

Análisis semántico -traductor descendente- usando polimorfismo con C#

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 un analizador semántico – traductor descendente- en C#. La gramática que se traduce consiste de 5 tipos de sentencias : (1) declaración de constantes, (2) declaración de variables, (3) asignación, (4) lectura y (5) visualización –salida-. El traductor descendente se compone de 5 objetos traductores; uno por cada tipo de instrucción. El polimorfismo que se propone, consiste de una clase base denominada *Traductor* y de 5 clases derivadas : *TradConst*, *TradVar*, *TradAsig*, *TradLeer* y *TradVisua*. La clase *Traductor* tiene un solo atributo `_errores`, el cual es un arreglo de **string** que contiene a cada uno de los errores que es posible encontrar durante el proceso de traducción de cada sentencia. La clase *Traductor* también contiene un método llamado *Traducir()*, y que es definido en cada una de las clases *TradConst*, *TradVar*, *TradAsig*, *TradLeer* y *TradVisua*. La llamada al método *Traducir()* es insertada dentro del método *Analiza()* de la clase *SintDescNRP*, a la que pertenece el objeto *oAnaSint* que efectúa el análisis sintáctico descendente no recursivo predictivo –ver el trabajo en el sitio [www.monografias.com/trabajos-pdf/software-didactico-construccion-analizadores-sintacticos.shtml](http://www.monografias.com/trabajos-pdf/software-didactico-construccion-analizadores-sintacticos/software-didactico-construccion-analizadores-sintacticos.shtml) -.

INDICE.

1. Introducción.	3
2. Tipos de sentencias y ejemplos de código fuente.	4
3. Gramática de contexto libre $G=(Vt, Vn, S, \Phi)$.	5
4. Analizador léxico.	6
5. Tabla de símbolos.	19
6. Analizador sintáctico descendente.	28
7. Traducciones a realizar.	39
8. Construcción del traductor descendente.	40
9. Traducción (a) : ALMACENAR EL TIPO DE DATO, NÚMERO DE BYTES Y EL VALOR DE LAS CONSTANTES DECLARADAS, EN LOS ATRIBUTOS RESPECTIVOS DE LA TABLA DE SÍMBOLOS.	43
10. Traducción (b) : ALMACENAR EL TIPO DE DATO, NÚMERO DE BYTES DE LAS VARIABLES DECLARADAS, EN LOS ATRIBUTOS RESPECTIVOS DE LA TABLA DE SÍMBOLOS.	57
11. Traducción (c) : INDICAR ERROR SI UNA CONSTANTE O UNA VARIABLE ESTÁ DUPLICADA EN SU DECLARACIÓN.	68
12. Traducción (d) : INDICAR ERROR SI SE TRATA DE ASIGNAR A UNA CONSTANTE EL VALOR DE UNA EXPRESIÓN DENTRO DE UNA SENTENCIA DE ASIGNACIÓN.	68
13. Traducción (e) : INDICAR ERROR SI UNA VARIABLE SE UTILIZA EN UNA EXPRESIÓN DE ASIGNACIÓN, SIN HABER SIDO DECLARADA.	71
14. Traducción (f) : REVISAR EL TIPO EN LAS EXPRESIONES -SENTENCIAS- DE ASIGNACIÓN. QUE CORRESPONDAN LOS TIPOS DEL OPERANDO IZQUIERDO Y DEL DERECHO.	74

1 Introducción.

El análisis semántico es una de las 3 fases de análisis que realiza un compilador. El programa fuente se ingresa a un análisis léxico que se encarga de reconocer tokens que existen en el programa fuente. El analizador léxico es un programa que utiliza AFD's para efectuar el reconocimiento de los tokens.

La fase del análisis sintáctico revisa que las sentencias compuestas por tokens respeten la sintáxis para cada una de ellas. La sintáxis se expresa por medio de una gramática de contexto libre. Los analizadores sintácticos de mas amplio uso son : los reconocedores descendentes y los reconocedores ascendentes. Una vez que el análisis sintáctico establece que no existen errores de sintáxis, es posible traducir las instrucciones en el programa fuente.

El análisis semántico permite traducir las sentencias además de realizar el chequeo de tipos de datos en las expresiones. En la traducción puede emplearse la herramienta formal teórica denominada traducción dirigida por sintáxis, donde se establecen las reglas semánticas para cada una de las producciones de la gramática de contexto libre, que requieren de traducción. En la etapa del análisis semántico es frecuente la comunicación con la tabla de símbolos, sea para escritura como para lectura de la información almacenada en dicha estructura de datos. Durante el proceso de traducción también es posible que se detecten errores, por lo que deberá el análisis semántico guardar estrecha relación con el administrador de errores –programa que maneja el error dentro de la fase de análisis-. En la fig. 1.1 se muestra el desglose de la etapa del análisis en un compilador.

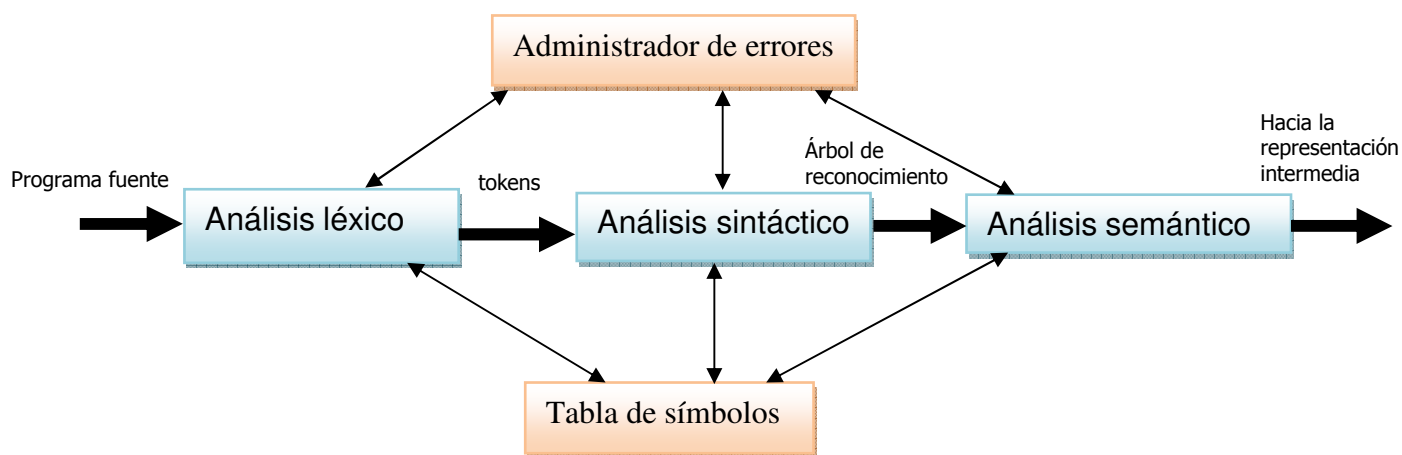


Fig. No. 1 Fase de análisis.

El libro *Compiladores : principios, técnicas y herramientas* de los autores Aho, Sethi y Ulman –conocido como el libro del “dragón”-, contiene los fundamentos teóricos y metodologías utilizados en este trabajo. Es recomendable hecharle un vistazo a los capítulos 1, 3, 4 y 5 antes de iniciar con esta lectura.

La finalidad de esta lectura consiste en escribir en C# una aplicación Windows que haga las funciones de un *traductor descendente* –analizador semántico-. Este traductor descendente deberá efectuar traducciones de sentencias previamente especificadas mediante una gramática de contexto libre. Para efectuar las diferentes traducciones que mas tarde serán mencionadas, se proponen 5 objetos traductores –uno para cada tipo de sentencia- los cuales formarán una lista heterogénea de objetos que se derivarán de una clase base llamada *Traductor*. El proceso de traducción de cada una de las 5 sentencias es introducido como un mensaje –rutina, como se conocía en la programación orientada a funciones-, dentro del método que efectúa el análisis sintáctico.

Las etapas propuestas para lograr el traductor descendente son :

- Tipos de sentencias y ejemplos de programa fuente a reconocer y traducir.
- Escritura de la gramática de contexto libre $G=(V_t, V_n, S, \phi)$, que denota al lenguaje en que es escrito el programa fuente.
- Construcción del analizador léxico que reconoce los símbolos terminales, V_t .
- Construcción de la tabla de símbolos.
- Construcción del analizador sintáctico descendente.
- Especificación de las traducciones a efectuar.
- Construcción del analizador semántico –traductor descendente-.

2 Tipos de sentencias y ejemplos de código fuente.

Durante algunos semestres he propuesto un lenguaje para el estudio de las diferentes fases de análisis, consistente de las instrucciones –sentencias- :

- 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 fuente para este lenguaje los muestro a continuación. Notemos que tenemos 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

```

3 Gramática de contexto libre $G=(Vt, Vn, S, \phi)$.

La gramática propuesta para denotar al lenguaje descrito en la sección 2 consiste de las producciones :

```

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

```

Además es simple reconocer los símbolos terminales que se encuentran en la gramática :

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

Los símbolos no terminales son :

$Vn = \{ P, C, K, R, Y, Z, V, B, I, S, A, E, T, F, L, O, W \};$

El símbolo de inicio S es :

$S = P$

4 Analizador léxico.

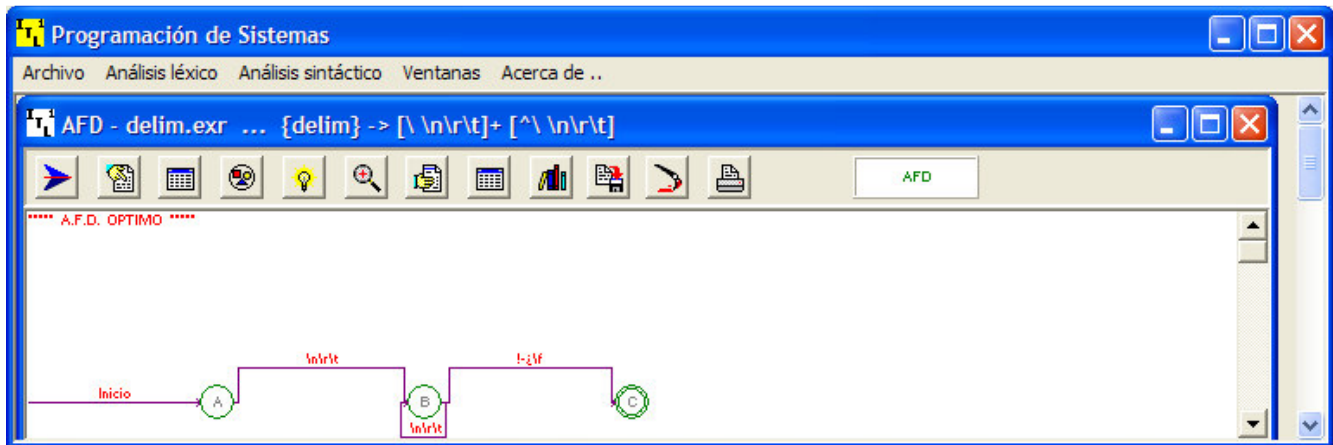
Agrupemos a los símbolos terminales de manera que a cada grupo le corresponda un AFD en el análisis léxico :

- *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 y números reales con al menos una cifra entera y al menos una cifra decimal. 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, en la dirección <http://www.monografias.com/trabajos-pdf/codigo-analisis-lexico-windows/codigo-analisis-lexico-windows.shtml>. 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]



Análisis semántico –traductor descendente- usando polimorfismo con C#

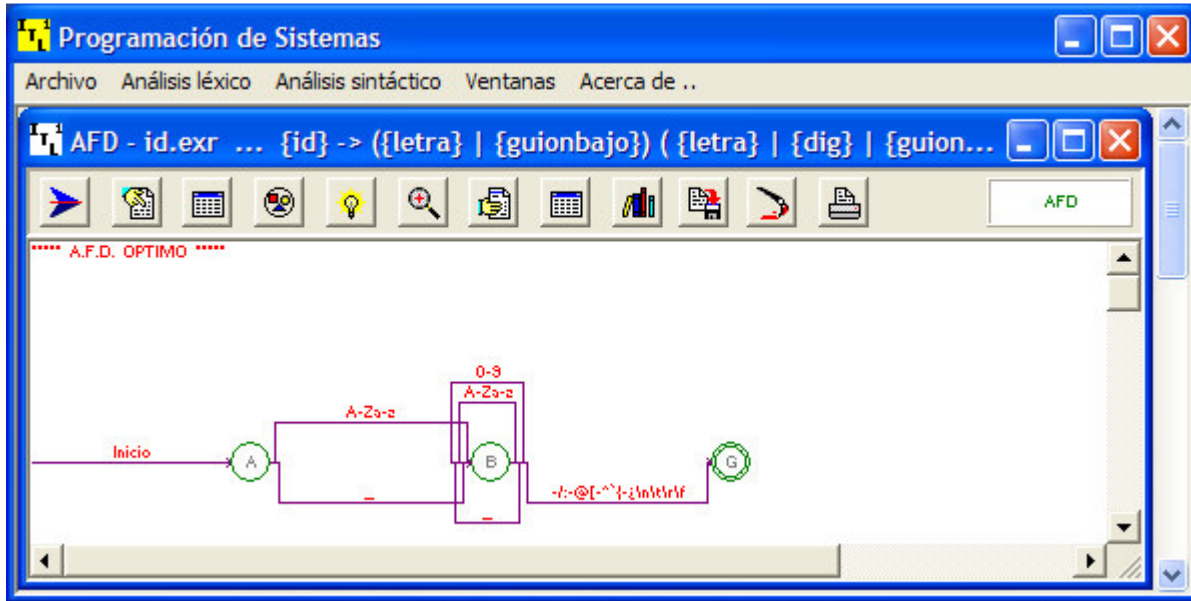
Ing. Francisco Ríos Acosta

Instituto Tecnológico de la Laguna, septiembre del 2009.

pag. 7 de 74

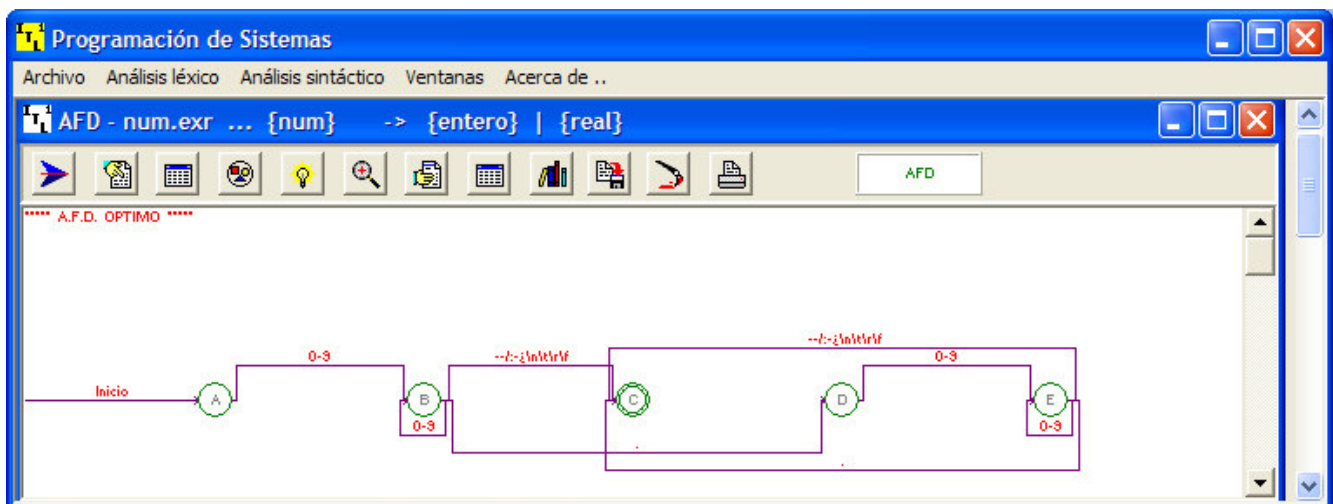
AFD id.- Reconoce al lenguaje de los identificadores permitidos en C#. La regla que usaremos está restringida a : cadenas que inicien con letra o guion bajo, seguida de 0 o mas letras, dígitos o guion bajo. También usaremos este autómata para reconocer a las palabras reservadas. Su expresión regular y su AFD son :

```
{letra} -> [A-Za-z]
{dig} -> [0-9]
{guionbajo} -> _
{id} -> ({letra} | {guionbajo}) ( {letra} | {dig} | {guionbajo} ) * [^A-Za-z0-9_]
```



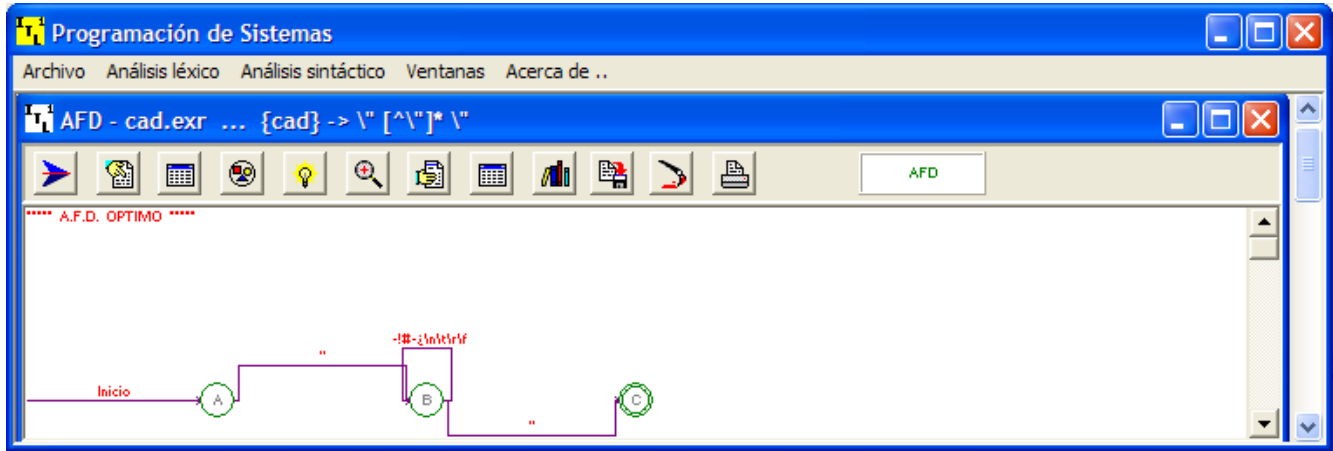
AFD num.- Vamos a limitar a los lexemas del token num a sólo a enteros y reales de al menos una cifra entera y al menos una cifra decimal. La expresión regular y su AFD óptimo son :

```
{entero} -> [0-9]+ [^.0-9]
{real} -> [0-9]+ \. [0-9]+ [^0-9]
{num} -> {entero} | {real}
```



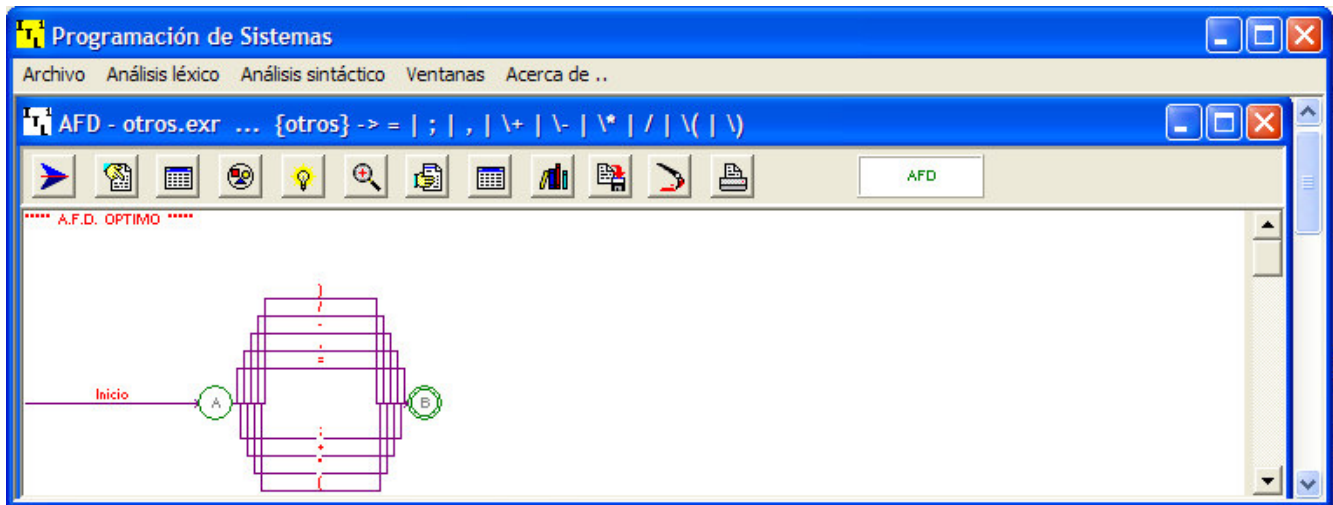
AFD *cad*.- Reconoce a las cadenas de caracteres encerradas entre comillas.

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



AFD *otros*.- Su expresión regular es :

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



El código generado por SP-PS1 para la clase *Lexico* es –se incluyen las modificaciones que puedes encontrar explicadas a detalle en la dirección <http://www.monografias.com/trabajos-pdf/software-didactico-construccion-analizadores-sintacticos/software-didactico-construccion-analizadores-sintacticos.shtml> - :

```
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"; ← 1
                    break;
                //----- Automata id-----
                case 1 : _tokens[_noTokens] = "id"; ← 2
                    break;
                //----- Automata num-----
                case 2 : _tokens[_noTokens] = "num";
                    break;
                //----- Automata cad-----
                case 3 : _tokens[_noTokens] = "cad";
                    break;
                //----- Automata otros-----
                case 4 : _tokens[_noTokens] = "otros";
                    break;
            }
            _lexemas[_noTokens++] = _lexema; ← 3
        }
        else ← 4
            _i++;
            _iniToken = _i;
    } ← 5
} // fin de la clase Lexico

```

La clase Automata es :

```

class Automata
{
    string _textoIma;
    int _edoAct;

    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 cad-----
        case 3 : _edoAct = 11;
                break;
        //----- Automata otros-----
        case 4 : _edoAct = 14;
                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„...†‡^%Š<@Ž□□'""
"•--™š>œ□žŸ ;ç£¤¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿\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="ABCDEFGHIJKLMNopqrstuvwxyz").IndexOf(c)>=0) _edoAct=4; else
                    if ((lenguaje="_").IndexOf(c)>=0) _edoAct=4; else
                    { i=iniToken;
                      return false; }
                    break;
            case 4 : c=SigCar(ref i);
                    if
                    ((lenguaje="ABCDEFGHIJKLMNopqrstuvwxyz").IndexOf(c)>=0) _edoAct=4; else
                    if ((lenguaje="_").IndexOf(c)>=0) _edoAct=4; else
                    if ((lenguaje="0123456789").IndexOf(c)>=0) _edoAct=4; else
                    if ((lenguaje=" !\"#$%&\'()*+,-./:;<=>?@[\]^`{|}~□€□, f„...†‡^%Š<@Ž□□'""
                    "•--™š>œ□žŸ ;ç£¤¥¦§¨©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;
            //----- Automata num-----
            case 6 : c=SigCar(ref i);
                    if ((lenguaje="0123456789").IndexOf(c)>=0) _edoAct=7; else
                    { i=iniToken;
                      return false; }
                    break;

```


Sigamos con la construcción de la aplicación Windows C# que sirva para analizar léxicamente un programa fuente de entrada, y que reconozca a los tokens en el conjunto V_t de la gramática propuesta.

La interfase gráfica de la aplicación Windows C# contiene los componentes : 2 Label's, 1 TextBox, 1 dataGridView y 1 Button. Agrega los componentes dejándoles su propiedad *Name* intacta. Caracteriza al componente dataGridView1 con la adición de sus 2 columnas TOKEN y LEXEMA.

La interfase gráfica de la aplicación es la mostrada en la figura #4.1.

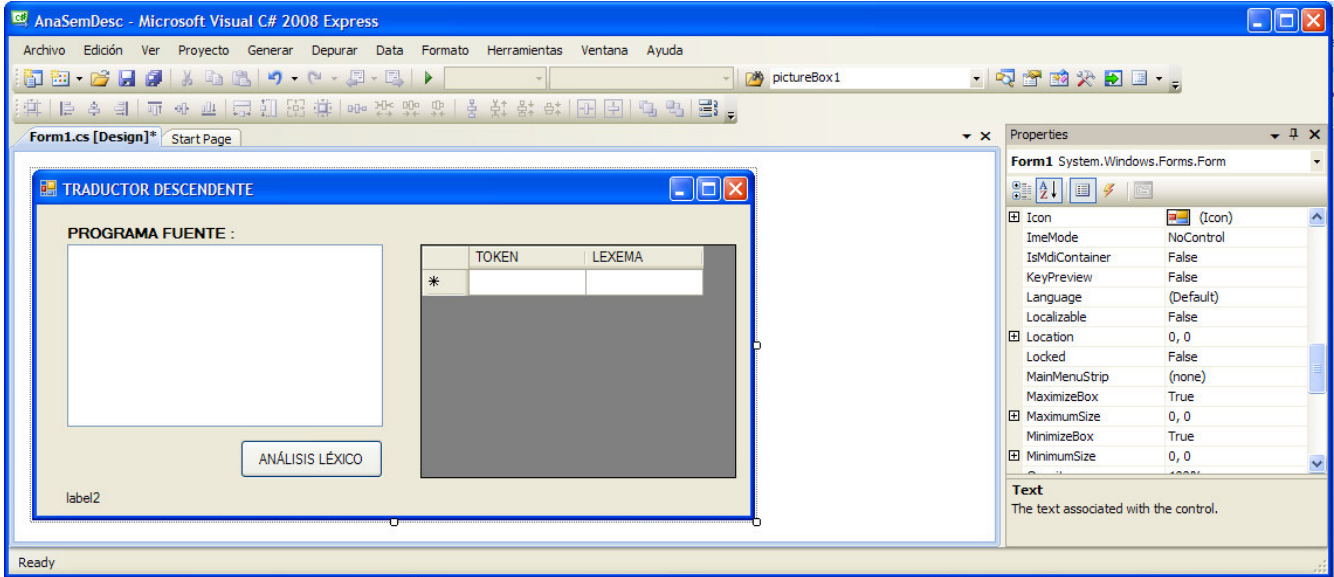


Fig. No. 4.1 Aplicación Windows C#, para el análisis léxico.

Ahora añadimos las clases *Lexico* y *Automata* al proyecto y les pegamos el código que hemos mostrado antes en esta misma sección, figura #4.2.

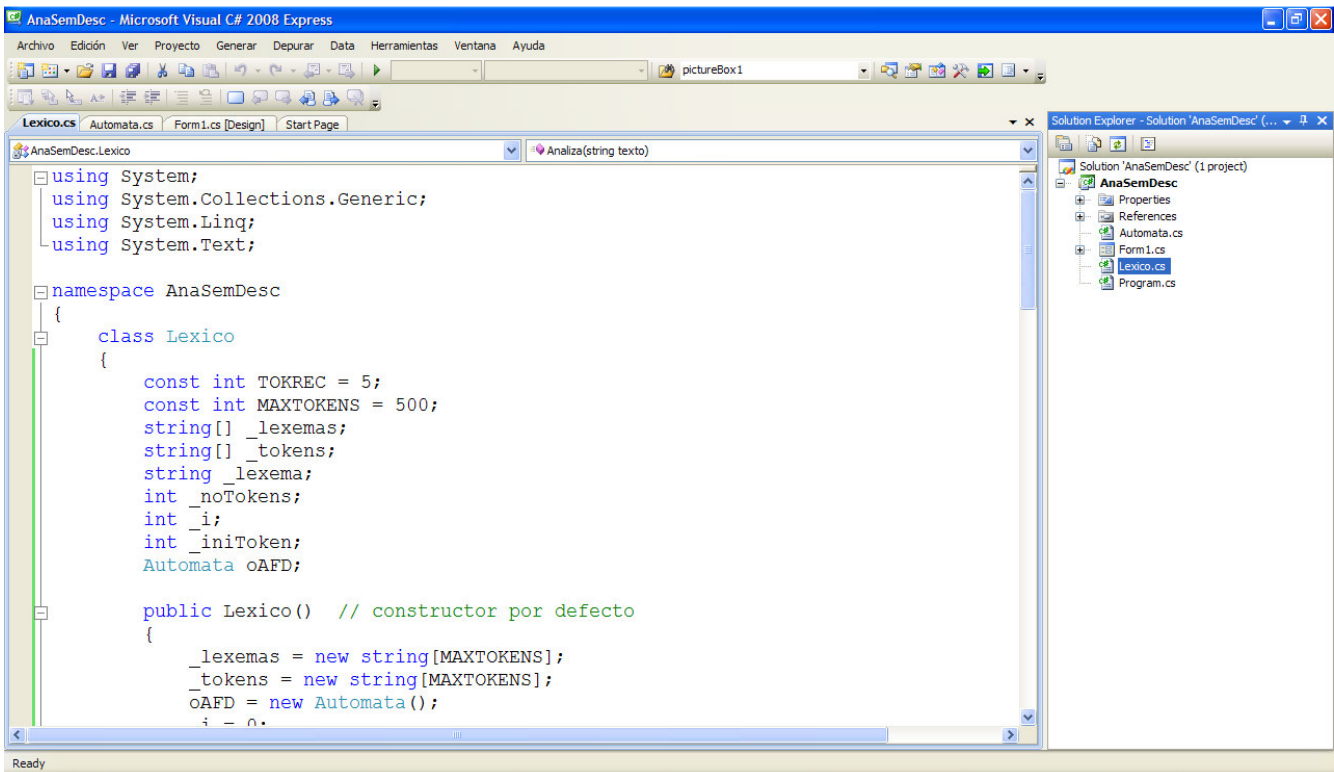


Fig. No. 4.2 Clases Lexico y Automata incluidas en el proyecto.

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

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

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 que hemos escrito en el evento `button1_Click()` :

```
public int NoTokens
{
    get { return _noTokens; }
}
public string[] Lexema
{
    get { return _lexemas; }
}
public string[] Token
{
    get { return _tokens; }
}
```

La figura #4.3 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 estan los tokens **cad**.

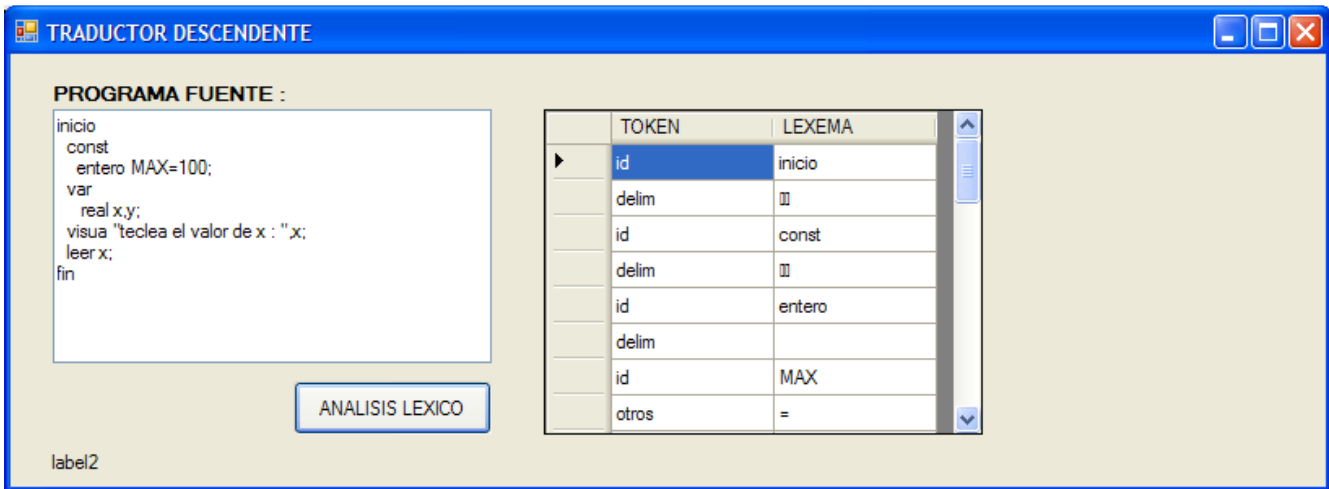


Fig. No. 4.3 Ejecución de la aplicación con el análisis léxico.

El análisis léxico no está bien del todo, ya que es necesario efectuar ciertas modificaciones las cuales han sido señaladas con cajas numeradas en la clase `Lexico`. Iniciemos con los cambios.

1

Los delimitadores no deben ser almacenados ya que sólo sirven para darle mayor legibilidad al programa fuente y como separadores de los tokens que se están reconociendo. Vayamos al método `Analiza()` de la clase `Lexico` para comentar la sentencia que almacena al token `delim`. La figura #4.4 muestra el cambio que ha incluido el comentario citado.

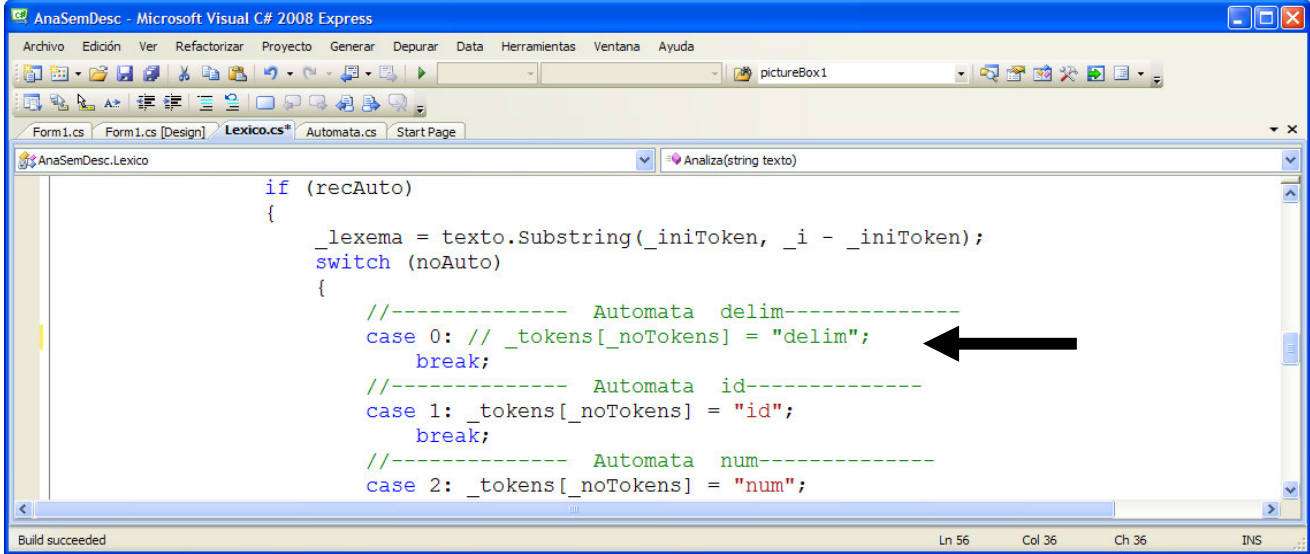


Fig. No. 4.4 Comentario de la sentencia para no almacenar al token `delim` en el método `Analiza()` de la clase `Lexico`.

3

Ahora debemos no almacenar el lexema del token `delim` que se ha comentado. Vemos que el número del autómata que reconoce al token `delim` es el 0, así que insertamos un `if` para evitar guardar al lexema correspondiente, figura #4.5.

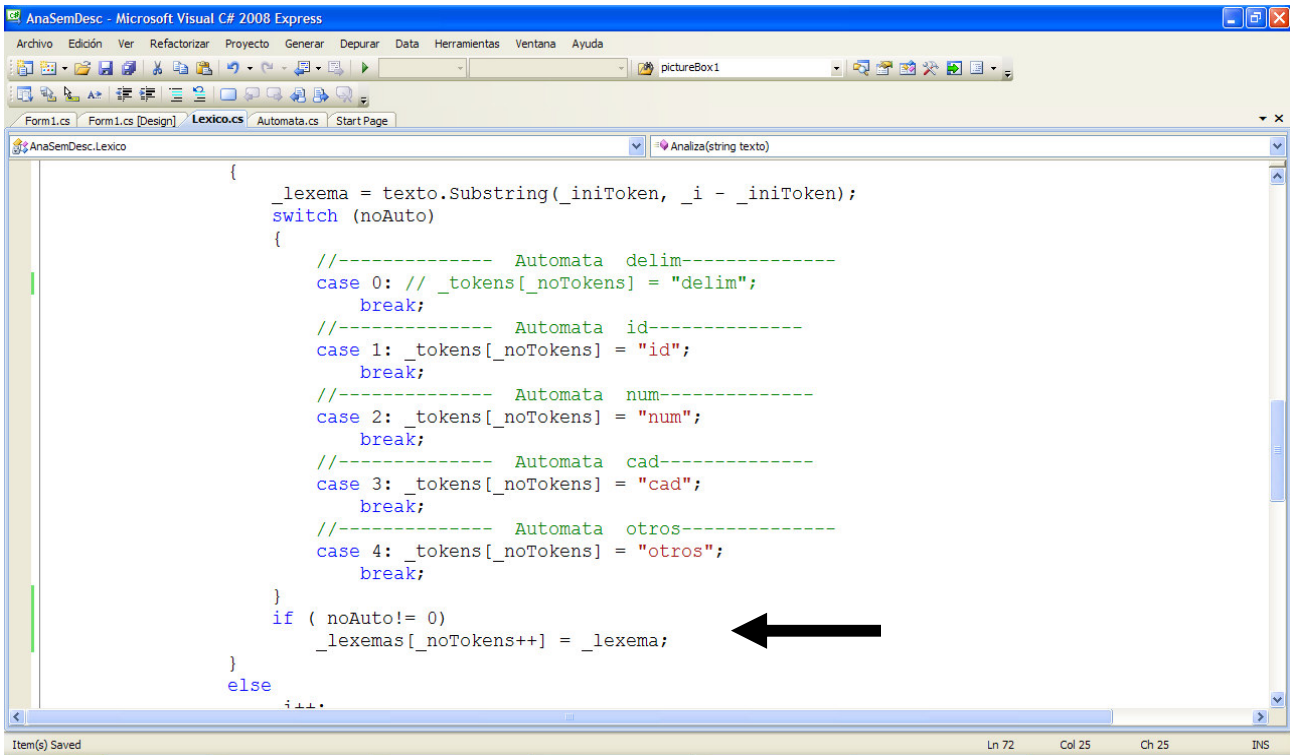
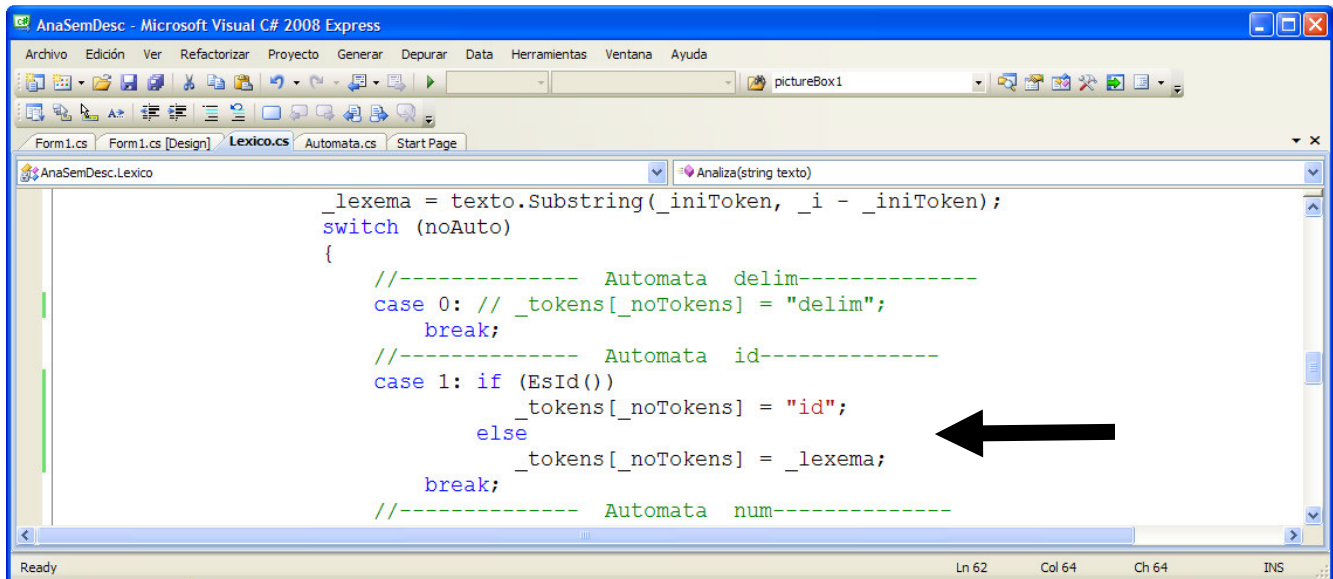


Fig. No. 4.5 Evitamos guardar el lexema para el token `delim`.

2

El AFD que reconoce a los *id* también lo usaremos para reconocer las palabras reservadas. Agregaremos la llamada al método `EsId()` que retorna **true** si el lexema es un *id*, retorna **false** si es una palabra reservada, figura #4.6. Consulta la referencia <http://www.monografias.com/trabajos-pdf/software-didactico-construccion-analizadores-sintacticos/software-didactico-construccion-analizadores-sintacticos.shtml>.

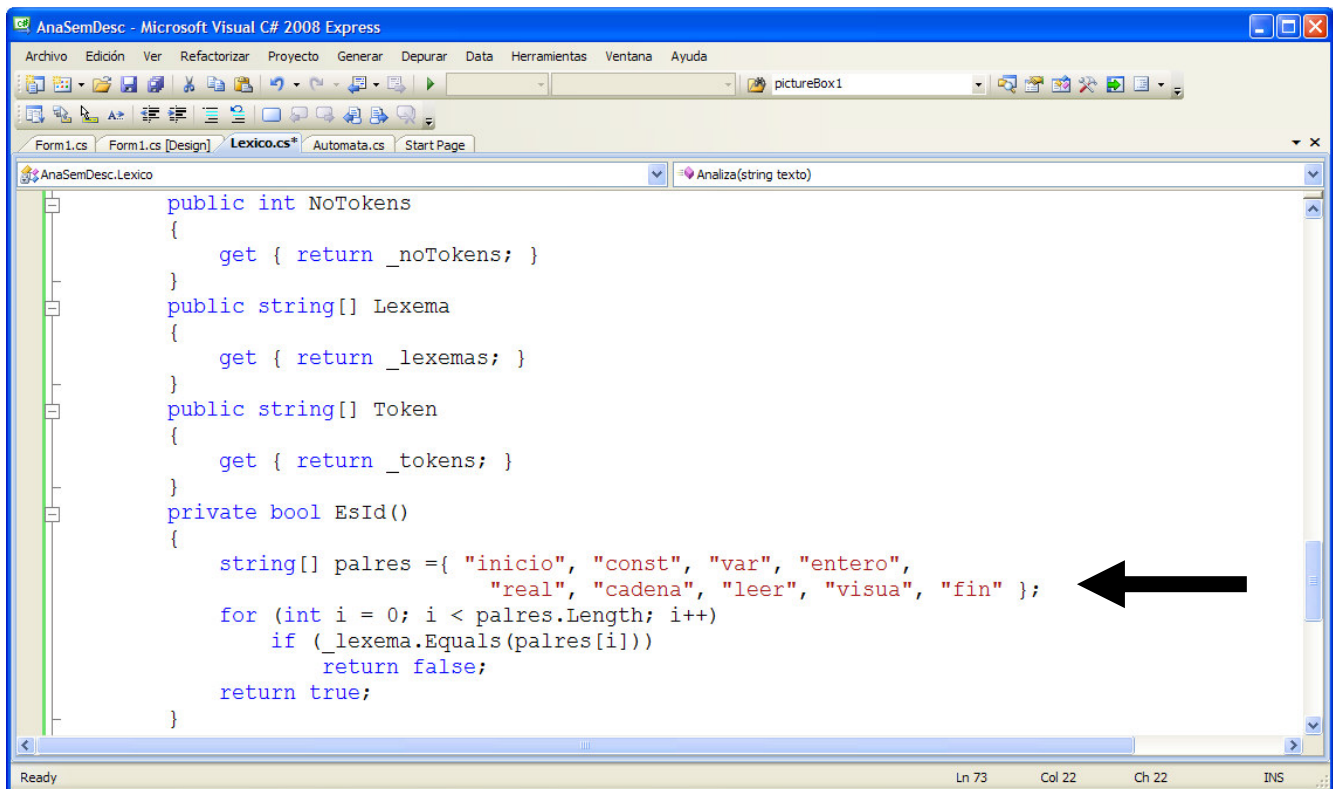


```
Lexico.cs
public class Lexico
{
    private string _lexema;
    private int _iniToken;
    private int _finToken;
    private string[] _tokens;
    private int _noTokens;

    public Lexico(string texto)
    {
        _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-----
        }
    }
}
```

Fig. No. 4.6 Llamada al método `EsId()`.**5**

El método `EsId()` lo agregamos en la clase `Lexico`, figura #4.7.



```
Lexico.cs
public class Lexico
{
    public int NoTokens
    {
        get { return _noTokens; }
    }
    public string[] Lexema
    {
        get { return _lexemas; }
    }
    public string[] Token
    {
        get { return _tokens; }
    }
    private bool EsId()
    {
        string[] palres = { "inicio", "const", "var", "entero",
                           "real", "cadena", "leer", "visua", "fin" };
        for (int i = 0; i < palres.Length; i++)
            if (_lexema.Equals(palres[i]))
                return false;
        return true;
    }
}
```

Fig. No. 4.7 Definición del método `EsId()`.

Sólo queda por sustituir en los tokens *otros* el guardar el lexema en lugar del nombre del token. Observemos que también lo hicimos en las palabras reservadas, donde almacenamos el atributo `_lexema` en lugar del nombre *palres*. La figura #4.8 muestra la modificación del código.

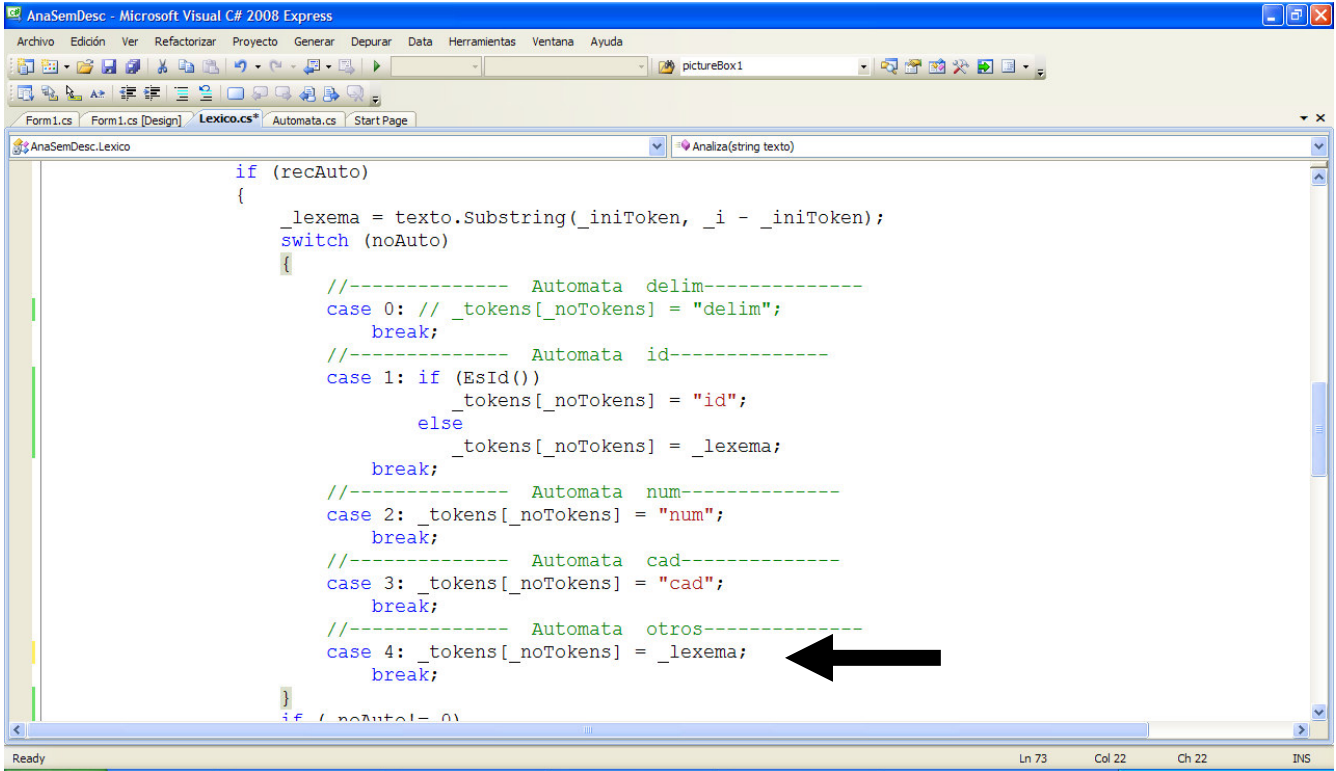


Fig. No. 4.8 Atributo `_lexema` es guardado en lugar del nombre del token *otros*.

Una vez hechas las modificaciones (menos una, que explicaremos en los párrafos siguientes), ejecutamos la aplicación para observar que todo esté bien, según vemos en la figura #4.9.



Fig. No. 4.9 Ejecución del análisis léxico.

4

Esta última modificación al analizador léxico consiste en la forma de recuperarse de un error durante el análisis léxico. Notemos que la recuperación del error consiste de “saltar” el carácter que no es reconocido por nuestro analizador léxico, mediante el incremento del atributo `_i`.

Vamos a cambiar la forma de recuperación del error mientras se efectúa el análisis léxico, modificando el tipo de retorno del método `Analiza()` de `void` a `bool`. De esta manera, si ocurre un error retornaremos **false** como resultado del análisis léxico, de lo contrario retornamos un **true** indicando la ausencia de errores durante el análisis léxico. La figura #4.10 contiene el cambio mencionado.

```

public bool 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;
    }
}

```

Fig. No. 4.10 Modificación del tipo de retorno del método `Analiza()`.

Agreguemos los retornos en el caso de error y en el caso de éxito, según se indica en la figura #4.11.

```

case 1: if (EsId())
        _tokens[_noTokens] = "id";
        else
            _tokens[_noTokens] = _lexema;
        break;
//----- Automata num-----
case 2: _tokens[_noTokens] = "num";
        break;
//----- Automata cad-----
case 3: _tokens[_noTokens] = "cad";
        break;
//----- Automata otros-----
case 4: _tokens[_noTokens] = _lexema;
        break;
    }
    if (noAuto != 0)
        _lexemas[_noTokens++] = _lexema;
    else
        return false;
    _iniToken = _i;
}
return true;
// fin del método Analiza()
public int NoTokens

```

Fig. No. 4.11 Inclusión de las sentencias `return` para error y éxito del análisis léxico.

Lo restante es modificar en nuestra aplicación el código del botón `button1`, según se indica en la figura #4.12.

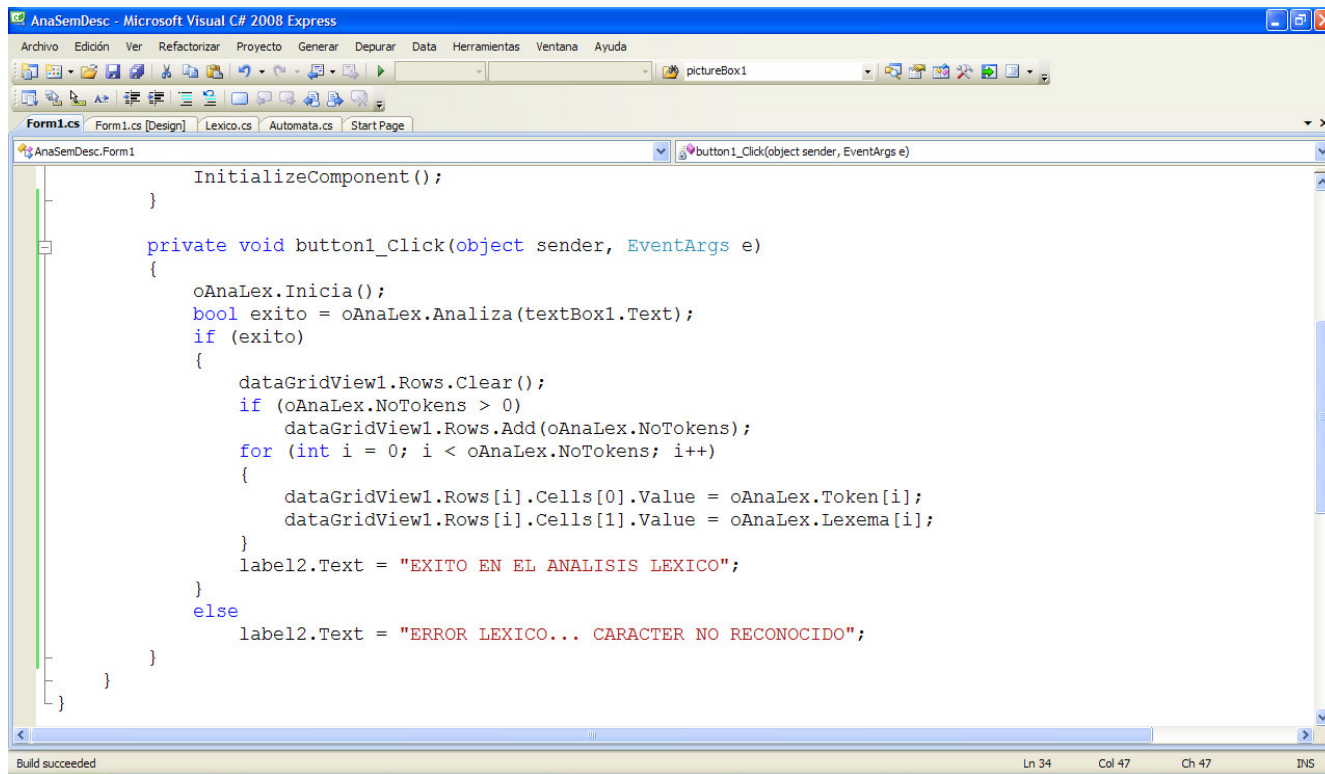


Fig. No. 4.12 Código `button1_Click()` modificado.

Ejecutemos la aplicación para observar si no hay errores.

5 Tabla de símbolos.

En esta sección nos encargaremos de construir la tabla de símbolos y la ensamblaremos junto al analizador léxico que ya tenemos en la aplicación. Consulta la referencia <http://www.monografias.com/trabajos-pdf/tabla-simbolos-interaccion-analizador-lexico/tabla-simbolos-interaccion-analizador-lexico.shtml> para mayor detalle de los conceptos que en esta sección mencionaremos.

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 sólo al token *id*. Los tokens *num* y *cad* los dejaremos para verlos posteriormente. El token *id* puede representar a 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 –programa fuente-.

El analizador léxico al reconocer el token *id*, lo instala en la tabla de símbolos almacenando 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*. Según la referencia que hemos citado en esta misma sección, los nodos de las listas de la tabla de símbolos contienen objetos de una clase, que llamaremos *TipoElem*.

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

La agregamos de una buena vez a nuestra aplicación, figura #5.1.

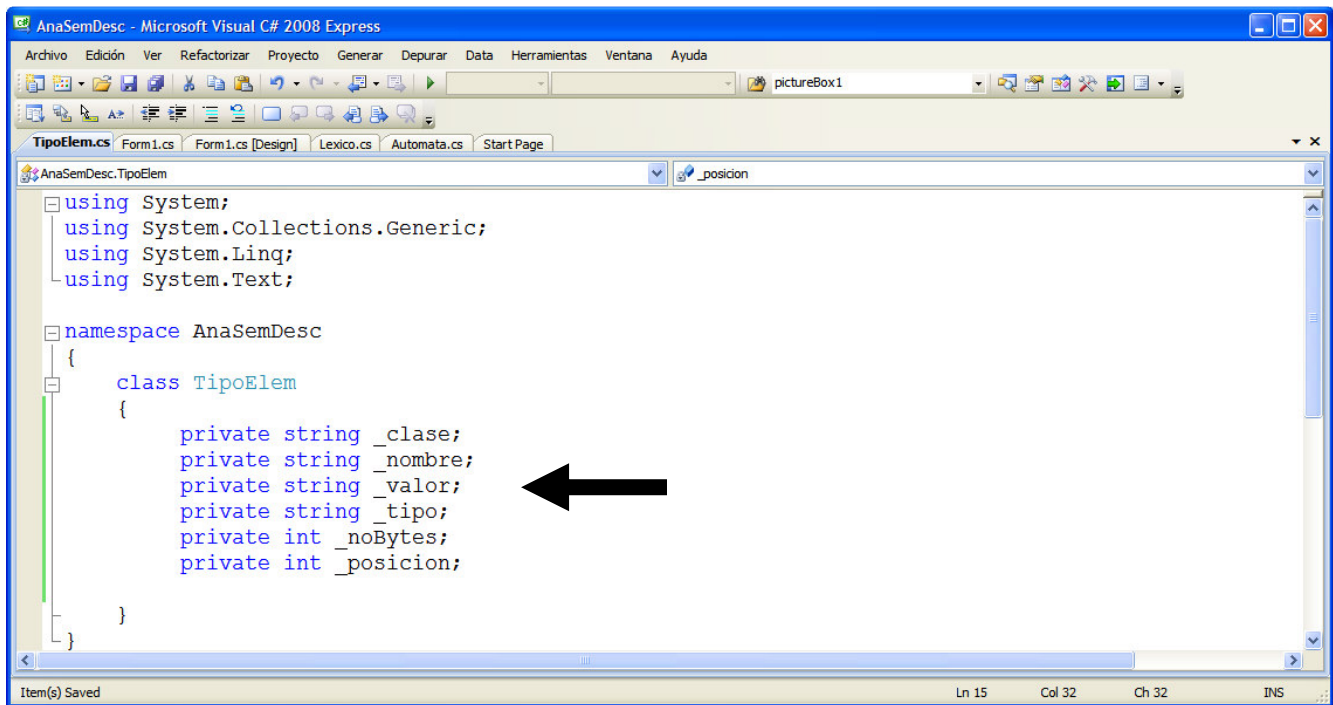


Fig. No. 5.1 Clase *TipoElem* agregada a la aplicación.

Sigamos con la adición de la clase *TablaSimbolos* al proyecto, incluyendo los métodos que se indican a continuación.

```
class TablaSimbolos
{
    Lista[] _elems;

    // métodos -----
    public TablaSimbolos(int noElems)
    {
        _elems = new Lista[noElems];
        for (int i = 0; i < _elems.Length; i++)
            _elems[i] = new Lista();
    }

    public void Inicia()
    {
        for (int i = 0; i < _elems.Length; i++)
        {
```

```

        _elems[i].Cab = null;
        _elems[i].NoNodos = 0;
    }
}

} // fin de la clase TablaSimbolos

```

Les toca el turno a las clases `Lista` y `Nodo` para ser añadidas al proyecto.

```

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; }
    }
} // fin de la clase Lista

```

```

class Nodo
{
    TipoElem _oInfo;
    Nodo _sig;

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

} // fin de la clase Nodo

```

El explorador de soluciones después de la inserción de estas 4 clases, se muestra en la figura 5.2.

Ahora definimos 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 carácter 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.

```

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

```



```
TablaSimbolos oTablaSimb = new TablaSimbolos(26);
```

```
public Form1()
{
    InitializeComponent();
}
```

...

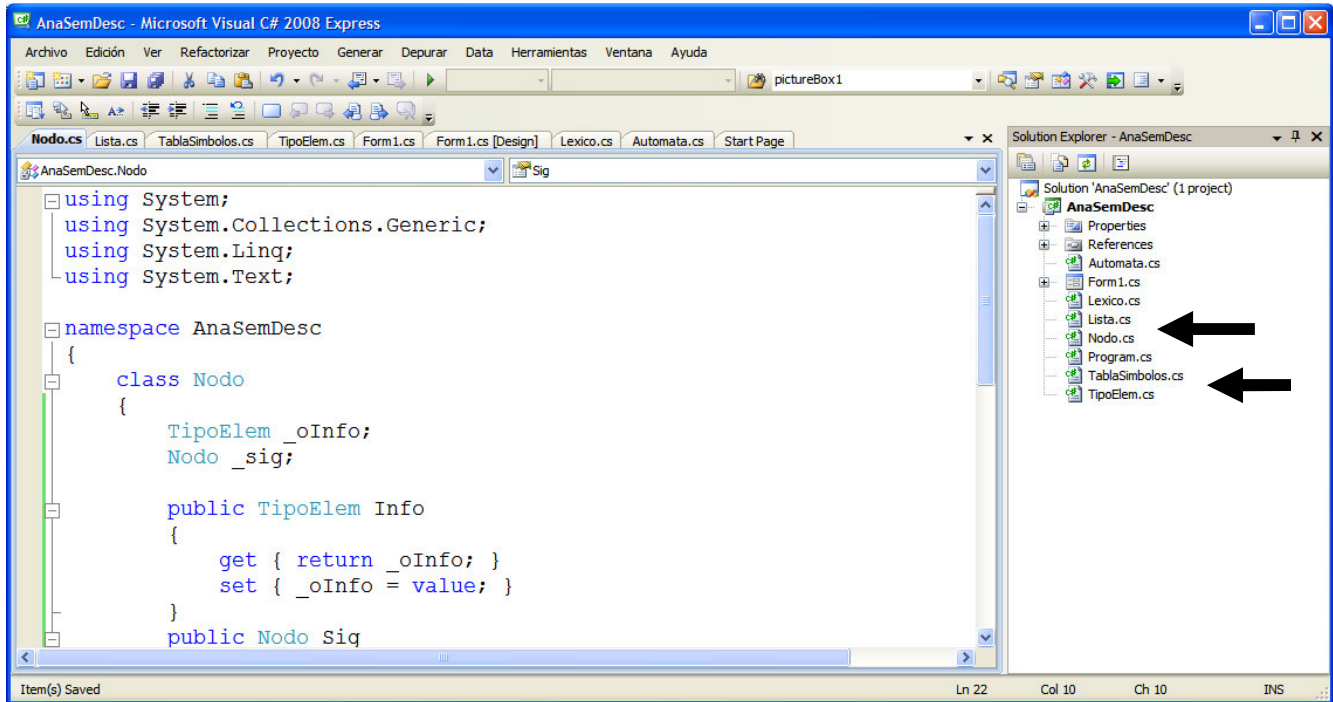


Fig. No. 5.2 Explorador de soluciones mostrando las 4 clases incrustadas.

Compilamos y ejecutemos la aplicación sólo para observar que no existan errores en el código que hemos añadido, figura #5.3.

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.

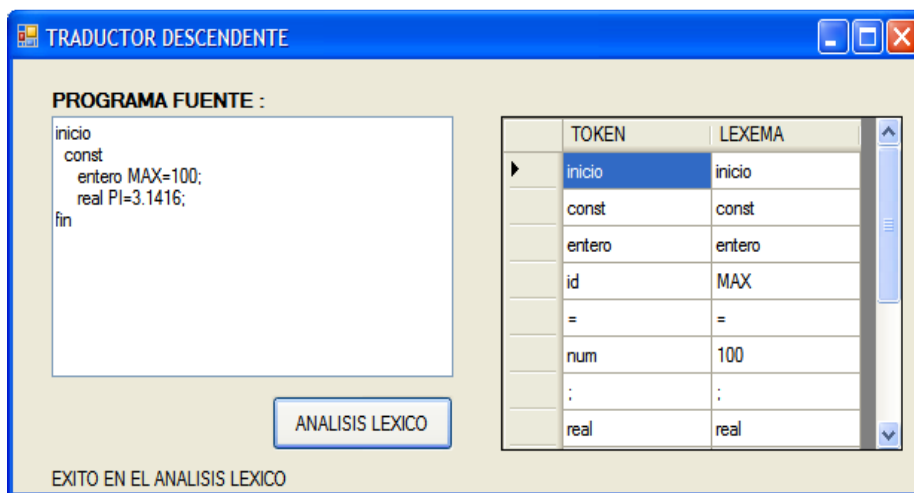
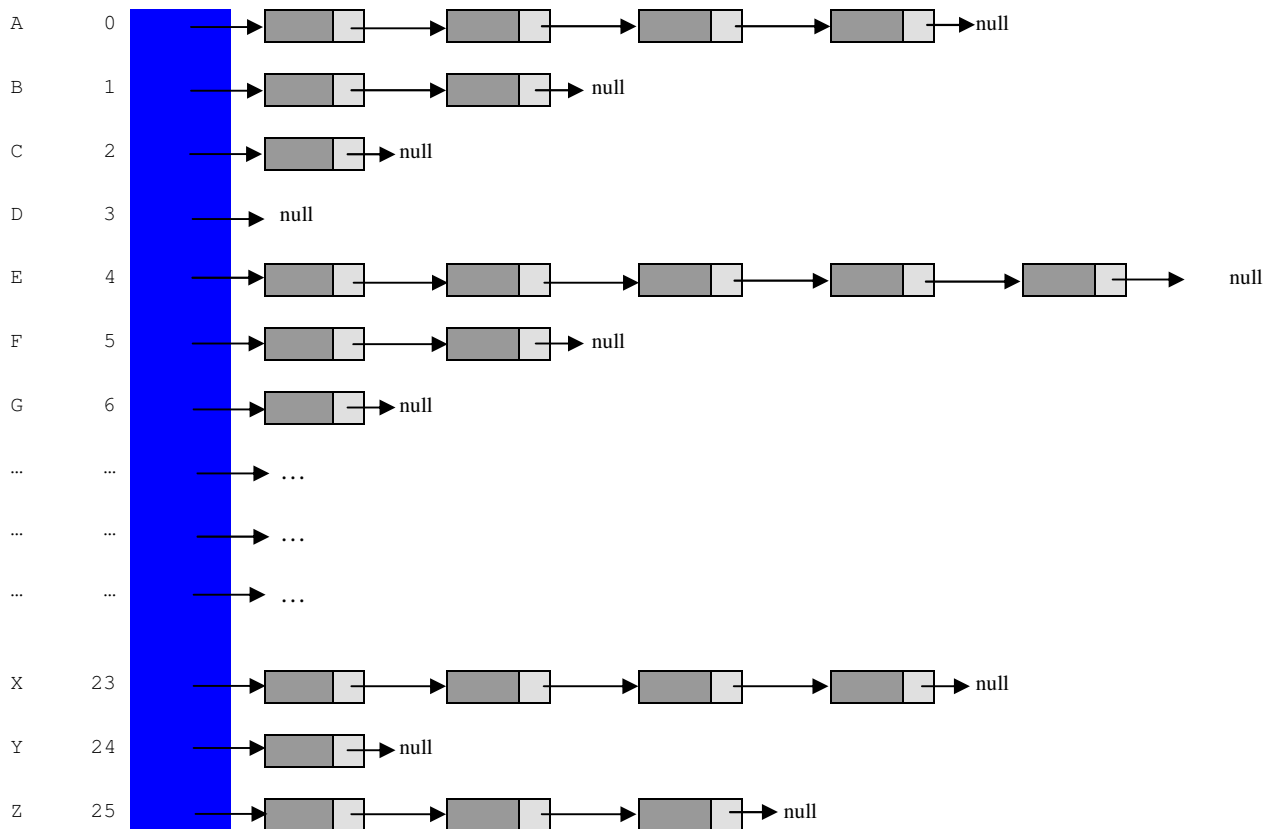


Fig. No. 5.3 Ejecución de la aplicación.

La tabla de símbolos se dimensionó a 26 listas enlazadas, cada una correspondiente a una letra del abecedario.



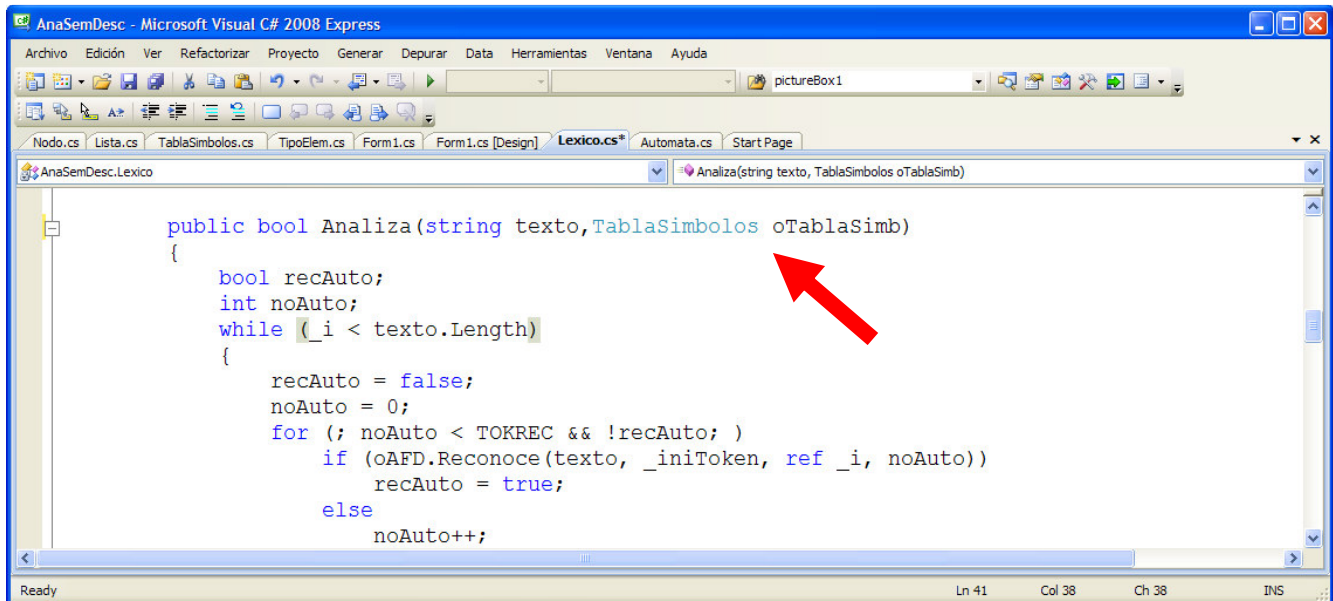
Vamos a insertar la instalación de un *id* durante el análisis léxico siguiendo los pasos :

- 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**. También se incluye el mensaje que inicializa al objeto *oTablaSimb*. Recordemos que la llamada se encuentra dentro del método *button1_Click()* de la aplicación *Form1.cs*.

```
private void button1_Click(object sender, EventArgs e)
{
    oAnaLex.Inicia();
    oTablaSimb.Inicia();
    bool exito = oAnaLex.Analiza(textBox1.Text, oTablaSimb);
    if (exito)
    {
        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];
        }
        label2.Text = "EXITO EN EL ANALISIS LEXICO";
    }
    else
        label2.Text = "ERROR LEXICO... CARACTER NO RECONOCIDO";
}
}
```

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 mostrado en la figura #5.4 :



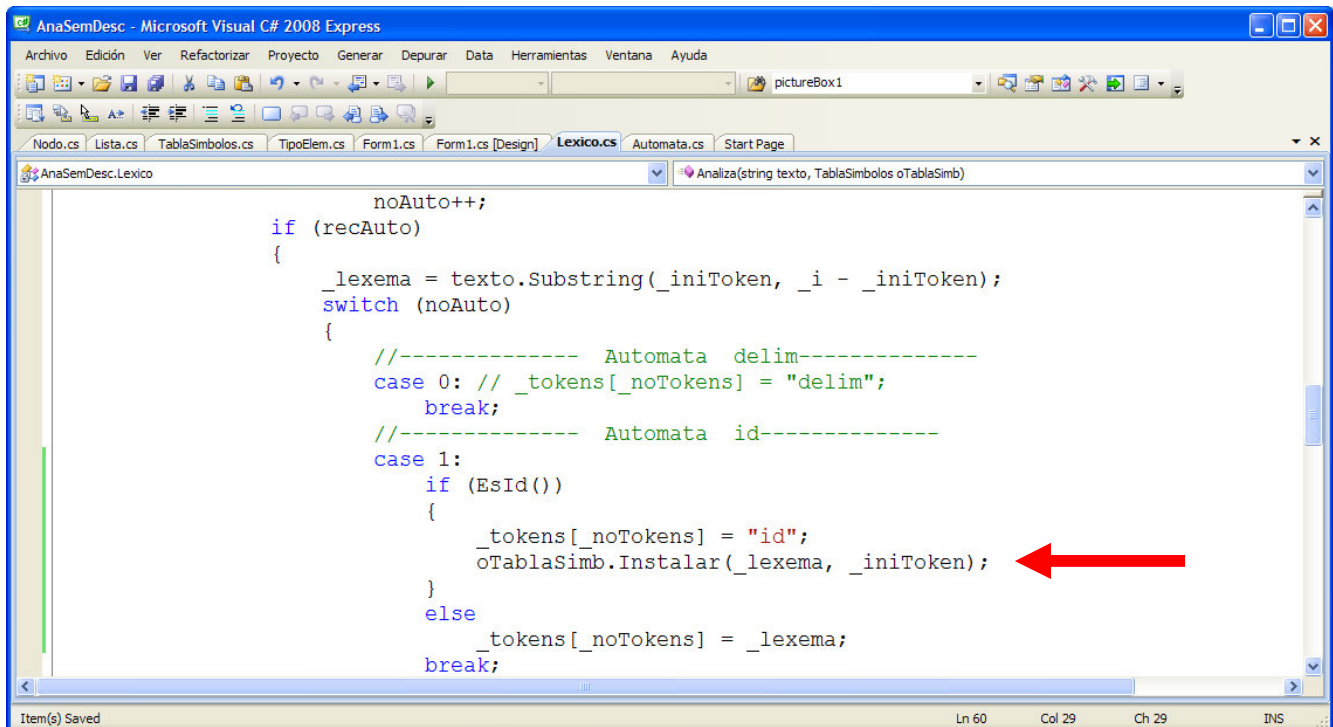
```

public bool Analiza(string texto, TablaSimbolos oTablaSimb)
{
    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++;
    }
}

```

Fig. No. 5.4 Nuevo encabezado del método *Analiza()* de la clase *Lexico*.

Instalación del *id* dentro del método *Analiza()* .- 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 #5.5.



```

        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";
                    oTablaSimb.Instalar(_lexema, _iniToken);
                }
                else
                    _tokens[_noTokens] = _lexema;
                break;
        }
    }
}

```

Fig. No. 5.5 Mensaje al objeto *oTablaSimb* para instalar el *id*.

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)==null)
    {
        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 es 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 *una referencia al nodo donde reside el token*, de lo contrario retorna el valor **null**. Necesitamos agregar la definición de este método en la clase `TablaSimbolos` según lo mostramos enseguida :

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

Notemos que el método retorna la referencia `refLista` 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 :

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

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

Para visualizar la tabla de símbolos utilizaremos una nueva forma *Form2.cs*, que añadimos usando la opción del menú *Project | Add Windows Form*. La propiedad *Text* de la forma *Form2.cs* debe asignarse el valor : TABLA DE SIMBOLOS.

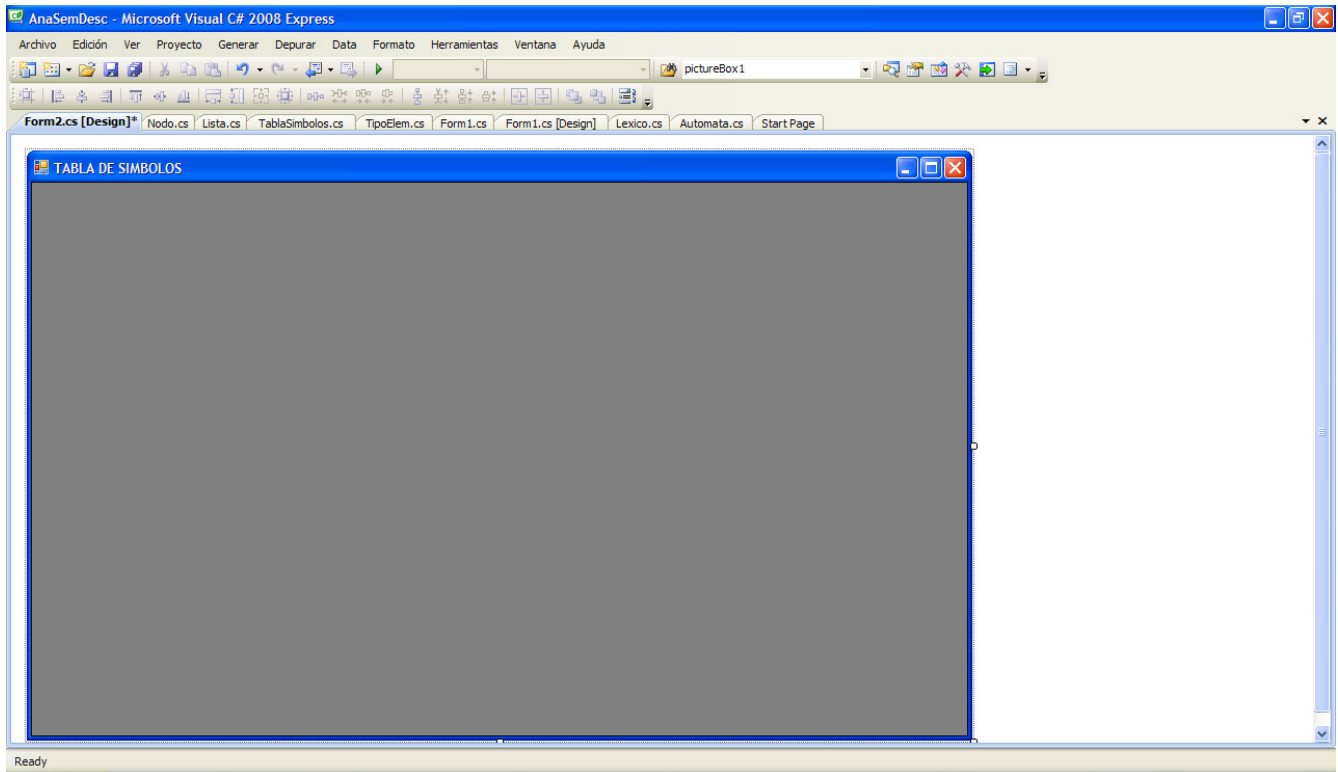


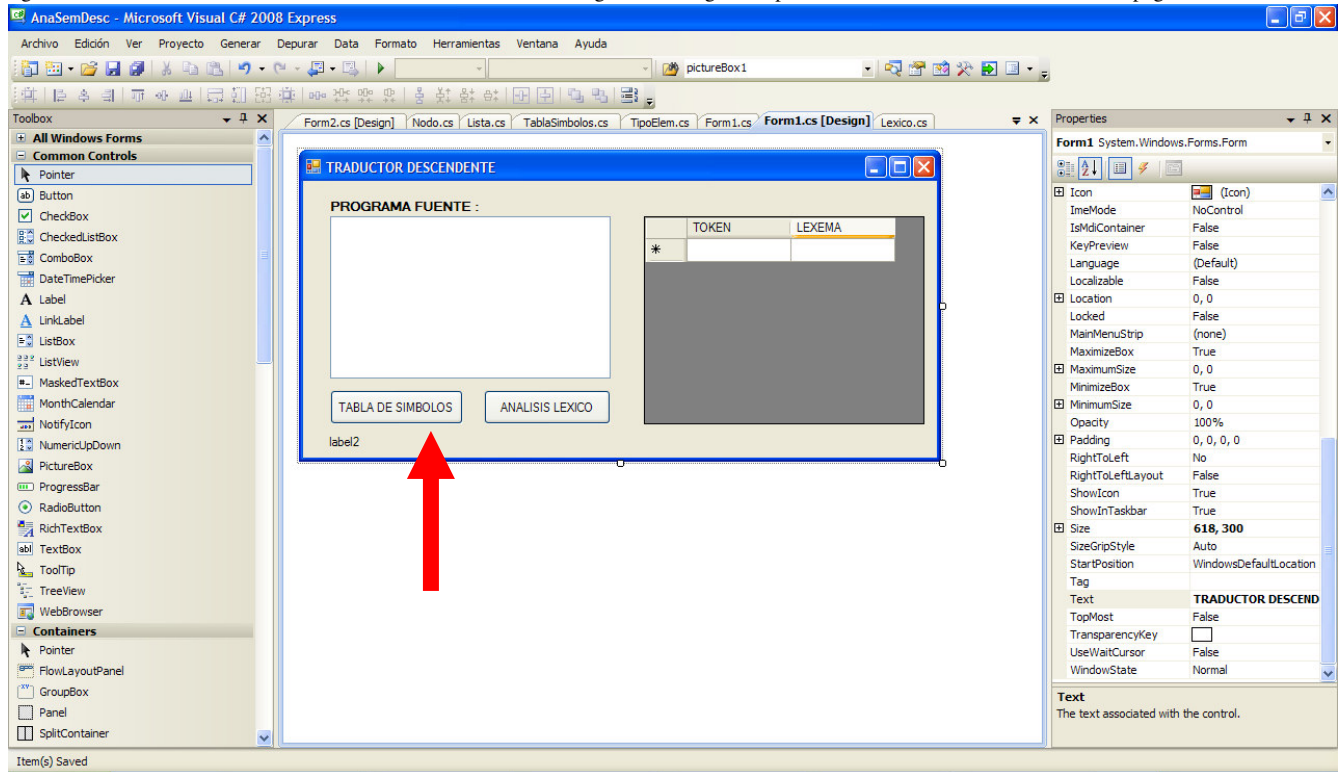
Fig. No. 5.6 *Form2.cs* para visualizar la tabla de símbolos.

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 #5.7 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();
}
```

Fig. No. 5.7 Botón `button2` para visualizar la forma `Form2.cs`.

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`.

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();
}
```

Observemos el mensaje `formaTablaSimb.DataGridView1` que retorna una referencia al componente `dataGridView1` por medio de la propiedad `DataGridView1`. Debemos agregar esta propiedad en la forma citada en su archivo `Form2.cs` –ver figura #5.8–.

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.Nombre + " - " +
                refNodo.Info.Posicion.ToString();
        }
    }
}
```

```

        }
    }
}

```

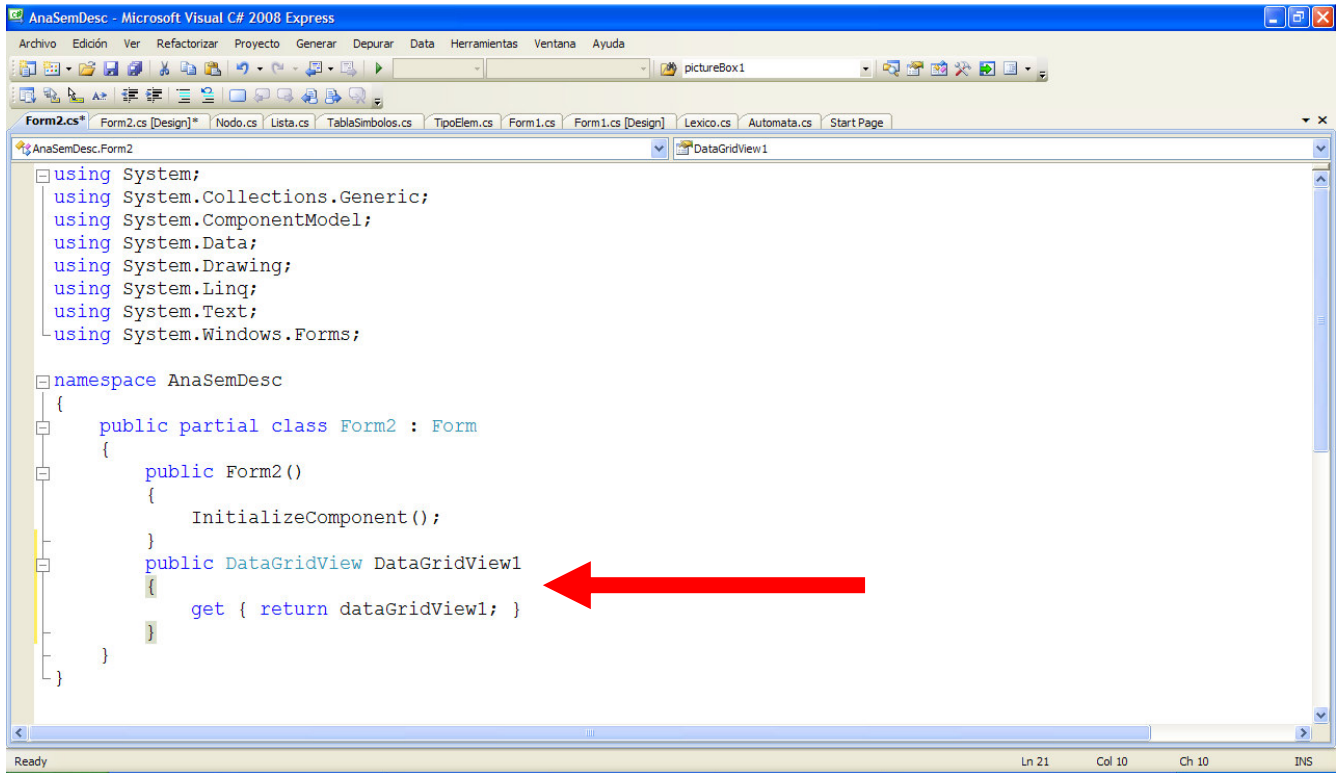


Fig. No. 5.8 Definición de la propiedad *DataGridView1* en *Form2.cs*.

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; }
}

```

La figura #5.9 muestra la ejecución de la aplicación para la entrada :

```

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

```

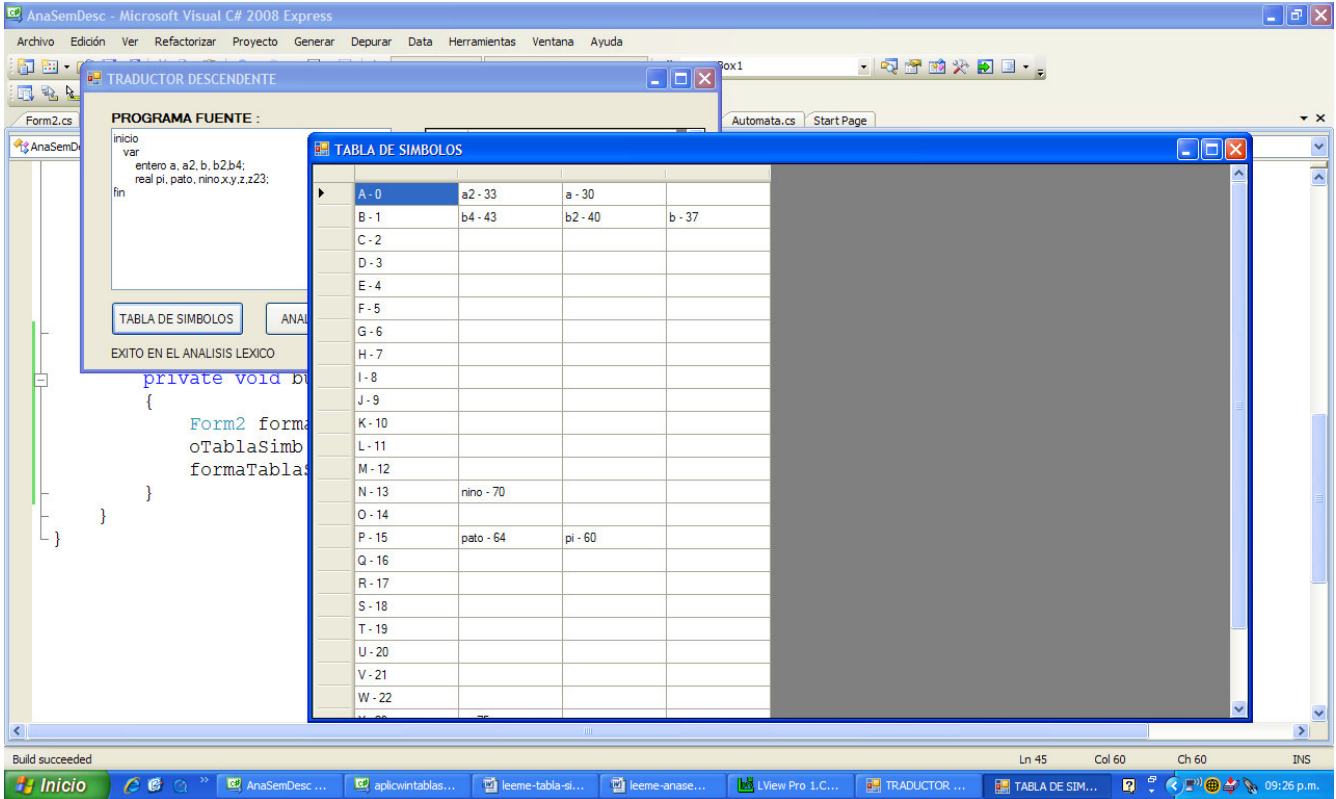



Fig. No. 5.9 Visualización de la tabla de símbolos.

6 Analizador sintáctico descendente.

Después de haber escrito el analizador léxico y la tabla de símbolos, estamos listos para agregar el código que cumple con el análisis sintáctico. El tipo de reconocedor que vamos a escribir es un reconocedor descendente, debido a que el objetivo de este trabajo es la construcción de un traductor descendente –analizador semántico descendente-.

Nos basaremos en el trabajo <http://www.monografias.com/trabajos-pdf/software-didactico-construccion-analizadores-sintacticos/software-didactico-construccion-analizadores-sintacticos.shtml> y el software RD-NRP para escribir el código del analizador sintáctico.

Usemos pues el RD-NRP para generar el código de la clase SintDescNRP para la gramática de contexto libre que hemos propuesto en la sección 3 y que es listado a continuación.

```

class SintDescNRP
{
    public const int NODIS=5000;

    private Pila _pila;
    private string[] _vts={ "", "inicio", "fin", "const", "id", "=", ";", "entero", "cadena", "real",
        "num", "cad", "var", "(", "+", "-", "*", "/", "(", ")", "leer", "visua", "$" };
    private string[] _vns={ "", "P", "C", "C'", "K", "R", "R'", "Y", "Z", "V", "B", "B'", "I", "I'",
        "S", "S'", "A", "A'", "E", "E'", "T", "T'", "F", "L", "O", "W", "W'" };
    private int[,] _prod={{1,3,-1,2,-2,0,0,0}, // P->inicio C fin
        {2,2,4,3,0,0,0,0}, // C->K C'
        {2,2,9,14,0,0,0,0}, // C->V S
        {2,1,14,0,0,0,0,0}, // C->S
        {3,1,14,0,0,0,0,0}, // C'->S
        {3,2,9,14,0,0,0,0}, // C'->V S
        {4,2,-3,5,0,0,0,0}, // K->const R
        {5,6,7,-4,-5,8,-6,6}, // R->Y id = Z ; R'
        {6,6,7,-4,-5,8,-6,6}, // R'->Y id = Z ; R'
        {6,0,0,0,0,0,0,0}, // R'->ε
    
```



```

    {7,1,-7,0,0,0,0,0}, // Y->entero
    {7,1,-8,0,0,0,0,0}, // Y->cadena
    {7,1,-9,0,0,0,0,0}, // Y->real
    {8,1,-10,0,0,0,0,0}, // Z->num
    {8,1,-11,0,0,0,0,0}, // Z->cad
    {9,2,-12,10,0,0,0,0}, // V->var B
    {10,4,7,12,-6,11,0,0}, // B->Y I ; B'
    {11,4,7,12,-6,11,0,0}, // B'->Y I ; B'
    {11,0,0,0,0,0,0,0}, // B'->ε
    {12,2,-4,13,0,0,0,0}, // I->id I'
    {13,3,-13,-4,13,0,0,0}, // I'->, id I'
    {13,0,0,0,0,0,0,0}, // I'->ε
    {14,2,16,15,0,0,0,0}, // S->A S'
    {14,2,23,15,0,0,0,0}, // S->L S'
    {14,2,24,15,0,0,0,0}, // S->O S'
    {15,2,16,15,0,0,0,0}, // S'->A S'
    {15,2,23,15,0,0,0,0}, // S'->L S'
    {15,2,24,15,0,0,0,0}, // S'->O S'
    {15,0,0,0,0,0,0,0}, // S'->ε
    {16,3,-4,-5,17,0,0,0}, // A->id = A'
    {17,2,18,-6,0,0,0,0}, // A'->E ;
    {17,2,-11,-6,0,0,0,0}, // A'->cad ;
    {18,2,20,19,0,0,0,0}, // E->T E'
    {19,3,-14,20,19,0,0,0}, // E'->+ T E'
    {19,3,-15,20,19,0,0,0}, // E'->- T E'
    {19,0,0,0,0,0,0,0}, // E'->ε
    {20,2,22,21,0,0,0,0}, // T->F T'
    {21,3,-16,22,21,0,0,0}, // T'->* F T'
    {21,3,-17,22,21,0,0,0}, // T'->/ F T'
    {21,0,0,0,0,0,0,0}, // T'->ε
    {22,1,-4,0,0,0,0,0}, // F->id
    {22,1,-10,0,0,0,0,0}, // F->num
    {22,3,-18,18,-19,0,0,0}, // F->( E )
    {23,3,-20,-4,-6,0,0,0}, // L->leer id ;
    {24,3,-21,25,-6,0,0,0}, // O->visua W ;
    {25,2,-4,26,0,0,0,0}, // W->id W'
    {25,2,-10,26,0,0,0,0}, // W->num W'
    {25,2,-11,26,0,0,0,0}, // W->cad W'
    {26,2,-13,25,0,0,0,0}, // W'->, W
    {26,0,0,0,0,0,0,0} // W'->ε
};
private int[,] _m={{1,1,0},
    {2,3,1},
    {2,12,2},
    {2,4,3},
    {2,20,3},
    {2,21,3},
    {3,4,4},
    {3,20,4},
    {3,21,4},
    {3,12,5},
    {4,3,6},
    {5,7,7},
    {5,8,7},
    {5,9,7},
    {6,7,8},
    {6,8,8},
    {6,9,8},
    {6,12,9},
    {6,4,9},
    {6,20,9},
    {6,21,9},
    {7,7,10},
    {7,8,11},
    {7,9,12},
    {8,10,13},
    {8,11,14},
    {9,12,15},
    {10,7,16},
    {10,8,16},
    {10,9,16},
    {11,7,17},
    {11,8,17},
    {11,9,17},

```

```

        {11, 4, 18},
        {11, 20, 18},
        {11, 21, 18},
        {12, 4, 19},
        {13, 13, 20},
        {13, 6, 21},
        {14, 4, 22},
        {14, 20, 23},
        {14, 21, 24},
        {15, 4, 25},
        {15, 20, 26},
        {15, 21, 27},
        {15, 2, 28},
        {16, 4, 29},
        {17, 4, 30},
        {17, 10, 30},
        {17, 18, 30},
        {17, 11, 31},
        {18, 4, 32},
        {18, 10, 32},
        {18, 18, 32},
        {19, 14, 33},
        {19, 15, 34},
        {19, 6, 35},
        {19, 19, 35},
        {20, 4, 36},
        {20, 10, 36},
        {20, 18, 36},
        {21, 16, 37},
        {21, 17, 38},
        {21, 14, 39},
        {21, 15, 39},
        {21, 6, 39},
        {21, 19, 39},
        {22, 4, 40},
        {22, 10, 41},
        {22, 18, 42},
        {23, 20, 43},
        {24, 21, 44},
        {25, 4, 45},
        {25, 10, 46},
        {25, 11, 47},
        {26, 13, 48},
        {26, 6, 49}
    };

private int _noVts;
private int _noVns;
private int _noProd;
private int _noEnt;
private int[] _di;
private int _noDis;

// Metodos

public SintDescNRP() // Constructor -----
{
    _pila=new Pila();
    _noVts = _vts.Length;
    _noVns = _vns.Length;
    _noProd = 50;
    _noEnt = 77;
    _di=new int[NODIS];
    _noDis=0;
} // Fin del Constructor -----

public void Inicia() // Constructor -----
{
    _pila.Inicia();
    _noDis=0;
}

public int Analiza(Lexico oAnaLex)
{

```

```

SimbGram x=new SimbGram("");
string a;
int noProd;
_pila.Inicia();
_pila.Push(new SimbGram("$"));
_pila.Push(new SimbGram(_vns[1]));
oAnaLex.Anade("$", "$");
int ae = 0;
do
{
    x = _pila.Tope;
    a = oAnaLex.Token[ae];
    if (EsTerminal(x.Elem))
        if (x.Elem.Equals(a))
        {
            _pila.Pop();
            ae++;
        }
        else
            return 1;
    else
    {
        if ((noProd = BusqProd(x.Elem, a)) >= 0)
        {
            _pila.Pop();
            MeterYes(noProd);
            _di[_noDis++] = noProd;
        }
        else
            return 2;
    }
} while(!x.Elem.Equals("$"));
return 0;
}

public bool EsTerminal(string x)
{
    for(int i=1;i<_noVts;i++)
        if(_vts[i]==x)
            return true;
    return false;
}

public int BusqProd(string x, string a)
{
    int indiceX=0;
    for(int i=1;i<_noVns;i++)
        if (_vns[i]==x)
        {
            indiceX = i;
            break;
        }
    int indiceA=0;
    for(int i=1;i<_noVts;i++)
        if (_vts[i]==a)
        {
            indiceA = i;
            break;
        }
    for (int i = 0; i < _noEnt; i++)
        if (indiceX == _m[i, 0] && indiceA == _m[i, 1])
            return _m[i, 2];
    return -1;
}

public void MeterYes(int noProd)
{
    int noYes = _prod[noProd, 1];
    for (int i = 1; i <= noYes; i++)
        if (_prod[noProd, noYes + 2 - i] < 0)
            _pila.Push(new SimbGram(_vts[-_prod[noProd, noYes + 2 - i]]));
        else
            _pila.Push(new SimbGram(_vns[_prod[noProd, noYes + 2 - i]]));
}

```

```

public string[] Vts
{
    get { return _vts; }
}

public string[] Vns
{
    get { return _vns;}
}

public int[,] M
{
    get { return _m; }
}

public int NoDis
{
    get { return _noDis; }
}

public int [] Di
{
    get { return _di; }
}

public int IndiceVn(string vn)
{
    for (int i = 1; i < _noVns; i++)
        if (_vns[i] == vn)
            return i;
    return 0;
}

public int IndiceVt(string vt)
{
    for (int i = 1; i < _noVts; i++)
        if (_vts[i]==vt)
            return i;
    return 0;
}

public int NoProd
{
    get { return _noProd; }
}

} // fin de la clase SintDescNRP

```

Entonces lo primero que hacemos es agregar la clase `SintDescNRP` al proyecto, con el código generado incluido según se indica en la figura #6.1.

También debemos añadir las clases `Pila` y `SimbGram` que se toman del RD-NRP, figura #6.2.

```

class SimbGram
{
    string _elem;

    public SimbGram(string sValor)
    {
        _elem = sValor;
    }

    public string Elem
    {
        get { return _elem; }
    }
} // Fin de la clase SimbGram

```

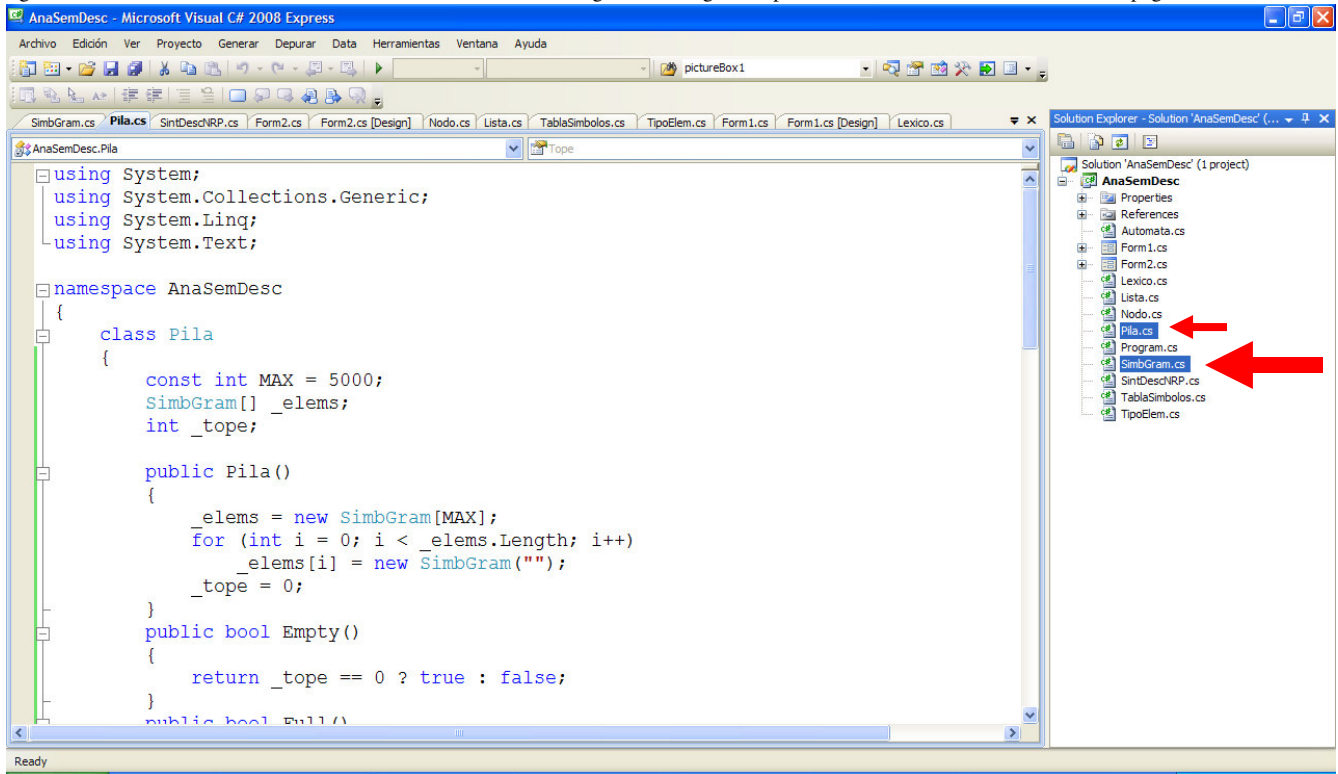



Fig. No. 6.2 Adición de la Clases Pila y SimbGram al proyecto.

Ahora cambiemos la leyenda del botón `button1` a ANALISIS SINTACTICO, e introduzcamos el código que listamos a continuación dentro del método `button1_Click()`.

```
private void button1_Click(object sender, EventArgs e)
{
    oAnaLex.Inicia();
    oTablaSimb.Inicia();
    bool exito = oAnaLex.Analiza(textBox1.Text, oTablaSimb);
    if (exito)
    {
        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];
        }
        label2.Text = "EXITO EN EL ANALISIS LEXICO";
        oAnaSintDesc.Inicia();
        string error;
        error = oAnaSintDesc.Analiza(oAnaLex, oTablaSimb);
        if (error == "OK")
            label2.Text = "ANALISIS SINTACTICO EXITOSO";
        else
            label2.Text = error; //se muestra el tipo de error encontrado
    }
    else
        label2.Text = "ERROR LEXICO... CARACTER NO RECONOCIDO";
}
}
```

Antes de ejecutar la aplicación para intentar el análisis sintáctico, es necesario modificar el código del método `Analiza()` que produce el RD-NRP. Veamos la línea de código que efectúa el análisis sintáctico :

```
error = oAnaSintDesc.Analiza(oAnaLex, oTablaSimb);
```

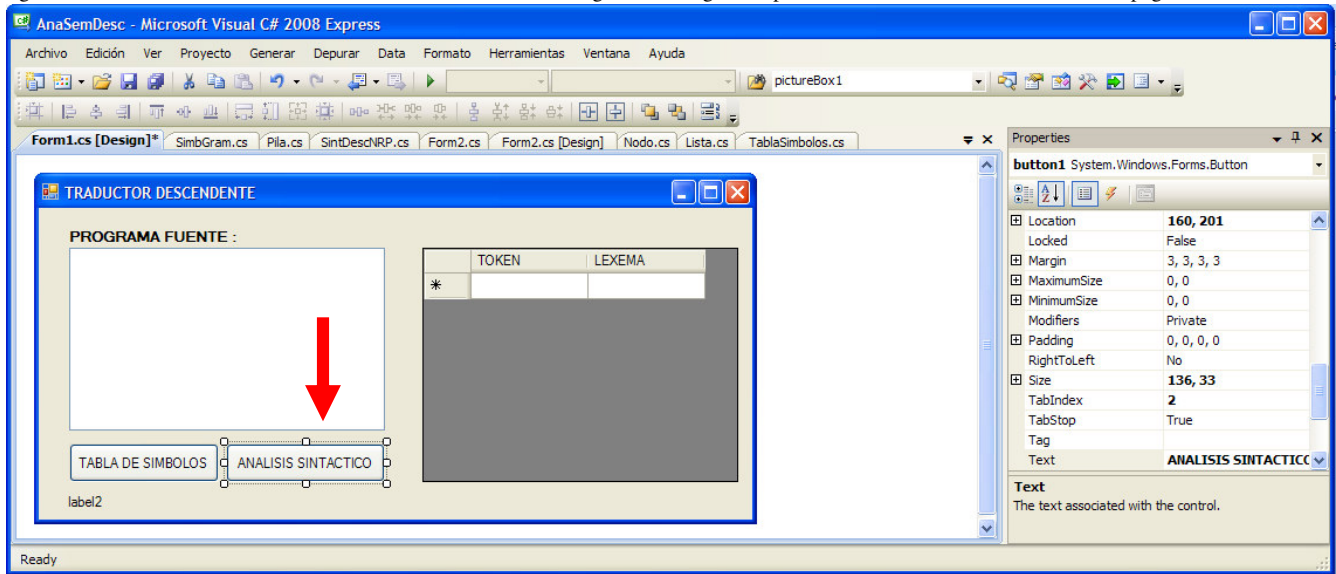


Fig. No. 6.3 Cambio de la propiedad Text del button1.

Vemos que el tipo de retorno del método Analiza() es un **string**. El RD-NRP produce el método Analiza() de manera que retorna un tipo **int**. Así que manos a la obra, cambiemos el tipo de retorno a **string**. También incluiremos al parámetro oTablaSimb según la figura #6.4.

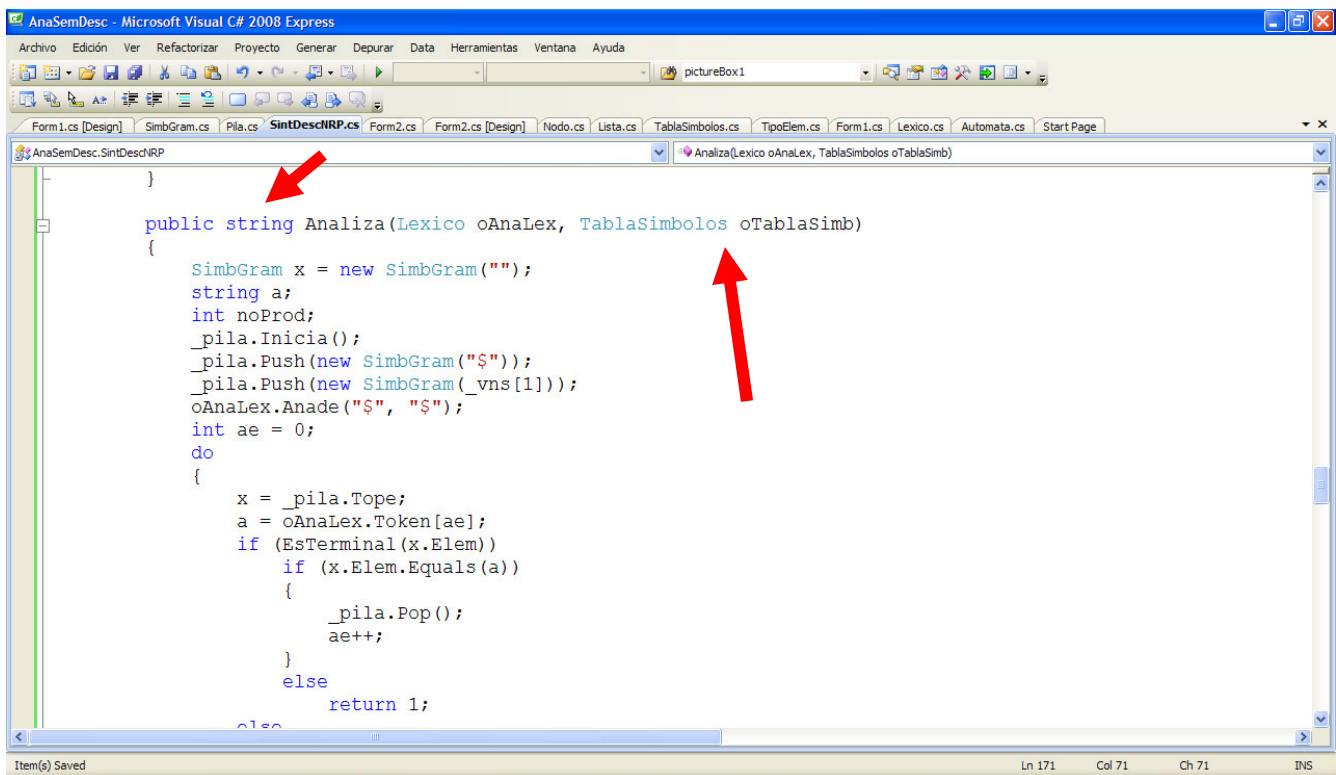


Fig. No. 6.4 Cambios en Analiza() de la clase SintDescNRP : tipo de retorno a **string**, y un nuevo parámetro oTablaSimb.

Ahora cambiemos los valores tipo **int** que retornan las sentencias return, a valores tipo **string** que correspondan a la lógica del método : mensajes de error para el caso de un error de sintaxis, y la cadena "OK" para el caso de éxito en el reconocimiento. Recordemos que estamos modificando el método Analiza() de la clase SintDescNRP.

El método Analiza() después de efectuar los cambios es :

```

public string Analiza(Lexico oAnaLex, TablaSimbolos oTablaSimb)
{
    SimbGram x = new SimbGram("");
    string a;
    int noProd;
    _pila.Inicia();
    _pila.Push(new SimbGram("$"));
    _pila.Push(new SimbGram(_vns[1]));
    oAnaLex.Anade("$", "$");
    int ae = 0;
    do
    {
        x = _pila.Tope;
        a = oAnaLex.Token[ae];
        if (EsTerminal(x.Elem))
            if (x.Elem.Equals(a))
            {
                _pila.Pop();
                ae++;
            }
            else
                return "ERROR DE SINTAXIS ... POR TERMINAL";
        else
        {
            if ((noProd = BusqProd(x.Elem, a)) >= 0)
            {
                _pila.Pop();
                MeterYes(noProd);
                _di[_noDis++] = noProd;
            }
            else
                return "ERROR DE SINTAXIS ... NO SE ENCONTRO LA PRODUCCION";
        }
    } while (!x.Elem.Equals("$"));
    return "OK";
}

```

Seguimos con la definición del objeto oAnaSintDesc en *Form1.cs* de la aplicación, figura #6.5..

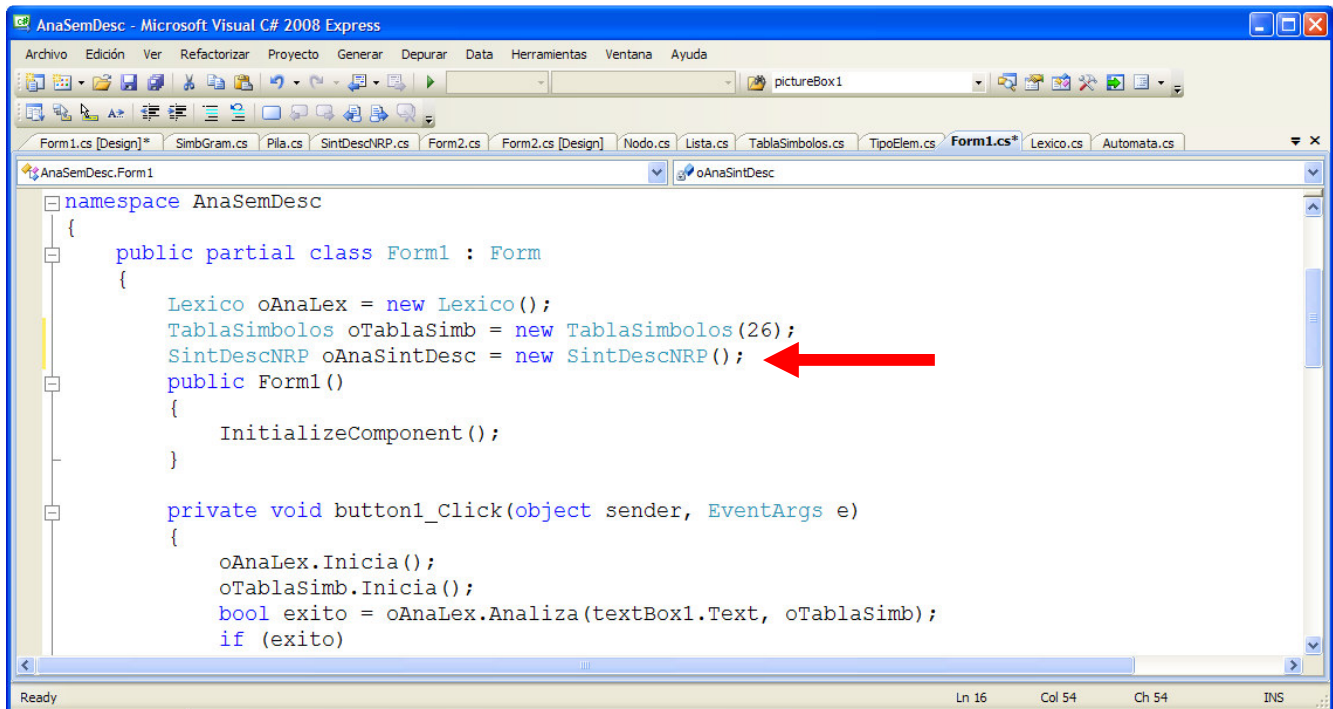


Fig. No. 6.5 Definición del objeto oAnaSintDesc.

Nos falta agregar el método Anade () en la clase Lexico. Este método es llamado dentro del método Analiza () del reconocedor descendente para añadir el símbolo \$ en la cadena de entrada. Ver figura #6.6.

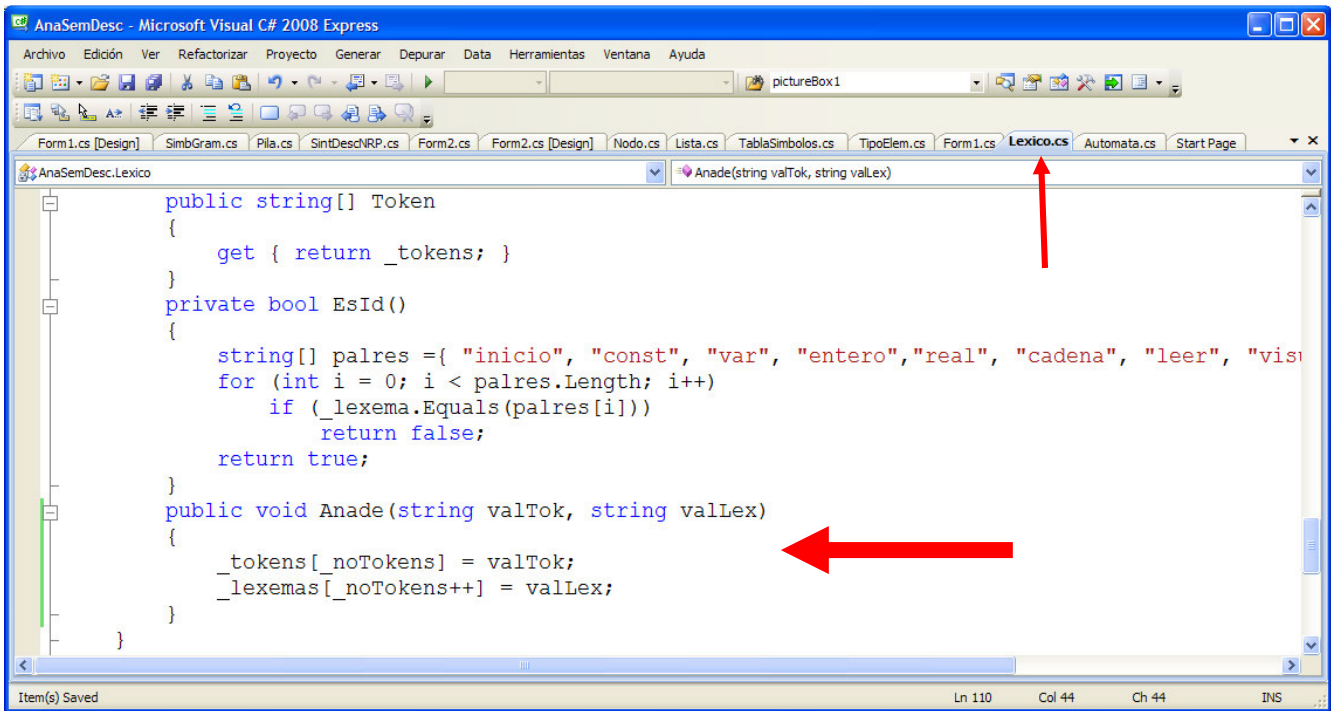


Fig. No. 6.6 Definición del método Anade () en la clase Lexico.

Compilamos y ejecutamos la aplicación, debemos tener algo parecido a lo mostrado en la figura 6.7.

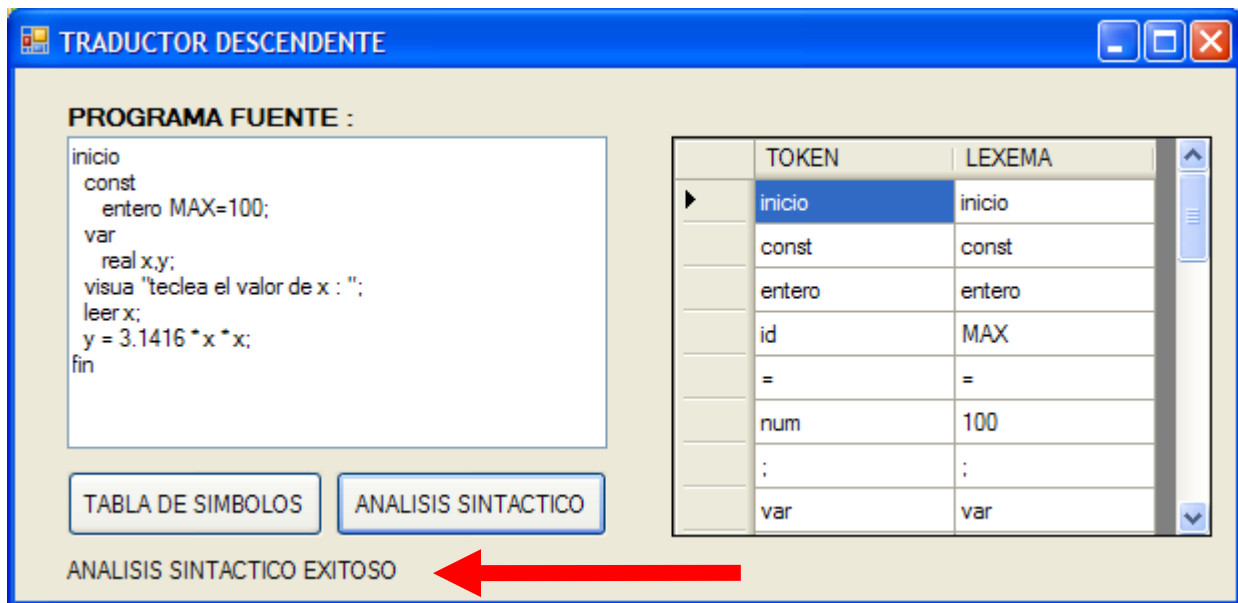


Fig. No. 6.7 Análisis sintáctico con éxito.

En las figuras 6.8 y 6.9 se muestran ejecuciones de la aplicación para un error léxico y un error de sintaxis.

Estamos listos para empezar la construcción del traductor descendente.

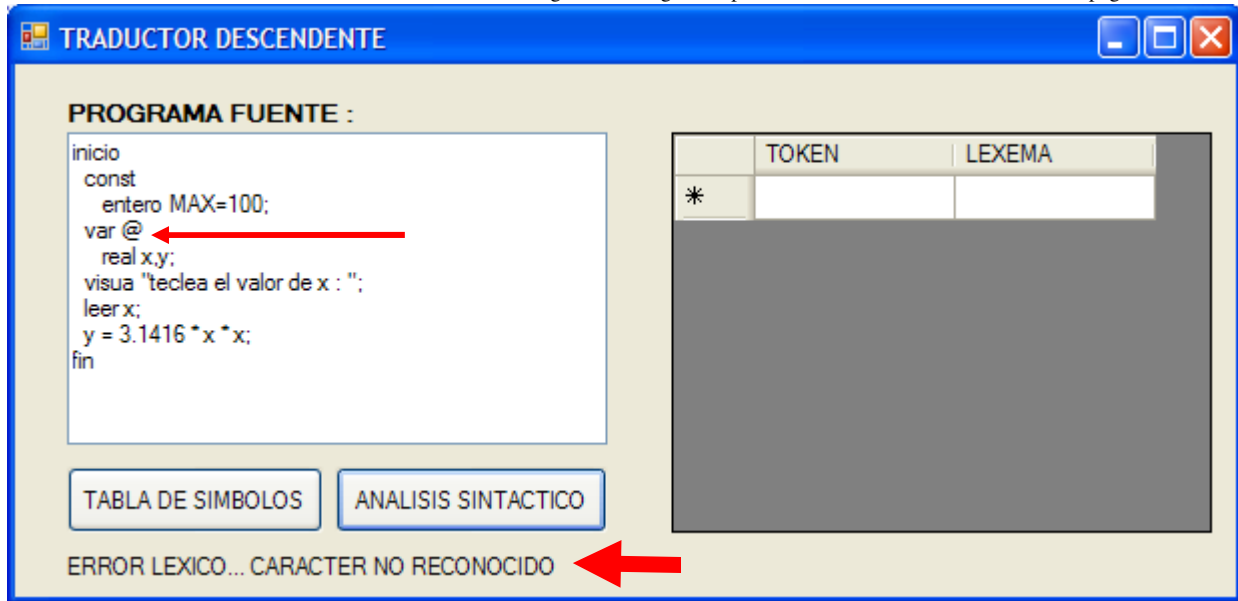


Fig. No. 6.8 Error léxico.

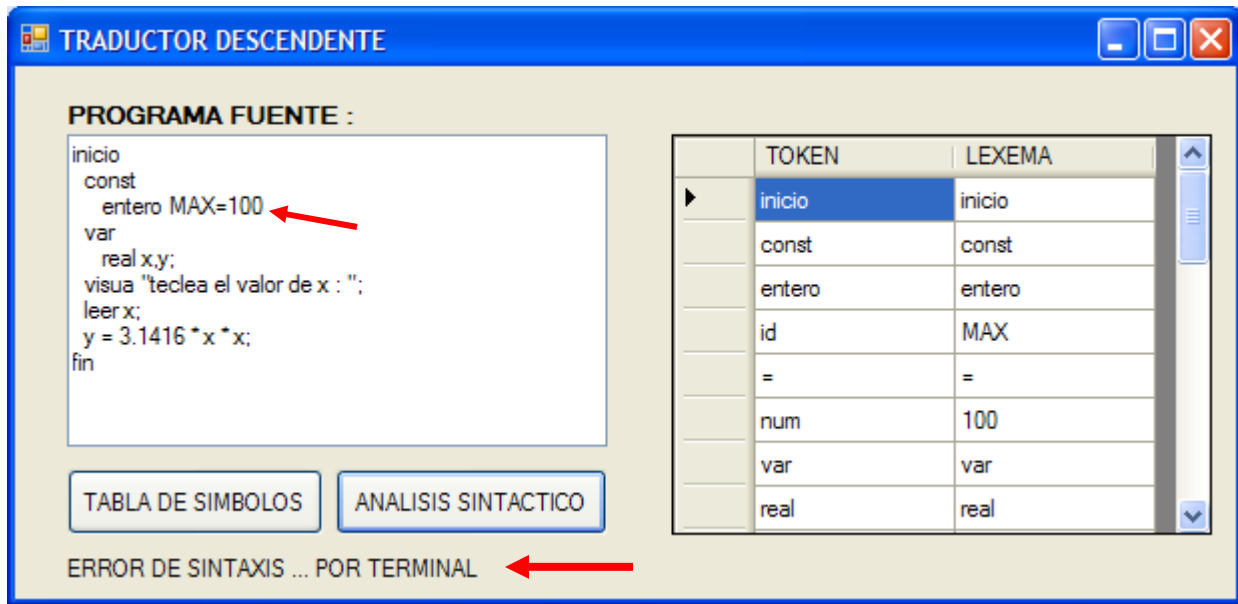


Fig. No. 6.9 Error de sintaxis, falta del caracter ;.

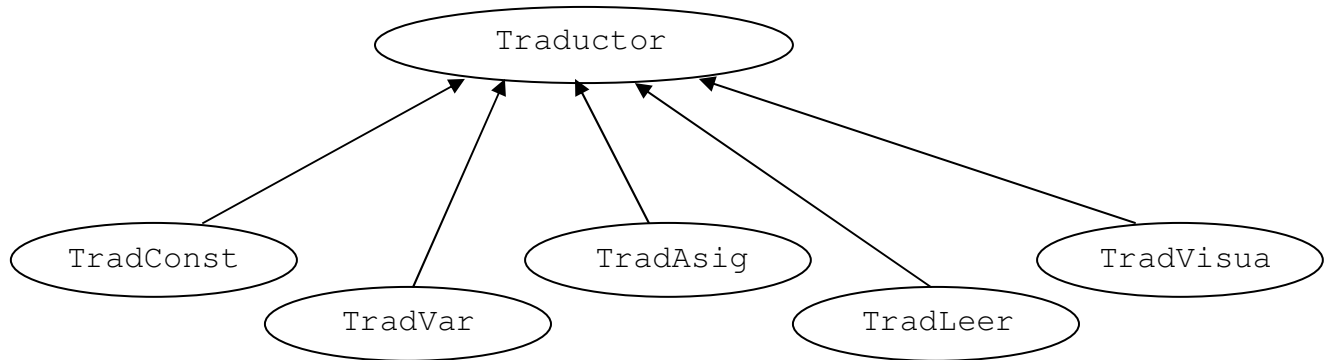
7 Traducciones a realizar.

Las traducciones y revisiones que implementaremos en el traductor descendente son las siguientes :

- Almacenar el tipo de dato, número de bytes y el valor, de las constantes declaradas en la tabla de símbolos.
- Almacenar el tipo de dato y el número de bytes de las variables declaradas, en la tabla de símbolos.
- Indicar error si una constante o una variable está duplicada en su declaración.
- Indicar error si se trata de asignar a una constante el valor de una expresión dentro de una sentencia de asignación.
- Indicar error si una variable se utiliza en una expresión, sin haber sido declarada previamente.
- Revisar el tipo en las expresiones –sentencias- de asignación. Que correspondan los tipos del operando izquierdo y del derecho.

8 Construcción del traductor descendente.

Iniciaremos con la definición del árbol de herencia necesario para construir el conjunto de traductores que forman al analizador semántico –traductor descendente-.



Notemos que la clase base es la clase Traductor y las clases derivadas son las clases : TradConst, TradVar, TradAsig, TradLeer y TradVisua.

Escribamos en C# el árbol de herencia mostrado agregando las 6 clases al proyecto sin incluir atributos ni métodos, solamente indicaremos la herencia. En la figura #8.1 se muestra una de las clases : TradConst.

```

class Traductor
{
}

class TradConst : Traductor
{
}

class TradVar : Traductor
{
}
  
```

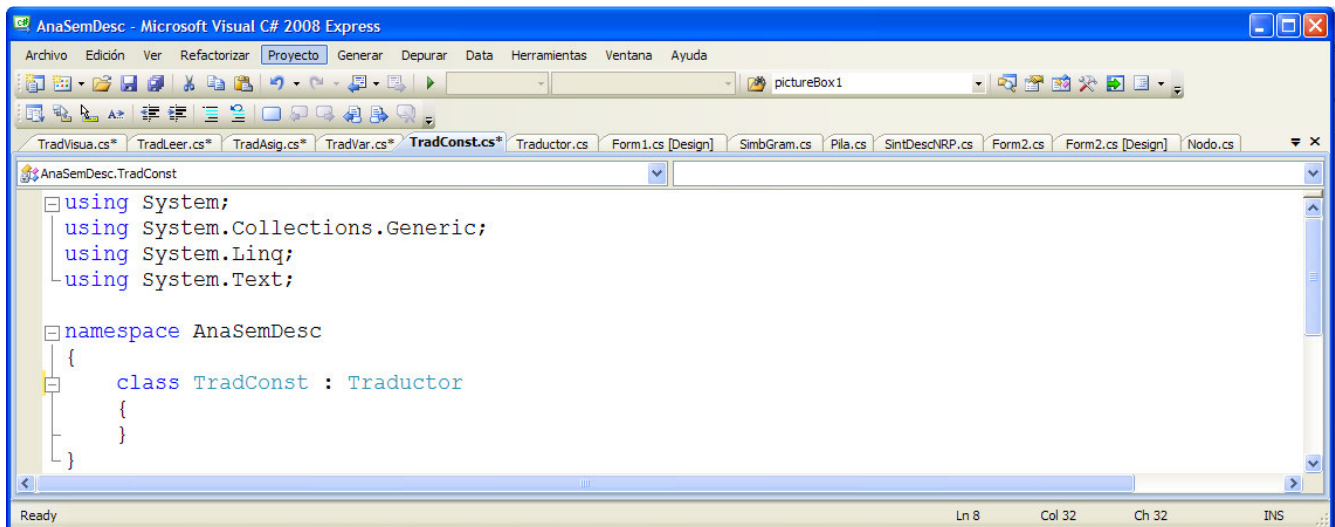


Fig. No. 8.1 Clase derivada TradConst.

```

class TradAsig : Traductor
{
}
  
```

```
class TradLeer : Traductor
{
}

class TradVisua : Traductor
{
}
```

Ahora definamos los 5 objetos traductores dentro de la clase SintDescNRP debido a que ellos actuarán –traducirán- dentro del método Analiza() de dicha clase, figura #8.2.

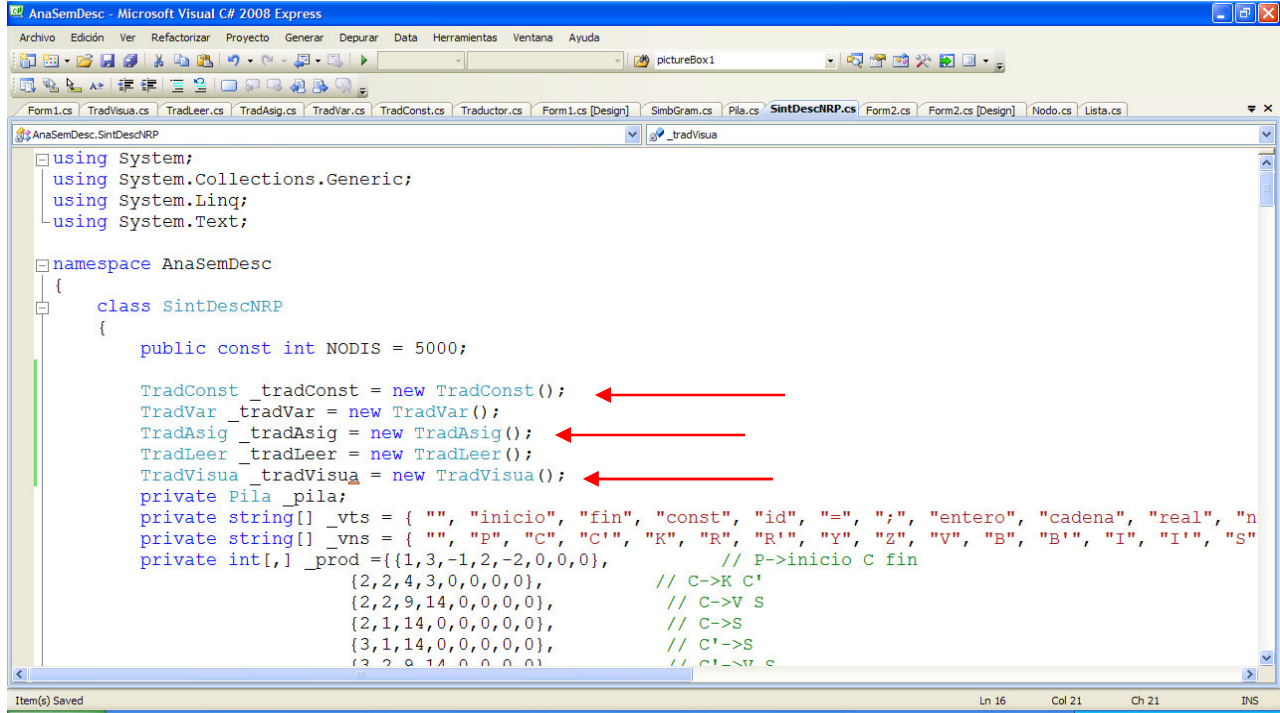


Fig. No. 8.2 Definición de los 5 traductores _tradCons, _tradVar, _tradAsig, _tradLeer, _tradVisua.

Para las traducciones y revisiones que efectuaremos, los traductores requieren de al menos uno de los parámetros listados a continuación :

- noProd, número cde la producción que se aplica en un determinado instante del reconocimiento.
- ae, el índice del token que actualmente está viendo el analizador sintáctico, de la sentencia que se está reconociendo.
- oAnaSintDesc, el propio analizador sintáctico.
- oAnaLex, el objeto analizador léxico.
- oTablaSimb, la tabla de símbolos.

Vamos a definir el método Traducir() para cada una de las clases ahora que sabemos los parámetros que se requieren en la llamada a dicho método. El tipo de retorno será un **string** cuyo significado es el mensaje de error o de éxito del objeto traductor que está efectuando la traducción.

Definamos pues el método Traducir() con su cuerpo vacío en cada una de las clases, cuidando de hacerla **virtual** en la clase base Traductor, y hacer la anulación –override- en las clases derivadas.

```
class Traductor
{
    protected string[] _resultTraduccion = { "EXITO" };
    virtual public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
    {
        return "";
    }
}
```

```
class TradConst : Traductor
{
    override public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
    {
        return _resulTraduccion[0];
    }
}

class TradVar : Traductor
{
    override public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
    {
        return _resulTraduccion[0];
    }
}

class TradAsig : Traductor
{
    override public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
    {
        return _resulTraduccion[0];
    }
}

class TradLeer : Traductor
{
    override public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
    {
        return _resulTraduccion[0];
    }
}

class TradVisua : Traductor
{
    override public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
    {
        return _resulTraduccion[0];
    }
}
```

La figura #8.3 muestra la clase base Traductor con las modificaciones hechas, y la figura #8.4 le corresponde mostrar a la clase TradConst con su método Traducir() indicado con la palabra reservada **override**.

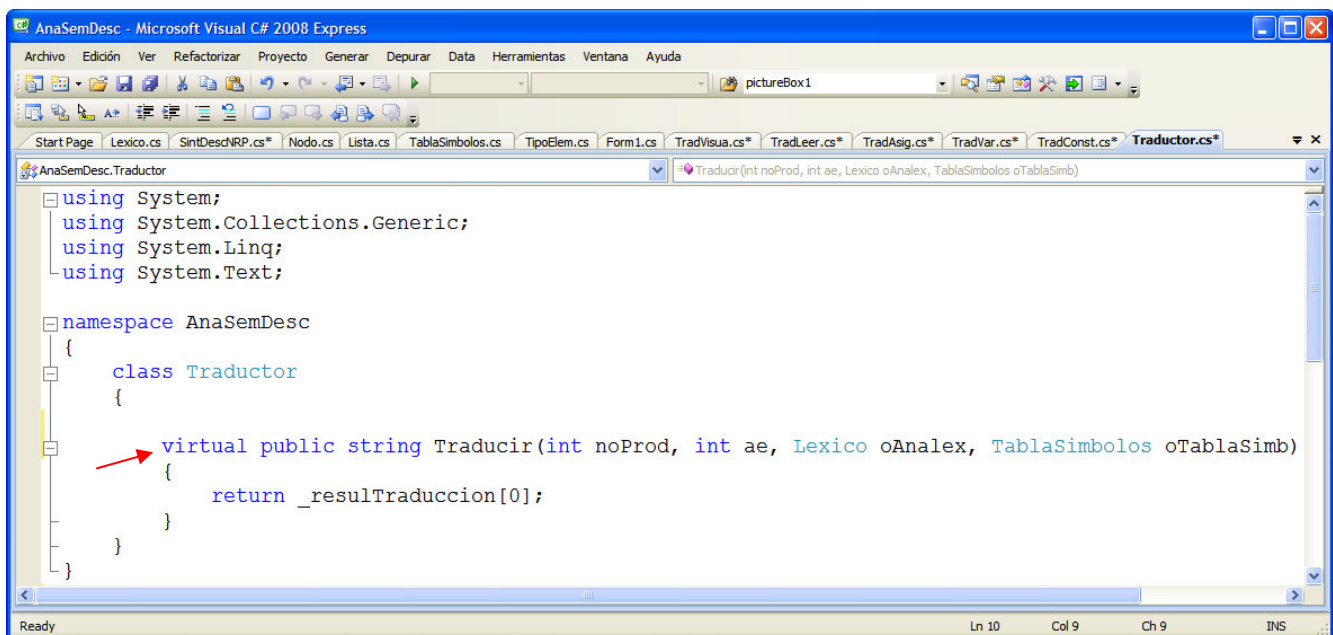


Fig. No. 8.3 Clase Traductor con su atributo protected y su método virtual.

Observamos que por ahora retornamos siempre el mensaje “ÉXITO” en cada una de las implementaciones del método Traducir(), definido en cada una de las 6 clases.

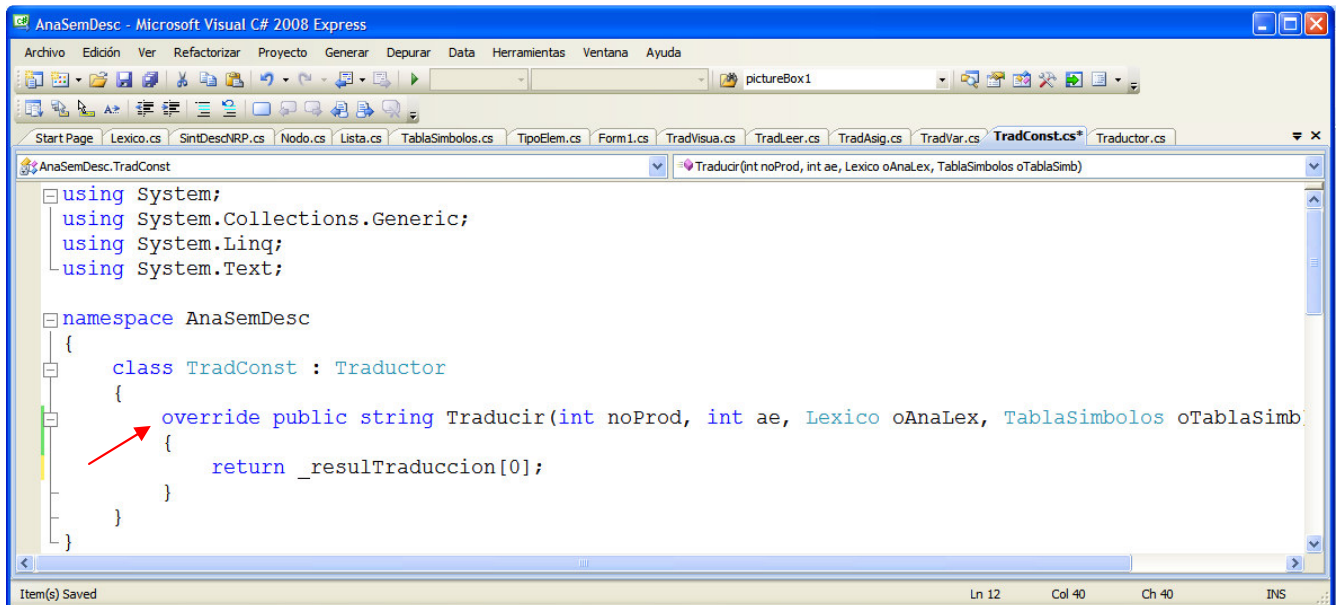


Fig. No. 8.4 Clase TradConst con su método Traducir() marcado como **override**.

Cambiamos ahora la leyenda del botón *button1* de nuestra aplicación a “ANALISIS SEMANTICO” para indicar el hecho de que se realizará también el análisis semántico además del análisis léxico y el análisis sintáctico.

La ejecución de la aplicación nos dice si hay ausencia de errores al teclear los cambios realizados hasta este momento, figura #8.5.

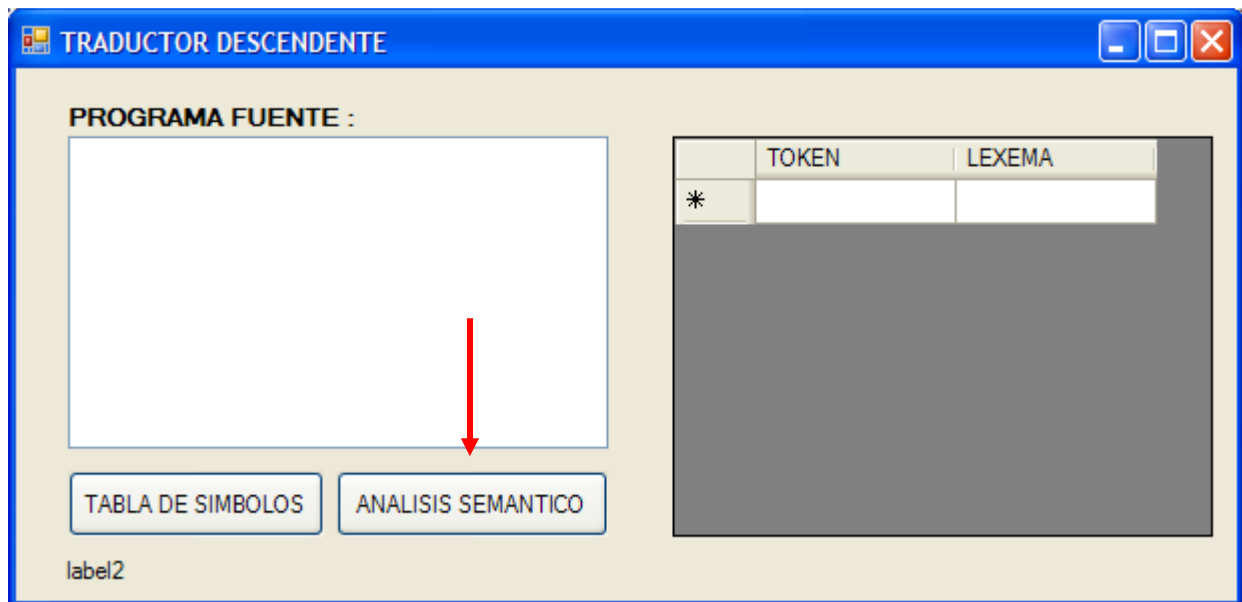


Fig. No. 8.5 Aplicación en ejecución con los cambios efectuados.

9 Traducción (a) : ALMACENAR EL TIPO DE DATO, NÚMERO DE BYTES Y EL VALOR DE LAS CONSTANTES DECLARADAS, EN LOS ATRIBUTOS RESPECTIVOS DE LA TABLA DE SÍMBOLOS.

Recordemos la clase TipoElem utilizada para representar la información de cada token que es almacenado en la tabla de símbolos :

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

La definición dirigida por sintaxis que escribamos deberá afectar los atributos _clase, _valor, _tipo y _noBytes del *identificador* declarado como constante, y cuyo atributo _nombre corresponda al lexema del identificador –constante-. Usemos el RD-NRP para mostrar de manera gráfica una ayuda que nos permita entender de mejor manera, lo que ocurre cuando el analizador sintáctico se encuentra reconociendo la declaración de una constante, figura #9.1.

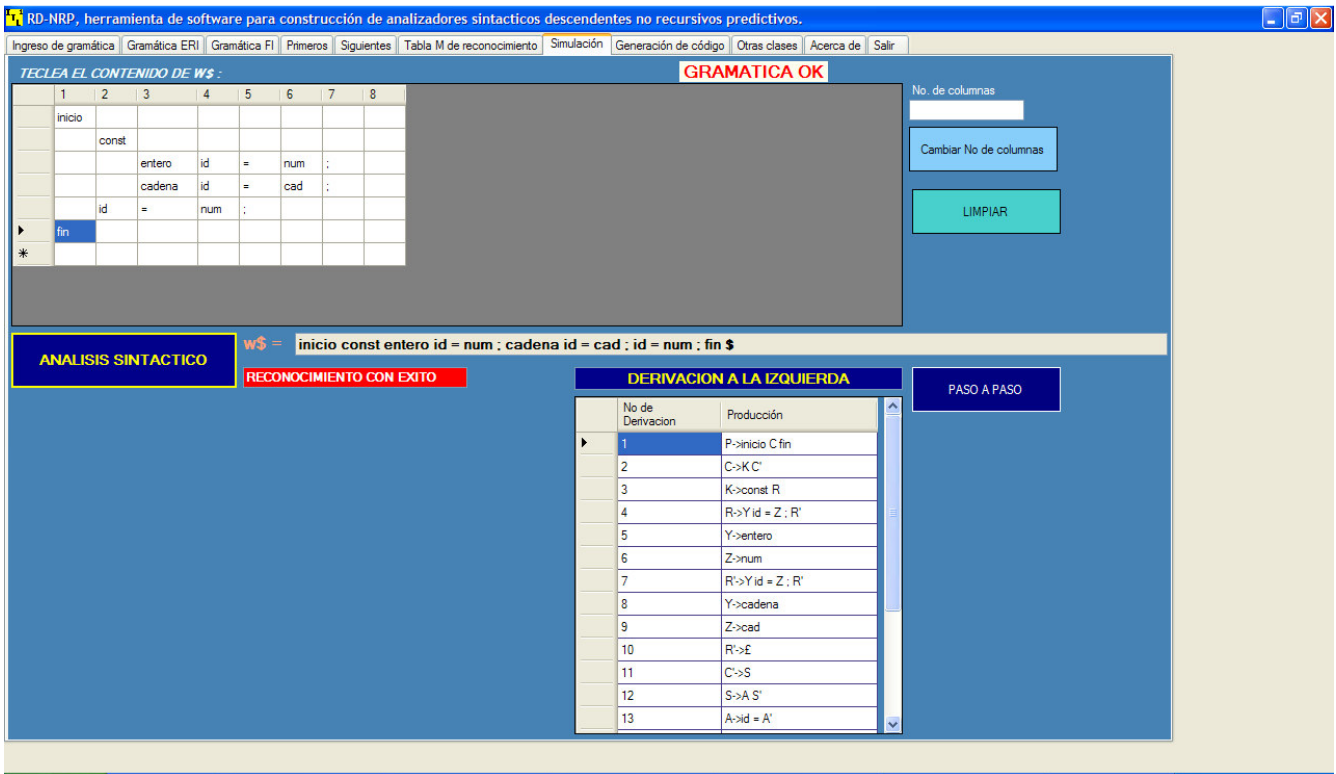


Fig. No. 9.1 Simulación para la entrada mostrada usando el RD-NRP.

La entrada mostrada en la figura #9.1 es la que recibe el analizador sintáctico por parte del analizador léxico, para un programa fuente –sin lógica, con buena sintáxis pero con error semántico que por ahora no nos va a interesar- de ejemplo :

```
inicio
const
    entero MAX = 100;
    cadena MENSAJE = "ERROR ... ";
    x = 0;
fin
```

Para este ejemplo, el traductor de constantes deberá de hacer las asignaciones :


```

_clase = "constante"
_tipo = entero
_valor = 100
_noBytes = 4 // suponemos que un entero se almacena en 4 bytes
    
```

¿Cuándo entra en acción el objeto `_tradCons`? Para contestar la pregunta nos ayudamos precisamente del RD-NRP según se indica en la figura #9.2.

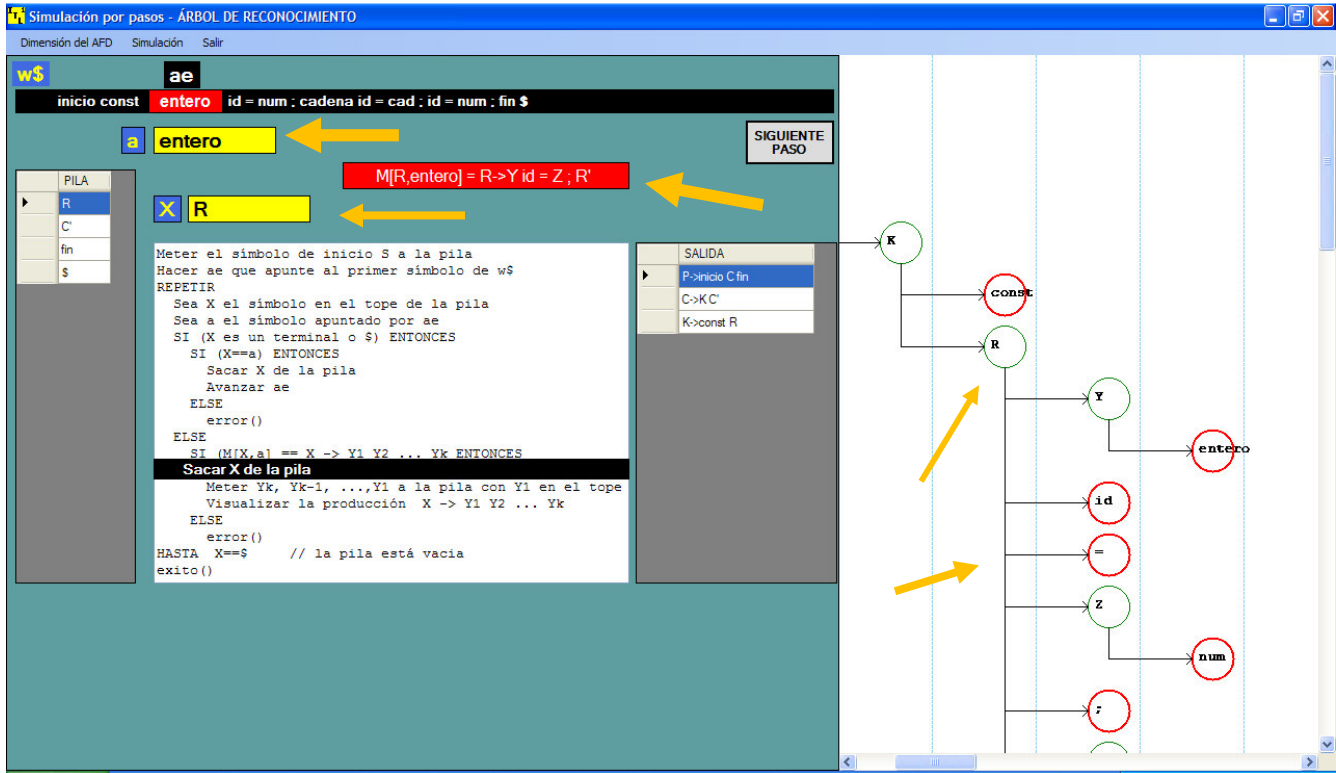


Fig. No. 9.2 Uso de la producción $R \rightarrow Y \text{ id} = Z ; R'$ para reconocer la constante MAX del ejemplo.

Observamos de la figura 9.2 que la producción $R \rightarrow Y \text{ id} = Z ; R'$ es usada cuando en la entrada el analizador está viendo al token `entero` y la pila tiene en su cima al símbolo no terminal `R`. Esta producción es la utilizada cuando el analizador está reconociendo la declaración de la constante `MAX`.

Podemos seguir paso a paso el reconocimiento usando el botón con la leyenda `SIGUIENTE PASO`, pero mejor nos ayudaremos del árbol de reconocimiento que se observa en la figura #9.3. En esta figura se muestra el árbol de reconocimiento para la declaración de la constante `MENSAJE`, donde es utilizada la producción $R' \rightarrow Y \text{ id} = Z ; R'$.

Ahora vayamos al RD-NRP en la pestaña de generación de código para encontrar el número de las producciones que hemos identificado, figura #9.4. Encontramos que el número de las producciones son 7 y 8 respectivamente.

$R \rightarrow Y \text{ id} = Z ; R'$... `noProd = 7`

$R' \rightarrow Y \text{ id} = Z ; R'$... `noProd = 8`

Bien, ya estamos listos para escribir las reglas semánticas utilizando la definición dirigida por sintaxis y los esquemas de traducción.

Recordemos que una definición dirigida por sintaxis sirve para especificar las reglas semánticas a un alto nivel, y los esquemas de traducción se emplean para describir las reglas semánticas a un nivel bajo, es decir, a nivel de programación podríamos decirlos así aunque el nivel es decidido por el diseñador –analista–.

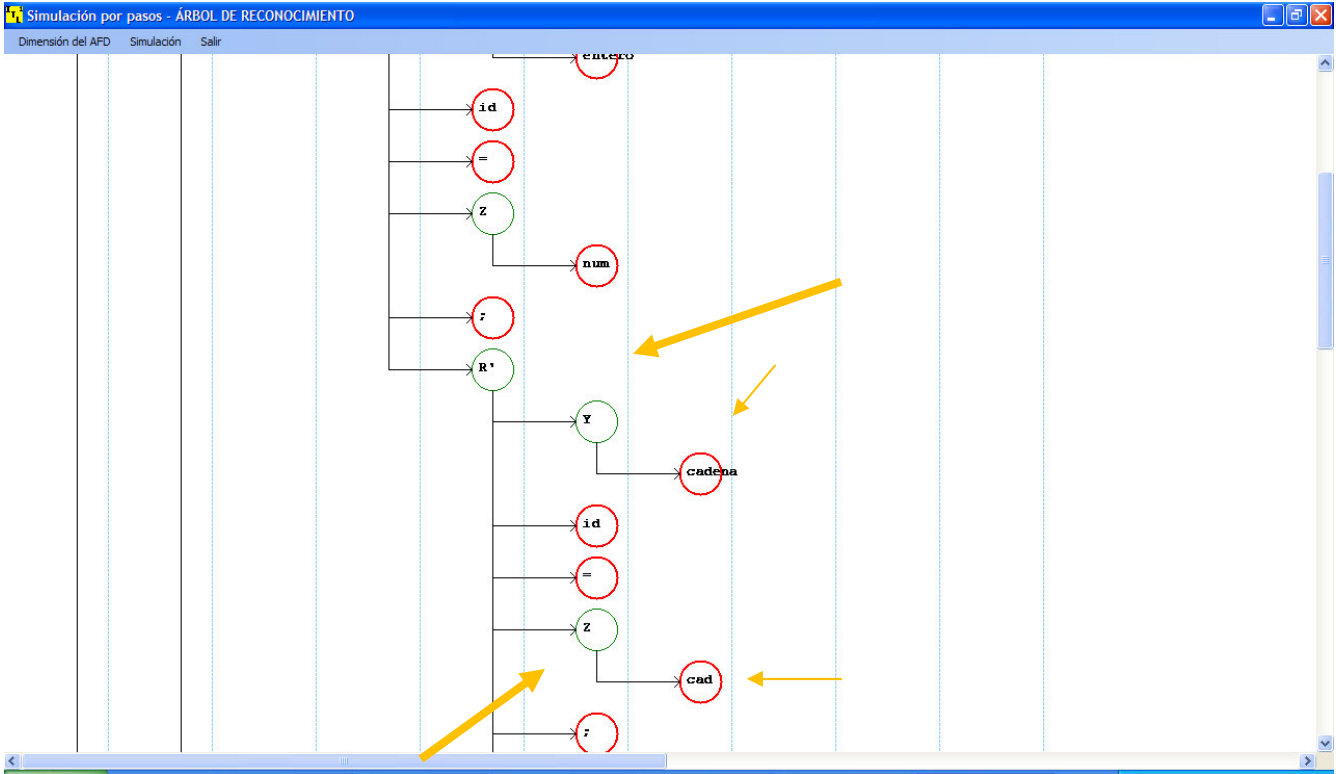


Fig. No. 9.3 Uso de la producción $R' \rightarrow Y \text{ id } = Z ; R'$ para reconocer la declaración de la constante cuyo identificador es MENSAJE.

Estado de la gramática: OK NoProd=50, NoVns=26, NoVts=21

No.Prod	MI	NoYes	Y1	Y2	Y3	Y4	Y5	Y6
0	P	3	inicio	C	fin			
1	C	2	K	C'				
2	C	2	V	S				
3	C	1	S					
4	C'	1	S					
5	C'	2	V	S				
6	K	2	const	R				
7	R	6	Y	id	=	Z	:	R'
8	R'	6	Y	id	=	Z	:	R'
9	R'	0	E					
10	Y	1	entero					
11	Y	1	cadena					

VTS: inicio, fin, const, id, =, :, entero, cadena

VNS: P, C, C', K, R, R', Y, Z

PRIMEROS:

Vn	PRIMEROS	SIGUIENTES
P	inicio	\$
C	const var id leer visua	fin
C'	var id leer visua	fin
K	const	var id leer visua
R	entero cadena real	var id leer visua

```

class SintDescNRP
{
    public const int NODIS=5000;

    private Pila _pila;

    private int _noVts;
    private int _noVns;
    private int _noProd;
    private int _noEnt;
    private int[] _di;
    private int _noDis;

    // Metodos

    public SintDescNRP() // Constructor -----
    {
        _pila=new Pila();
        _noVts = _vts.Length;
        _noVns = _vns.Length;
        _noProd = ?noProd;
        _noEnt = ?noEnt;
        _di=new int[NODIS];
        _noDis=0;
    } // Fin del Constructor -----

    public void Inicia() // Constructor -----
    {
        _pila.Inicia();
        _noDis=0;
    }

    public int Analiza(Lexico oAnaLex)
    {
        SimbGram x=new SimbGram("");
        string a;
        int noProd;
    }
}
    
```

Fig. No. 9.4 Visualización del número de producción usando el RD-NRP.

DEFINICIÓN DIRIGIDA POR SINTAXIS

noProd	Producción	Reglas semánticas
7	R -> Y id = Z ; R'	id.clase = "constante" id.tipo = Y.tipo id.valor = Z.valor Y.noBytes = Z.valor *2 // para Y.tipo = "cadena" id.noBytes = Y.noBytes
8	R' -> Y id = Z ; R'	id.clase = "constante" id.tipo = Y.tipo id.valor = Z.valor Y.noBytes = Z.valor *2 // para Y.tipo = "cadena" id.noBytes = Y.noBytes
10	Y->entero	Y.tipo = "entero" Y.noBytes = 4
11	Y->cadena	Y.tipo = "cadena"
12	Y->real	Y.tipo = "real" Y.noBytes = 8
13	Z->num	Z.valor = num.lexema
14	Z->cad	Z.valor = cad.lexema

La descripción de los símbolos gramaticales y sus atributos en cuanto a la definición dirigida por sintaxis mostrada es :

- id.clase se refiere si el id corresponde a una constante.
- id.tipo es el tipo de dato de la constante cuyo identificador es id.
- id.valor corresponde al valor de la constante cuyo identificador es id.
- id.noBytes es el número de bytes que se usan para almacenar la constante id.
- Y.tipo es el tipo de dato para el símbolo no terminal Y.
- Y.noBytes representa el número de bytes para el no terminal Y.
- Z.valor es el valor del no terminal Z.
- num.lexema es precisamente el correspondiente lexema al terminal num.
- cad.lexema representa el lexema del terminal cad.

Una vez hecha la definición dirigida por sintaxis se tienen las reglas semánticas definidas, de manera que la traducción que se está intentando es mas clara para el propio diseñador –programador-.

El paso siguiente corresponde a la definición del esquema de traducción para cada producción y sus reglas semánticas indicadas en la definición dirigida por sintaxis. Un esquema de traducción permite establecer las reglas semánticas implicadas en un nivel mas próximo a la implementación –codificación-.

Vamos a dividir el proceso de construcción del esquema de traducción, de forma que escribamos una a una cada regla semántica y su codificación en C#. Empecémos por la regla id.clase = "constante".

ESQUEMA DE TRADUCCION (1)

noProd	Producción	Reglas semánticas
7	R -> Y id = Z ; R'	p=oTablaSimb.EncuentraToken(Hash(id.lexema), id.lexema) p.Info.Clase = "constante"
8	R' -> Y id = Z ; R'	p=oTablaSimb.EncuentraToken(Hash(id.lexema), id.lexema) p.Info.Clase = "constante"

La regla `p=oTablaSimb.EncuentraToken(hash(id.lexema), id.lexema)` asigna a `p`, la referencia al nodo que contiene al lexema que corresponde al `id` que denota a la constante.

La asignación de constante se hace en la segunda regla `p.Info.Clase = "constante"` que accede a la propiedad `Clase` del atributo `_oInfo` del nodo que contiene al `id` buscado. La propiedad `Info` retorna al atributo `_oInfo` del nodo encontrado.

Otra observación es la manera en que se encuentra el índice de la lista enlazada de la tabla de símbolos. Mostramos el uso de un método al que denominamos `Hash()` que recibe de parámetro el lexema correspondiente al `id`. Realmente este método nos permite escribir la regla semántica dejando para después la manera en que obtendremos el índice de la lista.

Pongamos manos a la obra y vayamos a la clase `SintDescNRP` donde se encuentra el método `Analiza()`, y agreguemos la definición del objeto local `trad`, figura #9.5.

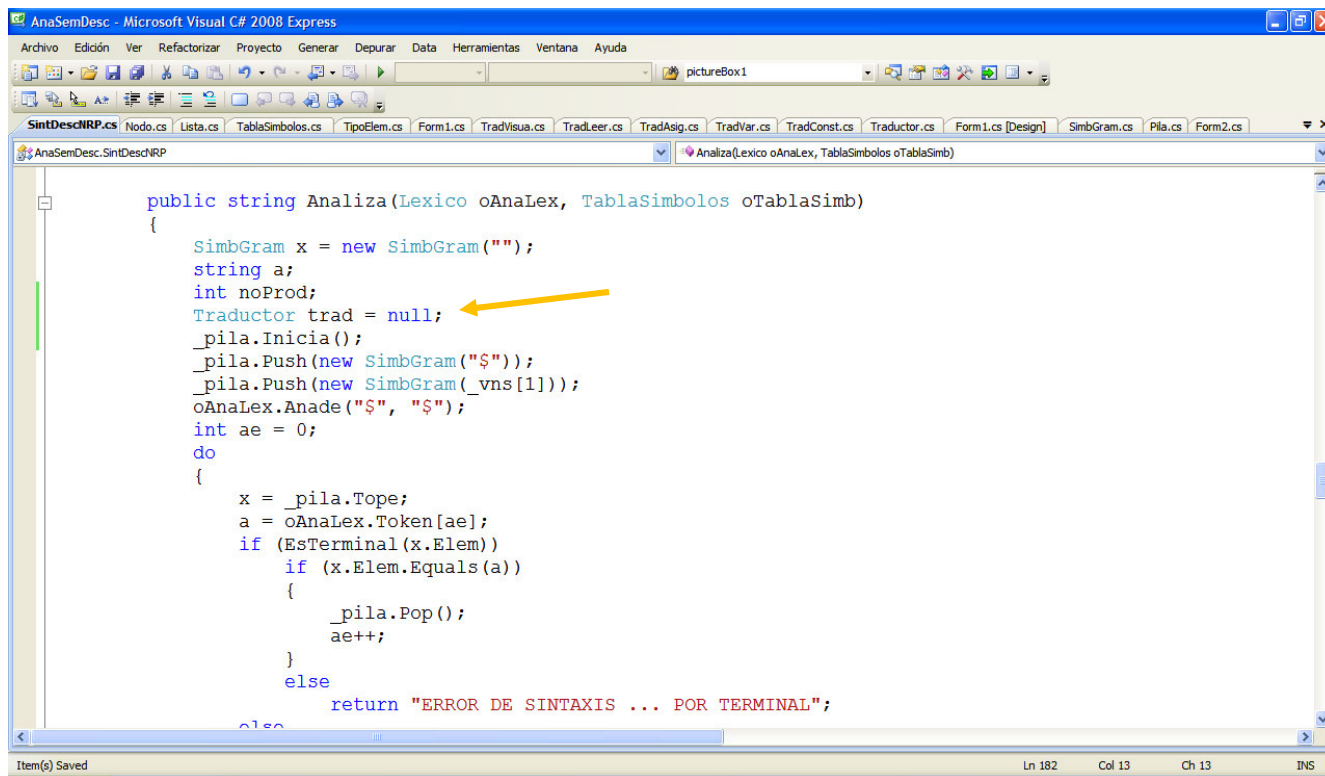


Fig. No. 9.5 Definición del objeto trad.

Luego añadimos el `switch()` que nos permitirá manejar el polimorfismo para utilizar el objeto traductor adecuado, dependiendo del número de producción. Agreguemos el código siguiente también dentro del método `Analiza()` de la clase `SintDescNRP`.

```
switch (noProd)
{
    case 7 :
    case 8 :
        trad = _tradConst;
        break;
    default :
        trad = new Traductor();
        break;
}
string resulTrad = trad.Traducir(noProd, ae, this, oAnaLex, oTablaSimb);
if (resulTrad != "EXITO")
    return resulTrad;
```

Hacemos uso del polimorfismo para enlazar en tiempos de ejecución al objeto traductor adecuado. En este caso lo asignamos al objeto traductor de constantes `_tradConst` para luego enviar el mensaje que incluye al método `Traducir()` definido en la clase `TradConst`. Claro que lo anterior se hace sólo en el caso en que el número de la producción es 7 o es 8.

En la figura #9.6 se muestra el lugar exacto en donde debemos insertar el código anteriormente mencionado.

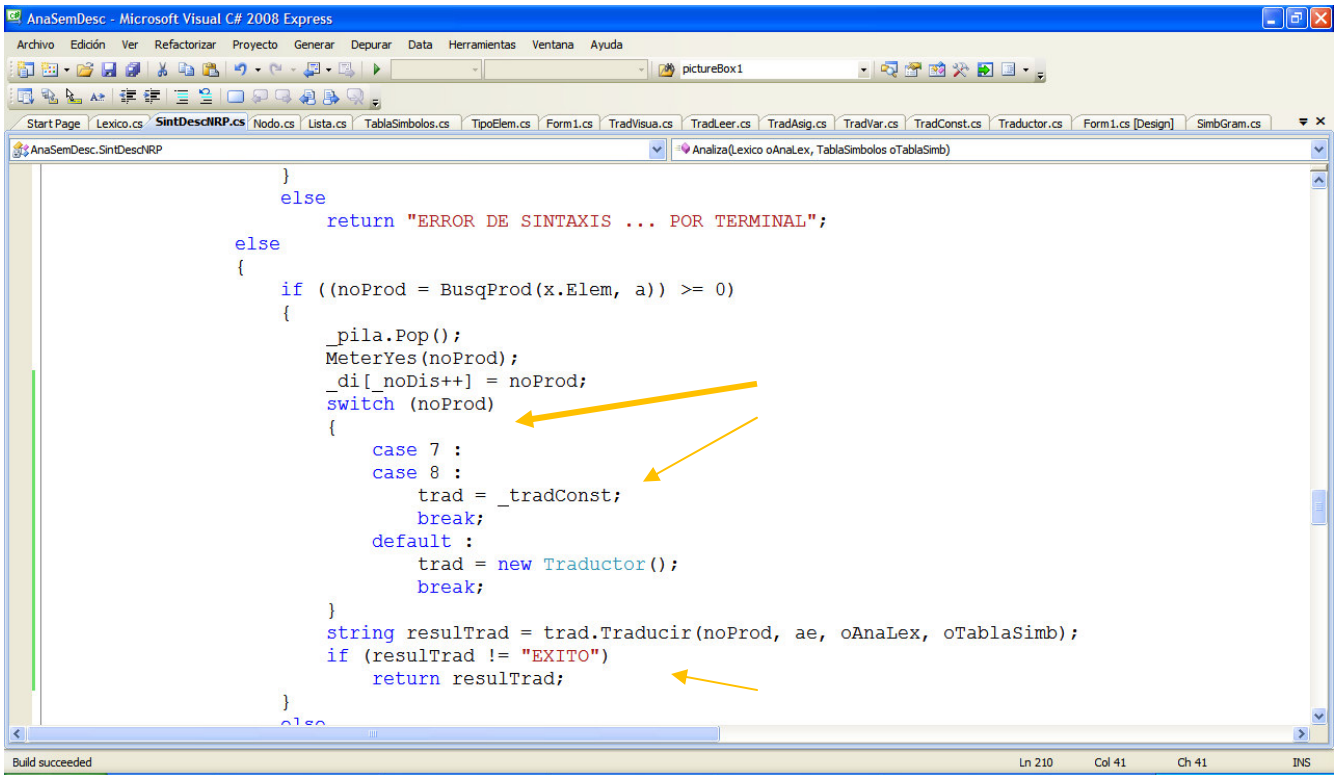


Fig. No. 9.6 Enlace del traductor `trad` al objeto `_tradConst` cuando es utilizada la producción 7.

El código del método `Traducir()` en la clase `TradConst` es el que debemos escribir, de manera que implementemos la traducción que nos está ocupando. Hasta ahora tenemos el método `Traducir()` en la clase `TradConst` sólo con la sentencia que retorna el éxito en la traducción.

Modifica al método `Traducir()` de la clase `TradConst` según se indica :

```
class TradConst : Traductor
{
    override public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
    {
        if (oAnaLex.Token[ae+1]=="id")
        {
            char car = oAnaLex.Lexema[ae + 1].ToUpper()[0];
            int indice = Convert.ToInt32(car) - 65;
            Nodo p = oTablaSimb.EncuentraToken(indice, oAnaLex.Lexema[ae + 1]);
            if (p == null)
                return _resulTraduccion[1];
            else if (p.Info.Clase != "")
                return _resulTraduccion[2];
            else
            {
                p.Info.Clase = "constante";
                return _resulTraduccion[0];
            }
        }
        else
            return _resulTraduccion[1];
    }
}
```

Antes de explicar el código anterior, añade la propiedad Clase en la clase TipoElem :

```
public string Clase
{
    get { return _clase; }
    set { _clase = value; }
}
```

También debemos agregar los 2 errores que estamos manejando en el método Traducir() de la clase TradConst, dentro del atributo _resulTraduccion de la clase Traductor :

```
class Traductor
{
    protected string[] _resulTraduccion = { "EXITO",
                                            "ERROR ... SE ESPERABA UN IDENTIFICADOR",
                                            "ERROR ... DECLARACION DE CONSTANTE DUPLICADA" };
    virtual public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
    {
        return _resulTraduccion[0];
    }
}
```

El primer if comprueba si el token que sigue al símbolo de tipo en w\$ es el token id, ver figura #9.2.

```
if (oAnaLex.Token[ae+1]=="id")
```

Si no es un id, el método retorna un mensaje de error cuyo índice es el 1 y que se define en la clase Traductor.

```
else
    return _resulTraduccion[1];
```

Si se cumple la condición del if, entonces calculamos el índice usando la función de desmenuzamiento que obtiene al primer caracter del lexema correspondiente al token id, cuyo atributo _clase asignaremos al valor “constante” .

```
char car = oAnaLex.Lexema[ae + 1].ToUpper()[0];
int indice = Convert.ToInt32(car) - 65;
```

La siguiente sentencia busca al token dentro de la tabla de símbolos y retorna una referencia al nodo si lo encuentra, de lo contrario retorna null.

```
Nodo p = oTablaSimb.EncuentraToken(indice, oAnaLex.Lexema[ae + 1]);
```

Después probamos si no se encontró el lexema para retornar el error 1 :

```
if (p == null)
    return _resulTraduccion[1];
```

Cuando si se encuentra el lexema, entonces validamos que no haya sido registrada la constante previamente. Si ya se encontraba registrada entonces retornamos el error 2 :

```
else if (p.Info.Clase != "")
    return _resulTraduccion[2];
```

Una vez que nos cercioramos de no haber encontrado un error, entonces escribimos al atributo _clase del id en la tabla de símbolos. Entonces retornamos el mensaje “ÉXITO” como resultado de la traducción.

```
else
{
    p.Info.Clase = "constante";
    return _resulTraduccion[0];
}
```

En la figura #9.7 se muestra la ejecución de la aplicación para el ejemplo en estudio, sin errores y visualizando la tabla de símbolos con el registro de las 2 constantes MAX y MENSAJE.

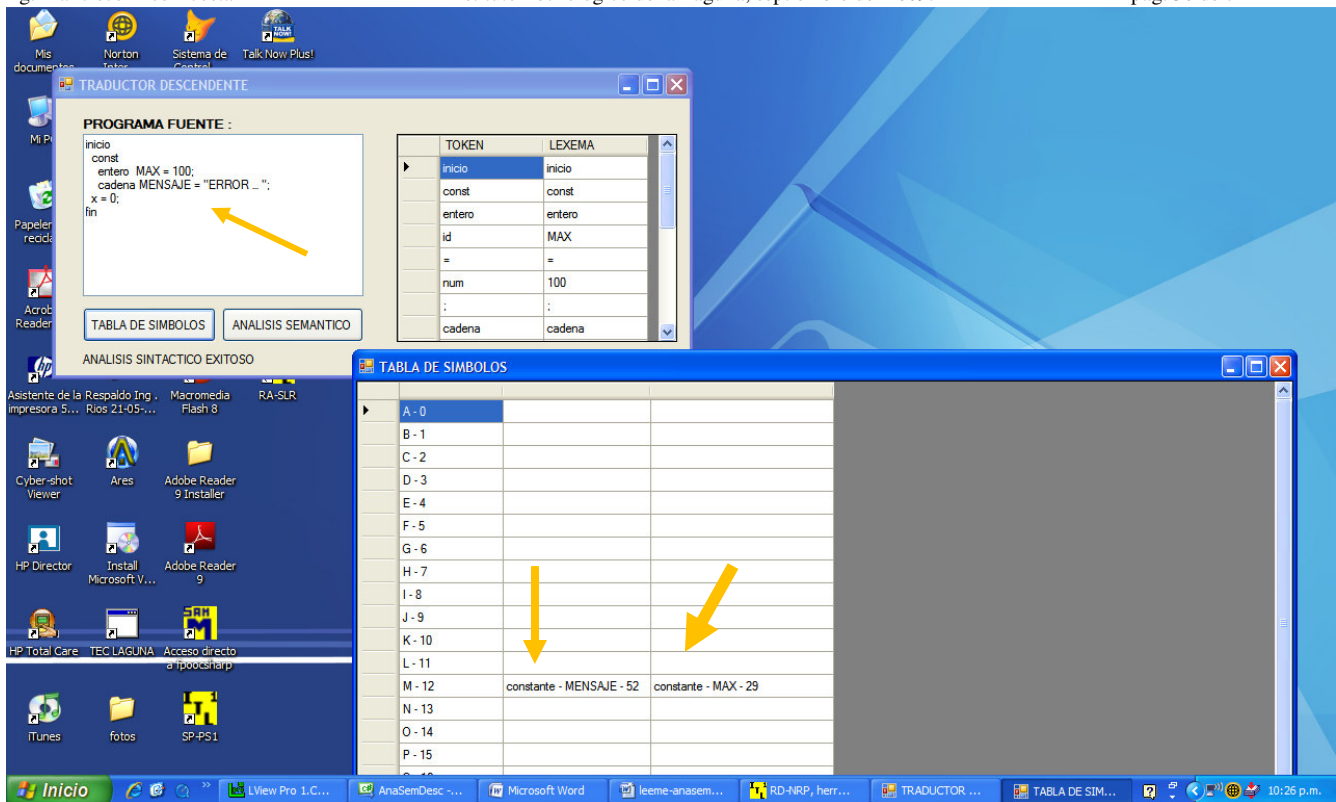


Fig. No. 9.7 Ejecución de la traducción sin error. Registro de MAX y MENSAJE como constantes.

Es necesario modificar el método `Visua()` en la clase `TablaSimbolos` para obtener el resultado de la figura #9.7.

```
public void Visua(System.Windows.Forms.DataGridView dGV)
{
    Nodo refNodo;
    int col;
    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;
        }
    }
}
```

Ahora provoquemos el error 1 eliminando el identificador MAX inicialmente, figura #9.8. Enseguida eliminamos al identificador MENSAJE no sin antes haber tecleado de nuevo al identificador MAX, figura #9.9.

El error 2 lo provocamos agregando la definición de la constante MAX pero ahora con otro tipo y otro valor, figura #9.10.

```
inicio
const
    entero MAX = 100;
    cadena MENSAJE = "ERROR ... ";
    real MAX = 3.14;
x = 0;
fin
```

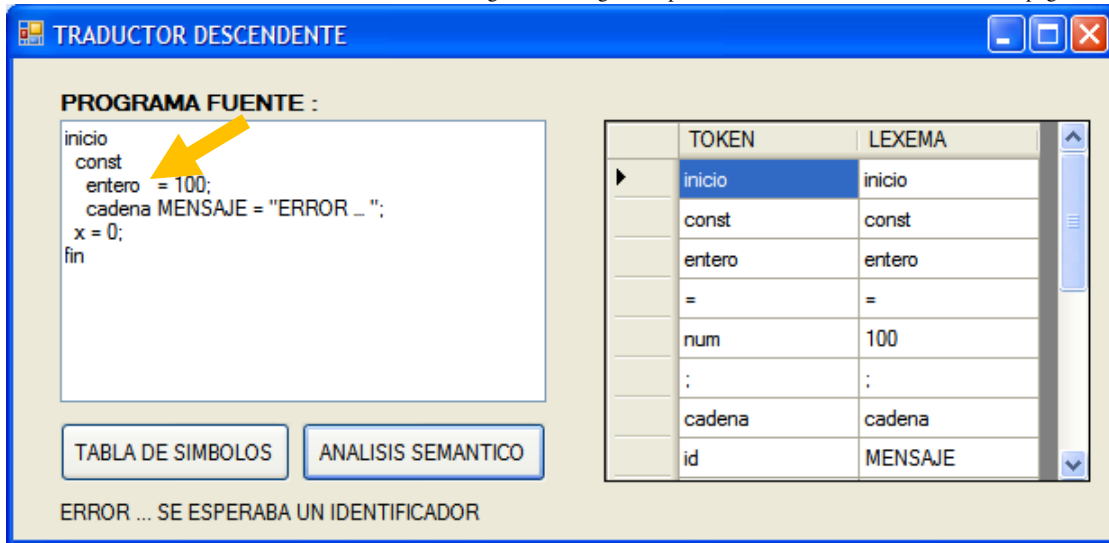


Fig. No. 9.8 Provocando al error 1 eliminando a MAX.

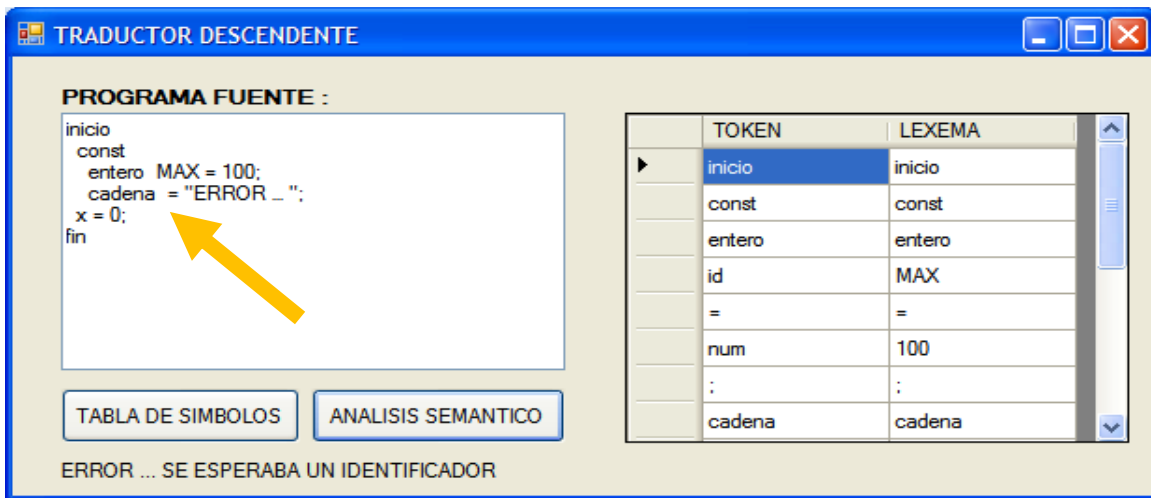


Fig. No. 9.9 Provocando al error 1 eliminando a MENSAJE.

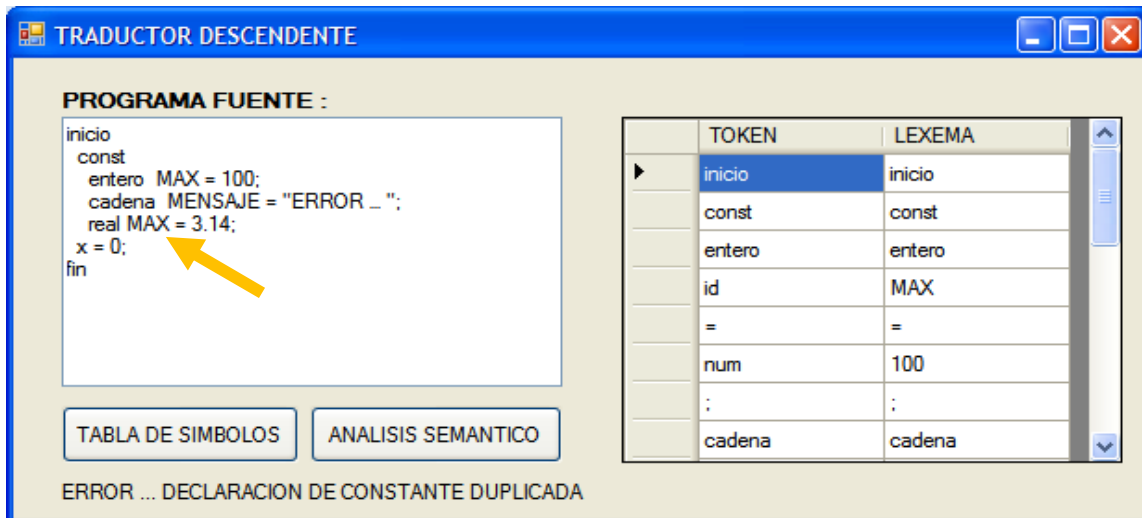


Fig. No. 9.10 Provocando al error 2 "CONSTANTE DUPLICADA".

Sigamos con la traducción que consiste en asignar el tipo de dato de la constante y almacenarla en la tabla de símbolos. Añadimos las nuevas reglas semánticas al esquema de traducción.

ESQUEMA DE TRADUCCION (versión 2)

noProd	Producción	Reglas semánticas
7	R -> Y id = Z ; R'	p=oTablaSimb.EncuentraToken(Hash(id.lexema), id.lexema) p.Info.Clase = "constante" p.Info.Tipo = Y.tipo
8	R' -> Y id = Z ; R'	p=oTablaSimb.EncuentraToken(Hash(id.lexema), id.lexema) p.Info.Clase = "constante" p.Info.Tipo = Y.tipo
10	Y -> entero	Y.tipo = "entero"
11	Y -> cadena	Y.tipo = "cadena"
12	Y -> real	Y.tipo = "real"

Sabemos que cuando el reconocedor utiliza cualquiera de las producciones 7 u 8, el símbolo apuntado por ae es el tipo de la constante declarada, así que podemos efectuar la traducción en ese instante, es decir, cuando se ha usado en la derivación de la sentencia, a las producciones correspondientes a los números mencionados. NO ES NECESARIO esperarnos al empleo de las producciones con números 10, 11 y 12.

De acuerdo a lo anterior, sólo es necesario codificar la regla semantica :

```
p.Info.Tipo = Y.tipo
```

Agregamos la línea de código señalada que implementa la regla semántica :

```
class TradConst : Traductor
{
    override public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
    {
        if (oAnaLex.Token[ae+1]=="id")
        {
            char car = oAnaLex.Lexema[ae + 1].ToUpper()[0];
            int indice = Convert.ToInt32(car) - 65;
            Nodo p = oTablaSimb.EncuentraToken(indice, oAnaLex.Lexema[ae + 1]);
            if (p == null)
                return _resulTraduccion[1];
            else if (p.Info.Clase != "")
                return _resulTraduccion[2];
            else
            {
                p.Info.Clase = "constante";
                p.Info.Tipo = oAnaLex.Lexema[ae];
                return _resulTraduccion[0];
            }
        }
        else
            return _resulTraduccion[1];
    }
}
```

El mensaje oAnaLex.Lexema[ae] retorna el tipo de la constante

También tenemos que agregar la propiedad Tipo a la clase TipoElem :

```
public string Tipo
{
    get { return _tipo; }
    set { _tipo = value; }
}
```


Por último modificamos el método `Visua()` de la clase `TablaSimbolos` de manera que muestre el tipo del `id` –en este caso la constante–.

```
public void Visua(System.Windows.Forms.DataGridView dGV)
{
    Nodo refNodo;
    int col;
    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.Tipo+" - " +
                refNodo.Info.Nombre + " - " + refNodo.Info.Posicion.ToString();
            refNodo = refNodo.Sig;
        }
    }
}
```

Veamos la ejecución de la aplicación en la figura #9.11 donde se muestra la tabla de símbolos que incluye la visualización del atributo `_tipo`.

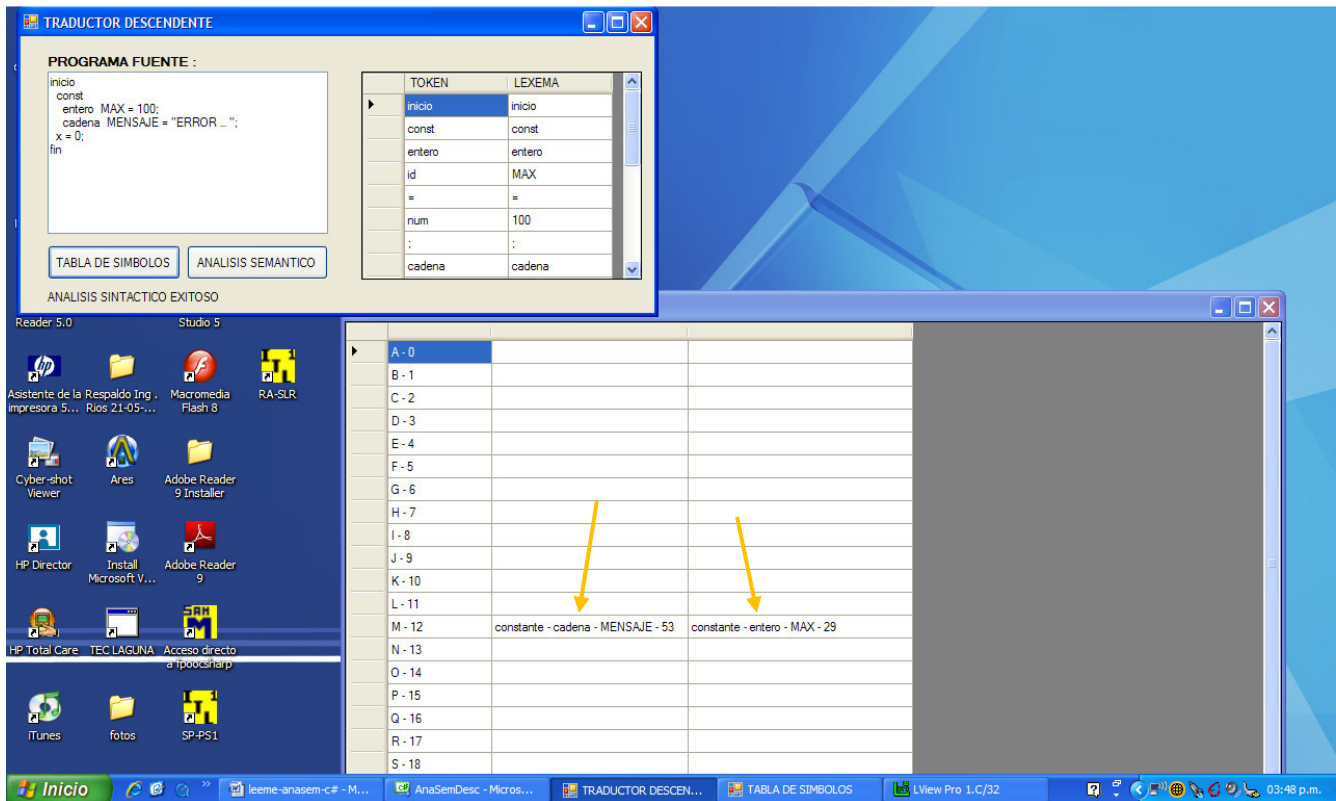


Fig. No. 9.11 Atributo `_tipo` registrado en la tabla de símbolos.

Seguimos con la adición de las reglas semánticas incluídas para lograr la traducción que almacena el número de bytes reservados y asignación del valor para una constante.

Para esta traducción haremos uso de un método que asigne el atributo `_noBytes` al valor adecuado según sea el tipo declarado para la constante. Este método lo llamamos `CalcNoBytes()` y tiene dos parámetros : el tipo de la constante declarada y su valor. El valor de la constante es necesario debido a que el número de bytes que se deben reservar para una constante cadena es dos veces su longitud.

Las reglas semánticas las relacionamos con las producciones 7 y 8.

noProd	Producción	Reglas semánticas
7	R -> Y id = Z ; R'	p=oTablaSimb.EncuentraToken(Hash(id.lexema), id.lexema) p.Info.Clase = "constante" p.Info.Tipo = Y.tipo p.Info.NoBytes = CalcNoBytes(Y.tipo , Z.valor)
8	R' -> Y id = Z ; R'	p=oTablaSimb.EncuentraToken(Hash(id.lexema), id.lexema) p.Info.Clase = "constante" p.Info.Tipo = Y.tipo p.Info.NoBytes = CalcNoBytes(Y.tipo , Z.valor)
10	Y -> entero	Y.tipo = "entero"
11	Y -> cadena	Y.tipo = "cadena"
12	Y -> real	Y.tipo = "real"
13	Z->num	Z.valor = num.lexema
14	Z->cad	Z.valor = cad.lexema

El método CalcNoBytes() lo insertamos en la clase Traductor debido a que lo podremos usar en otra tarea, por ejemplo cuando traduzcamos la declaración de variables y requiramos de almacenar el número de bytes para una variable. Veamos la figura #9.12 que indica el código del método en cuestión.

```

virtual public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
{
    return _resulTraduccion[0];
}
public int CalcNoBytes(string tipo, string lexema)
{
    switch (tipo)
    {
        case "entero":
            return 4;
            break;
        case "real":
            return 8;
            break;
        case "cadena":
            return lexema.Length * 2;
            break;
        default :
            return 0;
            break;
    }
}
    
```

Fig. No. 9.12 Definición del método CalcNoBytes() en la clase Traductor.

La llamada al método CalcNoBytes () la insertamos dentro del método Traducir () de la clase TradConst según se muestra a continuación.

```

override public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
{
    if (oAnaLex.Token[ae+1]=="id")
    {
        char car = oAnaLex.Lexema[ae + 1].ToUpper()[0];
        int indice = Convert.ToInt32(car) - 65;
        Nodo p = oTablaSimb.EncuentraToken(indice, oAnaLex.Lexema[ae + 1]);
        if (p == null)
            return _resulTraduccion[1];
        else if (p.Info.Clase != "")
            return _resulTraduccion[2];
        else
        {
            p.Info.Clase = "constante";
            p.Info.Tipo = oAnaLex.Lexema[ae];
            if (oAnaLex.NoTokens > ae + 3)
                if (oAnaLex.Token[ae + 3] == "num" || oAnaLex.Token[ae + 3] == "cad")
                {
                    p.Info.NoBytes = CalcNoBytes(oAnaLex.Lexema[ae], oAnaLex.Lexema[ae + 3]);
                    p.Info.Valor = oAnaLex.Lexema[ae + 3];
                    return _resulTraduccion[0];
                }
            return _resulTraduccion[3];
        }
    }
    else
        return _resulTraduccion[1];
}

```

La validación permite evitar un error fatal al tratar de acceder a un elemento fuera de los límites del arreglo, cuando la entrada –programa fuente- se compone de menos tokens que $ae + 3$. Cuando así sucede, retornamos el error 3 "ERROR ... NO SE ESPECIFICA VALOR PARA LA CONSTANTE" el cual debemos agregar al atributo `_resulTraduccion` de la clase Traductor. Este mismo error lo retornamos cuando en $ae + 3$ no tenemos un token num o bien, un token cad. Esta tarea la cumple la segunda validación :

```

if (oAnaLex.Token[ae + 3] == "num" || oAnaLex.Token[ae + 3] == "cad")

```

Falta añadir la definición de las propiedades NoBytes y Valor en la clase TipoElem :

```

public int NoBytes
{
    get { return _noBytes; }
    set { _noBytes = value; }
}

public string Valor
{
    get { return _valor; }
    set { _valor = value; }
}

```

También requerimos modificar el método Visua () en la clase TablaSimbolos tal y como lo hemos hecho para los atributos `_clase` y `_tipo`.

```

public void Visua(System.Windows.Forms.DataGridView dGV)
{
    Nodo refNodo;
    int col;
    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.Tipo+" - "+
            refNodo.Info.NoBytes.ToString()+" - "+
            refNodo.Info.Nombre + " - " +
            refNodo.Info.Valor + " - " +
            refNodo.Info.Posicion.ToString();
        refNodo = refNodo.Sig;
    }
}
}

```

Una vez hechas todas las modificaciones podemos ejecutar nuestra aplicación. Si todo está bien entonces tendremos algo como lo mostrado en la figura #9.13.

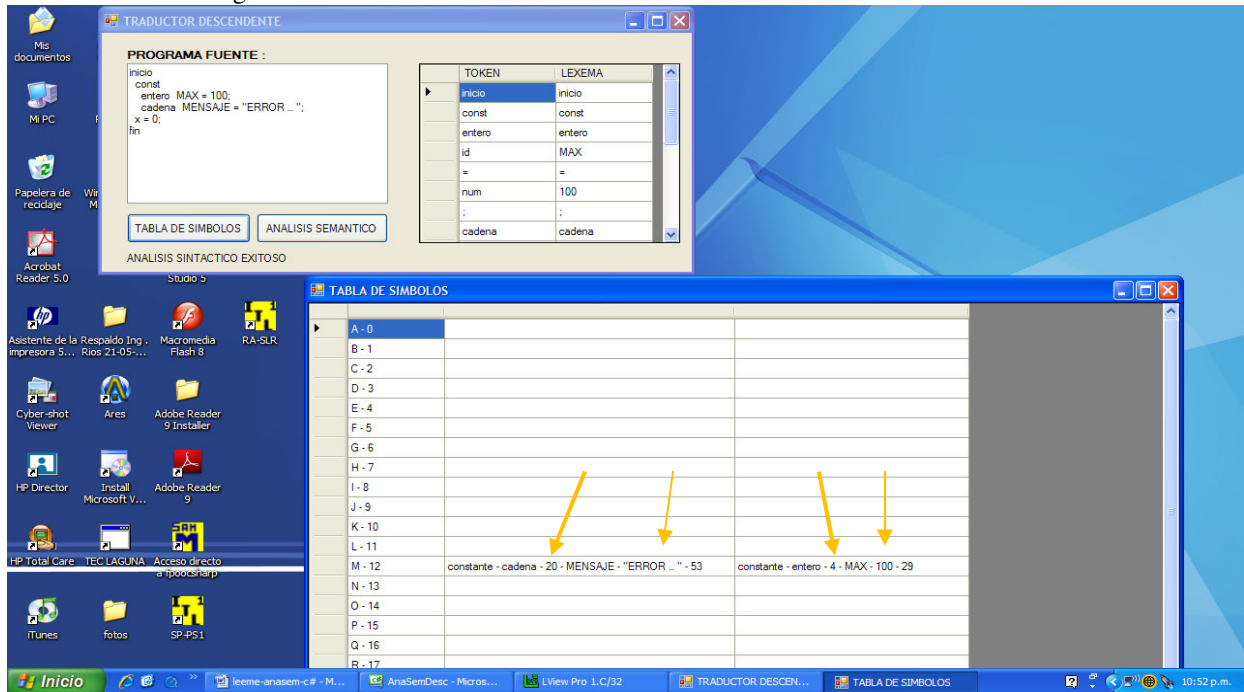


Fig. No. 9.13 Visualización del atributo _noBytes.

Notemos el registro de los valores para cada constante, para MAX el valor es de 100 y para MENSAJE el valor es "ERROR ...". La figura #9.14 muestra la provocación del error 3 dejando sin escribir el valor para la constante MAX.

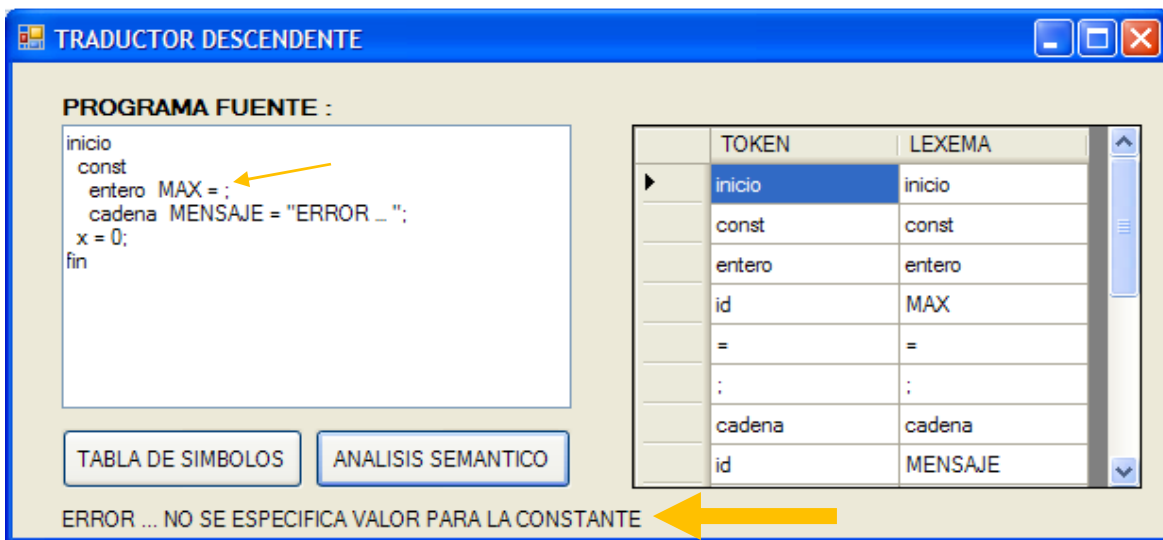


Fig. No. 9.14 Provocación del error 3.

10 Traducción (b) : ALMACENAR EL TIPO DE DATO, NÚMERO DE BYTES DE LAS VARIABLES DECLARADAS, EN LOS ATRIBUTOS RESPECTIVOS DE LA TABLA DE SÍMBOLOS.

La definición dirigida por sintaxis que escribamos deberá afectar los atributos `_clase`, `_tipo` y `_noBytes` del `identificador` declarado como variable, y cuyo atributo `_nombre` corresponda al lexema del identificador –variable-. Usemos el ejemplo :

```

inicio
const
    entero MAX = 100;
    cadena MENSAJE = "ERROR ... ";
var
    real x,y,z;
    entero i,j;
    cadena s;
x = 0;
fin
    
```



Ingreseemos este ejemplo en la ventana de simulación del RD-NRP sólo para darnos una idea de lo que ocurre durante el proceso de derivación del programa fuente, figura 10.1. El proceso de reconocimiento de variables, es similar al de las constantes pero existe una diferencia : *la declaración de variables puede ser de una o mas involucrando al mismo tipo de dato.*

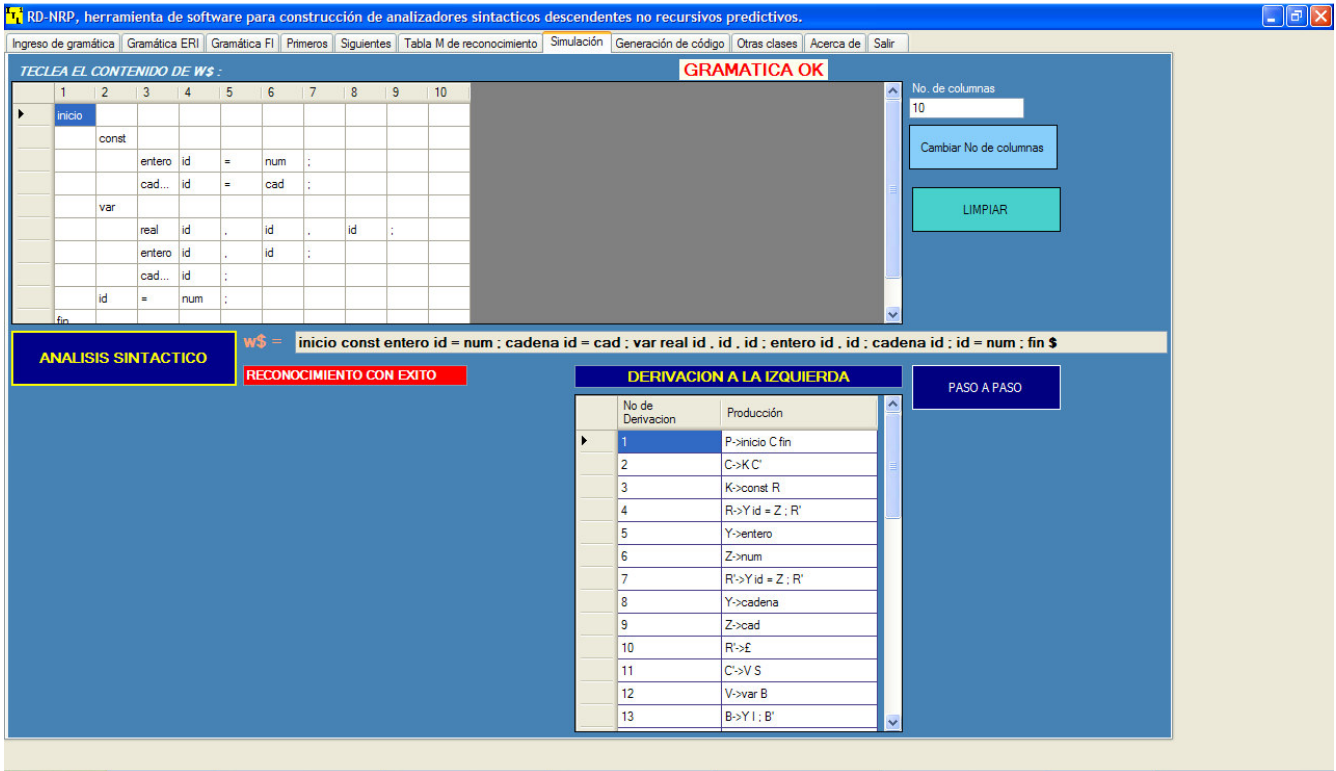


Fig. No. 10.1 Ingreso del ejemplo al RD-NRP.

Hacemos el análisis sintáctico cuyo resultado es exitoso. Luego seguimos con la simulación paso a paso hasta llegar a la sustitución de las producciones con número 10, 11, 12, 16, 17, 18, 19, 20 y 21.

- B -> Y I ; B' ... 16
- B' -> Y I ; B' ... 17
- B' -> £ ... 18
- Y -> entero ... 10
- Y -> cadena ... 11
- Y -> real ... 12

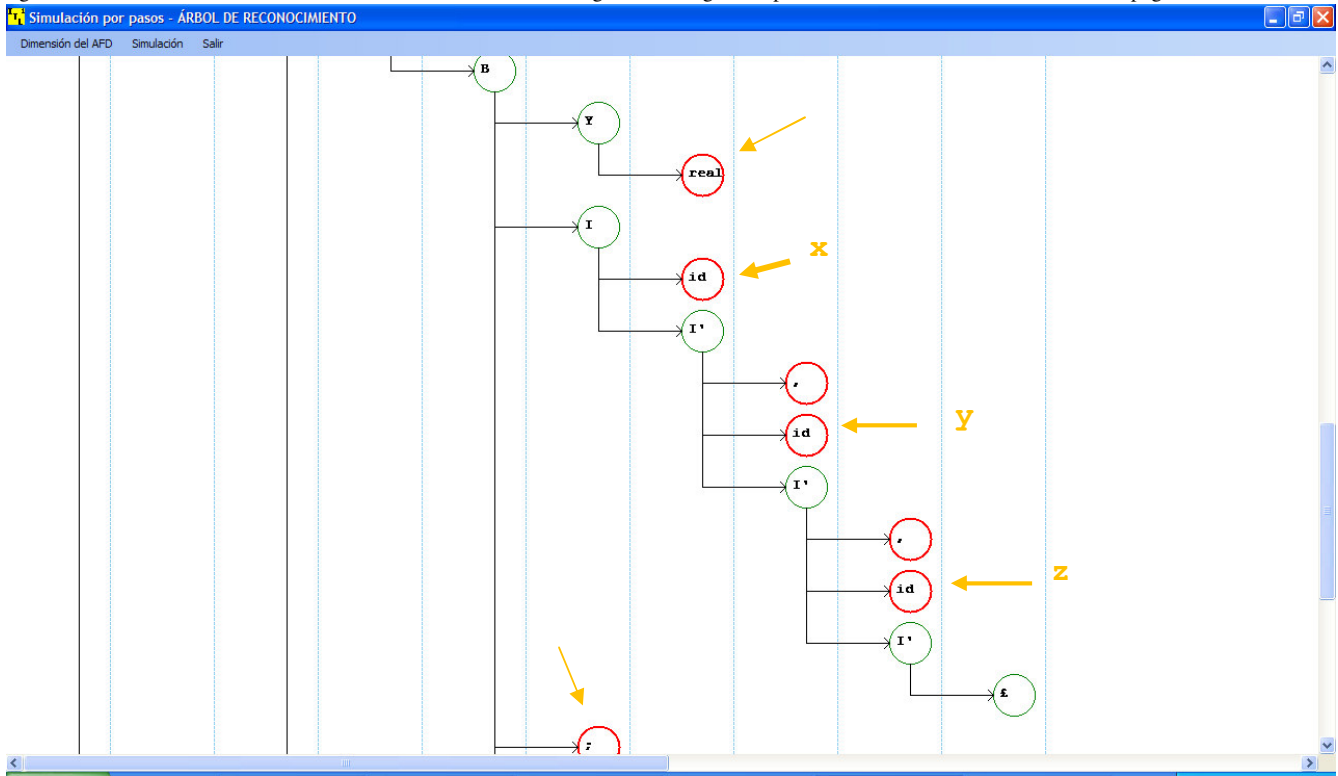


Fig. No. 10.2 Árbol de reconocimiento para la declaración de variables : real x,y,z;

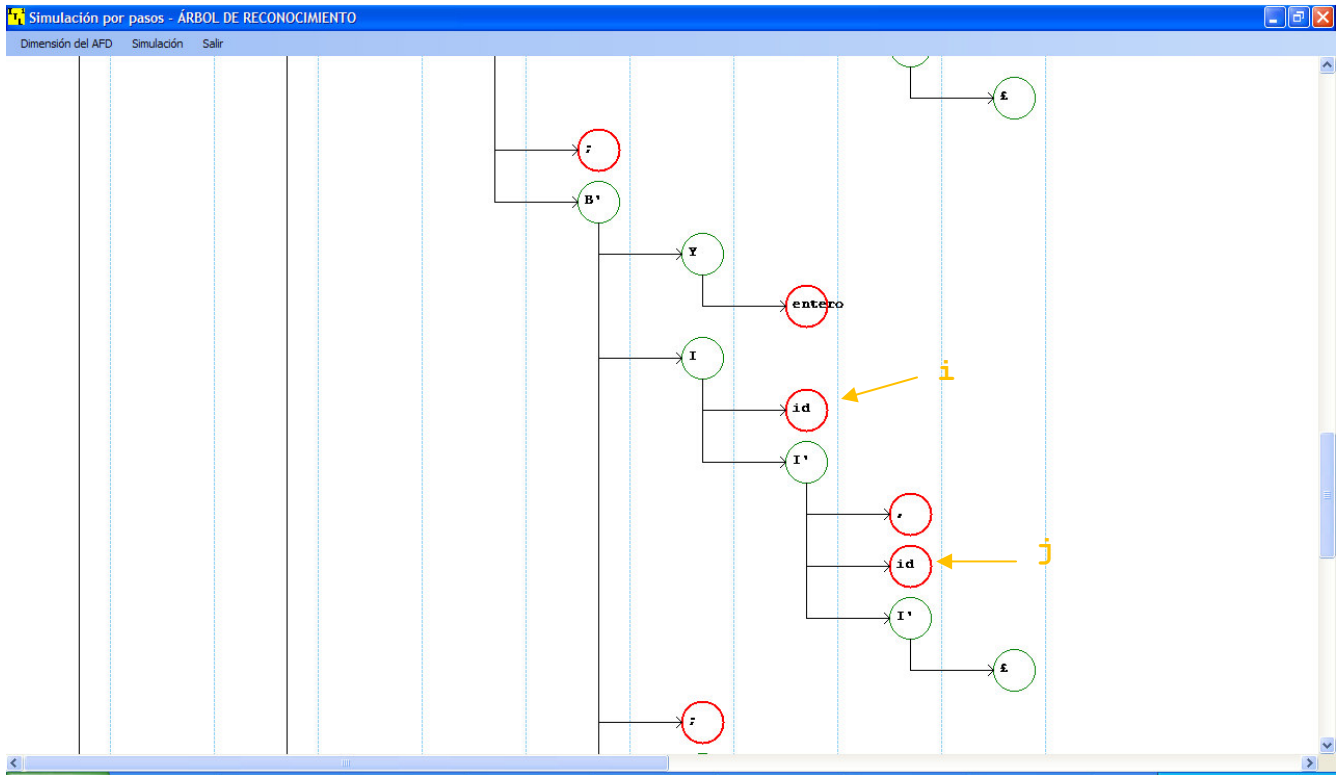


Fig. No. 10.3 Árbol de reconocimiento para la declaración de variables : entero i,j;

$I \rightarrow id I'$... 19
 $I' \rightarrow , id I'$... 20
 $I' \rightarrow \epsilon$... 21

La figura #10.2 muestra el árbol de reconocimiento para los 2 identificadores de variables reales : x, y, z. La figura #10.3 muestra el árbol para la declaración entero i, j; y la figura #10.4 el de la declaración cadena s;

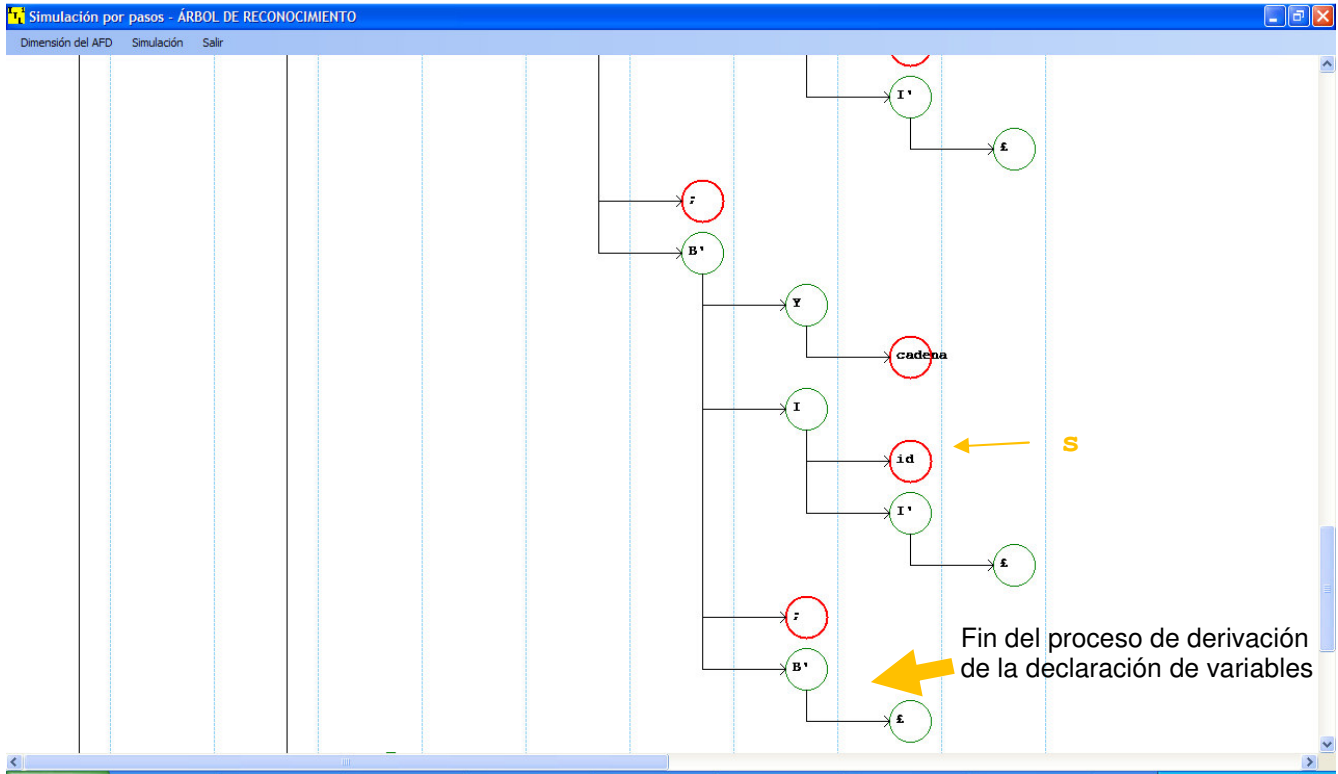


Fig. No. 10.4 Árbol de reconocimiento para la declaración de variables : cadena s;

La sustitución de las no terminales B o B' inician el reconocimiento de una declaración de variable, salvo el caso en que se sustituye la B' por el €. La definición dirigida por sintaxis propuesta es :

DEFINICIÓN DIRIGIDA POR SINTAXIS

noProd	Producción	Reglas semánticas
16	B -> Y I ; B'	I.tipo = Y.tipo I.noBytes = Y.noBytes
17	B' -> Y I ; B'	I.tipo = Y.tipo I.noBytes = Y.noBytes
18	B' -> €	
19	I -> id I'	id.tipo = I.tipo id.clase = "variable" id.noBytes = I.noBytes
20	I' -> , id I'	id.tipo = I'.tipo id.clase = "variable" id.noBytes = I'.noBytes
21	I' -> €	
10	Y -> entero	Y.tipo = "entero" Y.noBytes = 4

11	Y -> cadena	Y.tipo = "cadena" Y.noBytes = 0 // NO DEFINIDO EN TIEMPOS DE DECLARACION, ES DINAMICO EL NÚMERO DE BYTES
12	Y -> real	Y.tipo = "real" Y.noBytes = 8

Iniciemos por especificar un primer esquema de traducción que incluya solamente la asignación, y el almacenamiento en la tabla de símbolos del atributo `_clase`, el cual deberá tener el valor "variable".

La asignación la debemos efectuar cuando el proceso de reconocimiento aplica la producción 19 o la 20. En la figura 10.5 observamos los valores de `ae`, `X` y `a` cuando es sustituida la producción 19: `I -> id I'`. En este caso hay que resaltar que `ae` referencia al token `id` que representa la variable declarada.

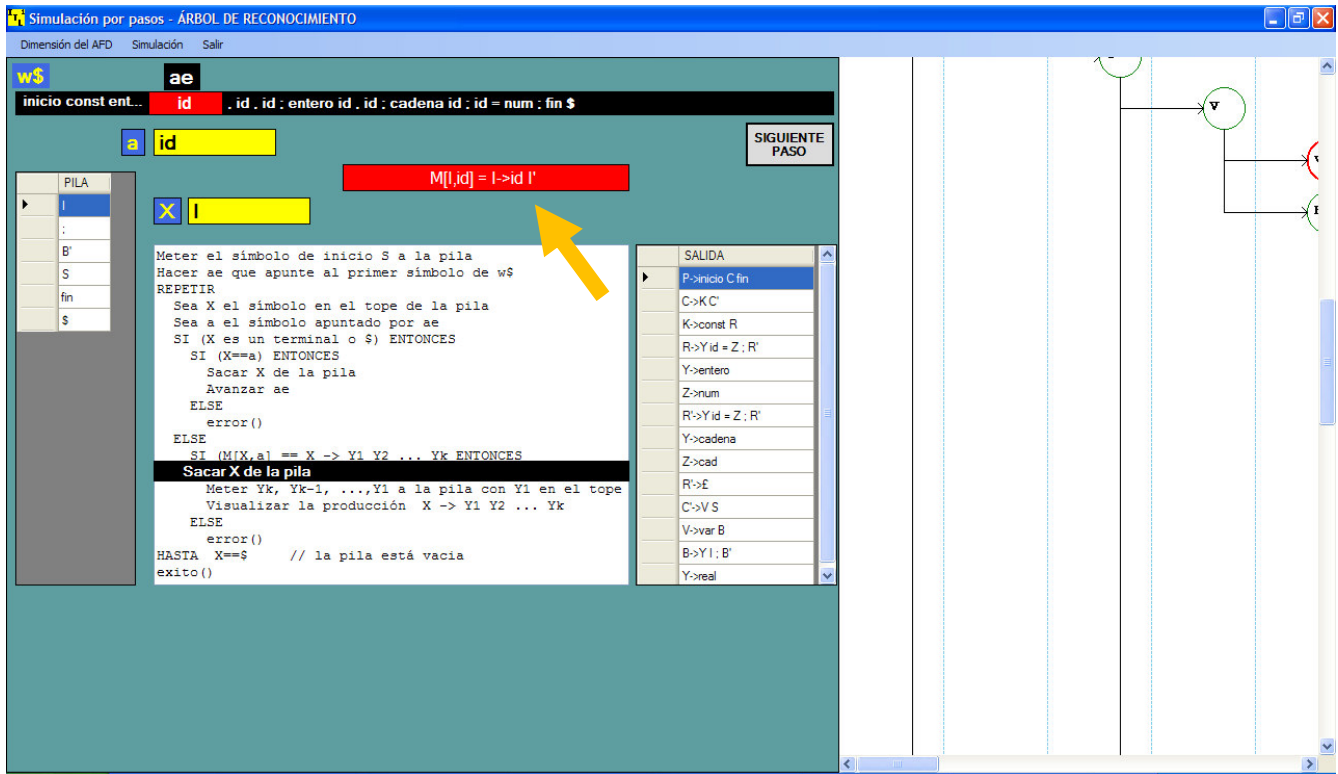


Fig. No. 10.5 Uso de la producción 19: `I -> id I'` para reconocer declaración de variables.

Ahora veamos cuando se utiliza la producción 20: `I' -> , id I'` que símbolo referencia `ae`. Encontramos según se indica en la figura #10.6 que `ae` apunta al símbolo terminal coma (,) mientras que el token `id` se accede mediante el índice `ae + 1`. Tomando en cuenta estas consideraciones, escribimos el esquema de traducción:

ESQUEMA DE TRADUCCION (versión 1)

noProd	Producción	Reglas semánticas
19	<code>I -> id I'</code>	<code>p=TablaSimb.EncuentraToken(Hash(id.lexema), id.lexema)</code> <code>p.Info.Clase = "variable"</code>
20	<code>I' -> , id I'</code>	<code>p=TablaSimb.EncuentraToken(Hash(id.lexema), id.lexema)</code> <code>p.Info.Clase = "variable"</code>

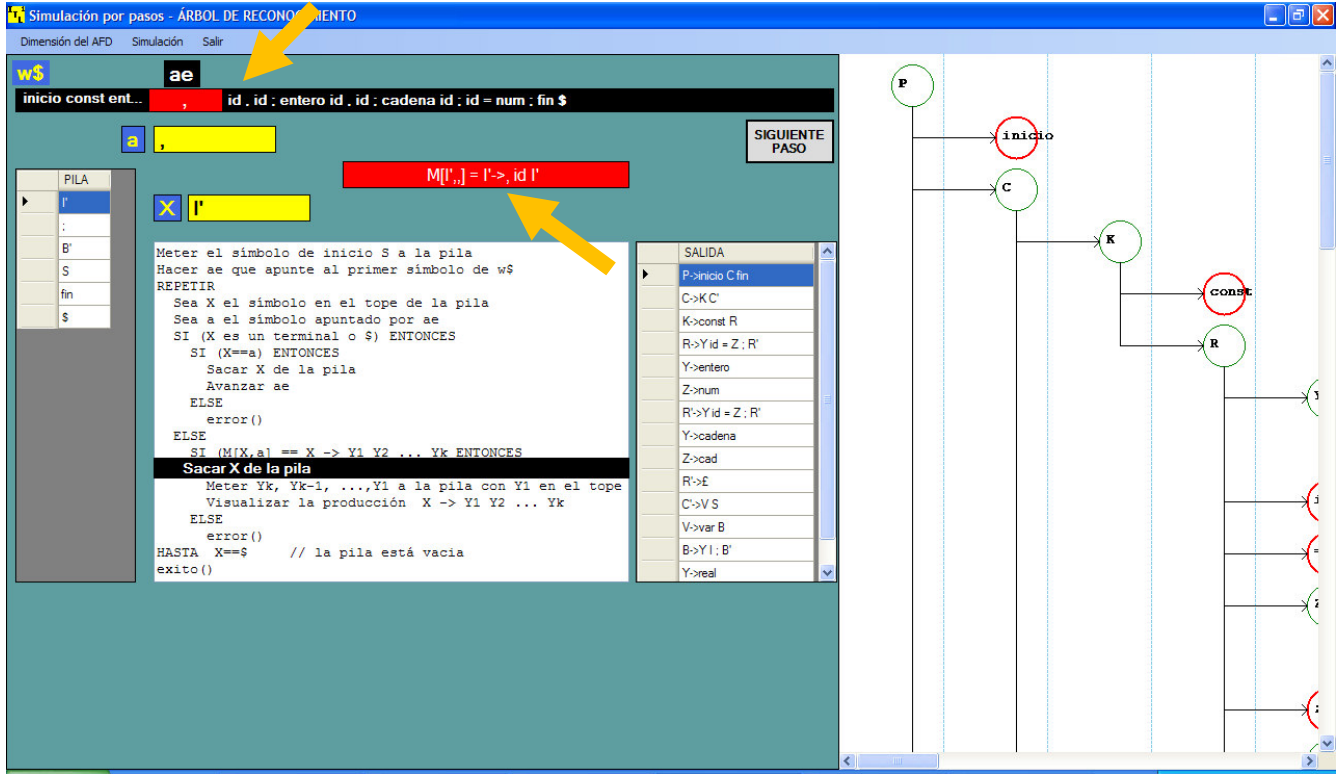


Fig. No. 10.6 Uso de la producción 20 : I' -> , id I' para reconocer declaración de variables.

Tenemos que incluir el caso del uso de estas 2 producciones 19 y 20, dentro del método Analiza() de la clase SintDescNRP :

```

else
{
    if ((noProd = BusqProd(x.Elem, a)) >= 0)
    {
        _pila.Pop();
        MeterYes(noProd);
        _di[_noDis++] = noProd;
        switch (noProd)
        {
            case 7 :
            case 8 :
                trad = _tradConst;
                break;
            case 19 :
            case 20 :
                trad = _tradVar;
                break;
            default :
                trad = new Traductor();
                break;
        }
        string resulTrad = trad.Traducir(noProd, ae, oAnaLex, oTablaSimb);
        if (resulTrad != "EXITO")
            return resulTrad;
    }
    else
        return "ERROR DE SINTAXIS ... NO SE ENCONTRO LA PRODUCCION";
}
    
```

En esta traducción sobre declaración de variables pueden generarse 2 errores mas : cuando el id de la variable ya se ha usado para una constante, y cuando el id de la variable ya ha sido utilizado para nombrar a otra variable.

Los llamaremos :

- "ERROR ... LA VARIABLE YA SE ENCUENTRA DECLARADA COMO CONSTANTE",
- "ERROR ... DECLARACION DE VARIABLE DUPLICADA"

Agreguemos estos 2 errores al atributo `_resulTraduccion` de la clase `Traductor`, según se muestra en la figura #10.7 :

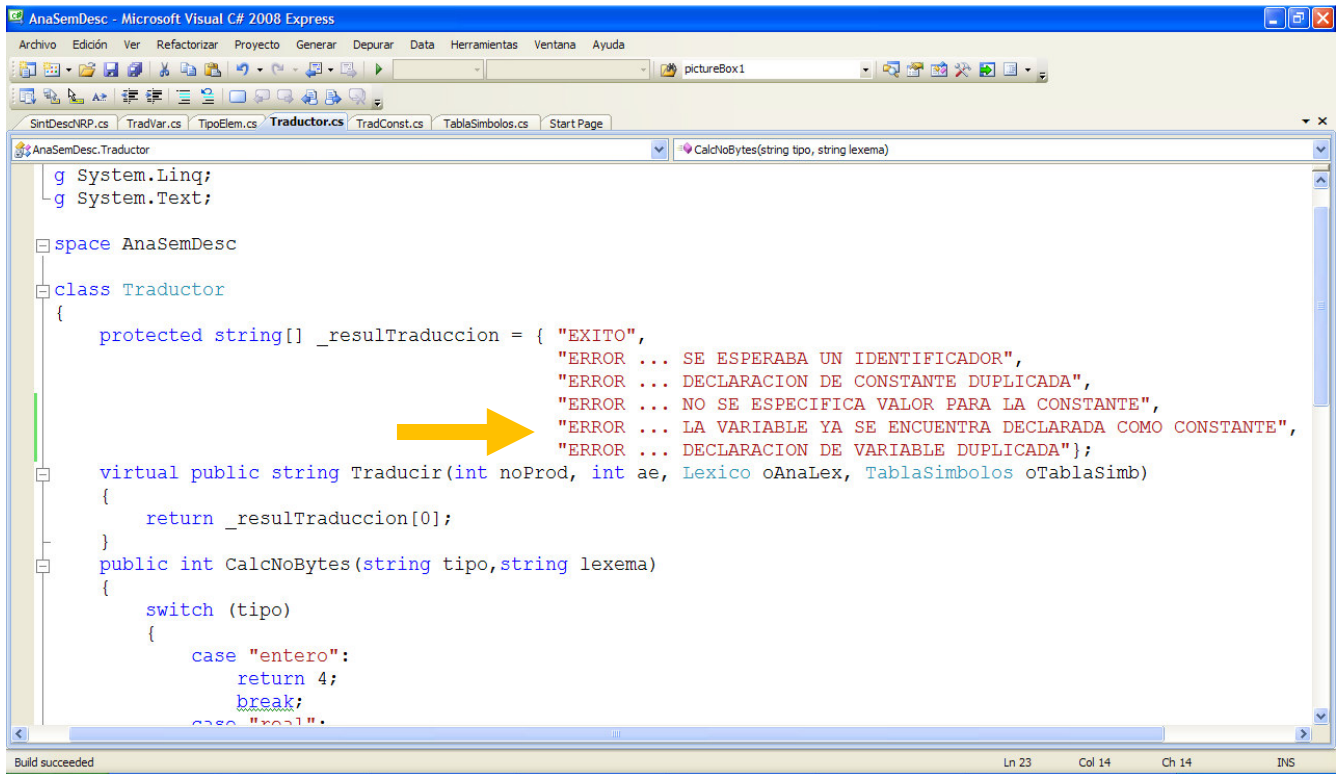


Fig. No. 10.7 Adición de los 2 nuevos errores con índices 4 y 5 en el atributo `_resulTraduccion`.

Ahora escribamos el método `Traducir()` de la clase `TradVar` según se indica enseguida :

```
class TradVar : Traductor
{
    override public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
    {
        string idVar = noProd == 19 ? oAnaLex.Token[ae] : (noProd == 20 ? oAnaLex.Token[ae+1] : "");
        string idLex = noProd == 19 ? oAnaLex.Lexema[ae] : (noProd == 20 ? oAnaLex.Lexema[ae + 1] : "");
        if (idVar == "id")
        {
            char car = idLex.ToUpper()[0];
            int indice = Convert.ToInt32(car) - 65;
            Nodo p = oTablaSimb.EncuentraToken(indice, idLex);
            if (p == null)
                return _resulTraduccion[1];
            else if (p.Info.Clasa != "")
                return p.Info.Clasa == "constante" ? _resulTraduccion[4] : _resulTraduccion[5];
            else
            {
                p.Info.Clasa = "variable";
                return _resulTraduccion[0];
            }
        }
        else
            return _resulTraduccion[1];
    }
}
```

Las primeras 2 instrucciones se encargan de acceder al token con índice `ae` o `ae + 1` según el número de la producción utilizada en el llamado y al lexema para ese id.

```
string idVar = noProd == 19 ? oAnaLex.Token[ae] : (noProd == 20 ? oAnaLex.Token[ae+1] : "");
string idLex = noProd == 19 ? oAnaLex.Lexema[ae] : (noProd == 20 ? oAnaLex.Lexema[ae + 1] : "");
```

La explicación para las demás sentencias es la misma que para el método Traducir () que ya hemos hecho para la clase TradConst, con tal vez una cuestión nueva :

```
else if (p.Info.Clase != "")
    return p.Info.Clase == "constante" ? _resulTraduccion[4] : _resulTraduccion[5];
```

Probamos con el operador ternario si el atributo _clase se ha asignado al valor de "constante", entonces retornamos el error 4 si no, será el valor "variable" por lo que regresamos el error 5.

Sólo falta ejecutar la aplicación. La figura #10.8 muestra cuando no existen errores en el análisis, la figura #10.9 expone el error número 4, y la figura #10.10 visualiza el error número 5.

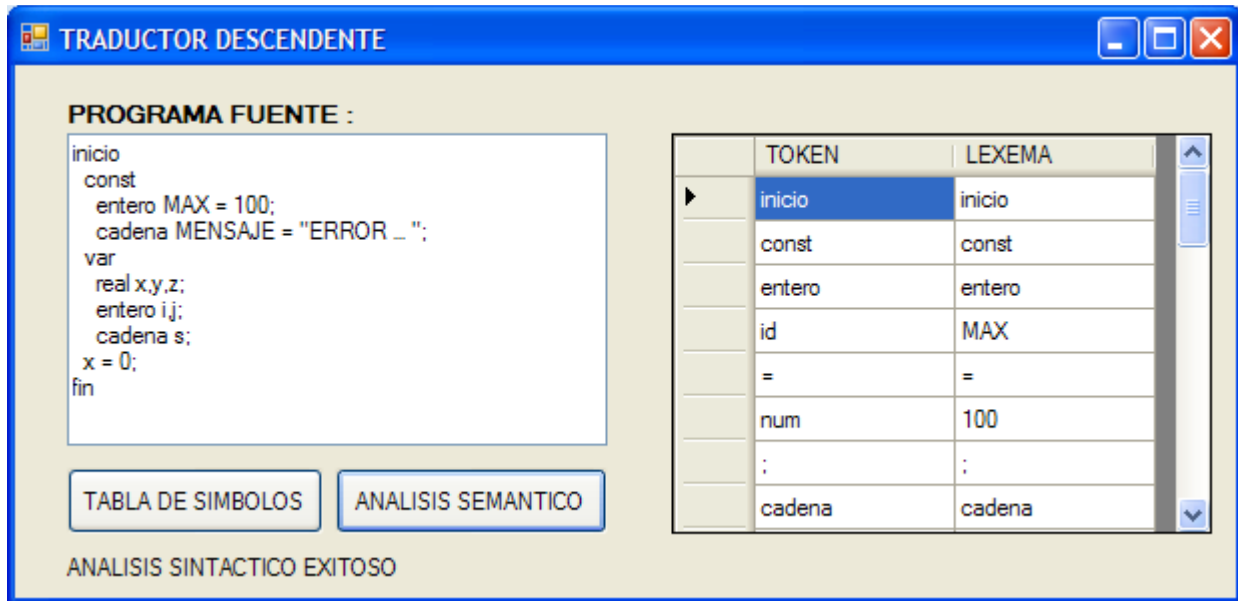


Fig. No. 10.8 Ejecución sin error.

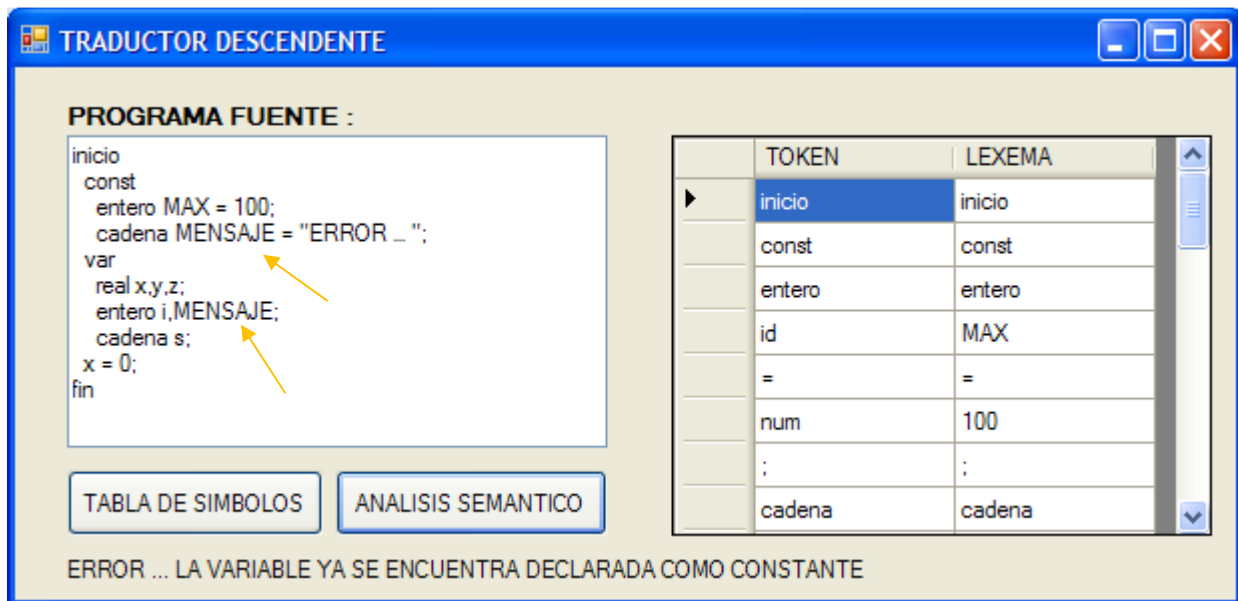


Fig. No. 10.9 Variable MENSAJE previamente definida como constante.

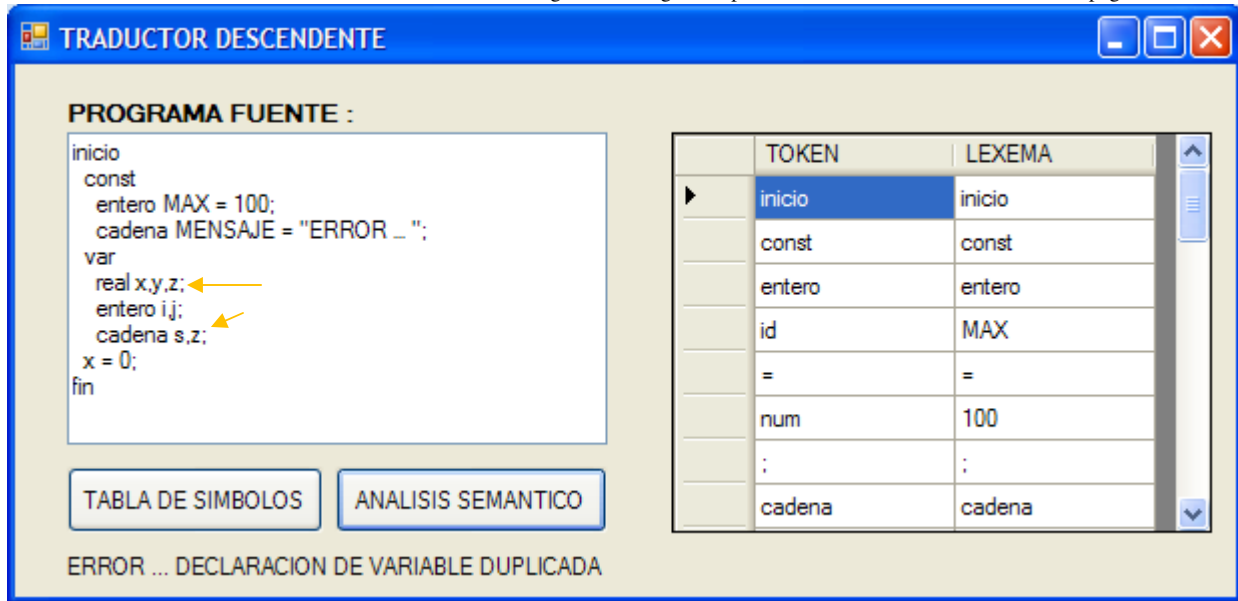


Fig. No. 10.10 Variable z duplicada en su declaración.

La figura 10.11 muestra la tabla de símbolos con el atributo `_clase` asignado para las variables declaradas.

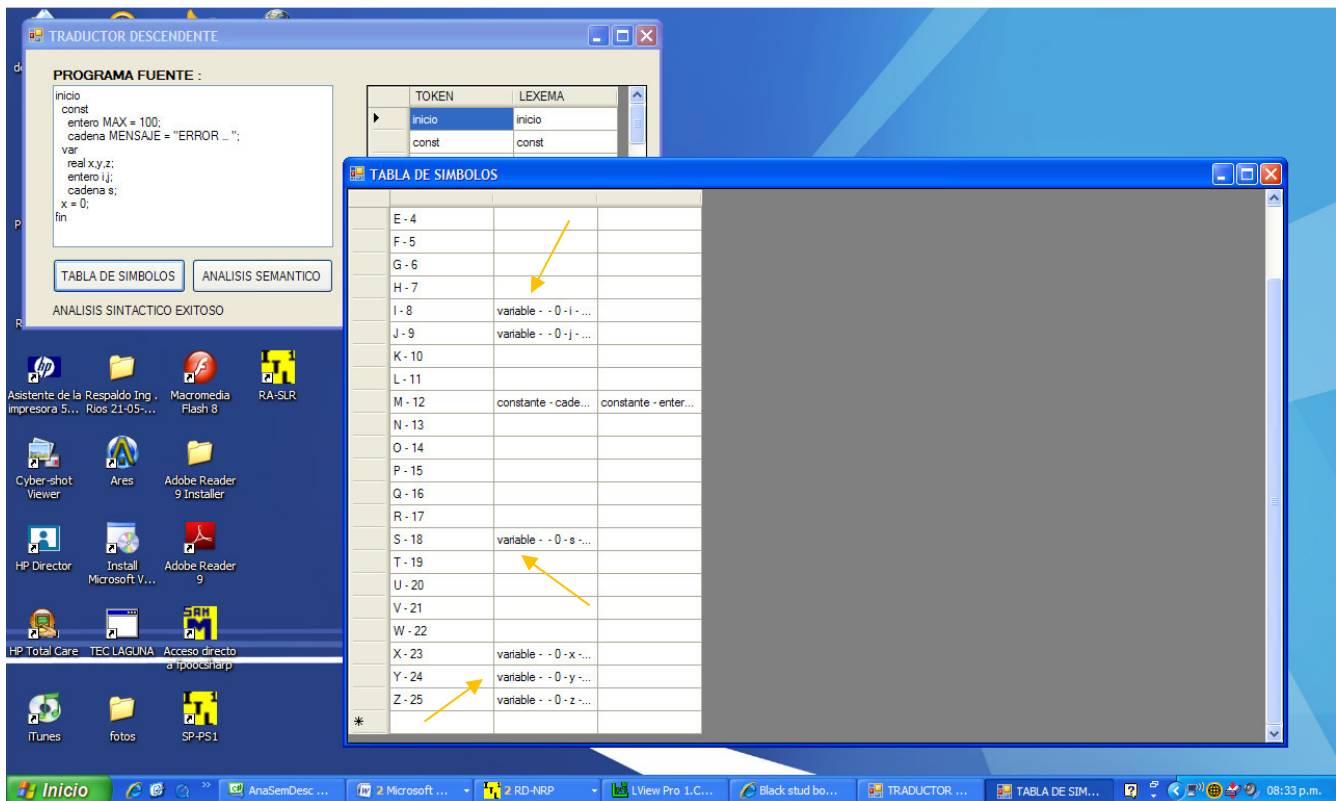


Fig. No. 10.11 Tabla de símbolos con las variables declaradas asignadas en su atributo `_clase`.

La etapa siguiente es agregar al esquema de traducción las reglas semánticas para almacenar el tipo y el número de bytes de una variable declarada.

Una segunda versión del esquema de traducción que incluye a las nuevas reglas semánticas es :

noProd	Producción	Reglas semánticas
16	B -> Y I ; B'	_tradVar._enTraduccion = true
17	B' -> Y I ; B'	_tradVar._enTraduccion = true
18	B' -> ε	_tradVar._enTraduccion = false
10	Y -> entero	_tradVar._tipo = "entero" _tradVar._noBytes = 4
11	Y -> cadena	_tradVar._tipo = "cadena" _tradVar._noBytes = 0 // NO DEFINIDO EN TIEMPOS DE DECLARACION, ES DINAMICO EL NÚMERO DE BYTES
12	Y -> real	_tradVar._tipo = "real" _tradVar._noBytes = 8
19	I -> id I'	p=oTablaSimb.EncuentraToken(Hash(id.lexema), id.lexema) p.Info.Clase = "variable" p.Info.Tipo = _tradVar._tipo p.Info.NoBytes = _tradVar._noBytes
20	I' -> , id I'	p=oTablaSimb.EncuentraToken(Hash(id.lexema), id.lexema) p.Info.Clase = "variable" p.Info.Tipo = _tradVar._tipo p.Info.NoBytes = _tradVar._noBytes

La regla semántica `_tradVar._enTraduccion = true` establecida para las producciones 16 y 17 hace la tarea de señalar cuando el proceso de derivación de la sentencia está reconociendo la declaración de variables. Es necesario definir un atributo en la clase `TradVar` que nos permita registrar el hecho antes mencionado. A este atributo le llamaremos `_enTraduccion`. El proceso de reconocimiento de la declaración de variables siempre termina con el uso de la producción 18 según se indica en la figura #10.4, así que la regla semántica `_tradVar._enTraduccion = false` usada para la producción 18 indica que se acabó la traducción de declaración de variables.

Requerimos del atributo `_enTraduccion` debido a que las producciones 10, 11 y 12, también son usadas durante el proceso de derivación de la declaración de constantes. Estas producciones son utilizadas antes que las producciones 19 y 20, las cuales derivan a los identificadores `id` de las variables declaradas.

Entonces la traducción que estamos efectuando acerca del tipo y número de bytes de las variables declaradas, las enfocamos a definir el atributo `_tipo` dentro de la clase `TradVar` y de acuerdo al valor de este atributo, asignaremos el valor del número de bytes a registrar en la tabla de símbolos.

Hagamos la adición de los 2 atributos `_enTraduccion` y `_tipo` a la clase `TradVar` :

```
class TradVar : Traductor
{
    bool _enTraduccion;
    string _tipo;

    override public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
    {
        string idVar = noProd == 19 ? oAnaLex.Token[ae] : (noProd == 20 ? oAnaLex.Token[ae+1] : "");
        string idLex = noProd == 19 ? oAnaLex.Lexema[ae] : (noProd == 20 ? oAnaLex.Lexema[ae + 1] : "");
        ...
    }
}
```

Ahora agregamos las producciones 10, 11, 12, 16, 17 y 18, a la sentencia **switch ()** dentro del método **Analiza ()** de la clase **SintDescNRP**, figura #10.12.

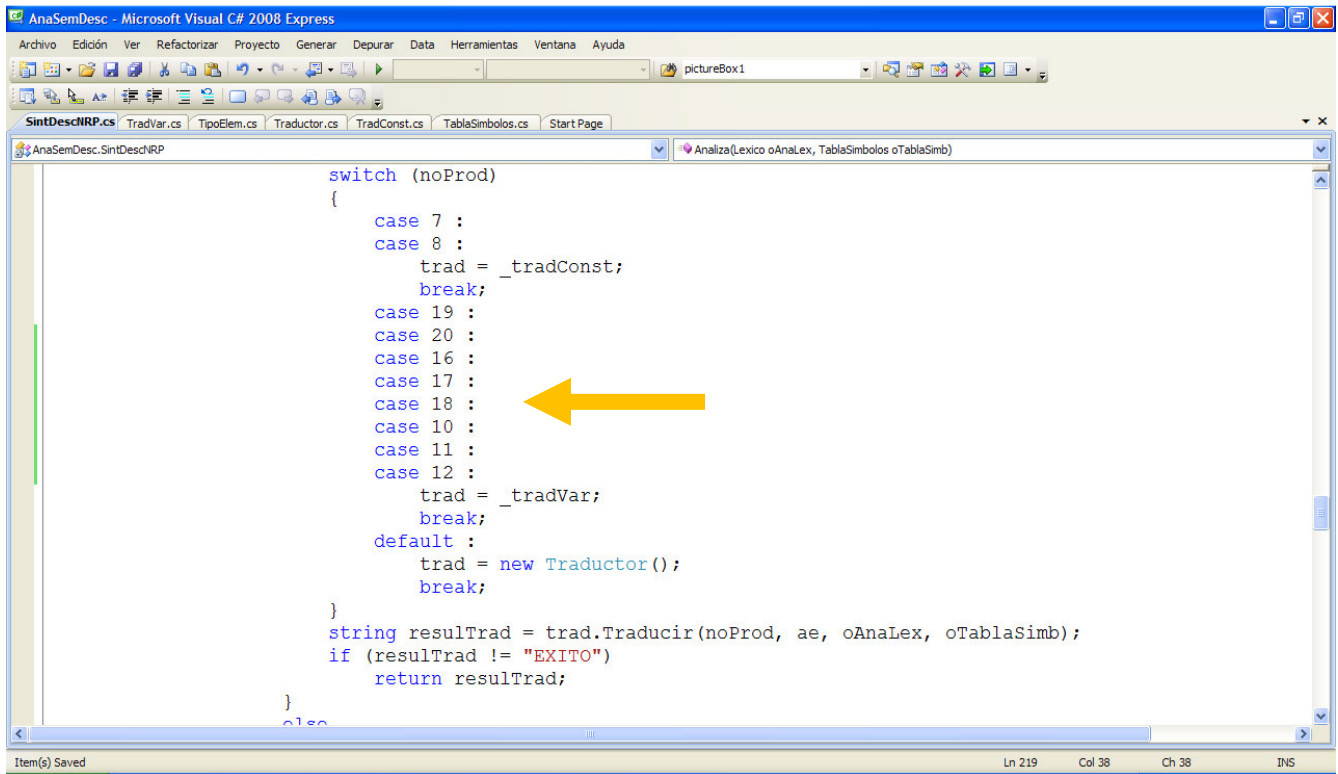


Fig. No. 10.12 Inclusión de los case para las producciones 16,17,18,10,11,12 para declaración de variables.

La forma de hacer la traducción implica efectuar decisiones dentro del método **Traducir ()** de la clase **TradVar**, con el fin de diferenciar el número de producción que es recibido por este método. El código para el método **Traducir ()** lo modificamos añadiendo una nueva sentencia **switch ()** que maneje el flujo de secuencia de ejecución.

```

override public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
{
    switch(noProd)
    {
        case 19 :
        case 20 :
            string idVar = noProd == 19 ? oAnaLex.Token[ae] : (noProd == 20 ? oAnaLex.Token[ae+1] : "");
            string idLex = noProd == 19 ? oAnaLex.Lexema[ae] : (noProd == 20 ? oAnaLex.Lexema[ae + 1] : "");
            if (idVar == "id")
            {
                char car = idLex.ToUpper()[0];
                int indice = Convert.ToInt32(car) - 65;
                Nodo p = oTablaSimb.EncuentraToken(indice, idLex);
                if (p == null)
                    return _resulTraduccion[1];
                else if (p.Info.Clase != "")
                    return p.Info.Clase == "constante" ? _resulTraduccion[4] : _resulTraduccion[5];
                else
                {
                    p.Info.Clase = "variable";
                    p.Info.Tipo = _tipo;
                    p.Info.NoBytes = CalcNoBytes(_tipo, "");
                    return _resulTraduccion[0];
                }
            }
            else
                return _resulTraduccion[1];
            break;
        case 16 :
    }
}
    
```

```

case 17 :
    _enTraduccion = true;
    return _resulTraduccion[0];
    break;
case 18 :
    _enTraduccion = false;
    return _resulTraduccion[0];
    break;
case 10 :
case 11 :
case 12 :
    _tipo = oAnaLex.Token[ae];
    return _resulTraduccion[0];
    break;
default :
    return _resulTraduccion[0];
    break;
}
}

```

Para las producciones 16 y 17 asignamos el valor de **true** al atributo `_enTraduccion` y para la producción 18 el atributo `_enTraduccion` es asignado al valor de **false**.

Cuando es utilizada una de las producciones 10, 11 o 12, el índice `ae` contiene la referencia al token en `w$` que contiene el tipo de la variable declarada correspondiente a la producción 19 o a la 20. De acuerdo a lo anterior justificamos la asignación :

```
_tipo = oAnaLex.Token[ae];
```

El atributo `_tipo` es registrado en la tabla de símbolos cuando es sustituida en la derivación de la sentencia la producción 19 o la 20. Lo mismo se hace con el *número de bytes* que es calculado con el método `CalcNoBytes()` heredado de la clase `Traductor`. Notemos que enviamos la cadena nula al parámetro `lexema` para el caso que el tipo sea cadena.

```

p.Info.Clase = "variable";
p.Info.Tipo = _tipo;
p.Info.NoBytes = CalcNoBytes(_tipo, "");
return _resulTraduccion[0];

```

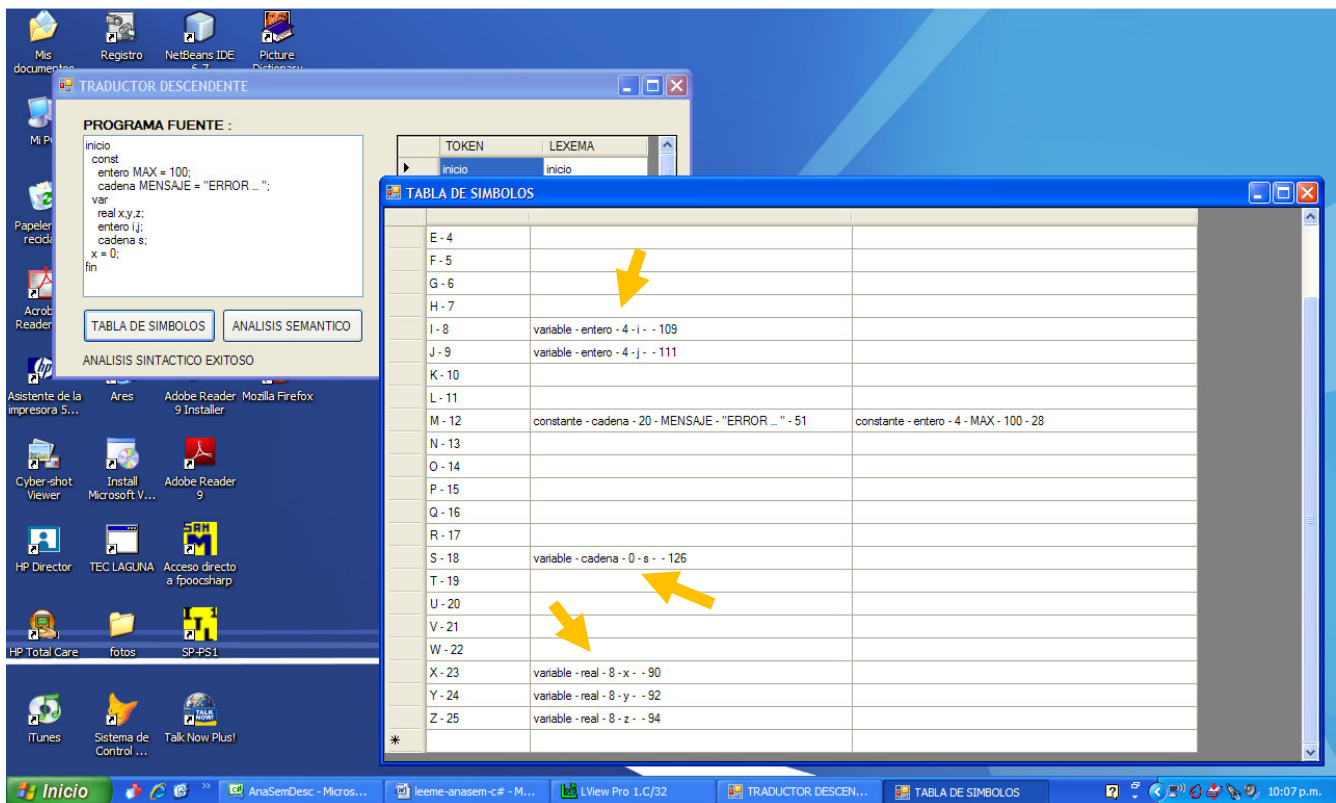


Fig. No. 10.13 Registro del tipo y número de bytes en la tabla de símbolos, para variables declaradas.

11 Traducción (c) : INDICAR ERROR SI UNA CONSTANTE O UNA VARIABLE ESTÁ DUPLICADA EN SU DECLARACIÓN.

Esta traducción ya la realizamos en las secciones anteriores 10 y 11.

12 Traducción (d) : INDICAR ERROR SI SE TRATA DE ASIGNAR A UNA CONSTANTE EL VALOR DE UNA EXPRESIÓN DENTRO DE UNA SENTENCIA DE ASIGNACIÓN.

Sigamos con nuestro ejemplo donde hemos escrito una sentencia de asignación $x = 0;$

```

inicio
const
    entero MAX = 100;
    cadena MENSAJE = "ERROR ... ";
var
    real x,y,z;
    entero i,j;
    cadena s;
x = 0;
fin
    
```

Lo que buscamos validar en la traducción de la sentencia, es que el identificador x no sea constante. La producción involucrada en esta traducción es la número 29 $A \rightarrow id = A'$ según se muestra en la simulación con el RD-NRP figura #12.1.

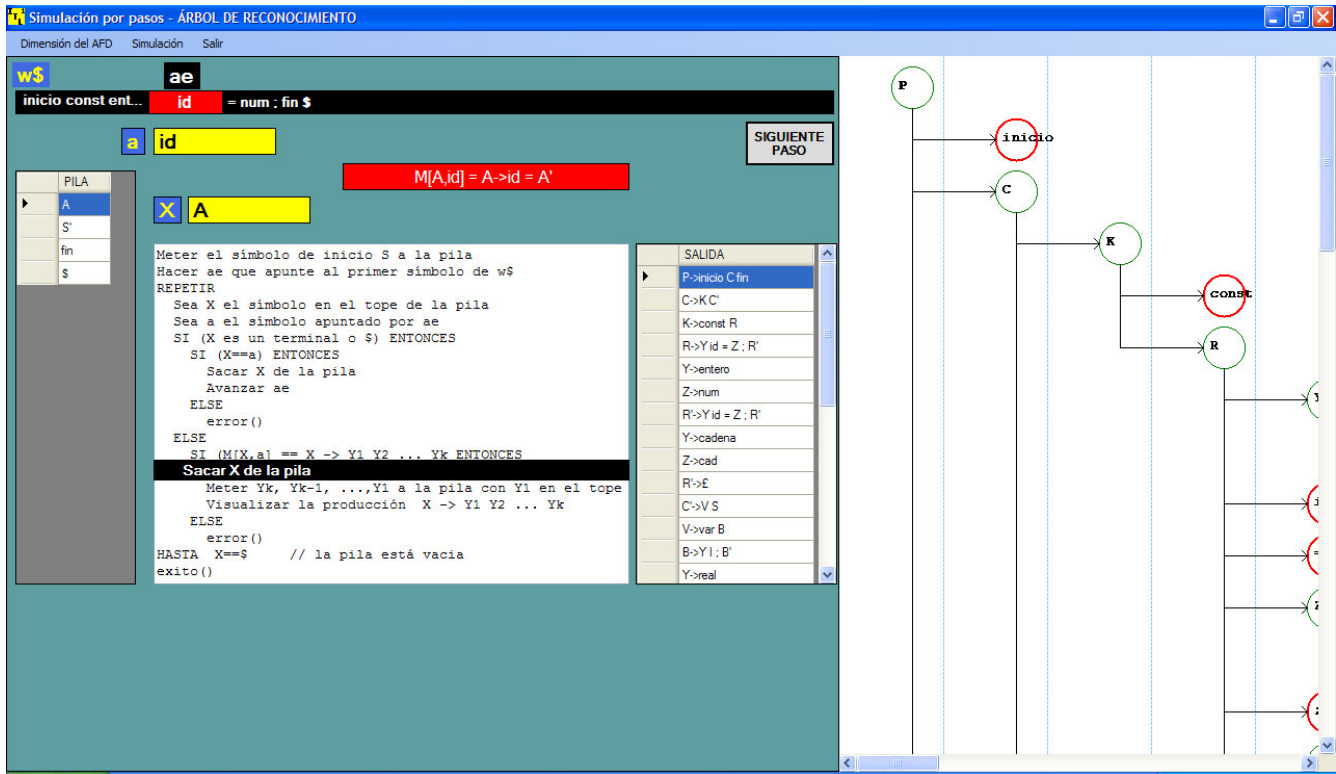


Fig. No. 12.1 Uso de la producción $A \rightarrow id = A'$

La traducción consistirá de comprobar que el identificador id en la producción $A \rightarrow id = A'$ tenga registrado el valor "variable" en el atributo $_clase$.

La figura #12.2 muestra el árbol de reconocimiento donde se ha utilizado la producción 29 $A \rightarrow id = A'$.

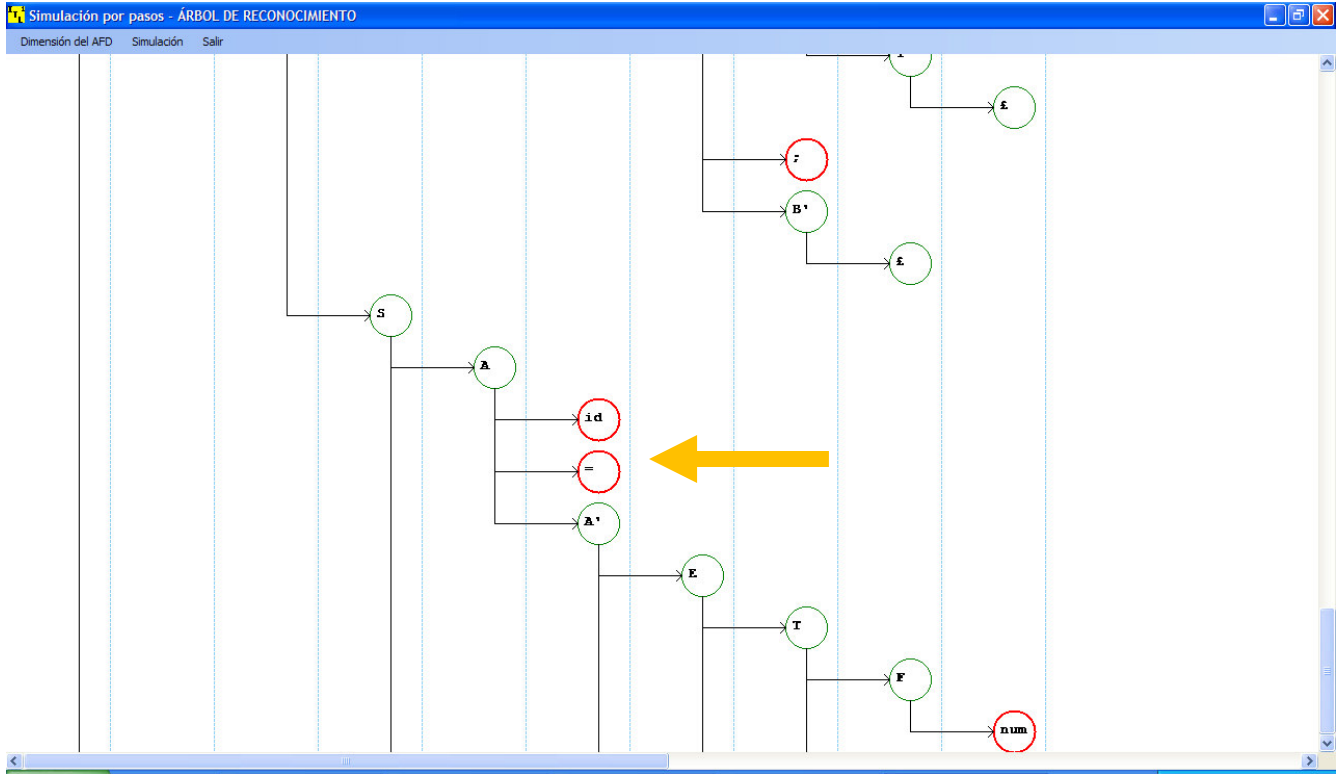


Fig. No. 12.2 Árbol de reconocimiento para la sentencia x = 0;

DEFINICIÓN DIRIGIDA POR SINTAXIS

noProd	Producción	Reglas semánticas
29	A -> id = A´	id.clase == "constante" ? Error() : Exito()

ESQUEMA DE TRADUCCIÓN (Versión 1)

noProd	Producción	Reglas semánticas
29	A -> id = A´	p=oTablaSimb.EncuentraToken(Hash(id.lexema), id.lexema) (p.Info.Clase == "constante") ? Error() : Exito()

Tanto la definición dirigida por sintaxis como el esquema de traducción son simples : prueban si el atributo `_clase` registrado en la tabla de símbolos para el identificador `id` es igual a "constante", si es así disparan un error, de lo contrario retornan éxito en la traducción.

Vayamos a la clase `SintDesNRP` para incluir el número de producción 29 dentro del `switch ()` que prepara la traducción :

```
switch (noProd)
{
    case 7 :
    case 8 :
        trad = _tradConst;
        break;
    case 19 :
    case 20 :
    case 16 :
    case 17 :
    case 18 :
    case 10 :
```

```

case 11 :
case 12 :
    trad = _tradVar;
    break;
case 29 :
    trad = _tradAsig;
    break;
default :
    trad = new Traductor();
    break;
}

```

Luego agregamos el código mostrado en la figura #12.3 encargado de la traducción, dentro del método Traducir() de la clase TradAsig.

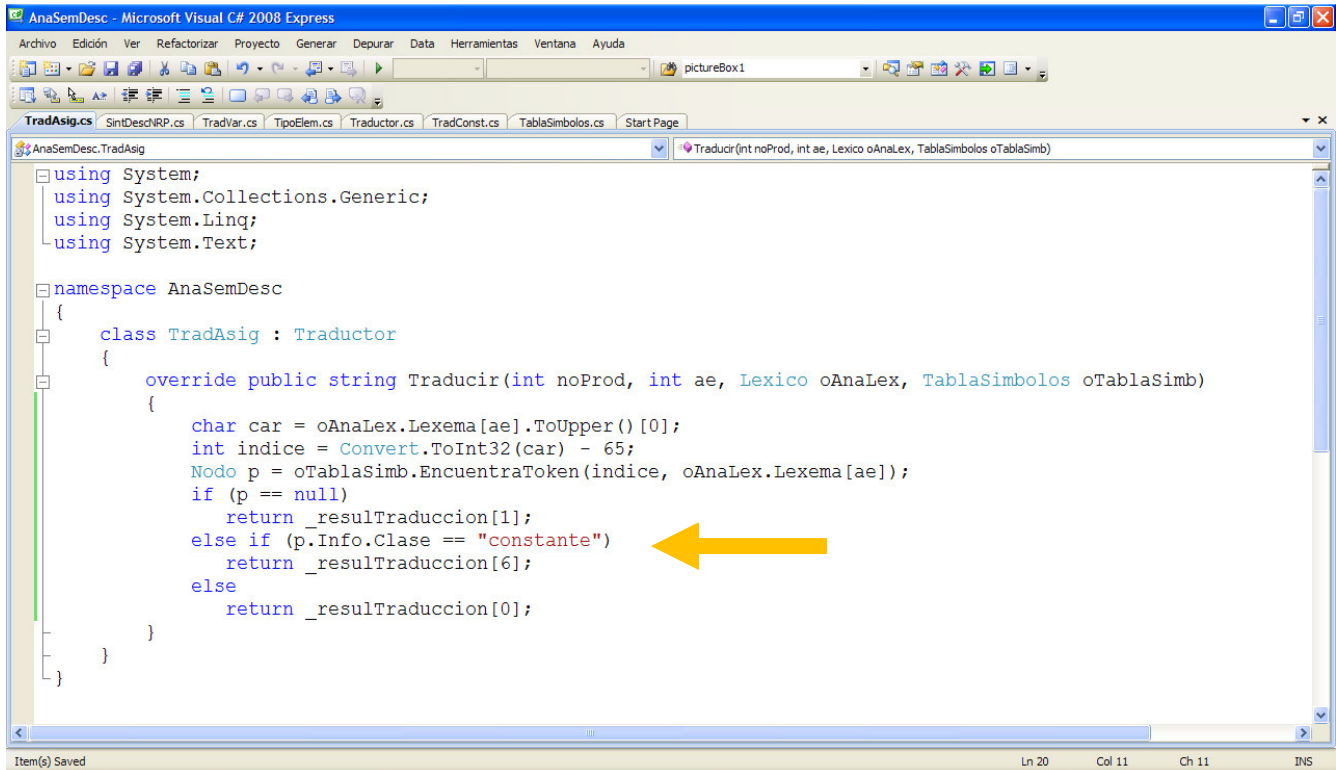


Fig. No. 12.3 Método Traducir() de la clase TradAsig.

Tenemos que agregar el nuevo error "ERROR ... UNA CONSTANTE NO PUEDE SER ASIGNADA" al que le corresponde el índice 6, dentro de la clase base Traductor :

```

class Traductor
{
    protected string[] _resulTraduccion = { "EXITO",
        "ERROR ... SE ESPERABA UN IDENTIFICADOR",
        "ERROR ... DECLARACION DE CONSTANTE DUPLICADA",
        "ERROR ... NO SE ESPECIFICA VALOR PARA LA CONSTANTE",
        "ERROR ... LA VARIABLE YA SE ENCUENTRA DECLARADA COMO
            CONSTANTE",
        "ERROR ... DECLARACION DE VARIABLE DUPLICADA",
        "ERROR ... UNA CONSTANTE NO PUEDE SER ASIGNADA"};

    virtual public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
    {
        return _resulTraduccion[0];
    }

    public int CalcNoBytes(string tipo, string lexema)
    {
        ...
    }
}

```

En la figura #12.4 ejecutamos la aplicación con el mismo ejemplo que hemos usado hasta ahora, donde incluimos una modificación a la sentencia de asignación que consiste en cambiar la x por la constante MAX.

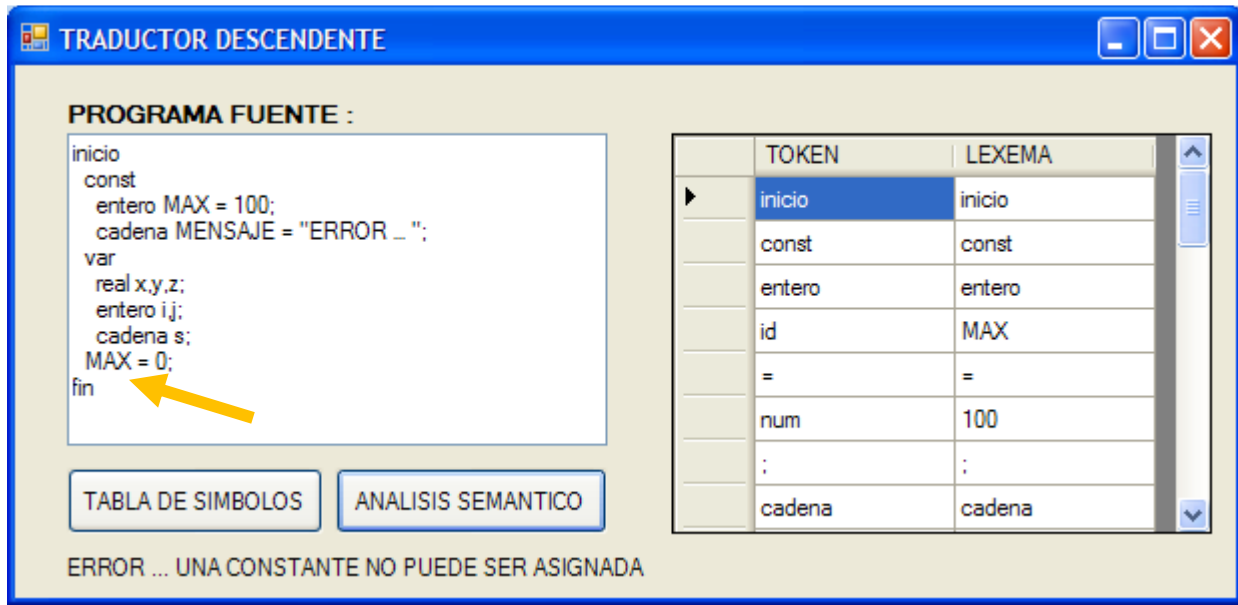


Fig. No. 12.4 Error 6 : "ERROR ... UNA CONSTANTE NO PUEDE SER ASIGNADA"

13 Traducción (e) : INDICAR ERROR SI UNA VARIABLE SE UTILIZA EN UNA EXPRESIÓN DE ASIGNACIÓN, SIN HABER SIDO DECLARADA.

Las producciones implicadas en esta traducción claramente son :

```
29 A -> id = A´    y
40 F -> id
```

La definición dirigida por sintaxis y el esquema de traducción son similares al ejercicio anterior :

DEFINICIÓN DIRIGIDA POR SINTAXIS

noProd	Producción	Reglas semánticas
29	A -> id = A´	id.clase == "" ? Error() : Exito()
40	F -> id	id.clase == "" ? Error() : Exito()


ESQUEMA DE TRADUCCIÓN (Versión 1)

noProd	Producción	Reglas semánticas
29	A -> id = A´	p=oTablaSimb.EncuentraToken(Hash(id.lexema), id.lexema) (p.Info.Clase == "") ? Error() : Exito()
40	F -> id	p=oTablaSimb.EncuentraToken(Hash(id.lexema), id.lexema) (p.Info.Clase == "") ? Error() : Exito()

Si un identificador de variable id no tiene instanciado su atributo _clase, entonces no ha sido declarada. El constructor de la clase TipoElem asigna la cadena nula al atributo _clase.


Entonces añadimos el nuevo error 7 "ERROR ... VARIABLE NO DECLARADA" a la clase Traductor :

```
class Traductor
{
    protected string[] _resulTraduccion = { "EXITO",
        "ERROR ... SE ESPERABA UN IDENTIFICADOR",
        "ERROR ... DECLARACION DE CONSTANTE DUPLICADA",
        "ERROR ... NO SE ESPECIFICA VALOR PARA LA CONSTANTE",
        "ERROR ... LA VARIABLE YA SE ENCUENTRA DECLARADA COMO
                                                    CONSTANTE",
        "ERROR ... DECLARACION DE VARIABLE DUPLICADA",
        "ERROR ... UNA CONSTANTE NO PUEDE SER ASIGNADA",
        "ERROR ... VARIABLE NO DECLARADA"};
```




Agregamos el número de producción 40 dentro del **switch (noProd)** del método Analiza() de la clase SintDescNRP :

```
switch (noProd)
{
    case 7 :
    case 8 :
        trad = _tradConst;
        break;
    case 19 :
    case 20 :
    case 16 :
    case 17 :
    case 18 :
    case 10 :
    case 11 :
    case 12 :
        trad = _tradVar;
        break;
    case 29 :
    case 40 :
        trad = _tradAsig;
        break;
    default :
        trad = new Traductor();
        break;
}
```




Luego modificamos el método Traducir() de la clase TradAsig de manera que contenga los cambios necesarios para esta nueva traducción –validación–.

```
class TradAsig : Traductor
{
    override public string Traducir(int noProd, int ae, Lexico oAnaLex, TablaSimbolos oTablaSimb)
    {
        char car = oAnaLex.Lexema[ae].ToUpper()[0];
        int indice = Convert.ToInt32(car) - 65;
        Nodo p = oTablaSimb.EncuentraToken(indice, oAnaLex.Lexema[ae]);
        if (p == null)
            return _resulTraduccion[1];
        else if (p.Info.Clase == "constante" && noProd==29)
            return _resulTraduccion[6];
        else if (p.Info.Clase == "")
            return _resulTraduccion[7];
        else
            return _resulTraduccion[0];
    }
}
```



Hemos modificado el código que valida que el id no sea una constante y que sea asignada, agregando la máscara del número de producción 29 ya que ahora entramos al método por la sustitución de 2 producciones : la 29 y la 40.

```
else if (p.Info.Clase == "constante" && noProd==29)
    return _resulTraduccion[6];
```



Lo demás se explica por si mismo :

```
else if (p.Info.Clase == "")
    return _resulTraduccion[7];
```



La ejecución que provoca el error de "VARIABLE NO DECLARADA" se muestra en las figuras #12.5 y figura #12.6.

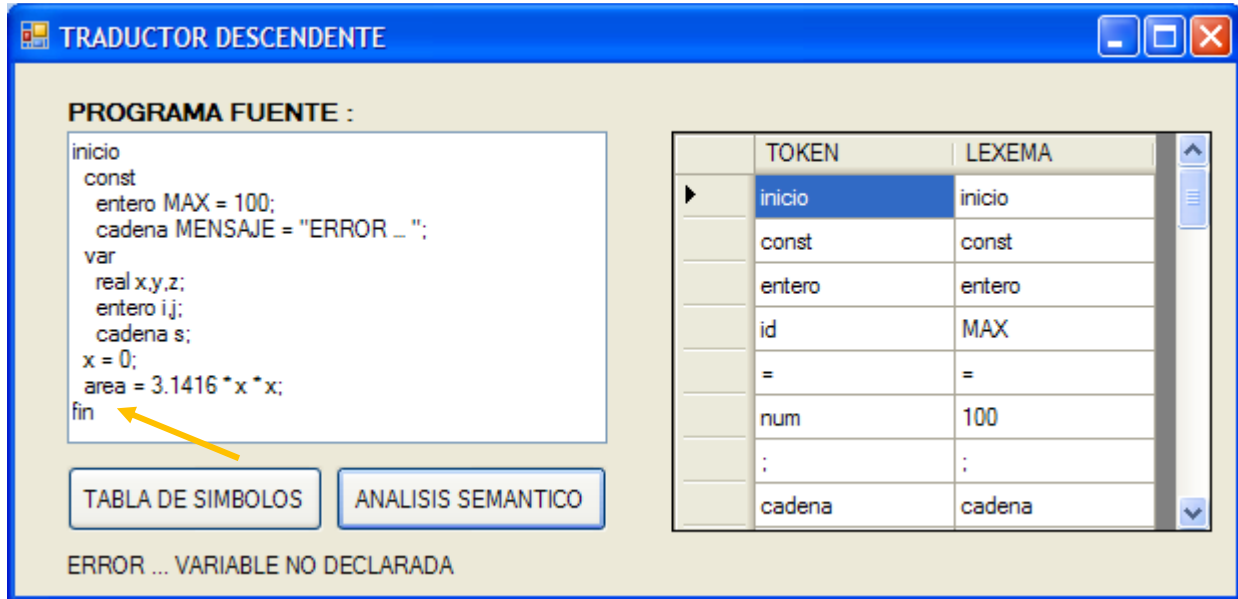


Fig. No. 12.5 Error 7 : "ERROR ... VARIABLE NO DECLARADA". La variable area no se declaró.

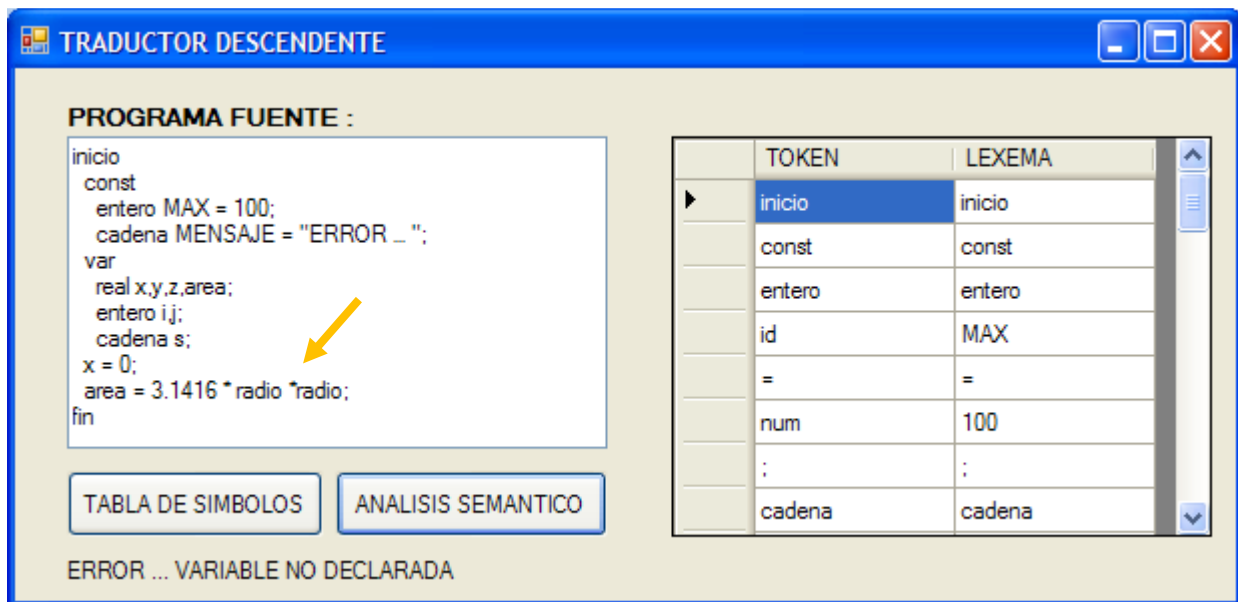


Fig. No. 12.6 Error 7 : "ERROR ... VARIABLE NO DECLARADA". La variable radio no se declaró.

La figura #12.7 muestra la corrección de las 2 variables area y radio que ya han sido declaradas, por lo que el análisis del programa fuente queda exento de error.

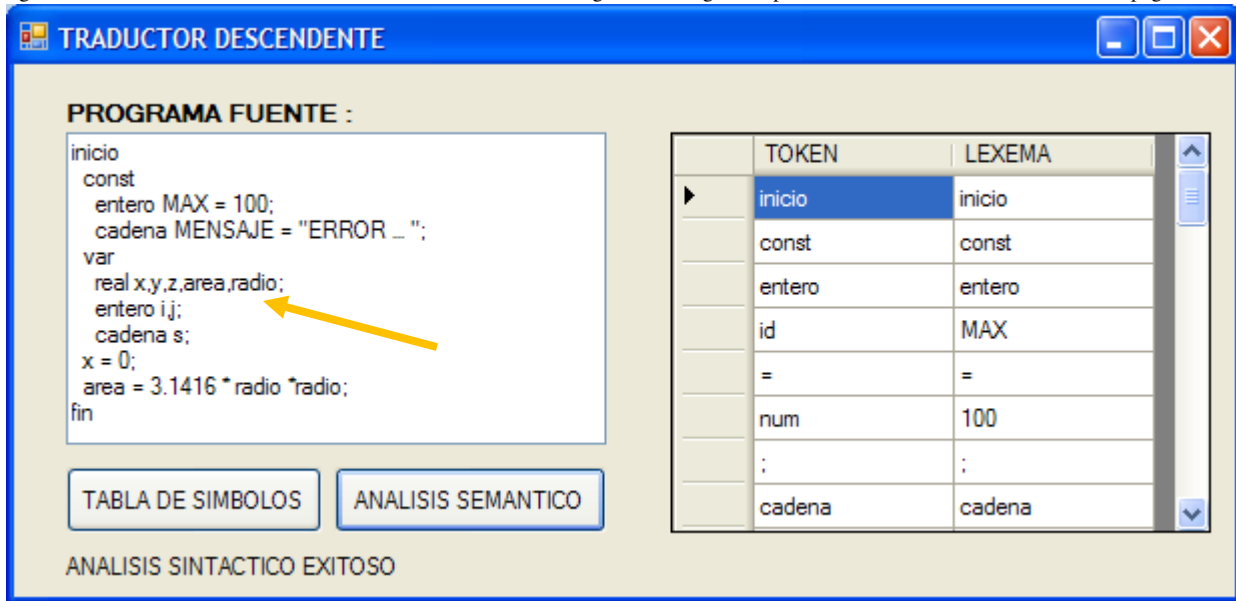


Fig. No. 12.7 Variables area y radio declaradas.

14 Traducción (f) : REVISAR EL TIPO EN LAS EXPRESIONES –SENTENCIAS- DE ASIGNACIÓN. QUE CORRESPONDAN LOS TIPOS DEL OPERANDO IZQUIERDO Y DEL DERECHO.

Esta traducción se la dejo a mis alumnos y lectores. Les puedo decir que no hay jiribilla, se resuelve aplicando los mismos conceptos vertidos en los ejercicios anteriores.