
	Guía No. 4	Análisis Sintáctico de un Compilador.		Departamento de informática.
	Universidad Don Bosco Facultad de Ingeniería Es cuela de Computación			
	Asignatura		: Compiladores	
	Ciclo		: 01 – 2004	
	Lugar de ejecución:		: Centro de Computo	

I. OBJETIVOS

- ☐ Conocer las características básicas de los metalenguajes.
- ☐ Analizar e interpretar como un compilador realiza el análisis sintáctico de un programa.

II. INTRODUCCIÓN

Función del Analizador Sintáctico

En la guía anterior se mencionó que **sintaxis** es un conjunto de reglas formales que especifican la composición de los programas a base de letras, dígitos y otros caracteres. Por ejemplo, las reglas de sintaxis especifican en C/C++ que cada sentencia o línea de programa debe terminar con un “;”, o que la declaración de tipos debe ir antes que la de variables. (int var;).

La principal tarea del *analizador sintáctico* no es comprobar que la sintaxis del programa fuente sea correcta, sino construir una representación interna de ese programa y en el caso en que sea un programa incorrecto, dar un mensaje de error.

Para ello, el analizador sintáctico (A.S.) comprueba que el orden en que el analizador léxico le va entregando los *tokens* es válido. Si esto es así significará que la sucesión de símbolos que representan dichos *tokens* puede ser generada por la gramática correspondiente al lenguaje del código fuente.

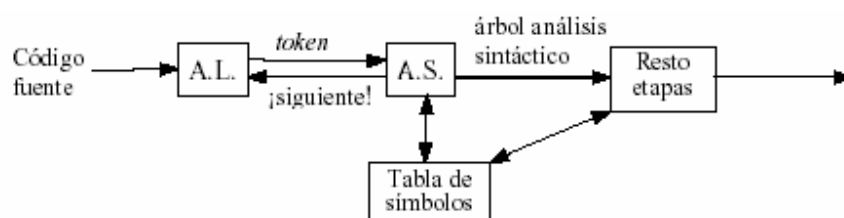


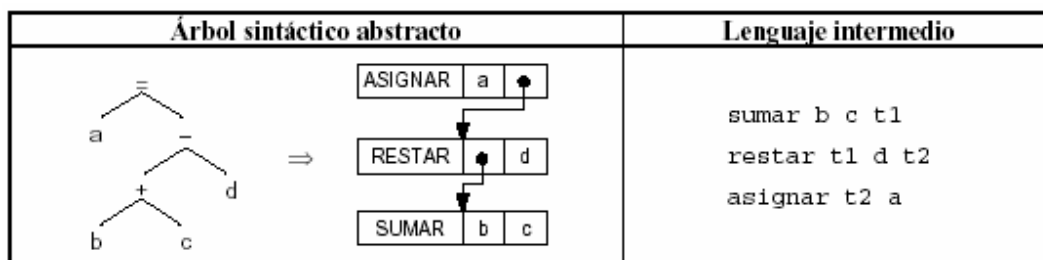
Fig. 1

La forma más habitual de representar la sintaxis de un programa es el árbol de análisis sintáctico, y lo que hacen los analizadores sintácticos es construir una derivación por la izquierda o por la derecha del programa fuente, que en realidad son dos recorridos determinados del árbol de análisis sintáctico. A partir de ese recorrido el analizador sintáctico debe construir una representación intermedia de ese programa fuente: un árbol sintáctico abstracto o bien un programa en un lenguaje intermedio; por este motivo, es muy importante que la gramática esté bien diseñada, e incluso es frecuente rediseñar la gramática original para facilitar la tarea de obtener la representación intermedia mediante un analizador sintáctico concreto.

Ejemplo:

Los *árboles sintácticos abstractos* son materializaciones de los árboles de análisis sintáctico en los que se implementan los nodos de éstos, siendo el nodo padre el operador involucrado en cada instrucción y los hijos sus operandos. Por otra parte, las representaciones intermedias son lenguajes en los que se han eliminado los

conceptos de mayor abstracción de los lenguajes de programación de alto nivel. Sea la instrucción “ $a=b+c-d$ ”, a continuación se muestra su representación mediante árboles sintácticos abstractos y un lenguaje intermedio:



El A.S. constituye el esqueleto principal del compilador. Habitualmente el analizador léxico se implementa como una rutina dentro del sintáctico, al que devuelve el siguiente *token* que encuentre en el *buffer* de entrada cada vez que éste se lo pide. Así mismo, gran parte del resto de etapas de un programa traductor están integradas de una u otra forma en el analizador sintáctico.

Metalinguajes y Notación BNF (Backus-Naur Form)

Los lenguajes de programación son un conjunto finito o infinito de sentencias. Por lo tanto el medio habitual para describir un lenguaje es su gramática.

Los metalinguajes, *son herramientas para la descripción formal de los lenguajes*, facilitando no solo la comprensión del lenguaje, sino también el desarrollo del compilador correspondiente. Ejemplos: las expresiones regulares que describen los componentes léxicos del lenguaje; las notaciones BNF, EBNF y los diagramas sintácticos que describen la sintaxis de los lenguajes.

Notación BNF

John Backus participo a principios de los años 60 en el desarrollo del lenguaje ALGOL, que utilizo por primera vez la forma BNF.

La notación **BNF** utiliza los siguientes metasímbolos:

- <> Encierra conceptos definidos o por definir.
- ::= Sirve para definir o indicar equivalencias.
- | Separa las distintas alternativas.
- " " Indica que el metasímbolo que aparece entre comillas es un carácter que forma parte de la sintaxis del lenguaje.
- () Se permite el uso de paréntesis para hacer agrupaciones.

Existen símbolos con entidad propia llamados símbolos terminales. También existen otros que se deben definir y se denominan no terminales.

Ejemplo:

Definición de un identificador

La definición de un identificador en BNF es la siguiente: (observe que se permiten definiciones recursivas)

```

<ident> ::= <letra> | <ident><letra> | <ident><digito>
<letra> ::= a | b | c | ... | y | z
<digito> ::= 0 | 1 | 2 | ... | 8 | 9

```

En este caso a, b, c..., y, z y 0, 1, 2,..., 9 son los símbolos terminales, y el resto los símbolos no terminales. Puede observarse que se ha definido *identificador* de una manera recursiva.

Notación EBNF (Extended BNF)

La notación EBNF añade el metasímbolo {} a la notación BNF, para indicar que lo que aparece entre las llaves puede repetirse 0 o más veces.

Ejemplo:

Expresión aritmética con uno o varios factores

Un término de una expresión aritmética se puede expresar en BNF como:

$$\langle \text{term} \rangle ::= \langle \text{factor} \rangle \mid \langle \text{term} \rangle * \langle \text{factor} \rangle$$

Sin embargo usando EBNF se puede expresar como:

$$\langle \text{term} \rangle ::= \langle \text{factor} \rangle \{ * \langle \text{factor} \rangle \}$$

Análisis Sintáctico

El análisis sintáctico es un análisis a nivel de sentencias, y es mucho más complejo que el análisis léxico. Su función es tomar el programa fuente en forma de *tokens*, que recibe del analizador léxico, y determinar la estructura de las sentencias del programa. Este proceso es similar a determinar la estructura de una frase en Castellano, determinando quien es el sujeto, predicado, el verbo y los complementos.

El análisis sintáctico agrupa a los *tokens* en clases sintácticas (denominadas **no terminales** en la definición de la gramática), tales como expresiones, procedimientos,... El analizador sintáctico o *parser* obtiene un árbol sintáctico (u otra estructura equivalente) en la cual las hojas son los tokens, y cualquier nodo que no sea una hoja, representa un tipo de clase sintáctica (operaciones). Por ejemplo el análisis sintáctico de la siguiente expresión:

$$(A+B)*(C+D)$$

con las reglas de la gramática que se presenta a continuación dará lugar al árbol sintáctico de la figura 3:

$\langle \text{expresión} \rangle$	$::= \langle \text{término} \rangle \langle \text{más términos} \rangle$
$\langle \text{más términos} \rangle$	$::= + \langle \text{término} \rangle \langle \text{más términos} \rangle$ $\mid - \langle \text{término} \rangle \langle \text{más términos} \rangle$ $\mid \langle \text{vacío} \rangle$
$\langle \text{término} \rangle$	$::= \langle \text{factor} \rangle \langle \text{más factores} \rangle$
$\langle \text{más factores} \rangle$	$::= * \langle \text{factor} \rangle \langle \text{más factores} \rangle$ $\mid / \langle \text{factor} \rangle \langle \text{más factores} \rangle$ $\mid \langle \text{vacío} \rangle$
$\langle \text{factor} \rangle$	$::= (\langle \text{expresión} \rangle) \mid \langle \text{variable} \rangle \mid \langle \text{constante} \rangle$

La estructura de la gramática anterior refleja la prioridad de los operadores, así los operadores “+” y “-” tienen la prioridad más baja, mientras que “*” y “/” tienen una prioridad superior. Se evaluarán en primer lugar las constantes, variables y expresiones entre paréntesis.

Los árboles sintácticos se construyen con un conjunto de reglas conocidas como **gramática**, y que definen con total precisión el lenguaje fuente.

Al proceso de reconocer la estructura del lenguaje fuente se conoce con el nombre de análisis sintáctico (*parsing*). Hay distintas clases de analizadores o reconocedores sintácticos, pero en general se clasifican en 2 grandes grupos: A.S. Ascendentes y A.S. Descendentes.

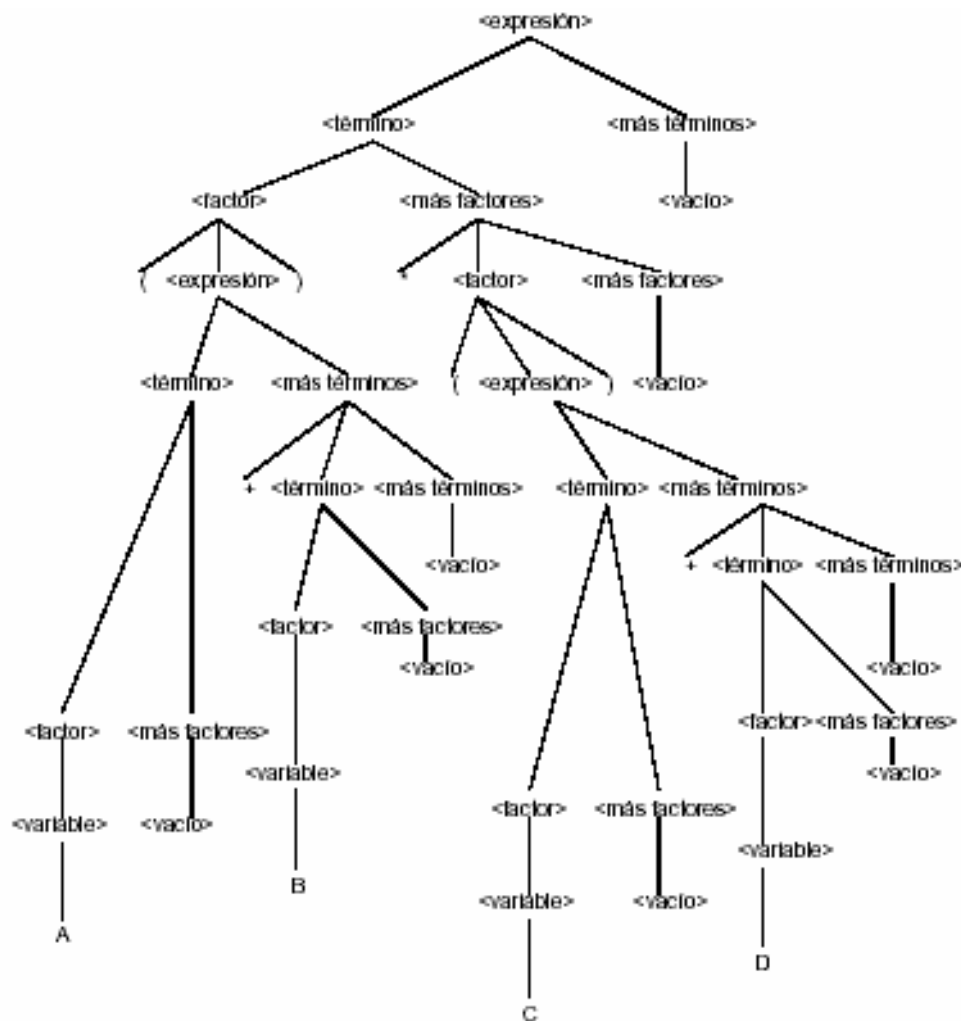


Fig. 3

Tipos de Análisis Sintácticos

Desde el punto de vista de la teoría de Análisis Sintáctico, hay dos estrategias para construir el árbol sintáctico:

- **Análisis descendente:** partimos de la raíz del árbol (donde estará situado el axioma o símbolo inicial de la gramática) y se van aplicando reglas por la izquierda de forma que se obtiene una derivación por la izquierda de la cadena de entrada. Para decidir qué regla aplicar, se lee un *token* de la entrada. Recorriendo el árbol de análisis sintáctico resultante, en profundidad de izquierda a derecha, encontraremos en las hojas del árbol los *tokens* que nos devuelve el Analizador Léxico (A.L.) en ese mismo orden.
- **Análisis ascendente:** partiendo de la cadena de entrada, se construye el árbol de análisis sintáctico empezando por las hojas (donde están los *tokens*) y se van creando nodos intermedios hasta llegar a la raíz (hasta el símbolo inicial), construyendo así el árbol de abajo a arriba. El recorrido del árbol se hará desde las hojas hasta la raíz. El orden en el que se van encontrando las producciones corresponde a la inversa de una derivación por la derecha.

Las dos estrategias recorren la cadena de entrada de izquierda a derecha una sola vez, y necesitan (para que el análisis sea eficiente) que la gramática no sea ambigua.

Ejemplo:

Entrada: "num*num+num"

Gramática:

$E ::= E + T \mid T$

$T ::= T * F \mid F$

$F ::= \text{num}$

(E: expresión, T: término, F: factor)

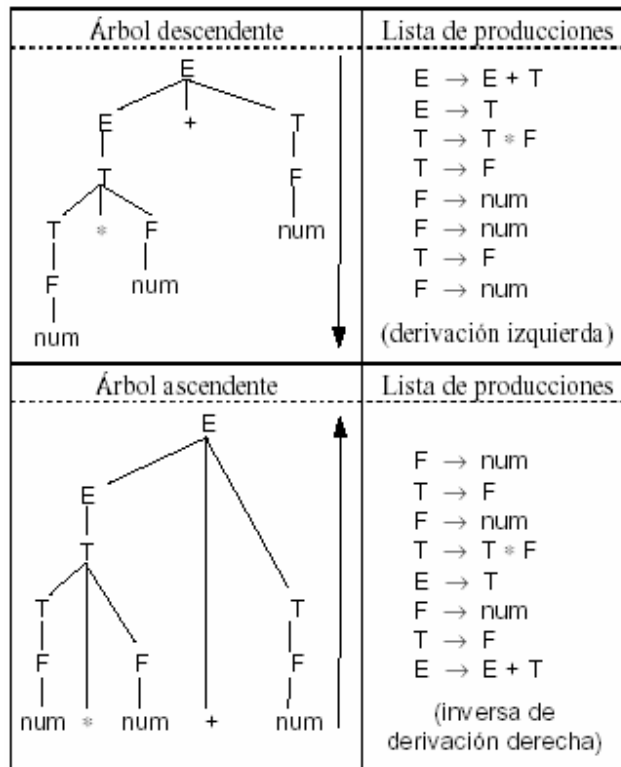


Fig. 4

Obsérvese que en el análisis descendente, partiendo del símbolo inicial hasta alcanzar las hojas, obtenemos una derivación por la izquierda. En el ascendente, partiendo de las hojas hasta llegar al axioma obtenemos la inversa de una derivación por la derecha.

Ambos tipos de análisis son eficientes pero no son capaces de trabajar con todo tipo de gramáticas (el análisis de tipo general sí es capaz de tratar cualquier gramática, pero es ineficiente para el diseño de compiladores), pero algunas gramáticas adecuadas para estos tipos de análisis son muy convenientes para describir la mayoría de lenguajes de programación, por ejemplo los siguientes:

a) **Análisis LL(n)**

b) **Análisis LR(n)**

Donde:

L \rightarrow Left to Right: la secuencia de *tokens* de entrada se analiza de izquierda a derecha. (L de a y b).

L \rightarrow Left-most: utiliza las derivaciones más a la izquierda.

R \rightarrow Right-most: utiliza las derivaciones más a la derecha.

$n \rightarrow$ es el número de símbolos de entrada que es necesario conocer en cada momento para poder hacer el análisis.

Con respecto al análisis sintáctico descendente, se plantea el problema del retroceso (*backtracking*).

El **retroceso**, se produce cuando se eligen mal las producciones para alcanzar la sentencia a analizar, y debe darse marcha atrás en el proceso de análisis, para elegir otra alternativa. El proceso se repite hasta que se ha elegido la alternativa correcta; evidentemente esto hace lento el proceso de análisis.

Para evitar el retroceso, las gramáticas deben de cumplir unas determinadas propiedades, dando lugar a los tipos de gramáticas **LL(k)**, descritas anteriormente.

En general, las gramáticas LL(k) realizan un análisis descendente determinista por medio del reconocimiento de la cadena de entrada de izquierda a derecha y va tomando las derivaciones más a la izquierda (*left-most*) con solo mirar los **k** tokens situados a continuación de donde se halla.

Por ejemplo:

- Se utilizan las gramáticas LL(1) para la descripción de lenguajes, que sólo miran un token (figura 3).
- Una gramática LL(2) es aquella cuyas cadenas son analizadas de izquierda a derecha y para las que es necesario mirar dos *tokens* para saber qué derivación tomar para el no terminal más a la izquierda en el árbol de análisis.

Con estos tipos de análisis se puede implementar un analizador para cualquier gramática de contexto libre.

III. PROCEDIMIENTO

A continuación se presenta la gramática aceptada por nuestro lenguaje de programación llamado "**LPTLP**", utilizando la notación BNF:

<Programa>	::=	M "{ <bloque> }"
<bloque>	::=	<sentencia> <otra_sentencia>
<otra_sentencia>	::=	; <sentencia> <otra_sentencia> <vacío>
<sentencia>	::=	<asignación> <lectura> <escritura>
<asignación>	::=	<variable> = <expresión>
<expresión>	::=	<termino> <mas términos>
<mas términos>	::=	+ <termino> <mas términos> - <termino> <mas términos> <vacío>
<termino>	::=	<factor> <mas factores>
<mas factores>	::=	* <factor> <mas factores> / <factor> <mas factores> % <factor> <mas factores> <vacío>
<factor>	::=	(<expresión>) <variable> <constante>
<lectura>	::=	R <variable>
<escritura>	::=	W <variable>
<variable>	::=	a b c ... z
<constante>	::=	0 1 2 ... 9

Puede observarse que este lenguaje sólo permite 3 tipos de sentencias lectura, asignación y escritura. Tiene un sólo tipo de datos: entero. Las variables están formadas por una única letra minúscula, y las constantes son de un dígito. Tiene 5 operadores + (adición), - (diferencia), * (producto), / (división) y % (módulo). Se permite el uso de paréntesis.

La precedencia de los operadores es la siguiente (de mayor a menor):

- 1ª. Evaluación de expresiones entre paréntesis ().
- 2ª. Producto, División entera y Módulo.
- 3ª. Suma y Resta.

Las reglas sintácticas siguientes en BNF:

```
<bloque> ::= <sentencia> <otra_sentencia>
<otra_sentencia> ::= ; <sentencia> <otra_sentencia>
                | <vacío>
```

también puede escribirse en una sola EBNF de la siguiente forma:

```
<bloque> ::= <sentencia> { ; <sentencia> }
```

Creación del Analizador Sintáctico

La técnica a utilizar es la denominada *recursivo descendente sin retroceso*, para su utilización se deben de cumplir dos requisitos:

- El lenguaje fuente debe estar definido con una gramática LL(1). Con un solo token se decide la regla de la gramática que debe aplicarse.
- El lenguaje de implementación debe de permitir la recursividad.

Antes de correr y probar el Analizador Sintáctico, cree las librerías **Léxico.h** y **Sintactico.h** (desde Visual C++ o BorlandC) siguiendo estos pasos:

- Ingrese al Menu **File**, luego seleccione **New**
- Seleccionamos **C/C++ Header File**
- Le ponemos un nombre al archivo, en este caso **Lexico**.

A continuación se presenta el código que debe insertar en la librería: (la **Clase Léxico se implemento en la guía anterior**)

```
/* Para evitar definiciones múltiples entre módulos */
#ifndef LEXICO_H
#define LEXICO_H

#include <iostream.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

# define TAM_BUFFER 100

class Lexico
{
    char *nombreFichero; // Nombre del fichero fuente de entrada
    FILE *entrada;       // Fichero de entrada
    int nl;              // Número de línea
    int traza;           // Control de traza
    char buffer[TAM_BUFFER]; // Buffer auxiliar de caracteres
    int pBuffer;         // posición en el buffer auxiliar

public:
    Lexico(char *unNombreFichero, int una_traza=0);
    ~Lexico(void);
    char siguienteToken(void);
    void devuelveToken(char token);
    int lineaActual(void) {return nl;};
    int existeTraza(void) {if (traza) return 1; else return 0;}
};

Lexico::Lexico(char *unNombreFichero, int una_traza)
{
```

```

if((entrada=fopen(unNombreFichero,"r"))==NULL)
{
    cout<<"No se puede abrir el fichero "<<unNombreFichero<<endl;
    exit (-2);
}

```

```

if (una_traza) traza=1;
else traza=0;
nl=1;    // Se inicializa el contador de líneas
pBuffer=0; //Se inicializa la posición del buffer
}

```

```

Lexico::~Lexico(void)

```

```

{
    fclose(entrada);
}

```

```

char Lexico::siguienteToken (void)

```

```

{
    char car;
    while ((car=
        ((pBuffer>0) ? buffer[--pBuffer] : getc(entrada))
        )!=EOF)
    { if (car==' ')continue;
      if (car=='\n'){++nl;continue;}
      break;
    }
}

```

```

if (traza)
{
    cout<<"ANALIZADOR LEXICO: Lee el token "<<car<<endl;
    //system("pause");//nuevo
}

```

```

switch (car)
{
    case 'M':
    case 'R':
    case 'W':    // palabras reservadas
    case '=':    // asignacion
    case '(':    // par,ntesis
    case ')':
    case ':':    // separadores
    case '{':
    case '}':
    case '.':    // fin de programa
    case '+':    // operadores aritm,ticos
    case '-':
    case '*':
    case '/':
    case '%': return (car);
}

```

```

if (islower(car)) return(car); //variables
else if (isdigit(car)) return (car); //constantes
else
{
    cout<<"ERROR LEXICO: TOKEN DESCONOCIDO"<<endl;
    exit(-4);
}
return(car);

```



```

}

void Lexico::devuelveToken (char token)
{
    if (pBuffer>TAM_BUFFER)
    {
        cout<<"ERROR: DESBORDAMIENTO DEL BUFFER DEL ANALIZADOR LEXICO"<<endl;
        exit(-5);
    }
    else
    {
        buffer[pBuffer++]=token;
        if (existeTraza())
            cout<<"ANALIZADOR LEXICO: Recibe en buffer el token "<<token<<endl;
    }
}
#endif

```

d) Cree otra librería (**Sintacti.h**) para insertar el código que se presenta a continuación: (Este código es la implementación del analizador léxico).

```

#ifndef SINTACTI_H
#define SINTACTI_H

#include "lexico.h"
#include <stdlib.h>

class Sintactico
{
    void programa (void);
    void bloque (void);
    void sentencia (void);
    void otra_sentencia (void);
    void asignacion(void);
    void lectura (void);
    void escritura(void);
    void variable(void);
    void expresion(void);
    void termino(void);
    void mas_terminos(void);
    void factor(void);
    void mas_factores(void);
    void constante(void);
    void errores (int codigo);
    Lexico lexico; //objeto léxico miembro de la clase
public:
    Sintactico(char *fuente, int traza);
    ~Sintactico(void);
};

Sintactico::Sintactico(char *fuente, int traza)
    :lexico(fuente, traza) //se inicializa el constructor de la clase léxico
{
    if (lexico.existeTraza())
        cout<<"INICIO DE ANALISIS SINTACTICO"<<endl;
    programa();
}

```

```

/*****/

Sintactico::~Sintactico(void)
{
    if (lexico.existeTraza())
    {
        cout<<"FIN DE ANALISIS SINTACTICO"<<endl;
        cout<<"FIN DE COMPILACION"<<endl;
    }
}

/*****/
void Sintactico::programa(void)
{
    char token;
    if (lexico.existeTraza())
        cout<<"ANALISIS SINTACTICO: <PROGRAMA>"<<endl;

    token=lexico.siguienteToken();
    if (token=='M') cout <<"M";
    else errores(8);

    token=lexico.siguienteToken();
    if (token!='{') errores(9);

    bloque();

    token=lexico.siguienteToken();
    if (token=='}')
    {
        cout<<" ";
    }
    else errores(2);
}

/*****/
void Sintactico::bloque(void)
{
    char token=' ';
    if (lexico.existeTraza())
        cout<<"ANALISIS SINTACTICO: <BLOQUE>"<<endl;

    sentencia();
    otra_sentencia();
}

/*****/
void Sintactico::otra_sentencia(void)
{
    char token;
    if (lexico.existeTraza())
        cout<<"ANALISIS SINTACTICO: <OTRA_SENTENCIA>"<<endl;

    token=lexico.siguienteToken();
    if (token==';')
    {
        sentencia();
        otra_sentencia();
    }
}

```

```

    else lexico.devuelveToken(token); //vacio
}

/*****/

void Sintactico::sentencia(void)
{
    char token;
    if (lexico.existeTraza())
        cout<<"ANALISIS SINTACTICO: <SENTENCIA>"<<endl;

    token=lexico.siguienteToken();

    if ((token>='a') && (token<='z'))
    {
        lexico.devuelveToken(token);
        asignacion();
    }
    else if (token=='R') lectura();
    else if (token=='W') escritura();
    else errores(6);
}

/*****/

void Sintactico::asignacion()
{
    char token;
    if (lexico.existeTraza())
        cout<<"ANALISIS SINTACTICO: <ASIGNACION>"<<endl;

    variable();
    token=lexico.siguienteToken();
    if (token!='=') errores(3);
    expresion();
}

/*****/

void Sintactico::variable(void)
{
    char token;
    if (lexico.existeTraza())
        cout<<"ANALISIS SINTACTICO: <VARIABLE>"<<endl;

    token=lexico.siguienteToken();
    if ((token>='a') && (token<='z')) cout<<token;
    else errores(5);
}

/*****/

void Sintactico::expresion(void)
{
    if (lexico.existeTraza())
        cout<<"ANALISIS SINTACTICO: <EXPRESION>"<<endl;

    termino();
    mas_terminos();
}

/*****/

void Sintactico::termino(void)
{
    if (lexico.existeTraza())

```

```

        cout<<"ANALISIS SINTACTICO: <TERMINO>"<<endl;
factor();
mas_factores();
}

/*****/

void Sintactico::mas_terminos(void)
{
    char token;
    if (lexico.existeTraza())
        cout<<"ANALISIS SINTACTICO: <MAS_TERMINOS>"<<endl;
    token=lexico.siguienteToken();
    if (token=='+')
    {
        termino();
        mas_terminos();
    }
    else if (token == '-')
    {
        termino();
        mas_terminos();
    }
    else lexico.devuelveToken(token); // <vacio>
}
/*****/

void Sintactico::factor(void)
{
    char token;
    if (lexico.existeTraza())
        cout<<"ANALISIS SINTACTICO: <FACTOR>"<<endl;
    token=lexico.siguienteToken();

    if ((token>='0') && (token<='9'))
    {
        lexico.devuelveToken(token);
        constante();
    }
    else if (token=='(')
    {
        expresion();
        token=lexico.siguienteToken();
        if (token!=')') errores(4);
    }
    else
    {
        lexico.devuelveToken(token);
        variable();
    }
}

/*****/

void Sintactico::mas_factores(void)
{
    char token;
    if (lexico.existeTraza())
        cout<<"ANALISIS SINTACTICO: <MAS_FACTORES>"<<endl;
    token=lexico.siguienteToken();

    switch (token)

```

```

{
    case '*':factor();
        mas_factores();
        break;
    case '/':factor();
        mas_factores();
        break;
    case '%':factor();
        mas_factores();
        break;
    default: //<vacio>
        lexico.devuelveToken(token);
}
}

/*****/
void Sintactico::lectura(void)
{
    char token;
    token=lexico.siguienteToken();

    if (lexico.existeTraza())
        cout<<"ANALISIS SINTACTICO: <LECTURA> "<<token<<endl;

    if((token<'a')||(token>'z')) errores(5);
}

/*****/
void Sintactico::escritura(void)
{
    char token;
    token=lexico.siguienteToken();

    if (lexico.existeTraza())
        cout<<"ANALISIS SINTACTICO: <ESCRITURA> "<<token<<endl;

    if ((token<'a' || (token>'z')) errores(5);
}

/*****/
void Sintactico::constante(void)
{
    char token;
    if (lexico.existeTraza())
        cout<<"ANALISIS SINTACTICO: <CONSTANTE>"<<endl;
    token=lexico.siguienteToken();
    if ((token>='0') && (token<='9')) cout<<token;
    else errores(7);
}

/*****/
void Sintactico::errores(int codigo)
{
    cout<<"LINEA "<<lexico.lineaActual();
    cout<<" ERROR SINTACTICO "<<codigo;
    switch (codigo)
    {
        case 1 :cout<<" :ESPERABA UN ;"<<endl;break;
        case 2 :cout<<" :ESPERABA UNA }"<<endl;break;
        case 3 :cout<<" :ESPERABA UN ="<<endl;break;
    }
}

```

```

        case 4 :cout<<" :ESPERABA UN )" <<endl;break;
        case 5 :cout<<" :ESPERABA UN IDENTIFICADOR" <<endl;break;
        case 6 :cout<<" :INSTRUCCION DESCONOCIDA" <<endl;break;
        case 7 :cout<<" :ESPERABA UNA CONSTANTE" <<endl;break;
        case 8 :cout<<" :ESPERABA UNA M DE MAIN" <<endl;break;
        case 9 :cout<<" :ESPERABA UNA {" <<endl;break;
        default:
        cout<<" :NO DOCUMENTADO" <<endl;
        }
        exit(-(codigo+100));
    }
#endif

```

e) A continuación cree un archivo de texto (EJEM2.txt), guárdelo en la unidad C y digite el siguiente código en LPTLP:

```

M
{
R a;
R b;
c = a + b - 2;
W c;
}

```

f) Finalmente cree el programa principal (ppal.cpp) para probar el analizador sintáctico:

```

#include <iostream.h>
#include "sintacti.h"

void main(){

Sintactico sintactico("c:\\EJEM2.txt",1); //se crea una instancia de la clase

}

```

Análisis del programa:

En el analizador (clase Sintactico) se define un método por cada símbolo no terminal de la gramática. Además se define un método para el tratamiento de errores sintácticos.

El analizador estudia el programa fuente, sentencia a sentencia, en primer lugar revisa si la sentencia es de lectura (R) o de escritura (W), en caso contrario es una sentencia de asignación.

Detrás de cada sentencia de lectura o escritura espera una variable y el final de sentencia o final de programa. En las sentencias de asignación espera en primer lugar una variable seguida del símbolo de asignación y una expresión. En este caso se analiza la expresión según la gramática definida por el lenguaje LPTLP, tomando primero <termino> y después <mas términos>, aplicándose recursivamente hasta el análisis completo de la expresión.

La prioridad de los operadores ya esta reflejada en la gramática. Finalmente el analizador detecta errores sintacticos y envía mensaje de error.

Actividad 1

Una vez corrido el programa, debe verificar la salida en pantalla para ver como se construye el arbol sintáctico descrito anteriormente según la gramática del lenguaje. Para ello, ingrese a la línea de comando y ubíquese en el directorio donde se creo el archivo ejecutable, por ejemplo:

c:\>Guia5\Debug>

Una vez ubicado en el directorio ejecute el programa de la siguiente forma:

c:\>Guia5\Debug>ppal.exe > salida.txt

Lo anterior permite que la salida en pantalla (consola - DOS) se almacene (se direcciona) en el archivo **salida.txt**.

Finalmente ingrese a un editor de texto (edit o notepad) para ver el archivo.

Actividad 2

Analice la salida del programa y construya (dibuje) el árbol sintáctico generado por el analizador, para una mejor comprensión del programa.

Actividad 3

En la implementación de la clase Sintactico, solo falta traducir o convertir el lenguaje LPTLP a un programa objeto. Trate de implementar un programa objeto (lenguaje máquina u otro que usted quiera) a partir del analizador sintáctico.

IV. INVESTIGACIÓN Y EJERCICIOS COMPLEMENTARIOS

- Investigar sobre la notación LLn y LLr