Generación de código en C# para un reconocedor sintactico ascendente.

Ing. Francisco Ríos Acosta
Instituto Tecnológico de la Laguna
Torreón,Coah; a 17 de diciembre del 2007.

I N D I C E.

1. INTRODUCCIÓN.

En las secciones siguientes voy a escribir acerca de 4 cuestiones fundamentales :

- Aplicación Windows C# típica para un reconocedor sintactico ascendente.
- Resúmen y prueba de la propuesta de clases R.A.F. para un analizador léxico. Construcción de un analizador léxico usando el código generado por *SP-PS1*.
- Mi propuesta de clases para un objeto reconocedor sintactico ascendente SLR, -propuesta R.A.F.-. Atributos, propiedades y métodos.
- Presentación y manejo del programa *RA-SLR* que genera código de acuerdo a lo tratado en el libro del dragón –Ulman- sobre un reconocedor sintactico ascendente SLR, tomando en cuenta también a la propuesta de clases de R.A.F.

Este esfuerzo de programación del software *RA-SLR*, comenzó durante el periodo de clases del semestre AGO-DIC del 2007. Agradezco a todos mis alumnos de la materia "*Programación de Sistemas*" de dicho periodo, ya que ellos me tuvieron mucha paciencia durante el desarrollo de los algoritmos presentados aquí. Todos los algoritmos que desarrollé en clase fueron probados por mis alumnos –cuestión que les agradezco-, retroalimentandome los errores que fueron corregídos en equipo maestro-alumno. Al final del curso, tomé la tarea de programar la aplicación *RA-SLR* que genera el código C# de 4 clases : *SintAscSLR, Item, Pila y SimbGram*. La clase *Pila* la tomamos de los apuntes de mi clase "Estructura de Datos".

2. INICIANDO LA APLICACIÓN WINDOWS C#.

Iniciaremos con una aplicación típica. que permite la entrada de un texto para luego analizarlo léxicamente finalizando con un análisis sintactico usando un reconocedor ascendente SLR.
La interfase gráfica de la aplicación es la mostrada en la figura #2.1.

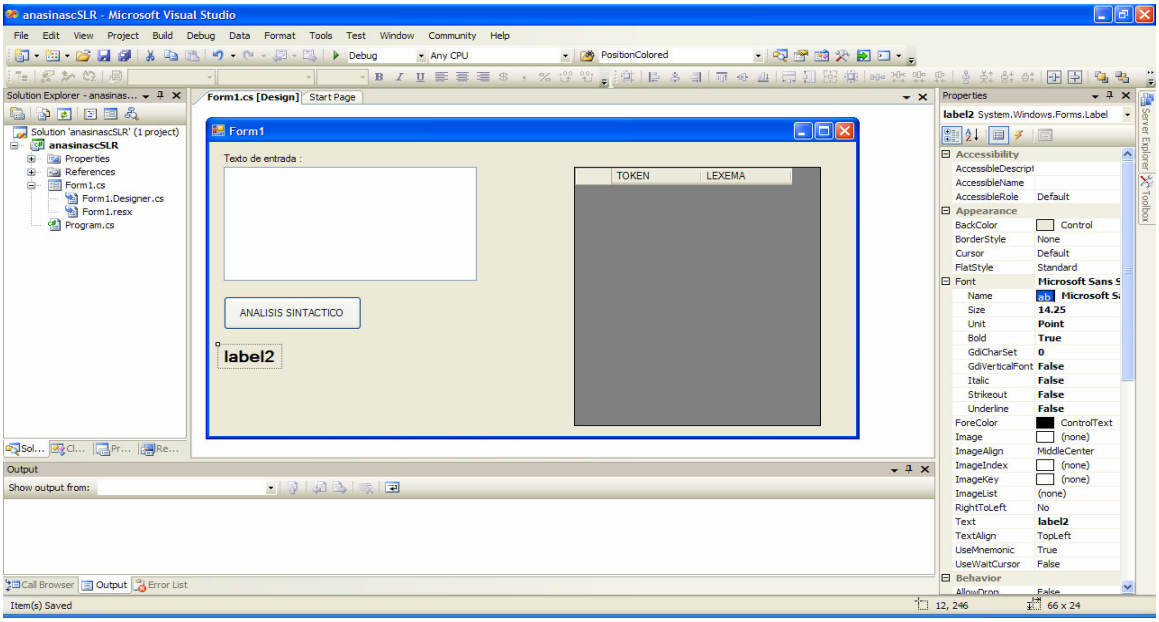


Fig. No. 2.1 Interfase gráfica típica para un análisis sintactico – ascendente SLR.

De la figura #2.1 podemos observar que visualizamos las parejas token-lexema reconocidas y almacenadas en un objeto al que llamaremos *oAnaLex*. Este objeto analiza lexicamente al texto de entrada. El componente utilizado para la visualización es el *dataGridView1*.
El componente *label2* es usado para visualizar un mensaje de análisis sintactico EXITOSO, o de otra forma un ERROR DE SINTAXIS.
La idea es que al hacer click sobre el componente *button1* con la propiedad Text = "ANALISIS SINTACTICO", realicemos un análisis léxico y luego un análisis sintactico (en adelante supondré que es lo mismo *análisis sintactico* que *reconocedor ascendente SLR*).
El código para el botón *button1* es –en **negritas**-:

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

        private void button1_Click(object sender, EventArgs e)
        {
            oAnaLex.Inicia();
            oAnaLex.Analiza(textBox1.Text);
            oAnaSintAscSLR.Inicia();
            if (oAnaSintAscSLR.Analiza(oAnaLex) == 0)
                label2.Text = "ANALISIS SINTACTICO EXITOSO";
            else
                label2.Text = "ERROR DE SINTAXIS";
        }
```

Notemos que usamos 2 objetos dentro del evento Click del *button1* : *oAnaLex* y *oAnaSintAscSLR*. Estos 2 objetos son definidos como atributos de la forma *Form1*.

objeto	clase	
oAnaLex	**Lexico**	Efectúa el análisis léxico del texto de entrada en el componente *textBox1*.
oAnaSintAscSLR	SintAscSLR	Efectúa el análisis sintactico del programa fuente cuyo análisis léxico ha sido previamente realizado.

Lo que sigue es definir las clases *Lexico* y *SintAscSLR*. La primera de ellas ya ha sido probada y la segunda, es tema que debemos desarrollar en este trabajo.

Desde luego que antes, tenemos que establecer que tipo de instrucciones vamos a reconocer con nuestro analizador sintactico ascendente. La gramática de contexto libre **G=(Vt, Vn, S, O)** que contiene la sintaxis de las sentencias, tiene los siguientes valores en sus componentes :

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

Vn = { "P", "C", "K", "R", "T", "V", "B", "L", "O", "A", "E", "M", "F", "U", "S", "I" };

S = "P"

O = { P->inicio C fin
C->K V O
C->K O
C->V O
C->O
K->const R
R->RTid=num;
R->RTid=cad;
R->Tid=num;
R->Tid=cad;
T->entero
T->real
T->cadena
V->var B
B->BTL;
B->TL;
L->L,id
L->id
O->OA
O->OU
O->OS
O->A
O->U
O->S
A->id=E;
E->E+M
E->E-M
E->M
M->M*F
M->M/F
M->F
F->id
F->num
F->(E)
U->leer id;
S->visua I;
I->I,id
I->I,cad
I->id
I->cad }

Recomiendo que el lector agrupe las producciones de la gramática. ¿Cuántas producciones tiene la gramática **G**?.

La gramática **G** reconoce sentencias de declaración de constantes, declaración de variables, de asignación, de entrada –lectura- y salida –visualización-. Todas estas sentencias existen en el ámbito acotado por una sentencia de **inicio** y una sentencia de **fin**. Un ejemplo es :

```
inicio
  const
    entero MAX=10;
    cadena mensaje=”VALORES FUERA DE RANGO”;
  var
    real x,y;
    entero i,j,k;
    cadena nombre;
  visua “teclea el valor de i : “;
  leer i;
  visua “teclea el valor de j : “;
  leer j;
  k = (i + j)*3;
  visua “la suma k = i + j es = “,k;
fin
```

El lector deberá de derivar a la derecha el ejemplo anterior, utilizando la gramática **G**.

La gramática **G** tiene restricciones y le corresponde al lector experimentarlas. La gramática sería mejor escribirla en notación *Backus-Naur*, sin embargo el alcance de estas propuestas no es suficiente.

He seleccionado mostrar los símbolos terminales *Vts* y los símbolos no terminales *Vns*, usando la notación de constante cadena –valor entre comillas-.

Además, el símbolo "$" lo incluí en los *Vts* aunque en realidad, este caracter no forma parte de los símbolos terminales.

Antes de hacer el análisis sintactico, debemos construir la clase *Lexico* (y la clase *Automata*) que permita definir al objeto *oAnaLex* en la aplicación sin error, que reconozca a todos los símbolos terminales –tokens- definidos en *Vt*, incluyendo al $.

3. CLASES LEXICO Y AUTOMATA.

Los atributos de la clase *Lexico* son :

```
class Lexico
    {
        const int TOKREC = 6;
        const int MAXTOKENS = 500;
        string[] _lexemas;
        string[] _tokens;
        string _lexema;
        int _noTokens;
        int _i;
        int _iniToken;
        Automata oAFD;

        // definición de métodos
    }
```

A continuación describo a los atributos de la clase *Lexico* :

TOKREC	Indica el número de tokens a reconocer. Para este ejemplo su valor es 6.
MAXTOKENS	Es una constante que limita el número de parejas token-lexema a reconocer. Para nuestro caso fueron suficientes 500.
_lexemas	Es un arreglo unidimensional de cadenas. Contiene cada uno de los lexemas reconocidos. Es dimensionado en el constructor de la clase.
_tokens	Es un arreglo unidimensional de cadenas. Contiene cada uno de los tokens reconocidos. Es dimensionado en el constructor de la clase.

_lexema	Es una cadena que contiene al lexema de un token reconocido en cada acción de aceptación que realice el analizador léxico.
_noTokens	Atributo entero que contiene el número de parejas token-lexema reconocidos durante el proceso del análisis léxico.
_i	Constituye un puntero al caracter i-ésimo del texto de entrada, que está leyendo para su análisis, el programa analizador léxico.
_iniToken	Es un puntero al primer caracter donde empieza el siguiente reconocimiento de la pareja token-lexema.
oAFD	Es un objeto que contiene la representación de los AFD's utilizados. Para nuestro caso deberán existir 6 autómatas.

La clase *Automata* :

```
class Automata
    {
        string _textoIma;
        int _edoAct;

        // definición de los métodos.
    }
```

Atributos de la clase *Automata* :

textoIma	Es una referencia a un string. Es asignado al valor del texto de entrada.
_edoAct	Permite saber en todo momento el valor del estado actual del AFD, que está siendo utilizado en un determinado momento, por el analizador léxico.

Usemos el *SP-PS1* para generar el código C#, definiendo las 6 expresiones regulares para los 6 AFD's que reconocen a los tokens siguientes :

1. *AFD* para el token **delim**.- Debemos reconocerlo pero no almacenar la pareja token-lexema.
2. AFD para el token **id** y **palres**.- Usaremos un solo AFD para reconocer a los identificadores y a las palabras reservadas. Las palabras reservadas son : *inicio, fin, const, entero, real, cadena, var, leer y visua*. Agreguemos el metodo *EsID()* a la clase *Lexico* que determina si se almacena al token **id** o a la palabra reservada –su lexema-. Para el caso de que el token sea **id**, **num**, o **cad**, guardaremos al nombre del token, de otra manera almacenamos al lexema en el arreglo *_tokens*. Para todos los casos, siempre guardamos en el arreglo *_lexemas*, el lexema encontrado.
3. Dos AFD's para el token **num**.- Debemos construir 2 AFD's para reconocer al token **num**, uno para reconocer a los reales y otro para reconocer a los enteros.
4. AFD para el token **cad**.- Aquí especificamos todas las secuencias de caracteres, encerradas entre comillas. La cadena nula también es reconocida por este AFD.
5. AFD para el token **otros**.- Este autómata sirve para reconocer a todos los demás símbolos terminales : `=, ;, ",", +, -, *, /, (, ).` La pareja que se almacena es lexema-lexema, es decir, en el arreglo *_tokens* almacenamos al lexema encontrado.

Agreguemos las 2 clases : *Lexico* y *Automata* a la aplicación Windos C#. Debemos rellenar las clases con el código que produce *SP-PS1*. Ensambla los 6 AFDs en el orden en que los enumeramos.

Recordemos que los tokens **id**, **num**, **delim**, requieren de un *Retraer()* en su AFD.

TOKEN	EXPRESION REGULAR	Retraer()	Guardamos
delim	{delim} -> [\ \n\r\t]+ [^\ \n\r\t]		nada
id	{Dig} -> [0-9] {Letra} -> [A-Za-z] {Sub} -> _ {id} -> {Letra} ({Letra} | {Dig} | {Sub}) * [^_A-Za-z0-9]	*	token-lexema para *id* lexema-lexema para *palres*
real	{real} -> ([0-9]+\.[0-9]+ | [0-9]+\. | \.[0-9]+) [^0-9]	*	token-lexema
entero	{entero} -> [0-9]+ [^0-9]	*	token-lexema
cad	{cad} -> \"[^\"]*\"		token-lexema
otros	{otros} -> = | ; | , | \+ | \- | * | / | \(| \)		lexema-lexema

El código generado por *SP-PS1* para las 2 clases *Lexico* y *Automata*, se muestra en la tabla siguiente :

Clase Lexico

```
class Lexico
 {
   const int TOKREC = 6;
   const int MAXTOKENS = 500;
   string[] _lexemas;
   string[] _tokens;
   string _lexema;
   int _noTokens;
   int _i;
   int _iniToken;
   Automata oAFD;

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

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

   public void Analiza(string texto)
   {
     bool recAuto;
     int noAuto;
     while (_i < texto.Length)
     {
       recAuto=false;
       noAuto=0;
       for(;noAuto<TOKREC&&!recAuto;)
         if(oAFD.Reconoce(texto,_iniToken,ref _i,noAuto))
           recAuto=true;
         else
           noAuto++;
       if (recAuto)
       {
```

```
          _lexema = texto.Substring(_iniToken, _i - _iniToken);
          switch (noAuto)
          {
            //--------------  Automata  delim--------------
           case 0 : _tokens[_noTokens] = "delim";
                        break;
            //--------------  Automata  Id--------------
           case 1 : _tokens[_noTokens] = "Id";
                        break;
            //--------------  Automata  real--------------
           case 2 : _tokens[_noTokens] = "real";
                        break;
            //--------------  Automata  entero--------------
           case 3 : _tokens[_noTokens] = "entero";
                        break;
            //--------------  Automata  cad--------------
           case 4 : _tokens[_noTokens] = "cad";
                        break;
            //--------------  Automata  otros--------------
           case 5 : _tokens[_noTokens] = "otros";
                        break;
          }
          _lexemas[_noTokens++] = _lexema;
        }
        else
          _i++;
        _iniToken = _i;
      }

    }

  }  // fin de la clase Lexico
```

Clase Automata

```
class Automata
{
  string _textoIma;
  int _edoAct;

  char SigCar(ref int i)
  {
    if (i == _textoIma.Length)
    {
      i++;
      return '\0';
    }
    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  real--------------
      case 2 : _edoAct = 6;
                   break;
       //--------------  Automata  entero--------------
      case 3 : _edoAct = 11;
                   break;
       //--------------  Automata  cad--------------
      case 4 : _edoAct = 14;
```

```
                    break;
       //--------------  Automata  otros--------------
      case 5 : _edoAct = 17;
                    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="!\"#$%&\'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~□€□,ƒ„…†‡ˆ‰Š‹Œ□Ž□□‘’“”•-
—˜™š›œ□žŸ ¡¢£¤¥¦§¨©ª«¬-®¯°±²³´µ¶·¸¹º»¼½¾¿\f").IndexOf(c)>=0) _edoAct=2;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 2 : i--;
                    return true;
                    break;
       //--------------  Automata  Id--------------
      case 3 : c=SigCar(ref i);
                    if ((lenguaje="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz").IndexOf(c)>=0)
_edoAct=4;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 4 : c=SigCar(ref i);
                    if ((lenguaje="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz").IndexOf(c)>=0)
_edoAct=4;  else
                    if ((lenguaje="0123456789").IndexOf(c)>=0)  _edoAct=4;  else
                    if ((lenguaje="_").IndexOf(c)>=0)  _edoAct=4;  else
                    if ((lenguaje=" !\"#$%&\'()*+,-./:;<=>?@[\\]^`{|}~□€□,ƒ„…†‡ˆ‰Š‹Œ□Ž□□‘’“”•——
˜™š›œ□žŸ ¡¢£¤¥¦§¨©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  real--------------
      case 6 : c=SigCar(ref i);
                    if ((lenguaje="0123456789").IndexOf(c)>=0)  _edoAct=7;  else
                    if ((lenguaje=".").IndexOf(c)>=0)  _edoAct=8;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 7 : c=SigCar(ref i);
                    if ((lenguaje="0123456789").IndexOf(c)>=0)  _edoAct=7;  else
                    if ((lenguaje=".").IndexOf(c)>=0)  _edoAct=9;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 8 : c=SigCar(ref i);
                    if ((lenguaje="0123456789").IndexOf(c)>=0)  _edoAct=9;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 9 : c=SigCar(ref i);
                    if ((lenguaje="0123456789").IndexOf(c)>=0)  _edoAct=9;  else
                    if ((lenguaje=".").IndexOf(c)>=0) _edoAct=10;  else
                    if ((lenguaje=" !\"#$%&\'()*+,-
/:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~□€□,ƒ„…†‡ˆ‰Š‹Œ□Ž□□‘’“”•——
˜™š›œ□žŸ ¡¢£¤¥¦§¨©ª«¬-®¯°±²³´µ¶·¸¹º»¼½¾¿\n\t\r\f").IndexOf(c)>=0) _edoAct=10;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 10 : i--;
                    return true;
```

```
                    break;
       //--------------  Automata  entero--------------
      case 11 : c=SigCar(ref i);
                    if ((lenguaje="0123456789").IndexOf(c)>=0)  _edoAct=12;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 12 : c=SigCar(ref i);
                    if ((lenguaje="0123456789").IndexOf(c)>=0)  _edoAct=12;  else
                    if ((lenguaje=" !\"#$%&\'()*+,-
./:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~□€□,ƒ„…†‡ˆ‰Š‹Œ□Ž□□‘’“”•––
˜™š›œ□žŸ ¡¢£¤¥¦§¨©ª«¬-®¯°±²³´µ¶·¸¹º»¼½¾¿\n\t\r\f").IndexOf(c)>=0) _edoAct=13;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 13 : i--;
                    return true;
                    break;
       //--------------  Automata  cad--------------
      case 14 : c=SigCar(ref i);
                    if ((lenguaje="\"").IndexOf(c)>=0)  _edoAct=15;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 15 : c=SigCar(ref i);
                    if ((lenguaje="\"").IndexOf(c)>=0)  _edoAct=16;  else
                    if ((lenguaje=" !#$%&\'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~□€□,ƒ„…†‡ˆ‰Š‹Œ□Ž□□‘’“”•–
—˜™š›œ□žŸ ¡¢£¤¥¦§¨©ª«¬-®¯°±²³´µ¶·¸¹º»¼½¾¿\n\t\r\f").IndexOf(c)>=0)  _edoAct=15;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 16 : return true;
                    break;
       //--------------  Automata  otros--------------
      case 17 : c=SigCar(ref i);
                    if ((lenguaje="=").IndexOf(c)>=0)  _edoAct=18;  else
                    if ((lenguaje=";").IndexOf(c)>=0)  _edoAct=18;  else
                    if ((lenguaje=",").IndexOf(c)>=0)  _edoAct=18;  else
                    if ((lenguaje="+").IndexOf(c)>=0)  _edoAct=18;  else
                    if ((lenguaje="-").IndexOf(c)>=0)  _edoAct=18;  else
                    if ((lenguaje="*").IndexOf(c)>=0)  _edoAct=18;  else
                    if ((lenguaje="/").IndexOf(c)>=0)  _edoAct=18;  else
                    if ((lenguaje="(").IndexOf(c)>=0)  _edoAct=18;  else
                    if ((lenguaje=")").IndexOf(c)>=0)  _edoAct=18;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 18 : return true;
                    break;
       }
     switch (_edoAct)
     {
            case 2 :      // Autómata  delim
            case 5 :      // Autómata  Id
            case 10 :      // Autómata  real
            case 13 :      // Autómata  entero
                           --i;
                           return true;
     }
     return false;
  }

} // fin de la clase Automata
```

Si aún no lo habias hecho, agrega a las clases *Lexico* y *Automata* de tu aplicación C#, el código generado por *SP-PS1*.

4. PRUEBA DE FUNCIONAMIENTO DEL ANÁLIZADOR LÉXICO oAnaLex.

Si ejecutamos la aplicación C# tendremos errores. Además debemos efectuar los cambios :

- Evitar almacenar a los delimitadores,
- Reconocer a las palabras reservadas con el mismo AFD que se usa para el token **id**.
- También debemos guardar el lexema en lugar del token, para cuando reconocemos a los caracteres que agrupamos como **otros**.
- Otra consideración es juntar el reconocimiento de los token **real** y **entero**, y agruparlos en el reconocimiento del token **num**.

Primero evitemos el error en la compilación, situemos entre comentarios a las llamadas y definiciones sobre el objeto *oAnaSinAscSLR*. Entremos al código de la forma *Form1* t hagamos los cambios mostrados en el listado siguiente :

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

        private void button1_Click(object sender, EventArgs e)
        {
            oAnaLex.Inicia();
            oAnaLex.Analiza(textBox1.Text);
     /*       oAnaSintAscSLR.Inicia();
            if (oAnaSintAscSLR.Analiza(oAnaLex) == 0)
                label2.Text = "ANALISIS SINTACTICO EXITOSO";
            else
                label2.Text = "ERROR DE SINTAXIS"; */
        }
    }
```

Ejecutemos la aplicación y debemos de obtener la ventana de la aplicación, fig#4.1.

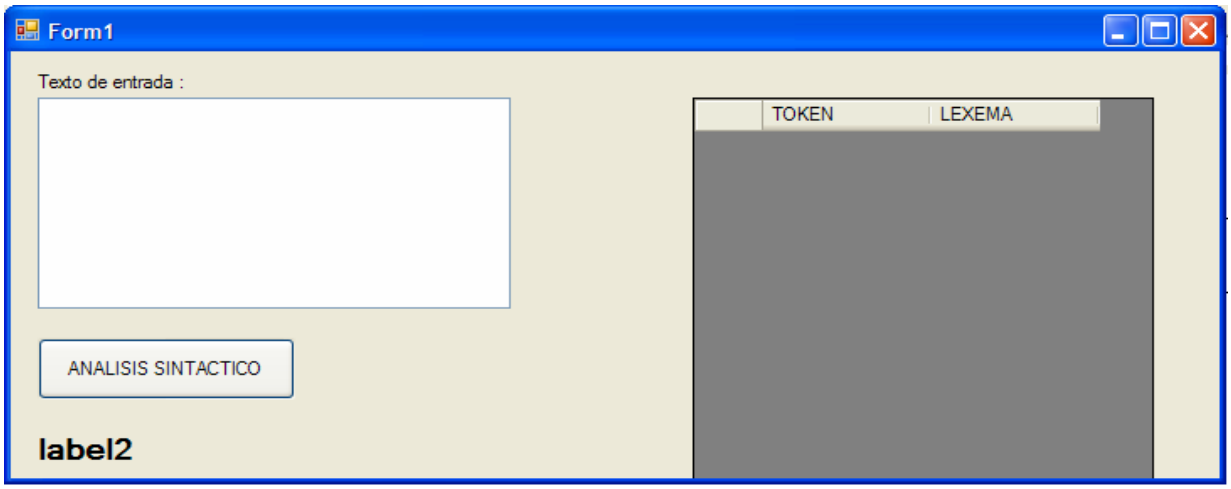


Fig. No. 4.1 Aplicación C# en ejecución, después de la compilación con comentarios.

Ahora empecemos a programar los cambios antes mencionados. Modifiquemos el método *Analiza()* de la clase *Lexico* según se muestra enseguida :

```
public void Analiza(string texto)
        {
            bool recAuto;
            int noAuto;
            while (_i < texto.Length)
            {
                recAuto = false;
                noAuto = 0;
                for (; noAuto < TOKREC && !recAuto; )
                    if (oAFD.Reconoce(texto, _iniToken, ref _i, noAuto))
                        recAuto = true;
                    else
                        noAuto++;
                if (recAuto)
                {
                    _lexema = texto.Substring(_iniToken, _i - _iniToken);
                    switch (noAuto)
                    {
                        //--------------  Automata  delim-------------
                        case 0: // _tokens[_noTokens] = "delim";
                            break;
                        //--------------  Automata  Id--------------
                        case 1: _tokens[_noTokens] = "Id";
                            break;
                        //--------------  Automata  real--------------
                        case 2: _tokens[_noTokens] = "real";
                            break;
                        //--------------  Automata  entero--------------
                        case 3: _tokens[_noTokens] = "entero";
                            break;
                        //--------------  Automata  cad--------------
                        case 4: _tokens[_noTokens] = "cad";
                            break;
                        //--------------  Automata  otros--------------
                        case 5: _tokens[_noTokens] = "otros";
                            break;
                    }
                    if (noAuto!=0)
                       _lexemas[_noTokens++] = _lexema;
                }
                else
                    _i++;
                _iniToken = _i;
            }

        }
```

La primera flecha señala que hemos puesto entre comentarios la sentencia que almacena el token **delim**. La segunda flecha, señala que hemos condicionado el almacenamiento del lexema, sólo para cuando el *noAuto* que se reconoció es diferente al **delim** (*noAuto!=0*). Lo anterior condiciona a que el delimitador sea el token reconocido por el primer autómata, *noAuto==0*.

Para reconocer a las palabras reservadas usando el AFD para el token **id**, debemos modificar el método *Analiza()* de la clase *Lexico*. También debemos agregar un nuevo método a la clase *Lexico*, el método *EsId()*. El método *EsId()* retorna un *true* si el lexema encontrado es un identificador, retorna *false* si el lexema es una palabra reservada registrada en el arreglo local *palRes*.

La definición del método *EsId()* la muestro para las palabras reservadas de nuestro lenguaje de ejemplo. Debemos añadirla dentro de la clase *Lexico*. Házlo.

```
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;
        }
```

Ahora insertemos la llamada dentro del método *Analiza()* :

```
public void Analiza(string texto)
        {
            bool recAuto;
            int noAuto;
            while (_i < texto.Length)
            {
                recAuto = false;
                noAuto = 0;
                for (; noAuto < TOKREC && !recAuto; )
                    if (oAFD.Reconoce(texto, _iniToken, ref _i, noAuto))
                        recAuto = true;
                    else
                        noAuto++;
                if (recAuto)
                {
                    _lexema = texto.Substring(_iniToken, _i - _iniToken);
                    switch (noAuto)
                    {
                        //--------------  Automata  delim--------------
                        case 0: // _tokens[_noTokens] = "delim";
                            break;
                        //--------------  Automata  Id--------------
                        case 1: if (EsId())
                                    _tokens[_noTokens] = "id";
                                else
                                    _tokens[_noTokens] = _lexema;
                            break;
                        //--------------  Automata  real--------------
                        case 2: _tokens[_noTokens] = "real";
                            break;
                        //--------------  Automata  entero--------------
                        case 3: _tokens[_noTokens] = "entero";
                            break;
                        //--------------  Automata  cad--------------
                        case 4: _tokens[_noTokens] = "cad";
                            break;
                        //--------------  Automata  otros--------------
                        case 5: _tokens[_noTokens] = "otros";
                            break;
                    }
                    if (noAuto!=0)
                       _lexemas[_noTokens++] = _lexema;
                }
                else
                    _i++;
                _iniToken = _i;
            }

        }
```

Si el lexema es una palabra reservada, almacenamos el lexema en el arreglo *_tokens*

Otra cuestión es modificar el método *Analiza()* de la clase *Lexico* para que almacene una pareja lexema-lexema, cuando el token es cualquier símbolo definido en la expresión regular, **otros**.
La modificación la muestro en el listado siguiente :

```
public void Analiza(string texto)
        {
            bool recAuto;
            int noAuto;
```

```
            while (_i < texto.Length)
            {
                recAuto = false;
                noAuto = 0;
                for (; noAuto < TOKREC && !recAuto; )
                    if (oAFD.Reconoce(texto, _iniToken, ref _i, noAuto))
                        recAuto = true;
                    else
                        noAuto++;
                if (recAuto)
                {
                    _lexema = texto.Substring(_iniToken, _i - _iniToken);
                    switch (noAuto)
                    {
                        //--------------  Automata  delim--------------
                        case 0: // _tokens[_noTokens] = "delim";
                            break;
                        //--------------  Automata  Id--------------
                        case 1: if (EsId())
                                    _tokens[_noTokens] = "id";
                                else
                                    _tokens[_noTokens] = _lexema;
                            break;
                        //--------------  Automata  real--------------
                        case 2: _tokens[_noTokens] = "real";
                            break;
                        //--------------  Automata  entero--------------
                        case 3: _tokens[_noTokens] = "entero";
                            break;
                        //--------------  Automata  cad--------------
                        case 4: _tokens[_noTokens] = "cad";
                            break;
                        //--------------  Automata  otros--------------
                        case 5: _tokens[_noTokens] = _lexema;
                            break;
                    }
                    if (noAuto!=0)
                       _lexemas[_noTokens++] = _lexema;
                }
                else
                    _i++;
                _iniToken = _i;
            }

        }
```

Guardamos el atributo *_lexema* que tiene el valor del lexema encontrado.

La última cuestión por modificar es el agrupar los tokens **real** y **entero**, en un solo token llamado **num**. A continuación tenemos el listado que contiene la modificación.

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

```
                                _tokens[_noTokens] = "id";
                            else
                                _tokens[_noTokens] = _lexema;
                        break;
                    //--------------  Automata  real--------------
                    case 2:
                    //--------------  Automata  entero--------------
                    case 3: _tokens[_noTokens] = "num";
                        break;
                    //--------------  Automata  cad--------------
                    case 4: _tokens[_noTokens] = "cad";
                        break;
                    //--------------  Automata  otros--------------
                    case 5: _tokens[_noTokens] = _lexema;
                        break;
                }
                if (noAuto!=0)
                   _lexemas[_noTokens++] = _lexema;
            }
            else
                _i++;
            _iniToken = _i;
        }

    }
```

Eliminamos una sentencia y modificamos el valor del token a **num**.

Finalmente para hacer la prueba sobre el funcionamiento del analizador léxico, agergamos el código en la clase *Form1* del archivo *Form1.cs*.
Este código insertado dentro del evento Click del *button1* –ANALISIS SINTACTICO-, lo usamos para visualizar en el *dataGridView1* las parejas token-lexema reconocidas durante el análisis léxico hecho por el objeto *oAnaLex*.

```
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];
            }
     /*       oAnaSintAscSLR.Inicia();
            if (oAnaSintAscSLR.Analiza(oAnaLex) == 0)
                label2.Text = "ANALISIS SINTACTICO EXITOSO";
            else
                label2.Text = "ERROR DE SINTAXIS"; */
        }
```

Observa que usamos 3 propiedades de la clase Lexico :

- *NoTokens*.- retorna el número de parejas token-lexema que se reconocieron en el análisis.
- *Token*.- retorna al arreglo_tokens.
- *Lexema*.- retorna al arreglo _lexemas.

Agrega la definición de estas propiedades a la clase *Lexico*.

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

        public string[] Lexema
        {
            get { return _lexemas; }
```

```
        }

        public string[] Token
        {
            get { return _tokens; }
        }
```

En la figura #4.2 tenemos la visualización de las parejas token-lexema encontrados durante el análisis léxico hecho por el objeto *oAnaLex*.

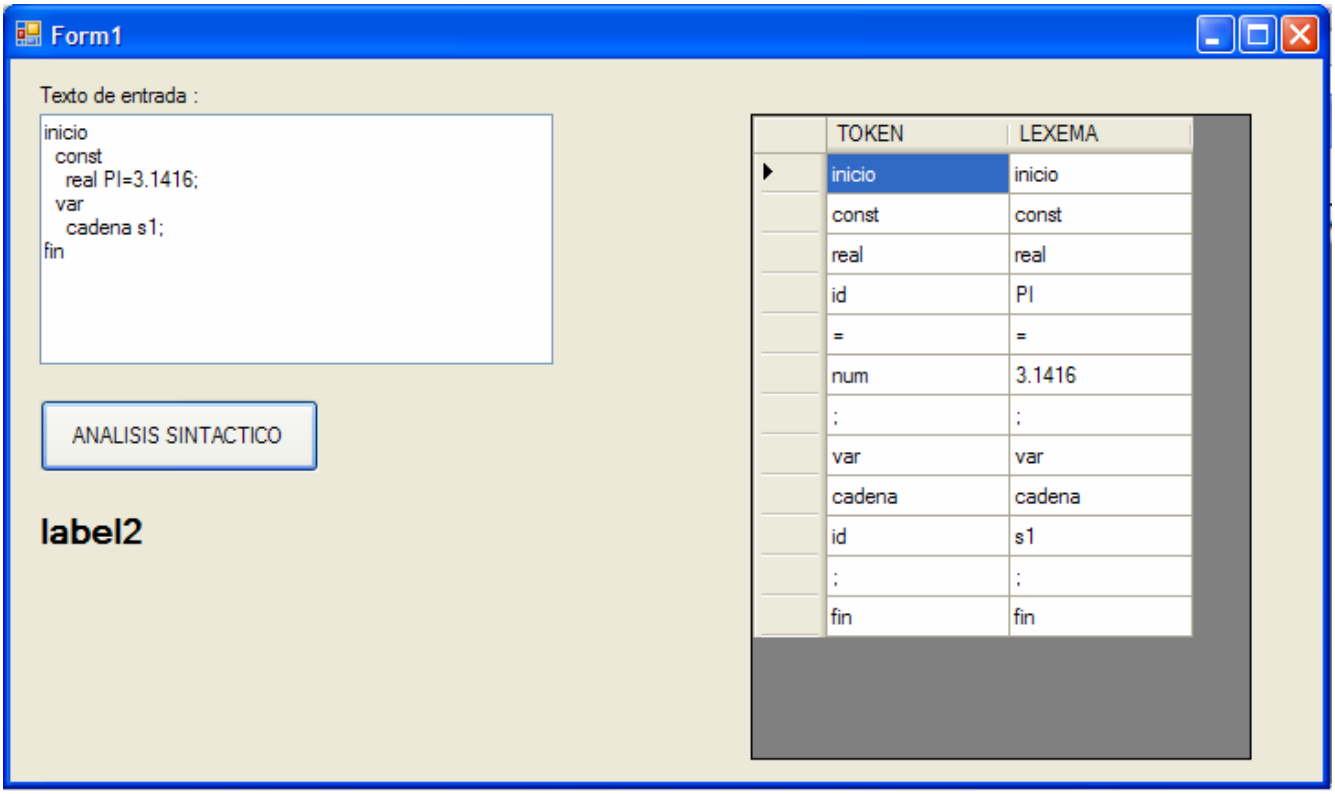


Fig. No. 4.2 Prueba del buen funcionamiento del analizador léxico *oAnaLex*.

5. CLASES PROPUESTAS.

La propuesta de clases que propongo de acuerdo a la teoría presentada por el libro del dragón, son las enumeradas a continuación :

- class *SintAscSLR*
- class *Item*
- class *Pila*
- class *SimbGram*

La clase *SintAscSLR* es la fundamental. Esta clase hace uso de objetos pertenecientes a las otras clases. La clase *Item* es usada para representar a los conjuntos de items que forman parte de la colección canónica de items. La colección canónica es un atributo de la clase *SintAscSLR*, y representa la base para la construcción de la tabla M de reconocimiento.

En el código –clases- que genera el programa *RA-SLR*, la tabla M no existe. Esta tabla es representada por los atributos *_action* y *_goTo*.

Las clase *Pila* es necesaria, para representar a la pila que encontramos en el modelo conceptual del analizador sintactico ascendente SLR.

Los elementos que contiene la pila son símbolos gramaticales y estados. Tanto los símbolos gramaticales como los estados, son objetos pertenecientes a la clase *SimbGram*.

En la sección siguiente trataremos acerca de los atributos de las clases *SintAscSLR* e *Item*. Respecto a las clases *Pila* y *SimbGram*, su explicación es trivial y se deja al lector su análisis.

6. ATRIBUTOS EN LAS CLASES *SintAscSLR* e *Item* PARA RECONOCEDORES ASCENDENTES *SLR, PROPUESTA POR R.A.F.*

Mi propuesta de atributos para la clase *SintAscSLR* es muy simple. Se basa en la teoría expresada en el libro del dragón, en su capítulo 4 tema análisis sintactico ascendente SLR.

```
class SintAscSLR
    {
        public const int NOPROD = 40;
        public const int NODDS = 1000;
        public const int NOACTIONS = 1000;
        public const int NOGOTOS = 1000;
        string[] _vts;
        string[] _vns;
        int[,] _prod;
        int[,] _sig;
        Pila _pila;
        int[,] _action;
        int _noActions;
        int _noGoTos;
        int[,] _goTo;
        int[] _dd;
        int _noDds;
        Item [] _c;
        int _noItems;

        //métodos de la clase

    }
```

La constante NOPROD corresponde al número de producciones de la gramática de contexto libre, que tiene la sintaxis de las sentencias a reconocer.

Descripción de atributos de la clase *SintAscSLR*.

ATRIBUTO	DEFINICIÓN	
NOPROD	`public const int NOPROD = 40;`	Constante que representa el número de producciones de la gramática.
NODDS	`public const int NODDS = 1000;`	Indica el número de producciones máximo para una derivación a la derecha de salida del reconocedor, para una sentencia dada.
NOACTIONS	`const int NOACTIONS = 1000;`	Se usa para dimensionar los renglones del arreglo *_action*.
NOGOTOS	`const int NOGOTOS = 1000;`	Se usa para dimensionar los renglones del arreglo *_goTo*.
_vts	`string[] _vts;`	Arreglo unidimensional de cadenas. Contiene a todos los símbolos terminales Vts de la gramática.

_vns	`string[] _vns;`	Es un arreglo de cadenas de una dimensión, cuyos elementos son los símbolos no terminales Vns –variables sintacticas- de la gramática.
_prod	`int[,] _prod;`	Arreglo de dos dimensiones. Sus elementos son enteros. Cada uno de los renglones del arreglo, representa una producción de la gramática. Cada columna representa al miembro izquierdo de la producción, el número de Y's de la producción, y cada Yi del miembro derecho de la producción A -> Y_1 Y_2 … Y_n
_sig	`int[,] _sig;`	Arreglo bidimensional de enteros. Cada renglón contiene los siguientes –conjuntos de tokens- de cada una de las variables sintacticas de la gramática Vns.
_pila;	`Pila _pila;`	Objeto que representa a la pila usada en el modelo de un reconocedor ascendente.
_action	`int[,] _action;`	Representa a las acciones de la tabla de reconocimiento. Cada renglón del arreglo bidimensional es una acción para un cierto estado. Sus 4 columnas representan : el estado, el token –símbolo terminal-, el tipo de acción -0 cambio, 1 reducción, 2 acceptación-, y el número de producción cuando es reducción.
_noActions	`int _noActions;`	Número de acciones registradas en el arreglo *_action*.
_goTo	`int[,] _goTo;`	Representa la parte del GOTO de la tabla de reconocimiento. Cada renglón del arreglo bidimensional es un goto de un estado a otro, dado un símbolo no terminal. Las 3 columnas representan : el estado, el índice del símbolo gramatical no terminal, y el estado al que transiciona –GOTO-.
_noGoTos	`int _noGoTos;`	Número de GOTOs registrados en el arreglo *_goTo*.
_dd	`int[] _dd;`	Arreglo unidimensional cuyos elemenots enteros corresponden al número de producción usada en la derivación a la derecha de la sentencia.
_noDds	`int _noDds;`	Número de producciones registradas en el arreglo *_dd*.
_c	`Item [] _c;`	Arreglo de objetos de la clase *Item*. Representa la colección canónica de items usada por un reconocedor ascendente SLR. Cada elemento *_c[i]* es un conjunto de items, ya sea I_0, I_1, I_2, …, I_n.
noItems	`int _noItems;`	Número de items registrados en la colección canónoca *_c*.

La colección canónica es un conjunto de items. Un item es representado por la clase *Item*.

```
class Item
    {
        int[,] _item;
        int _noItems;

        // definición de métodos
    }
```

Descripción de atributos de la clase *Item*.

ATRIBUTO	DEFINICIÓN	
_item	`int[,] _item;`	*_item* es in arreglo bidimensional de enteros. Los renglones son dimensionados a NOPROD. Las columnas son 2 : la primera representa al número de la producción, y la segunda es la posición del punto en el item.

`_noItems int _noItems;`	Indica el número de items registrados para este objeto de la clase *Item*. Realmente un objeto de la clase *Item* es un conjunto de items –arreglo *_item*-.

Continuaremos con los métodos definidos en las clases *SintAscSLR* e *Item*. Empezaremos con los constructores.

7. CONSTRUCTORES EN LAS CLASES *SintAscSLR* e *Item*.

La clase *SintAscSLR* tiene un solo constructor :

```
public SintAscSLR()
        {
            _pila = new Pila();
            _dd = new int[NODDS];
            _noDds = 0;
            _action = new int[NOACTIONS * (_vts.Length - 1), 4];
            _goTo = new int[NOGOTOS * (_vns.Length - 1), 3];
            _noActions = 0;
            _noGoTos = 0;
            _noItems = 0;
        }
```

SENTENCIA	***DESCRIPCION***
`_pila = new Pila();`	Reserva el espacio para el objeto de la clase *Pila*, usado en el modelo del reconocedor ascendente SLR.
`_dd = new int[NODDS];`	Dimensiona al arreglo *_dd* a la constanrte definida en la clase.
`_noDds = 0;`	Inicializa el número de producciones registradas en la derivación a la derecha, al valor de 0.
`_action = new int[NOACTIONS * (_vts.Length - 1), 4];`	Dimensiona el arreglo _action. Los renglones han sido multiplicados por la constante NOACTIONS, declarada en la clase. Las columnas siempre serán 4 como se ha indicado previamente.
`_goTo = new int[NOGOTOS * (_vns.Length - 1), 3];`	Dimensiona el arreglo _goTo. Los renglones han sido multiplicados por la constante NOGOTOS, declarada en la clase. Las columnas siempre serán 3 como se ha indicado previamente.
`_noActions = 0;`	Inicializa el número de acciones registradas en la tabla de reconocimiento SLR.
`_noGoTos = 0;`	Inicializa el número de GOTOs registrados en la tabla de reconocimiento SLR.
`_noItems = 0;`	Asigna el valor de 0, al número de items registrados en la colección canónica de items *_c*.

A diferencia de la clase *SintAscSLR*, la clase *Item* tiene 2 constructores :

```
public Item(int[,] arre, int len)
        {
            _noItems = len;
            _item = new int[SintacticoAsc.NOPROD, 2];
            for (int i = 0; i < len; i++)
                for (int j = 0; j < 2; j++)
                    _item[i, j] = arre[i, j];
        }

public Item()
        {
            _item = new int[SintacticoAsc.NOPROD, 2];
            _noItems = 0;
        }
```

El primer constructor de la clase *Item* recibe 2 parámetros : un arreglo bidimensional de enteros que contiene un conjunto de items, y la longitud del arreglo –número de renglones-, que a su vez es el número de items. El arreglo *arre* es copiado al arreglo *_item*.

El segundo constructor es el llamado por defecto, no tiene parámetros. Sólo dimensiona al arreglo *_item* y asigna al atributo *_noItems* el valor de 0.

8. MÉTODOS DE LA CLASE *SintAscSLR*.

Esta clase contiene los 11 métodos –además del constructor-, que se listan en la tabla siguiente :

MÉTODO	***DESCRIPCION***
`public void Inicia()`	Inicializa los atributos de la clase tal y como lo hace el constructor. Dimensiona la colección canónica de items *_c* a 1000 conjuntos de items. También reserva el espacio para cada objeto *Item* del atributo *_c*. Crea el item I_0 *S' -> .S* y calcula la cerradura del mismo. Crea item I_1 *S' -> S.* y lo asigna usando uno de los constructores de la clase *Item*. Calcula el resto de items de la colección canónica _c usando el método *AgregarConjItems(i)*. Crea los *goTos* del item *S'->.S* correspondiente a la gramática aumentada. Genera cambios y reducciones correspondientes a la sección de *acciones* de la tabla de reconocimiento SLR, utilizando los métodos *GeneraCambios(i)*, y *GeneraReducciones(i)*.
`public Item Cerradura(Item oItem)`	Calcula la cerradura del objeto oItem que recibe de parámetro. Hace uso del método *AgregarItems(i, ref oItem)*.
`public void AgregarConjItems(int i)`	Este método es utilizado cuando calculamos la colección canónica de items *_c* dentro del método *Inicia()*. Su función entre otras es la de calcular y agregar nuevos items a *_c* tomando de referencia al conjunto de items *_c[i]*. Recibe de parámetro un entero *i* que corresponde al índice del conjunto de items I_i de la colección *_c*. Este *_c[i]* sirve para generar nuevos conjuntos de items, si así resulta del algoritmo. Otra tarea importante que realiza este método, es la de calcular los *goTos* del estado I_i representado por *_c[i]*.

	Dentro de este método también son usados los métodos cerradura() y *EstaNuevoItem().*
`public int AgregarItems(int i, ref Item oItem)`	Es usado por el método *Cerradura()*. Recibe de parámetros al índice *i* que corresponde al número de item dentro del objeto *oItem*. Su función es agregar nuevos items a la cerradura si asi corresponde, generados por el item con índice *i*. Hace uso del método *ExisteItem()* para no agregar si el item ya existe dentro del objeto *oItem*.
`public bool EstaNuevoItem(Item oNuevoItem,out int edoYaExiste)`	Este método es utilizado por el método *AgregarConjItems(int i)*. Busca dentro de la colección canónica *_c* al objeto *oNuevoItem* que recibe de parámetro. Retorna en el parámetro *edoYaExiste* que recibe, el número del estado I_i dentro de la colección *_c* que corresponde al estado que ya existe. Retorna *true* si el objeto *oNuevoItem* existe en *_c*, de lo contrario retorna *false*.
`public void GeneraReducciones(int i)`	Genera las reducciones si las hay, producidas por el conjunto de items I_i de la colección canónica *_c*. Recibe de parámetro al índice *i* correspondiente al estado I_i.
`public  void GeneraCambios(int i)`	Genera los cambios si los hay, producidos por el conjunto de items I_i de la colección canónica *_c*. Recibe de parámetro al índice *i* correspondiente al estado I_i.
`public int Analiza(Lexico oAnalex)`	Este método es la codificación en C# del algoritmo para un reconocedor ascendente SLR presentado en el libro del dragón, usando la propuesta de clases de R.A.F. Recibe como parámetro al objeto *oAnaLex* que contiene a los tokens reconocidos en la etapa previa del análisis léxico.
`public string Accion(string s, string a)`	Retorna una cadena que corresponde al tipo de acción según los parámetros *s* y *a* que recibe. Los parámetros representan al estado *s* y *a* representa al token para el que está registrado una acción en la sección ACTION de la tabla M.
`public void SacarDosBeta(string accion)`	Es el código C# para el procedimiento indicado en el libro del dragón dentro del algoritmo del reconocedor ascendente.
`public void MeterAGoTo(string accion)`	Es el código C# para el procedimiento indicado en el libro del dragón dentro del algoritmo del reconocedor ascendente.

La codificación propuesta es mostrada enseguida :

```
class SintAscSLR
    {
        public const int NOPROD = ;
        public const int NODDS = 1000;
        public const int NOACTIONS = 1000;
        public const int NOGOTOS = 1000;


        Pila _pila;
        int[,] _action;
        int _noActions;
        int _noGoTos;
        int[,] _goTo;
        int[] _dd;
```

```
        int _noDds;
        Item [] _c;
        int _noItems;

        // Metodos

        public SintacticoAsc()   // Constructor-----------------------------
        {
            _pila = new Pila();
            _dd = new int[NODDS];
            _noDds = 0;
            _action = new int[1000 * (_vts.Length - 1), 4];
            _goTo = new int[1000 * (_vns.Length - 1), 3];
            _noActions = 0;
            _noGoTos = 0;
            _noItems=0;
        }  // Fin del constructor ---------------------------------------------

        public void Inicia()  //-------------------------------------------
        {
            _pila.Inicia();
            _noDds = 0;
            _noActions = 0;
            _noGoTos = 0;
            _c = new Item[1000];
            _noItems = 0;
            for (int i = 0; i < _c.Length; i++)
                _c[i] = new Item();

           //crea item 0 y calcula la cerradura del mismo---------------
            int [,] arre={{-1,0}};
            _c[_noItems++] = Cerradura(new Item(arre, 1));

            //crea item 1 y lo asigna ----------------------------------
            int[,] arreItem1 ={ { -1, 1 } };
            _c[_noItems++] = new Item(arreItem1, 1);

            //calcula la coleccion canonica de la gramatica-------------
            for (int i = 0; i < _noItems; i++)
                if (i != 1)
                    AgregarConjItems(i);

            //crear los goTos del item  S'->.S gramatica aumentada------------
            _goTo[_noGoTos, 0] = 0;
            _goTo[_noGoTos, 1] = 1;
            _goTo[_noGoTos++, 2] = 1;

            //genera cambios y reducciones de la tabla M----------------------
            for (int i = 0; i < 1000; i++)
            {
                GeneraCambios(i);
                GeneraReducciones(i);
            }

        }  // fin de Inicia() ---------------------------------------------------------------------

        public Item Cerradura(Item oItem)  // Cerradura de un item-------------------------------------
        {
            bool cambios = true;
            while (cambios)
            {
                for (int i = 0; i< oItem.NoItems; i++)
                {
                    int noItemsAgregado = AgregarItems(i, ref oItem);
                    if (noItemsAgregado > 0)
                    {
                        cambios = true;
                        break;
                    }
                    else
                        cambios = false;
                }
            }
            return oItem;
```

```
        }  // Fin de Cerradura() ----------------------------------------------------------------------

        public void AgregarConjItems(int i)  //-------------------------------------------------------
        {
            bool[] marcaItems = new bool[NOPROD];
            for (int j = 0; j < NOPROD; j++)
                marcaItems[j] = false;
            marcaItems[0] = i == 0 ? true : false;
            for(int j=0;j< _c[i].NoItems;j++)
                if (!marcaItems[j])
                {
                    int noProd = _c[i].NoProd(j);
                    int posPto = _c[i].PosPto(j);
                    if (posPto != _prod[noProd, 1])
                    {
                        Item oNuevoItem = new Item();
                        int indSimGoTo = _prod[noProd, posPto + 2];
                        for(int k=0;k< _c[i].NoItems;k++)
                            if (!marcaItems[k])
                            {
                                int nP = _c[i].NoProd(k);
                                int pP = _c[i].PosPto(k);
                                if (indSimGoTo == _prod[nP, pP + 2])
                                {
                                    oNuevoItem.Agregar(nP, pP + 1);
                                    marcaItems[k] = true;
                                }
                            }
                        int edoYaExiste;
                        _goTo[_noGoTos, 0] = i;
                        _goTo[_noGoTos, 1] = indSimGoTo;
                        oNuevoItem = Cerradura(oNuevoItem);
                        if (!EstaNuevoItem(oNuevoItem,out edoYaExiste))//verifica si el item no existe
                        {
                            _goTo[_noGoTos++,2]=_noItems;
                            _c[_noItems++] = oNuevoItem;
                        }
                        else
                            _goTo[_noGoTos++,2]=edoYaExiste;//calcular el goTo cuando el item no existe
                        }
                }
        }  // Fin de AgregarConjItems()-----------------------------------------------------------------
----

        public int AgregarItems(int i, ref Item oItem)  //-------------------------------------------
-----
        {
            int noItemsAgregado = 0;
            int posPto = oItem.PosPto(i);
            int noProd = oItem.NoProd(i);
            int indVns = noProd == -1 ? 1 : (posPto == _prod[noProd, 1] ? 0 : (_prod[noProd, posPto +
2] < 0 ? 0 : _prod[noProd, posPto + 2]));
             if(indVns>0)
                 for(int j=0;j<NOPROD;j++)
                     if (indVns == _prod[j, 0] && !oItem.ExisteItem(j,0)) //busca si existe una
produccion con
                     {                                                    //ese indice y que no exista
el item
                         oItem.Agregar(j, 0);
                         noItemsAgregado++;
                     }
             return noItemsAgregado;
        }  // Fin de AgregarItems() ---------------------------------------------------------------
------

        public bool EstaNuevoItem(Item oNuevoItem,out int edoYaExiste)  //----------------------------
------
        {
            edoYaExiste = -1;
            for(int i=0;i< _noItems;i++)
                if (_c[i].NoItems == oNuevoItem.NoItems)
                {
                    int aciertos = 0;
                    for(int j=0;j< _c[i].NoItems;j++)
```

```
                        for(int k=0;k<oNuevoItem.NoItems;k++)
                            if (_c[i].NoProd(j) == oNuevoItem.NoProd(k) && _c[i].PosPto(j) ==
oNuevoItem.PosPto(k))
                            {
                                aciertos++;
                                break;
                            }
                    if (aciertos == _c[i].NoItems) //si numero de items son iguales a los aciertos,
entonces ya existe
                    {
                        edoYaExiste = i;
                        return true;
                    }

                }
            return false;
        }  // Fin de EstaNuevoItem()  ----------------------------------------------------------------
-

        public void GeneraReducciones(int i)  // reducciones del Item _c[i] ---------------------------
-
        {
            for (int j = 0; j < _c[i].NoItems; j++)
            {
                int noProd = _c[i].NoProd(j);
                int posPto = _c[i].PosPto(j);
                if (i == 1) //cuando el item es 1 se realiza lo siguiente
                {
                    _action[_noActions, 0] = i;
                    _action[_noActions, 1] = _vts.Length-1;
                    _action[_noActions, 2] = 2;
                    _action[_noActions++, 3] = -1;
                }
                else
                    if (noProd != -1 && posPto == _prod[noProd, 1])
                    {
                        int indVns= _prod[noProd, 0];
                        for (int k = 1; k <=_sig[indVns,0]; k++)
                        {
                            _action[_noActions, 0] = i;
                            _action[_noActions, 1] = _sig[indVns,k];
                            _action[_noActions, 2] = 1;
                            _action[_noActions++, 3] = noProd;
                        }
                    }
            }
        }  // Fin de GeneraReducciones()----------------------------------------

        public  void GeneraCambios(int i)  // cambios del Item _c[i]-------------------------
        {
            for (int j = 0; j < _c[i].NoItems; j++)
            {
                int noProd = _c[i].NoProd(j);
                int posPto = _c[i].PosPto(j);
                if (noProd != -1)
                {
                    if (posPto != _prod[noProd, 1])
                    {
                        int indSim = _prod[noProd, posPto + 2];
                        if (indSim < 0)
                        {
                            int edoTrans = -1;
                            for (int k = 0; k < _noGoTos; k++)
                                if (_goTo[k, 0] == i && _goTo[k, 1] == indSim)
                                {
                                    edoTrans = _goTo[k, 2];
                                    break;
                                }
                            _action[_noActions, 0] = i;
                            _action[_noActions, 1] = -indSim;
                            _action[_noActions, 2] = 0;
                            _action[_noActions++, 3] = edoTrans;
                        }
                    }
```

```
                }
            }
        }  // Fin de GeneraCambios() ----------------------------------------------------------------
--

        public int Analiza(Lexico oAnalex)
        {
            int ae = 0;
            oAnalex.Añade("$", "$");
            _pila.Push(new SimbGram("0"));
            while (true)
            {
                string s = _pila.Tope.Elem;
                string a = oAnalex.Token[ae];
                string accion = Accion(s,a);
                switch (accion[0])
                {
                    case 's': _pila.Push(new SimbGram(a));
                              _pila.Push(new SimbGram(accion.Substring(1)));  // caso en que la accion
es un cambio
                              ae++;
                              break;
                    case 'r': SacarDosBeta(accion);//sacar dos veces Beta simbolos de la pila
                              MeterAGoTo(accion);  //meter Vns y goTos a la pila
                              _dd[_noDds++]=Convert.ToInt32(accion.Substring(1));  // caso en que la
accion es una
                              break;                                               // reduccion
                    case 'a': return 0;  // aceptacion
                    case 'e': return 1;  // error
                }
            }
        }  // Fin de Analiza() ----------------------------------------------------------------------
----------

        public string Accion(string s, string a)  // ------------------------------------------------
----------
        {
            //metodo que determina que accion se realizara
            int tipo=-1, no=-1;
            int edo = Convert.ToInt32(s);
            int inda = 0;
            bool enc = false;
             for(int i=1;i<_vts.Length;i++)
                 if (_vts[i].Equals(a))
                 {
                     inda = i;
                     break;
                 }
            for(int i=0;i<_noActions;i++)
                if (_action[i, 0] == edo && _action[i, 1] == inda)
                {
                    tipo = _action[i, 2];
                    no = _action[i, 3];
                    enc = true;
                }
            if (!enc)
                return "error";
            else
                switch (tipo)
                {
                    case 0: return "s" + no.ToString();
                    case 1: return "r" + no.ToString();
                    case 2: return "acc";
                    default: return "error";
                }

        }  // Fin de Accion() ------------------------------------------------------------------

        public void SacarDosBeta(string accion)  //-------------------------------------------
        {
            int noProd = Convert.ToInt32(accion.Substring(1));
            int noVeces = _prod[noProd, 1] * 2;
            for (int i = 1; i <= noVeces; i++)
                _pila.Pop();
```

```
        }  // Fin de SacarDosBeta() ------------------------------------------------------------

        public void MeterAGoTo(string accion)  //---------------------------------------------
        {
            int sPrima = Convert.ToInt32(_pila.Tope.Elem);
            int noProd = Convert.ToInt32(accion.Substring(1));
            _pila.Push(new SimbGram(_vns[_prod[noProd, 0]]));
             for(int i=0;i<_noGoTos;i++)
                 if (sPrima == _goTo[i, 0] && _prod[noProd, 0] == _goTo[i, 1])
                 {
                   _pila.Push(new SimbGram(_goTo[i,2].ToString()));
                   break;
                 }
        }  // Fin de MeterAGoTo() ---------------------------------------------------------

    } // fin de la clase SintAscSLR
```

Observemos que en el código anterior no están definidos el número de producciones *NOPROD*, ni los atributos :

```
string[] _vts;
string[] _vns;
int[,] _prod;
int[,] _sig;
```

Estos atributos serán generados por el programa *SA-SLR*.

9. MÉTODOS DE LA CLASE *Item*.

Son 4 los métodos definidos en esta clase además de los 2 constructores previamente mencionados en la sección 7. También se ha incluido una propiedad en la clase.

MÉTODO	***DESCRIPCION***
`public int NoProd(int i)`	Retorna el número de producción del item *i*. El número de producción se encuentra en la columna 0 de cada renglón. Cada renglón denota a un item.
`public int PosPto(int i)`	Retorna la posición del punto en el item *i*. La posición del punto se encuentra en la columna 1 de cada renglón. Cada renglón denota a un item.
`public bool ExisteItem(int noProd, int posPto)`	Retorna *true* si el item cuyo número de producción *noProd* y posición del punto *posPto*, ya se encuentra en el arreglo *_item* del objeto de la clase *Item*. Retorna *false* si no se encuentra.
`public void Agregar(int noProd, int posPto)`	Este método agrega al arreglo *_item* un nuevo item cuyo número de producción es *noProd* y cuya posición del punto es *posPto*. *noProd* y *posPto* son recibidos como parámetros.
`public int NoItems`	*NoItems* es una propiedad de sólo lectura que retorna al número de items del objeto de la clase *Item*.

La clase *Item* con sus atributos, métodos y propiedades es listada a continuación :

```
class Item
{
    int[,] _item;
    int _noItems;

    // Metodos ------------------------------------------------------------

    public Item(int[,] arre, int len)
    {
        _noItems = len;
        _item = new int[SintAscSLR.NOPROD, 2];
        for (int i = 0; i < len; i++)
            for (int j = 0; j < 2; j++)
                _item[i, j] = arre[i, j];
    }

    public Item()
    {
        _item = new int[SintAscSLR.NOPROD, 2];
        _noItems = 0;
    }

    public int NoItems
    {
        get { return _noItems; }
    }

    public int NoProd(int i)
    {
        return _item[i, 0];
    }

    public int PosPto(int i)
    {
        return _item[i, 1];
    }

    public bool ExisteItem(int noProd, int posPto)
    {
        for (int i = 0; i < _noItems; i++)
            if (_item[i, 0] == noProd && _item[i, 1] == posPto)
                return true;
        return false;
    }

    public void Agregar(int noProd, int posPto)
    {
        _item[_noItems, 0] = noProd;
        _item[_noItems++, 1] = posPto;
    }

}  // Fin de la clase Item
```

10. EL PROGRAMA *RA-SLR*.

RA-SLR es un programa que consiste de un ejecutable, y de 4 archivos tipo texto. Cada archivo texto contiene a una de las clases : *SintAscSLR*, *Item*, *Pila* y *SimbGram*.

- **RA-SLR.exe**
- **molde-clase-sintascslr**
- **clase-item**
- **clase-pila**
- **clase-simbgram**

Los 5 archivos deben residir en el mismo directorio. Los archivos texto contienen los atributos, métodos y propiedades que han sido descritos en las secciones previas.

Su interfase consiste de 4 carpetas : *Generar gramática*, *Generación de código*, *Otras clases*, *Acerca de* y *Salir*, fig#10.1.

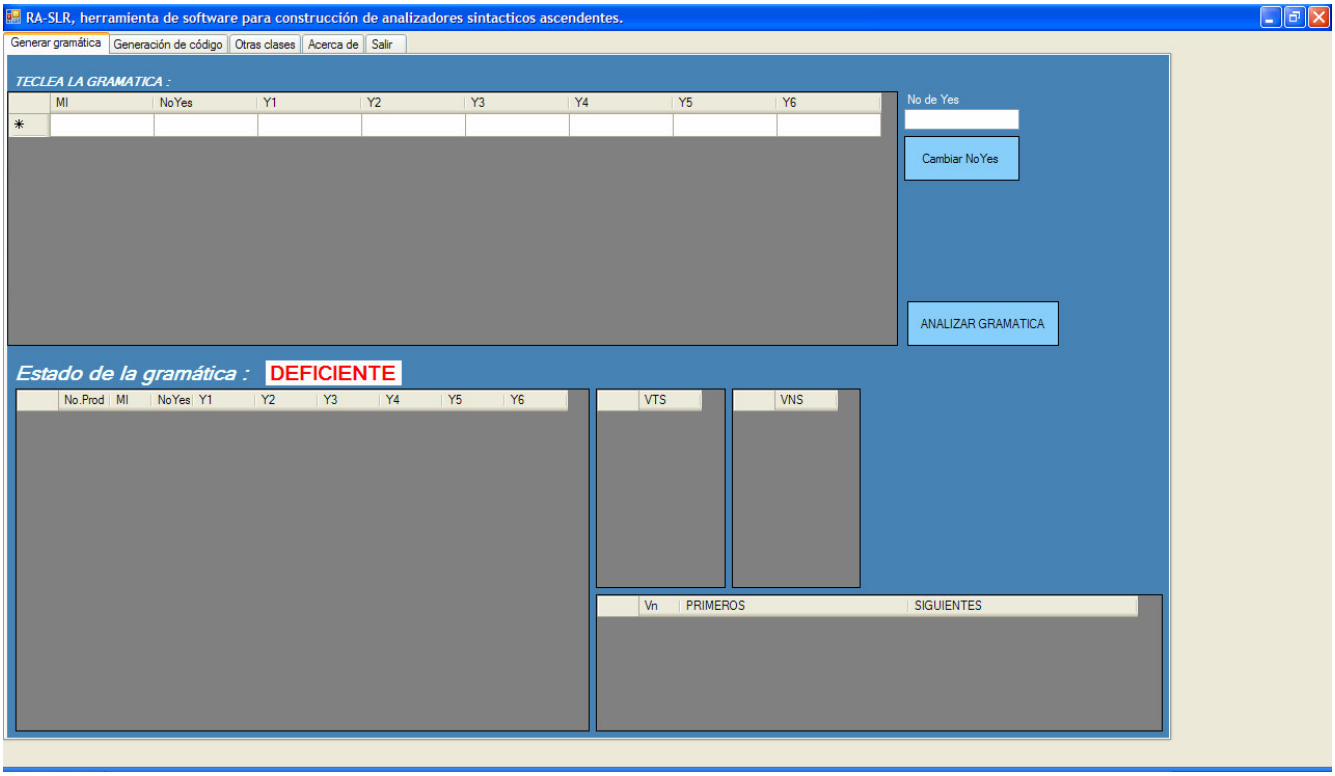


Fig. No. 10.1 Interfase del programa *RA-SLR*.

La tarea fundamental de *RA-SLR* es la de permitir ingresar una gramática de contexto libre, que contenga las reglas de sintaxis de determinadas sentencias, para luego producir un análisis de esta gramática. El resultado del análisis es la identificación y registro de los componentes de la gramática : símbolos terminales *Vts*, símbolos no terminales*Vns*, símbolo de inicio *S* y el conjunto de producciones sin agrupar. También calcula los conjuntos *Primeros* y *Siguientes* para cada símbolo no terminal de la gramática.

Ya que la gramática ha sido ingresada –dentro de la carpeta *Generar gramática*-, el programa *RA-SLR* permite generar el código C# que contiene a las 4 clases propuestas por R.A.F. que podrá ser copiado e insertado a una aplicación C# que haga un reconocimiento sintactico ascendente SLR.

Antes de usar el programa debemos haber hecho las siguientes tareas :

- Diseño de la gramática, donde debemos tener a cada producción de la gramática SIN AGRUPAR.
- Conocer el número máximo de *Y's* que podrá tener una producción de la gramática. Cada *Y* representa a un símbolo gramatical, ya sea no terminal, ya sea terminal.
- Debemos identificar cual es el número de *Y's* de cada producción, además de su miembro izquierdo que siempre será un no terminal.

Las carpetas *Otras clases*, *Acerca de* y *Salir* no tienen gran dificultad en su comprensión. *Otras clases* sólo visualiza las clases *Item*, *Pila* y *SimbGram*, las cuales podemos copiar e insertar en la aplicación que estemos construyendo, fig#10.2.

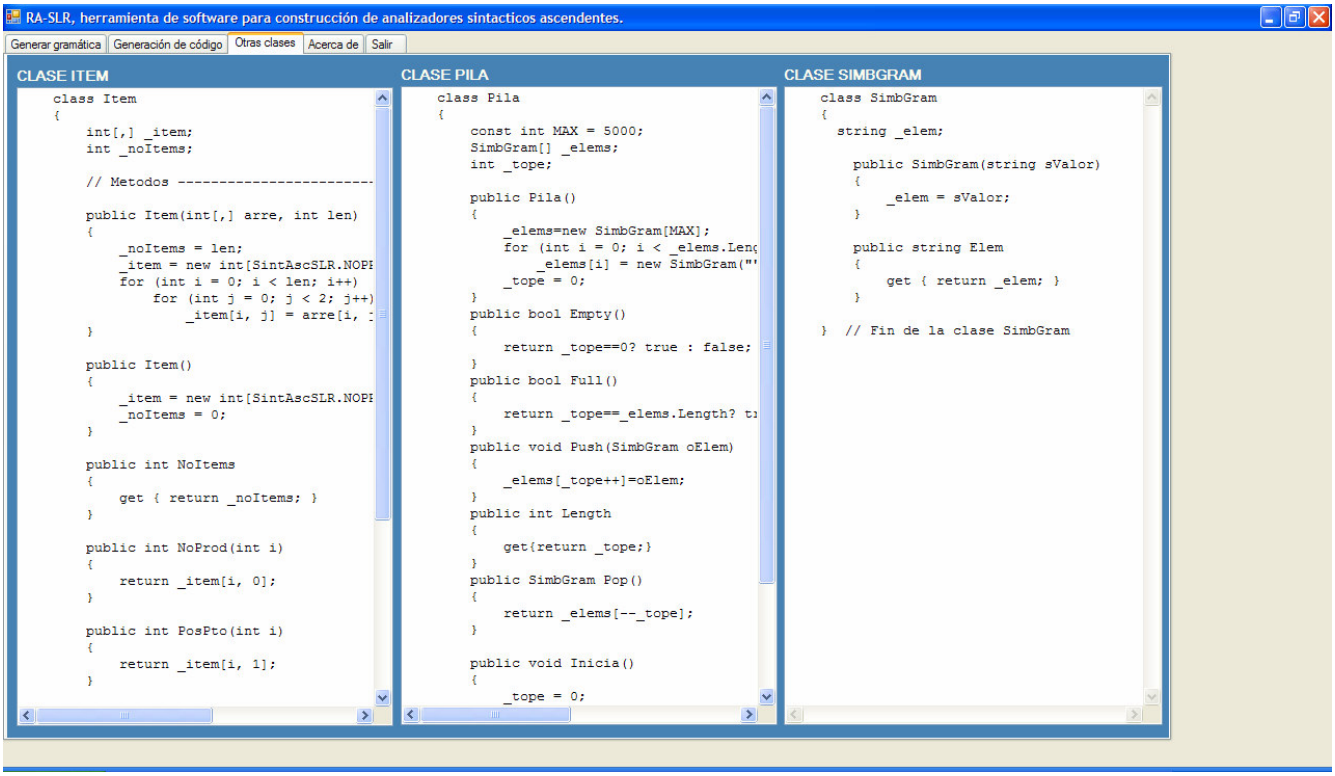


Fig. No. 10.2 Carpeta *Otras clases*.

La carpeta *Acerca de* contiene información del autor del programa –R.A.F.-. La carpeta *Salir* permite abandonar el programa *RA-SLR*.

En las secciones siguientes veremos el uso de las carpetas *Generar gramática* y *Generación de código*. Seguiremos construyendo el ejemplo citado en la sección **2**, para el cual ya se ha presentado la gramática y se ha construido el analizador léxico usando el programa *SP-PS1*.

11. CARPETA *Generar gramática*.

Permite ingresar la grámatica dentro de un componente *DataGridView*. Debemos tener previamente diseñada la gramática. La gramática debe ser de contexto libre, es decir, en el miembro izquierdo de cada producción consiste de sólo un símbolo no terminal –variable sintáctica-.

Existen ciertas reglas que deben seguirse cuando se ingresa la gramática a la rejilla :

- Un renglón representa a una sóla producción. NO debe ingresarse diferentes alternativas –producciones- en un mismo renglón.
- Los símbolos no terminales sólo pueden nombrarse con una sóla letra y debe ser una letra MAYÚSCULA. De aquí que sólo podemos tener en esta versión del *RA-SLR* hasta **26** símbolos no terminales –variables sintacticas-.
- Los símbolos terminales no deben empezar con letra MAYÚSCULA y pueden contener cualquier caracter menos los espacios blancos y caracteres no visibles.
- El número de *Y's* tecleada en la segunda columna de la rejilla no puede ser 0. Por consecuencia, la gramática no puede contener producciones empty –vacias-.
- El número de *Y's* tecleado debe coincidir con las *Y's* que se han ingresado en la columna correspondiente *Y1, Y2, Y3, …,Yn*.

Conociendo estas sencillas reglas, iniciemos con el ejemplo de la sección 2. La gramatica citada en esa sección, tiene 40 producciones cuyo máximo número de *Y's* es de 6. por omisión el número de *Y's* es de 6, asi que en este ejemplo no es necesario cambiar el número de *Y's*.
Si se requiriera cambiar el número de *Y's*, debemos teclear el nuevo valor en el *TextBox* etiquetado con la leyenda *No de Yes* y hacer click en el botón *Cambiar No Yes*.

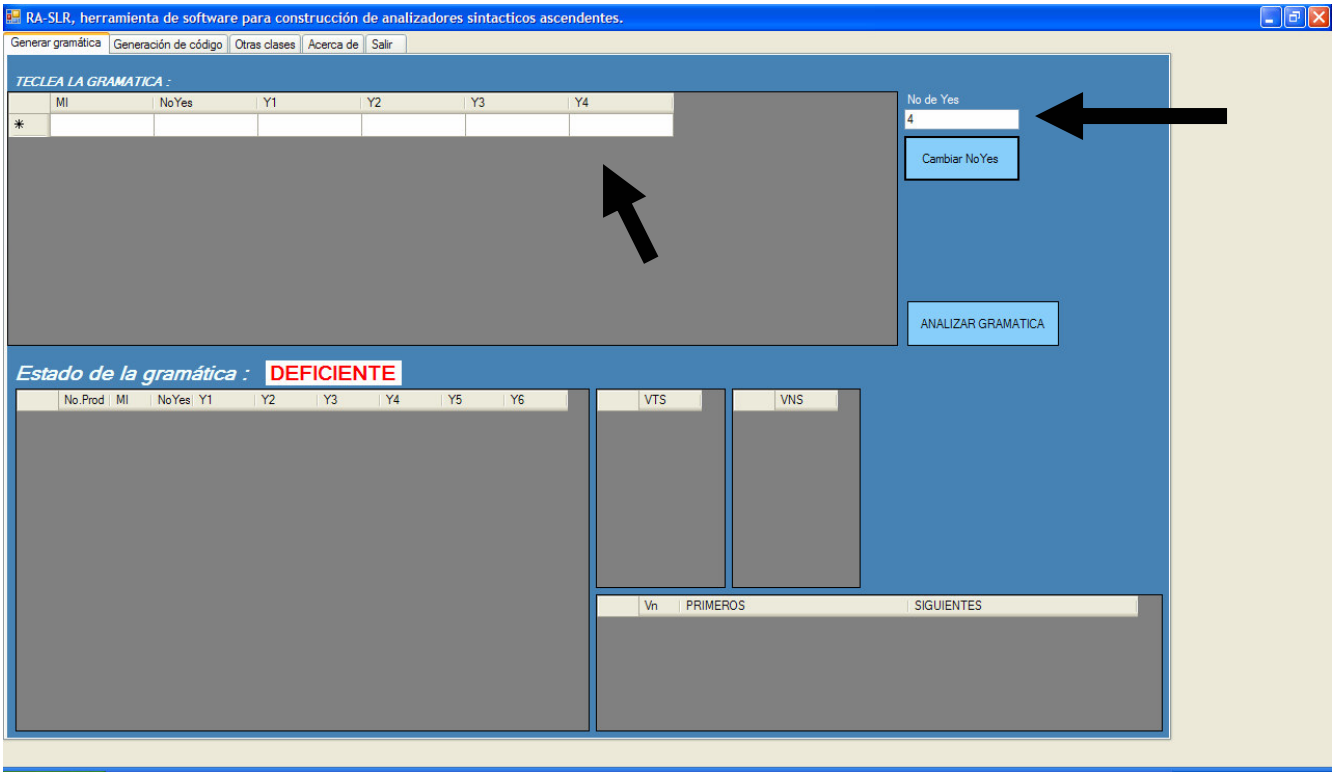


Fig. No. 11.1 Cambio del número de *Y's* a 4.

Tecleemos las 40 producciones de la gramática en la rejilla con leyenda TECLEA LA GRAMATICA, usando las teclas de flechas para navegar en la rejilla, e ir agregando renglones, fig#11.2 :

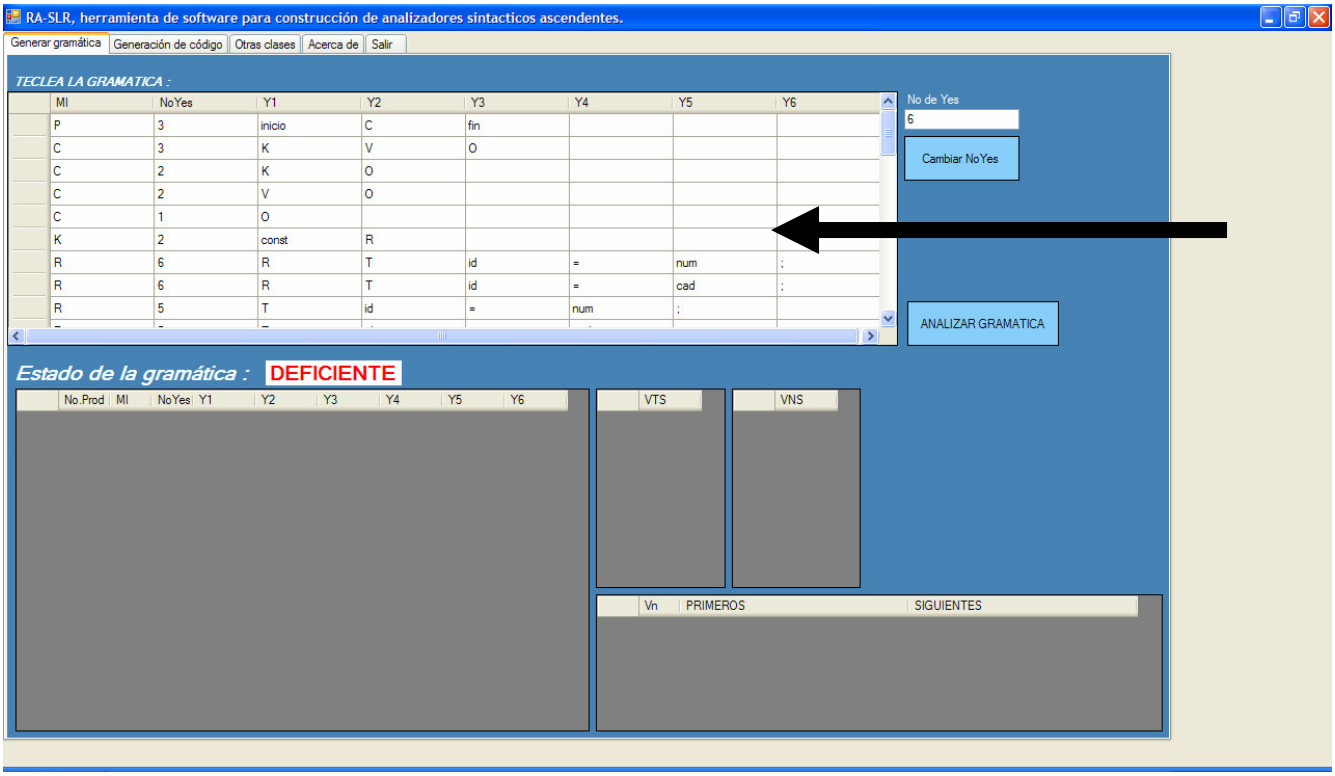


Fig. No. 11.2 Gramática ingresada antes de ser analizada.

Una vez que hemos ingresado a la gramática, hacemos click en el botón con leyenda ANALIZAR GRAMATICA. Si la gramática ha sido tecleada sin errores, el programa *RA-SLR* contesta con una caja de mensajes, fig#11.3.

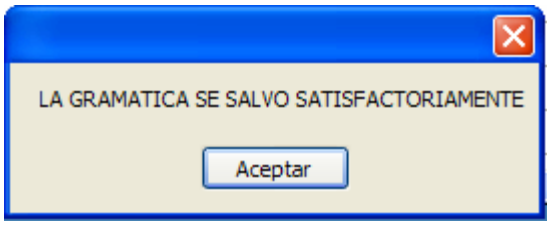


Fig. No. 11.3 Mensaje de éxito en el análisis.

Luego de que hacemos click en el botón de la caja de mensajes, *RA_SLR* visualiza las producciones, los *Vts*, los *Vns* además de los primeros y los siguientes para cada *Vn* de la gramática, fig#11.4.

Observemos en la figura#11.4, que el letrero de estado de la gramática se ha actualizado a OK, además que se pueden comprobar los resultados con los nuestros.

Para nuestro ejemplo tenemos 40 producciones, 21*Vts* y 16 *Vns*.

Recomiendo al lector que haga pruebas no respetando las reglas citadas anteriormente en esta sección, con el fin de observar los mensajes que produce *RA-SLR*.

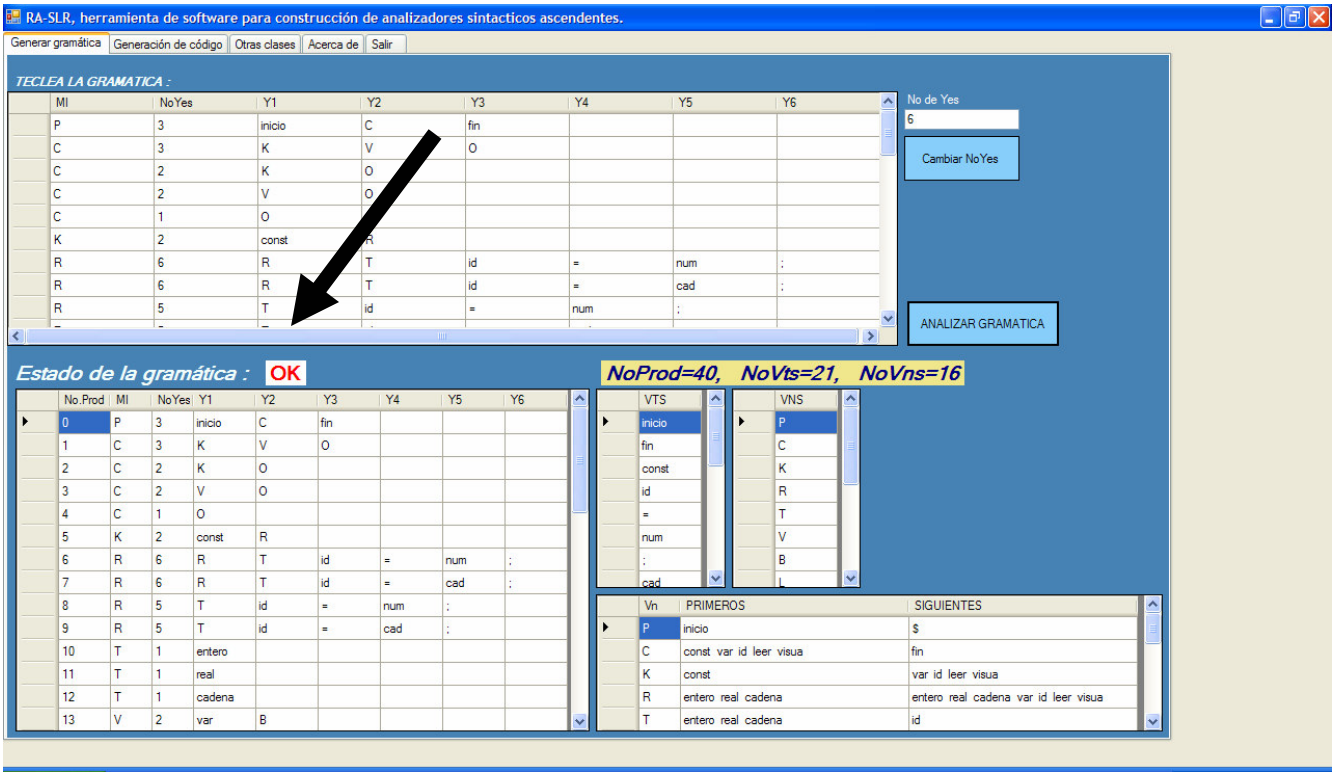


Fig. No. 11.4 Análisis de la gramática con resultado OK.

Ahora seguiremos con la generación del código de la clase *SintAscSLR*.

12. CARPETA *Generación de código*.

La carpeta *Generación de código* presenta las producciones de la gramática, los símbolos terminales *Vts*, los símbolos no terminales *Vns*, los *Primeros* y los *Siguientes* de cada *Vn* de la gramática, además de la clase *SintAscSLR* sin código incrustado, figura#11.5.

La generación de código sólo podemos efectuarla cuando el estado de la gramática es OK. Cuando el estado de la gramática es DEFICIENTE, el botón con leyenda GENERAR CODIGO es deshabilitado.

Cuando hacemos click en el botón GENERAR CODIGO simplemente insertamos el valor para la constante NOPROD, la definición de los atributos *_vts*, *_vns*, *_prod* y *_sig*.

La tabla M que contiene las acciones y los gotos, la colección canónica de items, son construidos dentro del método *Inicia()* de la clase *SintAscSLR*.

Como hemos visto, la generación de código es muy simple. Lo que realmente vale la pena es explicar lo propuesto por R.A.F. y que ha sido programado en los métodos de la clase *SintAscSLR* y las otras clases.

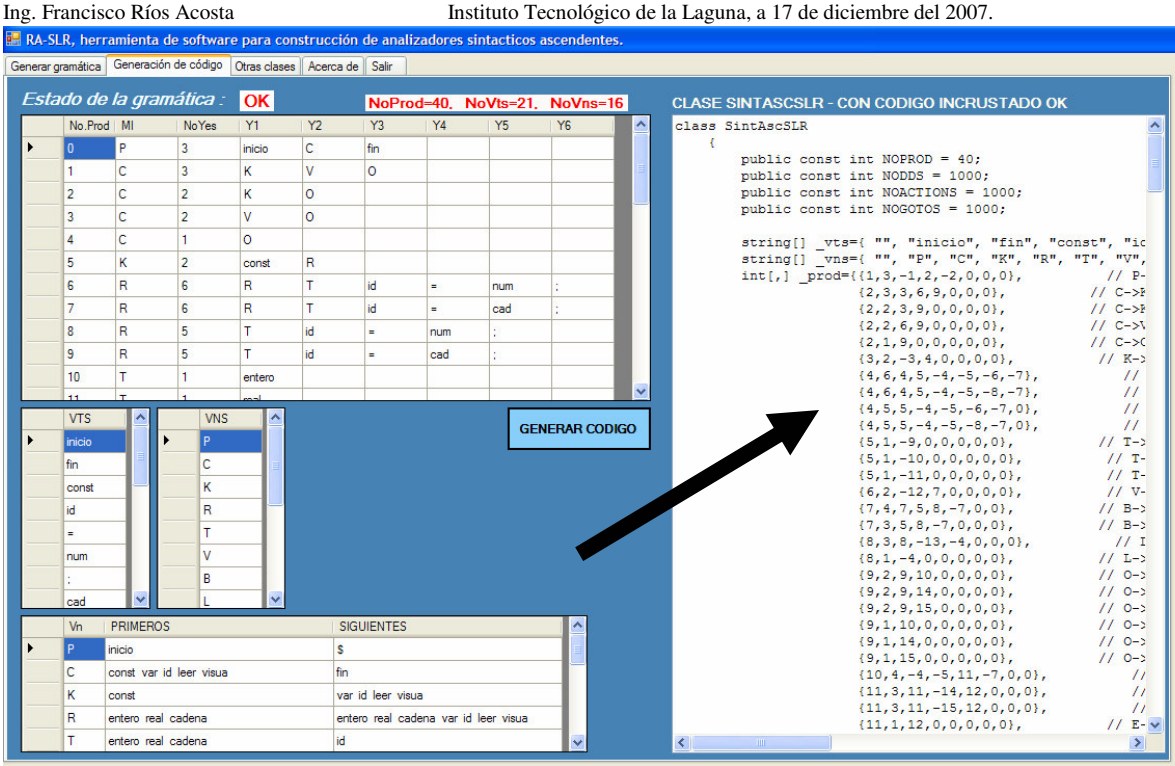


Fig. No. 12.1 Código generado por *RA-SLR* para nuestra gramática de ejemplo.

13. INSCRUSTACIÓN DE CÓDIGO EN LA APLICACIÓN C# INICIADA EN LA SECCIÓN 2.

Abramos la aplicación Windows C# que hará el análisis sintactico de sentencias que responden a las reglas de sintaxis contenidas en la gramática que ingresamos en la sección anterior.

Lo que sigue es agregar el código generado por *RA-SLR* al proyecto C# que estamos construyendo. Recordemos que *RA-SLR* genera el código de 4 clases : *SintAscSLR*, *Item*, *Pila* y *SimbGram*.

Entonces, agreguemos 4 clases al proyecto usando la trayectoria del menu *Project* | *Add Class*, y asignemos el nombre de a cada clase que añadimos :

- SintAscSLR.cs
- Item.cs
- Pila.cs
- SimbGram.cs

El cuerpo de estas clases inicialmente está vacio, solamente contiene la inclusión de espacios de nombres y el encabezado con el nombre de la clase, ver figura #13.1.

Todas las clases pertenecen al espacio de nombres de nuestro proyecto –aplicación Windows C#-. En nuestro caso el espacio de nombres es *anasinascSLR*.

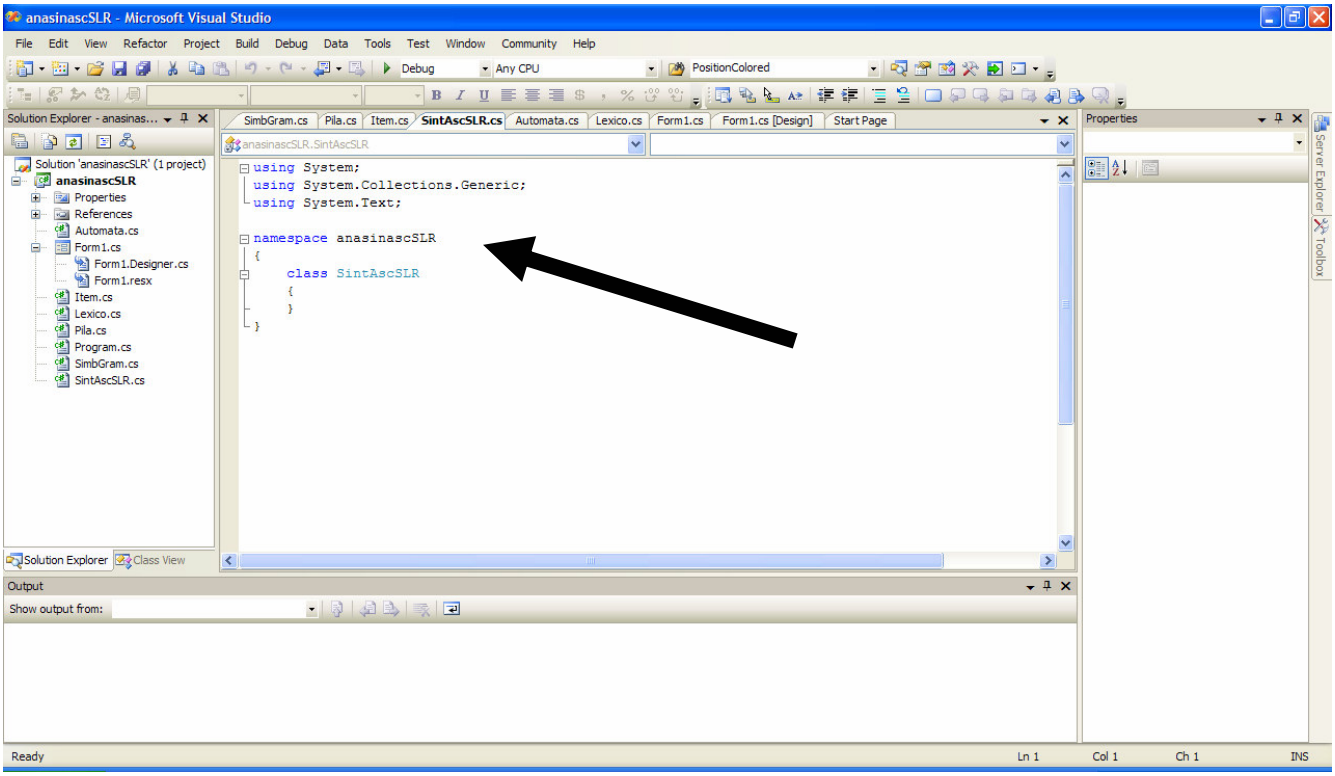


Fig. No. 13.1 Aplicación C# con las 4 clases añadidas.

Procedamos a insertar el código de las clases que genera *RA-SLR* a nuestro proyecto. La cuestión es muy simple, sólo copia el código de cada clase al portapapeles e insertalo en la clase adecuada. La figura #13.2 muestra el código de la clase *SintAscSLR* añadido.

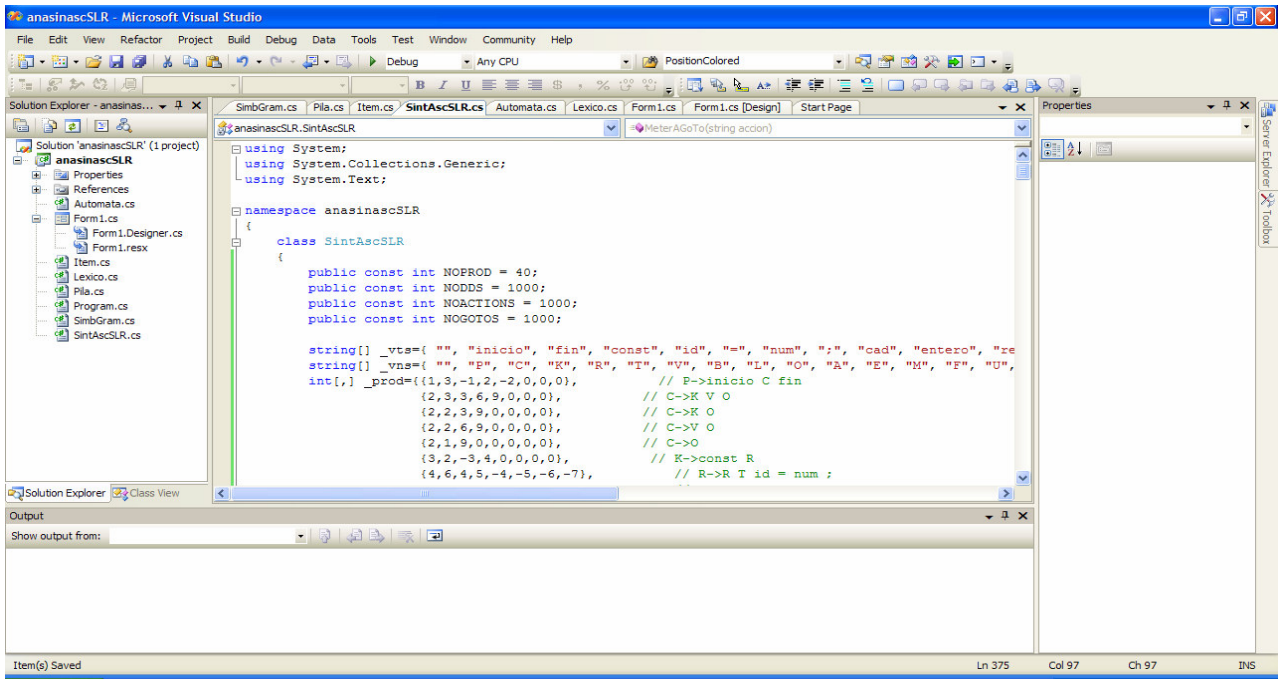


Fig. No. 13.2 Código insertado en las 4 clases.

14. PUESTA A PUNTO DE LA APLICACIÓN WINDOWS C#.

Sólo nos resta quitar los comentarios a los mensajes sobre el objeto *oAnaSintAscSLR* definido en el archivo *Form1.cs* de nuestro proyecto.

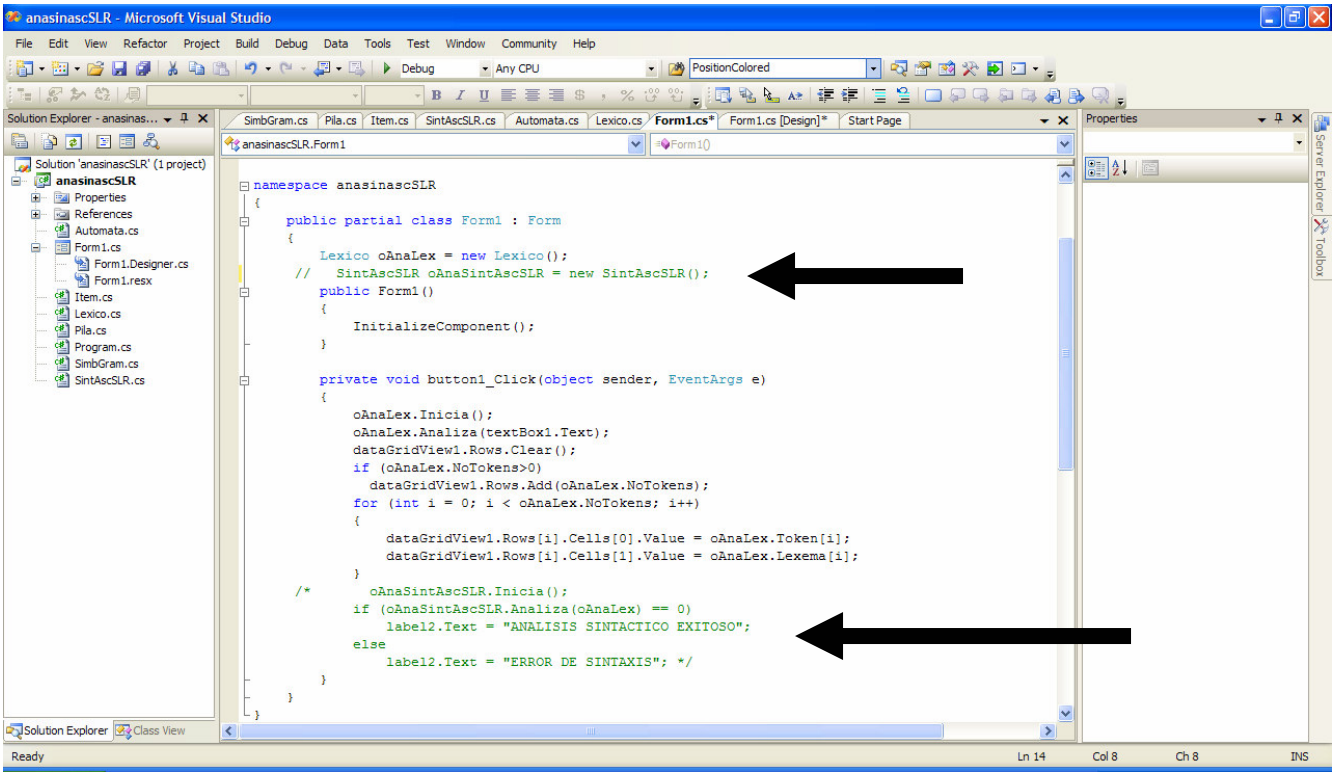


Fig. No. 14.1 Comentarios que debemos eliminar.

La ejecución de la aplicación C# nos reporta un error. El error consiste en que no hemos definido el método *Anade()* en la clase *Lexico*.

El método *Anade()* es usado dentro del método *Analiza()* de la clase *SintAscSLR*. Su propósito es añadir el símbolo **$** a los arreglos *_tokens* y *_lexemas*, atributos de la clase *Lexico*. Es necesario este método, debido a que el símbolo $ es usado dentro del modelo conceptual de un reconocedor ascendente expuesto en el libro del dragón. Su uso es un truco para lograr la aceptación de una sentencia bien formada.

Así que debemos agregar este método dentro de la clase *Lexico*. Su código es mostrado a continuación.

```
public void Anade(string valTok, string valLex)
        {
            _tokens[_noTokens] = valTok;
            _lexemas[_noTokens++] = valLex;

        }
```

El método *Anade()* agrega la pareja token-lexema que recibe de parámetros, a los arreglos *_tokens* y *_lexemas*.

Las figuras #14.2 y 14.3, muestran el caso de un reconocimiento sintactico exitoso y uno erróneo, respectivamente. El error de sintaxis lo provocamos quitando un **;** de la terminación de la instrucción de visualización.

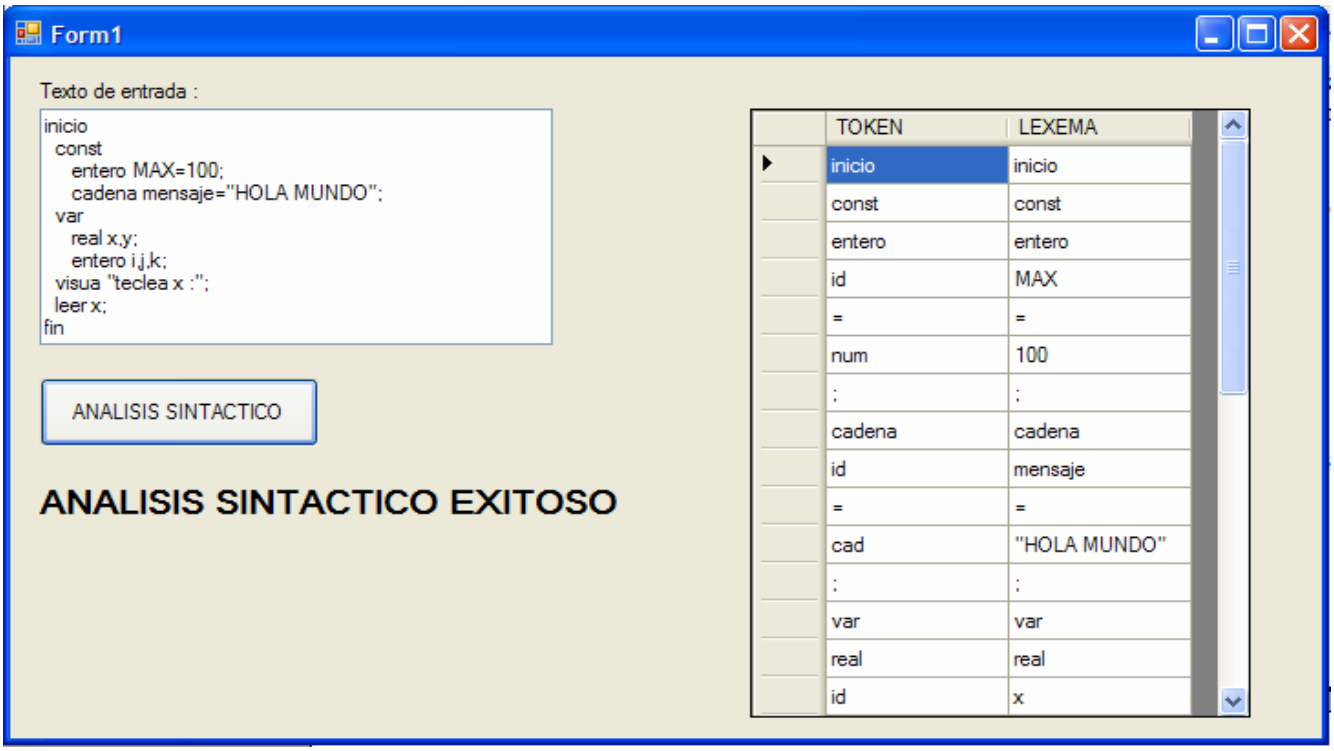


Fig. No. 14.2 Reconocimiento exitoso del texto de entrada.

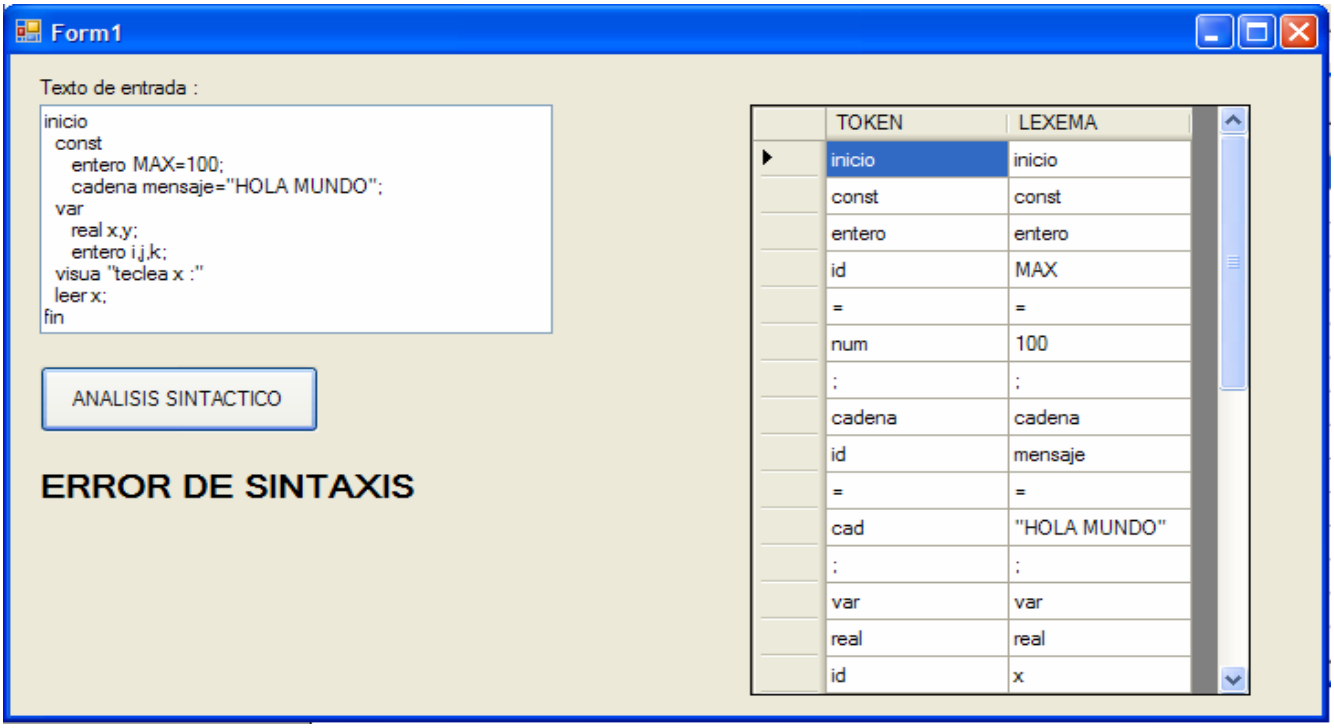


Fig. No. 14.3 Error de sintaxis, caracter de terminación **;** faltante en la sentencia *visua "teclea x:"*.

15. SALVANDO Y CARGANDO UNA GRAMÁTICA.

Podemos salvar las producciones que hemos tecleado para usarlas mas tarde. Lo anterior es hecho por medio del botón con leyenda GUARDAR GRAMATICA.
La gramática es salvada en un archivo binario cuyo nombre es tecleado por el usuario. La extensión no importa, puede añadirse o no.
La figura 15.1 muestra el cuadro de diálogo que presenta la aplicación *RA-SLR* cuando hacemos click sobre el botón antes mencionado.
Es importante mencionar que el estado de una gramática no es salvado en el archivo.

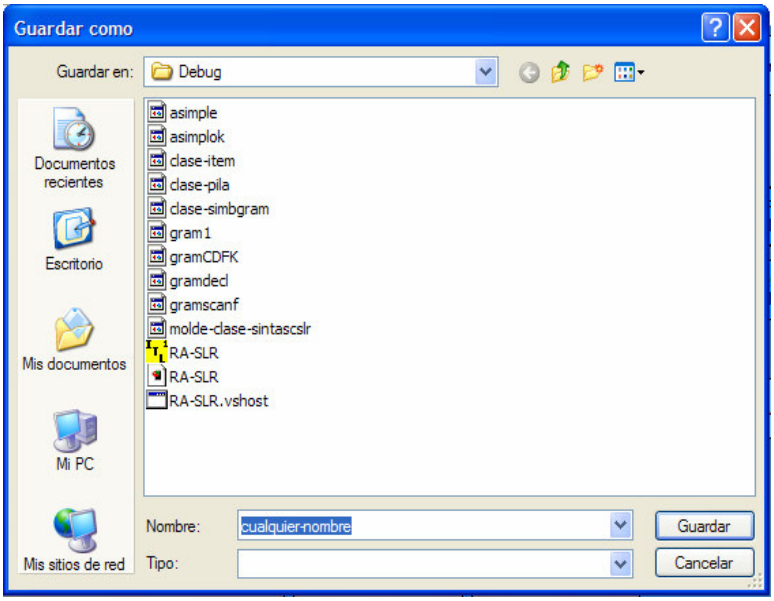


Fig. No. 15.1 Cuadro de diálogo para salvar una gramática.

Para cargar una gramática que reside en un archivo, sólo debemos hacer click sobre el botón marcado con la leyenda CARGAR GRAMATICA. La figura 15.2 muestra el cuadro de diálogo estandar que se presenta cuando el usuario requiere de cargar una gramática.

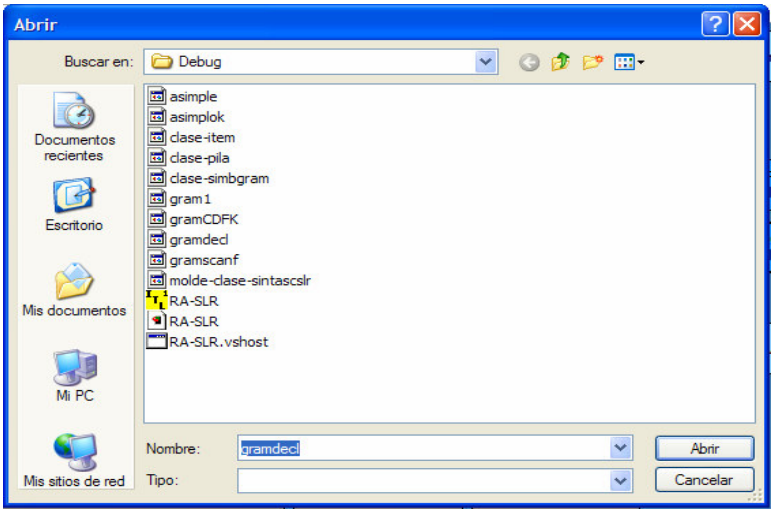


Fig. No. 15.2 Cuadro de diálogo que permite cargar una gramática.

16. LIMPIAR LA REJILLA DE INGRESO DE GRAMÁTICA.

Cuando necesitemos limpiar la rejilla de ingreso de la gramática que contiene la sintaxis de las sentencias a reconocer, debemos hacer click sobre el botón con leyenda LIMPIAR, fig#16.1.
La acción del botón es eliminar los renglones de todas las rejillas de visualización y de ingreso de datos, además de las etiquetas.

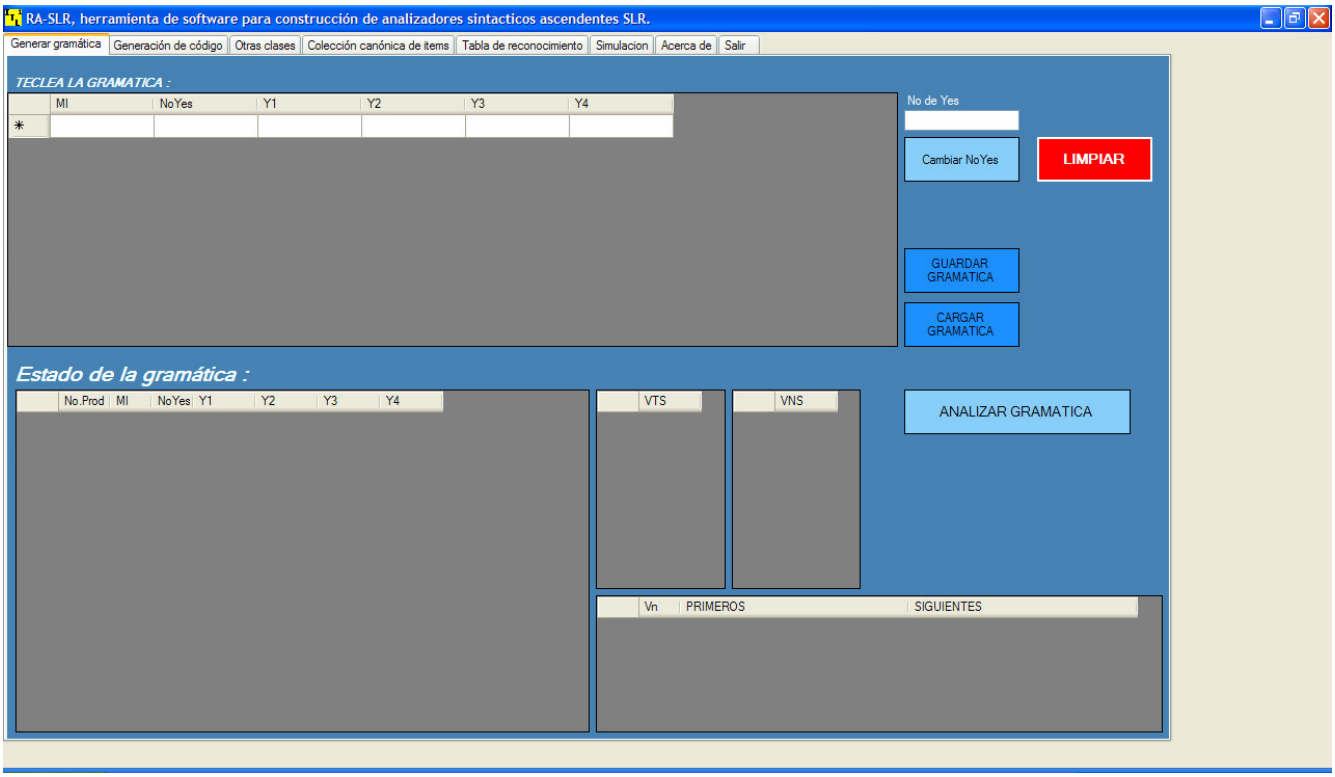


Fig. No. 16.1 Interfase de *RA-SLR* después de hacer click sobre el botón LIMPIAR.

17. VISUALIZACIÓN DE LA COLECCIÓN CANÓNICA DE ITEMS.

La colección canónica de items $C=\{I_0, I_1, I_2, \ldots, I_n\}$ es calculada al momento que es analizada una gramática. Si el resultado del análisis es OK, entonces podemos visualizar la colección canónica de items.

Un item es una producción de la gramática con un punto en cualquier parte del miembro derecho de la producción. Por ejemplo, para la producción :

```
A -> A , id
```

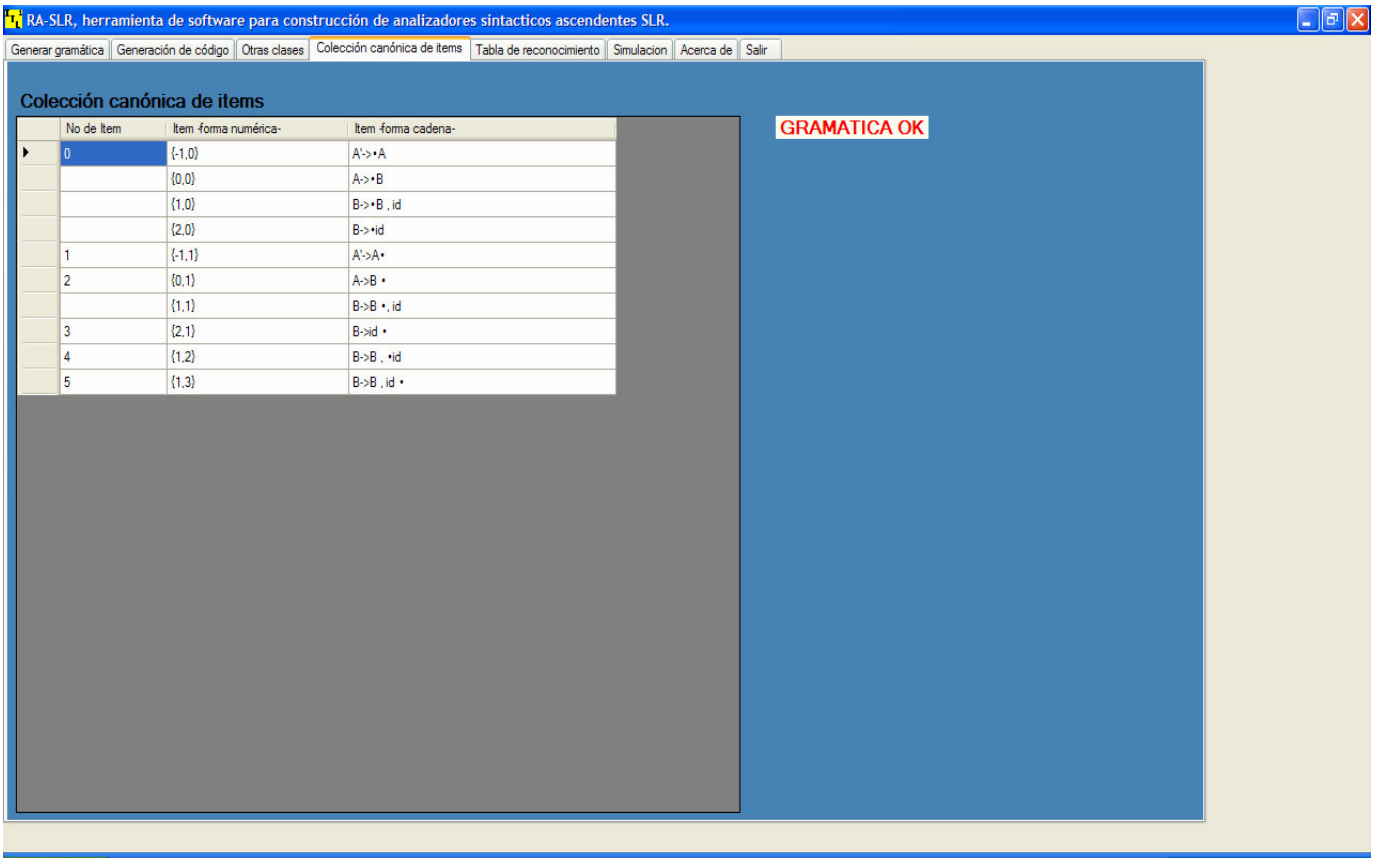
Podemos tener los siguientes items posibles :

```
A -> .A , id
A ->  A. , id
A ->  A , .id
A ->  A , id.
```

La figura 17.1 muestra la colección canónica de items para la gramática :

```
A -> B
B -> B , id
B -> id
```

Sólo es necesario hacer click en la pestaña de la aplicación con leyenda *Colección canónica de items*.



No de Item	Item -forma numérica-	Item -forma cadena-
0	{-1,0}	A'->•A
	{0,0}	A->•B
	{1,0}	B->•B , id
	{2,0}	B->•id
1	{-1,1}	A'->A•
2	{0,1}	A->B •
	{1,1}	B->B •, id
3	{2,1}	B->id •
4	{1,2}	B->B , •id
5	{1,3}	B->B , id •

Fig. No. 17.1 Colección canónica de items para la gramática con símbolo de inicio *A*.

El item en forma numérica indica el par *{noProd*, *posPto}*, donde *noProd* es el número de producción y *posPto* es la posición del punto.
La posición del punto inicia en 0 y su valor máximo es el número de *Y's* de la producción.
En cuanto al número de la producción debemos conocer lo siguiente :

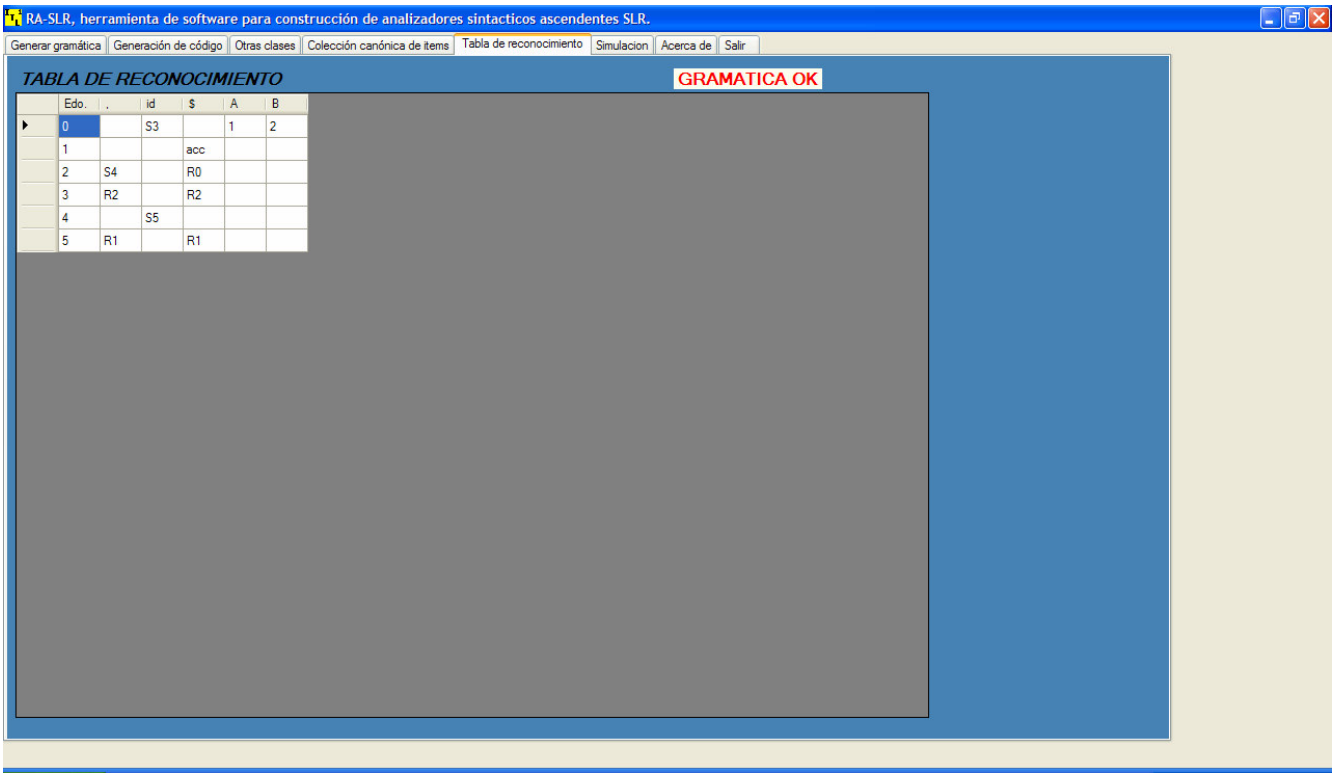
- El número de la producción que resulta de aumentar la gramática es el **-1**. Para nuestro caso la producción `A'-> A` tiene el número -1.
- El criterio para enumerar las producciones de la gramática, es iniciar con 0 la primera, 1 la segunda, hasta llegar a la última. Si tenemos una gramática con **n** producciones, la última tendrá el número **n-1**.

El número asignado a cada producción la podemos obtener de la interfase mostrada en la pestaña *Generar gramática*.

18. VISUALIZACIÓN DE LA TABLA DE RECONOCIMIENTO.

Otra de las tareas agregadas a *RA-SLR* es la visualización de la tabla de reconocimiento. La tabla de reconocimiento incluye las *acciones* y los *gotos* para cada estado generado por la colección canónica de items.
Recordemos que un estado es un conjunto de items en la colección canónica de items.
Para acceder a la tabla de reconocimiento generada por RA-SLR, es necesario que el estado de la gramática sea igual a OK. Si el estado es DEFICIENTE no será generada la tabla.
La pestaña con leyenda *Tabla de reconocimiento* permite la visualización de la tabla de reconicimiento, incluyendo a los estados, los *vts* en las *acciones* y los *vns* para los *gotos*.
La figura #18.1 muestra la tabla de reconocimiento para la gramática :

```
A -> B
B -> B , id
B -> id
```



Edo.	,	id	$	A	B
0		S3		1	2
1			acc		
2	S4		R0		
3	R2		R2		
4		S5			
5	R1		R1		

Fig. No. 18.1 Tabla de reconocimiento para la gramática con símbolo de inicio *A*.

19. RESTRICCIÓN IMPORTANTE.

Quizá el lector ya habrá observado que la gramática que hemos puesto de ejemplo en las 3 secciones anteriores :

```
A -> B
B -> B , id
B -> id
```

Puede ser simplificada a 2 producciones :

```
B -> B , id
B -> id
```

Y es correcto, el lenguaje generado por la gramática es el mismo. En esta primera versión de *RA-SLR* tenemos que observar esta restricción.

Podemos generalizar que cuando el símbolo de inicio de la gramática también sea encontrado en el miembro derecho de la producción, debemos agregar una producción que retrase la sustitución de este símbolo de inicio. Lo anterior lo hicimos cuando agregamos la producción :

```
A -> B
```

A la gramática :

```
B -> B , id
B -> id
```

Veamos otro ejemplo :

```
K -> K [ L ]
K -> [ L ]
L -> id
L -> num
```

La gramática anterior genera el lenguaje de las dimensiones de un arreglo en C. Cuando efectuamos la simulación en *RA-SLR* tendremos siempre un error, fig#19.1 :

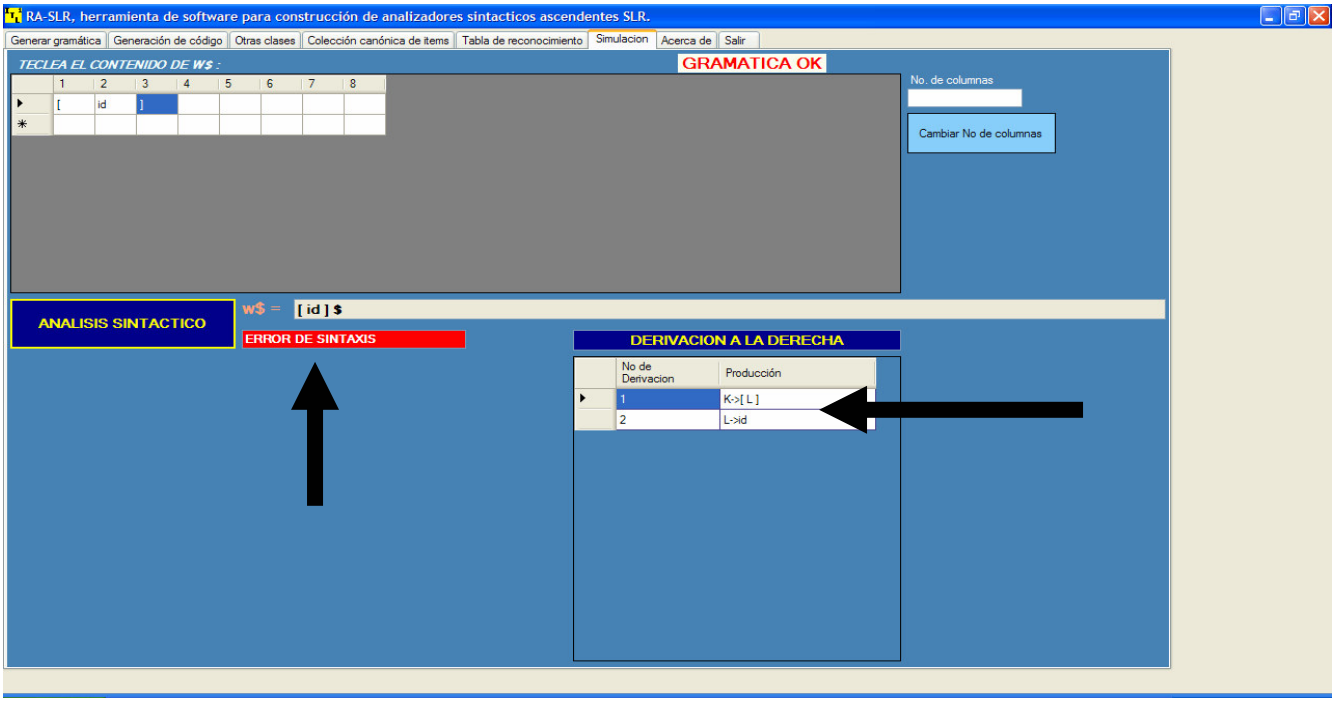


Fig. No. 19.1 “Error que no es un error”, recordando a *Mario Moreno*.

En la figura #19.1 vemos que la derivación a la derecha producida por el simulador integrado en *RA-SLR* muestra que el resultado de la simulación "ES UN ERROR QUE NO ES UN ERROR".

Para corregir esta deficiencia debemos agregar la producción `J -> K` a la gramática original. Ahora el símbolo de inicio es *J*.

```
J -> K
K -> K [ L ]
K -> [ L ]
L -> id
L -> num
```

Si ingresamos esta nueva gramática con la restricción observada, dentro de la pestaña de simulación tenemos la respuesta correcta deseada, fig#19.2.

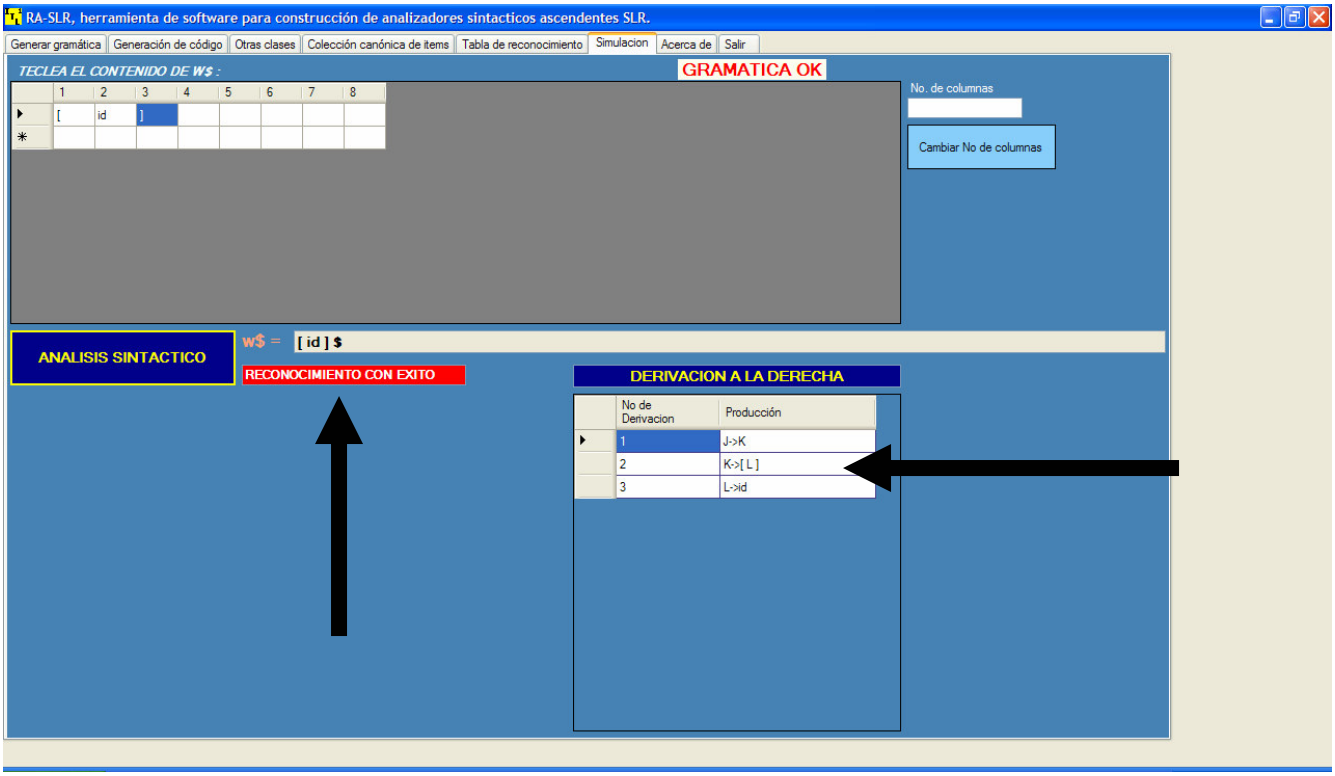


Fig. No. 19.2 Evitamos el "error que no es un error".

20. SIMULANDO EL RECONOCIMIENTO DE UNA SENTENCIA.

Una vez que hemos ingresado la gramática, podemos simular el reconocimiento usando la pestaña con leyenda *Simulación*.
Esta interfase permite ingresar una sentencia compuesta de tokens. Los tokens deberán ser conocidos por el usuario del RA-SLR.
Veamos el ejemplo de la sección anterior. la gramática usada sirve para reconocer las repeticiones de dimensión en una declaración de un arreglo.

```
J -> K
K -> K [ L ]
K -> [ L ]
L -> id
L -> num
```

Es fácil ver que la sentencia mínima es :

```
[id]
```

Las siguientes 2 serían :

```
[id][id] y
[id][id][id]
```

Ingresemos a la pestaña Simulación e ingresemos la 3 sentencia :

```
[id][id][id]
```

Notemos que necesitamos 9 celdas que acepten cada una de ellas a los tokens en la sentencia. Las celdas por omisión son 8. Lo primero que debemos hacer es asignar el valor de 9 a las celdas en la rejilla, usando el botón con leyenda *Cambiar No de columnas*. la figura #20.1 muestra la acción del botón cuando es leído un 9 en la ventana de lectura de *No. de columnas*.

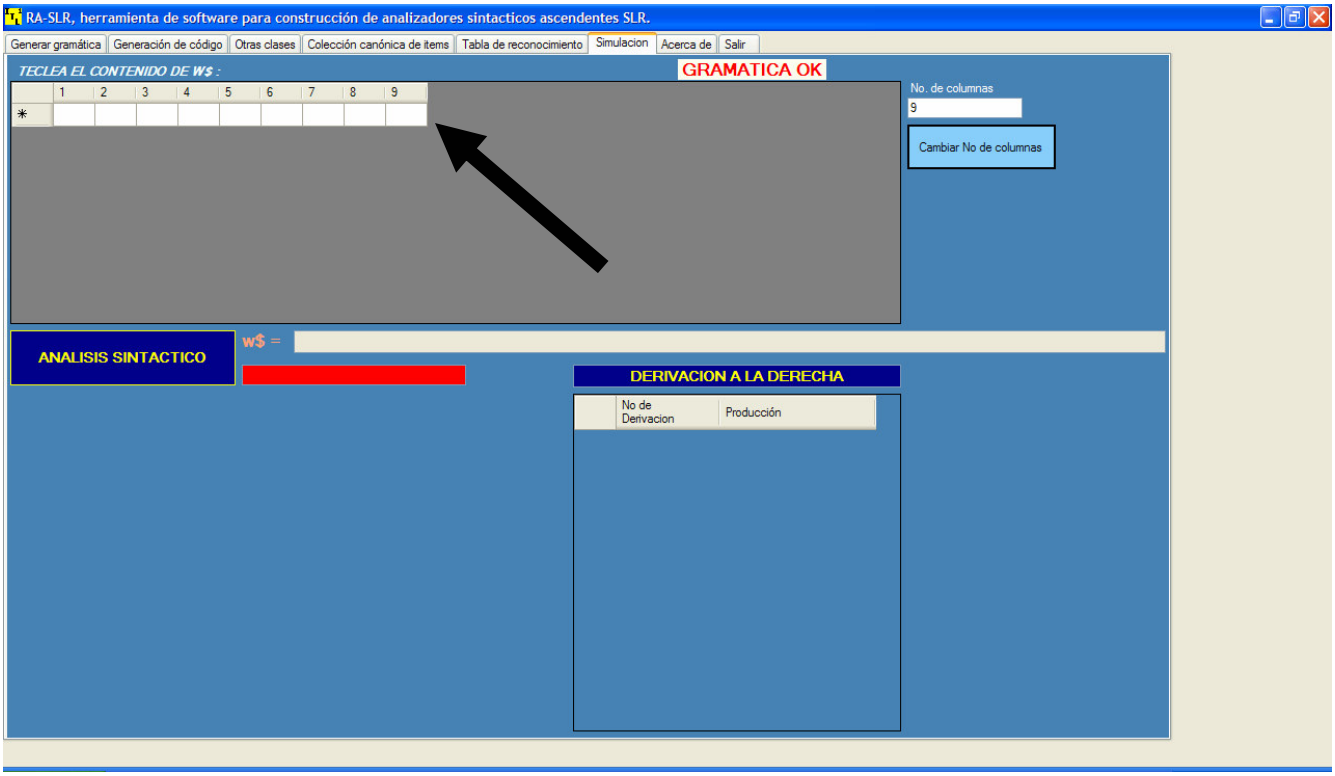


Fig. No. 20.1 Cambio del # de columnas a 9.

Ahora ingresemos la sentencia `[id][id][id]` en la rejilla con leyenda *TECLEA EL CONTENIDO DE W$*.

Luego hacemos click en el botón con leyenda ANALISIS SINTACTICO y obtenemos el resultado del análisis. El análisis puede ser exitoso o erróneo.
La derivación a la derecha que produce el reconocedor ascendente es mostrada en la rejilla con leyenda DERIVACION A LA DERECHA.
Las figuras #20.2 y #20.3 muestran un reconocimiento exitoso y uno erróneo.

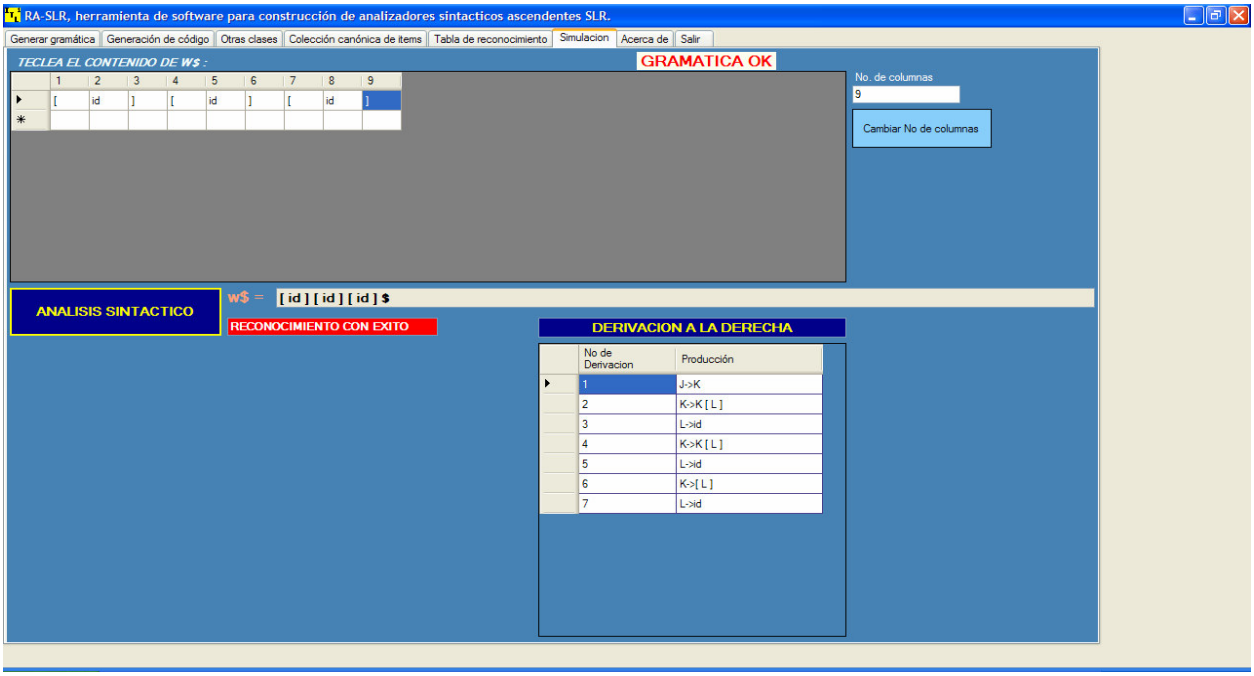


Fig. No. 20.2 Éxito en el reconocimiento de la sentencia [id][id][id].

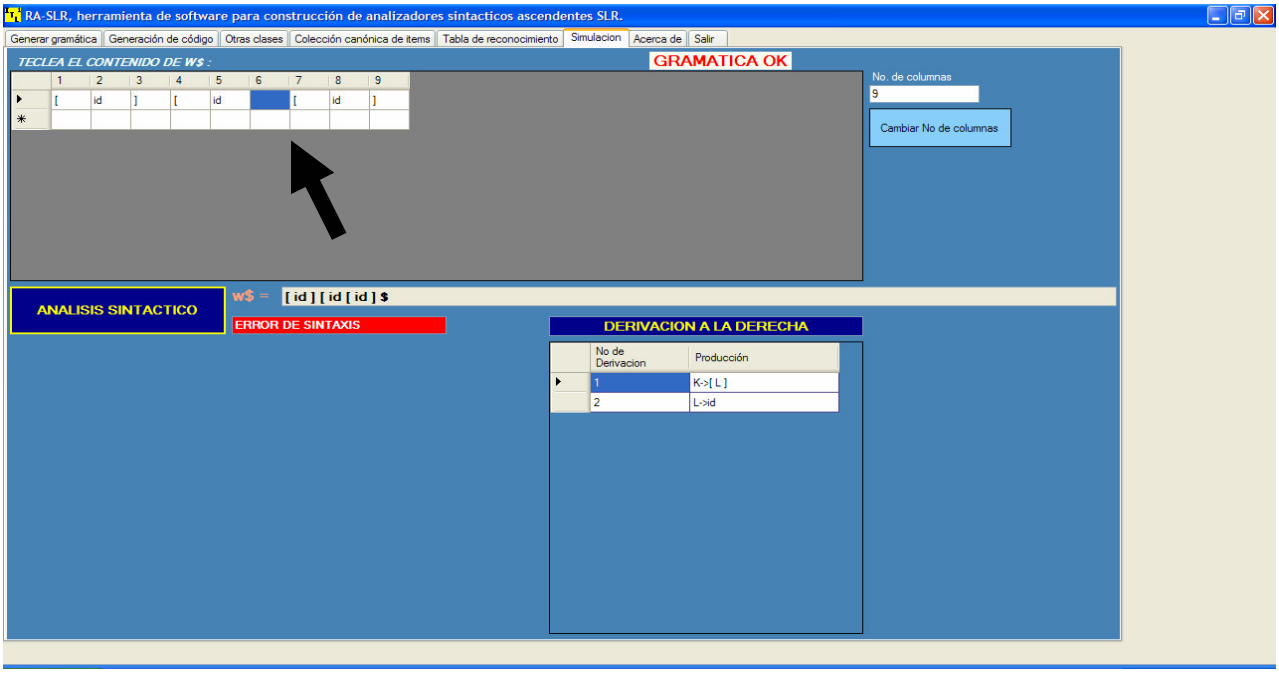


Fig. No. 20.3 Error de sintaxis en el reconocimiento de la sentencia [id][id[id], falta del corchete que cierra al segundo id.