



**TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE VERACRUZ
VERACRUZ, VERACRUZ**



**CARRERA
INGENIERÍA EN SISTEMAS COMPUTACIONALES**

**MATERIA
LENGUAJES Y AUTÓMATAS I**

**DOCENTE
OFELIA GUTIÉRREZ GUIRALDI**

Manual de Sistema

**ESTUDIANTES
JOSÉ MANUEL MIRANDA VILLAGRÁN
LUIS ENRIQUE ESCAMILLA ALVAREZ
BRYAN CASTILLO MARÍN**

24 de agosto de 2019

Índice

Introducción	4
Teoría y conceptos	6
Análisis lexicográfico	7
<i>Concepto de analizador léxico</i>	<i>7</i>
<i>Funciones del analizador léxico</i>	<i>8</i>
<i>Token, patrón y lexema</i>	<i>9</i>
El generador de analizadores lexicográficos: Lex / Flex	10
<i>Funcionamiento en general</i>	<i>10</i>
<i>Creación de un analizador léxico</i>	<i>11</i>
<i>El lenguaje lex</i>	<i>13</i>
<i>Área de definiciones y área de funciones</i>	<i>13</i>
<i>Área de reglas</i>	<i>14</i>
<i>Premisas para reconocer lexemas</i>	<i>14</i>
<i>Caracteres especiales de lex</i>	<i>14</i>
<i>Caracteres de sensibilidad al contexto</i>	<i>16</i>
<i>Funciones, macros y variables suministradas por Flex</i>	<i>17</i>
Variables	17
Funciones	18
Macros	21
<i>Compilación y ejecución de un programa Flex</i>	<i>21</i>
Análisis sintáctico	22
<i>Manejo de errores</i>	<i>23</i>
<i>Gramática utilizada por un analizador sintáctico</i>	<i>24</i>
Definición de gramáticas	24
Convención de notación	24
Derivaciones	25
Arboles de análisis sintácticos	25
El generador de analizadores sintácticos: Yacc / Bison	26
Funcionamiento general.....	26
Formato de un programa Yacc	27

Área de definiciones	27
Declaración %union	29
Declaración %token.....	29
Declaración %type	29
Área de reglas	30
Área de funciones	30
Tratamiento de errores	31
Notificación de errores	31
Recuperación de errores	32
Errores léxicos y sintácticos	33
Compilación y ejecución de un programa Bison	34
Notas finales	35
Requerimientos de sistema.....	36
Diseño del archivo Lex	39
Diseño del archivo Yacc	44
Diseño del lenguaje.....	51
Diseño Interfaz Gráfica	63
Funcionamiento de la interfaz.....	80
Bibliografía	83

Introducción

Introducción

Una de las diferencias más grande que hay entre la ingeniería en sistemas computacionales ante cualquier otra licenciatura relacionada a la informática, es el estudio de los lenguajes base de computación y todo lo que estos conllevan; A lo largo de esta materia “Lenguajes y Autómatas I” hemos partido desde lo más fundamental, respondiendo preguntas como ¿Qué es un lenguaje? ¿Cuál es la definición de un alfabeto?, dichas preguntas han tenido sus respectivas respuestas mientras avanzábamos en las diversas unidades que engloba esta materia.

Pero entre todos esos diversos temas hay uno que destaca ante el resto y esto es simplemente porque engloba todo lo que conlleva diseñar un lenguaje con su respectiva gramática, dicho tema es el de *compiladores* y este lleva como practica el realizar un compilador partiendo de cero, solo contando con nuestros conocimientos de lógica y programación además de utilizar diversas herramientas para su optimo desarrollo.

Este proyecto tiene como objetivo marcar un antes y un después en la manera en la que vemos las cosas, en como razonamos y entendemos el funcionamiento de una computadora, hay muchas cosas que ignorábamos antes de llevar esta materia, ahora sabemos el proceso interno de un compilador para *traducir* un programa, ahora sabemos que no existe un análisis semántico si no existe un sintactito, o que no existe un análisis sintáctico si no se realizó antes un análisis léxico y gracias a este proyecto lo hemos llevado a la práctica.

Este compilador lleva como nombre “MJU” y está diseñado para el desarrollo de programas basados en el lenguaje homónimo al nombre del compilador. En este manual se explica de una manera detallada y bien documentada; Todo el proceso que se lleva a cabo cuando se programa en él.

Introducción

Análisis lexicográfico

Concepto de analizador léxico

Se encarga de buscar los componentes léxicos o palabras que componen el programa fuente, según unas reglas o patrones. La entrada del analizador léxico podemos definirla como una secuencia de caracteres, que pueda hallarse codificada según cualquier estándar: ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary Coded Decimal Interchange Code), Unicode, etc. El analizador léxico divide esta secuencia en palabras con significado propio y después las convierte a una secuencia de terminales desde el punto de vista del analizador sintáctico. Dicha secuencia es el punto de partida para que el analizador sintáctico construya el árbol sintáctico que reconoce la/s sentencia/s de entrada, tal y como puede verse en la **Figura 1**.

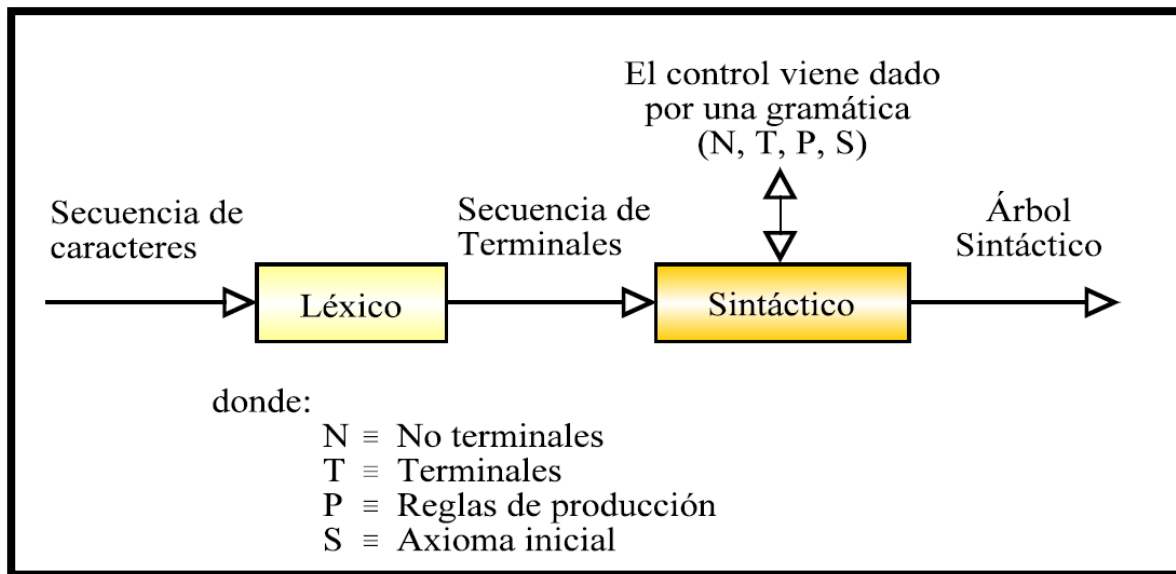


Figura 1 - Entradas y salidas de las dos primeras fases de la etapa de análisis. La frase “Secuencia de Terminales” hace referencia a la gramática del sintáctico; pero también es posible considerar que dicha secuencia es de no terminales si usamos el punto de vista del lexicográfico.

El analizador léxico reconoce las palabras en función de una gramática regular de manera que el alfabeto Σ de dicha gramática son los distintos caracteres del juego de caracteres del ordenador sobre el que se trabaja (que forman el conjunto de símbolos terminales), mientras que sus no terminales son las categorías léxicas en que se integran las distintas secuencias de caracteres. Cada no terminal o categoría léxica de la gramática regular del análisis léxico es considerado como un terminal de la gramática de contexto libre con la que trabaja el

analizador sintáctico, de manera que la salida de alto nivel (no terminales) de la fase léxica supone la entrada de bajo nivel (terminales) de la fase sintáctica. En el caso de Lex, por ejemplo, la gramática regular se expresa mediante expresiones regulares.

Funciones del analizador léxico

El analizador léxico es la primera fase de un compilador. Su principal función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis. Esta interacción suele aplicarse convirtiendo al analizador léxico en una subrutina o corrutina del analizador sintáctico. Recibida la orden “Dame el siguiente componente léxico” del analizador sintáctico, el léxico lee los caracteres de entrada hasta que pueda identificar el siguiente componente léxico, el cual devuelve al sintáctico según el formato convenido **Figura 2**.

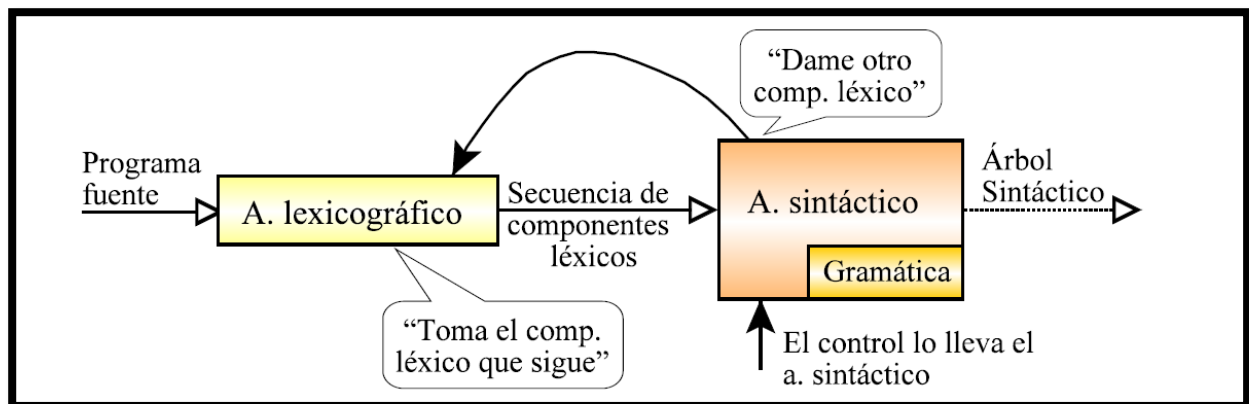


Figura 2 - La fase de análisis léxico se halla bajo el control del análisis sintáctico. Normalmente se implementa como una función de éste.

Además de esta función principal, el analizador léxico también realiza otras de gran importancia, a saber:

- Eliminar los comentarios del programa.
- Eliminar espacios en blanco, tabuladores, retorno de carro, etc., y en general, todo aquello que carezca de significado según la sintaxis del lenguaje.
- Reconocer los identificadores de usuario, números, palabras reservadas del lenguaje, etc., y tratarlos correctamente con respecto a la tabla de símbolos (solo en los casos en que este analizador deba tratar con dicha estructura).

- Llevar la cuenta del número de línea por la que va leyendo, por si se produce algún error, dar información acerca de dónde se ha producido.
- Avisar de errores léxicos. Por ejemplo, si el carácter '@' no pertenece al lenguaje, se debe emitir un error.
- También puede hacer funciones de preprocesador.

Token, patrón y lexema

Desde un punto de vista muy general, podemos abstraer el programa que implementa un análisis lexicográfico mediante una estructura como:

$(Expresion\ regular)_1$	$\{accion\ a\ ejecutar\}_1$
$(Expresion\ regular)_2$	$\{accion\ a\ ejecutar\}_2$
$(Expresion\ regular)_3$	$\{accion\ a\ ejecutar\}_3$
...	...
$(Expresion\ regular)_n$	$\{accion\ a\ ejecutar\}_n$

donde cada acción a ejecutar es un fragmento de programa que describe cuál ha de ser la acción del analizador léxico cuando la secuencia de entrada coincida con la expresión regular. Normalmente esta acción suele finalizar con la devolución de una categoría léxica.

Todo esto nos lleva a los siguientes conceptos de fundamental importancia a lo largo de nuestro estudio:

- **Patrón:** es una expresión regular.
- **Token:** es la categoría léxica asociada a un patrón. Cada *token* se convierte en un número o código identificador único. En algunos casos, cada número tiene asociada información adicional necesaria para las fases posteriores de la etapa de análisis. El concepto de *token* coincide directamente con el concepto de terminal desde el punto de vista de la gramática utilizada por el analizador sintáctico.
- **Lexema:** Es cada secuencia de caracteres concreta que encaja con un patrón. P.ej: "8", "23" y "50" son algunos lexemas que encajan con el patrón $(0|1|2| \dots |9)^+$. El número de lexemas que puede encajar con un patrón puede ser finito o infinito, p.ej. en el patrón $W'H'I'L'E$ sólo encaja el lexema "WHILE".

Una vez detectado que un grupo de caracteres coincide con un patrón, se considera que se ha detectado un lexema. A continuación, se le asocia el número de su categoría léxica, y dicho número o token se le pasa al sintáctico junto con información adicional, si fuera necesario.

El generador de analizadores lexicográficos: Lex / Flex

Lex es una herramienta de los sistemas UNIX/Linux que nos va a permitir generar código C que luego podremos compilar y enlazar con nuestro programa. La principal característica de Lex es que nos va a permitir asociar acciones descritas en C, a la localización de las Expresiones Regulares que le hayamos definido. Para ello Lex se apoya en una plantilla que recibe como parámetro, y que deberemos diseñar con cuidado. Internamente Lex va a actuar como un autómata que localizará las expresiones regulares que le describamos, y una vez reconocida la cadena representada por dicha expresión regular, ejecutará el código asociado.

Flex es una herramienta que permite generar analizadores léxicos. A partir de un conjunto de expresiones regulares, Flex busca concordancias en un fichero de entrada y ejecuta acciones asociadas a estas expresiones. Es compatible casi al 100% con Lex, una herramienta clásica de Unix para la generación de analizadores léxicos, pero es un desarrollo diferente realizado por GNU bajo licencia GPL.

Funcionamiento en general

Como ya sabemos, las reglas de reconocimiento son de la forma:

$$P_1\{accion_1\}$$

$$P_2\{accion_2\}$$

...

$$P_n\{accion_n\}$$

donde P_i es una expresión regular y $accion_i$ es un fragmento de programa que describe cuál ha de ser la acción del analizador léxico cuando se encuentra con un lexema que encaja por P_i . En Lex, las acciones se escriben en C, aunque existen multitud de meta compiladores

similares a Flex que permiten codificar en otros lenguajes, como por ejemplo Jflex que utiliza el lenguaje Java.

Un analizador léxico creado por Flex está diseñado para comportarse en sincronía con un analizador sintáctico. Para ello, entre el código generado existe una función llamada **yylex()** que, al ser invocada (normalmente por el sintáctico), comienza a leer la entrada, carácter a carácter, hasta que encuentra el mayor prefijo en la entrada que concuerde con una de las expresiones regulares P_i ; dicho prefijo constituye un lexema y, una vez leído, se dice que “ha sido consumido” en el sentido de que el punto de lectura ha avanzado hasta el primer carácter que le sigue. A continuación, **yylex()** ejecuta la acción. Generalmente esta acción devolverá el control al analizador sintáctico informándole del token encontrado. Sin embargo, si la acción no tiene un **return**, el analizador léxico se dispondrá a encontrar un nuevo lexema, y así sucesivamente hasta que una acción devuelva el control al analizador sintáctico, o hasta llegar al final de la cadena, representada por el carácter EOF (End Of File). La búsqueda repetida de lexemas hasta encontrar una instrucción **return** explícita permite al analizador léxico procesar espacios en blanco y comentarios de manera apropiada.

Antes de ejecutar la acción asociada a un patrón, **yylex()** almacena el lexema leído en la variable **yytext**. Cualquier información adicional que se quiera comunicar a la función llamante (normalmente el analizador sintáctico), además del token debe almacenarse en la variable global **yy1val**.

Creación de un analizador léxico

Como ya se ha comentado, Flex tiene su propio lenguaje, al que llamaremos Lex y que permite especificar la estructura abstracta de un analizador léxico, tanto en lo que respecta a las expresiones regulares como a la acción a tomar al encontrar un lexema que encaje en cada una de ellas.

Los pasos para crear un analizador léxico con esta herramienta son **Figura 3**:

- Construir un fichero de texto en lenguaje Lex que contiene la estructura abstracta del analizador.

- Meta compilar el fichero anterior con Flex. Así se obtendrá un fichero fuente en C estándar. Algunas veces hay que efectuar modificaciones directas en este código, aunque las últimas versiones de Flex han disminuido al máximo estas situaciones.
- Compilar la fuente en C generado por Flex con un compilador C, con lo que obtendremos un ejecutable que sigue los pasos descritos.

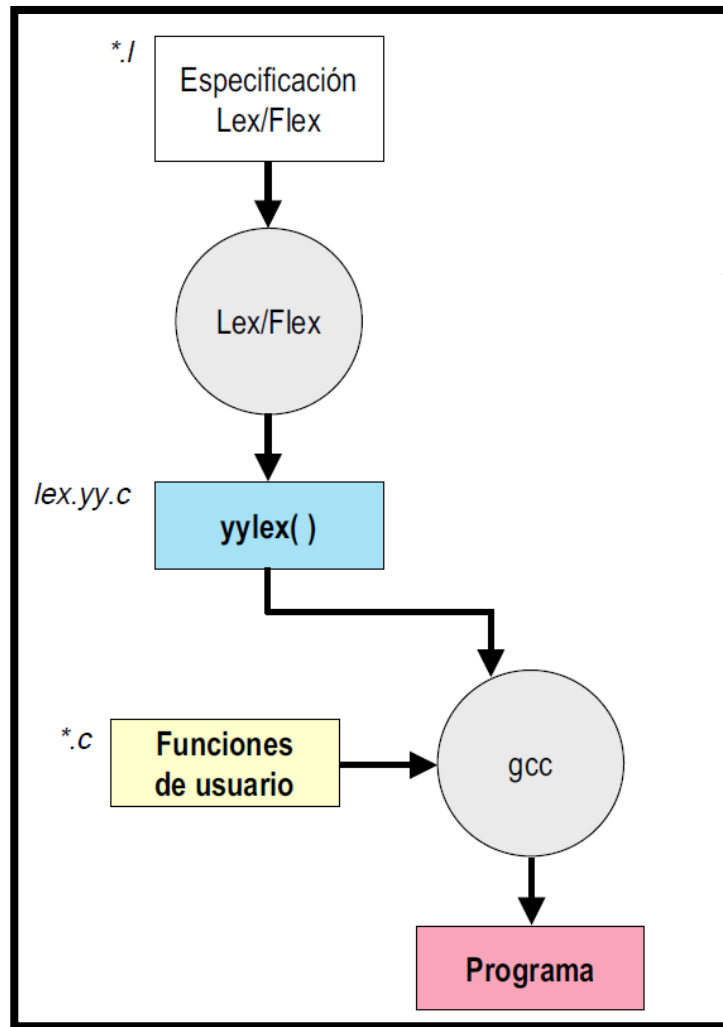


Figura 3 - Obtención de un programa ejecutable a partir de una especificación en Lex.

El lenguaje lex

Un programa fuente de Lex tiene el siguiente aspecto:

<sección de definiciones>

%%

<sección de reglas>

%%

<sección de funciones>

Como vemos, un programa LEX consta de 3 secciones, separadas por `%%`. La primera sección se denomina sección de definiciones, la segunda sección de reglas y la tercera sección de código. La primera y la última son opcionales.

Área de definiciones y área de funciones

El área de definiciones de un programa Lex tiene tres utilidades fundamentales:

- a) Definir los estados léxicos.
- b) Asignar un nombre a los patrones más frecuentes.
- c) Poner código C que será global a todo el programa.

En el área de definiciones podemos crear expresiones regulares auxiliares de uso frecuente y asignarles un nombre. Posteriormente, estos patrones pueden ser referenciados en el área de reglas sin más que especificar su nombre entre llaves: `{}`.

Así, el área de definiciones también permite colocar código C puro que se trasladará tal cual al comienzo del programa en C generado por Flex. Para ello se usan los delimitadores `%{` y `%}`. Es normal poner en esta zona las declaraciones de variables globales utilizadas en las acciones del área de reglas y un **#include** del fichero que implementa la tabla de símbolos.

Flex no genera **main()** alguno, sino que éste debe ser especificado por el programador. Por regla general, cuando se pretende ejecutar aisladamente un analizador léxico (sin un sintáctico asociado), lo normal es que las acciones asociadas a cada regla no contengan ningún **return** y que se incluya un **main()** que únicamente invoque a **yylex()**.

Para especificar el **main()** y cuantas funciones adicionales se estimen oportunas, se dispone del área de funciones. Dicha área es íntegramente copiada por Flex en el fichero C generado. Es por ello que, a efectos prácticos, escribir código entre “%{” y “%}” en el área de definiciones es equivalente a escribirlo en el área de funciones.

Área de reglas

Se definen los patrones de los lexemas que se quieren buscar a la entrada, y al lado de tales expresiones regulares, se detallan (en C) las acciones a ejecutar tras encontrar una cadena que se adapte al patrón indicado. Los separadores “%%” deben ir obligatoriamente en la columna 0, al igual que las reglas y demás información propia del lenguaje Lex.

Premisas para reconocer lexemas

El **yylex()** generado por Flex sigue dos directrices fundamentales para reconocer lexemas en caso de ambigüedad. Estas directrices son, por orden de prioridad:

1. Entrar siempre por el patrón que reconoce el lexema más largo posible.
2. En caso de conflicto usa el patrón que aparece en primera posición.

Como consecuencia de la segunda premisa: los patrones que reconocen palabras reservadas se colocan **siempre** antes que el patrón de identificador de usuario.

Caracteres especiales de lex

Los patrones en la entrada se escriben utilizando un conjunto extendido de expresiones regulares y usando como alfabeto cualquier carácter ASCII. Cualquier símbolo excepto el espacio en blanco, tabulador, cambio de línea y los caracteres especiales se escriben tal cual en las expresiones regulares (patrones) de Flex. Los caracteres especiales son:

Patrón	Descripción
"	Sirve para encerrar cualquier cadena de literales. Por regla general no es necesario encerrar los literales entre comillas a no ser que incluyan símbolos especiales, esto es, el patrón "WHILE" y el patrón WHILE son equivalentes; pero para representar p.ej. el inicio de comentario en Modula-2 sí es necesario entrecomillar los caracteres que componen al patrón: "(*", ya que éste contiene, a su vez, símbolos especiales.
\	Hace literal al siguiente carácter. También se utiliza para expresar aquellos caracteres que no tienen representación directa por pantalla: \n para el retorno de carro, \t para el tabulador, etc.
x	Empareja el carácter 'x'
[]	Permiten especificar listas de caracteres, o sea uno de los caracteres que encierra, ej.: [abc] reconoce o la 'a', o la 'b', o la 'c', ([abc] \equiv (a b c)). Dentro de los corchetes los siguientes caracteres también tienen un sentido especial: <ul style="list-style-type: none"> - : indica rango. Ej.: [A-Z0-9] reconoce cualquier carácter de la 'A' a la 'Z' o del '0' a '9'. ^ : indica compleción cuando aparece al comienzo, justo detrás de "[".
?	aquello que le precede es opcional. Ej.: a? \equiv (a g).
.	Representa a cualquier carácter (pero sólo a uno) excepto el retorno de carro \n. Es muy interesante porque nos permite recoger cualquier otro carácter que no sea reconocido por los patrones anteriores.
	Indica opcionalidad (OR).

*	Indica repetición 0 o más veces de lo que le precede.
+	Indica repetición 1 o más veces de lo que le precede.
()	Permiten la agrupación (igual que en las expresiones aritméticas).
{ }	indican rango de repetición. Ej.: a {1,3} ≡ a,aa,aaa. Las llaves vienen a ser algo parecido a un * restringido. También nos permite asignarle un nombre a una expresión regular para reutilizarla en múltiples patrones.
\x2a	El carácter cuyo código hexadecimal es 2a.
<<EOF>>	Final de un fichero.

Caracteres de sensibilidad al contexto

Carácter	Descripción
\$	El patrón que le precede sólo se reconoce si está al final de la línea. Ej.: (a b cd)\$. Que el lexema se encuentre al final de la línea quiere decir que viene seguido por un retorno de carro, o bien por EOF. El carácter que identifica el final de la línea no forma parte del lexema.
^	Fuera de los corchetes indica que el patrón que le sucede sólo se reconoce si está al comienzo de la línea. Que el lexema se encuentre al principio de la línea quiere decir que viene precedido de un retorno de carro, o que se encuentra al principio del fichero.

Funciones, macros y variables suministradas por Flex

El núcleo básico del programa en C generado por Flex es la función **yylex()**, que se encarga de buscar un lexema y ejecutar su acción asociada. Este proceso lo realiza iterativamente hasta que en una de las acciones se encuentre un return o se acabe la entrada. Además, Flex suministra otras funciones macros y variables de apoyo:

Variables

Variable	Tipo	Descripción
yytext	char* o char[]	Contiene el lexema actual.

En la sección de definiciones es posible utilizar las directivas **%pointer** o **%array**. Estas directivas hacen que **yytext** se declare como un puntero o un array respectivamente. La opción por defecto es declararlo como un puntero, salvo que se haya usado la opción **-l** en la línea de comandos, para garantizar una mayor compatibilidad con LEX. Sin embargo, y aunque la opción **%pointer** es la más eficiente (el análisis es más rápido y se evitan los buffers overflow), limita la posible manipulación de **yytext** y de las llamadas a **unput()**.

Variable	Tipo	Descripción
yylength	int	Número de caracteres del lexema actual. yylength = strlen(yytext)
yyin	FILE*	Apunta al fichero de entrada que se lee. Inicialmente apunta a stdin, por lo que el fichero de entrada coincide con la entrada estándar.
yyval	struct	Es una variable global que permite la comunicación con el sintáctico. Contienen la estructura de datos de la pila con la que trabaja YACC. Sirve para el intercambio de información entre ambas
yyval		

		herramientas.
yyout	FILE*	Apunta al fichero de salida (inicialmente stdout).
yylineno	int	Debe ser creada, inicializada y actualizada por el programador. Sirve para mantener la cuenta del número de línea que se está procesando. Normalmente se inicializa a 1 y se incrementa cada vez que se encuentra un retorno de carro.

Funciones

Método	Descripción
yylex()	Implementa el analizador lexicográfico.

Por defecto la función **yylex()** que realiza el análisis léxico es declarada como **int yylex()**. Es posible cambiar la declaración por defecto utilizando la macro **YY_DECL**. En el siguiente ejemplo la definición:

```
#define YY_DECL char *scanner(int *numcount, int *idcount)
```

hace que la rutina de análisis léxico pase a llamarse scanner y tenga dos parámetros de entrada, retornando un valor de tipo char *.

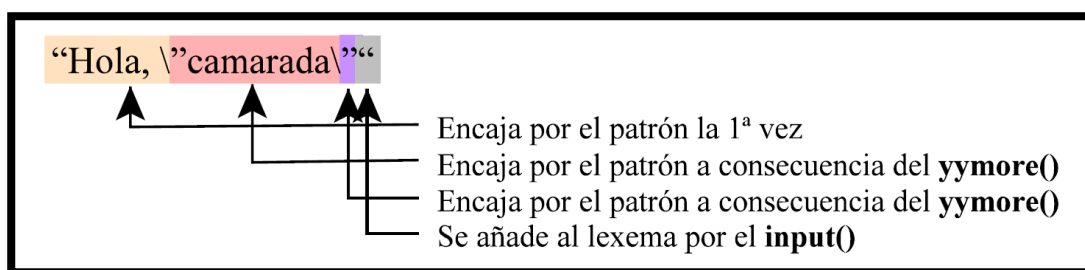
Método	Descripción
yymore()	Añade el yytext actual al siguiente reconocido.

Permite pasar a reconocer el siguiente lexema, pero sin borrar el contenido actual de **yytext**, por lo que el nuevo lexema leído se concatena al ya existente.

La utilidad principal de **yymore()** reside en que facilita el reconocer los literales entrecomillados. Supongamos que se desea construir un patrón para reconocer literales entrecomillados. Una primera idea podría ser el patrón:

```
\"[^"]]*\/"    { if(yytext[ yyleng - 1 ] == '\\')
                  yymore();
                  else
                    input(); }
```

Este patrón reconoce cadenas entrecomilladas (excepto las últimas comillas), de forma que las cadenas que tienen dentro comillas se reconocen por partes. Siguiendo con el ejemplo de antes, “Hola, \"camarada\"” se reconocería como:



Método	Descripción
<code>yylless(int n)</code>	Deja en <code>yytext</code> los <code>n</code> primeros caracteres del lexema actual. El resto los devuelve a la entrada, por lo que podemos decir que son desconsumidos. <code>yyleng</code> también se modifica convenientemente.
<code>yywrap()</code>	Se invoca automáticamente al encontrar el final del fichero de entrada.

Cuando el analizador léxico alcanza el final del fichero, el comportamiento en las subsiguientes llamadas a `yylex` resulta indefinido. En el momento en que **yylex()** alcanza el final del fichero llama a la función **yywrap**, la cual retorna un valor de 0 o 1 según haya más entrada o no. Si el valor es 0, la función `yylex` asume que la propia **yywrap** se ha encargado

de abrir el nuevo fichero y asignárselo a **yyin**. Otra manera de continuar es haciendo uso de la función **yyrestart(FILE *file)**.

La opción **noyywrap** evita se invoque a la función **yywrap()** cuando se encuentre un fin de fichero, y se asuma que no hay más ficheros que analizar. Esta solución es más cómoda que tener que escribir la función o bien enlazar con alguna librería.

Método	Descripción
<code>input()</code>	Consume el siguiente carácter del flujo de entrada y lo añade al lexema actual.

Por ejemplo, el programa:

```
%%  
abc      { printf ("%s", yytext);  
          input( );  
          printf("%s", yytext); }
```

ante la entrada “abcde” entraría por este patrón: el lexema antes del **input()** (en el primer **printf**) es “abc”, y después del **input** (en el segundo **printf**) es “abcd”.

Método	Descripción
<code>output(char c)</code>	Emite el carácter c por la salida estándar.
<code>unput(char c)</code>	Des-consume el carácter c y lo coloca al comienzo de la entrada.

Coloca el carácter c en el flujo de entrada, de manera que será el primer carácter leído en próxima ocasión.

Por ejemplo, supongamos que el programa:

```
%%  
abc      { printf("%s", yytext );  
          unput( 't' ); }  
tal      { printf( "%s", yytext ); }
```

recibe la entrada “abca1”. Los tres primeros caracteres del lexema coinciden con el primer patrón. Después queda en la entrada la cadena ”al“, pero como se ha hecho un `unput('t')`, lo que hay realmente a la entrada es “ta1”, que coincide con el segundo patrón.

Macros

Método	Descripción
ECHO	Macro que copia la entrada en la salida (es la acción que se aplica por defecto a los lexemas que no encajan por ningún patrón explícito).
REJECT	Rechaza el lexema actual, lo devuelve a la entrada y busca otro patrón.

Compilación y ejecución de un programa Flex

Al ejecutar el comando `flex nombre_fichero_fuente.l` se creará un fichero en C llamado “`lex.yy.c`”. Si se compila este fichero con la instrucción “`gcc lex.yy.c -o nombre_ejecutable`” se obtendrá como resultado un fichero ejecutable llamado `nombre_ejecutable.exe`).

Una vez se ha creado el fichero ejecutable se puede ejecutar directamente. Flex esperará que se introduzca texto por la entrada estándar (teclado) y lo analizará cada vez que se pulse el retorno de carro. Para terminar con el análisis normalmente hay que pulsar `Ctrl + C`. Para poder probarlo con entradas más largas, lo más sencillo es crear archivos de texto y usar redirecciones para que el ejecutable lea estos ficheros como entrada a analizar. Por ejemplo si hemos creado un analizador llamado `prueba1` y un fichero de texto con datos para su análisis, llamado `entrada.txt`, podemos ejecutar `prueba1 < entrada.txt`.

Análisis sintáctico

Es la fase del analizador que se encarga de chequear la secuencia de tokens que representa al texto de entrada, en base a una gramática dada **Figura 4**. En caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce en base a una representación computacional. Este árbol es el punto de partida de la fase posterior de la etapa de análisis: el analizador semántico.

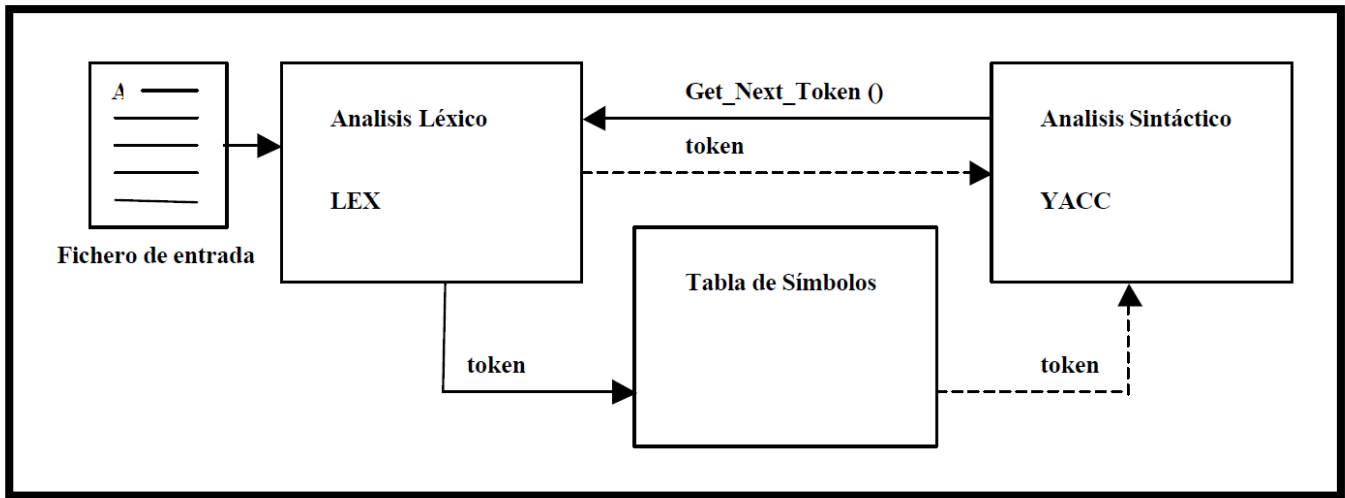


Figura 4 - Posición del analizador sintáctico en el modelo del compilador.

Pero esto es la teoría; en la práctica, el analizador sintáctico dirige el proceso de compilación, de manera que el resto de fases evolucionan a medida que el sintáctico va reconociendo la secuencia de entrada por lo que, a menudo, el árbol ni siquiera se genera realmente.

En la práctica, el analizador sintáctico también:

- Incorpora acciones semánticas en las que coloca en el resto de fases del compilador (excepto el analizador léxico): desde el análisis semántico hasta la generación de código.
- Informa de la naturaleza de los errores sintácticos que encuentra e intenta recuperarse de ellos para continuar la compilación.
- Controla el flujo de tokens reconocidos por parte del analizador léxico.

En definitiva, realiza casi todas las operaciones de la compilación, dando lugar a un método de trabajo denominado compilación dirigida por sintaxis.

Manejo de errores

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implementación se simplificarían mucho. Las primeras versiones de los programas suelen ser incorrectas, y un buen compilador debería ayudar al programador a identificar y localizar errores. Es más, considerar desde el principio el manejo de errores puede simplificar la estructura de un compilador y mejorar su respuesta a los errores.

Los errores en la programación pueden ser de los siguientes tipos:

- **Léxicos**, producidos al escribir mal un identificador, una palabra clave o un operador.
- **Sintácticos**, por una expresión aritmética o paréntesis no equilibrados.
- **Semánticos**, como un operador aplicado a un operando incompatible.
- **Lógicos**, puede ser una llamada infinitamente recursiva.
- **De corrección**, cuando el programa no hace lo que el programador realmente deseaba.

El manejo de errores de sintaxis es el más complicado desde el punto de vista de la creación de compiladores. Nos interesa que cuando el compilador encuentre un error, no cancele definitivamente la compilación, sino que se recupere y siga buscando errores. Recuperar un error no quiere decir corregirlo, sino ser capaz de seguir construyendo el árbol sintáctico a pesar de los errores encontrados. En vista de esto, el manejador de errores de un analizador sintáctico debe tener como objetivos:

- **Indicar los errores de forma clara y precisa.** Debe informar mediante los correspondientes mensajes del tipo de error y su localización.
- **Recuperarse del error**, para poder seguir examinando la entrada.
- **Distinción entre errores y advertencias.** Las advertencias se suelen utilizar para informar sobre sentencias válidas pero que, por ser poco frecuentes, pueden constituir una fuente de errores lógicos.
- **No ralentizar** significativamente la compilación.

Gramática utilizada por un analizador sintáctico

La formalización del lenguaje que acepta un compilador sistematiza su construcción y permite corregir, quizás más fácilmente, errores de muy difícil localización, como es la ambigüedad en el reconocimiento de ciertas sentencias.

La gramática que acepta el analizador sintáctico es una gramática de contexto libre, puesto que no es fácil comprender gramáticas más complejas ni construir automáticamente autómatas reducidos que reconozcan las sentencias que aceptan, se utiliza para especificar la sintaxis de un lenguaje.

Definición de gramáticas

Una gramática libre de contexto tiene cuatro componentes:

1. Un conjunto de símbolos terminales, a los que algunas veces se les conoce como “tokens”. Los terminales son los símbolos elementales del lenguaje definido por la gramática.
2. Un conjunto de no terminales, a las que algunas veces se les conoce como ‘Variables sintácticas’. Cada no terminal representa un conjunto de cadenas o terminales.
3. Un conjunto de producciones, en donde cada producción consiste en un no terminal, llamada encabezado o lado izquierdo de la producción, una flecha y una secuencia de terminales y no terminales, llamada cuerpo o lado derecho de la producción. La intención intuitiva de una producción es especificar una de las formas escritas de una instrucción; si el no terminal del encabezado representa a una instrucción, entonces el cuerpo representa una forma escrita de la instrucción.
4. Una designación de una de los no terminales como el símbolo inicial.

Convención de notación

✚ Estos símbolos son terminales:

- Las primeras letras minúsculas del alfabeto, como a, b, c.
- Los símbolos de operadores como +, *, etcétera.
- Los símbolos de puntuación como paréntesis, coma, etcétera.
- Los dígitos 0, 1, ..., 9.
- Las cadenas en negrita como **id** o **if**, cada una de las cuales representa un solo símbolo terminal.

✚ Estos símbolos son no terminales:

- Las primeras letras mayúsculas del alfabeto, como A, B, C.
- La letra S que es el símbolo inicial.
- Los nombres en cursiva y minúsculas, como *expr* o *instr*.
- Al hablar sobre las construcciones de programación, las letras mayúsculas pueden utilizarse para representar no terminales. Por ejemplo, los no terminales para las expresiones, los términos y los factores se representan a menudo mediante E, T y F, respectivamente.

Derivaciones

Una regla de producción puede considerarse como equivalente a una regla de reescritura, donde el no terminal de la izquierda es sustituido por la pseudocadena del lado derecho de la producción. Podemos considerar que una pseudocadena es cualquier secuencia de terminales y/o no terminales. Formalmente, se dice que:

$$S \rightarrow \epsilon \text{ o } A \rightarrow \alpha; \quad A \in \Sigma_N, \quad \alpha \in (\Sigma_T \cup \Sigma_N)^+$$

Puede verse que el símbolo no terminal A, puede ser remplazado por la cadena α de terminales y no terminales en cualquier lugar donde aparezca la producción, sin tener en cuenta el contexto en el que se encuentra. Esto hace que aparezcan los siguientes conceptos:

- *Derivación por la izquierda:* es aquella en la que la reescritura se realiza sobre el no terminal más a la izquierda de la pseudocadena de partida.
- *Derivación por la derecha:* es aquella en la que la reescritura se realiza sobre el no terminal más a la derecha de la pseudocadena de partida.

Arboles de análisis sintácticos

Un árbol de análisis sintáctico es una representación gráfica de una derivación que filtra el orden en el que se aplican las producciones para sustituir los no terminales. Cada nodo interior de un árbol de análisis sintáctico representa la aplicación de una producción. El nodo interior se etiqueta con el no terminal A en el encabezado de la producción; los hijos del nodo se etiquetan, de izquierda a derecha, mediante los símbolos en el cuerpo de la producción por la que se sustituyó esta A durante la derivación. Las hojas de un árbol de análisis

sintáctico se etiquetan mediante no terminales o terminales y, leídas de izquierda a derecha, constituyen una forma de frase, a la cual se le llama producto o frontera del árbol.

El generador de analizadores sintácticos: Yacc / Bison

Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción para una gramática independiente del contexto (en realidad de una subclase de éstas, las LALR) en un programa en C que analiza esa gramática. Es compatible al 100% con Yacc, una herramienta clásica de Unix para la generación de analizadores léxicos, pero es un desarrollo diferente realizado por GNU bajo licencia GPL. Todas las gramáticas escritas apropiadamente para Yacc deberían funcionar con Bison sin ningún cambio. Usándolo junto a Flex esta herramienta permite construir compiladores de lenguajes. Bison parte de unos símbolos terminales (tokens), que deben ser generados por un analizador lexicográfico que puede implementarse a través de Flex.

Funcionamiento general

La fuente de Bison se convierte en una función en C llamada **yyparse**. Se llama a la función **yyparse** para hacer que el análisis comience. Esta función lee tokens, ejecuta acciones, y por último retorna cuando se encuentre con el final del fichero o un error de sintaxis del que no puede recuperarse. Usted puede también escribir acciones que ordenen a **yyparse** retornar inmediatamente sin leer más allá. El valor devuelto por **yyparse** es 0 si el análisis tuvo éxito (el retorno se debe al final del fichero). El valor es 1 si el análisis.

La función del analizador léxico, **yyllex**, reconoce tokens desde el flujo de entrada y se los devuelve al analizador sintáctico, **yyparse**. Bison no crea esta función automáticamente; usted debe escribirla de manera que **yyparse** pueda llamarla, **Figura 5**.

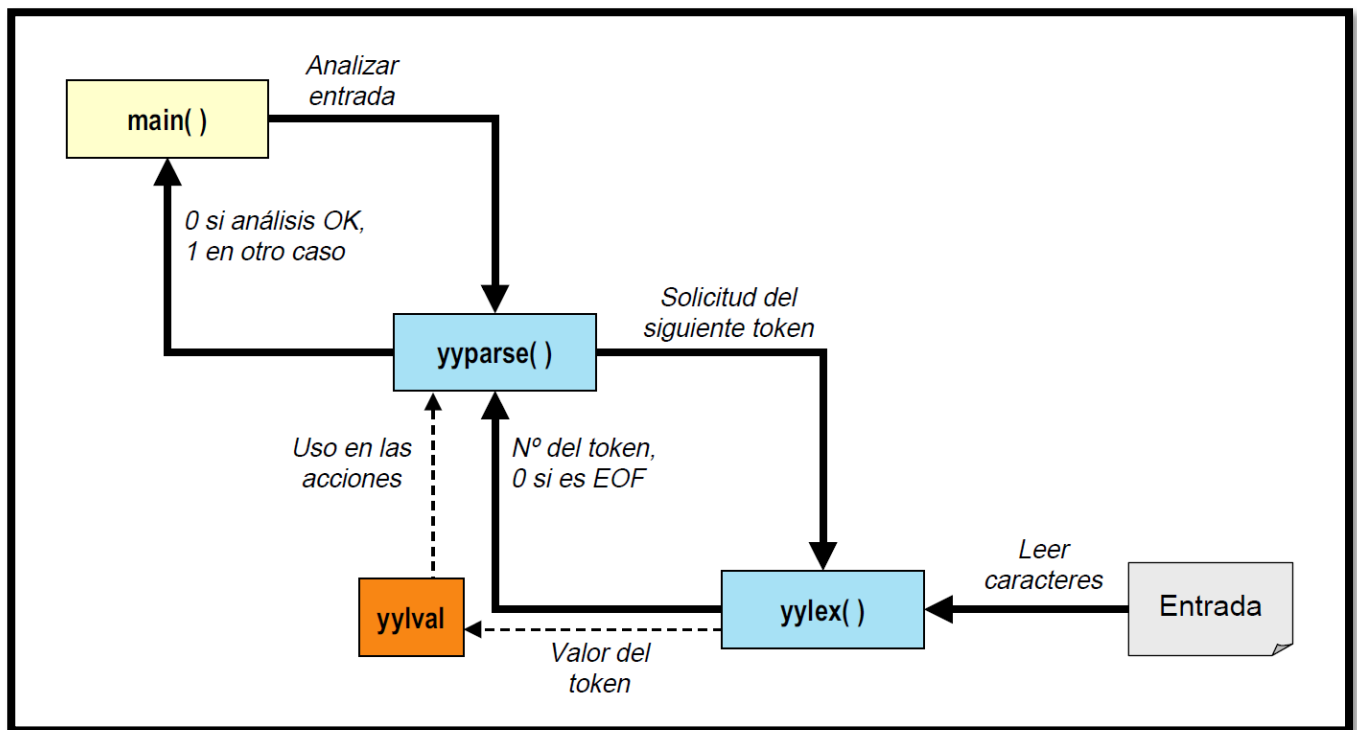


Figura 5 – Flujo de control de las funciones `yylex()` e `yyparse()`.

Formato de un programa Yacc

Un fuente de Bison (normalmente un fichero con extensión `.y`) describe una gramática. El ejecutable que se genera indica si un fichero de entrada dado pertenece o no al lenguaje generado por esa gramática. La forma general de una gramática de Bison es la siguiente:

Area de definiciones

%%

Area de reglas

%%

Area de funciones

Area de definiciones

Este área está compuesta por dos secciones opcionales. La primera de ellas está delimitada por los símbolos `%{` y `%}` y permite codificar declaraciones ordinarias en C. Todo lo que aquí se declare será visible en las áreas de reglas y de funciones, por lo que suele usarse para

crear la tabla de símbolos, o para incluirla (**#include**). Los símbolos **%{ }** deben comenzar obligatoriamente en la columna 0.

La segunda sección permite declarar los componentes léxicos (tokens) usados en la gramática. Para ello se utiliza la cláusula **%token** seguida de una lista de terminales (sin separarlos por comas) habría que declarar:

```
%token TERMINAL
```

Además, a efectos de aumentar la legibilidad, pueden utilizarse tantas cláusulas **%token** como sean necesarias; por convención, los tokens se escriben en **mayúsculas**.

Internamente, Bison codifica cada token como un número entero, empezando desde el 257 (ya que del 0 al 255 se utilizan para representar los caracteres ASCII, y el 256 representa al token de error). Por lo tanto la cláusula

```
%token TERMINAL
```

es equivalente a

```
#define TERMINAL 257
```

Evidentemente, los números o códigos asignados por Bison a cada token son siempre distintos. No obstante lo anterior, no todos los terminales han de enumerarse en cláusulas **%token**. En concreto, los terminales se pueden expresar de dos formas:

- Si su lexema asociado está formado por un solo carácter, y el analizador léxico lo devuelve tal cual (su código ASCII), entonces se pueden indicar directamente en una regla de producción sin declararlo en una cláusula **%token**. Ejemplo: '+'.
- Cualquier otro terminal debe declararse en la parte de declaraciones, de forma que las reglas harán referencia a ellos en base a su nombre (no al código, que tan sólo es una representación interna).

Finalmente, en este área también puede declararse el axioma inicial de la gramática de la forma:

```
%start axioma
```

Si esta cláusula se omite, PCYacc considera como axioma inicial el antecedente de la primera regla gramatical que aparece en el área de reglas.

Declaración %union

Por defecto, la variable **yyval** mediante la cual Lex/Flex le pasa a Yacc/Bison los valores semánticos de los tokens es de tipo "int". Pero habitualmente, los valores semánticos de los tokens son de distintos tipos. Por ejemplo, un token de tipo identificador (ID) tiene un valor semántico de tipo "char*", mientras que un token de tipo constante numérica (NUM) tiene un valor semántico de tipo "int". Con la declaración **%union** se define indirectamente una unión de C con un campo para cada tipo de valor semántico, por ejemplo:

```
%union
{
char* cadena;
int numero;
}
```

Declaración %token

Se utiliza para definir los símbolos no terminales (tokens) de la gramática.

Una forma más completa es:

```
%token <campo_union> NOMBRE_TOKEN
```

Ejemplo:

```
%token <cadena> ID
%token <numero> NUM
```

Desde Lex/Flex, los tokens cualificados se devolverán haciendo, por ejemplo:

```
[A-Z]+ { yyval.identificador = yytext; return ID; }
[0-9]+ { yyval.numero = atoi(yytext); return NUM; }
```

Declaración %type

Se utiliza cuando se ha hecho la declaración **%union** para especificar múltiples tipos de valores.

Permite declarar el tipo de los símbolos no terminales. Lo no terminales a los que no se les asigna ningún valor a través de \$\$ no es necesario declararlos. La declaración es de la forma:

```
%type <campo_union> nombre_no_terminal
```

Por convenio, los nombres de los no terminales se ponen en minúsculas.

Se pueden agrupar varios no terminales en una línea si tienen en mismo tipo.

Área de reglas

Éste es el área más importante ya que en él se especifican las reglas de producción que forman la gramática cuyo lenguaje queremos reconocer. Cada regla gramatical tienen la forma:

```
noTerminal : consecuente ;
```

donde consecuente representa una secuencia de cero o más terminales y no terminales.

Junto con los terminales y no terminales del consecuente, se pueden incluir acciones semánticas en forma de código en C delimitado por los símbolos { y }.

Ejemplo:

```
e      :      e '+' t      {printf("Esto es una suma.\n");};
```

Si hay varias reglas con el mismo antecedente, es conveniente agruparlas indicando éste una sola vez y separando cada consecuente por el símbolo '|', con tan sólo un punto y coma al final del último consecuente.

Área de funciones

La tercera sección de una especificación en Yacc consta de rutinas de apoyo escritas en C. Aquí se debe proporcionar un analizador léxico mediante una función llamada **yylex()**. En caso necesario se pueden agregar otros procedimientos, como rutinas de apoyo o resumen de la compilación. El analizador léxico **yylex()** produce pares formados por un token y su valor de atributo asociado. Si se devuelve un token como por ejemplo NUM, o ID, dicho token debe declararse en la primera sección de la especificación en Yacc, El valor del atributo asociado a un token se comunica al analizador sintáctico mediante la variable **yyval**. En caso de integrar Yacc con Lex, en esta sección habrá de hacer un **#include** del fichero generado por Flex.

En concreto, en esta sección se deben especificar como mínimo las funciones:

- **main()**, que deberá arrancar el analizador sintáctico haciendo una llamada a **yyparse()**.

- `void yyerror(char * s)` que recibe una cadena que describe el error. En este momento la variable `yychar` contiene el terminal que se ha recibido y que no se esperaba.

Tratamiento de errores

Notificación de errores

Quizás una de las mayores deficiencias de Bison es la referente al tratamiento de errores ya que, por defecto, cuando el analizador sintáctico que genera se encuentra con un error sintáctico, sencillamente se para la ejecución con un parco mensaje “syntax error”.

Cuando `yyparse` detecta un error, para notificarlo, invoca la función `yyerror` que debe proporcionar el usuario. La forma más sencilla de la función `yyerror` es:

```
void yyerror(char* msg){ printf(“%s\n”, msg); }
```

Cuando finaliza la función `yyerror`, `yyparse` intenta la recuperación del error (siempre que sea posible, ver apartado de *recuperación de errores*). Y si la recuperación es imposible, `yyparse` termina devolviendo un 1 y se causa una ruptura en la ejecución. **Figura 7.**

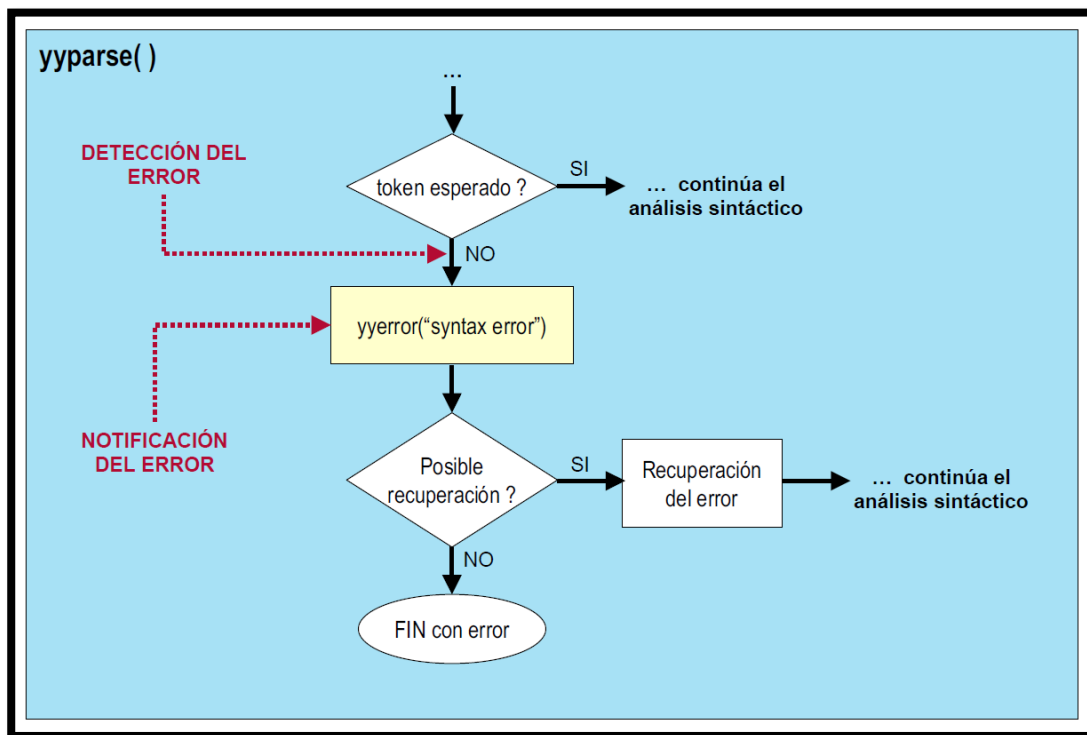


Figura 7 – Flujo de notificación de errores.

Recuperación de errores

Una ruptura de la ejecución puede evitarse mediante el correcto uso del *token* especial **error**. Básicamente, si se produce un error sintáctico el analizador generado por Bison desecha parte de la pila de análisis y parte de la entrada que aún le queda por leer, pero sólo hasta llegar a una situación donde el error se pueda recuperar, lo que se consigue a través del *token* reservado **error**. Un ejemplo típico del uso del token error es el siguiente:

```
program      :    program sentencia ';' | program error ';;';
sentencia    :    ID '=' NUM;
```

La última regla de este trozo de gramática sirve para indicarle a Bison que entre el no terminal **program** y el terminal **';**' puede aparecer un *error sintáctico*.

Supongamos que tenemos la sentencia a reconocer “a = 6; b = 7; **c.= 8**; d = 6;” en la que se ha “colado” un punto tras el identificador **c**; cuando el analizador se encuentre el token **'.'** se da cuenta de que hay un error sintáctico y busca en la gramática todas las reglas que contienen el símbolo error; a continuación va eliminando símbolos de la cima de la pila, hasta que los símbolos ubicados encima de la pila coincidan con los situados a la izquierda del símbolo error en alguna de las reglas en que éste aparece. A continuación va eliminando tokens de la entrada hasta encontrarse con uno que coincida con el que hay justamente a la derecha en la regla de error seleccionada. Una vez hecho esto, inserta el token **error** en la pila, desplaza y continúa con el análisis intentando reducir por esa regla de error. Al terminal que hay justo a la derecha de error en una regla de error se le denomina token de seguridad, ya que permite al analizador llegar a una condición segura en el análisis. Como token de seguridad suele escogerse el de *terminación de sentencia* o de *declaración*, como en nuestro caso, en el que hemos escogido el punto y coma.

Por último, para que todo funcione de forma adecuada es conveniente asociarle a la regla de error la invocación de una macro especial de Bison denominada **yerrorok**.

La ejecución de esta macro informa al analizador sintáctico de que la recuperación se ha realizado satisfactoriamente. Si no se invoca, el analizador generado por Bison entra en un estado llamado “condición de seguridad”, y permanecerá en ese estado hasta haber desplazado tres tokens. La condición de seguridad previene al analizador de emitir más mensajes de error (ante nuevos errores sintácticos), y tiene por objetivo evitar errores en

cascada debidos a una incorrecta recuperación del error. La invocación a **yyerrok** saca al analizador de la condición de seguridad, de manera que éste nos informará de todos, absolutamente todos, los errores sintácticos, incluso los debidos a una incorrecta recuperación de errores.

Errores léxicos y sintácticos

```
%%                                /* Opción 1 */
[0 - 9]+                          { return NUM; }
[A- Z][A - Z0 -9]*               { return ID; }
[ \t\n]                           { ; }
.                                  { return yytext[0]; }
/* Error sintáctico. Le pasa el error al analizador sintáctico */
```

y

```
%%                                /* Opción 2 */
[0 - 9]+                          { return NUM; }
[A- Z][A - Z0 -9]*               { return ID; }
[ \t\n]                           { ; }
‘(‘|’)|’*’|’+’                   { return yytext[0]; }
/* El analizador sintáctico sólo recibe tokens válidos */
.                                  { printf ("Error léxico.\n"); }
```

Estas opciones se comportan de manera diferente ante los errores léxicos, como por ejemplo cuando el usuario teclea una ‘@’ donde se esperaba un ‘+’ o cualquier otro operador. En la Opción 1, el analizador sintáctico recibe el código ASCII de la ‘@’ como token, debido al patrón .; en este caso el analizador se encuentra, por tanto, ante un token inesperado, lo que le lleva a abortar el análisis emitiendo un mensaje de error sintáctico. En la Opción 2 el analizador léxico protege al sintáctico encargándose él mismo de emitir los mensajes de error ante los lexemas inválidos; en este caso los mensajes de error se consideran lexicográficos.

En el desarrollo del analizador por la *opción 1*, es el método del mínimo esfuerzo y concentra la gestión de errores en el **analizador sintáctico**. Por la *opción 2*, la gestión de los errores se maneja separado, pudiendo personalizar los mensajes de error en la compilación.

Por último, nótese que el programa Lex no incluye la función **main()**, ya que Yacc proporciona un modelo dirigido por sintaxis y es al analizador sintáctico al que se debe ceder el control principal, y no al léxico. Por tanto, será el área de funciones del programa Yacc quien incorpore en el **main()** una llamada al analizador sintáctico, a través de la función **yyparse()**:

Compilación y ejecución de un programa Bison

Al ejecutar el comando **yacc nombre_fuente.y** se crea un fichero en C llamado **nombre_fuente.tab.c**. Por compatibilidad con Yacc, existe la opción **-y** que fuerza a que el archivo de salida se llame **y.tab.c**. A partir de ahora supondremos que se usa siempre esta opción para facilitar la escritura de este documento. Otra opción útil es la opción **-d**, que genera el archivo con las definiciones de tokens que necesita Flex (si se ha usado la opción **-y**, el archivo se llama **y.tab.h**). Este archivo **y.tab.h** normalmente se incluye en la sección de declaraciones del fuente de Flex. El fichero **y.tab.c** se puede compilar con la instrucción **gcc y.tab.c**, **Figura 6**.

Los pasos para usar conjuntamente Flex y Bison serán normalmente:

```
yacc -yd fuente.y
flex fuente.l
gcc y.tab.c lex.yy.c -o salida
```

Estos pasos generan un ejecutable llamado **salida** que nos permite comprobar qué palabras pertenecen al lenguaje generado por la gramática descrita en el fichero Bison.

Notas finales

Flex requiere un formato bastante estricto de su fichero de entrada. En particular los caracteres no visibles (espacios en blanco, tabuladores, saltos de línea) fuera de sitio causan errores difíciles de encontrar. Sobre todo es muy importante no dejar líneas en blanco de más ni empezar reglas con espacios en blanco o tabuladores.

Cuando una gramática resulta “demasiado ambigua” Bison dará errores. Definir cuando una gramática es “demasiado ambigua” queda fuera del alcance de esta introducción. Una posible solución a algunos tipos de ambigüedad es el uso adecuado de las declaraciones de precedencia de operadores. Si surgen otros conflictos, es útil invocar a Bison con la opción `-v`, lo que generará un archivo de salida `y.output` (si se usa la opción `-y`) donde se puede mirar qué reglas son las problemáticas (aquellas donde aparezca la palabra `conflict`). Hay algunas pistas más sobre esto en el manual de Bison.

Finalmente tened en cuenta algunas cosas:

- No usar el mismo nombre para no terminales y variables que defináis en el código C
- Bison distingue mayúsculas de minúsculas en los nombres de variables y tokens
- Bison no lee directamente la entrada, es necesario que exista una función `yylex()` que lea la entrada y la transforme en tokens (por ejemplo una como la que se crea usando Flex).

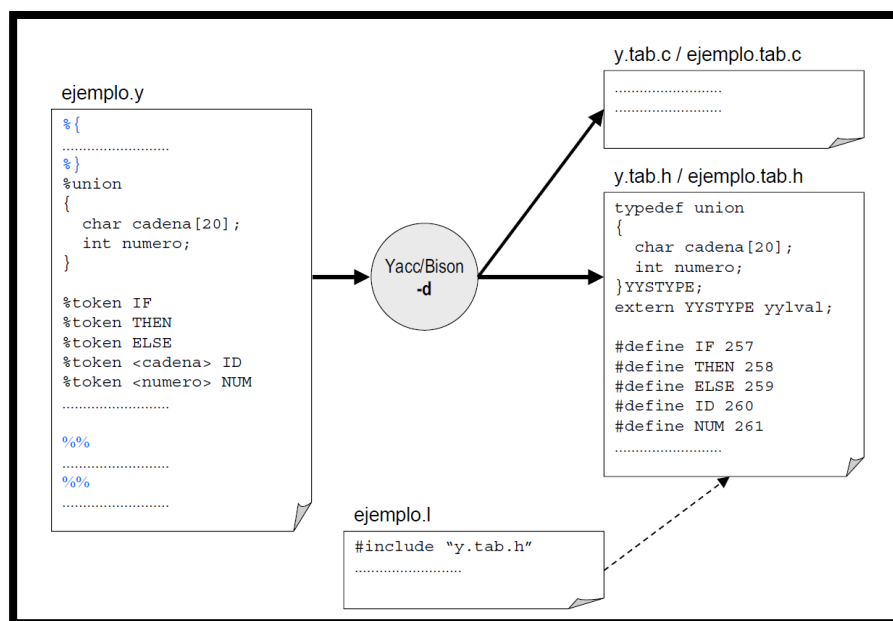


Figura 6 - Ejemplo de fichero `y.tab.h`.

Requerimientos de Sistema

Flex y Bison:

Son dos herramientas útiles para crear programas que reaccionen a una entrada de datos con una estructura y un lenguaje predeterminado.

Descarga e instalación

Es posible descargar las versiones de Flex y Bison como un conjunto o bien por separado

Flex: <http://gnuwin32.sourceforge.net/packages/flex.htm>

Bison: <http://gnuwin32.sourceforge.net/packages/bison.htm>

Para poder utilizar los paquetes de Flex y de Bison se necesita MS Windows 2000 o una versión posterior.

Ambas herramientas no requieren de muchos recursos de sistema para poder ser instaladas y ejecutadas.

Microsoft Visual Studio

Es un entorno de desarrollo integrado para sistemas operativos Windows. Soporta múltiples lenguajes de programación.

Los requerimientos para poder utilizar esta herramienta se enlistan en la siguiente tabla:

Sistemas Operativos admitidos	Windows 10: Home, Professional, Education y Enterprise (LTSC y S no se admiten) Windows Server 2016: Standard y Datacenter Windows 8.1: Core, Professional y Enterprise Windows Server 2012 R2: Essentials, Standard y Datacenter Windows 7 SP1 (con las actualizaciones más recientes de Windows): Home, Premium, Professional, Enterprise y Ultimate
-------------------------------	--

Procesador	Procesador de 1.8 GHz o superior Doble núcleo o superior recomendado
RAM	2 GB de RAM 4 GB de RAM recomendado
Disco Duro	Se necesitan entre 20 y 50 GB de espacio en disco para una instalación típica 130 GB para una instalación completa
Tarjeta de Video	Tarjeta de video que admita una resolución de pantalla mínima de 720p (1280 x 720) Visual Studio funcionará mejor con una resolución de WXGA (1366 x 768) o superior.

Herramientas utilizadas para el Desarrollo.

Para la elaboración de la aplicación se utilizaron los siguientes programas:

- Para el analizador léxico y sintáctico:
 - Flex for Windows versión 2.5.4a
- Para la interfaz gráfica de usuario:
 - Visual Studio Community 2017 version 15.6

Diseño del Archivo Lex

```

Lexico.l                               Sintactico.y
1  /*
2      ANALIZADOR LEXICO
3
4      AUTORES:      - JOSE MANUEL MIRANDA VILLAGRAN
5                    - BRYAN CASTILLO MARIN
6                    - LUIS ENRIQUE ESCAMILLA ALVAREZ
7
8      MATERIA:      LENGUAJES Y AUTOMATAS I
9
10     PROFERORA:     OFELIA GUTIERREZ GUIRALDI
11
12  */
13
14  %{
15      #include <stdio.h>
16      #include "Sintactico.tab.h"
17      int flag;
18      int noerror = 0;
19  %}
20
21  /* VARIABLES DE ER -----*/

```

Figura 8 – Sección de definiciones de FLEX. Código en C

En la sección de definiciones de Lex se hace la inclusión en código C de las librerías que se utilizarán, así como también el fichero *Sintactico.tab.h* el cual es un header que contiene los tokens con su respectivo tipo y valor que son necesarios para la comunicación entre Lex y Yacc

La declaración de las variables globales *flag* y *noerror* ambas de tipo *int* las cuales se utilizaran para el control de errores del compilador. *Flag* es la bandera que se utiliza para indicar que ha ocurrido un error léxico mientras que *noerror* se utiliza para indicar el numero de errores que han ocurrido.

```

Lexico.l                               Sintactico.y
20
21  /* VARIABLES DE ER -----*/
22
23  VCONSTANTE      [0-9]+
24  VREAL           [0-9]*\.[0-9]+
25  VLOGICO          "TRUE"|"FALSE"
26  VIDENTIFICADOR  [a-zA-Z]+[a-zA-Z0-9]*
27  VCADENA          \'^[^']*\'
28  ESPACIO          [" "\t\n]
29
30
31  /* OPCIONES DE FLEX -----*/
32
33  %option noyywrap
34  %option yylineno
35  %x COMENTARIO
36  %x COMENTERIOLINEA
37
38  %%

```

Figura 9 – Sección de definiciones de FLEX. Expresiones regulares generales y declaración de las opciones de FLEX

Dentro de la misma sección se crean expresiones regulares las cuales describen números enteros (*VCONSTANTE*), números con decimales (*VREAL*), valores lógicos (*VLOGICO*), identificadores (*VIDENTIFICADOR*), el manejo de cadenas (*VCADENA*) y el manejo de los espacios en blanco (*ESPACIO*).

Se hace uso de las opciones de Lex *yylineno* y *noyywrap*.

Las expresiones *COMENTARIO* y *COMENTARIOLINEA* serán necesarias para el control de los comentarios y los saltos de línea.

En la sección de reglas se hará uso de las expresiones anteriormente declaradas en la sección de definiciones. Para un mejor control de esta sección se clasificaron las expresiones regulares con base en su “uso” (comentarios, palabras reservadas ,operaciones, etc.).

Lexico.l	Sintactico.y
37	
38	%%
39	
40	/* ELIMINACION DE COMENTARIOS Y ESPACIOS -----*/
41	
42	{ESPACIO}* { ; }
43	
44	"#" {BEGIN(COMENTERIOLINEA); }
45	<COMENTERIOLINEA>[\n] {BEGIN(INITIAL); }
46	<COMENTERIOLINEA>[]+
47	<COMENTERIOLINEA>[\t]+
48	<COMENTERIOLINEA>.
49	
50	"#*" {BEGIN(COMENTARIO); }
51	<COMENTARIO>"#*" {BEGIN(INITIAL); }
52	<COMENTARIO>[\n]
53	<COMENTARIO>[]+
54	<COMENTARIO>[\t]+
55	<COMENTARIO>.
56	
57	
58	/* PALABRAS RESERVADAS -----*/
59	

Figura 10 – Sección de reglas, eliminación de comentarios y espacios

El símbolo que representa el inicio de un comentario es '#', con las diferentes expresiones descritas en la imagen anterior cualquier cadena de caracteres que se encuentre después de '#' no será tomada en cuenta a la hora de iniciar tanto el análisis léxico como el sintáctico.

En el caso de los espacios en blanco se pueden escribir tantos como sea posible y estos no afectaran al análisis.

Lexico.l	Sintactico.y
58	/* PALABRAS RESERVADAS -----*/
59	
60	"program" { return PROGRAM; }
61	"print" { return PRINT; }
62	"integer" { return TIPO_INTEGER; }
63	"logical" { return TIPO_LOGICAL; }
64	"string" { return TIPO_STRING; }
65	"numeric" { return TIPO_NUMERIC; }
66	"for" { return FOR; }
67	"while" { return WHILE; }
68	"do" { return DO_IF; }
69	"if" { return IF; }
70	"then" { return THEN; }
71	"else" { return ELSE; }
72	"scan" { return SCAN; }
73	"return" { return RETURN; }
74	"void" { return VOID; }
75	"import" { return IMPORT; }
76	"switch" { return SWITCH; }
77	"case" { return CASE; }
78	"break" { return BREAK; }
79	
80	
81	/* CABECERAS -----*/
82	
83	"math.gg" { return MATH; }
84	"io.gg" { return IO; }
85	

Figura 11 – Palabras reservadas y cabeceras

Definición de los lexemas que representan las palabras reservadas del lenguaje y las cabeceras del mismo.

Lexico.l	Sintactico.y
86	/* OPERADORES -----*/
87	
88	
89	"+" { return '+'; }
90	"-" { return '-'; }
91	"*" { return '*'; }
92	"/" { return '/'; }
93	"^" { return '^'; }
94	"<-" { return ASIGNA; }
95	"++" { return INCREMENTO; }
96	"--" { return DECREMENTO; }
97	"mod" { return MOD; }
98	"\$" { return '\$'; }
99	"<" { return '<'; }
100	">" { return '>'; }
101	">=" { return MAYOR_IGUAL; }
102	"<=" { return MENOR_IGUAL; }
103	"!!" { return DIFERENTE; }
104	"==" { return IGUAL; }
105	"&&" { return AND; }
106	" " { return OR; }
107	"!" { return '!'; }
108	"%s" { return ESPECIFICADOR_STRING; }
109	"%d" { return ESPECIFICADOR_INTEGER; }
110	"%l" { return ESPECIFICADOR_LOGICAL; }
111	"%n" { return ESPECIFICADOR_NUMERIC; }
112	
113	
114	/* DELIMITADORES -----*/
115	

Figura 12 – Operadores lógicos y matemáticos

Símbolos que definen las diferentes operaciones de tipo lógico y matemático, así como también los especificadores de los diferentes tipos de datos aceptados por el lenguaje (string, integer, logical y numeric).

```

112 Lexico.l
113 Sintactico.y
114 /* DELIMITADORES -----*/
115
116 "("      { return '('; }
117 ")"      { return ')'; }
118 "{"      { return '{'; }
119 "}"      { return '}'; }
120 ";"      { return ';'; }
121 ","      { return ','; }
122 ":"      { return ':'; }
123
124
125 /* VALORES -----*/
126
127 {VREAL}   { return VALOR_NUMERIC; }
128 {VCONSTANTE} { return VALOR_INTEGER; }
129 {VLOGICO}  { return VALOR_LOGICAL; }
130 {VCADENA}  { return VALOR_STRING; }
131 {VIDENTIFICADOR} { return IDENTIFICADOR; }
132
133
134 /* CONTADOR DE LINEAS Y ERRORES -----*/
135
136 \n        { yylineno++; }
137 .         { printf( "<lexico error> line: %d - unrecognized token '%s'\n", yylineno, yytext ); flag = 1; noerror++; }
138 <<EOF>>   { return 0; }
139
140 %%
141

```

Figura 13 – Delimitadores, tipos de datos y operaciones para el control de errores léxicos y numero de líneas

Expresiones que definen los delimitantes de funciones, parámetros y final de instrucciones; los tipos de datos del lenguaje.

También se definen las expresiones que llevan el control del número de líneas; el carácter especial '.' utilizado para mandar los mensajes de error. Notese que si ha ocurrido algún error la bandera 'se enciende' (se asigna con el valor 1) lo cual indicara que existe, por lo menos, un error. Además incrementará el número de errores totales ocurridos.

Se hace uso de los caracteres '<<EOF>>' para el final del fichero.

La sección de funciones no está definida en este caso debido a que algunas de las funcionalidades ya han sido definidas dentro de las expresiones regulares.

Diseño del Archivo Yacc

```

Sintactico.y
6      - LUIS ENRIQUE ESCAMILLA ALVAREZ
7
8      MATERIA:   LENGUAJES Y AUTOMATAS I
9
10     PROFERORA:  OFELIA GUTIERREZ GUIRALDI
11
12
13     RECOMENDACIONES PARA DECLARAR EL TOKEN ERROR EN LAS PRODUCCIONES DE LA GRAMATICA:
14
15     - Cuando hay una produccion recursiva es candidata a poner el token error, ej. instrucciones := instruccion instrucciones;
16       en el lugar de instruccion poner token el error; instrucciones := error instrucciones;
17     - En medio de los delimitadores como {}, ().
18     - Antes del terminal de fin de linea como ;
19     - En medio de la produccion en donde se crea que sea necesario
20 */
21
22 %{
23     #include <stdio.h>
24     #include <stdlib.h>
25     #include <string.h>
26
27     /*VARIABLES*/
28     extern char *yytext;
29     extern FILE *yyin;
30     extern int yylineno;
31     extern int noerror;
32     extern int flag;
33 }%
34
35 /* PALABRAS RESERVADAS DEL LENGUAJE */

```

Figura 14 –
Sección de
definiciones
de Yacc. Código
en C

En el área de definiciones en la primera sección se incluyen las librerías y se utilizan las variables declaradas en el archivo de lex mediante la palabra reservada de C ‘extern’.

Del mismo modo se declaran las variables propias de Lex *yytext* y *yyin*.

```

Sintactico.y
-- --
34
35 /* PALABRAS RESERVADAS DEL LENGUAJE */
36 %token PROGRAM PRINT TIPO_INTEGER TIPO_LOGICAL TIPO_STRING TIPO_NUMERIC FOR WHILE DO_IF
37 %token IF THEN ELSE SCAN RETURN VOID IMPORT SWITCH CASE BREAK
38
39 /* OPERADORES */
40 %token ASIGNA INCREMENTO DECREMENTO MOD MAYOR_IGUAL MENOR_IGUAL DIFERENTE IGUAL AND OR
41
42 /* VALORES */
43 %token VALOR_LOGICAL VALOR_STRING VALOR_NUMERIC VALOR_INTEGER IDENTIFICADOR
44
45 /* ESPECIFICADORES */
46 %token ESPECIFICADOR_STRING ESPECIFICADOR_NUMERIC ESPECIFICADOR_LOGICAL ESPECIFICADOR_INTEGER
47
48 /* CABECERAS */
49
50 %token MATH IO
51
52 %start axioma
53
54 %%%

```

Figura 15 –
Sección de
definiciones de
Yacc. Tokens
terminales
Declaración del
axioma inicial.

En la segunda sección se declaran los tokens terminales. Los tokens declarados en esta parte son aquellos que en la parte léxica devuelven algo distinto que el propio lexema. Por ejemplo:

“+”	{ return '+'; }	...Para este caso no se declara un token terminal
“mod”	{ return MOD; }	...Se declara un token terminal equivalente al valor devuelto

Dentro de esta misma sección se declara el axioma inicial de la gramática.

```

Sintactico.y
54 %%
55 axioma      : cabeceras main funciones;
56
57 /* CABECERAS */
58
59 cabeceras   : cabecera cabeceras | cabecera;
60 cabecera    : IMPORT '<' libreria '>';
61 libreria    : MATH | IO;
62 cabecera    : error '>' { yyerrok; };
63
64
65
66 /* METODO MAIN */
67
68 main        : VOID PROGRAM '(' parametros ')' '{' instrucciones '}';
69 main        : tipo PROGRAM '(' parametros ')' '{' instrucciones RETURN valor ';' '}';
70 main        : VOID error '{' { yyerrok; };
71 main        : tipo error '{' { yyerrok; };
72
73
74
75 /* FUNCIONES */
76
77 funciones   : funcion funciones | ;
78 funcion     : VOID IDENTIFICADOR '(' parametros ')' '{' instrucciones '}';
79 funcion     : tipo IDENTIFICADOR '(' parametros ')' '{' instrucciones RETURN valor ';' '}';
80 funcion     : VOID error '{' { yyerrok; };
81 funcion     : tipo error '{' { yyerrok; };
82 funcion     : error '(' error ')' '{' error '}';
83

```

Figura 16 –
Sección de
reglas de Yacc.
Producciones de:
axioma
,cabeceras
cabecera, main,
funcion y
funciones

Dentro de el área de reglas se define la estructura general del lenguaje comenzando por el axioma que inicializa la gramática.

Para una mejor comprensión de la gramática ésta se clasifica y ordenada de tal manera que para cada no terminal se definen sus producciones con sus respectivos terminales y no terminales y, adicionalmente, las producciones para el manejo de errores haciendo uso del token especial *error*.

```

Sintactico.y
86  /* PARAMETROS DE FUNCIONES */
87
88  parametros      : parametro | ;
89  parametro       : tipo IDENTIFICADOR ',' parametro | tipo IDENTIFICADOR;
90  parametro       : error parametro { yyerrok; };
91  parametro       : error IDENTIFICADOR { yyerrok; };
92
93
94
95  /* ARGUMENTOS */
96
97  argumentosFuncion : argumentoFuncion | ;
98  argumentoFuncion  : valor ',' argumentoFuncion | valor;
99  argumentoFuncion  : error argumentoFuncion { yyerrok; };
100
101  argumentosImprimir : argumentoImprimir | ;
102  argumentoImprimir  : valor '$' argumentoImprimir | valor;
103  argumentoImprimir  : error argumentoImprimir { yyerrok; };
104
105  argumentoFor      : variableControl condiciones ';' actualizacion;
106  argumentoFor      : variableControl condiciones ';';
107  argumentoFor      : variableControl ';' actualizacion;
108  argumentoFor      : ';' condiciones ';' actualizacion;
109  argumentoFor      : variableControl ';';
110  argumentoFor      : ';' condiciones ';';
111  argumentoFor      : error ';' error ';' error { yyerrok; };
112
113  actualizacion     : IDENTIFICADOR INCREMENTO | IDENTIFICADOR DECREMENTO;

```

Figura 17 – Producciones de: *parámetros,parámetro, argumentosFuncion, argumentoFucion, argumentosImprimir, argumentoImprimir, argumentosFor, argumentoFor y actualizacion*

Definición de la sintaxis para los parámetros de las funciones y sus argumentos (los argumentos de las funciones, los argumentos de la impresión de datos y los argumentos de la estructura for).

```

Sintactico.y
115  /* INSTRUCCIONES */
116
117  instrucciones    : instruccion instrucciones | ;
118  instruccion      : declaracion | asignacion | llamaFuncion | imprimir | leer | control | incremento | decremento;
119
120
121  /* DECLARACIONES */
122
123
124  declaracion      : tipo varios ';';
125  varios           : IDENTIFICADOR ',' varios | IDENTIFICADOR ;
126  varios           : IDENTIFICADOR ASIGNA valor ',' varios | IDENTIFICADOR ASIGNA valor;
127  declaracion      : tipo error ';' { yyerrok; };
128
129
130
131  /* ASIGNACIONES */
132
133  asignacion       : IDENTIFICADOR ASIGNA valor ';';
134  asignacion       : error ';' { yyerrok; };
135
136
137
138  /* LLAMADA DE FUNCIONES */
139
140  llamaFuncion     : IDENTIFICADOR '(' argumentosFuncion ')' ';';
141  llamaFuncion     : IDENTIFICADOR error ';' { yyerrok; };
142  llamaFuncion     : IDENTIFICADOR '(' error ')' ';' { yyerrok; };
143

```

Figura 18 – Producciones de: *instrucciones, instruccion, delcaracion, varios, asignacion y llamaFuncion*

Sintaxis para las instrucciones posibles a realizar, cómo declarar una variable y hacer asignaciones de valores así como las llamadas a funciones.

```

Sintactico.y
145
146  /* ENTRADA Y SALIDA */
147
148  imprimir      : PRINT '(' argumentosImprimir ')' ';' ;
149  leer          : SCAN '(' especificador ',' IDENTIFICADOR ')' ';' ;
150
151  imprimir      : PRINT '(' error ')' ';' { yyerrok; };
152  leer          : SCAN '(' error ')' ';' { yyerrok; };
153  imprimir      : PRINT error ';' { yyerrok; };
154  leer          : SCAN error ';' { yyerrok; };
155
156
157
158  /* CASOS SWITCH */
159
160  casos          : caso casos | caso;
161  caso           : CASE valorSwitch ':' instrucciones BREAK ';' ;
162  valorSwitch    : VALOR_INTEGER | VALOR_STRING;
163
164  caso           : error ':' error ';' { yyerrok; };
165
166
167
168  /* ESTRUCTURAS DE CONTROL */
169

```

Figura 19 –
Producciones de:
imprimir, leer,
casos, caso y
valorSwitch

Sintaxis para la entrada y salida de datos y los casos de la estructura de control 'switch'

```

Sintactico.y
166
167
168  /* ESTRUCTURAS DE CONTROL */
169
170  control        : if | if-else | do-if | for | while | switch;
171
172  if             : IF '(' condiciones ')' '{' instrucciones '}' ;
173  if-else        : IF '(' condiciones ')' '{' instrucciones '}' ELSE '{' instrucciones '}' ;
174  do-if          : DO_IF '{' instrucciones '}' IF '(' condiciones ')' ';' ;
175  for            : FOR '(' argumentoFor ')' '{' instrucciones '}' ;
176  while          : WHILE '(' condiciones ')' '{' instrucciones '}' ;
177  switch         : SWITCH '(' condiciones ')' '{' casos '}' ;
178
179
180  if             : IF '(' error ')' '{' error '}' { yyerrok; };
181  if-else        : IF '(' error ')' '{' error '}' ELSE '{' error '}' { yyerrok; };
182  do-if          : DO_IF '{' error '}' IF '(' error ')' ';' { yyerrok; };
183  for            : FOR '(' error ')' '{' error '}' { yyerrok; };
184  while          : WHILE '(' error ')' '{' error '}' { yyerrok; };
185  switch         : SWITCH '(' error ')' '{' error '}' { yyerrok; };
186
187  if             : IF error '{' error '}' { yyerrok; };
188  if-else        : IF error ELSE error { yyerrok; };
189  do-if          : DO_IF error ';' { yyerrok; };
190  for            : FOR error '{' error '}' { yyerrok; };
191  while          : WHILE error '{' error '}' { yyerrok; };
192  switch         : SWITCH error '{' error '}' { yyerrok; };
193
194
195  /* INCREMENTO y DECREMENTO */

```

Figura 20 –
Producciones de:
control, if, if-else,
do-if, for, while y
switch

Producciones de las estructuras de control.


```

Sintactico.y
194
195  /* INCREMENTO y DECREMENTO */
196
197  incremento      : IDENTIFICADOR INCREMENTO ';' ;
198  decremento      : IDENTIFICADOR DECREMENTO ';' ;
199
200
201
202  /* OPERACIONES CON JERARQUIA */
203
204  condiciones      : condiciones logico q | q ;
205  q                : q relacional w | w ;
206  w                : '(' condiciones ')' | valor1 | '!' valor1 ;
207  w                : '(' error ')' { yyerrok; } ;
208
209
210  aritmetica        : aritmetica '+' termino | aritmetica '-' termino | termino ;
211  termino           : termino '*' factor | termino '/' factor | factor ;
212  factor            : '(' aritmetica ')' | '(' aritmetica ')' '^' VALOR_INTEGER | VALOR_NUMERIC | VALOR_INTEGER | IDENTIFICADOR ;
213  factor            : '(' error ')' { yyerrok; } ;
214
215
216
217  /* OTRAS*/

```

Figura 21 – Producciones de incremento, decremento, condiciones, q, w, aritmética, termino y factor.

Producciones de las instrucciones incremento y decremento. Así como también las operaciones lógicas y matemáticas con estructura jerárquica.

```

Sintactico.y
---
217  /* OTRAS*/
218
219
220  valor1           : VALOR_STRING | VALOR_LOGICAL | VALOR_NUMERIC | VALOR_INTEGER | IDENTIFICADOR ;
221  valor1           : IDENTIFICADOR '(' argumentosFuncion ')' ;
222  valor1           : IDENTIFICADOR error ')' { yyerrok; } ;
223
224  variableControl  : declaracion | asignacion ;
225
226  valor            : VALOR_STRING | VALOR_LOGICAL | aritmetica ;
227  valor            : IDENTIFICADOR '(' argumentosFuncion ')' ;
228
229  tipo             : TIPO_STRING | TIPO_LOGICAL | TIPO_INTEGER | TIPO_NUMERIC ;
230  especificador     : ESPECIFICADOR_INTEGER | ESPECIFICADOR_LOGICAL | ESPECIFICADOR_NUMERIC | ESPECIFICADOR_STRING ;
231  relacional        : MAYOR_IGUAL | MENOR_IGUAL | IGUAL | DIFERENTE | '<' | '>' ;
232  logico           : AND | OR ;
233
234  %%
235

```

Figura 22 – Producciones de valor1, variableControl, valor, tipo, especificador, relacional y logico

Producciones varias que complementan algunas de las reglas establecidas anteriormente.

```

Sintactico.y
233
234 %%
235
236 yyerror(char* mensaje){
237     flag = 1;
238     noerror++;
239
240     if( strlen( yytext ) != 0)
241         printf( "<%= line: %d - unexpected '%s' token.\n", mensaje, yylineno, yytext );
242     else
243         printf( "<%= line: %d - expected valid token before end line.\n", mensaje, yylineno );
244 }
245
246 int main( int argc, char const *argv[] )
247 {
248     flag = 0;
249     yyin = fopen( argv[ 1 ], "r" );
250
251     yyparse();
252
253     switch(flag){
254     case 0: puts("SUCCESSFUL COMPILATION.\n");
255             break;
256     case 1: printf( "errors: %d\n", noerror );
257             puts( "FAILED COMPILATION." );
258             break;
259     }
260     return 0;
261 }

```

Figura 23 –
Sección de
funciones de Yacc
Codigo en C

Dentro del área de funciones se establece la función de error de Yacc *yyerror*. La cual ‘enciende’ la bandera, incrementa la variable del número de errores e imprime un mensaje de error sintáctico.

En esta sección se encuentra la función *main* la cual inicializa la bandera en 0, se inicializa la variable de Lex *yyin* a la cual se le asigna un fichero de entrada (un argumento enviado desde la aplicación que se detallará más adelante) que solamente será leído; además llama a la función *yyparse()* que inicia con el análisis de los lexemas.

Finalmente si se he encontrado con algun error tanto de tipo léxico como sintáctico se evalua la bandera e imprime los distintos mensajes de error si la bandera se encendió. En caso contrario se dice que el análisis fue exitoso.

Diseño de Lenguaje

Nombre del lenguaje:

MJU

Objetivo del lenguaje:

Lenguaje diseñado para el paradigma de programación estructurada, enfocada para el desarrollo de pequeños proyectos así como también fácil uso de procedimientos y funciones.

Estructura general del lenguaje

Sección	Descripción
<code>import<io.gg></code>	Algunas librerías que se utilizan para definir algunas funciones del lenguaje
<code>void program{</code>	Inicio del programa
<code>integer var1 <- 5 logical var3 <- true numeric var2 <- 3.5 string var4 <- "Hola"</code>	Declaracion de variables
<code>print("Hola mundo") print(var1) print(var1 * 2)</code>	Instrucciones para imprimir
<code>while(var1 < 0){ #INSTRUCCIONES }</code>	Estructura ciclica
<code>if(var1 == var2){ #INSTRUCCIONES }</code>	Estructura de control
<code>Void funcion(){ #Instrucciones } Void funcion (integer a){ #Instrucciones } String funcion(){ #Instrucciones Return String cadena; }</code>	Manejo de funciones. Las funciones de tipo VOID no devuelven ningún parámetro. En cambio las funciones de algún tipo de dato, deben de volver un valor igual que el valor especificado en la función.

Comentarios

En MJU los comentarios empiezan a partir del caracter '#'.

```
#Este es un comentario
```

Palabras reservadas

Las palabras reservadas estarán escritas con minúscula

Declaración de variables

Las variables son unidades básicas de almacenamiento. Una variable se define por la combinación de un dominio, identificador y una inicialización opcional. Para inicializarla se deberá usar el operador “assignar”, el cual se denota como una flecha con el signo menos y el símbolo “<”.

Los dominios que maneja MJU son integer, string, logical y numeric. Un identificador es el nombre de la variable. Es una secuencia ilimitada de caracteres alfabéticos o dígitos en mayúscula o en minúscula, siempre deberán comenzar con un carácter alfabético y en minúscula. Además no deberán contener espacios.

```
#Ejemplos de declaraciones de variables
```

```
integer var1;
```

```
numeric var2 <- 5.2;
```

Operaciones aritméticas

Para poder realizarse, toda la expresión deberá ser del mismo dominio. El orden de los cálculos depende del orden de prioridad de los operadores: primero los operadores unarios, después los multiplicativos, de izquierda a derecha, después los operadores aditivos, de izquierda a derecha, después los operadores de relación y por último los operadores de asignación.

```
integer var1 <- 5;
```

```
integer var2 <- 6 / 3 + 8 / 4;
```

```
integer var3 <- var1 + var2;
```

```
numeric var4 <- 3 / 6;
```

Operación con cadenas

La operación que maneja MJU es la concatenación. Para concatenar dos cadenas se usara el símbolo \$ entre las dos cadenas o entre las cadenas que se quieran concatenar. La cadena debe estar entre comillas simples “”, todo lo que esté dentro de las comillas deberá conservarse, incluyendo espacio.

```
string var1 <- ‘Hola’;  
string var2 <- ‘ Mundo.’;  
string var3 <- var1$var2;  
var3 $ ‘ Concatenacion’;  
#La salida será: “Hola Mundo. Concatenación”
```

Operaciones lógicas

En las operaciones lógicas deberan usarse con operadores relacionales, el resultado puede ser “TRUE” o “FALSE”. Para hacer uso de los operadores lógicos and, or y neg, sus operandos deberán ser expresiones lógicas con operadores relacionales.

```
integer var1 <- 3;  
integer var2 <- 5;  
logical var3 <- var1 > var2;  
logical var4 <- var1 < var2 && var1 != var2;    # != diferente de  
#El resultado de var3 es “FALSE”  
#El resultado de var4 es “FALSE”
```

Impresión en pantalla

La palabra reservada print seguida de un paréntesis que abre “(” se usa para imprimir en pantalla todo aquello que prosiga al paréntesis hasta encontrar un paréntesis que cierra “)”; puede contener operaciones artmeticas o con cadena.

```
print(var1);  
print(‘Hola mundo.’ $ var1);  
print(var2 + var3);
```

Lectura de datos

También es posible introducir datos directamente desde el teclado usando la palabra reservada `scan`. Los datos a introducir deben estar limitados por paréntesis. Dentro de los paréntesis se debe definir primero el tipo de dato que se va a introducir mediante un especificador (`%s`, `%d`, `%l`, `%n`) y separado por una coma el identificador que guardara la entrada de datos.

```
scan( # especificador, identificador);  
integer x;  
scan(%d, x );
```

Estructuras cíclicas

Los bucles, iteraciones o sentencias repetitivas modifican el flujo secuencial de un programa permitiendo la ejecución reiterada de una sentencia o sentencias. En MJU existen tres tipos de estructuras cíclicas `for`, `do-if` y `while`.

Sentencia for

La estructura de repetición `for` repite el bloque de sentencias mientras la condición del `for` es verdadera. Un `for` es un caso particular de la estructura `while`. Solo se debe utilizar cuando se sabe el número de veces que se debe repetir el bloque de sentencias. Debera constar de una variable de control inicializada, seguido de una condición de continuación de ciclo y al final una alteración de la variable de control. Es necesario aclarar que el ultimo parámetro del ciclo `for`, no debe ir forzosamente una alteración de la variable de control, ese espacio es la ultima instrucción que ejecuta el ciclo `for` antes de verificar la condición de continuación.

También puede no tener algún parámetro incluso ninguno.

```
for(integer var1 <- 10; var1 >0; var1--){  
  #Sentencias  
}
```

Sentencia do-if

La estructura de repetición do-if ejecuta el bloque de sentencias al menos una vez. Después comprueba la condición y repite el bloque de sentencias mientras la condición es verdadera.

```
do{  
    #Sentencias  
}if( var1 < var2 )
```

Sentencia while

La estructura de repetición while repite el bloque de sentencias mientras la condición del while es verdadera.

```
while(var1 < var2){  
    #Sentencias  
}
```

Estructuras de control

Estructura if

La estructura if se denomina estructura de selección única porque ejecuta un bloque de sentencias solo cuando se cumple la condición del if. Si la condición es verdadera se ejecuta el bloque de sentencias. Si la condición es falsa, el flujo del programa continúa en la sentencia inmediatamente posterior al if.

```
if (var < 1){  
    #Sentencias  
}
```


Estructura if-else

Se denomina de selección doble porque selecciona entre dos bloques de sentencias mutuamente excluyentes. Si se cumple la condición, se ejecuta el bloque de sentencias asociado al if. Si la condición no se cumple, entonces se ejecuta el bloque de sentencias asociado al else.

```
if( var1 > 10 ){  
    #Sentencias  
}else{  
    #Sentencias  
}
```

Estructura switch

La estructura switch es una estructura de selección múltiple que permite seleccionar un bloque de sentencias entre varios casos. En cierto modo, es parecido a una estructura de if-then-else anidados. La diferencia está en que la selección del bloque de sentencias depende de la evaluación de una expresión que se compara por igualdad con cada uno de los casos. La estructura switch consta de una expresión y una serie de etiquetas case y una opción default. La sentencia break indica el final de la ejecución del switch.

```
switch(var1){  
    case 1:  
        #Sentencias  
        break  
    case 2:  
        #Sentencias  
        break  
    default:  
        #Sentencias  
        break  
}
```

Palabras Reservadas:

Valor	Tipo	Descripción
program	PROGRAM	Palabra reservada para indicar el inicio de un programa.
print	PRINT	Palabra reservada para imprimir en pantalla.
scan	SCAN	Lectura de datos desde teclado.
integer	TIPO_INTEGER	Tipo de dato que contiene valores enteros.
logical	TIPO_LOGICAL	Tipo de dato que puede contener dos valores, ya sea verdadero o falso.
string	TIPO_STRING	Tipo de caracter que acepta cadenas de datos alfanuméricos.
numeric	TIPO_NUMERIC	Tipo de dato para números reales.
for	FOR	Forma parte de las sintaxis del bucle FOR.
while	WHILE	Ciclo que primero evalúa una condición y posteriormente realiza las instrucciones.
do	DO_IF	Ciclo que se utiliza para repetir la ejecución de sentencias y se ejecuta al menos una vez.
if	IF	Se utiliza para saber si se cumple con una condición.
else	ELSE	Segunda cláusula de la condición IF, realiza las instrucciones si la condición no se cumple.
switch	SWITCH	Estructura que evalúa un valor y realiza una instrucción dependiendo si coincide o no con alguno de sus "casos".
case	CASE	Realiza una instrucción para el caso de coincidencia.

break	BREAK	Indica la salida de uno de los casos de la estructura switch y sale de ella.
return	RETURN	Señala el valor a devolver al final de una función que no sea del tipo void
void	VOID	Se utiliza para declarar funciones las cuales no devuelven ningún parámetro
import	IMPORT	Sentencia utilizada para llamar librerías dentro del programa

Operadores Aritméticos:

Valor	Tipo	Descripción
+	SUM	Operador de suma.
-	RES	Operador de resta.
*	MULT	Operador de multiplicación.
/	DIV	Operador de división.
<=	ASIGNACION	Operador que asigna un valor.
++	INCREMENTO	Operador que sirve para incrementar en 1 un valor.
--	DECREMENTO	Operador para decrementar en 1 un valor.
mod	MOD	Operador que devuelve el módulo de una división.

Operadores Relacionales:

Valor	Tipo	Descripción
<	MENOR	Es un operando lógico que funciona para comparar dos valores y determinar el menor entre ellos.
>	MAYOR	Es un operando lógico que funciona para comparar dos valores y determinar el mayor entre ellos.
>=	MAYOR_IGUAL	Es un operando lógico que determina aquellos valores mayores o iguales a un valor dado.
<=	MENOR_IGUAL	Es un operando lógico que determina aquellos valores menores o iguales a un valor dado.
!!	DIFERENTE	Verifica si un valor es diferente de otro.
==	IGUAL	Funciona para verificar si un valor es igual a otro.

Operadores Lógicos:

Valor	Tipo	Descripción
&&	AND	Conjunción entre operandos.
	OR	Disyunción entre operandos.
!	!	Negación

Especificadores:

Valor	Tipo	Descripción
%n	ESPECIFICADOR_NUMERIC	Define el tipo de dato numeric
%d	ESPECIFICADOR_INTEGER	Define el tipo de dato entero
%s	ESPECIFICADOR_STRING	Define el tipo de datos para las cadenas
%l	ESPECIFICADOR_LOGICAL	Define el tipo de dato logico

Caracteres especiales:

Valor	Tipo	Descripción
((Denota el inicio de los parámetros de un método así como también para jerarquizar operaciones aritméticas y lógicas.
))	Denota el fin de los parámetros de un método así como también para jerarquizar operaciones aritméticas y lógicas.
{	{	Indica el inicio de un método.
}	}	Tiene la función de marcar el fin de un método.
;	;	Se usa en el ciclo for para separar los operandos. También para definir el final de una instrucción
'	'	Se utilizan para delimitar cadenas
\$	\$	Se usa para concatenar cadenas
,	,	Permite declarar varias variables al mismo tiempo si se separan mediante comas.
:	:	Dentro de la sentencia SWITCH, inicia las instrucciones de los diferentes casos posibles.

Ejemplos

#Programa que genera la tabla del 5

```
import <io.gg>

void program(){

    print('Tabla del numero 5');

    for( integer i <- 1; i <= 10; i++ ){
        print( i $ ' x 5 = ' $ i * 5 );
    }

}
```

#Programa que obtiene la raiz n-esima de un numero

```
import <io.gg>
import <math.gg>

void program(){

    integer n, indice;
    print('Raiz de un numero.\n');

    print('Dame un numero: ');
    scan(%d, n);

    print('Dame el indice de la raiz: ');
    scan(%d, indice);

    print('La raiz del numero es: '$ raiz(n, indice));

}

numeric raiz(integer n, integer indice){

    raiz <- pow(n, 1/indice);
    return raiz;

}
```

Diseño Interfaz Gráfica

Para realizar la interfaz de usuario se apoyo en el diseñador de formas de Microsoft Visual Studio.

Se utilizaron diversos objetos para un mejor control en la manipulación de los métodos y para hacer mas amigable la interfaz. Los objetos generales son los siguientes:

Objeto	Nombre	Descripción
MenuStrip	MenuStrip1	Crea una barra de menú
OpenFileDialog	abrirVentana	Despliega una ventana para poder abrir un archivo
SaveFileDialog	guardarVentana	Despliega una ventana para guardar un archivo
FontDialog	fontDialog1	Ventana emergente para cambiar el formato de texto
ColorDialog	colorDialog1	Ventana emergente para cambiar el color de texto
Timer	lineRefresh	Actualiza objetos cada determinado lapso de tiempo
RichTextBox	codeText	Recuadro para texto. Este se utilizará para escribir el código a analizar.
RichTexBox	outResult	Recuadro para texto. Aquí se mostrará la salida después del análisis
ToolStripLabel	etiquetaNomArch	Una etiqueta donde se mostrará el nombre del archivo abierto.
PictureBox	lineCode	Un área para dibujar. Aquí se mostrarán los números de línea.

Tambien se definen algunos botones (ToolStripButton) y elementos de menú (ToolStripMenuItem) que se detallarán mas adelante

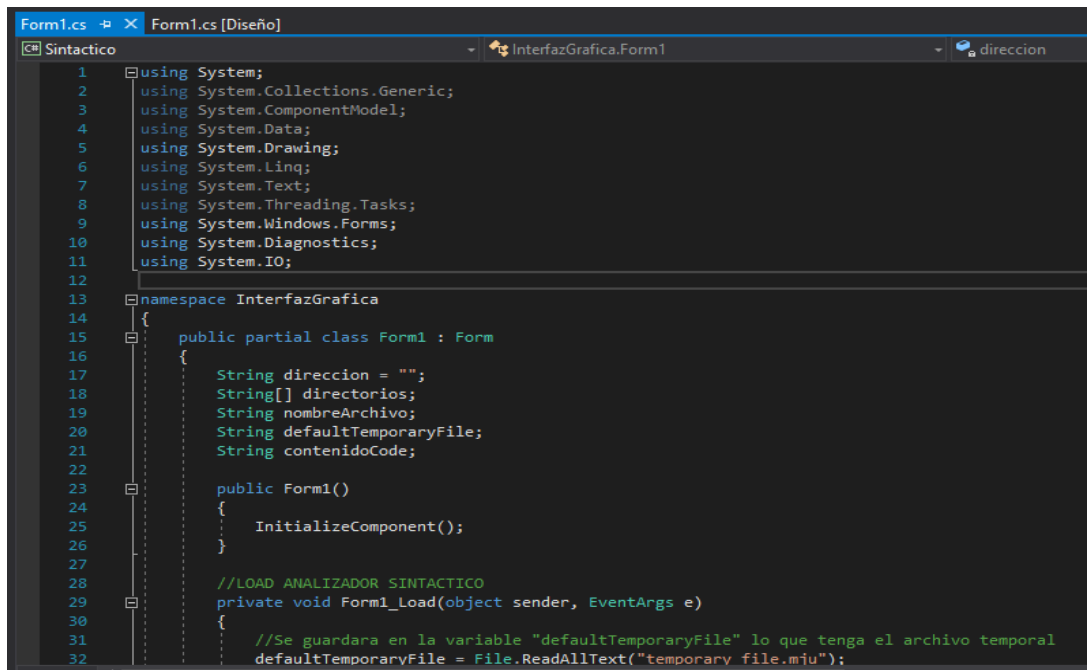


Figura 24 –
*librerías
necesarias para
hacer funcionar el
programa.
Declaración de
variables globales.*

Se declaran variables globales para tener control sobre los archivos que se estarán modificando mientras se hace uso de la aplicación.

Uso de las variable globales:

- String direccion: inicializada como '""', se utiliza para guardar la ruta del archivo que el usuario abra o guarde. Las funciones que modifican o utilizan esta variable son:

guardarExistente()

guardarComo()

abrirArchivo()

- String[] directorios: sin inicializar. Se utiliza para dividir la ruta que esta almacenada en la variable direccion. Su único propósito es el de obtener el nombre del archivo actualmente abierto. Funciones que modifican o utilizan esta variable:

String getNameFile(String direccion)

- String nombreArchivo: sin inicializar. Utilizada para almacenar el nombre del archivo con el que se esta trabajando en ese momento. Funciones que modifican o utilizan esta variable:

guardarComo()

abrirArchivo()

- String defaultTemporaryFile: sin inicializar. Variable que almacena el texto escrito dentro de un archivo plantilla. De nombre “temporary_file.mju”. Esta plantilla es leída por esta variable para mostrar su contenido dentro del área de código (un objeto richTextBox nombrado codeText) al iniciar la aplicación. Funciones que modifican o utilizan esta variable:

Form1_Load()

Form1_FormClosed()

- String contenidoCode: sin inicializar. Esta variable se utiliza para almacenar todo el texto escrito dentro del área de código codeText. Funciones que modifican o utilizan esta variable:

Form1_FormClosed()

guardarExistente()

guardarComo()

abrirArchivo()

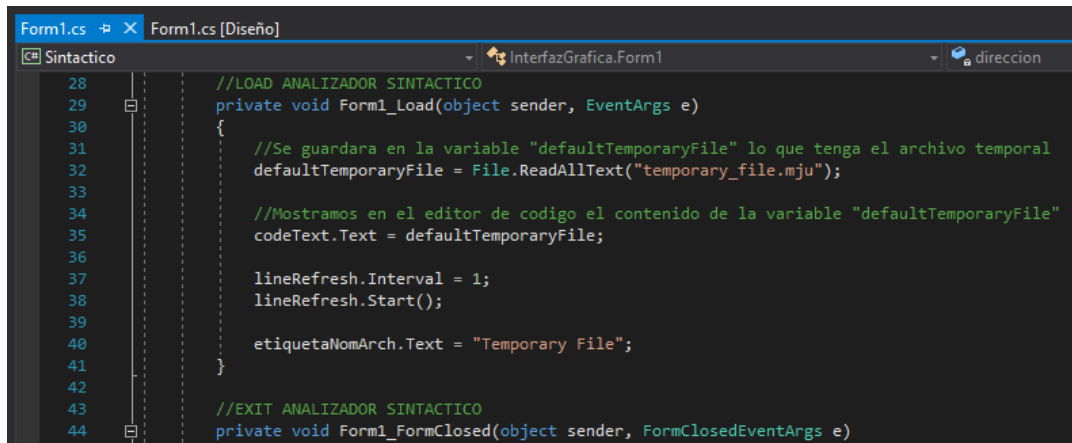
Las distintas operaciones sobre los archivos que se pueden realizar son:

- Abrir
- Guardar
- Guardar como
- Cerrar (Al salir de la aplicación)

La extensión que utilizan los archivos dentro de la interfaz es .mju

Inicio de la aplicación

Form_1Load()



```
Form1.cs [Diseño]
Sintactico
InterfazGrafica.Form1
direccion

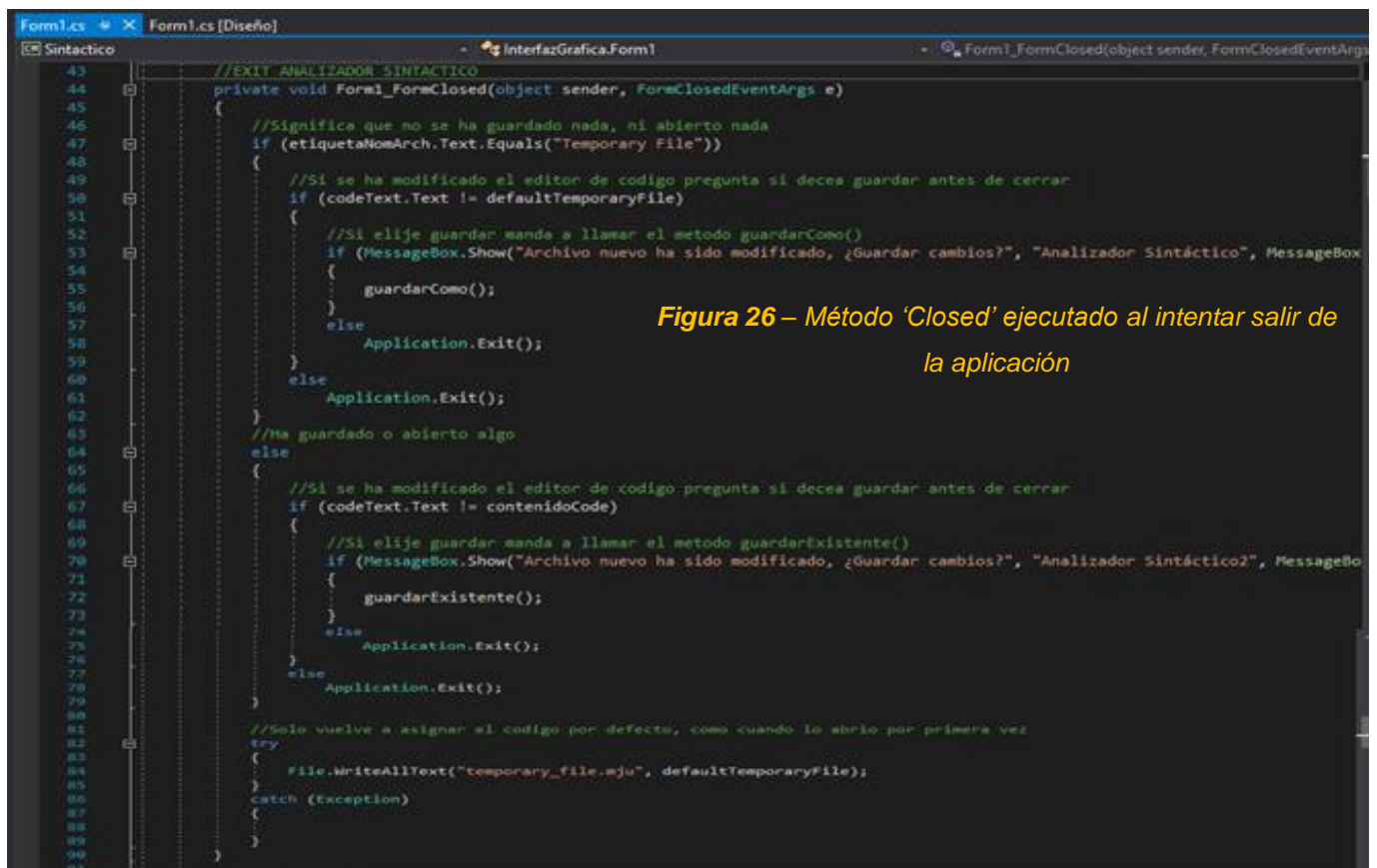
28 //LOAD ANALIZADOR SINTACTICO
29 private void Form1_Load(object sender, EventArgs e)
30 {
31     //Se guardara en la variable "defaultTemporaryFile" lo que tenga el archivo temporal
32     defaultTemporaryFile = File.ReadAllText("temporary_file.mju");
33
34     //Mostramos en el editor de codigo el contenido de la variable "defaultTemporaryFile"
35     codeText.Text = defaultTemporaryFile;
36
37     lineRefresh.Interval = 1;
38     lineRefresh.Start();
39
40     etiquetaNomArch.Text = "Temporary File";
41 }
42
43 //EXIT ANALIZADOR SINTACTICO
44 private void Form1_FormClosed(object sender, FormClosedEventArgs e)
```

Figura 25 –
Método 'Load'
ejecutado
automáticamente
al iniciar la
aplicación

Al iniciar la aplicación se inicializa la variable *defaultTemporaryFile* con el archivo “*temporary_file.mju*” el cual contiene una plantilla del lenguaje. Dicha plantilla es leída por el cuadro de texto (*codeText*) y mostrada en pantalla. Además una etiqueta (*etiquetaNomArch*), la cual contiene el nombre del archivo actual, comienza con “*Temporary File*”.

Cerrar la aplicación

Form1_FormClosed()



```
Form1.cs [Diseño]
Sintactico
InterfazGrafica.Form1
Form1_FormClosed(object sender, FormClosedEventArgs e)

43 //EXIT ANALIZADOR SINTACTICO
44 private void Form1_FormClosed(object sender, FormClosedEventArgs e)
45 {
46     //Significa que no se ha guardado nada, ni abierto nada
47     if (etiquetaNomArch.Text.Equals("Temporary File"))
48     {
49         //Si se ha modificado el editor de codigo pregunta si desea guardar antes de cerrar
50         if (codeText.Text != defaultTemporaryFile)
51         {
52             //Si elije guardar manda a llamar el metodo guardarComo()
53             if (MessageBox.Show("Archivo nuevo ha sido modificado, ¿Guardar cambios?", "Analizador Sintáctico", MessageBoxButtons.YesNo) == DialogResult.Yes)
54             {
55                 guardarComo();
56             }
57             else
58                 Application.Exit();
59         }
60         else
61             Application.Exit();
62     }
63     //Ha guardado o abierto algo
64     else
65     {
66         //Si se ha modificado el editor de codigo pregunta si desea guardar antes de cerrar
67         if (codeText.Text != contenidoCode)
68         {
69             //Si elije guardar manda a llamar el metodo guardarExistente()
70             if (MessageBox.Show("Archivo nuevo ha sido modificado, ¿Guardar cambios?", "Analizador Sintáctico2", MessageBoxButtons.YesNo) == DialogResult.Yes)
71             {
72                 guardarExistente();
73             }
74             else
75                 Application.Exit();
76         }
77         else
78             Application.Exit();
79     }
80
81     //Solo vuelve a asignar el codigo por defecto, como cuando lo abrio por primera vez
82     try
83     {
84         File.WriteAllText("temporary_file.mju", defaultTemporaryFile);
85     }
86     catch (Exception)
87     {
88     }
89 }
90
91
```

Figura 26 – Método 'Closed' ejecutado al intentar salir de la aplicación

Al salir de la aplicación primero se verifica que el nombre del archivo, dentro de la etiqueta del nombre del archivo, sea igual al nombre de la etiqueta original

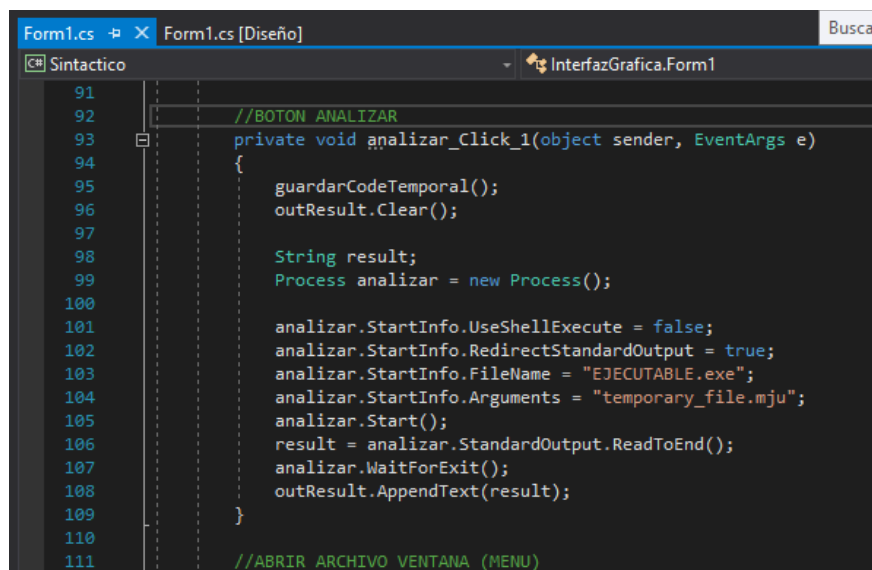
Posteriormente evalúa que el texto escrito dentro del cuadro de texto (codeText) sea igual que el texto dentro de la plantilla. En ese caso se abre un cuadro de diálogo el cual preguntara si se desea o no guardar el archivo, si se da una respuesta afirmativa se llama a la función guardarComo(). En caso contrario sale de la aplicación inmediatamente. Si los textos comparados anteriormente son exactamente iguales cierra la aplicación sin mostrar el cuadro de diálogo.

Si los nombres de las etiquetas son diferentes indica que se ha abierto un archivo o que se ha guardado al menos una vez. Se comparan los contenidos dentro de la variable contenidoCode y el objeto codeText. Si estos son diferentes abre un cuadro de dialogo que preguntará si se desean guardar cambios, si se da una respuesta afirmativa manda a llamar a la función guardarExistente(), si no sale de la aplicación. Si los campos comparados con anterioridad son iguales sale de la aplicación.

Finalmente se crea nuevamente la plantilla inicial, con sus valores por defecto haciendo uso de la variable *defaultTemporaryfile*.

Analizar

analizar_Click_1()



```
Form1.cs [Diseño]
Sintactico
InterfazGrafica.Form1
91
92 //BOTON ANALIZAR
93 private void analizar_Click_1(object sender, EventArgs e)
94 {
95     guardarCodeTemporal();
96     outResult.Clear();
97
98     String result;
99     Process analizar = new Process();
100
101     analizar.StartInfo.UseShellExecute = false;
102     analizar.StartInfo.RedirectStandardOutput = true;
103     analizar.StartInfo.FileName = "EJECUTABLE.exe";
104     analizar.StartInfo.Arguments = "temporary_file.mju";
105     analizar.Start();
106     result = analizar.StandardOutput.ReadToEnd();
107     analizar.WaitForExit();
108     outResult.AppendText(result);
109 }
110
111 //ABRIR ARCHIVO VENTANA (MENU)
```

Figura 27 –
Método ‘Analizar’
ejecutado tras
oprimir el botón
‘analizar’

Al iniciar el proceso de análisis en la aplicación, se llama a la función `guardarCodeTemporal()`, y se limpia la pantalla de salida (esta pantalla es un objeto `richTextBox` nombrada `outResult`). Posteriormente se declara una variable local de tipo `String` y se crea un objeto de la clase `Process` (analizar) el cual será el encargado de establecer la comunicación entre la aplicación y el fichero ejecutable obtenido tras realizar las acciones necesarias de Lex y Yacc.

Una vez instanciada la clase `Process` se modifican algunas de sus propiedades:

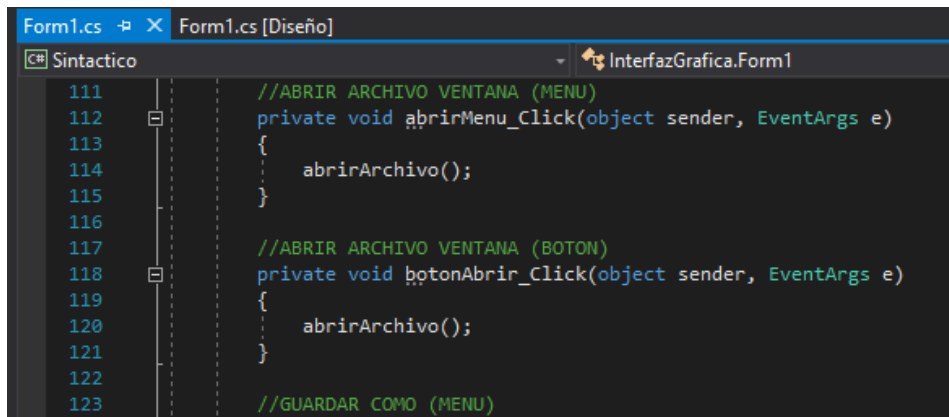
- `UseShellExecute`: Se deshabilita el uso del Shell del sistema operativo, al establecer esto lo que se pretende es dejar el inicio del proceso directamente al archivo ejecutable.
- `RedirectStandardOutput`: Al habilitar esta propiedad se espera salida de texto de la aplicación que se va a ejecutar.
- `FileName`: Nombre de la aplicación que se va a iniciar a la cual podemos pasarle, o no, argumentos de entrada. El nombre de la aplicación que se va a ejecutar es "EJECUTABLE.EXE"
- `Arguments`: Son los argumentos que se van a pasar a la aplicación especificada en `FileName`. "temporary_file.mju" es el archivo necesario para que la aplicación comience a trabajar.
- `Start()`: inicia el recurso del proceso especificado en la propiedad `StartInfo` de la clase `Process`

Cuando el proceso se ejecuta la variable local *result* guarda la salida de caracteres de la aplicación hasta su última línea. Esto es realizado por el método `ReadToEnd()`. Se espera a que el proceso termine mediante `WaitForExit()` y finalmente se muestra el resultado del proceso por medio del recuadro `outResult`.

Abrir un archivo

`abrirMenu_Click()`

`botónAbrir_Click()`



```

Form1.cs [Diseño]
C# Sintactico InterfazGrafica.Form1
111 //ABRIR ARCHIVO VENTANA (MENU)
112 private void abrirMenu_Click(object sender, EventArgs e)
113 {
114     abrirArchivo();
115 }
116
117 //ABRIR ARCHIVO VENTANA (BOTON)
118 private void botonAbrir_Click(object sender, EventArgs e)
119 {
120     abrirArchivo();
121 }
122
123 //GUARDAR COMO (MENU)

```

Figura 28 –
Eventos ‘Abrir’
producidos por
botones y eventos
de menú

Estos métodos son accedidos al seleccionar las opciones de ‘Abrir’ dentro del menú Archivo y el botón ‘abrir’ en la barra de herramientas. Ambas opciones invocan el método `abrirArchivo()`.

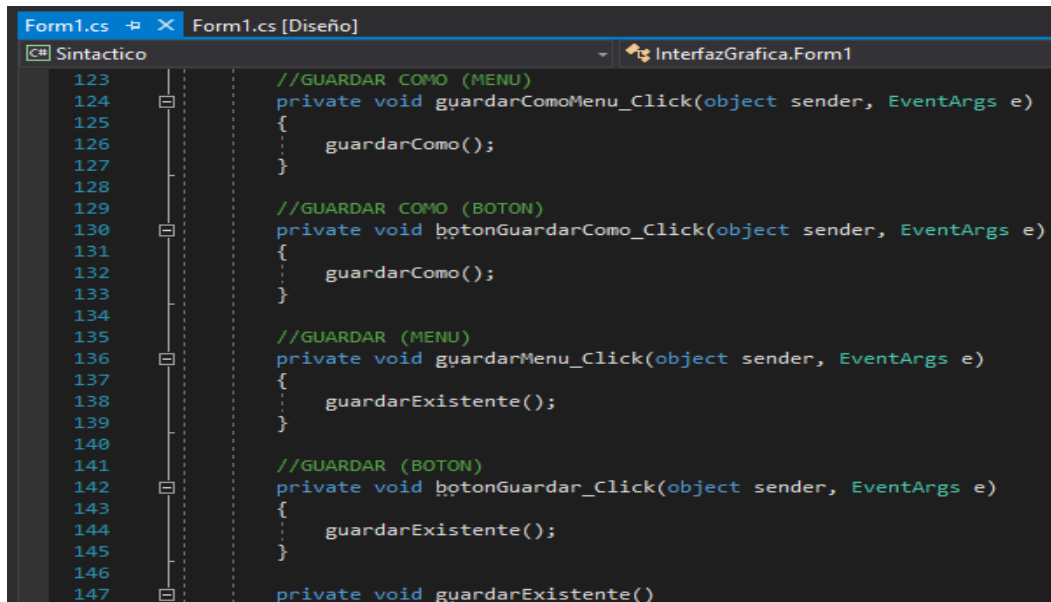
Guardar y Guardar Como

`guardarComoMenu_Click()`

`botonGuardarComo_Click()`

`guardarMenu_Click()`

`botonGuardar_Click()`



```

Form1.cs [Diseño]
C# Sintactico InterfazGrafica.Form1
123
124 //GUARDAR COMO (MENU)
125 private void guardarComoMenu_Click(object sender, EventArgs e)
126 {
127     guardarComo();
128 }
129
130 //GUARDAR COMO (BOTON)
131 private void botonGuardarComo_Click(object sender, EventArgs e)
132 {
133     guardarComo();
134 }
135
136 //GUARDAR (MENU)
137 private void guardarMenu_Click(object sender, EventArgs e)
138 {
139     guardarExistente();
140 }
141
142 //GUARDAR (BOTON)
143 private void botonGuardar_Click(object sender, EventArgs e)
144 {
145     guardarExistente();
146 }
147 private void guardarExistente()

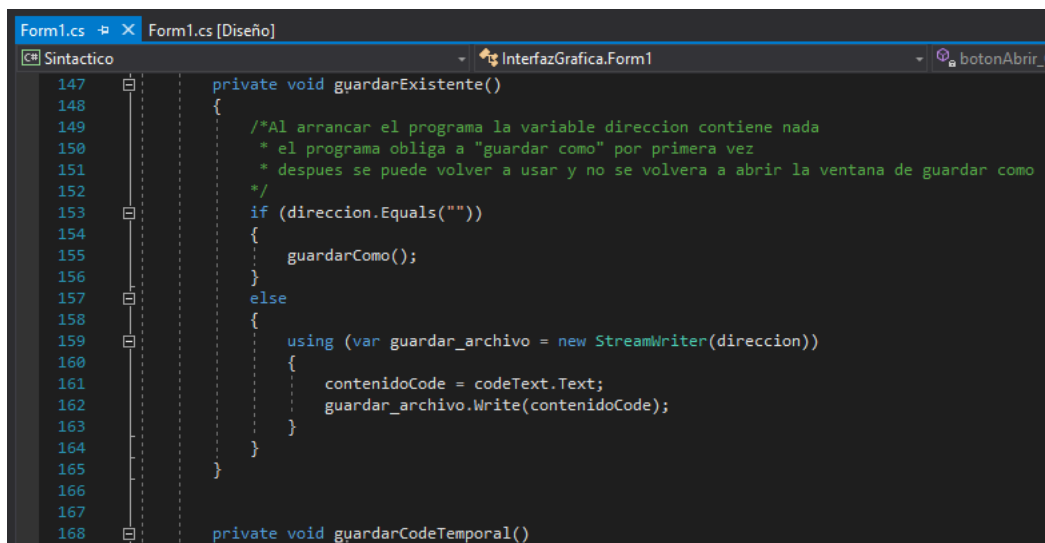
```

Figura 29 –
Eventos ‘Guardar’
y ‘Guardar Como’
producidos por
botones y
elementos de
menú

Los primeros dos métodos invocan `guardarComo()`, estos se ejecutan dentro del menú Archivo opción 'Guardar Como' y al oprimir el ícono 'Guardar Como' en la barra de herramientas.

Los otros dos métodos invocan `guardarExistente()`, similar que las opciones anteriores, estos métodos se ejecutan al seleccionar 'Guardar' dentro del menú Archivo y al seleccionar 'Guardar' en la barra de herramientas.

Método `guardarExistente()`



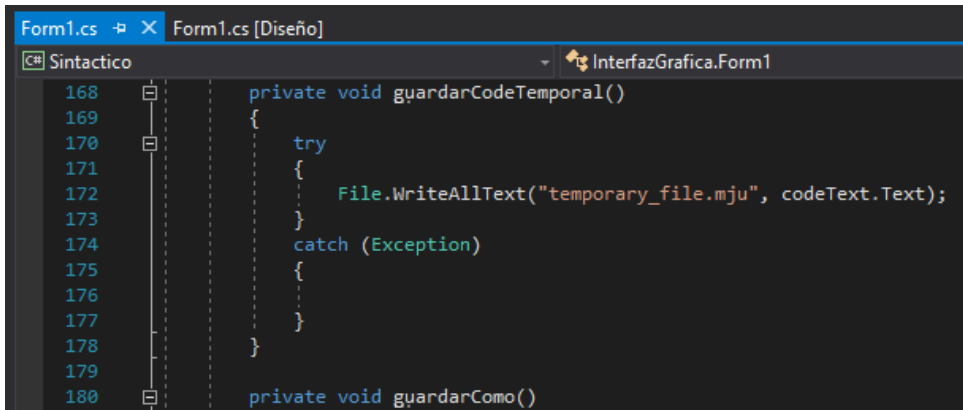
```
Form1.cs [Diseño]
Sintactico
InterfazGrafica.Form1
botonAbrir_C

147 private void guardarExistente()
148 {
149     /*Al arrancar el programa la variable direccion contiene nada
150     * el programa obliga a "guardar como" por primera vez
151     * despues se puede volver a usar y no se volvera a abrir la ventana de guardar como
152     */
153     if (direccion.Equals(""))
154     {
155         guardarComo();
156     }
157     else
158     {
159         using (var guardar_archivo = new StreamWriter(direccion))
160         {
161             contenidoCode = codeText.Text;
162             guardar_archivo.Write(contenidoCode);
163         }
164     }
165 }
166
167 private void guardarCodeTemporal()
168
```

Figura 30 –
Método
'`guardarExistente`'
ejecutado tras
intentar guardar un
archivo

Si al iniciar el programa no se abre un archivo y se esta trabajando con el archivo predeterminado la variable `direccion` contiene "" si esto es así al intentar guardar (directamente con esta opción y no Guardar Como) el programa obligará al usuario a "guardar como" en la primera ocasión; sin embargo si se abre un archivo o ya se había guardado con anterioridad simplemente se sobrescribira el archivo existente con ayuda de un objeto `StreamWriter` y la dirección en la cual esta ubicado el archivo. La variable `direccion` se actualiza cada vez que se ejecuten los métodos `abrirArchivo()` y `guardarComo()` (estos métodos se analizaran posteriormente).

Método guardarCodeTemporal()



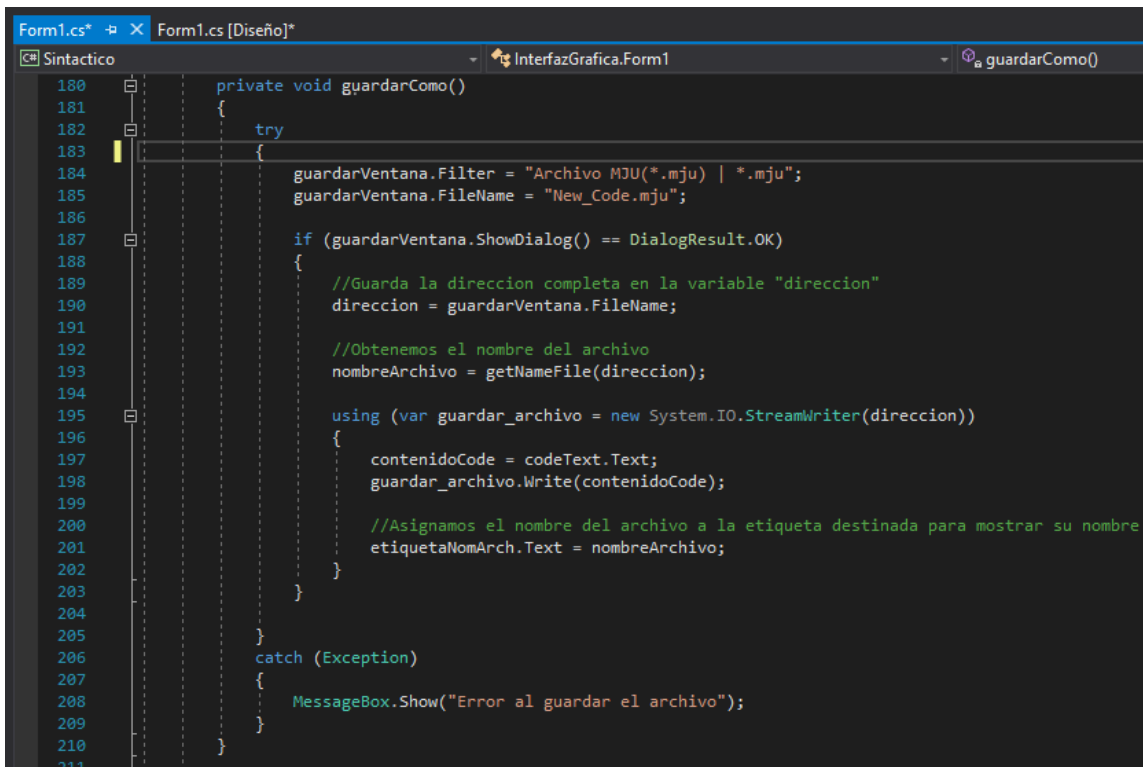
```
Form1.cs [Diseño]
C# Sintactico
InterfazGrafica.Form1

168 private void guardarCodeTemporal()
169 {
170     try
171     {
172         File.WriteAllText("temporary_file.mju", codeText.Text);
173     }
174     catch (Exception)
175     {
176     }
177 }
178
179
180 private void guardarComo()
```

Figura 31 –
Método
'guardarCodeTem
poral' ejecutado al
iniciar el proceso
de análisis

Este método sobrescribe el archivo temporal con el contenido actual de codeText, se ejecuta cada vez que se inicia el proceso de análisis. Su propósito es tener un lugar en donde almacenar el código escrito dentro de la aplicación sin necesidad de sobrescribir el archivo con el que se esta trabajando dejando la decisión al usuario de en que momento guardar los cambios realizados.

Método guardarComo()



```
Form1.cs* [Diseño]*
C# Sintactico
InterfazGrafica.Form1
guardarComo()

180 private void guardarComo()
181 {
182     try
183     {
184         guardarVentana.Filter = "Archivo MJU(*.mju) | *.mju";
185         guardarVentana.FileName = "New_Code.mju";
186
187         if (guardarVentana.ShowDialog() == DialogResult.OK)
188         {
189             //Guarda la direccion completa en la variable "direccion"
190             direccion = guardarVentana.FileName;
191
192             //Obtenemos el nombre del archivo
193             nombreArchivo = getNameFile(direccion);
194
195             using (var guardar_archivo = new System.IO.StreamWriter(direccion))
196             {
197                 contenidoCode = codeText.Text;
198                 guardar_archivo.Write(contenidoCode);
199
200                 //Asignamos el nombre del archivo a la etiqueta destinada para mostrar su nombre
201                 etiquetaNomArch.Text = nombreArchivo;
202             }
203         }
204     }
205     catch (Exception)
206     {
207         MessageBox.Show("Error al guardar el archivo");
208     }
209 }
210
211
```

Figura 32 –
Método
'guardarComo'
ejecutado tras
intentar guardar un
archivo

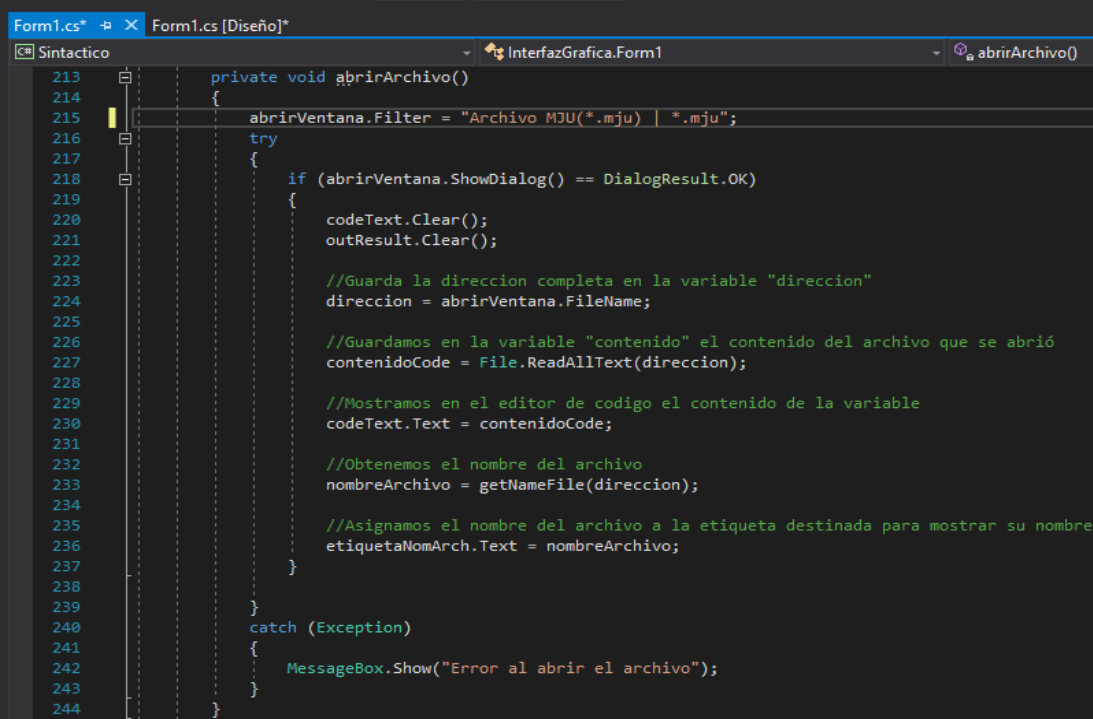
Se hace uso de un objeto `SaveFileDialog` llamado `guardarVentana`. Inicializa un filtro, por medio de la propiedad `Filter`, para los archivos con extensión `.mju`, del mismo modo se define un nombre por defecto para un archivo nuevo "New_Code.mju". La propiedad `FileName` contiene toda la ruta del archivo así como el nombre de éste.

Abre una ventana de dialogo de guardado de archivos donde el usuario define la ubicación en donde quiere guardar su archivo y le define un nombre (salvo que utilice el nombre por defecto). Si la operación es exitosa se actualizan las variables *direccion* (por medio de la propiedad `FileName`) y *nombreArchivo* (por medio del método `getNameFile()`).

Haciendo uso de un objeto `StreamWriter` (el cual recibe como parámetro la ruta del archivo) se procede a guardar el archivo, primero la variable *contenidoCode* obtiene el contenido de `codeText` y por medio del método `Write` de `StreamWriter` se hace la escritura del archivo con los caracteres de *contenidoCode*. Finalmente la etiqueta *etiquetaNomArch* cambia su valor asignándole *nombreArchivo*.

Si ocurre un error durante la operación se muestra un cuadro de dialogo informando que se ha producido un error.

Método `abrir()`



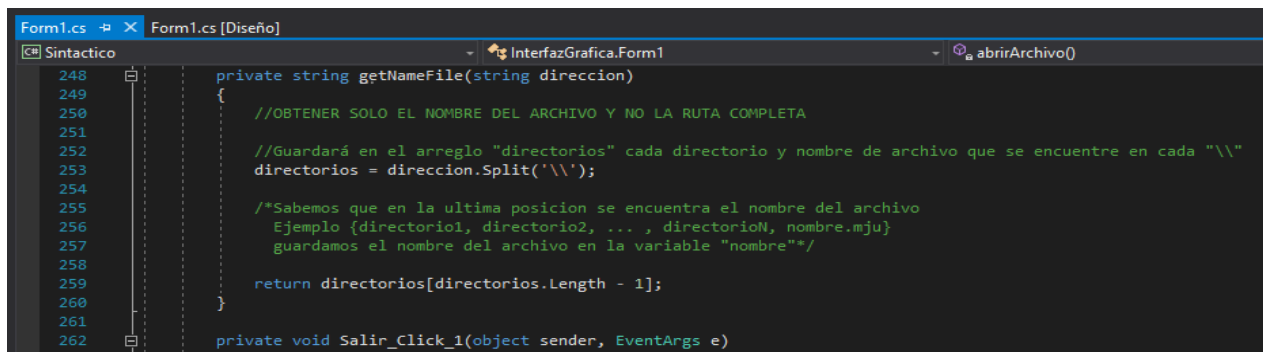
```
213 private void abrirArchivo()
214 {
215     abrirVentana.Filter = "Archivo MJU(*.mju) | *.mju";
216     try
217     {
218         if (abrirVentana.ShowDialog() == DialogResult.OK)
219         {
220             codeText.Clear();
221             outResult.Clear();
222
223             //Guarda la direccion completa en la variable "direccion"
224             direccion = abrirVentana.FileName;
225
226             //Guardamos en la variable "contenido" el contenido del archivo que se abrió
227             contenidoCode = File.ReadAllText(direccion);
228
229             //Mostramos en el editor de codigo el contenido de la variable
230             codeText.Text = contenidoCode;
231
232             //Obtenemos el nombre del archivo
233             nombreArchivo = getNameFile(direccion);
234
235             //Asignamos el nombre del archivo a la etiqueta destinada para mostrar su nombre
236             etiquetaNomArch.Text = nombreArchivo;
237         }
238     }
239     catch (Exception)
240     {
241         MessageBox.Show("Error al abrir el archivo");
242     }
243 }
244
```

Figura 33 –
Método
'*abrirArchivo*'
ejecutado tras
intentar abrir un
archivo

Haciendo uso de un objeto OpenFileDialog (abrirVentana) se crea un filtro, mediante la propiedad Filter, para los archivos con extensión .mju. Se abre la ventana para abrir los archivos, si se selecciona alguno se limpian las áreas de codeText y outResult por medio del método Clear(). La variable *direccion* obtiene la ruta de donde se abrió el archivo así como su nombre. Por medio de la clase File se lee el contenido del archivo con el método ReadAllText(direccion) el cual recibe como argumento la ruta y el nombre del archivo, este contenido se almacena en la variable *contenidoCode*, el recuadro *codeText* escribe el mismo contenido de *contenidoCode* mediante la propiedad Text. Se hace uso del método getNameFile(direccion) para obtener únicamente el nombre del archivo y guardarlo en la variable *nombreArchivo*. Finalmente *etiquetaNomArch* cambia su texto por el almacenado en *nombreArchivo*.

Si ocurrió algún error mientras se intentaba leer el archivo se muestra un cuadro de dialogo informando que se produjo un error.

Metodo String getNameFile(String direccion)



```
Form1.cs [Diseño]
Sintactico
InterfazGrafica.Form1
abrirArchivo()

248 private string getNameFile(string direccion)
249 {
250     //OBTENER SOLO EL NOMBRE DEL ARCHIVO Y NO LA RUTA COMPLETA
251
252     //Guardará en el arreglo "directorios" cada directorio y nombre de archivo que se encuentre en cada "\\"
253     directorios = direccion.Split('\\');
254
255     /*Sabemos que en la ultima posicion se encuentra el nombre del archivo
256     Ejemplo {directorio1, directorio2, ... , directorioN, nombre.mju}
257     guardamos el nombre del archivo en la variable "nombre"*/
258
259     return directorios[directorios.Length - 1];
260 }
261
262 private void Salir_Click_1(object sender, EventArgs e)
```

Figura 34 – Método 'getNameFile' se ejecuta automáticamente cuando se abre o guarda un archivo

Este método se utiliza para obtener únicamente el nombre del archivo y no la ruta completa. Se hace uso del arreglo *directorios*, el cual almacenara en cada una de sus posiciones cada directorio de la ruta en donde se encuentra el archivo. Los directorios se encuentran separados dentro de la ruta mediante doble diagonal invertida '\\' de modo que se debe de "partir" la dirección con esta condición para así inicializar el arreglo. Se sabe que dentro de la ruta el archivo siempre se encuentra al final es por eso que este método devuelve la ultima posición del arreglo.

Metodos:

Salir: salir_Click_1()

Copiar: copiar_Click_1()

Cortar: cortar_Click_1()

Pegar: pegar_Click_1()

Limpiar: limpiar_Click_1()

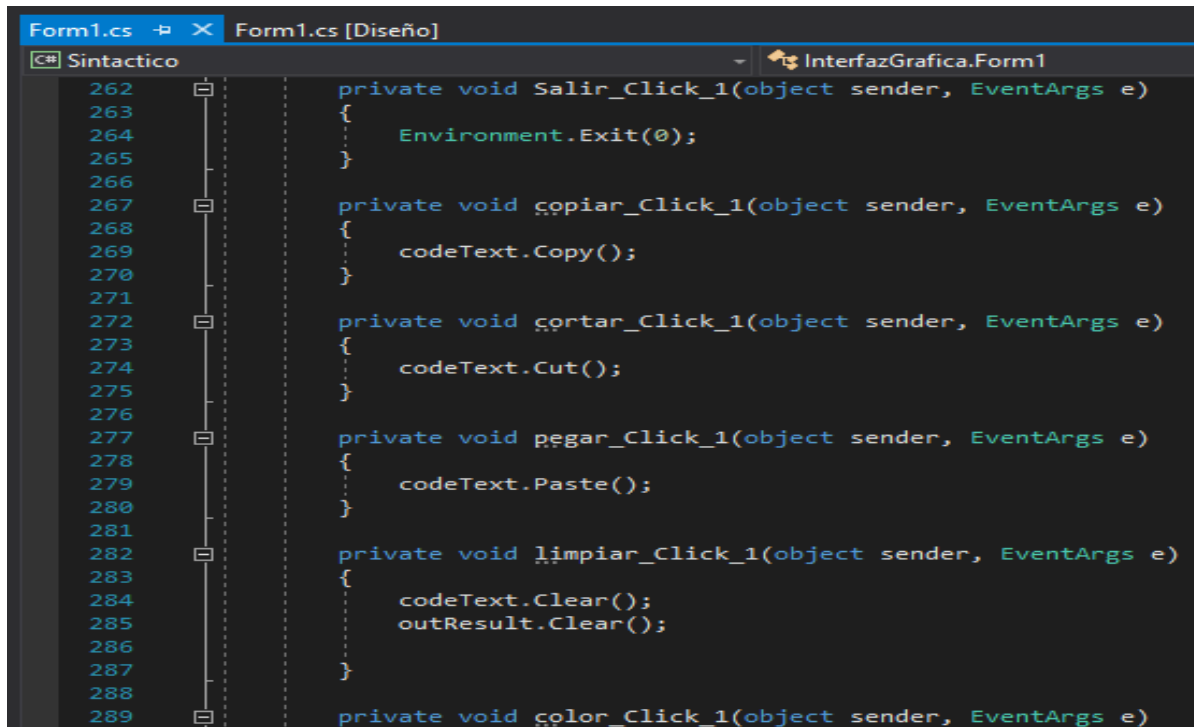


Figura 35 – Diversos métodos ejecutados desde eventos de menú

La opción Salir, dentro del menú Archivo, cierra la aplicación. Dentro del menú de Edición se encuentran diferentes opciones algunas de ellas son: Copiar, Cortar, Pegar y Limpiar. Las primeras tres opciones son las operaciones “clásicas” de edición de texto. Para hacer que estas sean funcionales basta con llamar a un objeto RichTextBox y escribir los métodos para su respectiva operación (estos métodos están especificados en inglés Copy() para copiar, Cut() para cortar y Paste() para pegar). La opción Limpiar, como su nombre lo indica, limpia ambos recuadros de texto, se hace llamado al método Clear() para cada objeto.

Métodos:

Color: color_Click_1()

Formato: formato_Click_1()

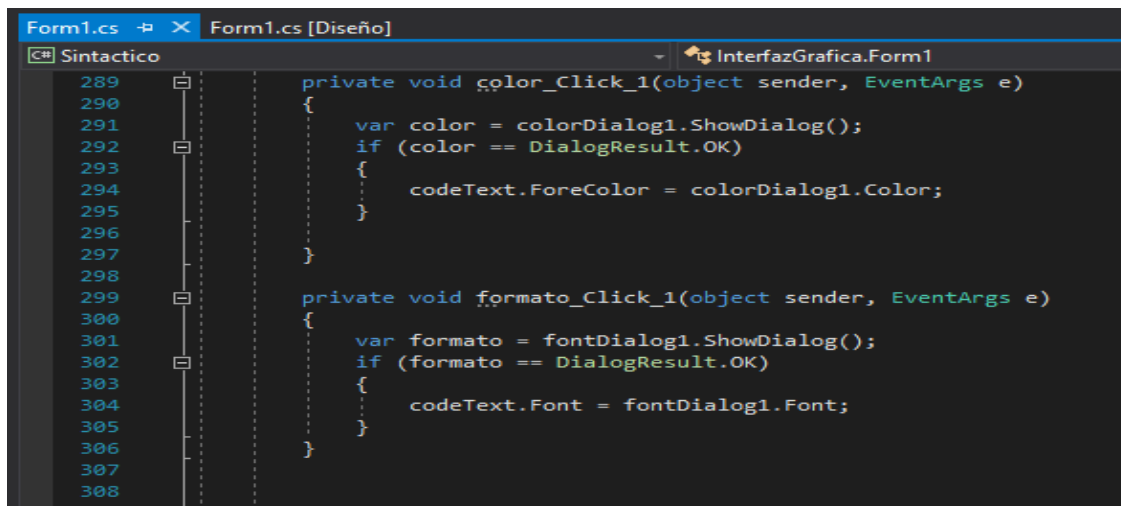
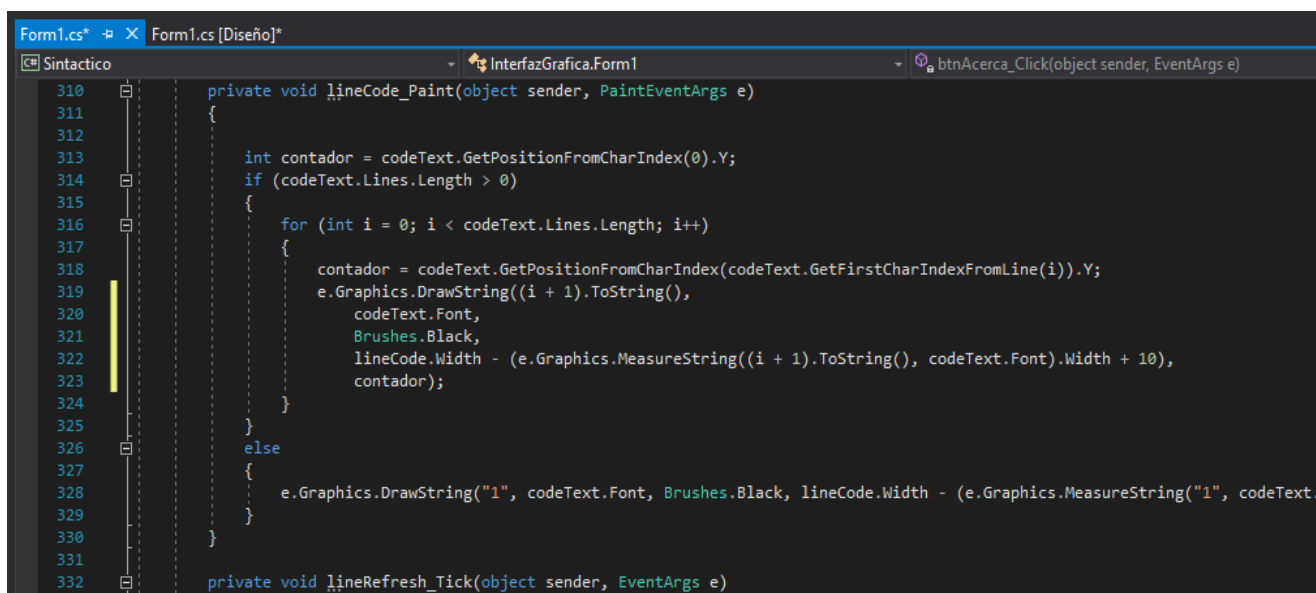


Figura 36 – Método 'Color' y 'Formato' producidos por eventos de menú

Los menus Color y Formato sirven para cambiar la apariencia del texto que se escribe dentro de codeText. Para hacer cambios de colores se usa un 'Dialogo de color' (ColorDialog) el cual abre una paleta en la que el usuario podrá escoger diferentes colores e incluso crear nuevos, una vez escogido un color se modifica la propiedad ForeColor de codeText que recibe el color seleccionado por el usuario. El formato de texto funciona de una forma similar, se hac uso de un FontDialog el cual ofrece al usuario diferentes opciones de formato. Cuando se define el nuevo formato se modifica la propiedad Font y el contenido de codeText cambia su apariencia, del mismo modo el contador de líneas *lineCode*. El formato por default es el siguiente: Fuente - *Microsoft Sans Serif*, Esitlo de Fuente – *Normal*, Tamaño – 8.

Método lineCode_Paint(Object sender, PaintEventArgs e)



```
Form1.cs* x Form1.cs [Diseño]*
Sintactico
InterfazGrafica.Form1
btnAcerca_Click(object sender, EventArgs e)

310 private void lineCode_Paint(object sender, PaintEventArgs e)
311 {
312
313     int contador = codeText.GetPositionFromCharIndex(0).Y;
314     if (codeText.Lines.Length > 0)
315     {
316         for (int i = 0; i < codeText.Lines.Length; i++)
317         {
318             contador = codeText.GetPositionFromCharIndex(codeText.GetFirstCharIndexFromLine(i)).Y;
319             e.Graphics.DrawString((i + 1).ToString(),
320                                 codeText.Font,
321                                 Brushes.Black,
322                                 lineCode.Width - (e.Graphics.MeasureString((i + 1).ToString(), codeText.Font).Width + 10),
323                                 contador);
324         }
325     }
326     else
327     {
328         e.Graphics.DrawString("1", codeText.Font, Brushes.Black, lineCode.Width - (e.Graphics.MeasureString("1", codeText.
329
330     }
331
332 private void lineRefresh_Tick(object sender, EventArgs e)
```

Figura 37 – Método 'lineCode_Paint', dibuja los números de líneas

Este método es utilizado para dibujar los números de líneas dentro del objeto PictureBox de nombre *lineCode*.

Se inicializa una variable de tipo int (*contador*) la cual obtiene la posición en Y del primer carácter de la primera línea mediante el método `GetPositionFromCharIndex(int posicion)` y se utiliza la propiedad 'Y' (en este caso la posición es 0). Se evalúa que la cantidad de saltos de línea dentro de `codeText` sea mayor que cero. Si esto es así se inicia un ciclo para dibujar todos los números de línea, el ciclo termina cuando se llega a la última línea obtenida mediante la propiedad `Lines.Length` de `codeText`. El contador se va actualizando en cada iteración con la posición en Y de cada línea.

Se hace llamado al método `Graphics.DrawString(String s, Font f, Brush b, float x, float y)` perteneciente al evento `PaintEventArgs`. Los argumentos de este método son: la cadena a dibujar, la fuente que se utilizara, la brocha que definirá el color de texto, la posición en X y la posición en Y.

Las cadenas serán definidas mediante el ciclo, notese que se hizo uso del método `toString()` esto es debido a que la variable *i* es de tipo entero. La fuente utilizada para dibujar las líneas será la misma que la de `codeText`. Para definir el color se llama a la clase `Brushes` y se define un color, en este caso negro (`Black`). Para obtener la posición en X se hacen unos cálculos matemáticos con el ancho de `lineCode` y el ancho de las cadenas que se pretende dibujar (estos cálculos pueden omitirse pero eso depende de la estética de cada programador. Finalmente la posición en Y ya ha sido obtenida con la variable *contador*.

Si solamente existe una línea entonces se dibuja un “1” de modo que, aunque no haya nada escrito en codeText, siempre se muestre al menos el número de la primer línea.

Método lineRefresh_Tick()

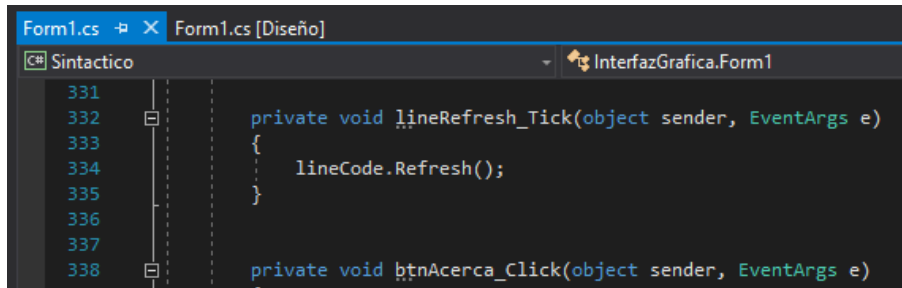


Figura 38 –
Método
'LineRefresh' este
método siempre
se encuentra en
ejecución

La única función de este método es la de actualizar los dibujos en lineCode en un intervalo de tiempo para mostrar los números de línea. El valor predeterminado para actualizar es de 100 milisegundos.

Métodos:

acerca: btnAcerca_Click()

Deshacer: botonDeshacer_Click() y deshacerMenu_Click()

Rehacer: rehacerBoton_Click() y rehacerMenu_Click

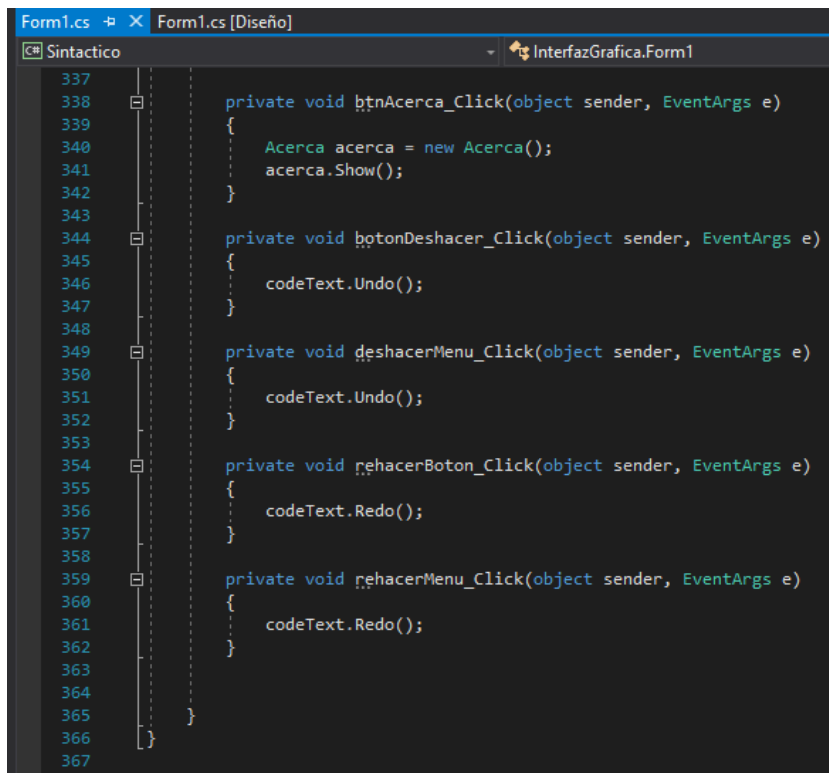


Figura 39 –
Diversos métodos
enviados por
eventos de
diferente tipo

Dentro del menú Ayuda se encuentra la opción 'acerca' la cual muestra información sobre la aplicación. La opción 'Deshacer' se puede encontrar dentro del menú 'Edición', en la barra de herramientas el botón 'Deshacer' o a través del atajo de teclado 'Ctrl + Z' las cuales llaman al mismo método de `codeText Undo()`. Así mismo la opción 'Rehacer' se encuentra en 'Edición' en la barra de herramientas 'Rehacer' con el atajo 'Ctrl + Y' y llama al método `Redo()`.

Varios de los métodos descritos anteriormente (en especial aquellos que en su nombre llevan '_Click') reciben dos argumentos uno de tipo `Object` (`sender`) y otro de tipo `EventArgs` (`e`). El primer parámetro es el objeto que produce el evento (que bien pueden ser los botones o las opciones de menú). El segundo contiene datos específicos del evento. Estos parámetros pueden ser utilizados si se desea obtener un mejor control sobre los eventos. Aunque dentro de esta interfaz no es necesario.

Funcionamiento de la interfaz

Vinculación del Fichero ejecutable con la aplicación de Visual.

Para poder vincular el archivo ejecutable que se obtiene tras compilar los ficheros de Flex y Bison con la interfaz que se ha creado, se hace uso de la clase Process. Dicha clase tiene métodos y propiedades para llamar procesos externos a la aplicación creada mediante Visual Studio y poder ejecutarlos.

Algunas de las propiedades y métodos mas importantes de la clase Process son:

Propiedad StartInfo: Hace la especificación de los valores que serán usados al iniciar el proceso.

- UseShellExecute: Tipo booleano. Hace uso del Shell del sistema operativo. Dependiendo del valor recibido activa o desactiva el uso de los comandos del sistema. Permitiendo al programador decidir si utilizarlos o no. Para este caso se apagará esta función pues el fichero ejecutable de Flex y Bison contiene su propio inicio de programa.
- RedirectStandardOutput: Tipo booleano. Esta propiedad de Process espera una salida de caracteres del proceso que se manda a llamar. Puesto que el ejecutable generado muestra mensajes en su salida esta opción se habilita. De este modo es posible capturar todas las salidas mediante métodos propios de la aplicación de Visual.
- FileName: Tipo String. Nombre de la aplicación que se pretende ejecutar o el nombre de un tipo de archivo que este asociado con una aplicación que tenga acciones para ejecutarse. El fichero generado, en este caso, por Flex y Bison se nombre "EJECUTABLE.EXE"
- Arguments: Tipo String. Son los argumentos que se pueden pasar a la aplicación especificada en FileName. Los argumentos son analizados e interpretados por la aplicación. El argumento necesario para el funcionamiento de la aplicación es un archivo temporal con el cual se realizan diferentes operaciones. Este archivo se nombra "temporary_file.mju"

- `Start()`: Tipo `Void`. inicia el recurso del proceso especificado en la propiedad `StartInfo` de la clase `Process`

Para que la clase `Process` haga uso del analizador léxico y sintáctico, el archivo ejecutable que se creó debe de estar ubicado en el directorio “bin\\debug” dentro del proyecto creado; en esta misma ubicación se debe de encontrar el archivo que se pasa como argumento del ejecutable.

También es posible hacer uso del ejecutable y del argumento de entrada sin necesidad de moverlos a la ubicación anteriormente mencionada, solo hay que buscar la ruta de los archivos y colocarla en sus respectivos métodos de `Process`; sin embargo esto puede alentar el proceso.

Bibliografía

Júlvez, J., Colom, J. and Álvarez, P. (2018). *Lenguajes Gramáticas y Autómatas. Ingeniería en Informática*. [online] Webdiis.unizar.es. Available at: <http://webdiis.unizar.es/asignaturas/LGA/> [Accessed 1 Apr. 2018].

Gálvez Rojas, S. and Mora Mata, M. (2005). *JAVA A TOPE: TRADUCTORES Y COMPILADORES CON LEX/YACC, JFLEX/CUP Y JAVACC*. 1st ed. Málaga.

R. León, C. (2012). *Análisis Léxico y Sintáctico*. 1st ed. [ebook] Available at: https://campusvirtual.ull.es/ocw/pluginfile.php/2209/mod_resource/content/0/perlexamples/perlexamples.pdf [Accessed 4 Apr. 2018].

Béjar Hernández, R. (2018). *Introducción a Flex y Bison*. 1st ed. [ebook] Zaragoza. Available at: http://webdiis.unizar.es/asignaturas/LGA/material_2004_2005/Intro_Flex_Bison.pdf [Accessed 4 Apr. 2018].

V. Aho, A., S. Lam, M., Sethi, R. and D. Ullman, J. (2008). *COMPILADORES. PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS*. 2nd ed. México: PEARSON EDUCACIÓN.

Simmross Wattenberg, F. (2018). *El generador de analizadores sintácticos yacc*. [ebook] Valladolid. Available at: <https://www.infor.uva.es/~mluisa/talf/docs/labo/L8.pdf> [Accessed 4 Apr. 2018].

De la Cruz, M. (2018). *Yacc/Bison*. [ebook] Available at: http://arantxa.ii.uam.es/~mdlcruz/docencia/compiladores/2002_2003/compiladores_02_03_yacc_bison.pdf [Accessed 4 Apr. 2018].

Donnelly, C. and Stallman, R. (2018). *Bison 1.27*. [online] Es.tldp.org. Available at: <http://es.tldp.org/Manuales-LuCAS/BISON/bison-es-1.27.html> [Accessed 4 Apr. 2018].

Paxson, V. (2018). Flex - un generador de analizadores léxicos. [online] Es.tldp.org. Available at: <http://es.tldp.org/Manuales-LuCAS/FLEX/flex-es-2.5.html> [Accessed 4 Apr. 2018].

De la Cruz, M. and Ortega, A. (2008). Construcción de un analizador léxico para ALFA con Flex. [ebook] Available at: http://arantxa.ii.uam.es/~mdlcruz/docencia/compiladores/2007_2008/lexico_07_08.pdf [Accessed 4 Apr. 2018].

NIETO BUTRÓN, M. (2013). *TRADUCTOR ENTRE HERRAMIENTAS CIENTÍFICA*. Licenciatura. Universidad Autónoma Nacional de México.

Bonet Esteban, E. (2018). Lenguaje C. 1st ed. [ebook] Xalapa. Available at: <http://informatica.uv.es/estguia/ATD/apuntes/laboratorio/Lenguaje-C.pdf> [Accessed 4 Apr. 2018].

M. Ritchie, D. and W. Kernighan, B. (1991). El Lenguaje de Programación C. 2nd ed. Mexico: PRENTICE HALL HISPANOAMERICANA.