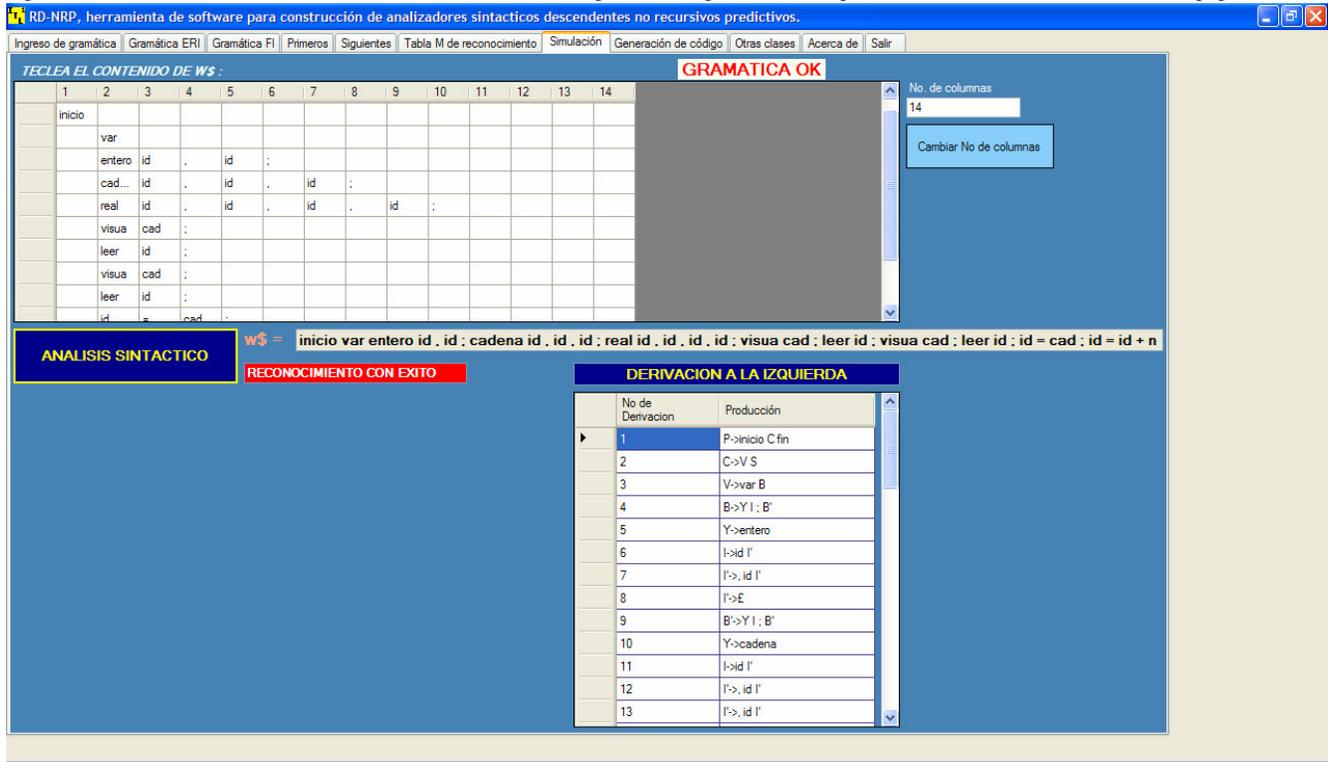


Fig. No. 3.3 Reconocimiento del eje. 2.

4 Analizador léxico.

El conjunto de símbolos terminales que el análisis léxico deberá reconocer, no depende del tipo de reconocedor ya sea descendente o bien ascendente, que usemos en el análisis sintáctico. El conjunto V_t s de la gramática es :

$V_t = \{ \text{inicio fin const id = ; entero cadena real num cad var , + - * / () leer visua } \}$

Agrupemos a los símbolos terminales de manera que a cada grupo le corresponda un AFD en el análisis léxico :

- *AFD id y palabras reservadas.*- inicio fin const id entero cadena real var leer visua. Con este AFD reconoceremos tanto a los identificadores como a las palabras reservadas, usando el método *Esld()* dentro de la clase *Lexico* que produce el software SP-PS1. En el caso de las palabras reservadas almacenaremos la pareja lexema-lexema, para el *id* almacenamos la pareja token-lexema.
- *AFD num.*- Reconoceremos sólo número enteros. Almacenamos la pareja token-lexema.
- *AFD cad.*- Reconoceremos secuencias de caracteres encerradas entre comillas. Incluimos el reconocimiento de la cadena nula. Almacenamos la pareja token-lexema.
- *AFD otros.*- En este AFD reconocemos a los demás símbolos terminales : = ; , + - * / () . Almacenamos la pareja lexema-lexema.
- *AFD delim.*- Reconocemos a los delimitadores pero no los almacenamos, ni al token, ni al lexema.

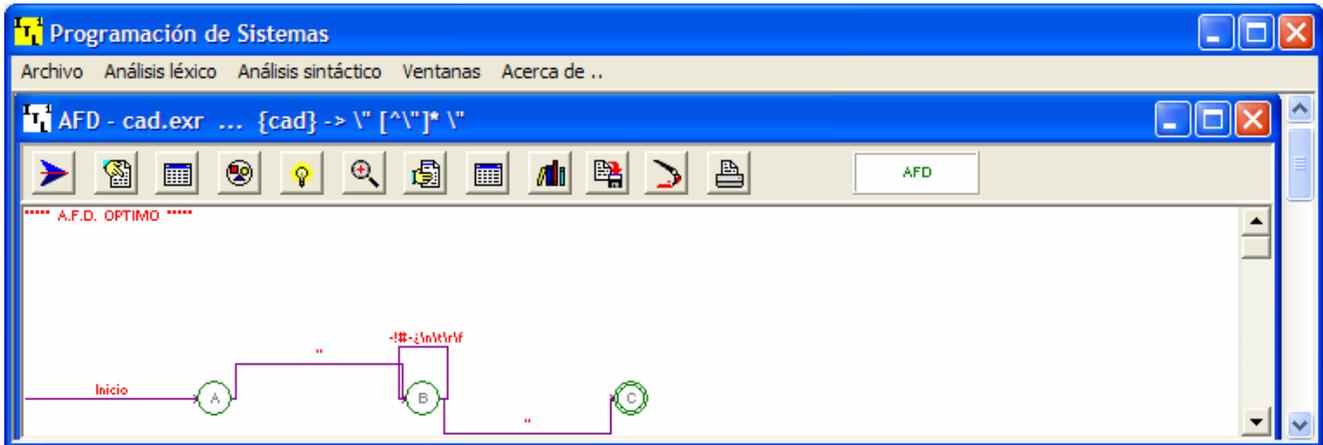
Recomendamos al lector que consulte el documento *leeme-analex-C#* para obtener información del uso del software SP-PS1 para generar código útil para construir un analizador léxico. También se sugiere consultar los documentos *leeme-RD-NRP* y *leeme-recasc-C#* donde se explica la modificación de la clase *Lexico* para cumplir con lo especificado en las viñetas del párrafo anterior. A continuación mostramos la expresión regular para cada token y su AFD.

AFD delim.- Sirve para reconocer a los caracteres delimitadores tales como el blanco, nueva línea, retorno de carro, tab. Su expresión regular y AFD óptimo construído por SP-PS1 son :

{delim} -> [\ \n\r\t]+ [^\ \n\r\t]

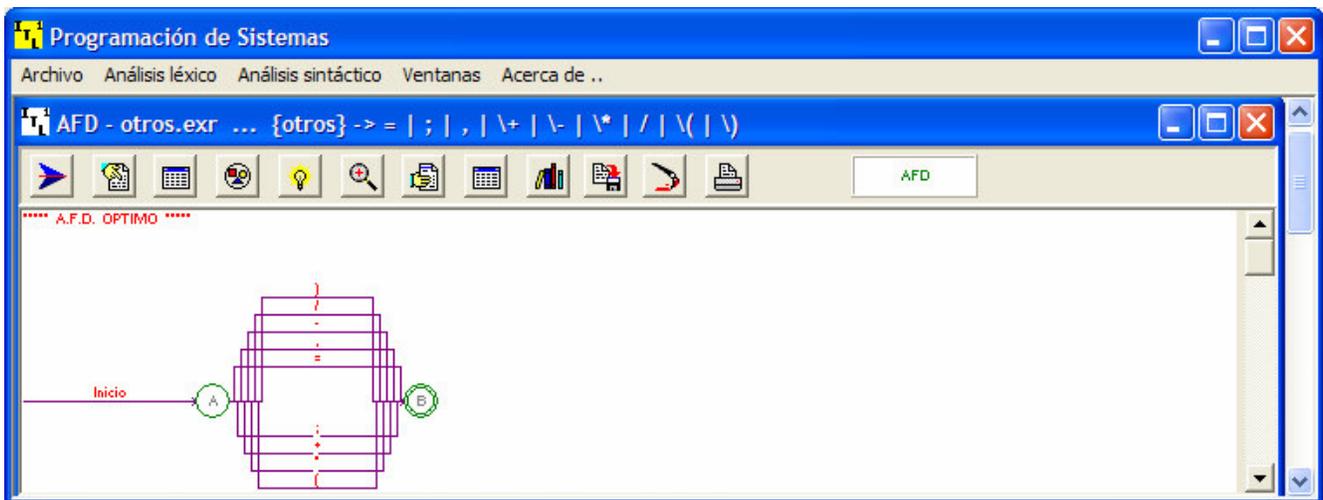
AFD *cad*.- Reconoce a las cadenas de caracteres encerradas entre comillas.

{cad} -> \" [^\"]* \"



AFD *otros*.- Su expresión regular es :

{otros} -> = | ; | | , | \+ | \- | * | / | \ (| \)



El código generado por SP-PS1 para la clase *Lexico* es –se incluyen las modificaciones- :

```
class Lexico
{
    const int TOKREC = 5;
    const int MAXTOKENS = 500;
    string[] _lexemas;
    string[] _tokens;
    string _lexema;
    int _noTokens;
    int _i;
    int _iniToken;
    Automata oAFD;

    public Lexico() // constructor por defecto
    {
        _lexemas = new string[MAXTOKENS];
        _tokens = new string[MAXTOKENS];
        oAFD = new Automata();
        _i = 0;
        _iniToken = 0;
        _noTokens = 0;
    }
}
```

```

}

public void Inicia()
{
    _i = 0;
    _iniToken = 0;
    _noTokens = 0;
}

public void Analiza(string texto)
{
    bool recAuto;
    int noAuto;
    while (_i < texto.Length)
    {
        recAuto=false;
        noAuto=0;
        for(;noAuto<TOKREC&&!recAuto;)
            if(oAFD.Reconoce(texto,_iniToken,ref _i,noAuto))
                recAuto=true;
            else
                noAuto++;
        if (recAuto)
        {
            _lexema = texto.Substring(_iniToken, _i - _iniToken);
            switch (noAuto)
            {
                //----- Automata delim-----
                case 0 : // _tokens[_noTokens] = "delim";
                    break;
                //----- Automata id-----
                case 1 : if (EsId())
                    _tokens[_noTokens] = "id";
                    else
                        _tokens[_noTokens] = _lexema;
                    break;
                //----- Automata num-----
                case 2 : _tokens[_noTokens] = "num";
                    break;
                //----- Automata otros-----
                case 3 : _tokens[_noTokens] = _lexema;
                    break;
                //----- Automata cad-----
                case 4 : _tokens[_noTokens] = "cad";
                    break;
            }
            if (noAuto != 0)
                _lexemas[_noTokens++] = _lexema;
        }
        else
            _i++;
        _iniToken = _i;
    }
}

private bool EsId()
{
    string[] palres ={ "inicio", "fin", "const", "var", "entero", "real", "cadena", "leer", "visua"};
    for (int i = 0; i < palres.Length; i++)
        if (_lexema==palres[i])
            return false;
    return true;
}

} // fin de la clase Lexico

```

La clase Automata es :

```

class Automata
{
    string _textoIma;
    int _edoAct;
}

```

Tabla de símbolos en C# y su interacción con el analizador léxico.

```
char SigCar(ref int i)
{
    if (i == _textoIma.Length)
    {
        i++;
        return '□';
    }
    else
        return _textoIma[i++];
}

public bool Reconoce(string texto,int iniToken,ref int i,int noAuto)
{
    char c;
    _textoIma = texto;
    string lenguaje;
    switch (noAuto)
    {
        //----- Automata delim-----
        case 0 : _edoAct = 0;
                break;
        //----- Automata id-----
        case 1 : _edoAct = 3;
                break;
        //----- Automata num-----
        case 2 : _edoAct = 6;
                break;
        //----- Automata otros-----
        case 3 : _edoAct = 9;
                break;
        //----- Automata cad-----
        case 4 : _edoAct = 11;
                break;
    }
    while(i<=_textoIma.Length)
        switch (_edoAct)
        {
            //----- Automata delim-----
            case 0 : c=SigCar(ref i);
                    if ((lenguaje=" \n\r\t").IndexOf(c)>=0) _edoAct=1; else
                    { i=iniToken;
                      return false; }
                    break;
            case 1 : c=SigCar(ref i);
                    if ((lenguaje=" \n\r\t").IndexOf(c)>=0) _edoAct=1; else
                    if ((lenguaje="!\\"#$%&'()*+,-
./0123456789;<=>?@ABCDEFGHIJKLMNopqrstuvwxyz[\]^_`abcdefghijklmnopqrstuvwxyz{|}~□€□, f„...t†^%Š<€□Ž□□''
""•---™š>œ□žŸ ;çŁπ¥|$"©ª«¬¬®¯°±²³´µ¶·,¹º»¼½¾¿\n\t\r\f").IndexOf(c)>=0) _edoAct=2; else
                    { i=iniToken;
                      return false; }
                    break;
            case 2 : i--;
                    return true;
                    break;
            //----- Automata id-----
            case 3 : c=SigCar(ref i);
                    if
                    ((lenguaje="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz").IndexOf(c)>=0) _edoAct=4; else
                    { i=iniToken;
                      return false; }
                    break;
            case 4 : c=SigCar(ref i);
                    if
                    ((lenguaje="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz").IndexOf(c)>=0) _edoAct=4; else
                    if ((lenguaje="0123456789").IndexOf(c)>=0) _edoAct=4; else
                    if ((lenguaje="_").IndexOf(c)>=0) _edoAct=4; else
                    if ((lenguaje="!\\"#$%&'()*+,-./:;<=>?@[\\]^`{|}~□€□, f„...t†^%Š<€□Ž□□''""•---
™š>œ□žŸ ;çŁπ¥|$"©ª«¬¬®¯°±²³´µ¶·,¹º»¼½¾¿\n\t\r\f").IndexOf(c)>=0) _edoAct=5; else
                    { i=iniToken;
                      return false; }
                    break;
            case 5 : i--;
                    return true;
                    break;
        }
}
```

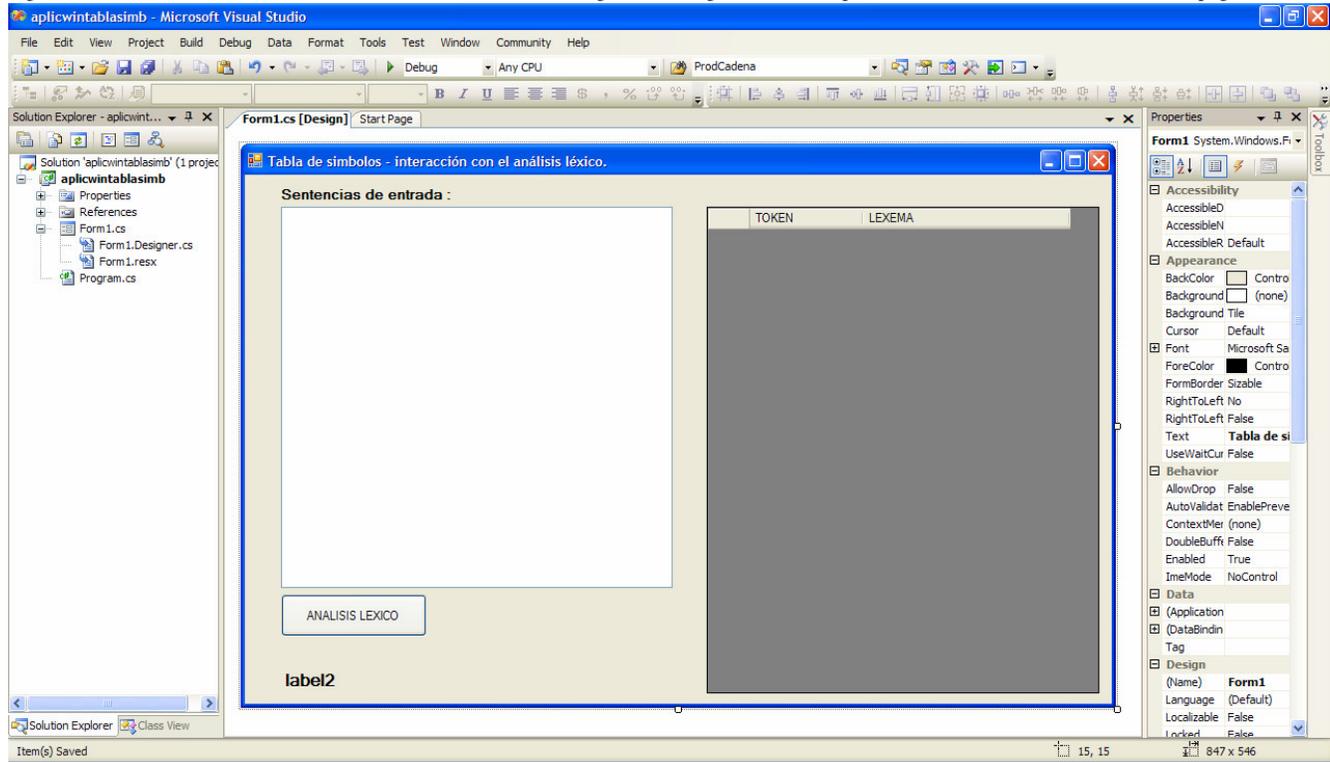



Fig. No. 4.1 Aplicación Windows C#, para el análisis léxico.

Agregamos la definición del objeto **oAnaLex** en *Form1.cs* :

```
Lexico oAnaLex = new Lexico();
```

Ahora añadimos las clases *Lexico* y *Automata* al proyecto y les pegamos el código que hemos mostrado antes en esta misma sección.

Agregamos en el click del botón ANALISIS LEXICO el código siguiente :

```
public partial class Form1 : Form
{
    Lexico oAnaLex = new Lexico();
    public Form1()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        oAnaLex.Inicia();
        oAnaLex.Analiza(textBox1.Text);
        dataGridView1.Rows.Clear();
        if (oAnaLex.NoTokens > 0)
            dataGridView1.Rows.Add(oAnaLex.NoTokens);
        for (int i = 0; i < oAnaLex.NoTokens; i++)
        {
            dataGridView1.Rows[i].Cells[0].Value = oAnaLex.Token[i];
            dataGridView1.Rows[i].Cells[1].Value = oAnaLex.Lexema[i];
        }
    }
}
```

Antes de ejecutar la aplicación, debemos agregar a la clase *Lexico* la propiedades siguientes :

```
public int NoTokens
{
    get { return _noTokens; }
}
```

```

Ing. Francisco Ríos Acosta
public string[] Lexema
{
    get { return _lexemas; }
}
public string[] Token
{
    get { return _tokens; }
}
    
```

La figura #4.2 muestra la ejecución de la aplicación. Notemos que hemos desplazado la barra de la rejilla de visualización de las parejas token-lexema, con el fin de mostrar las últimas parejas reconocidas. Entre ellas están los tokens **cad**.

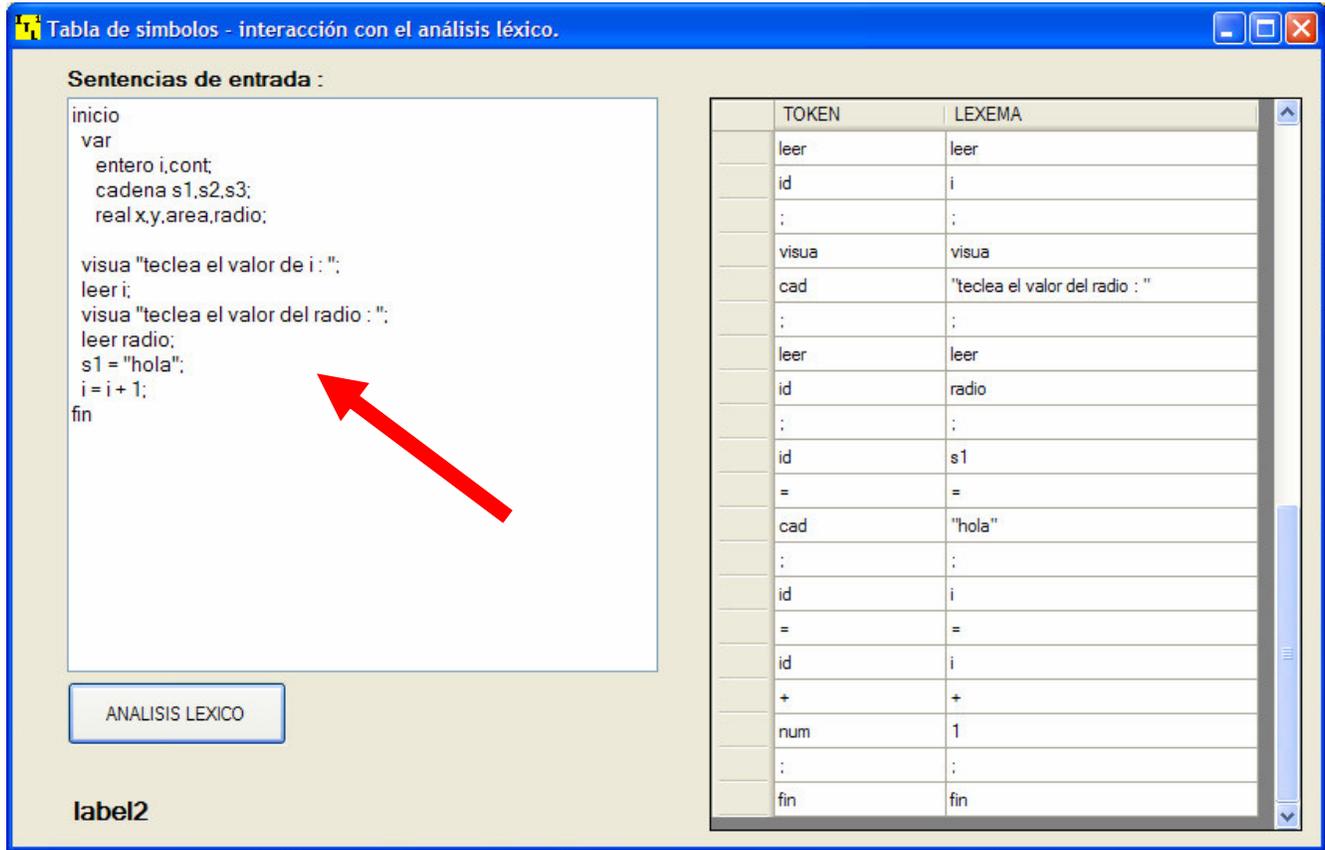


Fig. No. 4.2 Aplicación Windows C# con la entrada del eje. 2.

5 Análisis de los atributos de los objetos que se almacenan en la tabla de símbolos.

Vamos a limitar nuestro universo a los tokens que reconoce el analizador léxico de nuestro ejemplo y que necesiten de instalarse en la tabla de símbolos al momento del proceso del análisis léxico :

- *id*
- *num*
- *cad*

De los anteriores tomaremos para nuestro estudio al token *id*. Los tokens *num* y *cad* los dejaremos para verlos posteriormente. El token *id* puede representar o una variable o una constante. Los atributos que proponemos de inicio son :

- **clase**, sus valores posibles : *variable* , *constante*.
- **nombre**, es el lexema del token *id* (tal y como se encuentra en el texto de entrada).
- **valor**, representa el valor que se le asigna durante los diferentes procesos de análisis a la variable o a la constante.
- **tipo**, representa el tipo de dato al que pertenece la variable o la constante.
- **noBytes**, número de bytes en los que se almacena la variable o constante.
- **posicion**, se refiere al índice de inicio del lexema donde se reconoció al token *id* dentro del texto de entrada.

El analizador léxico en lo que respecta al token *id* al momento de reconocerlo, sólo puede instalarlo en la tabla de símbolos y almacenar los atributos *nombre* y *posicion*. Los atributos *clase*, *valor*, *tipo* y *noBytes* son asignados en la etapa del análisis semántico. Por lo anterior, la clase *Lexico* deberá tener un método que instale a un *id* en la tabla de símbolos, asignando sólo los atributos *nombre* y *posicion*.

Entonces, debemos pensar en que los nodos de las listas de la tabla de símbolos contienen objetos de una clase, cuyos atributos sean los mencionados. A esta clase la llamaremos *TipoElem*.

```
class TipoElem
{
    private string _clase;
    private string _nombre;
    private string _valor;
    private string _tipo;
    private int _noBytes;
    private int _posicion;
}
```

Quizá la única confusión que pudiera existir al analizar esta definición de atributos, es el tipo *string* que le hemos dado al atributo *valor*. En cuanto a esta cuestión, pensemos en realizar una conversión de acuerdo al atributo *tipo* que el análisis semántico registre.

Con los atributos iniciales ya definidos quedando especificado que sólo empezaremos por instalar al token *id* y sus atributos *nombre* y *posicion* durante el análisis léxico, estamos listos para definir la clase *TablaSimbolos*.

6 Clase *TablaSimbolos*.

De acuerdo al modelo conceptual presentado en la sección 2, la tabla de símbolos es un objeto que contiene un arreglo de listas enlazadas. Iniciemos por definir este atributo con elementos de la clase *Lista*.

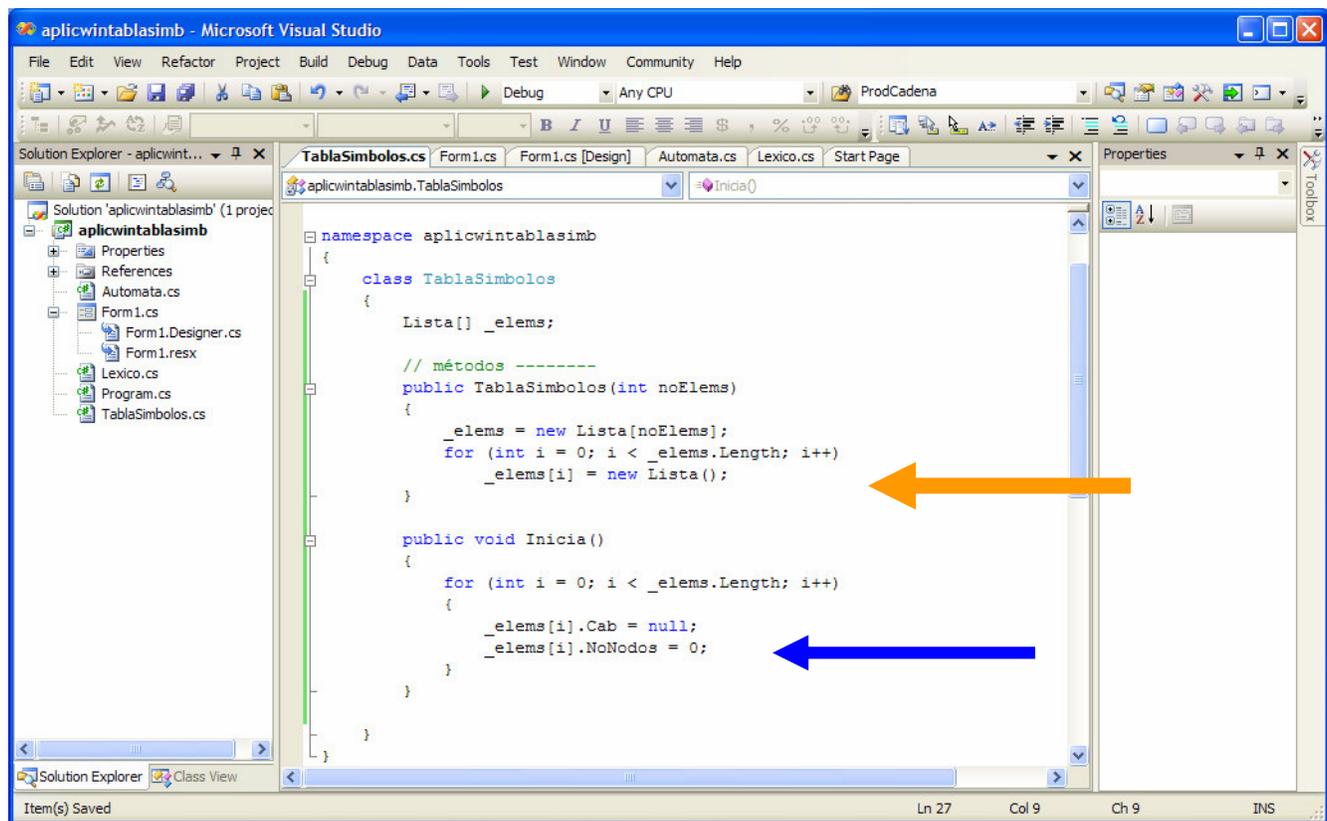


Fig. No. 6.1 Aplicación Windows C# con la clase *TablaSimbolos*, su constructor y el método *Inicia()*.

```
class TablaSimbolos
{
    Lista[] _elems;
}
```



Agreguemos esta clase a la aplicación Windows C# que ya contiene al analizador léxico para el lenguaje propuesto. Una vez añadida la clase *TablaSimbolos* al proyecto, escribamos la sentencia de definición del atributo *_elems*. También agregamos el constructor para la clase *TablaSimbolos*, además del método *Inicia()* utilizado tantas veces como sea necesario efectuar la inicialización del objeto tabla de símbolos al que mas tarde llamaremos **oTablaSimb**. La figura #6.1 contiene la aplicación Windows con la clase añadida y el código para los métodos que hemos mencionado.

El constructor de la clase *TablaSimbolos* permite fijar la dimensión del arreglo de listas por medio del parámetro de entrada *noElem*s. Además crea a los objetos *Lista* del arreglo mediante la llamada al operador **new**.

A diferencia del constructor parametrizado, el método *Inicia()* no crea a los objetos *Lista* del arreglo –ya que previamente han sido creados por el constructor-, sino que asigna las cabeceras de los objetos *Lista* a **null** y su número de nodos los asigna a 0, cumpliendo de esta manera su función.

Como ya estamos manejando a los objetos *Lista*, debemos agregar la definición de esta clase al proyecto. La definición de la clase *Lista* es :

```
class Lista
{
    Nodo _cabLista;
    int _noNodos;

    public Lista()
    {
        _cabLista = null;
        _noNodos = 0;
    }

    public Nodo Cab
    {
        get { return _cabLista; }
        set { _cabLista = value;}
    }

    public void InsInicio(TipoElem oElem)
    {
        Nodo nuevoNodo = new Nodo();
        nuevoNodo.Info = oElem;
        nuevoNodo.Sig = _cabLista;
        _cabLista = nuevoNodo;
        _noNodos++;
    }

    public int NoNodos
    {
        get { return _noNodos;}
        set { _noNodos = value;}
    }
}
```

La clase *Lista* representa a objetos con nodos simplemente enlazados, con campos de información y de enlace –referencias-. En cualquier curso de estructura de datos es manejada esta clase. Realmente los métodos que se han incluido son los menos.

Una buena mejora es agregar el método *InsOrden(oElem)* que agrega en orden ascendente al objeto **oElem**, en la lista enlazada.

Notemos que la clase *Lista* hace referencia a otras 2 clases : *Nodo* y *TipoElem*. La figura #6.2 muestra la aplicación Windows C# con la clase *Lista* añadida.

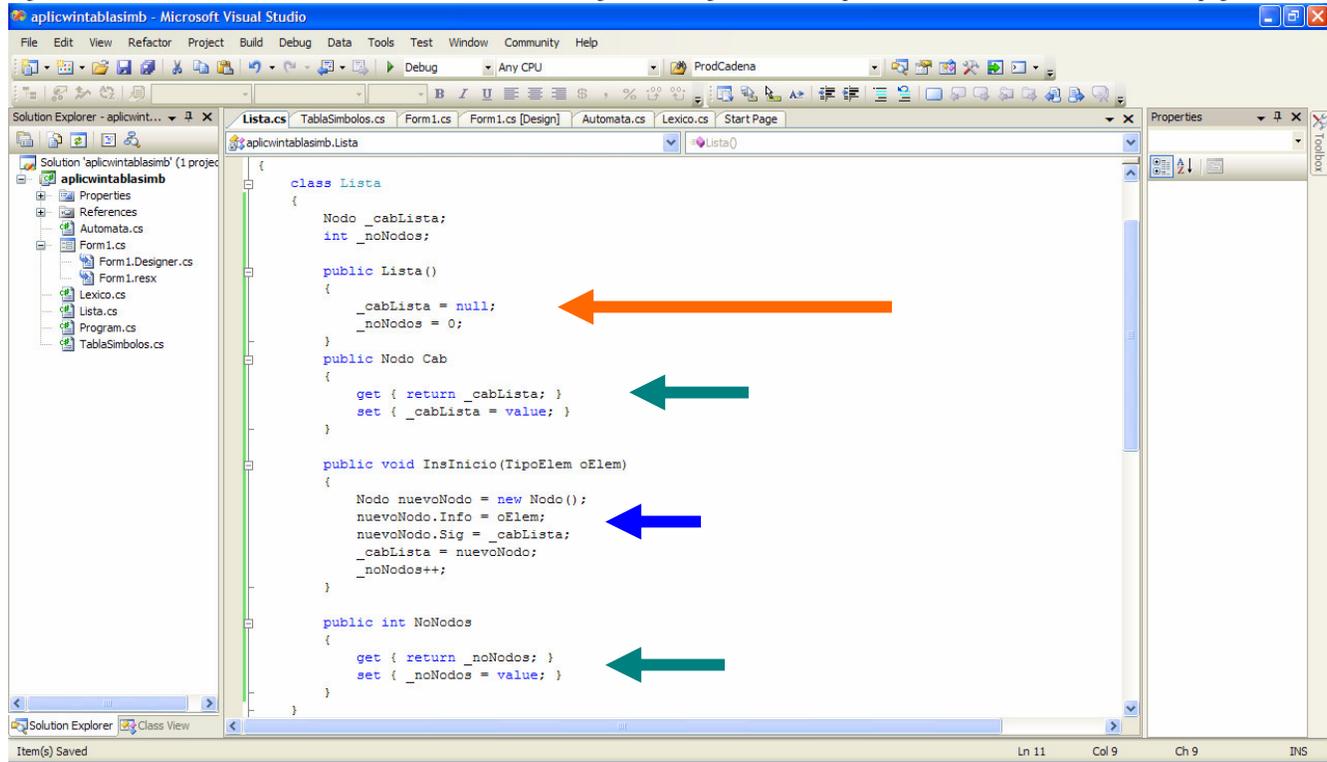


Fig. No. 6.2 Aplicación Windows C# con la clase *Lista* incluida.

Observemos que la clase *Lista* que hemos incluido al proyecto, contiene un **constructor por defecto**, 2 propiedades *Cab* y *NoNodos*, además del método *InsInicio()*.

La clase *Nodo* es la encargada de representar a los elementos de la lista enlazada. La añadimos al proyecto según el código que mostramos enseguida :

```

class Nodo
{
    TipoElem _oInfo;
    Nodo _sig;

    public TipoElem Info
    {
        get { return _oInfo; }
        set { _oInfo = value; }
    }

    public Nodo Sig
    {
        get { return _sig; }
        set { _sig = value; }
    }
}
    
```

La clase *Nodo* además de sus atributos sólo contiene 2 propiedades : *Info* y *Sig*. Lo que sigue es añadir al proyecto la clase *TipoElem* con los atributos que definimos anteriormente en esta misma sección.

Por ahora sólo declaramos los atributos en la clase *TipoElem*, dejando la inserción de los métodos para después. La figura muestra a la aplicación Windows después de haber agregado las 3 clases : *Lista*, *Nodo* y *TipoElem*.

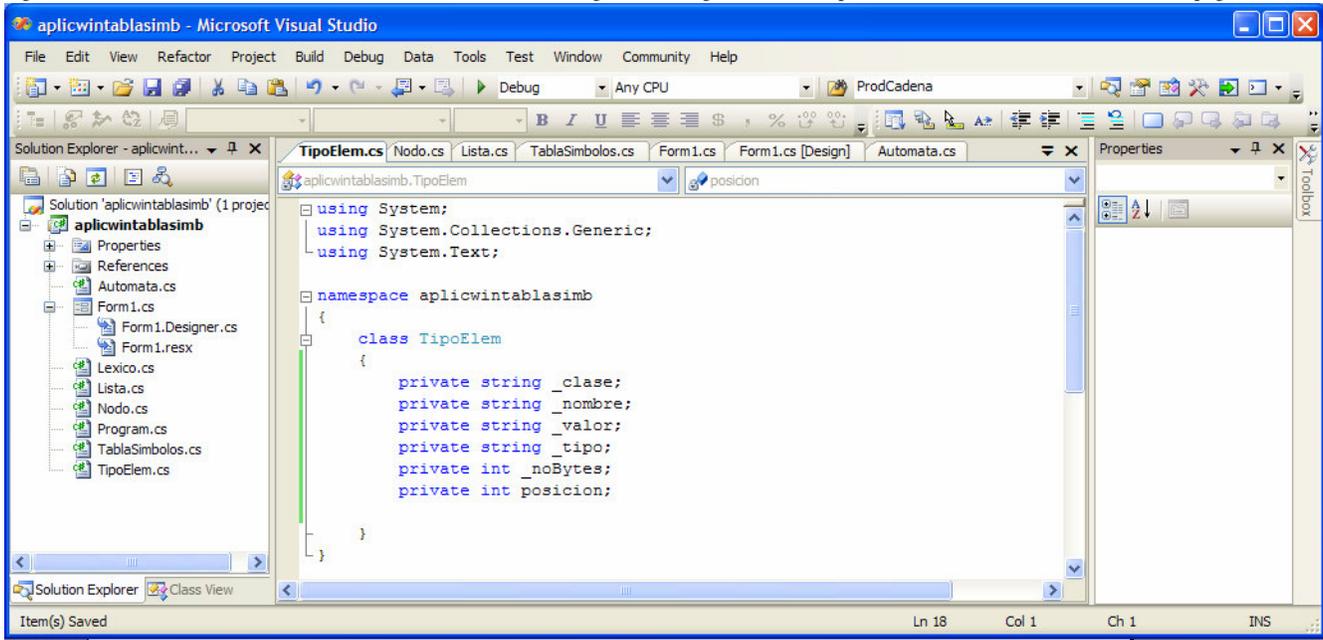


Fig. No. 6.3 Aplicación Windows C# con las clases *Nodo* y *TipoElem*.

Sólo falta agregar al objeto **oTablaSimb** dentro del archivo *Form1.cs*, dimensionando al arreglo de listas en 26. La razón de esta dimensión, es que la función de desmenuzamiento `-hash-` toma al primer caracter del lexema del *id* de manera que si es una A(a) el *id* se añade a la lista con índice 0, si empieza con B(b) se agrega a la lista con índice 1, y así hasta la letra Z(z) en donde se agrega al *id* a la lista con índice 25.

Agrega la definición del objeto **oTablaSimb** en la siguiente línea a la definición del objeto **oAnaLex**, según se muestra en el segmento de código de la clase *Form1* del archivo *Form1.cs*.

```
public partial class Form1 : Form
{
    Lexico oAnaLex = new Lexico();
    TablaSimbolos oTablaSimb = new TablaSimbolos(26);
    public Form1()
    {
        InitializeComponent();
    }
    ...
    ...
    ...
```

Compilamos la aplicación sólo para observar que no existan errores en el código que hemos añadido. El resultado de la compilación es exitoso, sin embargo tenemos unas advertencias que no les vamos a poner atención. Tecleemos el código siguiente en la ventana de entrada de texto, de manera que veamos las parejas token-lexema que el analizador léxico encuentra.

```
inicio
    visua "HOLA MUNDO";
fin
```

Hasta este punto, tenemos la definición del objeto **oTablaSimb** en nuestra aplicación, sin errores. Lo que sigue es instalar a los *id* reconocidos durante la etapa del análisis léxico, en la tabla de símbolos.

7 Instalación de identificadores.

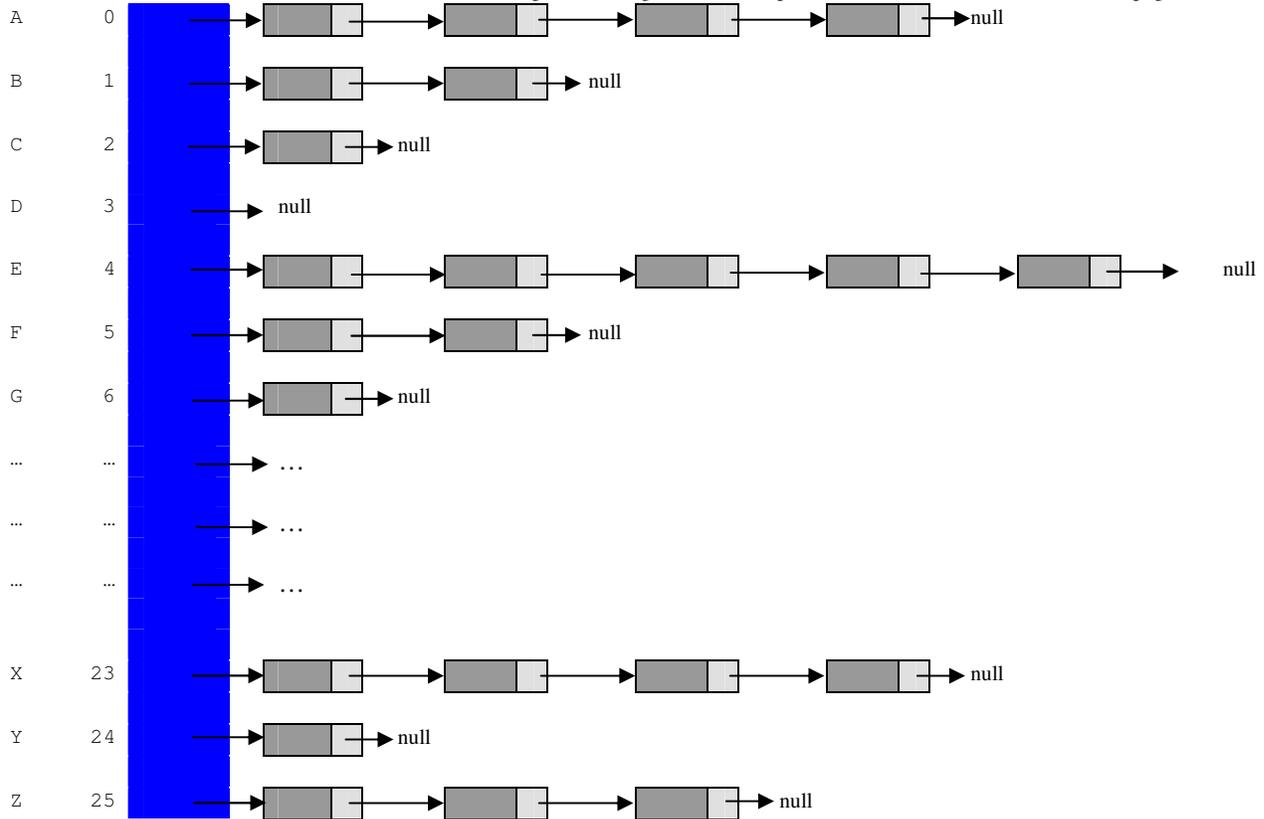
En la sección anterior la tabla de símbolos se dimensionó a 26 listas enlazadas cada una correspondiente a una letra del abecedario.

Tabla de símbolos en C# y su interacción con el analizador léxico.

Ing. Francisco Ríos Acosta

Instituto Tecnológico de la Laguna, a 15 de septiembre del 2008.

pag. 20 de 26



La pregunta es : ¿en qué parte del código vamos a insertar la instalación de un *id* durante el análisis léxico?. Bueno, vayamonos por pasos, así que lo primero que vamos a hacer es enumerar los pasos a seguir :

- Modificar la llamada al método *Analiza()* de la clase *Lexico*.
- Modificar la definición del método *Analiza()* en sus parámetros que recibe.
- Insertar el mensaje *oTablaSimb.Instalar()* dentro del método *Analiza()*.

Llamada al método Analiza().- Inicialmente, el método tiene sólo un parámetro : el componente *TextBox* que contiene a la cadena que se analiza léxicamente. Ahora, tendrá 2 parámetros el que ya conocemos y el objeto **oTablaSimb** que enviaremos como referencia, de manera que pueda modificarse a este objeto, es decir, a la tabla de símbolos. Entonces modificamos la llamada al método *Analiza()* según se muestra a continuación :

```
private void button1_Click(object sender, EventArgs e)
{
    oAnaLex.Inicia();
    oTablaSimb.Inicia();
    oAnaLex.Analiza(textBox1.Text, oTablaSimb);
    dataGridView1.Rows.Clear();
    if (oAnaLex.NoTokens > 0)
        dataGridView1.Rows.Add(oAnaLex.NoTokens);
    for (int i = 0; i < oAnaLex.NoTokens; i++)
    {
        dataGridView1.Rows[i].Cells[0].Value = oAnaLex.Token[i];
        dataGridView1.Rows[i].Cells[1].Value = oAnaLex.Lexema[i];
    }
}
```

Definición del método Analiza() en sus parámetros.- Desde luego que si compilamos existirá un error, así que debemos también modificar la definición del método *Analiza()* en la clase *Lexico*, según se te indica en el segmento de código siguiente :

```
public void Analiza(string texto, TablaSimbolos oTablaSimb)
{
```



Compilemos y ejecutemos sólo para observar que no existan errores al hacer nuestras modificaciones a la aplicación Windows C# que estamos construyendo.

Instalación del id dentro del método Analiza().- Es trivial que la instalación del id dentro de la tabla de símbolos deberá efectuarse al momento en que un token *id* sea reconocido, dentro del código del método *Analiza()*. Házlo según se muestra en la figura #7.1.

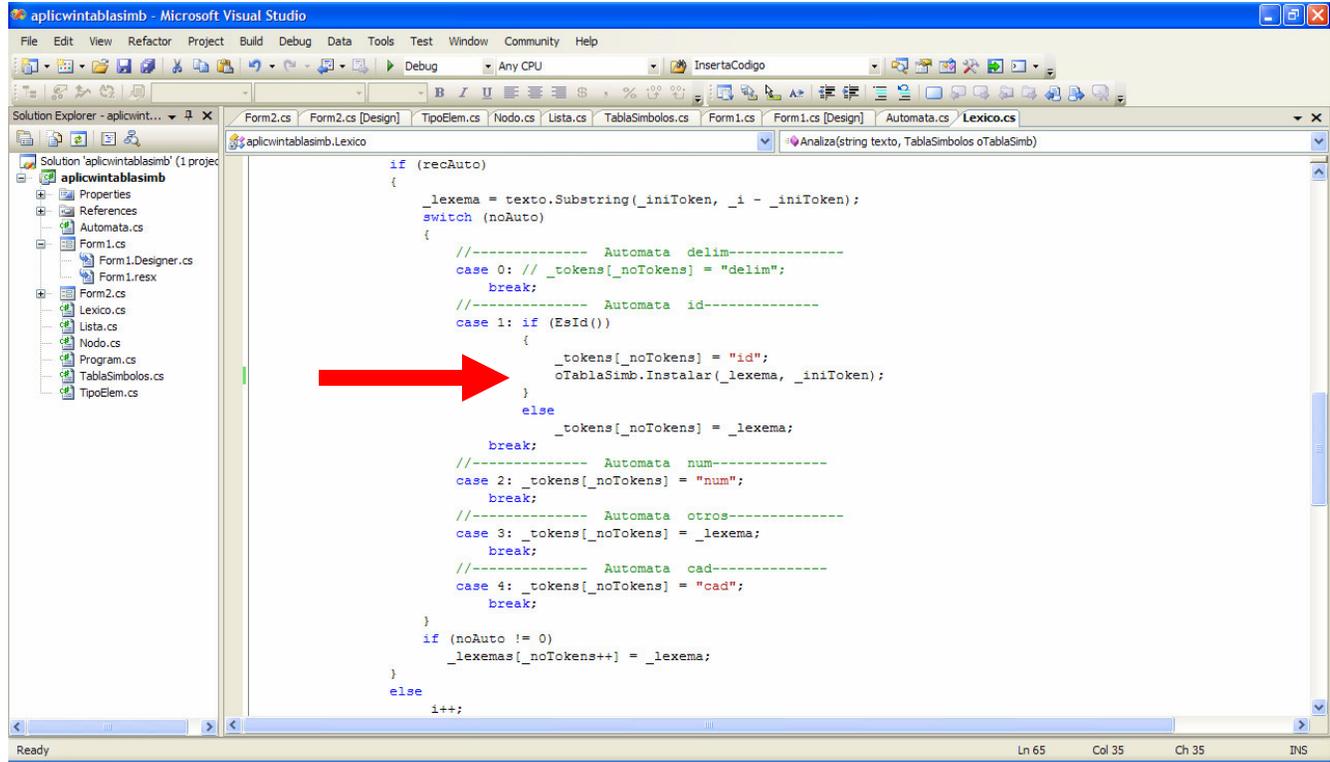


Fig. No. 7.1 Mensaje `oTablaSimb.Instalar()` en la definición del método `Analiza()` de la clase `Lexico`.

Veamos los parámetros que recibe el método `Instalar()` :

```
oTablaSimb.Instalar(_lexema, _iniToken);
```

- El parámetro `_lexema` es el dato con el que se va a instanciar al atributo `_nombre`.
- El parámetro `_iniToken` es el valor con el que se va a instanciar al atributo `_posicion`.

El método `Instalar()` lo definimos en la clase `TablaSimbolos` y consiste del código siguiente :

```
public void Instalar(string nombre, int posicion)
{
    char car = nombre.ToUpper()[0];
    int indice = Convert.ToInt32(car) - 65;
    if (! EncuentraToken(indice,nombre))
    {
        TipoElem oElem=new TipoElem(nombre, posicion);
        _elems[indice].InsInicio(oElem);
    }
}
```

La función de hash está compuesta de 2 líneas de código : la primera toma al primer caracter del identificador y lo convierte a mayúscula, la segunda línea de código obtiene el índice que le corresponde a la letra en el arreglo de listas enlazadas de la tabla de símbolos.

```
char car = nombre.ToUpper()[0];
int indice = Convert.ToInt32(car) - 65;
```

El identificador en instalado sólo si no se encuentra en la lista enlazada correspondiente al índice que ha calculado la función de desmenuzamiento `-hash-`. El método `EncuentraToken()` es el encargado de realizar la búsqueda, si lo encuentra retorna **true**, **false** de lo contrario. Necesitamos agregar la definición de este método en la clase `TablaSimbolos` según lo mostramos enseguida :

```
public bool EncuentraToken(int indice, string nombre)
{
    Nodo refLista = _elems[indice].Cab;
    while (refLista != null)
    {
        if (refLista.Info.Nombre==nombre)
            return true;
        refLista = refLista.Sig;
    }
    return false;
}
```

Notemos que el método retorna **true** cuando el atributo `_nombre` devuelto por la propiedad `Nombre`, es igual al parámetro `nombre` recibido por el método `EncuentraToken()`.

Si no se encuentra el identificador, entonces insertamos el nuevo elemento en la lista cuyo índice ha sido calculado por la función de desmenuzamiento, mediante el uso del método `InsInicio()`, definido en la clase `Lista`.

Antes de compilar la aplicación debemos añadir la propiedad `Nombre` en la clase `TipoElem` de acuerdo al código que mostramos a continuación :

```
public string Nombre
{
    get { return _nombre; }
    set { _nombre = value; }
}
```

Observemos que existe un constructor que debemos definir en la clase `TipoElem`. Este constructor es utilizado en el método `Instalar()` de la clase `TablaSimbolos`, como parámetro al crear el objeto que se va a instalar.

```
TipoElem oElem = new TipoElem(nombre, posicion);
```

El código del constructor es el siguiente, agreguemoslo a la clase `TipoElem` :

```
public TipoElem(string nombre, int posicion)
{
    _clase = "";
    _nombre = nombre;
    _posicion = posicion;
}
```

Compilemos el programa sólo para probar que no existan errores en el código que hemos añadido. La aplicación Windows debe ejecutarse sin problemas `-sólo advertencias-`.

Hasta aquí, ya tenemos instalados los tokens `id` que hayan sido reconocidos en el análisis léxico, en la tabla de símbolos representada por el objeto `oTablaSimb`. Sólo resta visualizar la tabla de símbolos, cuestión que haremos en la siguiente sección.

8 Visualización de la tabla de símbolos.

Para visualizar la tabla de símbolos utilizaremos una nueva forma `Form2.cs`, que añadimos usando la opción del menú `Project | Add Windows Form`. La figura #8.1 muestra a nuestro proyecto con la forma `Form2.cs` incluida. La propiedad `Text` de la forma `Form2.cs` se ha cambiado a : `TABLA DE SIMBOLOS`.

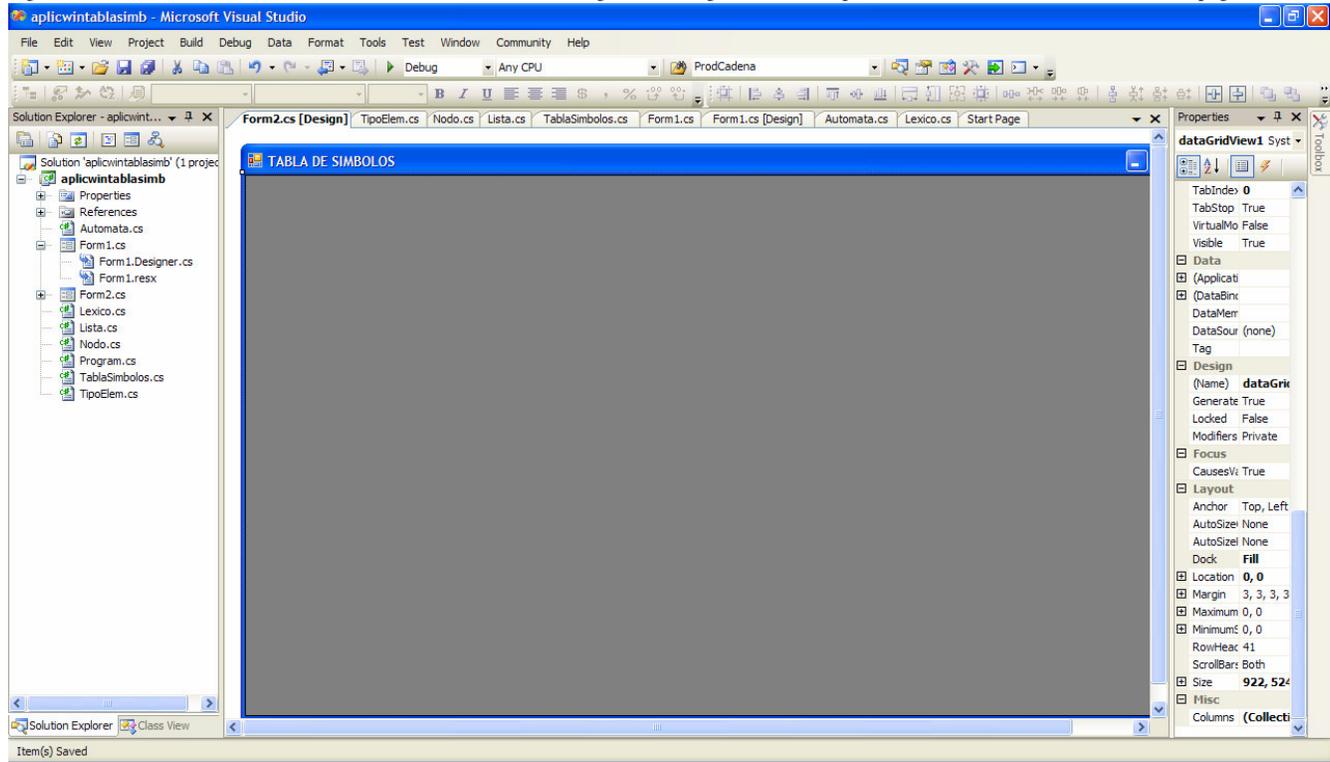


Fig. No. 8.1 Adición de la nueva *Form2.cs* al proyecto de la aplicación.

Notemos que también hemos agregado un componente *DataGridView* a la forma *Form2.cs*. Este componente se encargará de recibir y visualizar en él, los elementos de la tabla de símbolos que se encuentran en el objeto **oTablaSimb**. El componente *DataGridView* es el que aparece con color gris oscuro. La propiedad *Name* de este componente es el que por defecto recibe *dataGridView1*.

Volvamos a la forma *Form1.cs* para agregar un botón que precisamente visualice a la forma *Form2.cs*, de manera que después llenaremos el *dataGridView1* de la *Forma2.cs* con los elementos en el objeto **oTablaSimb**. Asignemos a la propiedad *Text* del nuevo botón *button2*, el valor TABLA DE SIMBOLOS. La figura #8.2 muestra la interfase con el botón *button2* insertado en la forma *Form1.cs*.

Agreguemos en el evento *Click* del *button2* el código que permite visualizar a la forma *Form2.cs* :

```
private void button2_Click(object sender, EventArgs e)
{
    Form2 formaTablaSimb = new Form2();
    formaTablaSimb.Show();
}
```

Si compilamos y ejecutamos nuestra aplicación, veremos que al hacer click en el *button2* tenemos a nuestra vista la forma *Form2.cs*. Lo que resta es caracterizar al *dataGridView1* de la forma *Form2.cs* de manera que acepte a los elementos en el objeto **oTablaSimb** definido en la forma *Form1.cs*.

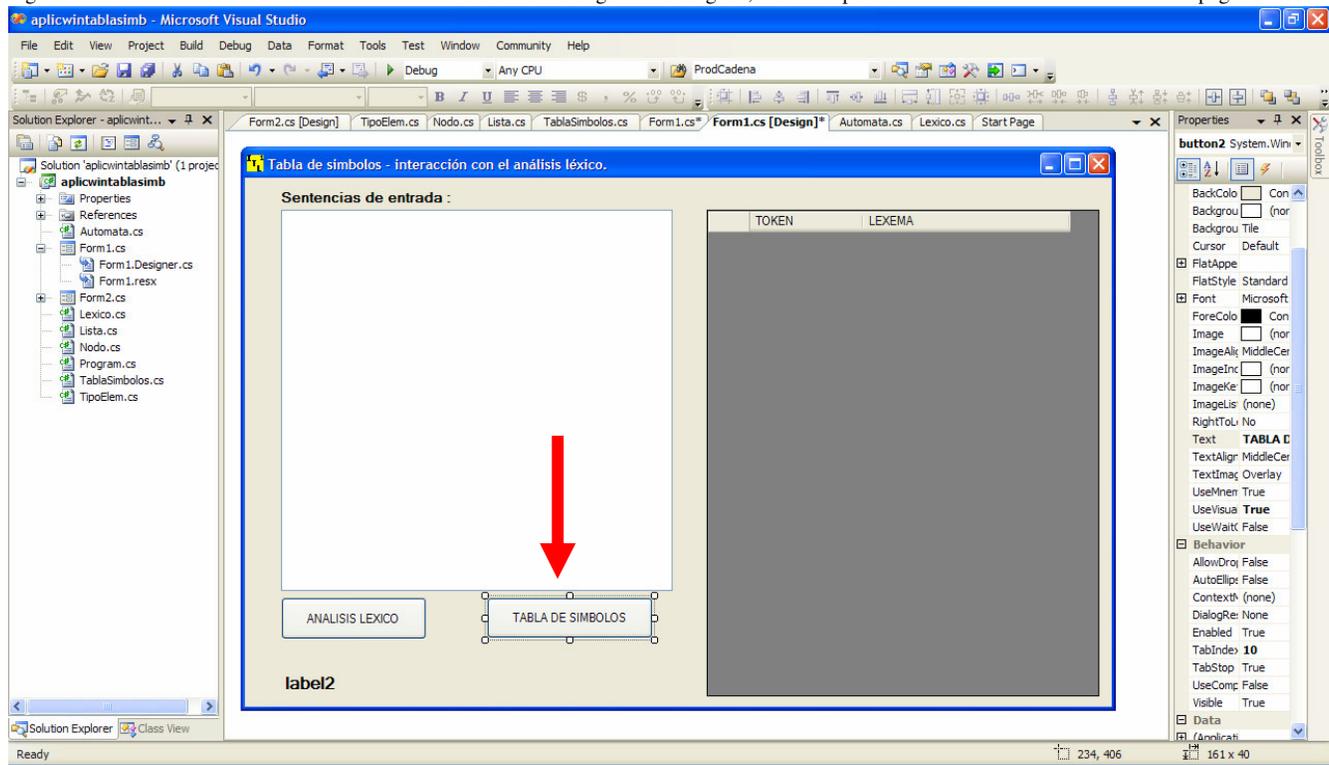


Fig. No. 8.2 Adición del *button2* para visualizar la forma *Form2.cs* –tabla de símbolos–.

Agreguemos ahora a la aplicación el mensaje al objeto **oTablaSimb** para que visualice a sus elementos en el componente *dataGridView1* de *Form2.cs*. Este mensaje debemos añadirlo en el código para el click del *button2*.

```
private void button2_Click(object sender, EventArgs e)
{
    Form2 formaTablaSimb = new Form2();
    oTablaSimb.Visua(formaTablaSimb.dataGridView1);
    formaTablaSimb.Show();
}
```



También debemos agregar la definición del método *Visua()* a la clase *TablaSimbolos*. El código es :

```
public void Visua(System.Windows.Forms.DataGridView dGV)
{
    Nodo refNodo;
    int col = 1;
    dGV.ColumnCount = this.Mayor() + 1;
    dGV.Rows.Add(_elems.Length);
    for (int i = 0; i < _elems.Length; i++)
    {
        col = 1;
        refNodo = _elems[i].Cab;
        dGV.Rows[i].Cells[0].Value = Convert.ToChar(65 + i).ToString() + " - " + i.ToString();
        while (refNodo != null)
        {
            dGV.Rows[i].Cells[col++].Value = refNodo.Info.Clase + " - " + refNodo.Info.Nombre +
                " - " + refNodo.Info.Posicion.ToString();
            refNodo = refNodo.Sig;
        }
    }
}
```



Definimos un nuevo método *Mayor()* para calcular el máximo número de elementos en todas las listas enlazadas. De esta forma, podemos dimensionar al componente *dataGridView1* para el manejo de las columnas del componente.

Así que debemos agregarlo a la clase *TablaSimbolos*.

```
public int Mayor()
{
    int mayor = 0;
    for (int i = 0; i < _elems.Length; i++)
        if (_elems[i].NoNodos > mayor)
            mayor = _elems[i].NoNodos;
    return mayor;
}
```

Se nos olvidaba que debemos agregar la propiedad *Posicion* a la clase *TipoElem*, que es usada en el método *Visua()* de la clase *TablaSimbolos*.

```
public int Posicion
{
    get { return _posicion; }
    set { _posicion = value; }
}
```

Si compilamos la aplicación observaremos que tenemos un error referente al acceso del *dataGridView1* de la forma *Form2.cs*. Este error es corregido fácilmente definiendo una propiedad en la forma *Form2.cs*, según lo indicamos en la figura #8.3.

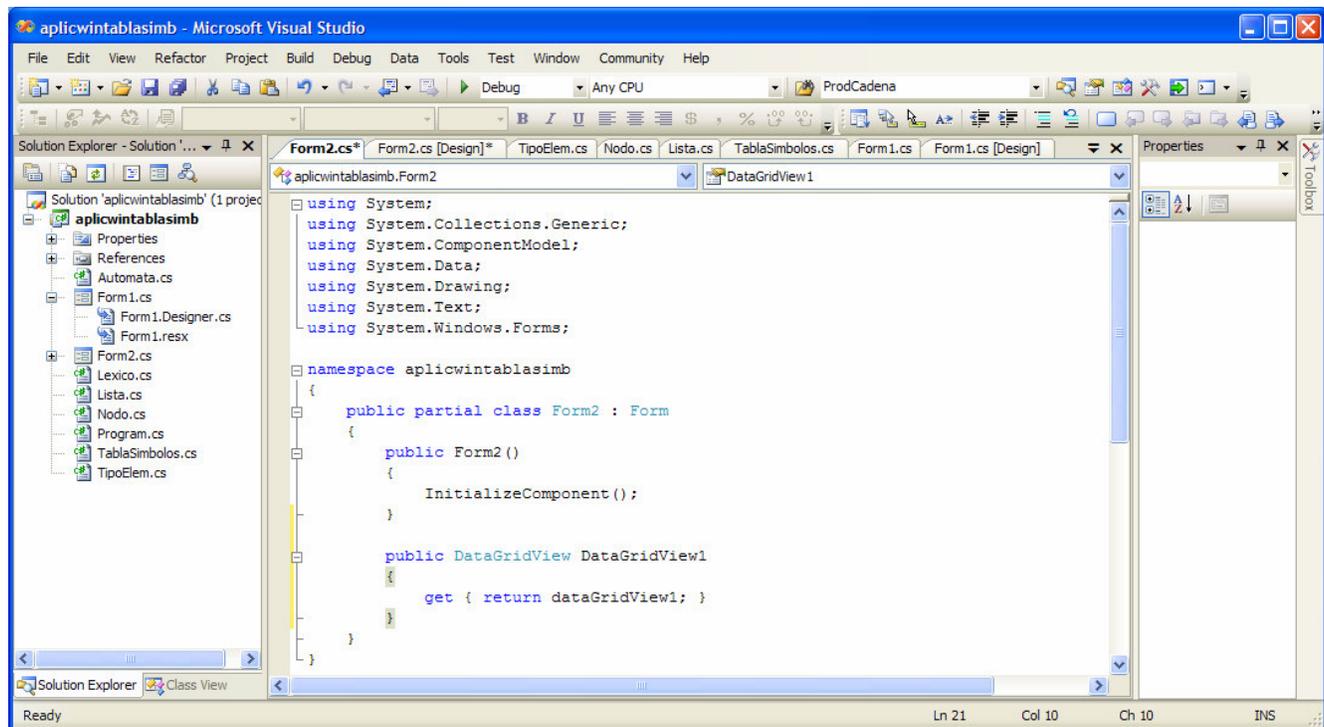
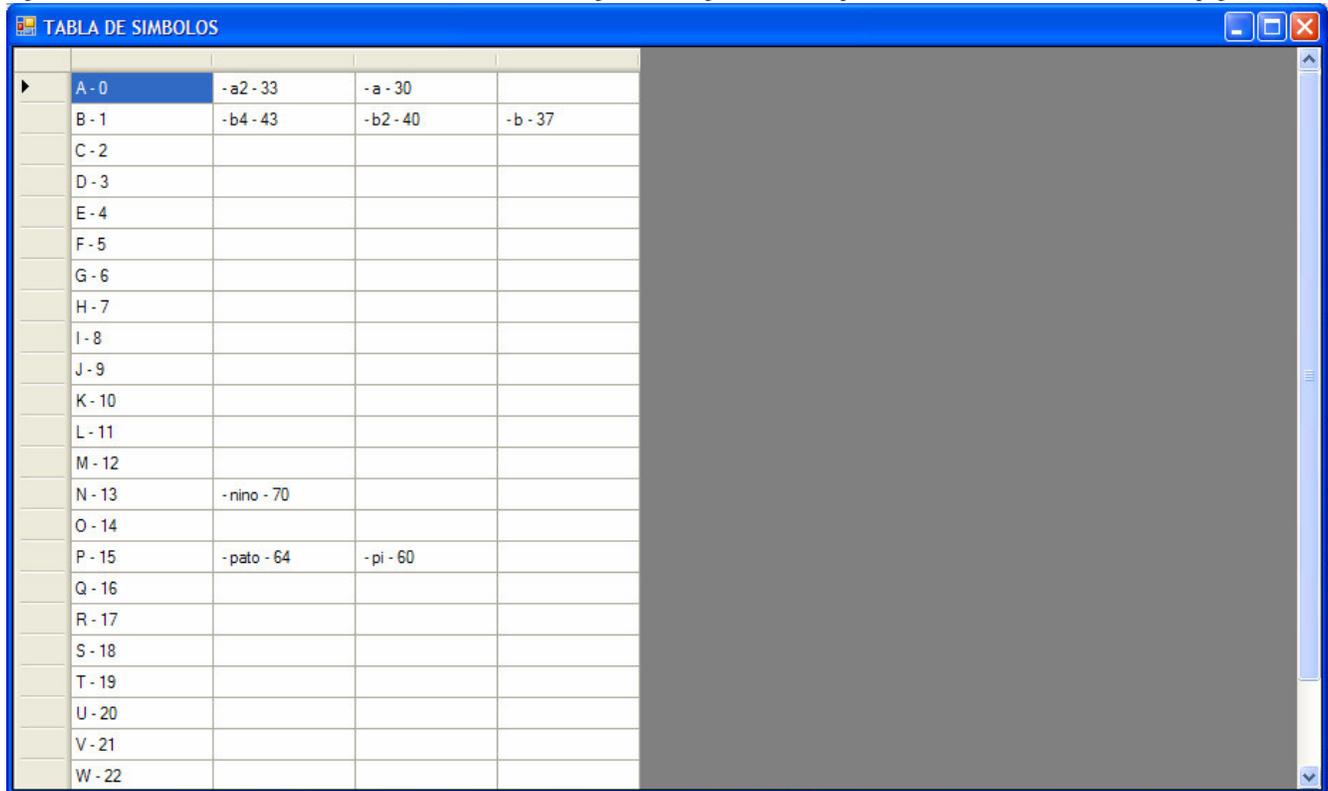


Fig. No. 8.2 Definición de la propiedad *DataGridview1* en *Form2.cs*.

El método *Mayor()* es usado para calcular el número máximo de columnas en el *dataGridView1*, suficientes para visualizar a cada lista enlazada de la tabla de símbolos. La columna 0 del componente *dataGridView1*, es reservada para visualizar la letra e índice correspondiente a la lista enlazada cuyos elementos se muestran en el renglón *i*-ésimo. La figura #8.3 muestra la ejecución para el grupo de sentencias, donde además se visualiza la tabla de símbolos.

```
inicio
var
    entero a, a2, b, b2,b4;
    real pi, pato, nino,x,y,z,z23;
fin
```

Tabla de símbolos en C# y su interacción con el analizador léxico.



▶	A - 0	- a2 - 33	- a - 30	
	B - 1	- b4 - 43	- b2 - 40	- b - 37
	C - 2			
	D - 3			
	E - 4			
	F - 5			
	G - 6			
	H - 7			
	I - 8			
	J - 9			
	K - 10			
	L - 11			
	M - 12			
	N - 13	- nino - 70		
	O - 14			
	P - 15	- pato - 64	- pi - 60	
	Q - 16			
	R - 17			
	S - 18			
	T - 19			
	U - 20			
	V - 21			
	W - 22			

Fig. No. 8.3 Tabla de símbolos con elementos y atributos.