Software didáctico para la construcción de analizadores sintacticos descendentes no recursivos predictivos.

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 un software didáctico denominado *RD-NRP* cuyo objetivo es ayudar en el proceso de enseñanza-aprendizaje del tema "análisis sintactico" dentro del curso de Programación de Sistemas. Además, permite la generación de código de una clase denominada *SintDescNRP* que es usada para definir objetos cuyo fin es analizar sintacticamente un grupo de sentencias. El software ofrece las facilidades : (1) Ingreso de una gramática de contexto libre no ambigua, (2) Transformación de la gramática : eliminación de la recursividad a la izquierda y factorización a la izquierda, (3) Obtención de los conjuntos : PRIMEROS y SIGUIENTES, (4) Construcción de la tabla M de reconocimiento, (5) Simulación del reconocedor descendente, y (6) generación de código en C# para la clase *SintDescNRP*. La teoría de base para la construcción del software fue tomada del libro "del dragón" de Aho, Sethi y Ullman. El trabajo termina mostrando la construcción de un analizador sintactico no recursivo predictivo escrito en C#, que usa el código producido por los sofware's didácticos SP-PS1 y RD-NRP.

I N D I C E.

1 Introducción.

El software didáctico RD-NRP es un ejecutable en ambiente Windows, escrito en C# de Visual Studio 2005 de MicroSoft. Su uso se suscribe a la materia de Programación de Sistemas en su tema Análisis Sintactico-Reconocedores Descendentes. El usuario de este software tiene acceso a las siguientes utilidades :

- Ingreso de una gramática de contexto libre no ambigua.
- Transformación de la gramática eliminando la recursividad a la izquierda E.R.I., haciendo no recursivo al reconocedor.
- Transformación de la gramática factorizando a la izquierda, haciendo predictivo al reconocedor.
- Obtención del conjunto PRIMERO para cada no terminal de la gramática transformada.
- Obtención del conjunto SIGUIENTE para cada no terminal de la gramática transformada.
- Construcción de la tabla M de reconocimiento, usada por el algoritmo del reconocedor descendente no recursivo predictivo.
- Simulación del reconocimiento de una sentencia, visualizando la derivación a la izquierda producida por el reconocedor descendente.
- Generación del código para la clase SintDescNRP, que permite definir objetos dentro de una aplicación C#, que analizan sintacticamente un grupo de sentencias que cumplen con la sintaxis descrita por una gramática de contexto libre.

Durante la exposición de este trabajo, veremos cómo se usa, la utilidad del RD-NRP, además de terminar explicando la construcción de un anañizador sintactico descendente no recursivo predictivo, que utiliza el código generado por este software didáctico.

2 Ingreso de una gramática de contexto libre no ambigua.

Esta característica representa el punto de inicio para la construcción del reconocedor descendente. Proporciona al usuario las funciones :

- Ingreso de la gramática desagrupada.
- Configuración del número de yes.
- Inserción de renglones para una edición mas amigable.
- Limpieza o inicialización de la rejilla de ingreso de la gramática, para iniciar en 0 producciones.
- Carga de una gramática previamente tecleada y salvada.
- Salvar la gramática que se ha tecleado en la rejilla de ingreso.
- Análisis de la gramática, visualizando sus producciones, los símbolos no terminales y los terminales. Si hay errores son comunicados al usuario, permaneciendo la gramática en un estado denominado DEFICIENTE. Si no existieron errores en el análisis, la gramática tiene un estado OK.

La figura #2.1 muestra la interfase para una gramática de asignación, analizada y con estado OK. El resultado del análisis indica 10 producciones, 10 símbolos terminales y 4 símbolos no terminales. Las producciones agrupadas de la gramática son :

```
A -> id = E ;

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

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

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

Para llegar a la interfase mostrada en la figura 2.1 debemos ingresar a cada producción de la gramática utilizando un renglón de la rejilla de entrada, para cada componente de la producción. Es decir, primero ingresamos al miembro izquierdo de la producción que será invariablemente un no terminal, debido a que manejamos sólo gramáticas de contexto libre. Los símbolos no terminales las denotamos con una letra mayúscula. En la columna siguiente debemos ingresar el número de yes de la producción. Las Y's son los símbolos terminales o no terminales que conforman al miembro derecho de la producción. Por último debemos ingresar a cada Y del miembro derecho de la producción : Y1, Y2, Y3, …, Yn.

La figura #2.2 muestra la interfase que contiene el estado de la gramática después de haber ingresado a cada producción de la gramática.

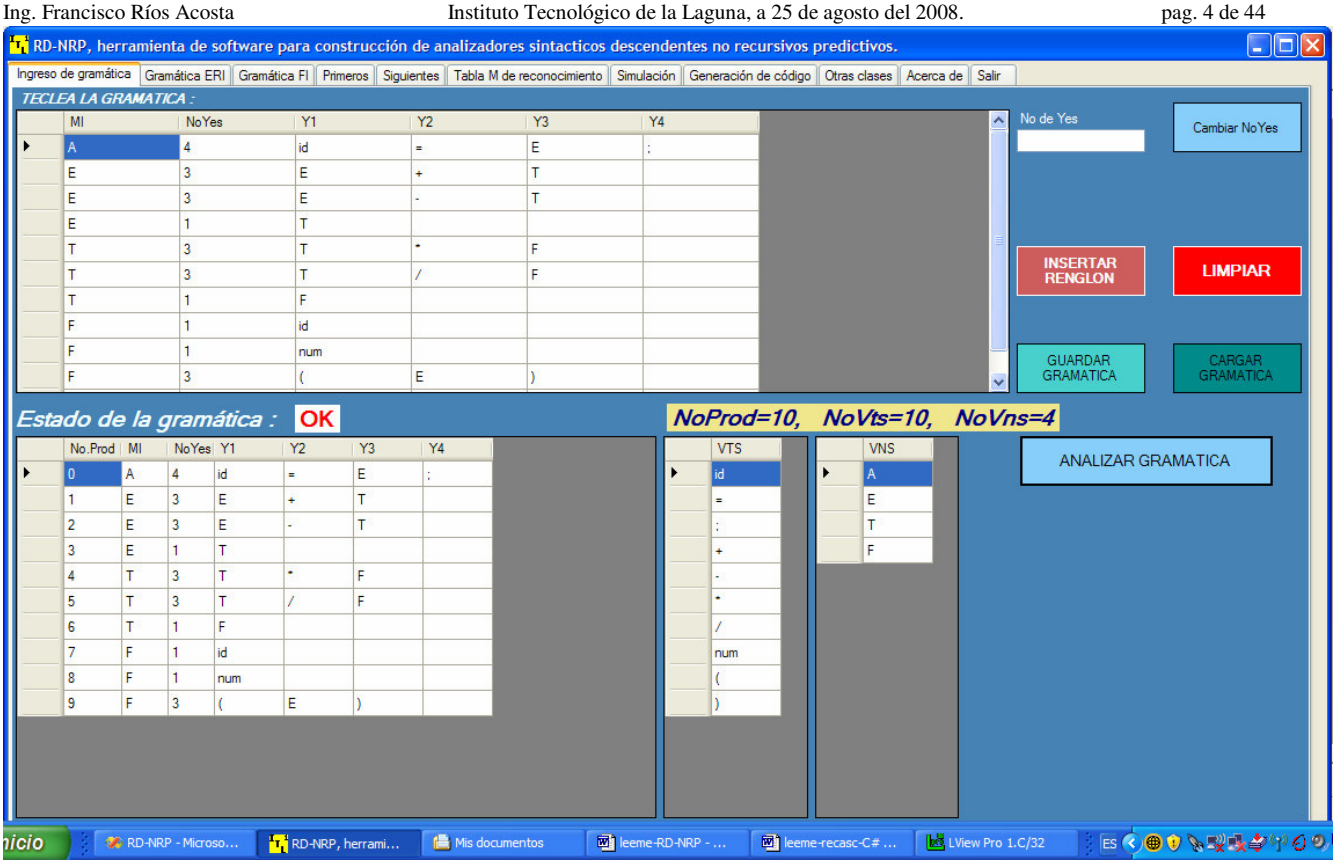


Fig. No. 2.1 Interfase de ingreso de una gramática de asignación con estado OK, después de su análisis.

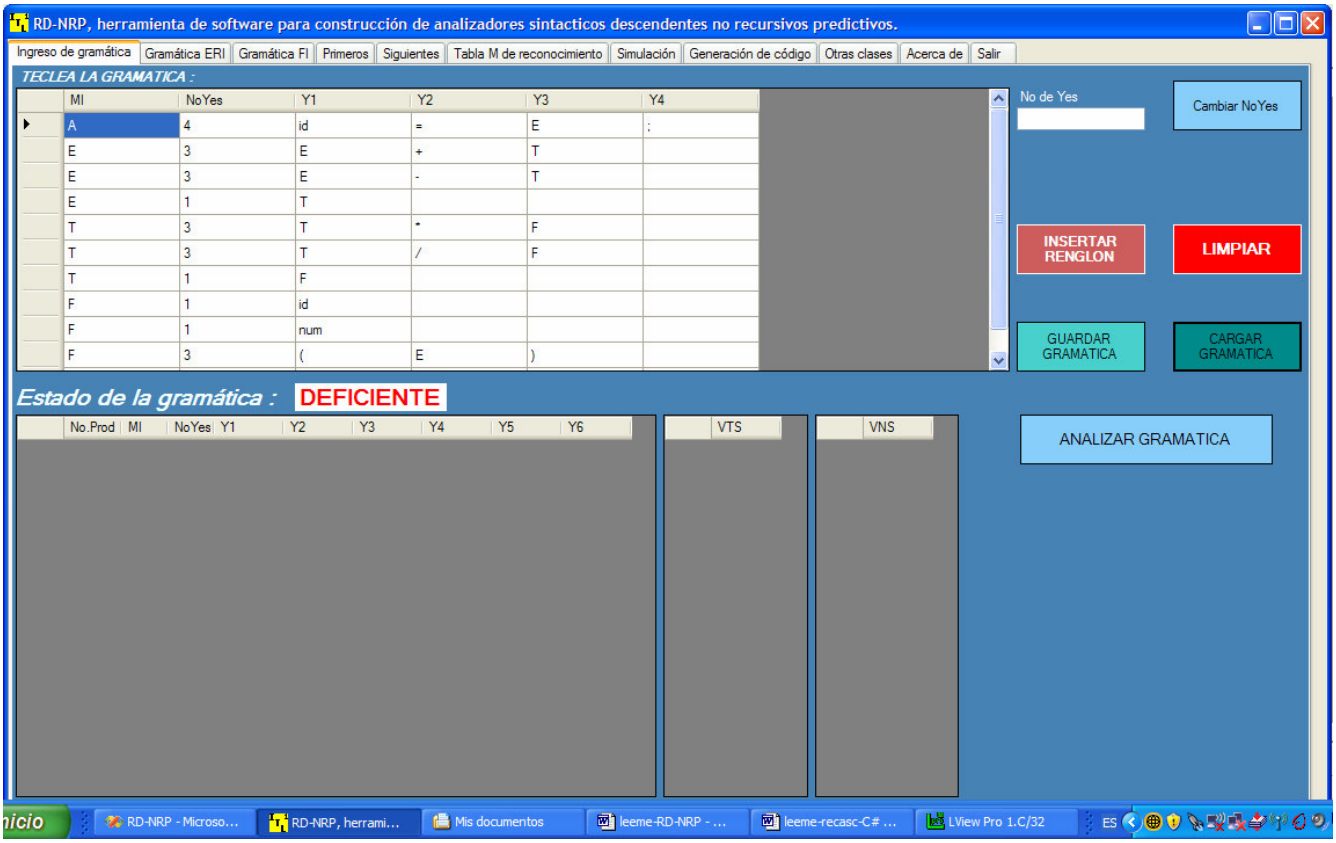


Fig. No. 2.2 Interfase de ingreso de una gramática de asignación, antes de su análisis.

En ocasiones tenemos una gramática cuyas producciones tienen un número mayor de yes que el valor por defecto : 6. Cuando esto sucede, cambiamos el número de yes que presenta la rejilla de entrada, tecleando el valor de yes en la centana con leyenda *No de yes*. Luego debemos hacer click sobre elbotón correspondiente *Cambiar No de yes*.

La inserción de un renglón trabaja de la misma forma en que lo hacen todos los programas, sólo debes seleccionar el renglón de la rejilla donde requieres la inserción, y luego haces click sobre el botón INSERTAR RENGLON. La eliminación de un renglón se realiza seleccionando al renglón que queremos eliminar haciéndo click sobre la columna de la izquierda de la rejilla de ingreso, mostrando la columna un aspecto de opresión según la figura #2.3.

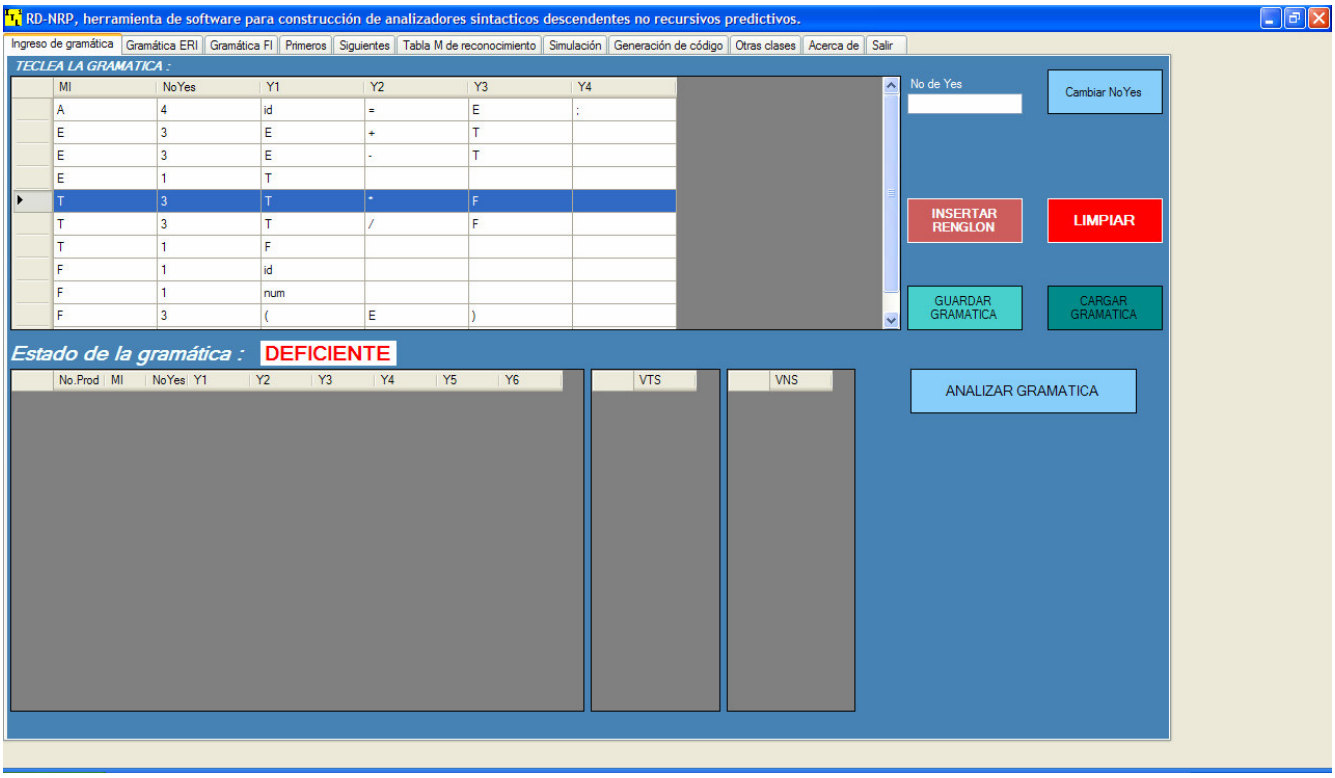


Fig. No. 2.3 Selección del renglón T->T*F para su eliminación.

Una vez seleccionado el renglón a eliminar, debemos teclear la combinación CTRL-ALT-SUPR.

Para limpiar la rejilla, es decir eliminar todos los renglones ingresados, sólo debemos hacer click sobre el botón LIMPIAR.

El análisis de la gramática se efectúa cuando hacemos un click sobre el botón ANALIZAR GRAMATICA. La acción del botón permite almacenar la información de la gramática ingresada : número de símbolos terminales, número de símbolos no terminales, número de producciones. También se efectúa un reconocimiento sobre lo ingresado de manera que detectemos errores, entre los mas comunes : no. de yes no es un número, el miembro izquierdo debe ser un no terminal, no se aceptan blancos en celdas intermedias cuyo contexto no esté libre. En la figura #2.4 se ha provocado un error al teclear el número de yes.

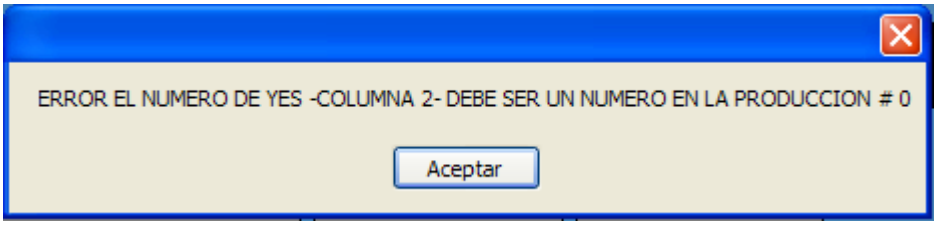


Fig. No. 2.4 No de yes no válido.

Cuando el análisis de la gramática no produce errores, el RD-NRP responde con una caja de mensajes donde indica que el estado de la gramática es OK, Figura #2.5.

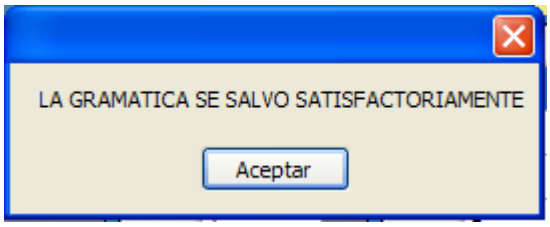


Fig. No. 2.5 Análisis de la gramática OK.

El botón GUARDAR GRAMATICA tiene la tarea de salvar en un archivo binario las producciones de la gramática. Puede ser accionado para una gramática deficiente o para una gramática con estado OK. Para recuperar la gramática hacemos click sobre el botón CARGAR GRAMATICA, cuya función es abrir el archivo donde se encuentra la gramática previamente almacenada y depositarla en la rejilla de entrada para su edición y análisis. La figura #2.6 muestra la caja de diálogo para cargar una gramática.

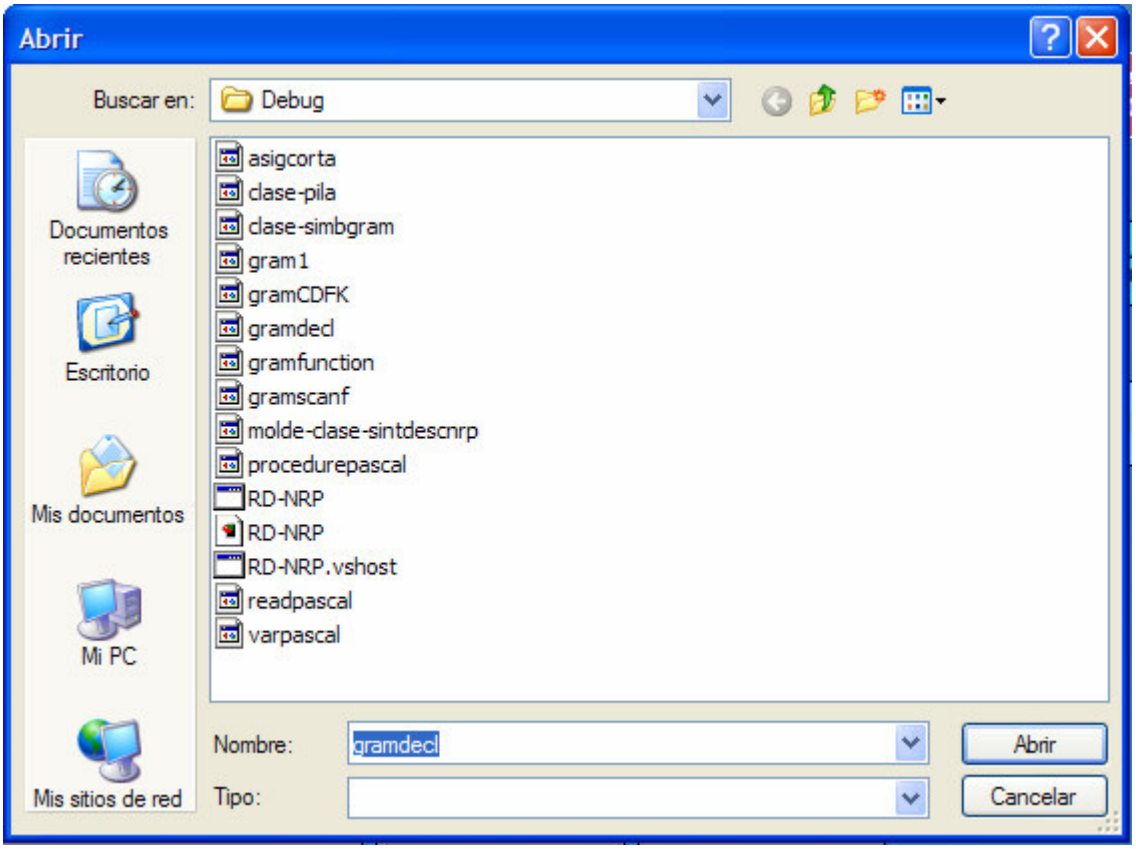


Fig. No. 2.6 Carga de la gramática que reside en el archivo *gramdecl*.

3 Eliminación de la recursividad a la izquierda, E.R.I.

La eliminación de la recursividad a la izquierda E.R.I. es la primera transformación de la gramática original ingresada, cuyo fin es convertir al reconocedor descendente de forma que sea no recursivo. La recursividad requiere de cierto cuidado en su manejo, ya que se puede caer en un ciclo infinito. E.R.I. es necesaria para construir la tabla M de reconocimiento usada por el reconocedor descendente para derivar la sentencia de entrada –reconocerla-.

El RD-NRP inicia visualizando la gramática agrupada ingresada previamente, figura #3.1.

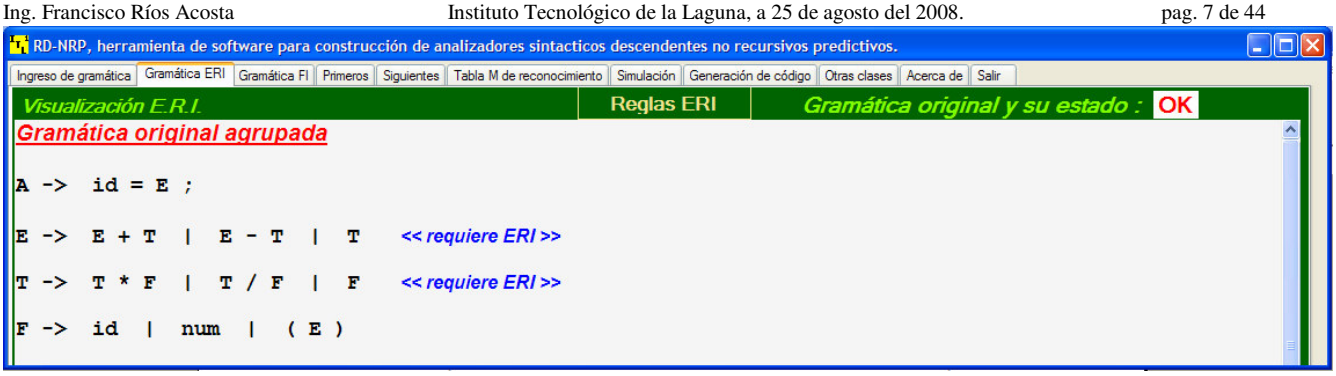


Fig. No. 3.1 Gramática original agrupada.

Observemos que también se marca a las producciones agrupadas, con un letrero que indica si el grupo de producciones requiere de la eliminación de recursividad a la izquierda. RD-NRP visualiza estas reglas E.R.I. cuando accionamos al botón *Reglas ERI*, figura #3.2.

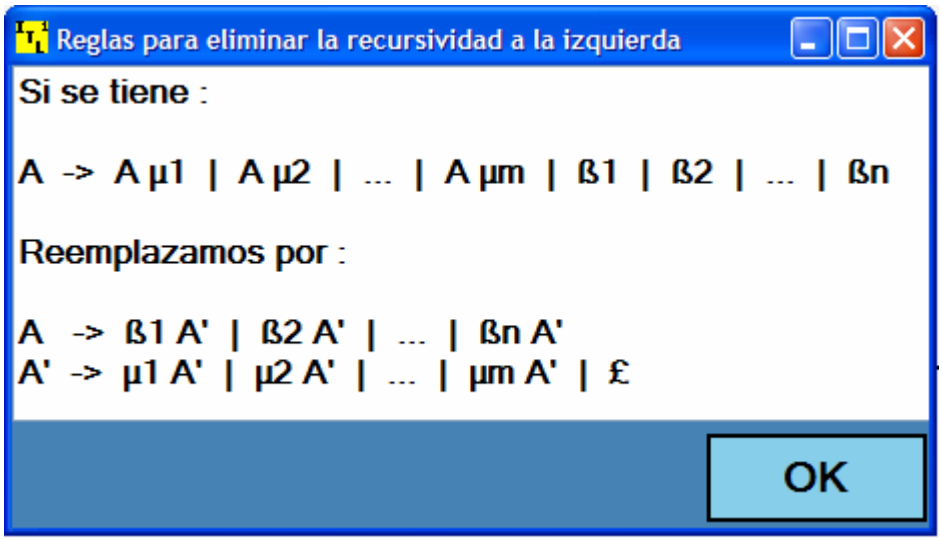


Fig. No. 3.2 Reglas E.R.I..

Luego que se conoce cuales de las producciones necesitan la E.R.I., el RD-NRP muestra la aplicación de las reglas E.R.I. paso a paso, identificando a cada elemento de la producción usando un color que es comparado con las reglas E.R.I..
Si se tiene :
A -> A µ1 | A µ2 | ... | A µm | ß1 | ß2 | ... | ßn
Reemplazamos por :
A -> ß1 A' | ß2 A' | ... | ßn A'
A' -> µ1 A' | µ2 A' | ... | µm A' | £

RD-NRP identifica con el color verde a µ1, µ2, …, µm y con el color azul a las ß1, ß2,…, ßn. También muestra los reemplazos emanados de las reglas E.R.I. indicando con mensajes escritos las sustituciones hechas por dicho software, como es mostrado enseguida. También se puede observar lo anterior en la figura #3.3 para la gramática de asignación de ejemplo.

```
E  ->  E + T  |  E - T  |  T     <<SI requiere ERI>>
A  ->  A µ1   |  A µ2   |  ß1     Identificamos µ's y ß's para aplicar ERI

Ahora aplicamos el reemplazo según las reglas ERI
E  ->  T E'
E' ->  + T E'   |  - T E'   |  £
Estas 4 nuevas producciones con la variable sintactica E', sustituyen a las producciones con RI : E ->
E + T  |  E - T  |  T
```

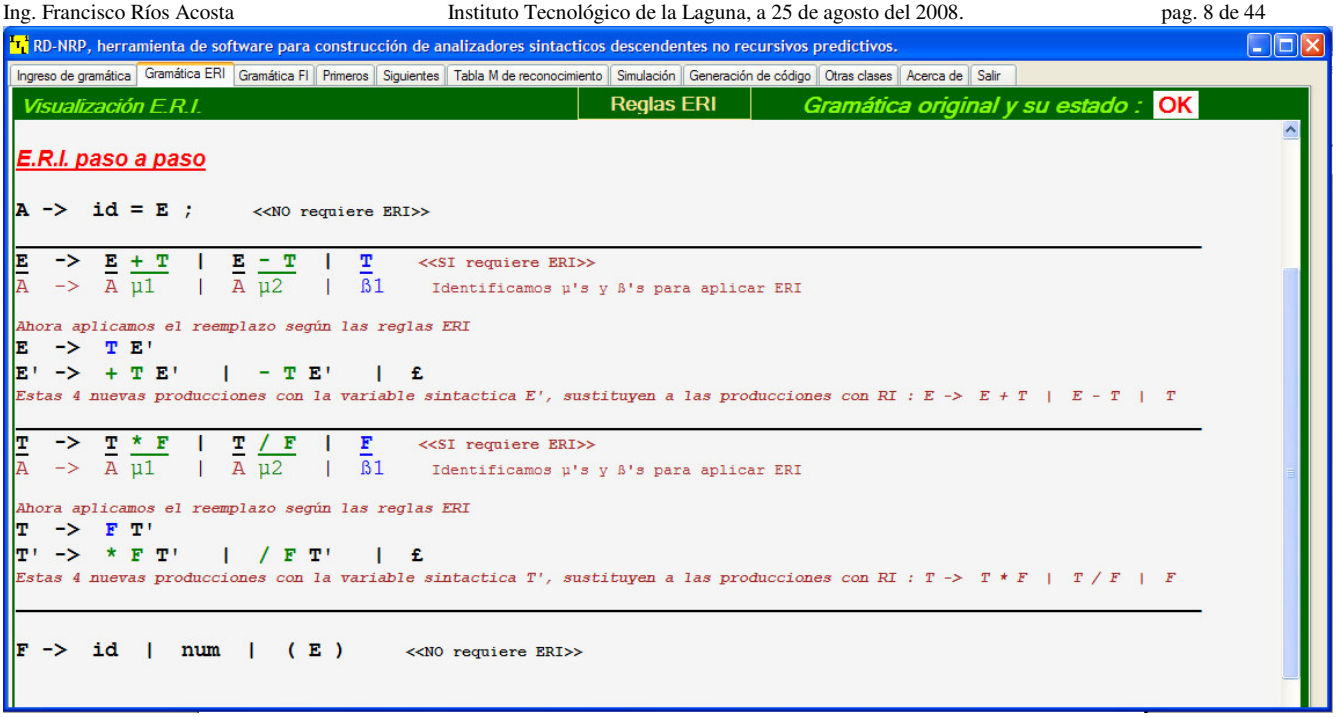


Fig. No. 3.3 Interfase del RD-NRP para la E.R.I. de la gramática de asignación, paso a paso.

NOTA .- La E.R.I. aplicada por el software RD-NRP no permite la eliminación para recursividades a la izquierda incrustadas. Existe otro algoritmo que permite hacer la E.R.I. para este tipo de gramáticas. Este algoritmo no es cubierto en esta primera versión del RD-NRP.

Después de mostrar la E.R.I. paso a paso, es visualizada la gramática con E.R.I., figura #3.4.

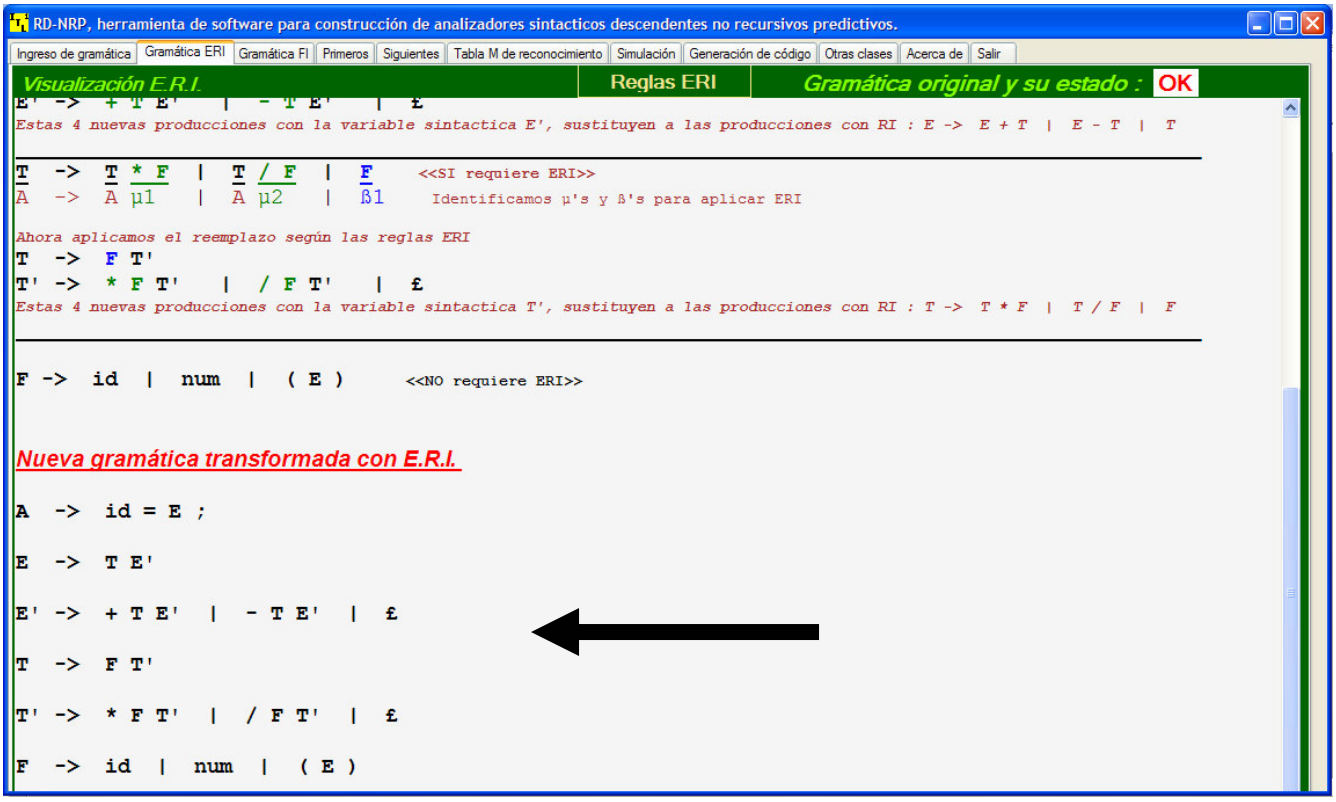


Fig. No. 3.4 Gramática E.R.I. resultante para sentencias de asignación.

4 Factorización a la izquierda, F.I.

La factorización a la izquierda se refiere a una transformación a la gramática E.R.I. de manera que el reconocedor sea predictivo. La F.I. evita al reconocedor la posibilidad de efectuar una decisión consistente en seleccionar una de varias producciones con prefijo común. Esta transformación también se requiere para construir un reconocedor descendente.
El RD-NRP inicia visualizando a la gramática E.R.I. agrupada, e identificando si un grupo de producciones requiere de la factorización a la izquierda, figura #4.1.

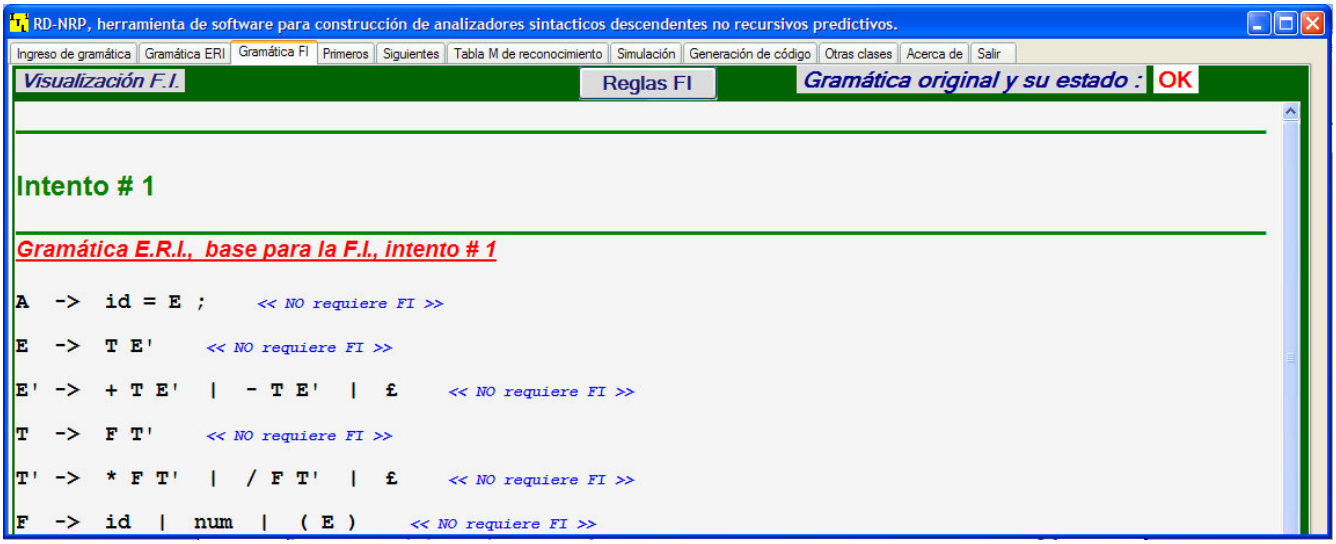


Fig. No. 4.1 Identificación de producciones que requieren F.I.

En la figura observamos que hay un rótulo indicando el número del intento de factorización a la izquierda. El intento se refiere al número de veces que intentamos factorizar a la izquierda, ya que algún grupo de producciones a veces requieren de mas de una factorización.
Podemos acceder a las reglas F.I. haciendo click sobre el botón Reglas F.I., figura #4.2.

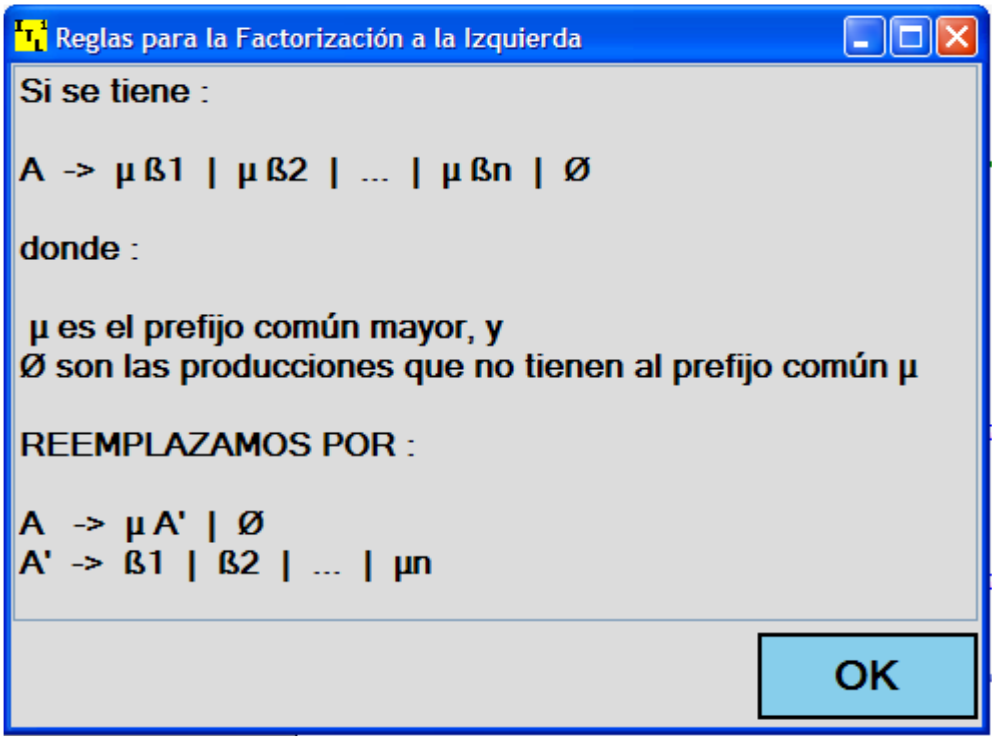


Fig. No. 4.2 Reglas F.I.

Para nuestro caso de ejemplo, la gramática E.R.I. no necesita de factorización por lo que el siguiente paso que realiza el RD-NRP, es visualizar la gramática resultante, figura #4.3 indicando previamente que la gramática no necesitó de F.I.

Fig. No. 4.3 Gramática E.R.I. es igual a la gramática F.I.

Veamos ahora la gramática para declaración de variables en C con la eliminación de la recursividad a la izquierda :

```
D  ->  T L ;
T  ->  int  |  float
L  ->  id L'  |  id H L'
L' ->  , id L'  |  , id H L'  |  £
H  ->  [ K ] H'
H' ->  [ K ] H'  |  £
K  ->  num
```

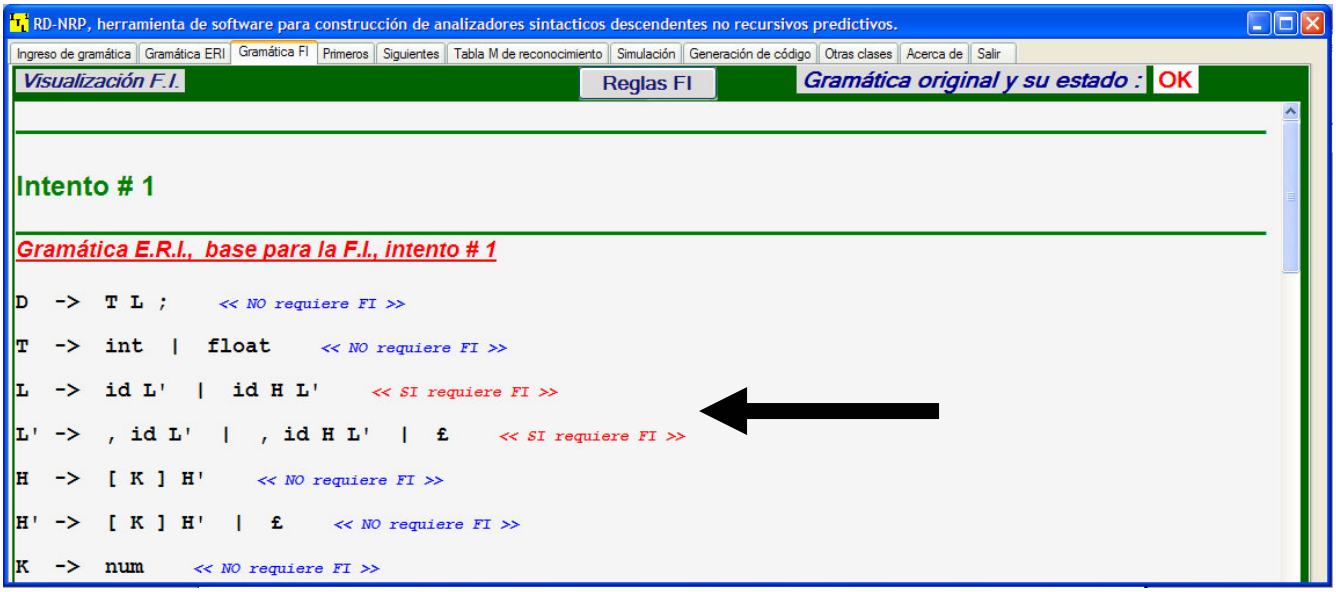


Fig. No. 4.4 Identificación de producciones que requieren F.I., primer intento.

En el primer intento, RD-NRP identifica 2 grupos de producciones que requieren F.I:, figura #4.5. El software separa a las producciones identificadas mediante el paso 1, para luego identificar y comparar contra la regla F.I. a cada una de las producciones que requieren F.I., proponiendo al prefijo común mayor, visualizando los reemplazos correspondientes.

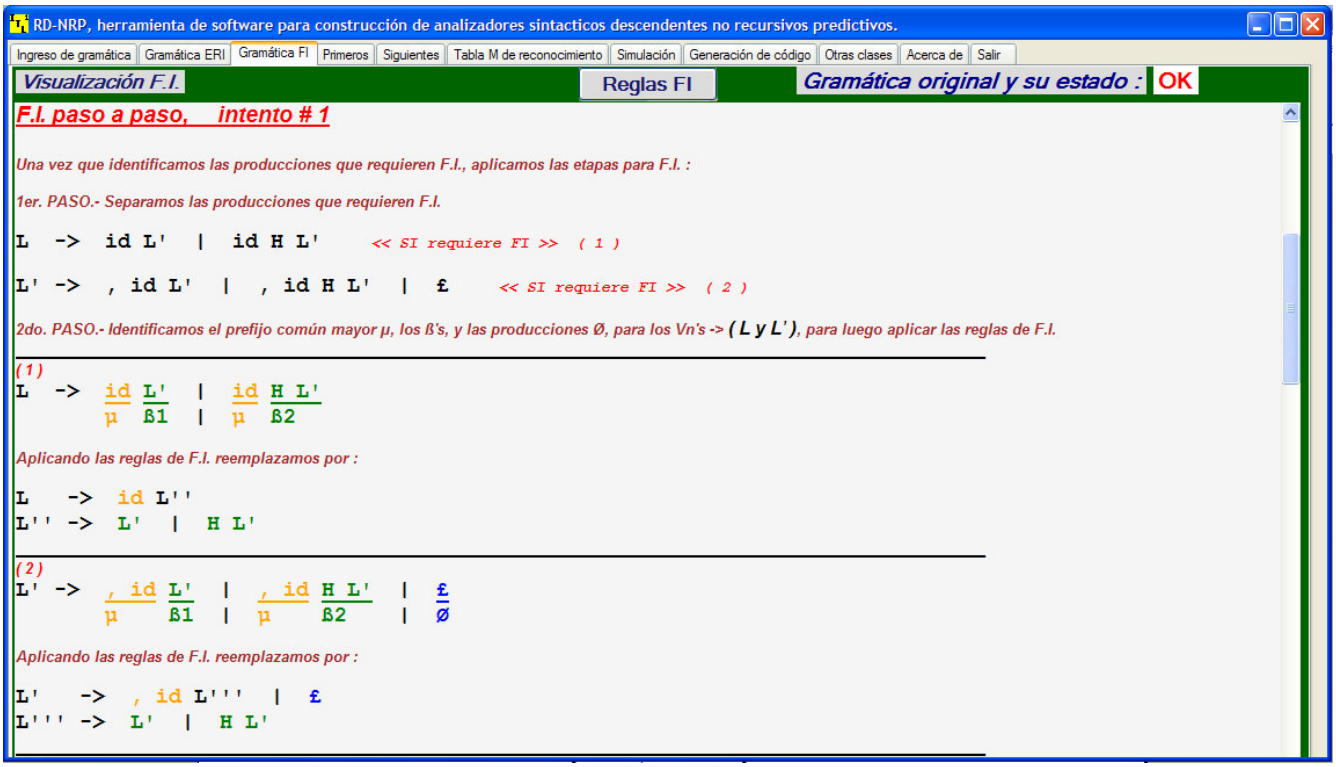


Fig. No. 4.5 Aplicación de las reglas F.I.

RD-NRP utiliza el color naranja para identificar al prefijo común mayor µ, al color verde para las ß's, y al color azul para identificar a las producciones Ø que no tienen el prefijo común. Por ejemplo :

```
L' ->   , id L'   |   , id H L'   |   £
        µ    ß1   |   µ    ß2     |   Ø
```

Aplicando las reglas de F.I. reemplazamos por :

```
L'   ->   , id L'''   |   £
L''' ->   L'   |   H L'
```

El siguiente paso que efectúa el programa RD-NRP, es buscar símbolos terminales iguales con el fin de simplificar a la gramática. En ocasiones cuando se aplican las reglas para F.I., sucede que las producciones para mas de un símbolo no terminal, son iguales en número y en Y's.

Antes de efectuar la eliminación -si existieran símbolos no terminales iguales-, RD-NRP visualiza la nueva gramática agrupada identificando los no terminales iguales.

```
D     ->   T L ;
T     ->   int   |   float
L     ->   id L''
L''   ->   L'   |   H L'          <<   L'' = L'''   >>
L'    ->   , id L'''   |   £
L'''  ->   L'   |   H L'          <<   L''' = L''   >>
H     ->   [ K ] H'
H'    ->   [ K ] H'   |   £
K     ->   num
```

Para nuestro caso de ejemplo, los no terminales L'' y L''' son iguales, figura #4.6.

```
L''  ->  L'  |  H L'
L''' ->  L'  |  H L'
```

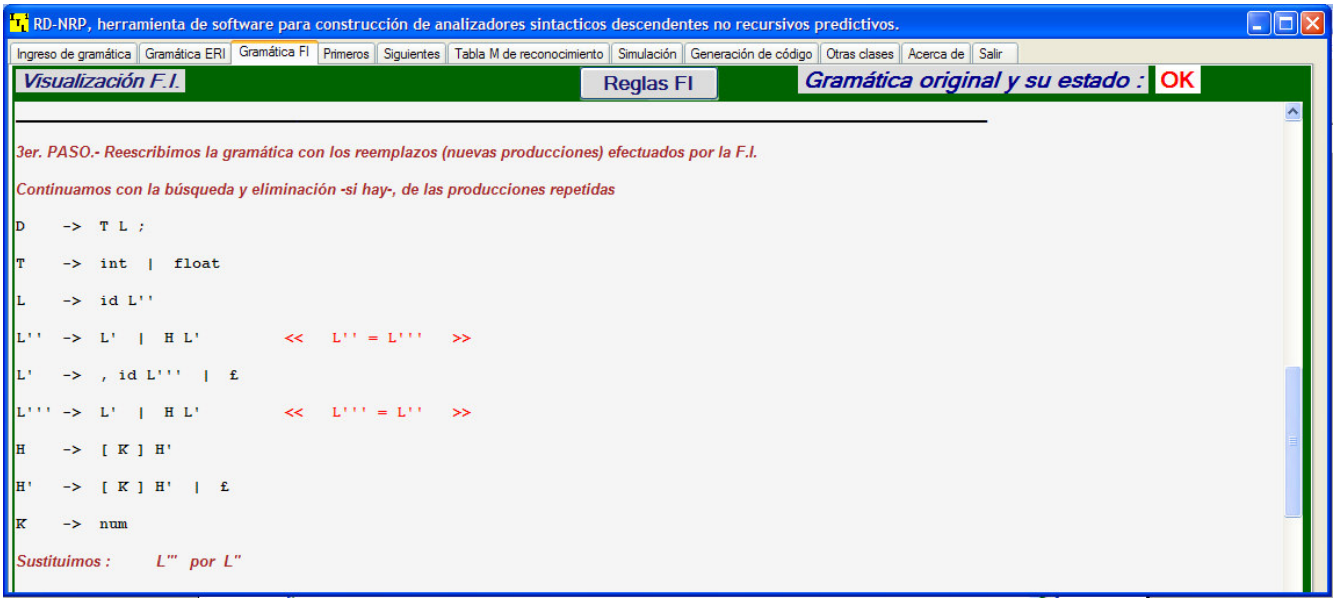


Fig. No. 4.6 L'' = L'''.

El software lo reconoce así, y elimina a uno de ellos. El criterio que utiliza el RD-NRP para efectuar la sustitución y la eliminación, es la longitud del identificador de los no terminales. En este ejemplo, se elimina al de longitud mayor L''' y se sustituyen sus ocurrencias en los miembros derechos de las producciones, por el no terminal L'', figura #4.7.

La gramática con la simplificación se muestra enseguida, continuando con la búsqueda de no terminales iguales hasta que ya no los encuentre.

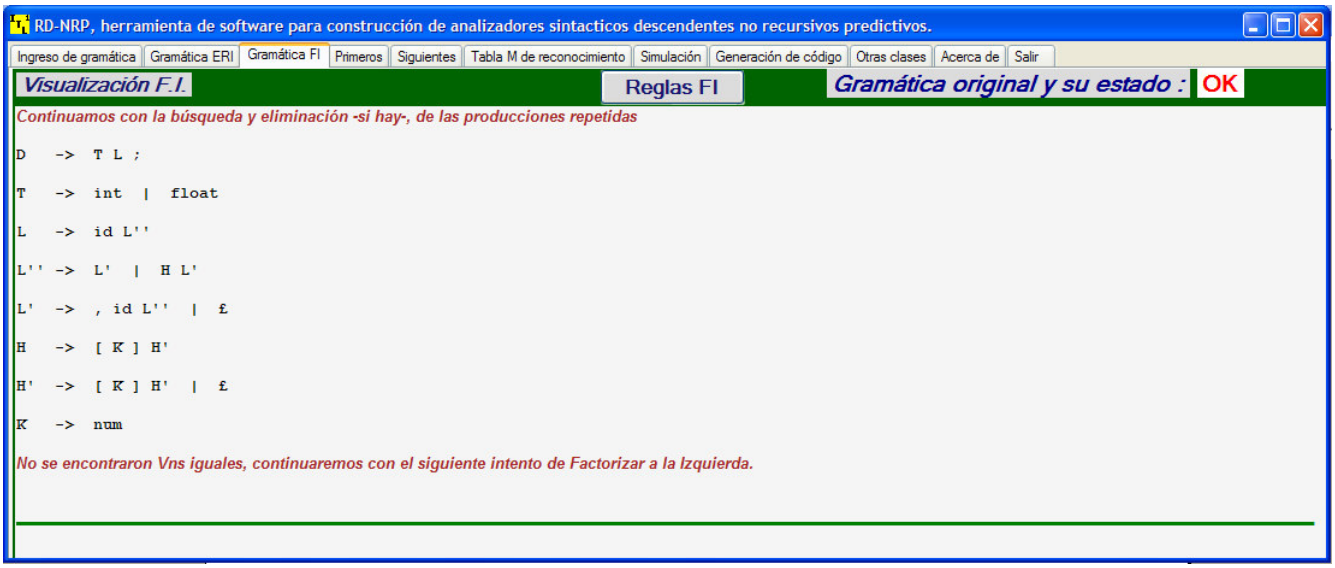


Fig. No. 4.7 Gramática simplificada con F.I., con eliminación de L''' y sustitución de L''' por L''.

Luego se sigue con el siguiente intento de factorización a la izquierda hasta que ya no existan grupos de producciones que demanden la factorización. La figura #4.8 indica que en el segundo intento, ya no existen porducciones que necesiten de la factorización a la izquierda.

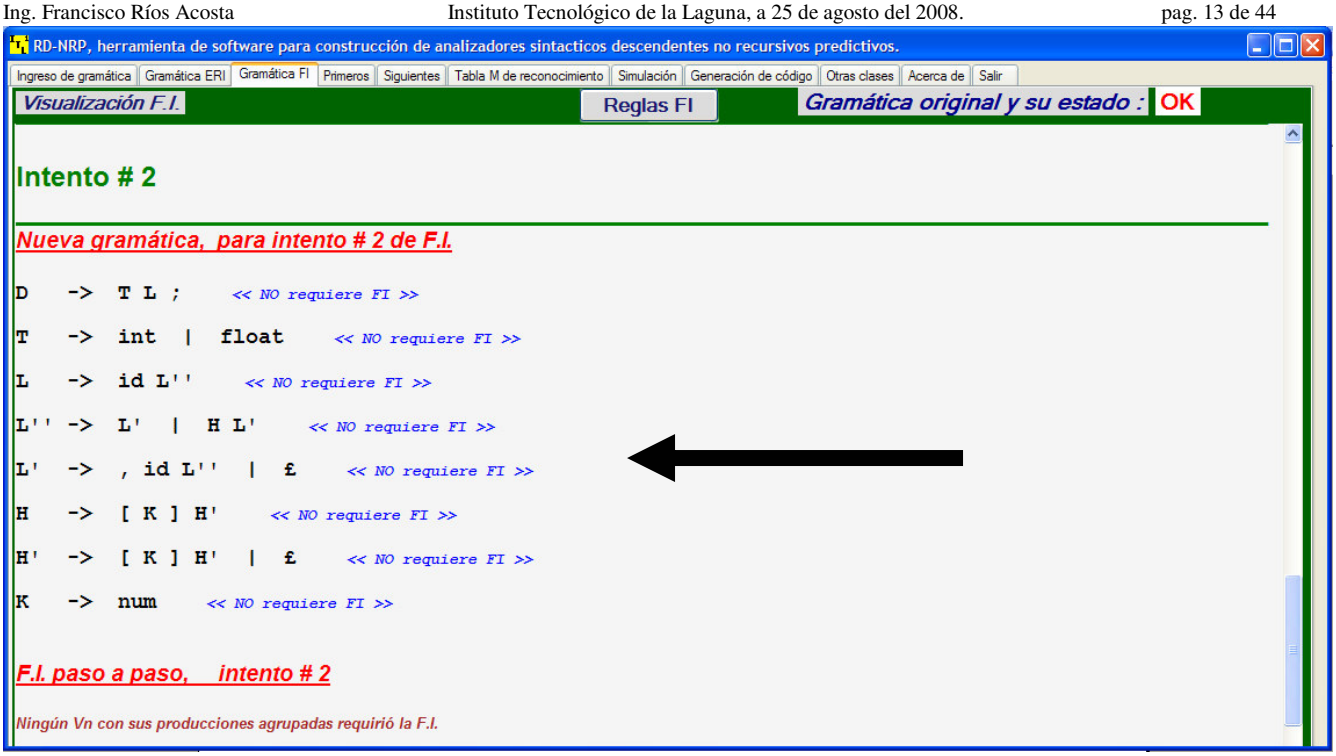


Fig. No. 4.8 Gramática en el segundo intento de F.I.

Una vez que ninguna otra producción requiera la F.I., el software RD-NRP muestra la gramática F.I. resultante, figura #4.9.

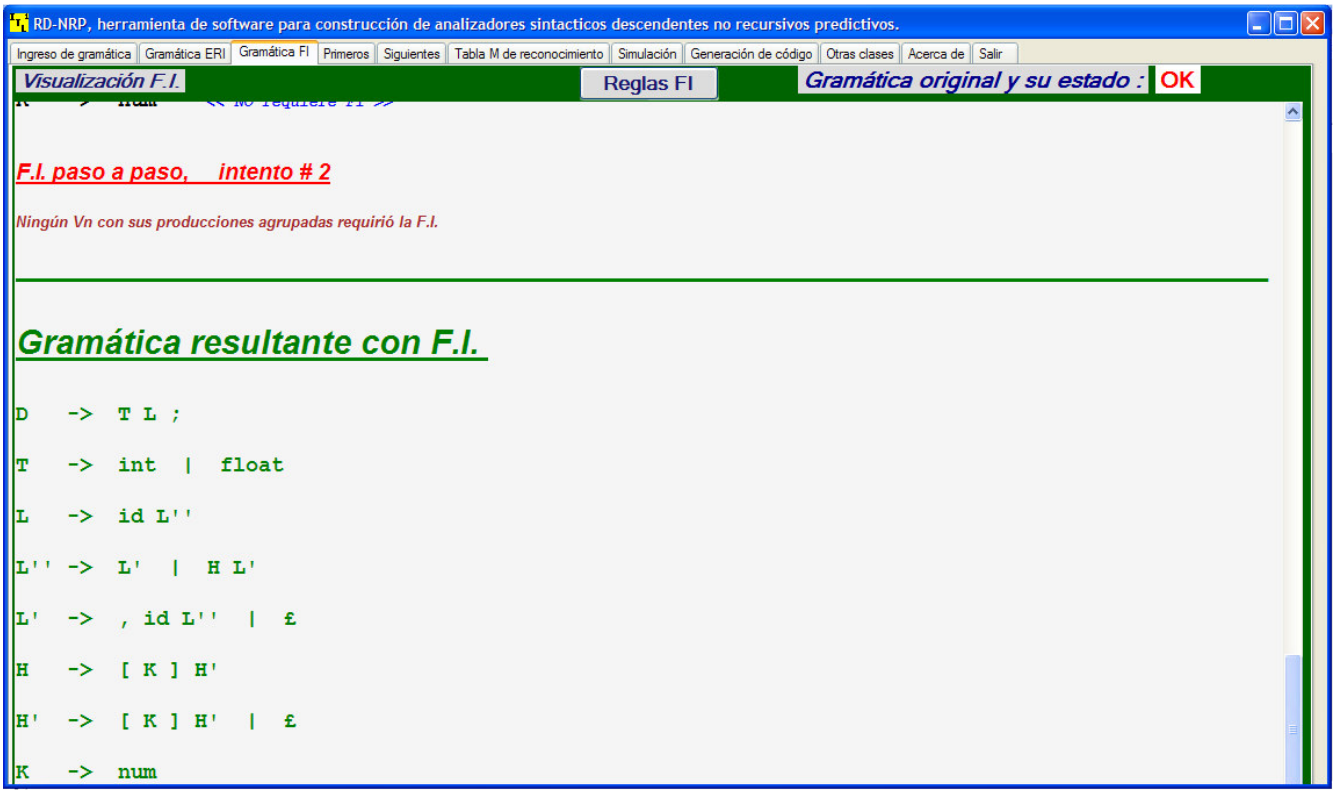


Fig. No. 4.9 Gramática F.I. final.

5 PRIMEROS.

La interfase presentada por RD-NRP para obtención de los PRIMEROS, consiste de 2 ventanas : una muestra a la gramática F.I. previamente calculada y la segunda ventana, es utilizada para visualizar la obtención de cada PRIMERO paso a paso, según lo vemos en la figura #5.1.

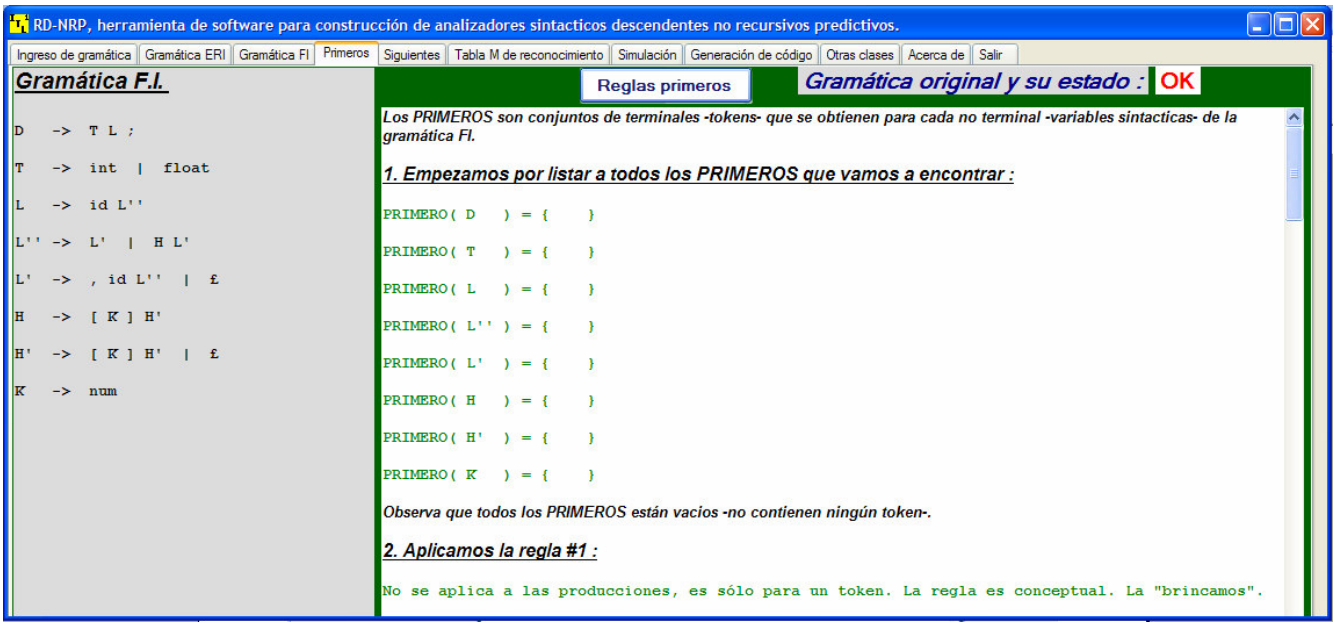


Fig. No. 5.1 Interfase cálculo de los PRIMEROS para la gramática con E.R.I. y con F.I..

El paso 1 que muestra RD-NRP en la ventana de la derecha, se visualizan inicialmente a los conjuntos PRIMERO para cada no terminal en la gramática F.I.. Estos PRIMEROS están vacios. El paso 2 visto en la figura #5.1 también lista la aplicación de la primera regla para el cálculo de los PRIMEROS, e indica que esta 1ra. regla no se aplica a las producciones sino a los símbolos terminales.

Las reglas para cálculo de los PRIMEROS –figura #5.2-, las contienen la ventana que es visualizada cuando accionamos el botón *Reglas primeros*.

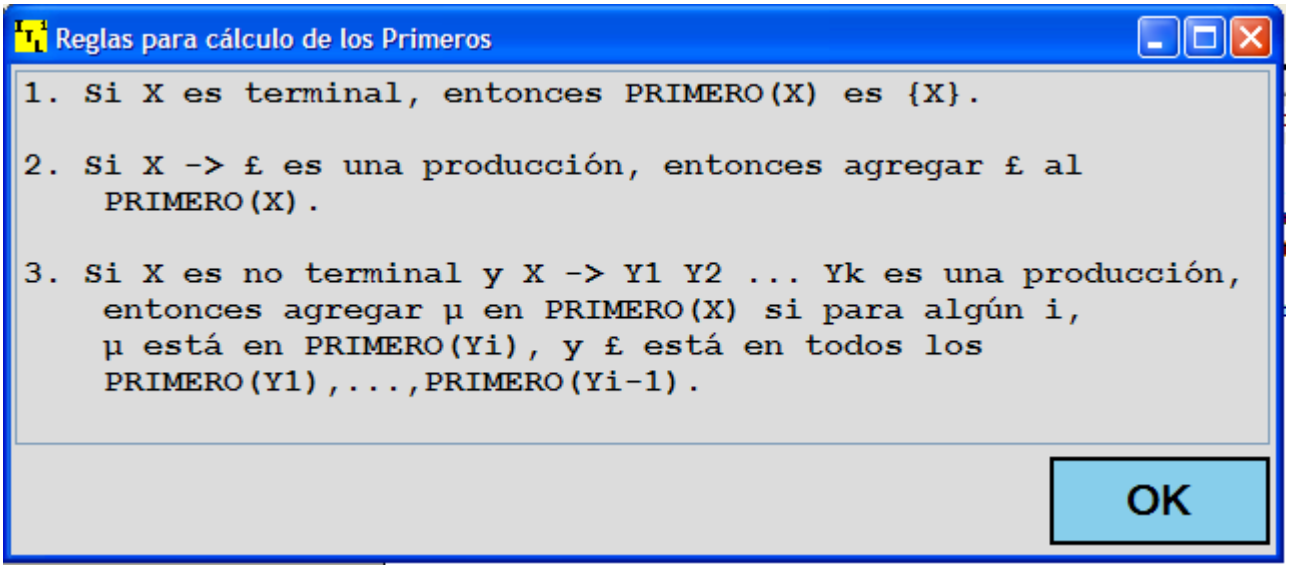


Fig. No. 5.2 Reglas para cálculo de los PRIMEROS.

En el paso 3, el software RD-NRP busca las producciones de la forma A -> £ en la gramática FI para aplicar la regla #2 del cálculo de los PRIMEROS. Siguiendo con el ejemplo de la gramática de declaración de variables en C, tenemos 2 producciones que cumplen con la forma A -> £ :

```
L' -> £   ,   metemos  £  en PRIMERO( L' )

H' -> £   ,   metemos  £  en PRIMERO( H' )
```

RD-NRP visualiza estas acciones para luego mostrar los cambios en los PRIMEROS, figura #5.3.

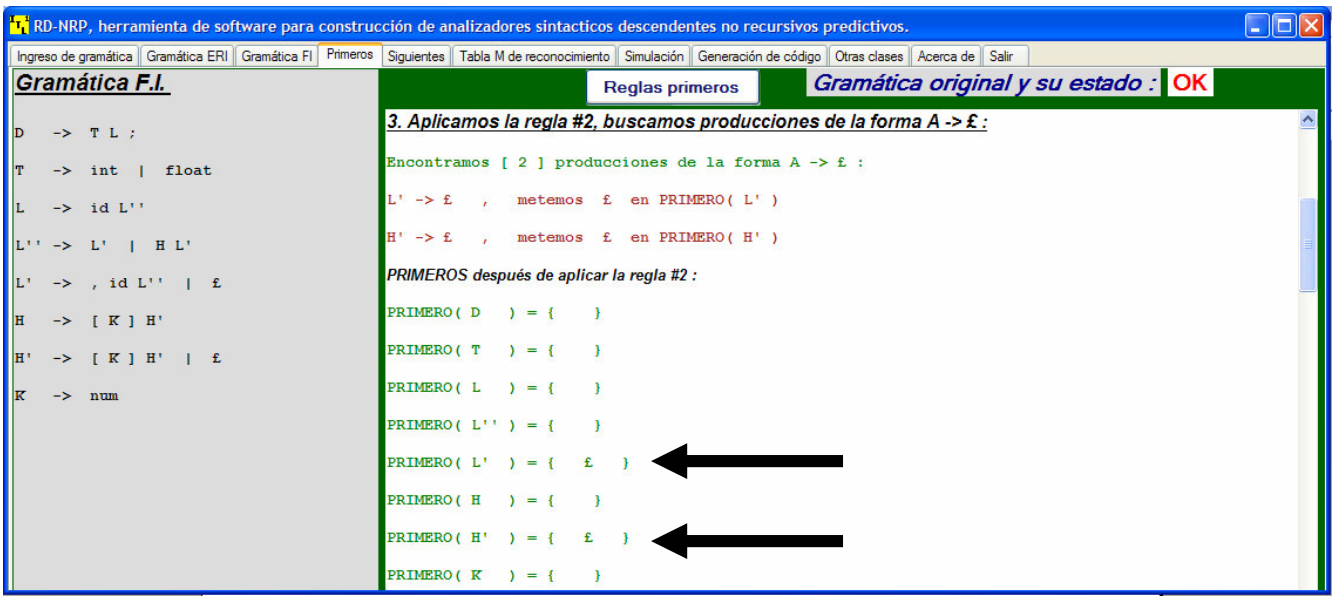


Fig. No. 5.3 Aplicación de la regla #2 para cálculo de los PRIMEROS.

La aplicación de la 3ra. regla para cálculo de los PRIMEROS concierne al 4º. paso que realiza el software RD-NRP. Esta regla debe aplicarse varias veces hasta que ya no haya cambios en los PRIMEROS de cada una de los no terminales de la gramática. La razón de esto, consiste en el orden de aplicación de dicha regla. Generalmente, nosotros empezamos por tratar de aplicar la regla a la producción primera, luego la segunda y así sucesivamente. En ocasiones, el PRIMERO de la Y-iésima que queremos meter al PRIMERO del no terminal en el miembro izquierdo de la producción, no está aún definido o está inconcluso su cálculo. La figura #5.4 muestra precisamente lo que estamos tratando de explicar.

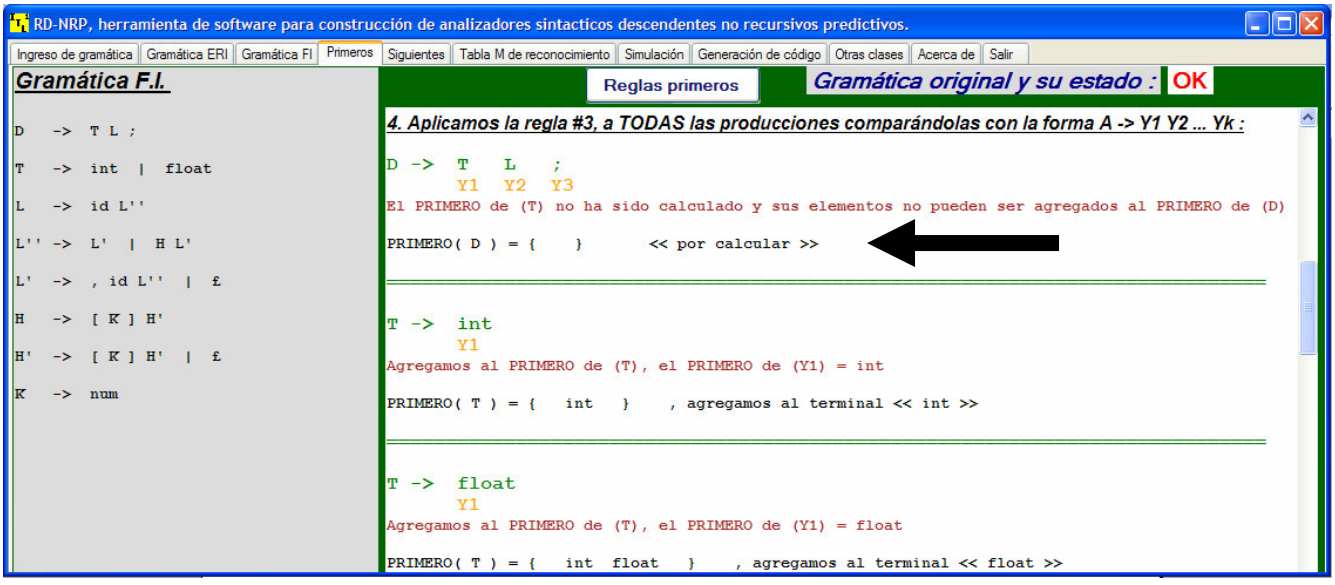


Fig. No. 5.4 Aplicación de la regla #3. PRIMERO(D) por calcular.

Observemos que el PRIMERO de (D) no puede ser calculado ya que el PRIMERO de (T) no se conoce aún. En la misma figura se muestra que el PRIMERO de (D) es calculado antes de calcular el PRIMERO de (T). Es por lo anterior que debemos de aplicar la 3ra. regla de forma repetitiva hasta que todos los PRIMEROS sean calculados adecuadamente. La figura #5.5 presenta el aviso que presenta el RD-NRP al usuario para indicarle que algunos PRIMEROS necesitan de volver a calcularse.

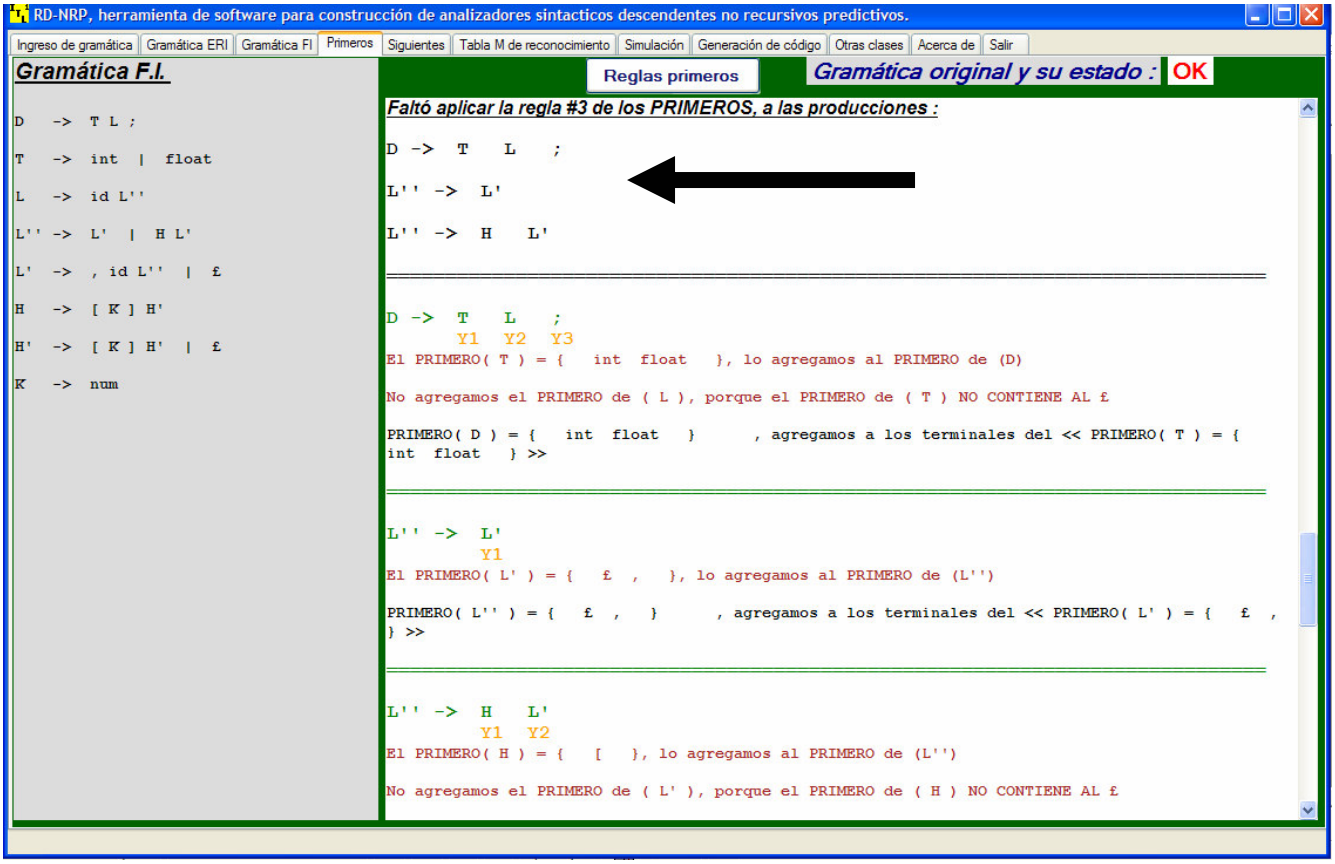


Fig. No. 5.5 Aviso de aplicación de la 3ra. regla, de nuevo.

Cuando RD-NRP trata de aplicar la 3ra. regla a producciones de la forma A -> £, responde con el mensaje que dichas producciones ya han sido analizadas cuando se les aplicó la 2da. regla del cálculo de los PRIMEROS.

```
-------------------------------------------------------

L' ->  £
       Y1
Esta forma de producción pertenece a la regla #2, el PRIMERO(L') permanece sin cambios.

PRIMERO( L' ) = {   £  ,   }
-------------------------------------------------------

H' ->  £
       Y1
Esta forma de producción pertenece a la regla #2, el PRIMERO(H') permanece sin cambios.

PRIMERO( H' ) = {   £  [   }
-------------------------------------------------------
```

Por último, son mostrados los PRIMEROS de todos los no terminales de la gramática transformada. Para nuestro ejemplo los PRIMEROS que visualiza el RD-NRP son los mostrados en la figura #5.6.

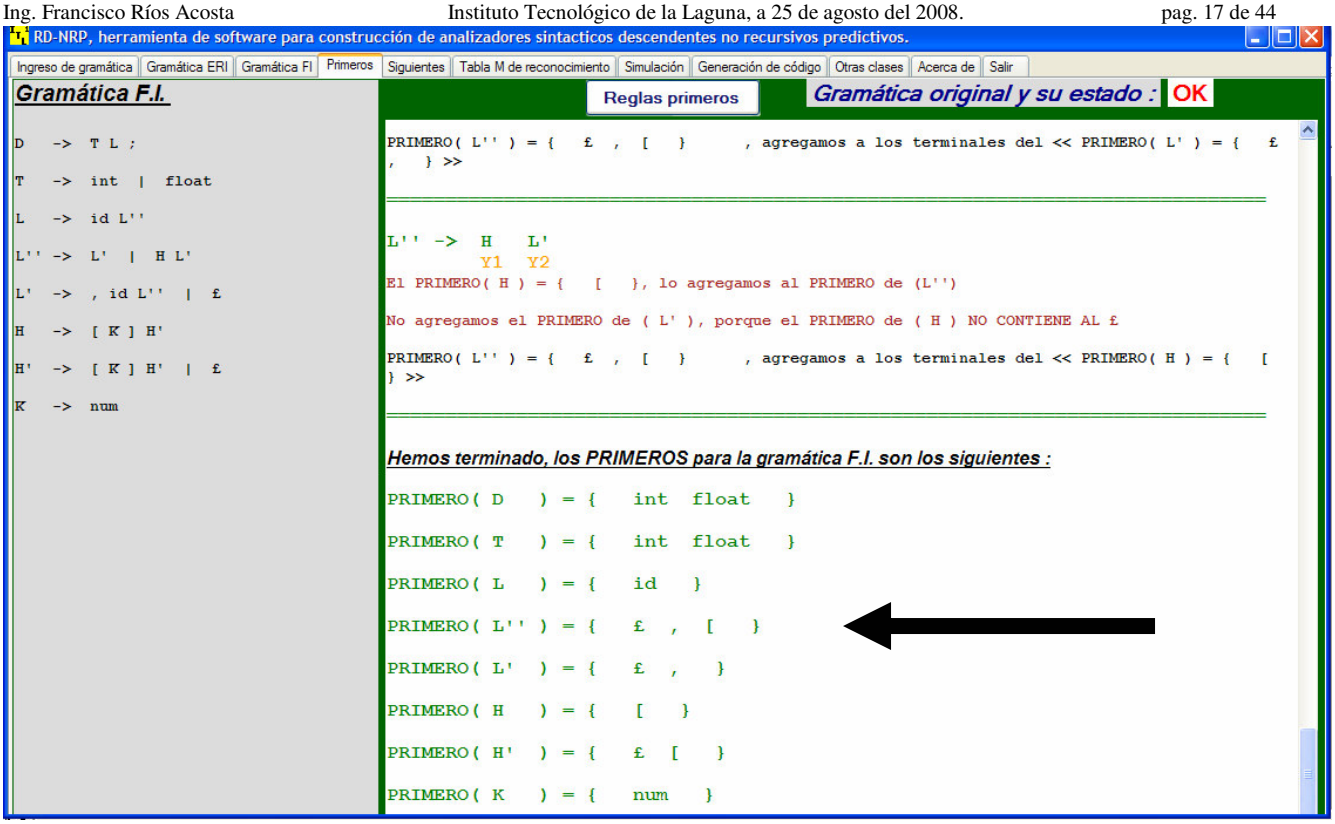


Fig. No. 5.6 PRIMEROS para la gramática de declaración de variables en C.

6 SIGUIENTES.

La interfase es muy parecida a la de los PRIMEROS, sólo vemos que se agrega en la ventana de la izquierda a los PRIMEROS previamente calculados. Inicialmente se visualizan los SIGUIENTES de los no terminales a calcular, notando que éstos están vacios -0 terminales agregados-, según apreciamos en la figura #6.1.

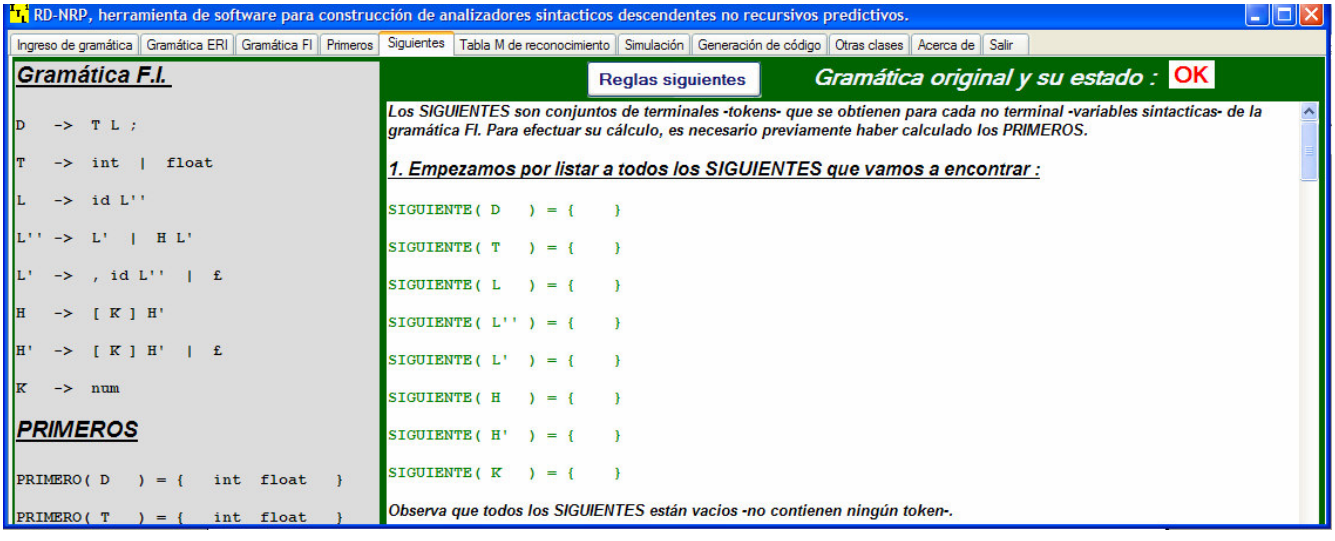


Fig. No. 6.1 SIGUIENTES a calcular, inicialmente vacios.

Las reglas para el cálculo de los SIGUIENTES, pueden ser visualizadas por el software RD-NRP accionando el botón con leyenda *Reglas siguientes*. La figura #6.2 muestra la ventana con la que responde el RD-NRP.

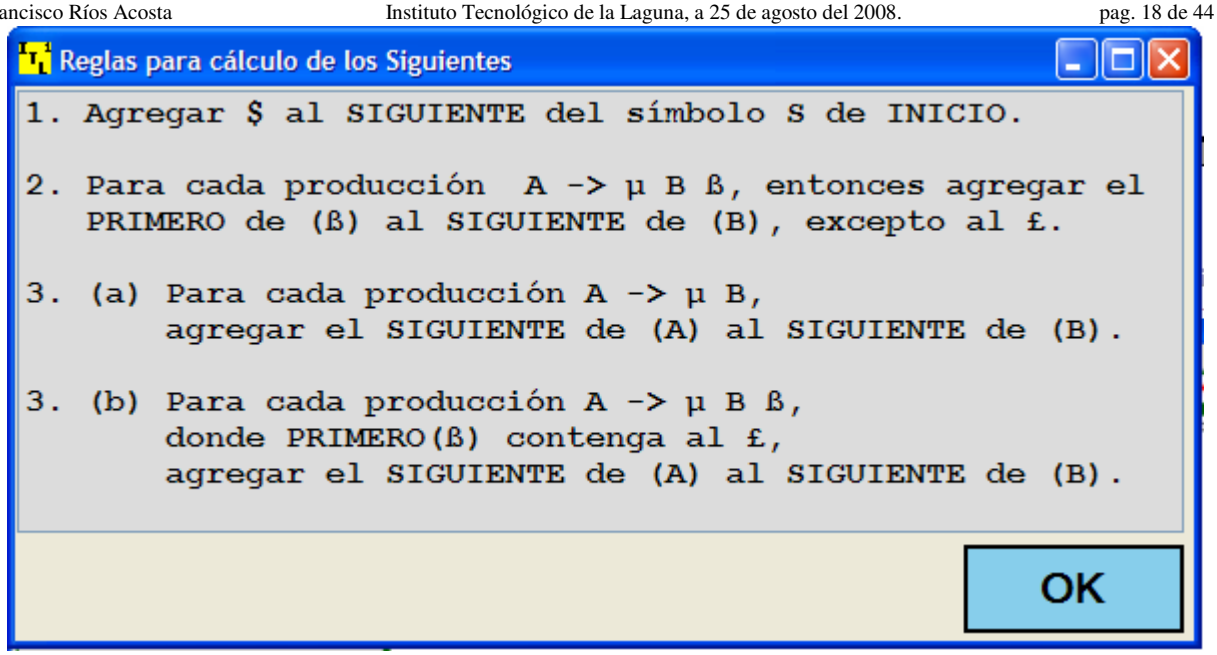


Fig. No. 6.2 Reglas para el cálculo de los SIGUIENTES.

El siguiente paso que ejecuta el RD-NRP es la aplicación de la 1ra. regla para el cálculo de los SIGUIENTES, la cual indica que el símbolo $ debe meterse en el SIGUIENTE del símbolo no terminal de inicio, para este ejemplo la D –marcado con rojo-, figura #6.3.

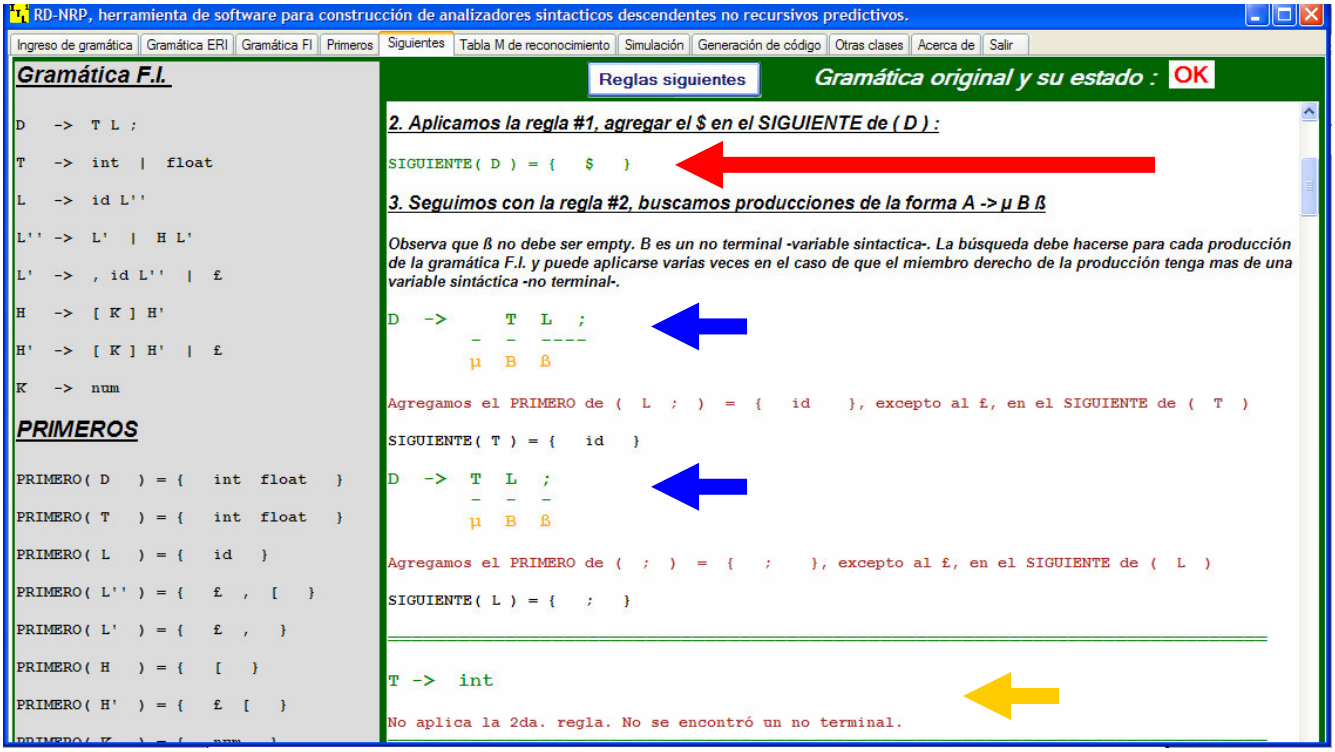


Fig. No. 6.3 Paso 2 : aplicación de la 1ra. regla.
Paso 3 : aplicación de la 2da. regla.

La 2da. regla es aplicada enseguida, buscando las producciones que cumplan con la forma A -> µ B ß. Pueden existir producciones a las cuales se les aplique la 2da. regla varias veces, dependiendo del número de símbolos no terminales que contengan en el miembro derecho. Para nuestro ejemplo, la producción D –> T L ; es un caso de aplicación 1+, marcado con color azul.

Cuando la regla no se aplique, entonces tendremos un letrero que así nos lo indica. La figura #6.3 muestra en color amarillo una producción que no cumple con la regla.

El resultado de la aplicación de la 2da. regla es el siguiente paso que efectúa RD-NRP de manera que el usuario vea los cambios que ha producido esta regla en los SIGUIENTES, figura #6.4.

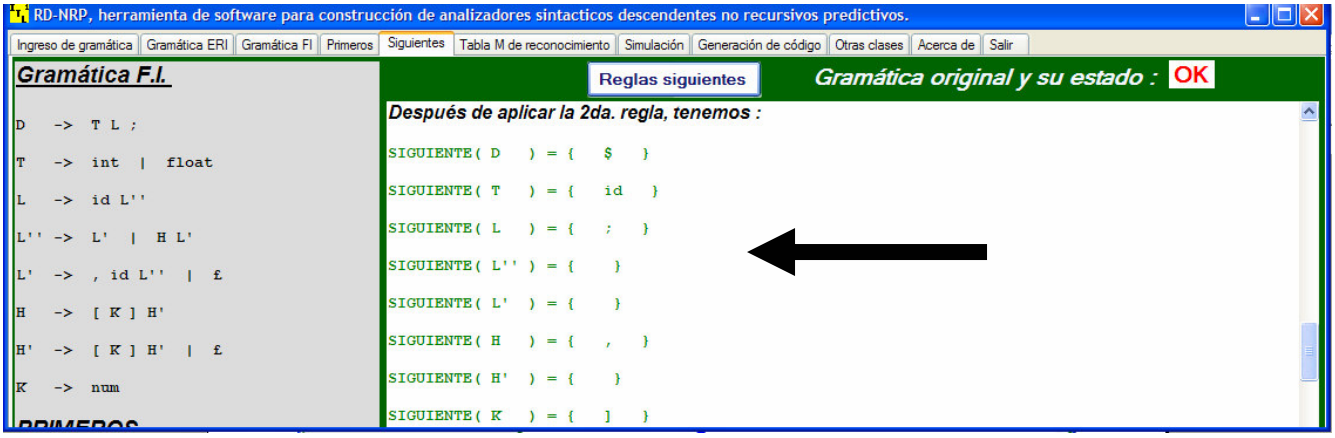


Fig. No. 6.4 SIGUIENTES después de la aplicación de la 2da. regla.

RD-NRP continúa con el paso 4 que consiste en aplicar la 3ra. regla inciso (a), mostrando al usuario la indicación cuando la regla no se aplica :

4. Ahora aplicamos la regla #3 (a), buscamos producciones de la forma A -> μ B

Observa que la producción debe terminar en variable sintactica -no terminal-. B es un no terminal -variable sintactica-.

```
D ->  T   L   ;

No aplica la 3ra. regla (a). La producción no termina en variable sintactica.
```

```
T ->  int

No aplica la 3ra. regla (a). La producción no termina en variable sintactica.
```

```
T ->  float

No aplica la 3ra. regla (a). La producción no termina en variable sintactica.
```

Cuando la regla 3ra. (a) si aplica, veremos lo siguiente :

```
L  ->  id  L''
       --  --
       μ   B

Agregamos el SIGUIENTE de ( L ), en el SIGUIENTE de ( L'' ).  << GUARDAR EN LA BITACORA >>

SIGUIENTE( L'' ) = {   ;   }
```

Observemos el letrero que indica que la aplicación de la regla debe guardarse en una bitácora. La regla 3ra. (a) y (b) deben aplicarse de manera repetida hasta que no tengamos cambios en los SIGUIENTES que se están calculando.

Los SIGUIENTES son mostrados luego que la 3ra. regla (a) es aplicada como lo presenta la figura #6.5.

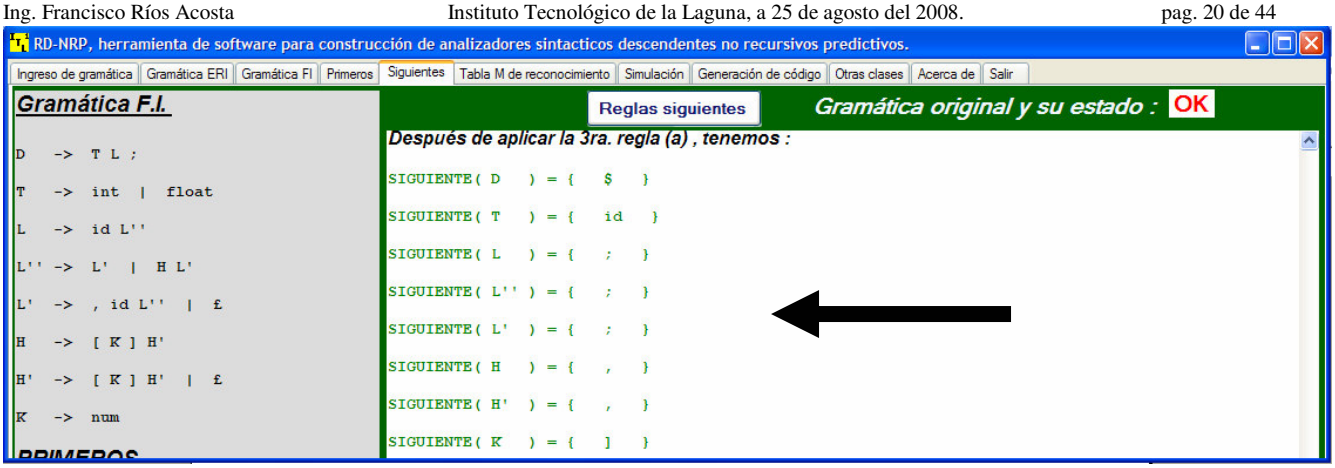


Fig. No. 6.5 SIGUIENTES después de la aplicación de la 3ra. regla (a).

RD-NRP sigue con la aplicación de la regla 3ra. (b). La figura #6.6 muestra casos en los que dicha regla no aplica y como responde con mensajes el software RD-NRP.

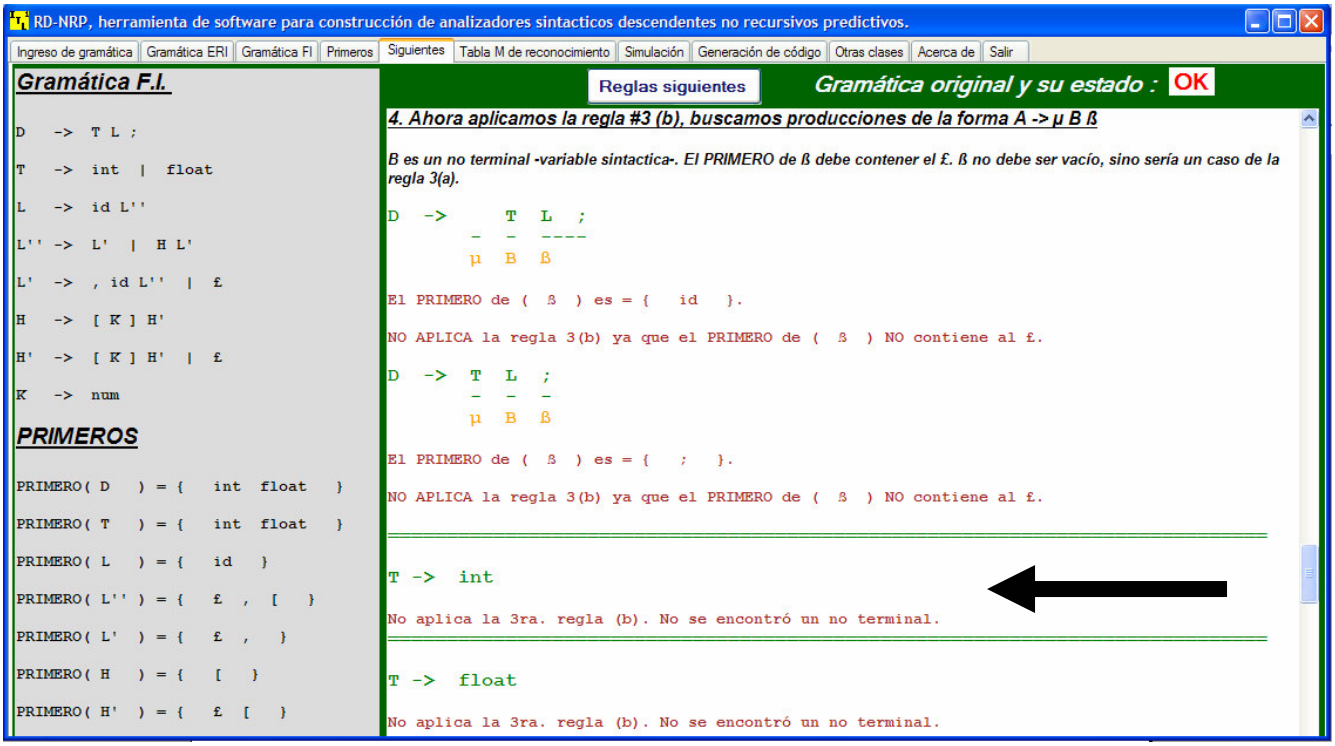


Fig. No. 6.6 Producciones en las que la regla 3 (b) no aplica.

Otro caso de nuestro ejemplo en que la regla 3ra. (b) no aplica, es el que a continuación listamos. Notemos cómo es que RD-NRP responde al caso en el que la única no terminal está al final de la producción, es decir pertenece a la 3ra. regla (a).

```
L ->  id   L''

No aplica la 3ra. regla (b). El no terminal está al final de la producción, por lo que ß es empty.

L'' ->  L'

No aplica la 3ra. regla (b). El no terminal está al final de la producción, por lo que ß es empty.
```

Veamos ahora un caso en el que si aplica la regla 3ra. (b) de manera que veamos como responde el software :

```
L''  ->     H  L'
          _  -  --
          µ  B  ß

El PRIMERO de (  ß  ) es = {   £  ,   }.

Observa que el PRIMERO de (  ß  ) SI contiene al £, la regla 3(b) SI aplica. Por lo tanto :

Agregamos el SIGUIENTE de ( L'' ), en el SIGUIENTE de ( H ).  << GUARDAR EN LA BITACORA >>

SIGUIENTE( L'' ) = {   ;   } en el SIGUIENTE( H ) = {   ,   ;   }
```

En esta producción si aplica la regla debido a que el PRIMERO de ß si contiene al empty. Después que se aplicaron la regla 3ra. (a) y (b), RD-NRP muestra los SIGUIENTES y la bitácora de aplicación de dicha regla e incisos, figura #6.7.

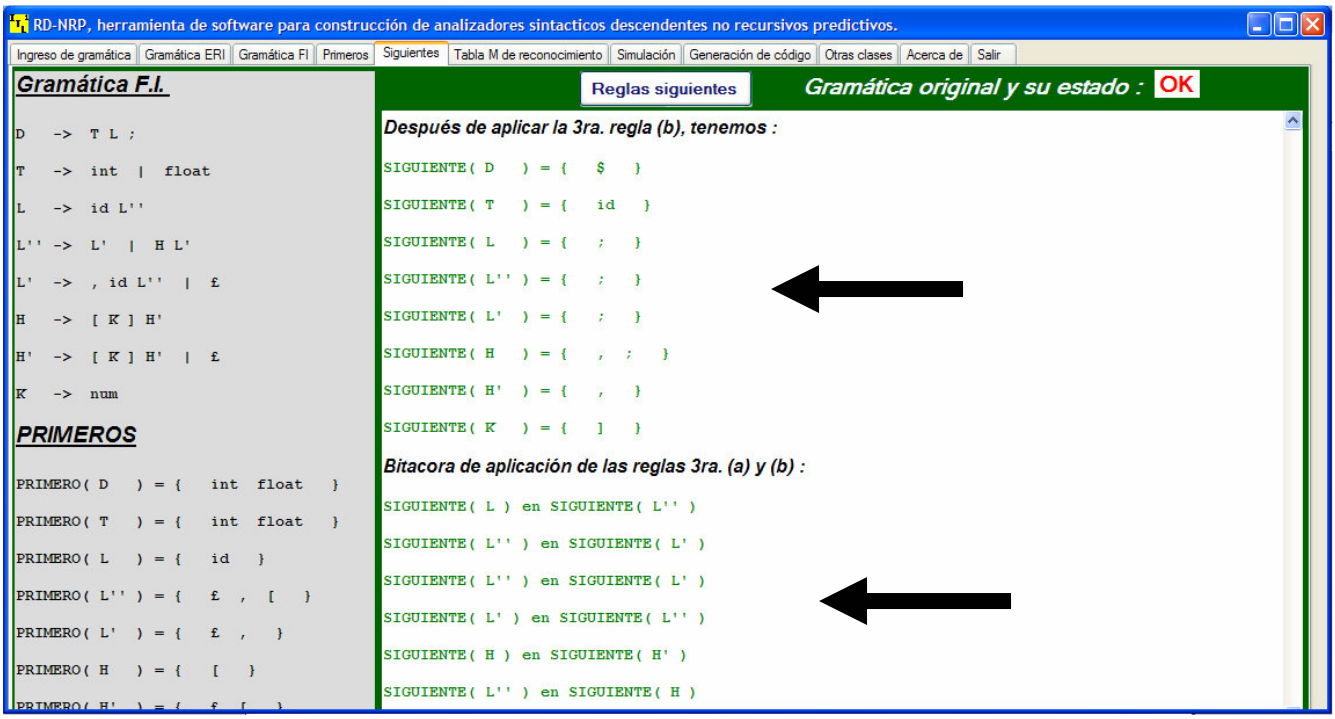


Fig. No. 6.7 SIGUIENTES y bitácora después de aplicar la 3ra. regla (a) y (b).

Una vez que se aplica la 3ra. regla se procede a su aplicación de manera repetitiva ayudándonos de la bitácora, hasta que no existan cambios en los SIGUIENTES. RD-NRP los hace así y lo reporta al usuario con los mensajes que a continuación se listan :

5. Aplicamos los movimientos indicados en la bitacora, hasta que no haya cambios :

```
SIGUIENTE( H ) en SIGUIENTE( H' ),  provocó cambios.

SIGUIENTE( H' ) = {   ,   ;   }
```

Hubo cambios en los SIGUIENTES, volvemos aplicar la bitacora :

Terminamos la obtención de los SIGUIENTES. NO hubo mas cambios.

Los cambios en los SIGUIENTES y su aplicación los reporta el RD-NRP en los mensajes descritos en el párrafo anterior. En este ejemplo, se reporta el cambio en el SIGUIENTE de (H) .

Al final son visualizados los SIGUIENTES para cada no terminal de la gramática transformada, figura #6.8.

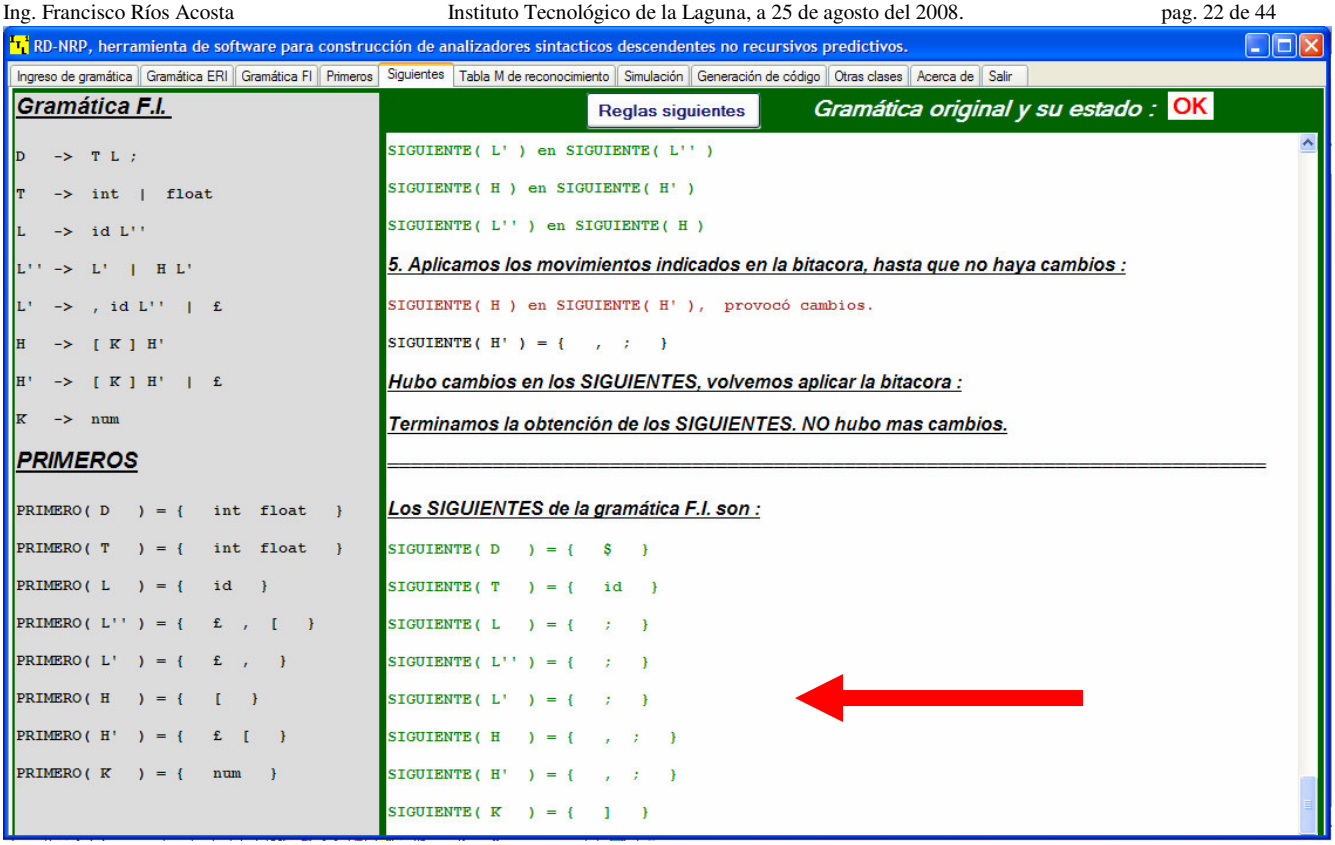


Fig. No. 6.8 SIGUIENTES resultantes reportados por RD-NRP.

7 Tabla M de reconocimiento.

La interfase de usuario presentada por RD-NRP para la tabla M de reconocimiento, consta de :

- Ventana que muestra la gramática F.I., los PRIMEROS y los SIGUIENTES.
- Un botón que permite visualizar el algoritmo de construcción de la tabla M.
- Un botón que visualiza en un componente DataGridView (tabla de datos) a la tabla M construída.
- Ventana que visualiza paso a paso cómo fue construida la tabla M de reconocimiento.

La tabla M de reconocimiento representa la etapa final de preparación para utilizar el algoritmo que analiza sintacticamente a una sentencia. Su construcción requiere de haber previamente transformado la gramática, además de la obtención de los PRIMEROS y de los SIGUIENTES.

Inicialmente en la ventana de construcción paso a paso de la tabla M de reconocimiento, se muestra un resúmen de conceptos importantes acerca de ella :

1. ***La tabla M de reconocimiento es construida a partir de la gramática F.I., los PRIMEROS y los SIGUIENTES. Si la gramática es ambigua, entonces pudieran existir mas de una producción para una cierta entrada de la tabla M.***
2. ***En este caso, el reconocedor descendente no recursivo predictivo, no podrá ser codificado. Debemos corregir la gramática original antes de su transformación.***

3. ***La tabla M tiene columnas y renglones. Las columnas son los símbolos terminales -tokens- de la gramática, además del $. Los renglones son los no terminales -variables sintácticas-. Las celdas de la tabla M pueden contener solamente una producción.***

4. ***Inicialmente la tabla M está vacia, es decir ninguna de sus celdas tiene una producción.***

En cualquier momento podemos acceder a las reglas de construcción de la tabla M, accionando el botón con leyenda : *Reglas tabla M*, figura #7.1.

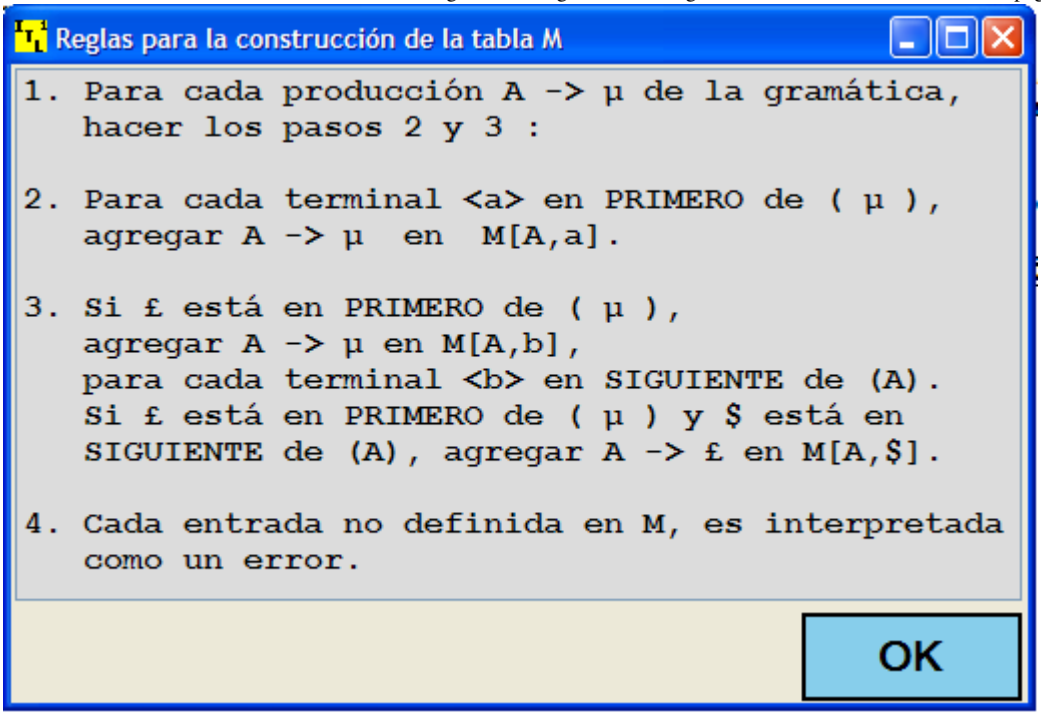


Fig. No. 7.1 Reglas para la construcción de la tabla M.

El sofware visualiza paso a paso la aplicación de los 2 procesos para cada producción de la gramática F.I. que llevan a la construcción de la tabla M de reconocimiento. Cada producción es comparada con la forma A -> μ para luego aplicar los 2 procesos definidos por las reglas de construcción de la tabla M, figura #7.2.

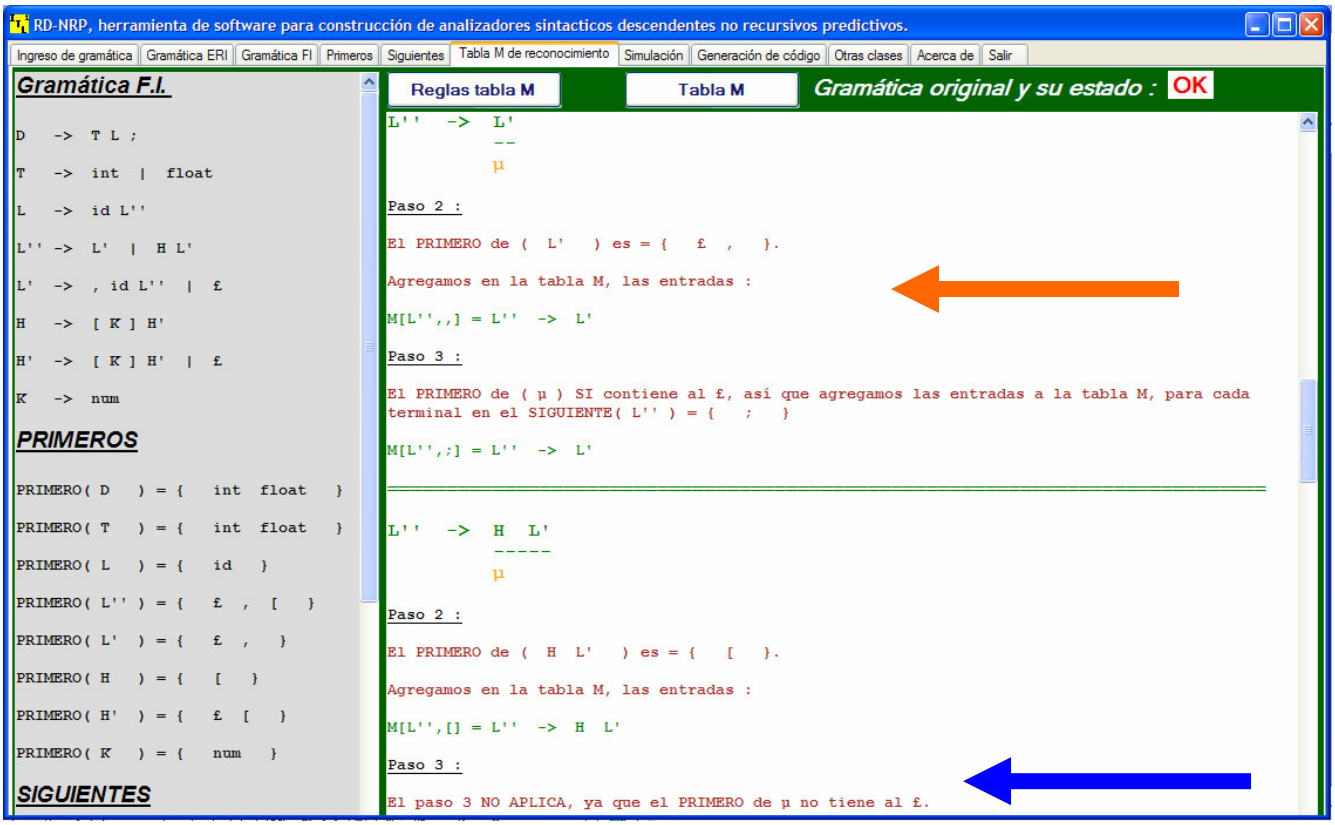
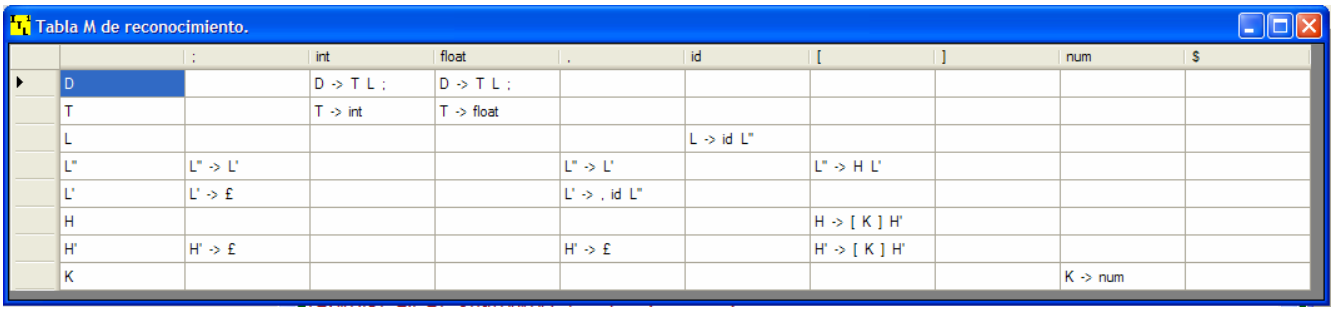


Fig. No. 7.2 L'' -> L' permite los 2 procesos. L'' -> H L' permite solo aplicar un proceso.

Otro ejemplo de aplicación de los 2 procesos a una producción es :

```
H'  ->  £
        –
        µ

Paso 2 :

El PRIMERO de ( £ ) es = {   £   }.

NO agregamos entrada a la tabla M, ya que el £ no es columna de la tabla. El paso 2 NO aplica.

Paso 3 :

El PRIMERO de ( µ ) SI contiene al £, así que agregamos las entradas a la tabla M, para cada terminal
en el SIGUIENTE( H' ) = {   ,   ;   }

M[H',,] = H'  ->  £

M[H',;] = H'  ->  £
```

Podemos acceder a la tabla M construida a partir de las reglas definidas para ella, usando el botón con leyenda *Tabla M*. La figura #7.3 muestra la tabla M de reconocimiento construida usando el RD-NRP.



	;	int	float	,	id	[	]	num	$
D		D -> T L ;	D -> T L ;						
T		T -> int	T -> float						
L					L -> id L"				
L"	L" -> L'			L" -> L'		L" -> H L'			
L'	L' -> £			L' -> , id L"					
H						H -> [K] H'			
H'	H' -> £			H' -> £		H' -> [K] H'			
K								K -> num	

Fig. No. 7.3 Tabla M para nuestro ejemplo.

8 Simulación.

En esta característica que presenta nuestro software RD-NRP podemos ingresar una sentencia, que será analizada por el reconocedor descendente no recursivo predictivo retroalimentando al usuario un mensaje de error de sintaxis o bien, de éxito en el reconocimiento.

El usuario debe teclear en la rejilla de ingreso de la sentencia con leyenda TECLEA EL CONTENIDO DE W$, el conjunto de tokens –terminales- en que consiste la sentencia. Depende del diseño, en ocasiones se debe teclear al token y en otras, debemos teclear el lexema.

Por ejemplo, para una sentencia que escribiría un programador en una declaración de 2 variables enteras –simple y arreglo- :

```
int x, y[10];
```

La cadena que debemos ingresar de acuerdo a nuestra gramática será :

```
int id,id[num];
```

El analizador léxico es el encargado de tomar la decisión del envío del token o del lexema al analizador sintactico. La figura #8.1 muestra la simulación para la sentencia mencionada anteiormente.

Estos últimos debemos teclearlos en la rejilla de CONTENIDOS DEL W$. Observemos en la figura que el reconocimiento ha sido exitoso y que además es mostrada la derivación a la izquierda de la sentencia.

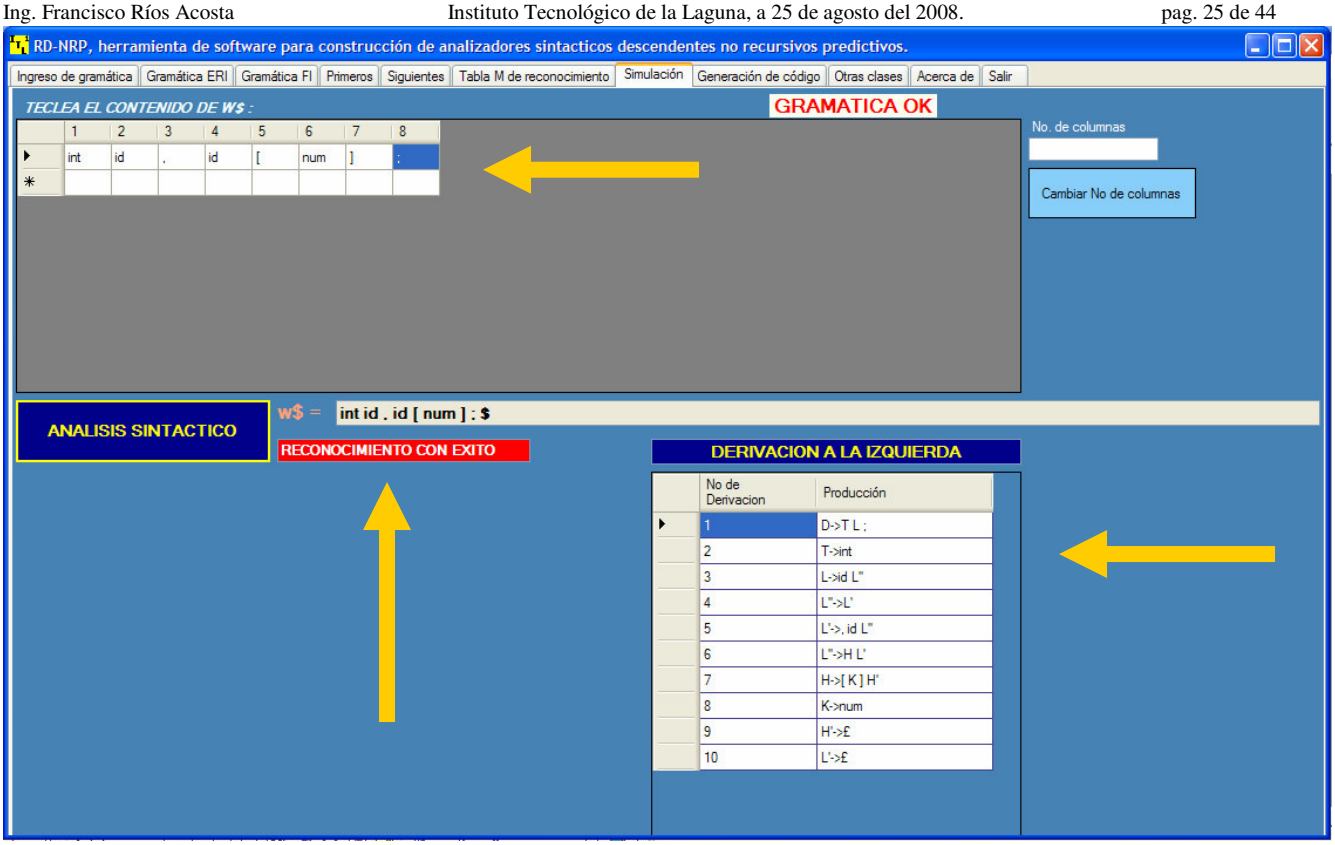


Fig. No. 8.1 Simulación para la sentencia int id,id[num];.

Quitemos un corchete a la sentencia anterior para observar la respuesta del RD-NRP al error, figura #8.2.

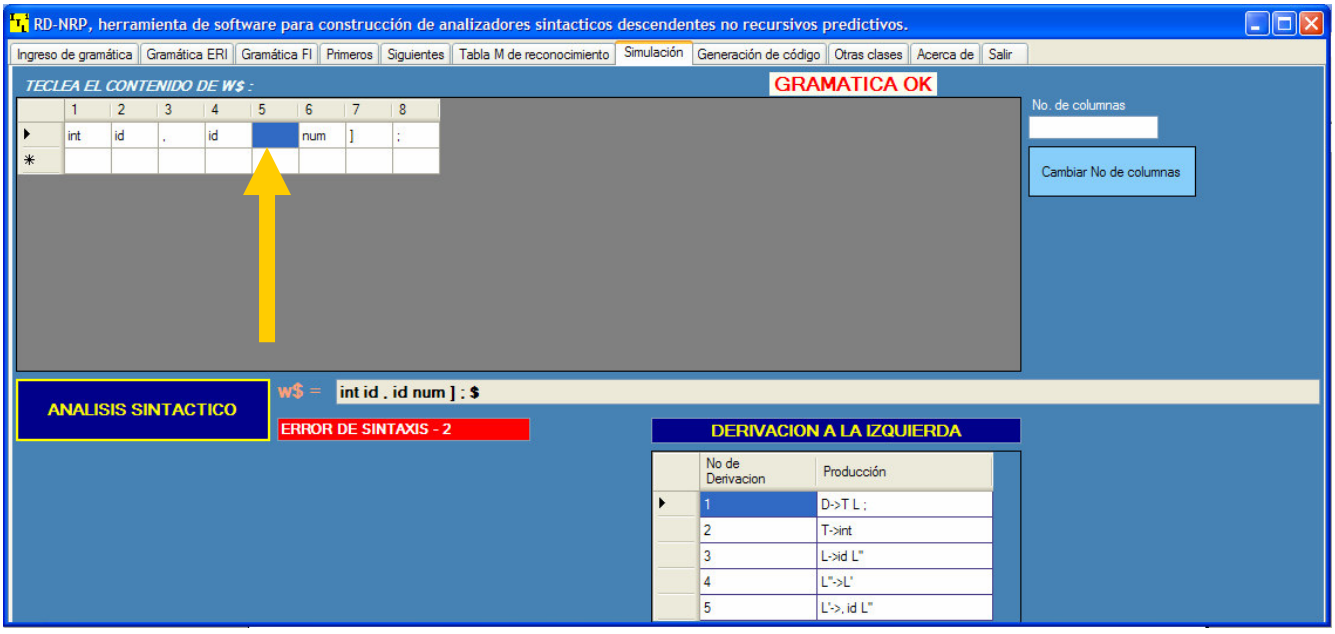


Fig. No. 8.2 Error en la sentencia de entrada.

9 Generación de código.

Esta capacidad del RD-NRP consiste en la generación de código para la clase *SintDescNRP* propuesta por Ríos A. Francisco. La clase generada permite definir objetos analizadores sintacticos que reconocen una sentencia previamente analizada por un analizador léxico. La interfase que se presenta al usuario es mostrada en la figura #9.1.

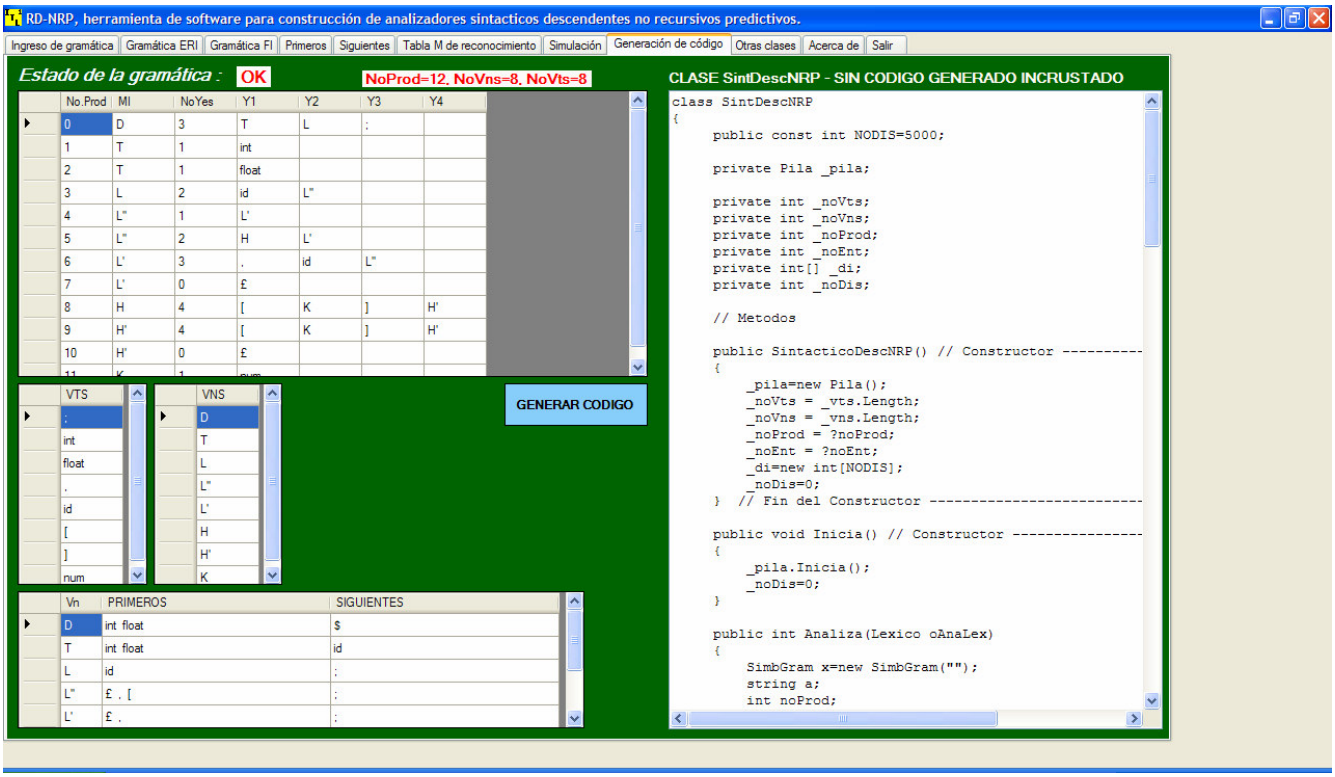


Fig. No. 9.1 Interfase para generación de código de la clase *SintDescNRP*.

La interfase consiste de :

- Rejilla de producciones desagrupadas de la gramática FI.
- Letrero que indica el número de producciones, de terminales y de no terminales de la gramática FI.
- Rejillas para visualización de los Vts y los Vns.
- Rejilla indicadora de los PRIMEROS y SIGUIENTES de cada no terminal.
- Venana que contiene la definición de la clase *SintDescNRP*.
- Botón para generar el código –inserción de código en la clase *SintDescNRP*-.

El botón de generación de código inserta la definición de los arreglos :

```
private string[] _vts;
private string[] _vns;
private int[,] _prod;
private int[,] _m;
```

Realmente los agrega efectúando la inicialización de ellos al momento de su declaración. También la generación de código agrega la inicialización del atributo **_noProd** y del atributo **_noEnt**, dentro del método constructor de la clase.

```
_noProd = 12;
_noEnt = 15;
```

La figura #9.2 muestra la generación de código para dichos arreglos dentro de la clase *SintDescNRP*. Notemos la inicialización de los arreglos al momento de su declaración.

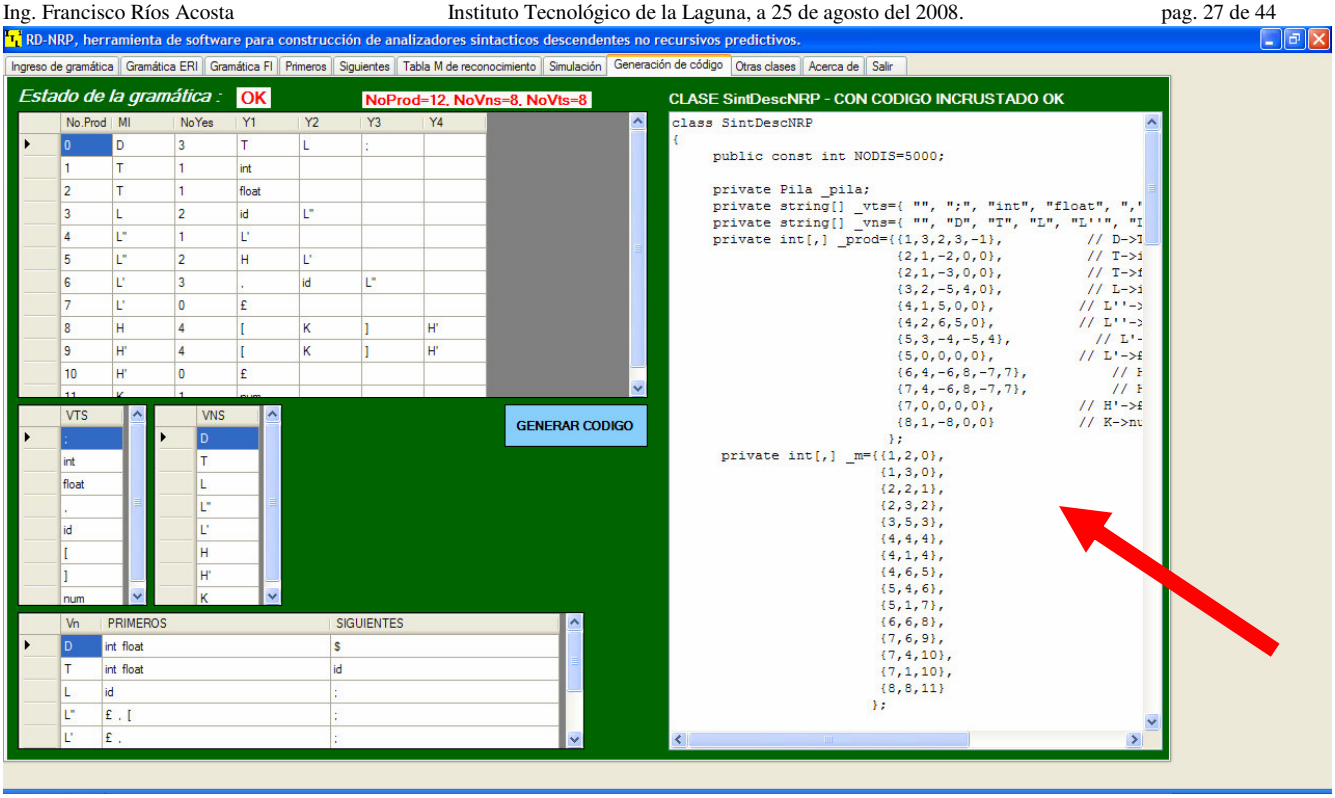


Fig. No. 9.2 Generación de los arreglos _vts, _vns, _prod y _m dentro de la clase SintDescNRP.

La incialización de los atributos _noProd y _noEnt dentro del constructor de la clase *SintDescNRP*, lo vemos señalado en la figura #9.3.

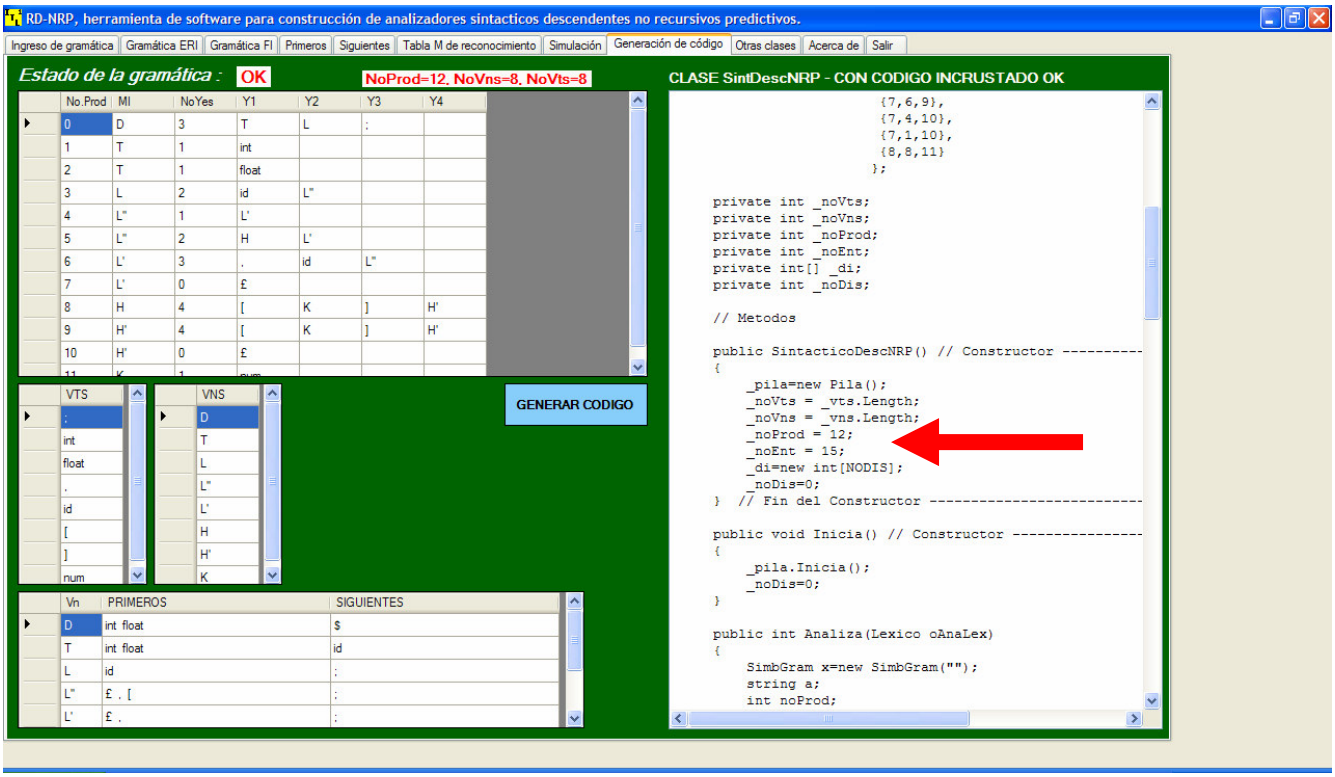


Fig. No. 9.3 Generación de los atributos _noProd y _noEnt.

10 Clase SintDescNRP.

A continuación mostramos la tabla que contiene la descripción de los atributos de la clase *SintDescNRP*.

DEFINICIÓN DEL ATRIBUTO	DESCRIPCIÓN
`public const int NODIS=5000;`	Es el número máximo de producciones usadas en la derivación a la izquierda de una sentencia. Representa el número de elementos al cual se dimensiona el arreglo `_di`. No lo modifica la generación de código, si quieres hacerlo basta con cambiar el valor de 5000 al que nosotros deseamos.
`private Pila _pila;`	Objeto perteneciente a la clase Pila usado en el algoritmo del reconocedor descendente. Almacena objetos de la clase SimbGram –símbolos terminales y no terminales-.
`private string[] _vts;`	Arreglo de cadenas de una dimensión. Se utiliza para almacenar a los símbolos terminales –tokens-. También es almacenado en el índice _noVts el símbolo $. El primer elemento siempre es la cadena nula “”. Este arreglo es inicializado con sus elementos al momento de la generación de código.
`private string[] _vns;`	Arreglo de cadenas de una dimensión. Almacena a los símbolos no terminales –variables sintacticas-. El primer elemento siempre es la cadena nula. El símbolo de inicio de la gramática corresponde al índice 1. Este arreglo es inicializado con sus elementos al momento de la generación de código.
`private int[,] _prod;`	Arreglo de enteros de 2 dimensiones. Cada renglón representa a una producción de la gramática. Las columnas indican : la 0 el índice del miembro izquierdo –siempre es un no terminal-, la columna 1 indica el número de yes de la producción, la columna 2 indica la Y1, la columna 3 indica la Y2, y así sucesivamente. Este arreglo es inicializado con sus elementos al momento de la generación de código. Las Y’s que representan a un no terminal son positivas, las Y’s que indican un terminal son negativas, y las Y’s que no se usan se establecen al valor 0.
`private int[,] _m;`	Arreglo de enteros de 2 dimensiones. Cada renglón indica una entrada en la tabla M de reconocimiento. Las columnas siempre son 3. La primer columna indica el índice del no terminal de la entrada, la columna 2 indica el índice del terminal de la entrada, y la columna 3 tiene el número de la producción registrada en la entrada. La entrada se refiere a una celda de la tabla M. Este arreglo es inicializado con sus elementos al momento de la generación de código.
`private int _noVts;`	Indica el número de símbolos terminales en la gramática FI + 1. Recordemos que el elemento con índice 0 no se utiliza.
`private int _noVns;`	Indica el número de símbolos no terminales en la gramática FI + 1. Recordemos que el elemento con índice 0 no se utiliza.
`private int _noProd;`	Se refiere al número de producciones registradas en la gramática FI. El índice de la producción inicial es el 0. Se inicializa al momento de generar el código.
`private int _noEnt;`	Contiene el número de entradas registradas para la tabla M de reconocimiento. Es inicializado cuando se efectúa la generación del código.
`private int[] _di;`	Es un arreglo de una dimensión, que contiene los números de producción utilizadas en la derivación a la izquierda que hace el reconocedor, durante el reconocimiento de una sentencia.
`private int _noDis;`	Contiene el número de producciones registradas en el arreglo `_di`.

Los métodos definidos para la clase SintDescNRP se detallan a continuación.

Prototipo del método	Descripción
`public SintacticoDescNRP();`	Constructor por defecto de la clase. Se dimensionan al atributo _di. Se crea el objeto _pila.
`public void Inicia();`	Método que sirve para inicializar la pila _pila, y el atributo _noDis.
`public int Analiza(Lexico oAnaLex);`	Contiene el algoritmo para un reconocedor descendente predictivo. Recibe a un objeto de la clase Lexico –propuesta por R.A.F.-. La clase Lexico es la generada por el software SP-PS1.
`public bool EsTerminal(string x);`	Retorna un true si el parámetro x es un terminal o $, de otra manera retorna un false.
`public int BusqProd(string x, string a);`	Método utilizado dentro del método Analiza(). Su tarea es buscar en la tabla M la entrada definida para el no terminal x y para el terminal a. Retorna el número de la producción si la encuentra, de lo contrario retorna un -1.
`public void MeterYes(int noProd);`	Su función es meter en el objeto _pila las Y's contenidas en el miembro derecho de la producción de número noProd. noProd es pasado como parámetro. MeterYes() es llamado dentro del método Analiza().
`public int IndiceVn(string vn);`	Retorna el índice del no terminal vn.
`public int IndiceVt(string vt);`	Retorna el índice del terminal vt.

11 Otras clases.

Esta interfase le proporciona al usuario las clases *Pila* y *SimbGram*. Estas clases son utilizadas dentro de la clase *SintDescNRP* y *Pila* respectivamente. La figura #11.1 muestra la interfase contenida en la pestaña con leyenda *Otras clases*.

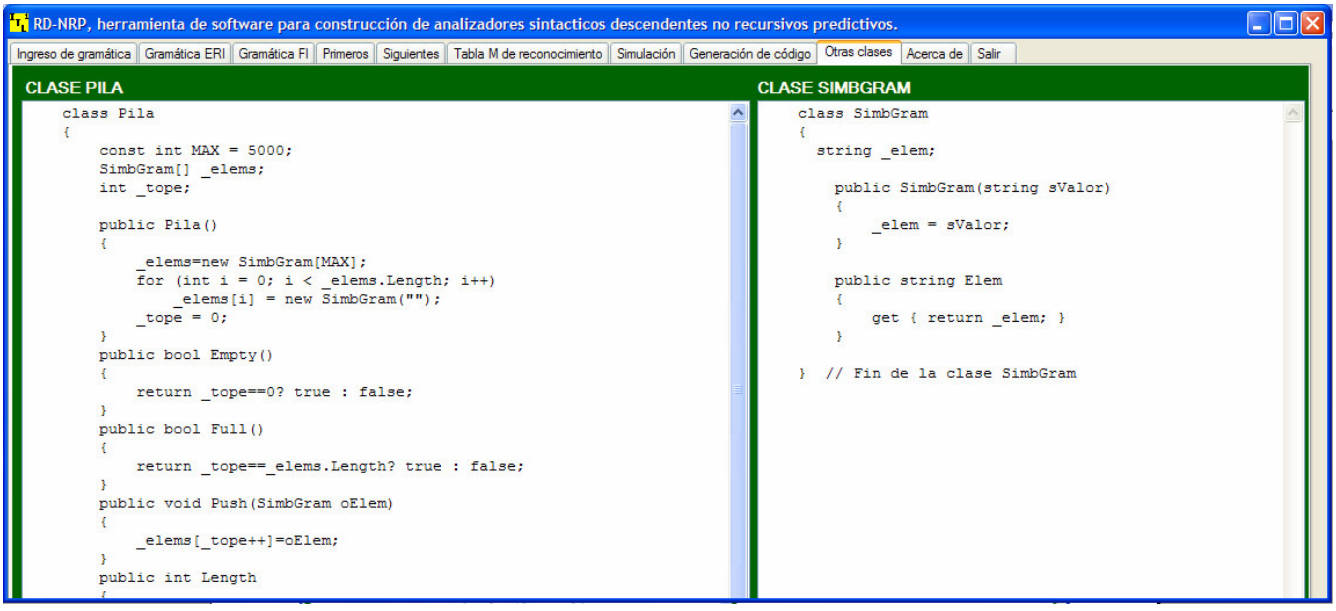


Fig. No. 11.1 Otras clases : class Pila, class SimbGram.

12 Acerca de.

Contiene información del autor, fecha de realización, versión y colaboradores, figura #12.1.

Fig. No. 12.1 Interfase *Acerca de*.

13 Construcción de la aplicación Windows C# para un reconocedor descendente no recursivo predictivo.

En esta sección veremos cómo aprovechar el código producido por los programas SP-PS1 y RD-NRP, para construir una aplicación Windows C# que reconozca un grupo de sentencias cuya sintaxis es definida por una gramática de contexto libre. Las sentencias que vamos a reconocer son las de una declaración de variables en C.

Las etapas que seguiremos para escribir nuestra aplicación Windows C# son :

- Construcción de la interfase gráfica de la aplicación.
- Gramática de contexto libre no ambigua, su transformación y sus componentes.
- Construcción de los AFD's para los terminales en la gramática.
- Analizador léxico para identificar los terminales de la gramática. Código generado por SP-PS1.
- Inclusión del código generado por RD-NRP.
- Prueba de la aplicación.

13.1 Interfase gráfica de la aplicación.

Iniciaremos con una aplicación típica. que permite la entrada de un texto para luego analizarlo léxicamente finalizando con un análisis sintactico usando un reconocedor descendente no recursivo predictivo.
La interfase gráfica contiene los siguientes 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 #13.1.

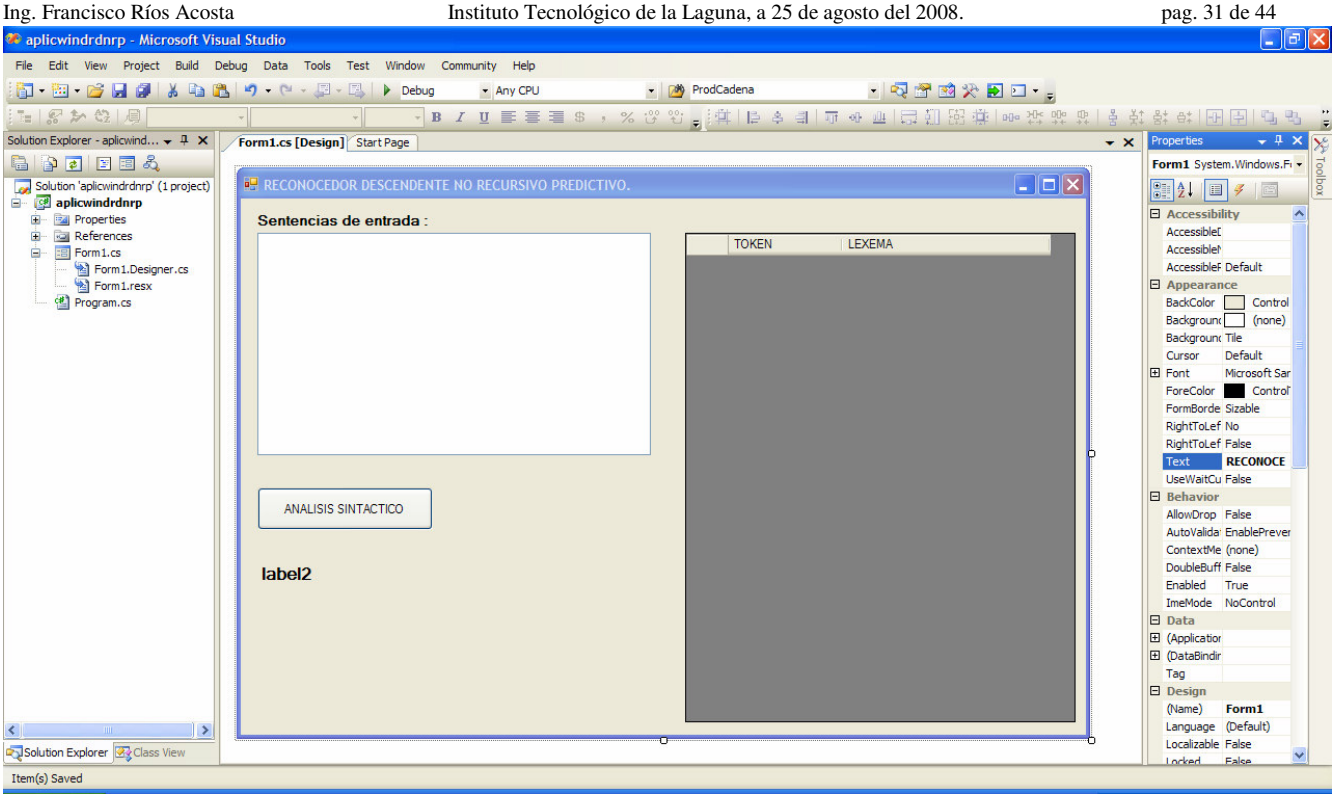


Fig. No. 13.1 Interfase gráfica de la aplicación.

Los componentes tienen las tareas siguientes :

`textBox1.`– Sirve como recipiente de las sentencias que el usuario teclee.
`dataGridView1.`– Su utilidad es visualizar las parejas token-lexema que el analizador léxico reconozca.
`label2.`– Aquí es donde se visualiza el resultado del análisis sintactico ERROR o ÉXITO.
`button1.`– Permite al usuario iniciar el análisis léxico y sintactico.

13.2 Gramática de contexto libre.

Las sentencias que vamos a reconocer son las correspondientes a la declaración de variables en C. La gramática para estas sentencias es la que hemos visto en las secciones anteriores. Desde luego que la gramática que nos interesa es la que se ha transformado aplicando la E.R.I. y la F.I., que se muestra enseguida de forma agrupada :

```
D   ->  T L ;

T   ->  int  |  float

L   ->  id L''

L'' ->  L'  |  H L'

L'  ->  , id L''  |  £

H   ->  [ K ] H'

H'  ->  [ K ] H'  |  £

K   ->  num
```

La gramática de contexto libre presentada tiene 10 producciones, 8 símbolos terminales y 5 no terminales. Para empezar a construir el analizador léxico debemos enfocarnos a los terminales :

Vts = { **; int float id , [ ] num** }

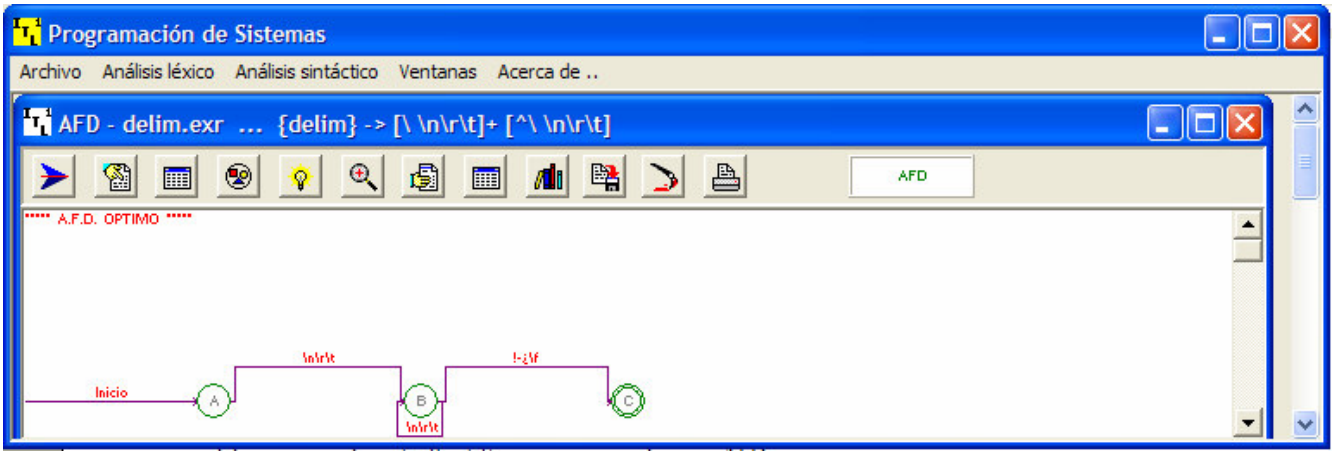
13.3 AFD's a construir.

De acuerdo al conjunto Vts de la gramática FI visto en la sección previa, observamos que debemos construir los AFD's :

- delim
- id
- num
- otros

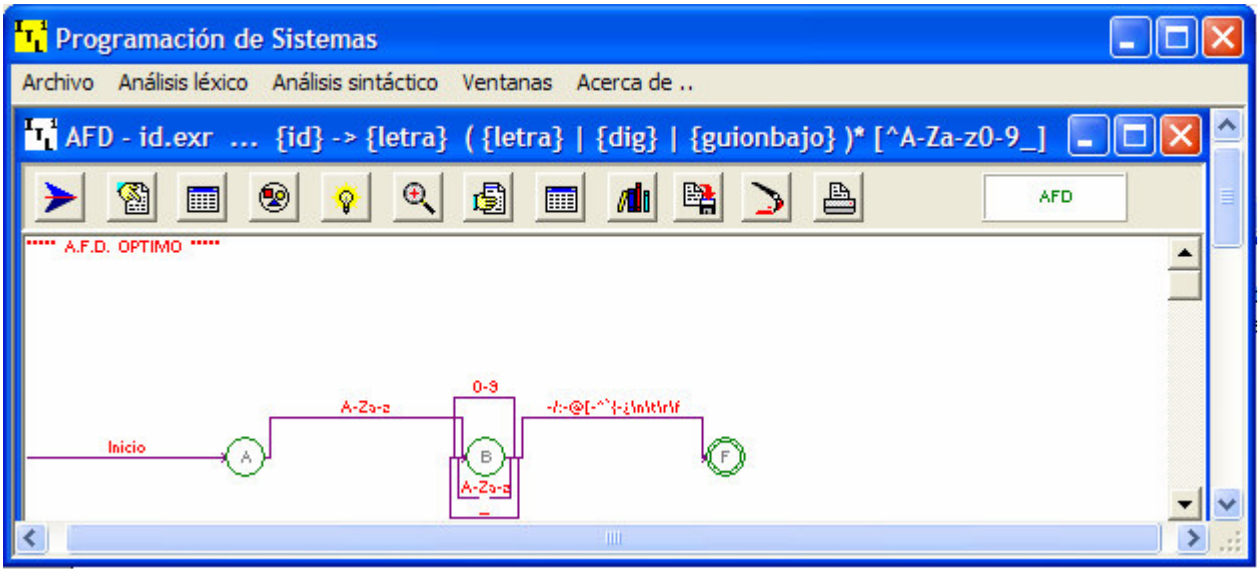
AFD delim.- Sirve para reconocer a los caracteres delimitadores tales como el blanco, nueva línea, retorno de carro, tab. Su expresión regular y AFD óptimo construído por SP-PS1 son :

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



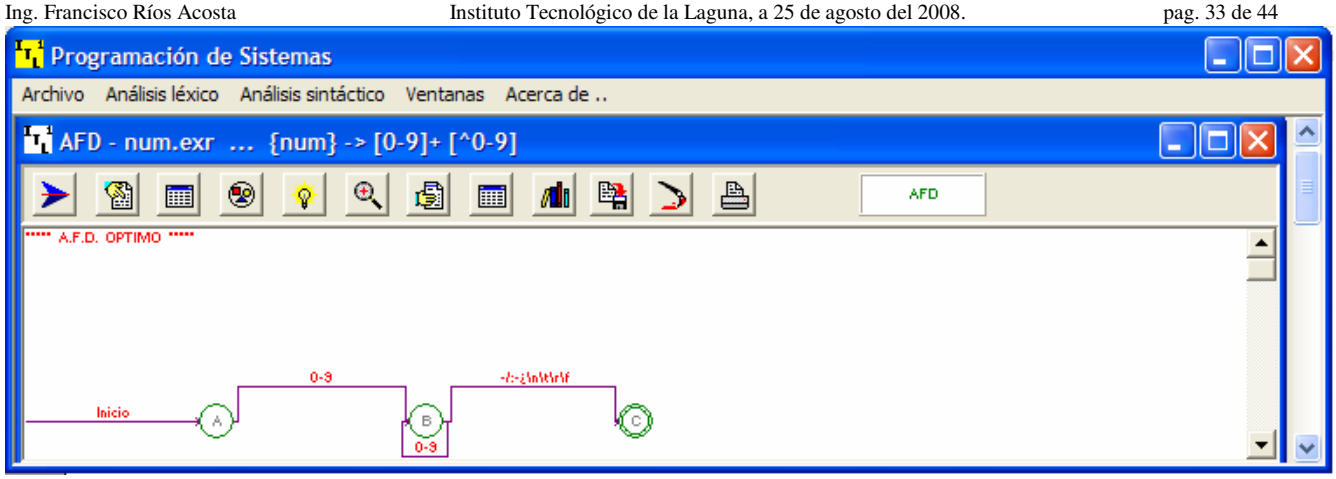
AFD id.- Reconoce al lenguaje de todos los identificadores en C. La regla que usaremos está restingida a : cadenas que inicien con letra seguida de 0 o mas letras, dígitos o subrayado. También usaremos este autómata para reconocer a las palabras reservadas *int* y *float*. Su expresión regular y su AFD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

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



AFD num.- Vamos a limitar a los lexemas del token num a sólo enteros. La expresión regular y su AFD óptimo son :

```
{num} -> [0-9]+ [^0-9]
```



AFD otros.- Este autómata reconoce a los caracteres coma, punto y coma, y los corchetes. Su expresión regular y AFD son :

13.4 Analizador léxico que reconoce a los tokens delim, id, num, otros.

Usando el SP-PS1 ensamblamos los AFD's para los terminales de la gramática en el siguiente orden :

1. delim
2. id
3. num
4. otros

Luego configuramos el RETRAER para los AFD's *delim, id, num*. El AFD *otros* no lo requiere, figura #13.2.

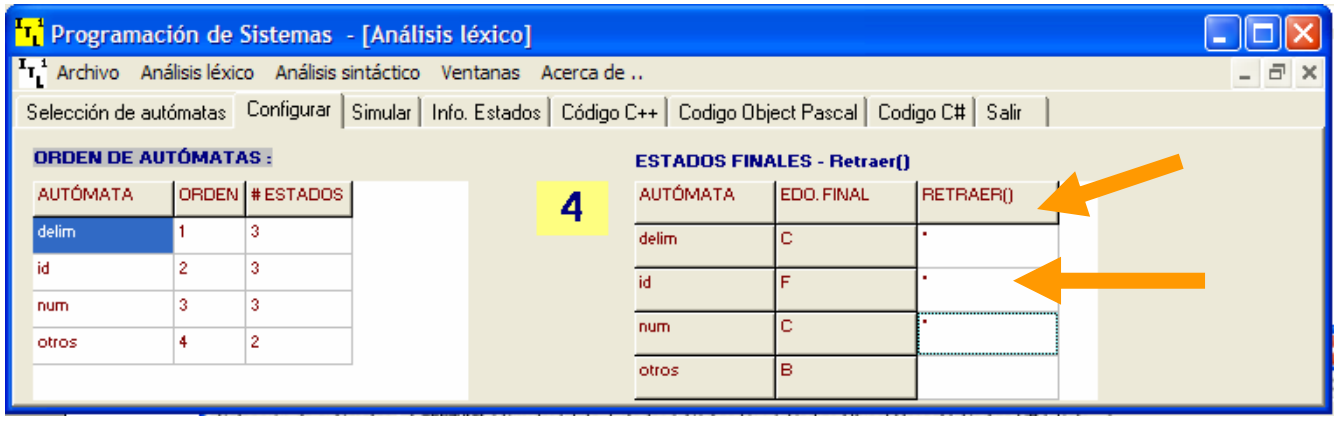


Fig. No. 13.2 Configuración del retraer para los AFD's ensamblados.

El paso siguiente es generar el código C# para las clases *Lexico* y *Automata* :

```
class Lexico
 {
   const int TOKREC = 4;
   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  num--------------
          case 2 : _tokens[_noTokens] = "num";
                   break;
           //--------------  Automata  otros--------------
          case 3 : _tokens[_noTokens] = "otros";
                   break;
         }
         _lexemas[_noTokens++] = _lexema;
       }
       else
         _i++;
       _iniToken = _i;
     }

   }

 }  // fin de la clase Lexico
```

```
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  otros--------------
      case 3 : _edoAct = 9;
                   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  num--------------
      case 6 : c=SigCar(ref i);
                    if ((lenguaje="0123456789").IndexOf(c)>=0)  _edoAct=7;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 7 : c=SigCar(ref i);
                    if ((lenguaje="0123456789").IndexOf(c)>=0)  _edoAct=7;  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=8;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 8 : i--;
                    return true;
                    break;
       //--------------  Automata  otros--------------
      case 9 : c=SigCar(ref i);
                    if ((lenguaje=";").IndexOf(c)>=0)  _edoAct=10;  else
                    if ((lenguaje=",").IndexOf(c)>=0)  _edoAct=10;  else
                    if ((lenguaje="[").IndexOf(c)>=0)  _edoAct=10;  else
                    if ((lenguaje="]").IndexOf(c)>=0)  _edoAct=10;  else
                     { i=iniToken;
                          return false; }
                    break;
      case 10 : return true;
                    break;
       }
     switch (_edoAct)
     {
            case 2 :      // Autómata  delim
            case 5 :      // Autómata  id
            case 8 :      // Autómata  num
                           --i;
                           return true;
     }
     return false;
  }

} // fin de la clase Automata
```

Bien, ya tenemos el código de las clases necesarias para poder definir un objeto analizador léxico en nuestra aplicación Windows C#. Volvamos a la aplicación y agreguemos las siguientes líneas de código en el archivo *Form1.cs* :

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

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

Notemos que hemos añadido la definición de un objeto ***oAnaLex*** perteneciente a la clase *Lexico*.

Si compilamos la aplicación, obtenemos un error ya que no hemos agregado al proyecto a la clase *Lexico*. Entonces debemos agregar la clase *Lexico* pero también a la clase *Automata* al proyecto. La figura #13.3 muestra el proyecto con las 2 nuevas clases añadidas. Observemos que en dicha figura la pestaña seleccionada es la de la clase Lexico, que se encuentra con su cuerpo vacio, es decir, no contiene ni atributos ni métodos.

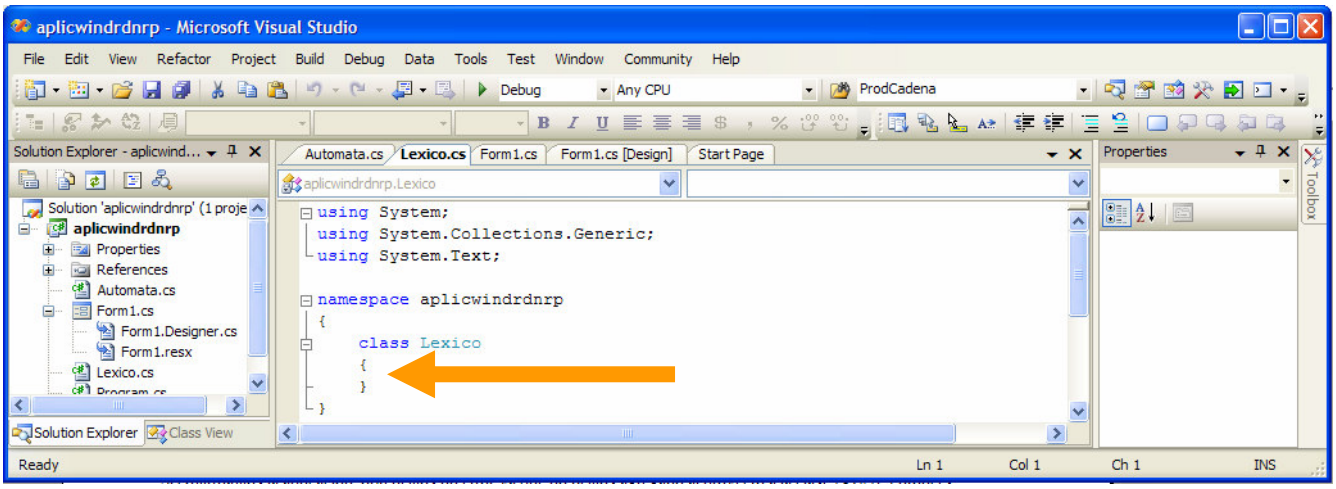


Fig. No. 13.3 Clases Lexico y Automata agregadas al proyecto, pero aún vacias.

Si compilamos hasta este punto, obtenemos un programa sin errores pero que no hace nada, figura #13.4.

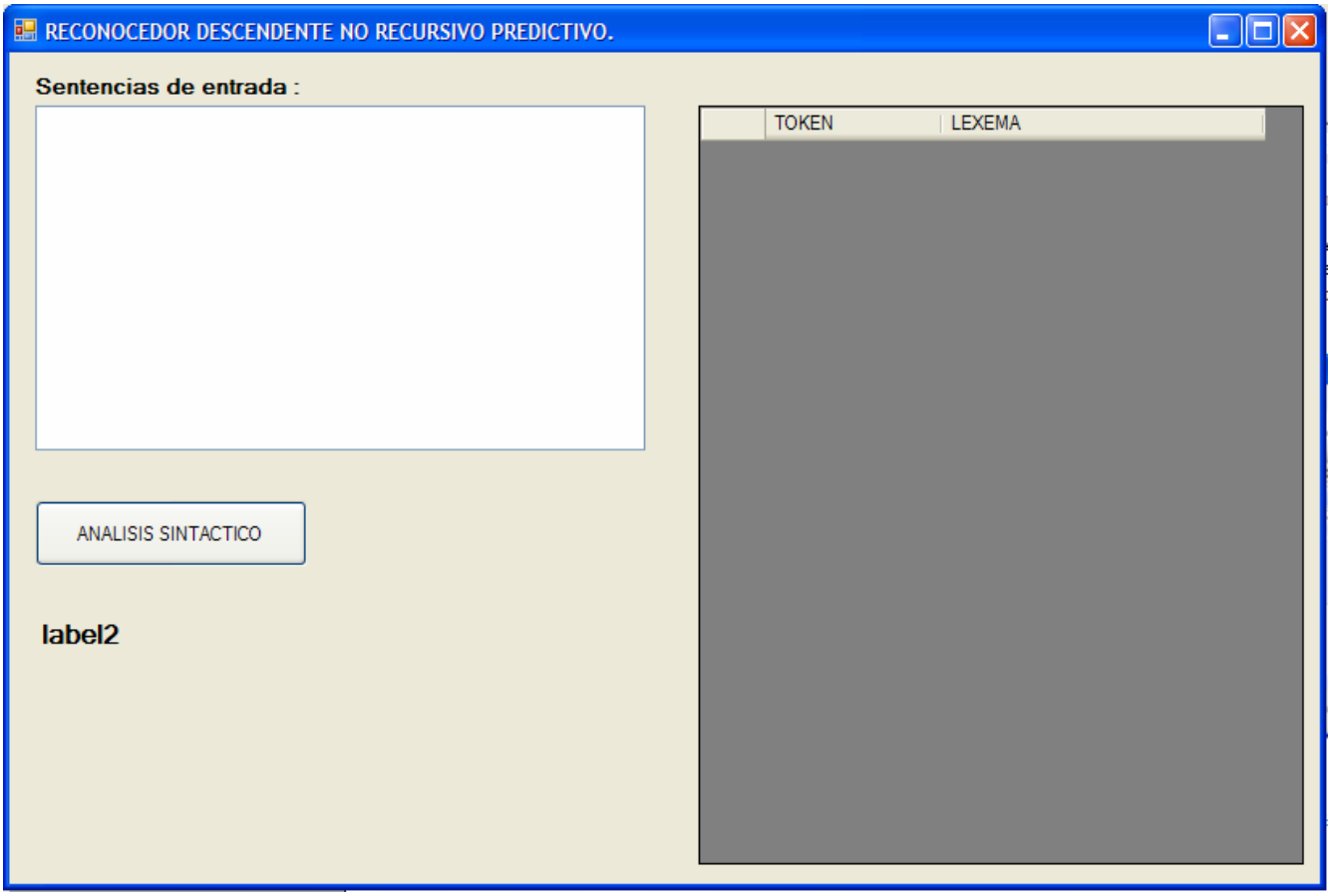


Fig. No. 13.4 Aplicación Windows en ejecución.

Continuamos agregando el código de las clases *Lexico* y *Automata* que generó el SP-PS1 a los cuerpos de las clases añadidas al proyecto. Cuidemos de no agregar el encabezado de la clase correspondiente ni las llaves del cuerpo, ya que el C# ya los añadió. Ya que añadimos el código al cuerpo de las 2 clases, debemos compilar el programa SIN ERRORES. La figura #13.5 muestra la clase *Lexico* con el código insertado.

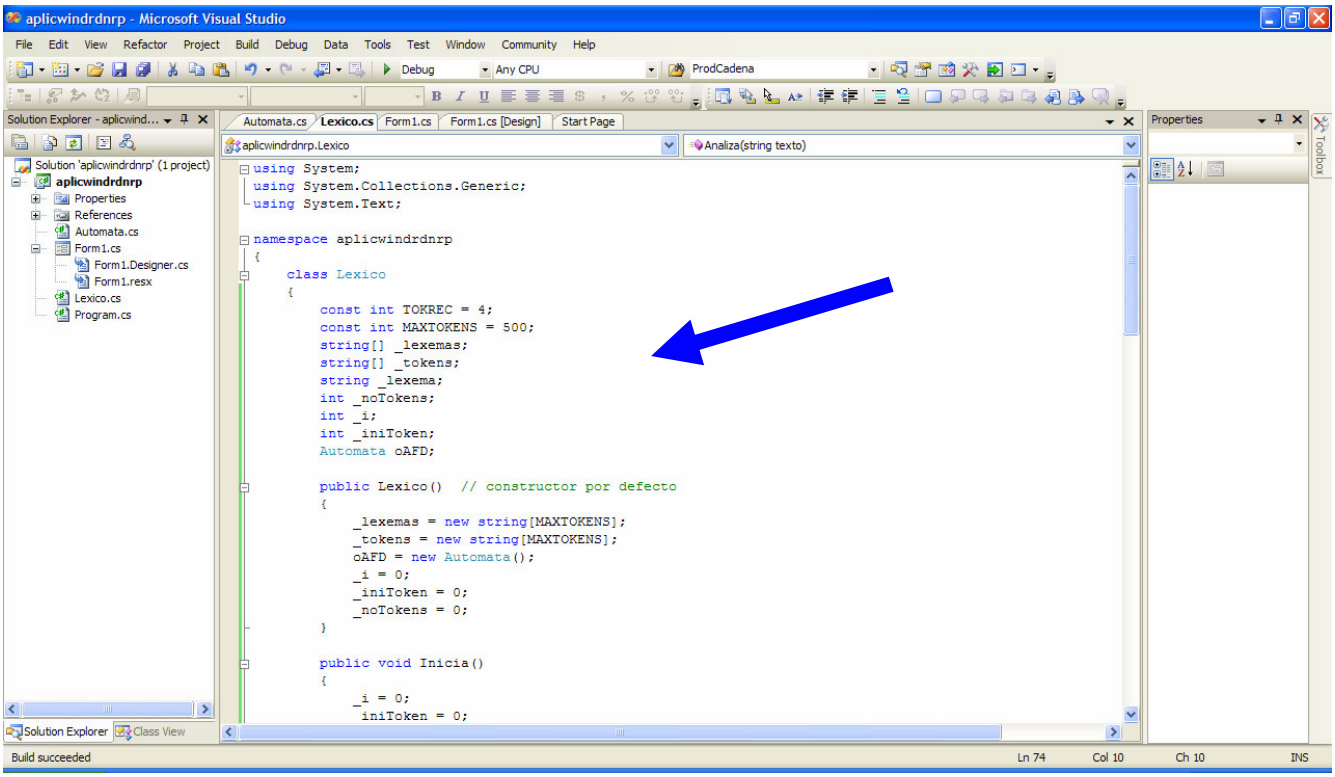


Fig. No. 13.5 Inserción del código generado en la clase Lexico.

Antes de probar al analizador léxico, debemos efectuar algunas modificaciones. Una de ellas es evitar almacenar al token *delim* y su lexema encontrado. Lo vamos a identificar pero no lo vamos almacenar dentro de los atributos del objeto **oAnaLex**. Entonces modificamos por medio de un comentario el caso en que se reconoce a un delimitador. Busca el código siguiente dentro de la clase *Lexico* en el 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: _tokens[_noTokens] = "id";
                            break;
```

Cambiemos lo señalado por la flecha roja comentando la instrucción donde almacenamos al token *delim*.

El código debe ser ahora :

```
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;
```

Si ya quitamos la instrucción donde almacenamos al token *delim*, debemos también no almacenar el lexema para dicho token. Veamos el código de este mismo método Analiza() unas líneas mas adelante :

```
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  num--------------
                        case 2: _tokens[_noTokens] = "num";
                            break;
                        //--------------  Automata  otros--------------
                        case 3: _tokens[_noTokens] = "otros";
                            break;
                    }
                    _lexemas[_noTokens++] = _lexema;
                }
                else
```

El código marcado `_lexemas[_noTokens++] = _lexema;` es el encargado de almacenar al lexema. NO debemos almacenar el lexema cuando el número del autómata **noAuto** es el 0 –*delim*-. Así que usemos un **if** para atrapar esta condición. La modificación para NO almacenar la pareja token-lexema para los delimitadores es entonces :

```
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  num--------------
                        case 2: _tokens[_noTokens] = "num";
                            break;
                        //--------------  Automata  otros--------------
                        case 3: _tokens[_noTokens] = "otros";
                            break;
                    }
                    if (noAuto != 0)
                        _lexemas[_noTokens++] = _lexema;
                }
                else
```

Compilemos y ejecutemos el programa de manera que veamos que NO tenemos errores. Si es así, ya estamos listos para seguir con la segunda modificación.

Otra modificación que debemos realizar es la de reconocer con el mismo AFD a los tokens *id* y *palres* –palabras reservadas *int* y *float*-. La manera de resolver esta cuestión es modificar el código del método *Analiza()* de la clase *Lexico*, cuando se almacena el token *id*.

```
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;
```

Definiremos un nuevo método llamado *EsId()* que retorna *true* si el lexema reconocido es un identificador, de lo contrario retorna *false*. La llamada a este método la efectuamos dentro del **case 1** marcado con la flecha roja :

```
case 1: if (EsId())
            _tokens[_noTokens] = "id";
        else
            _tokens[_noTokens] = _lexema;
        break;
```

La definición del método EsId() es la siguiente :

```
private bool EsId()
{
   string[] palres ={ "int", "float"};
   for (int i = 0; i < palres.Length; i++)
        if (_lexema==palres[i])
            return false;
   return true;
}
```

La última modificación al método *Analiza()* de la clase Lexico, es la que corresponde al AFD *otros*. Cuando reconoce el analizador léxico un caracter perteneciente al token *otros*, debemos de almacenar la pareja lexema-lexema, en lugar de la pareja token-lexema. Así que modificamos el código almacenando el atributo *_lexema* en lugar de la cadena "*otros*".

```
//--------------  Automata  otros--------------
    case 3: _tokens[_noTokens] = "otros";
            break;
```

Lo cambiamos por :

```
//--------------  Automata  otros--------------
    case 3: _tokens[_noTokens] = _lexema;
            break;
```

Ahora compilamos la aplicación. NO HAY ERRORES, sólo advertencias. Estamos listos para agregar código en el botón ANALISIS SINTACTICO. Vayamos al evento click del botón mencionado y agreguemos los mensajes que incluyen al 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];
        }
    }
}
```

Observemos que este código contiene mensajes que acceden a 3 propiedades de la clase *Lexico* :

- `NoTokens`
- `Token`
- `Lexema`

Antes de ejecutar la aplicación tecleemos estas 3 propiedades en la clase *Lexico* :

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

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

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

Ejecutemos la aplicación. En la ventana de entrada de texto tecleemos la sentencia **`int x, y[10];`** para luego hacer el análisis léxico accionando el botón ANALISIS SINTACTICO. La figura #13.6 muestra la respuesta de nuestra aplicación a dicha entrada y acción sobre el botón.

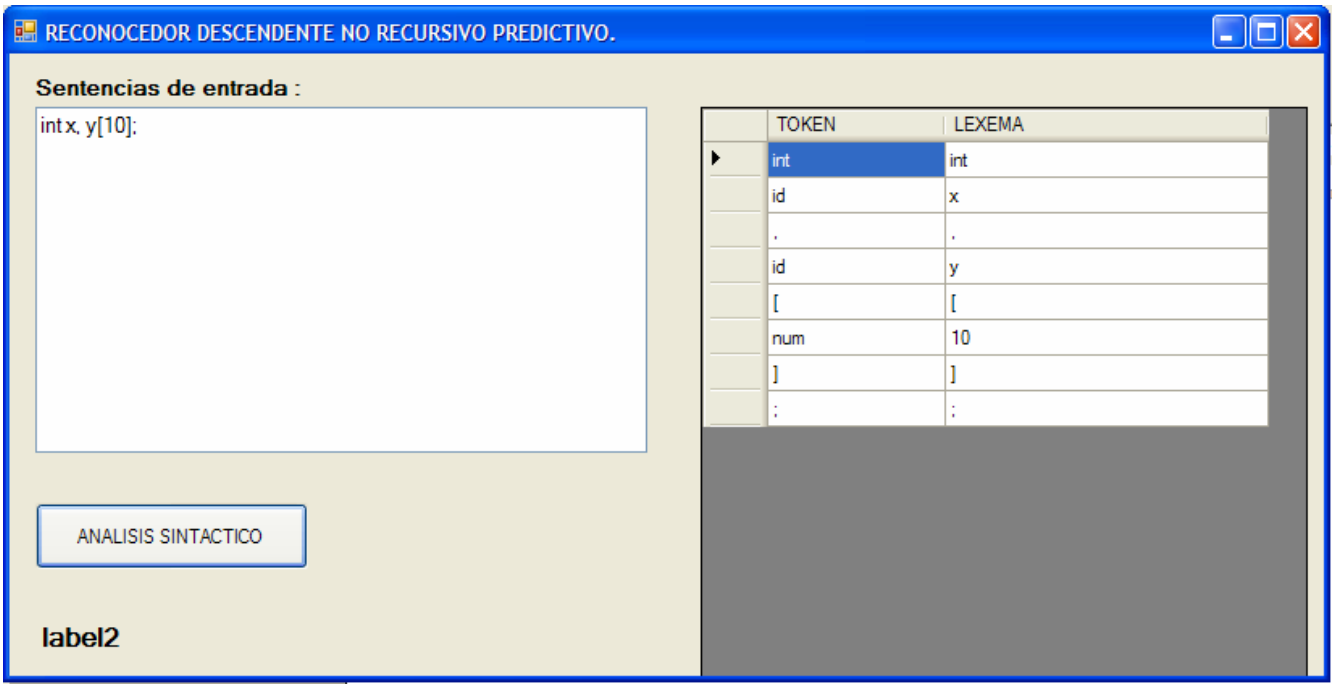


Fig. No. 13.6 Análisis léxico de la entrada **`int x, y[10];`**.

13.5 Inclusión del código generado por RD-NRP para la clase SintDescNRP.

Lo último que falta para completar la aplicación, es añadir el código generado por el programa RD.NRP, la clase *SintDescNRP*. Debemos añadir también las clases *Pila* y *SimbGram*.

Las clases *Pila* y *SimbGram* las podemos copiar de la pestaña *Otras clases* del software RD-NRP. Así que vamos a hacerlo, simultáneamente con la adición de las 2 clases, al proyecto de la aplicación que estamos construyendo.

Una vez hecha la adición de dichas clases al proyecto, debemos generar el código para la clase *SintDescNRP* usando el RD-NRP. La figura 13.7 muestra a las 3 clases *Pila*, *SimbGram* y *SintDescNRP* sumadas al proyecto.

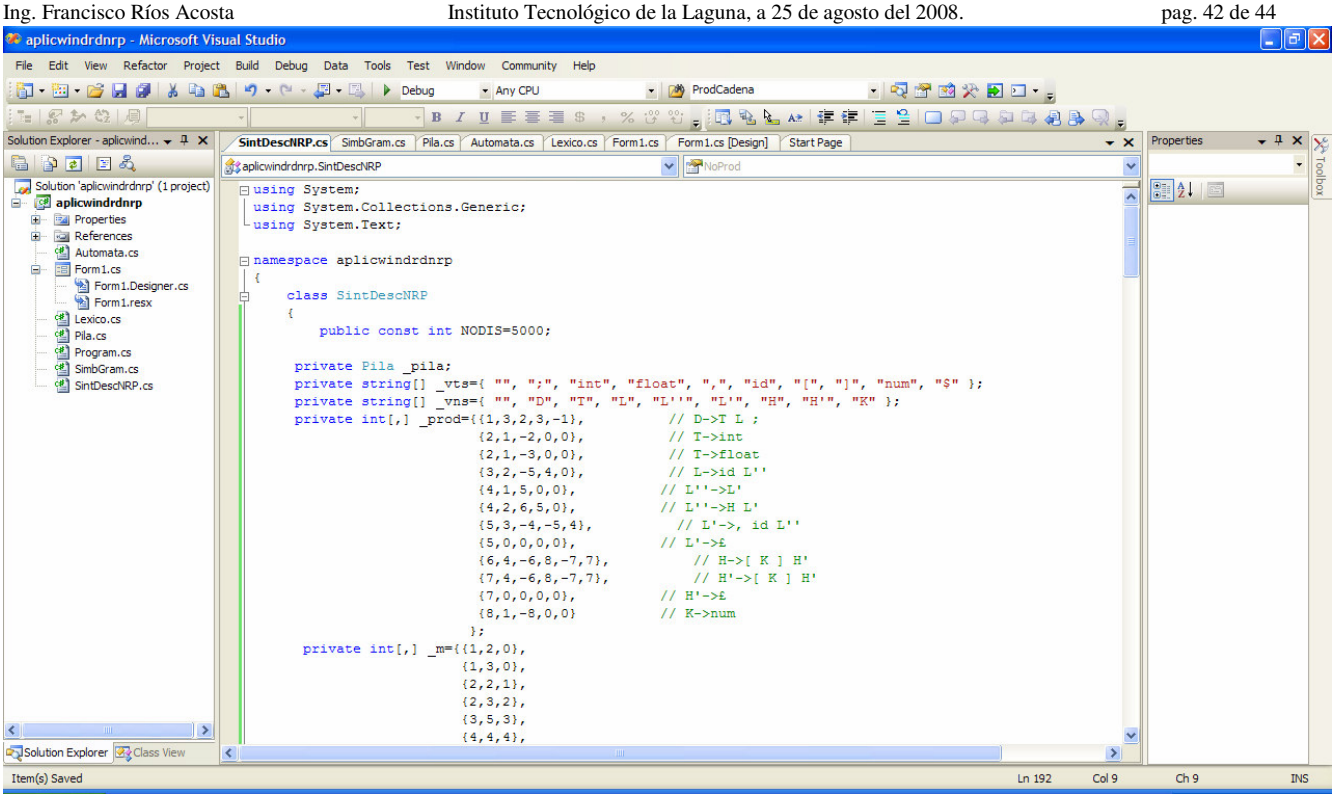


Fig. No. 13.7 Clases SintDescNRP, Pila y SimbGram.

Una vez que agregamos las 3 clases, seguimos con la inclusión de la definición del objeto *oAnaSintDesc* a nuestra aplicación. Vayamos al archivo *Form1.cs* e incluyamos el código señalado en la figura #13.8.

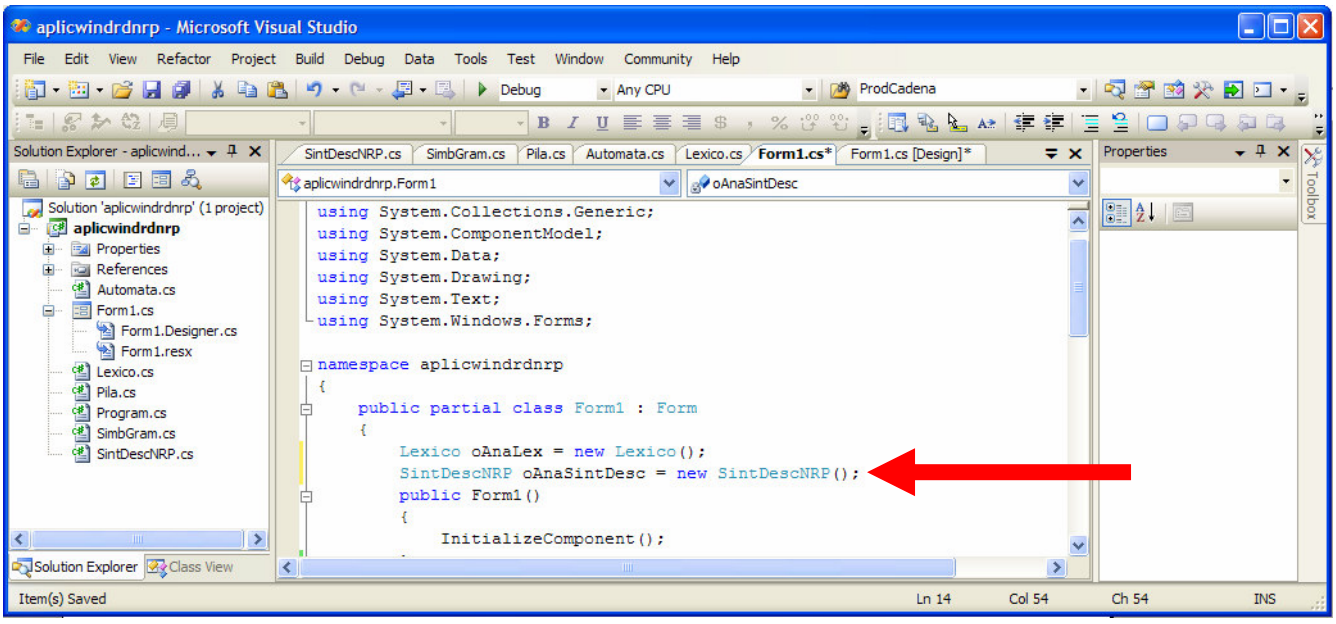


Fig. No. 13.7 Definición del objeto *oAnaSintDesc*.

Seguimos con la inclusión de los mensajes que inicializan al objeto *oAnaSintDesc* y que analizan la sentencia cuyos tokens que la componen son enviados por el objeto *oAnaLex* al objeto *oAnaSintDesc*.

Vayamos al botón ANALISIS SINTACTICO y agreguemos el código que se indica a continuación en su evento Click, figura #13.8.

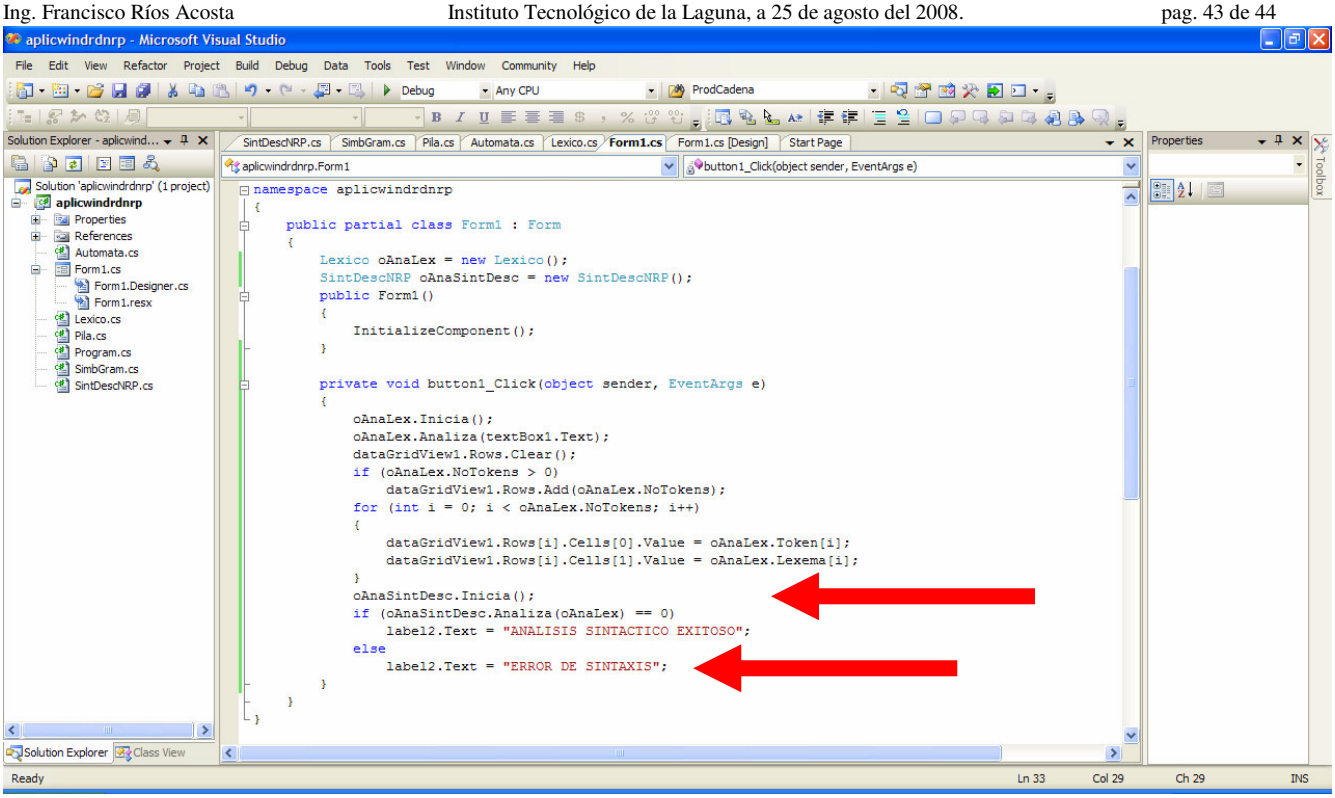


Fig. No. 13.8 Mensajes para el objeto *oAnaSintDesc* que efectúa el reconocimiento de la sentencia.

Compilemos la aplicación y obtenemos el mensaje de error mostrado en la figura 13.9. Nos hace falta definir un método dentro de la clase *Lexico* llamado *Anade()*.

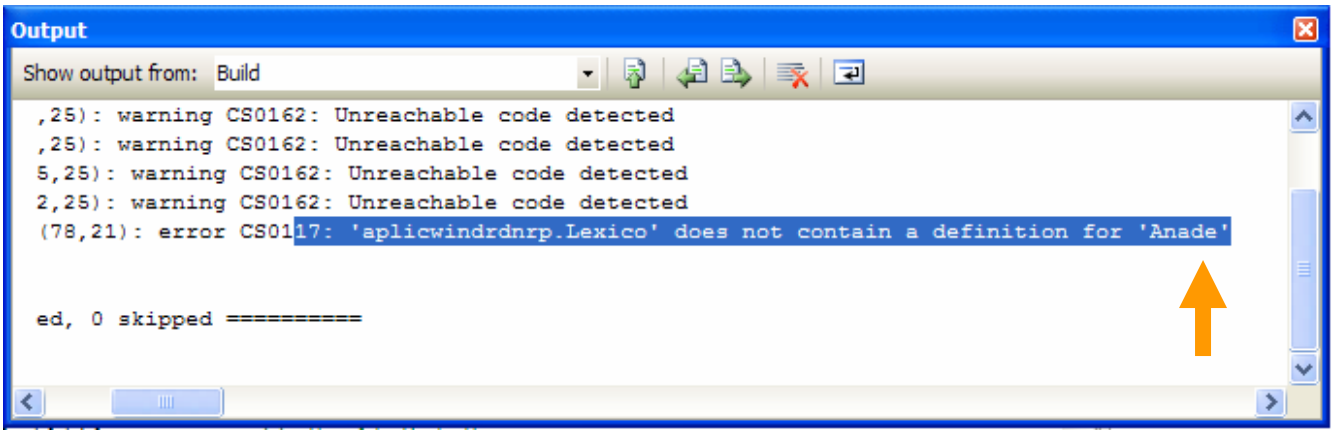


Fig. No. 13.9 Error al compilar la aplicación.

El método *Anade()* es llamado dentro del método *Analiza()* de la clase *SintDescNRP*, según lo vemos en el siguiente segmento de código de dicho método.

```
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;
```

La tarea del método *Anade()* es la de agregar una pareja token-lexema al objeto **oAnaLex** que lo llama. Es necesario hacer esto debido a que el modelo del libro del "dragón" así lo requiere en la estructura de datos de entrada **w$**.
Este arreglo de entrada contiene a los tokens que el analizador léxico ha reconocido previamente. En nuestra implementación, el objeto **oAnaLex** contiene el atributo *_tokens* que cumple con la tarea del **w$** conceptual.

La definición del método *Anade()* es :

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

Tenemos que agregar esta definición dentro de la clase *Lexico*. Una vez agregado el método Anade(), ya podemos ejecutar sin errores a nuestra aplicación Windows. Tecleemos la misma entrada `int x, y[10];` , notemos que el análisis sintactico es exitoso. La figura #13.10 así lo muestra.

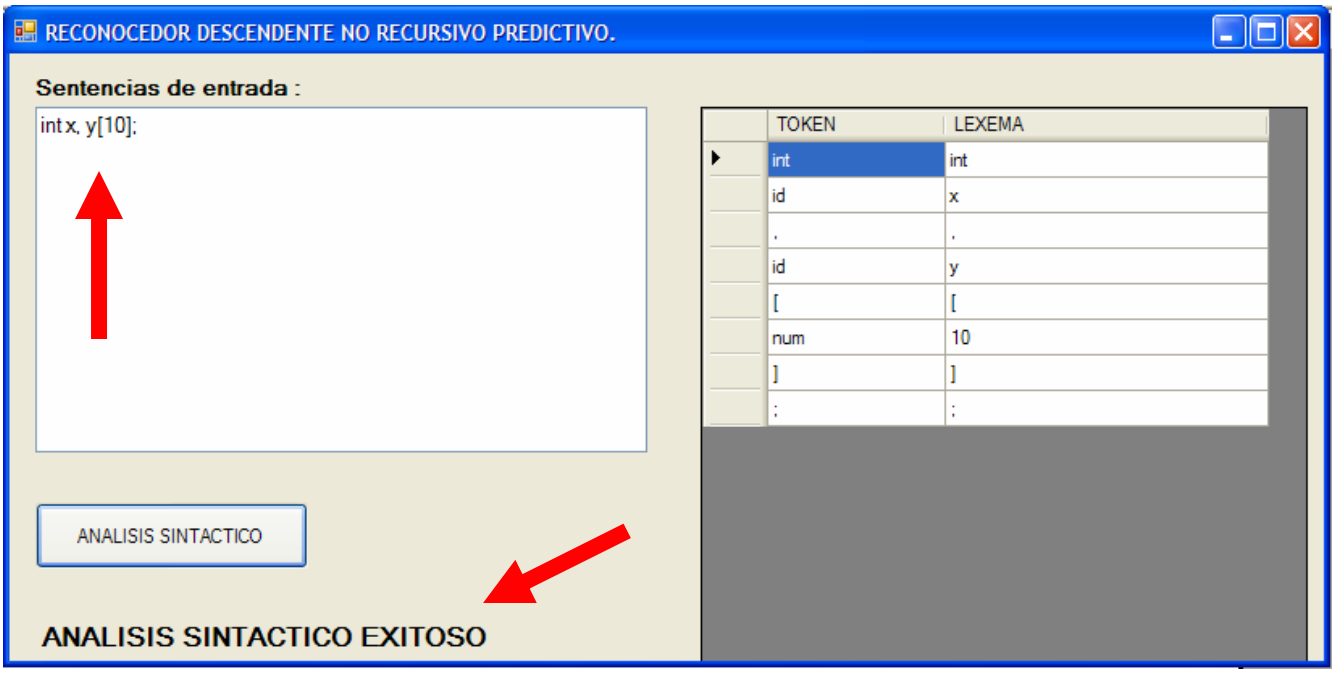


Fig. No. 13.10 Aplicación Windows C# reconociendo con éxito una sentencia de declaración de 2 variables enteras.