UNIVERSIDAD DE GUADALAJARA

Red Universitaria e Institución Benemérita de Jalisco

Centro Universitario de Ciencias Exactas e Ingenierias

Seminario de solución de problemas de traductores II

Profesor: José Juan Meza Espinoza

Sección: D01

Practica 2: Analizador sintáctico

Alumnos:

- Botello Martínez Nadia Noemi
- Ángel Gabriel Mercado Hernández
- Ivo Alberto Ramirez Gaeta
- Uziel Reyes Becerra

30 de octubre 2024

1. Introducción

En esta actividad se detalla el código escrito en C++ para implementar un analizador sintáctico, incluyendo ejemplos de su aplicación. Este programa representa una extensión del analizador léxico presentado anteriormente; por lo tanto, no se explicarán nuevamente esos segmentos de código.

2. Contenido

El código presentado es un analizador sintáctico básico diseñado para procesar y validar fragmentos de código en el lenguaje de programación C++. Su objetivo principal es identificar y clasificar diferentes componentes del código, como palabras reservadas, tipos de datos, operadores, identificadores, números, paréntesis, corchetes y caracteres especiales.

Para el desarrollo del analizador sintáctico, se requieren dos aspectos fundamentales:

- Determinar el tipo de analizador a implementar (ascendente o descendente).
- Establecer la gramática adecuada.
- De este modo, para este programa se decidió crear un analizador sintáctico descendente (LL(1)) cuyas reglas gramaticales, para los distintos tipos de operaciones *aceptadas en nuestro pequeño programa*, son las siguientes:

1) Gramática para asignaciones simples (forma a = b;):

$$
\begin{aligned}
G = \{&E \rightarrow VSDP;\\
&V \rightarrow LV;\\
&L \rightarrow a\ldots z;\\
&D \rightarrow 0\ldots 9D \mid \lambda;\\
&S \rightarrow \text{'='};\\
&P \rightarrow \text{';'}\}
\end{aligned}
$$

2) Gramática para asignaciones que contienen sumas o restas (forma a = b + c;)

$$
\begin{aligned}
G = \{&E \rightarrow VSDP;\\
&V \rightarrow LV;\\
&L \rightarrow a\ldots z;\\
&S \rightarrow \text{'='};\\
&D \rightarrow 0\ldots 9D \mid \lambda;
\end{aligned}
$$

P → ‘;’}

3) Gramática para declaraciones (forma int a;)

G = {E → DSVP;
D → ‘int’ | ‘float’ | …;
S → ‘ ’;
V → LV;
L → a…z;
P → ‘;’}

4) Gramática para instrucción printf (forma printf(“%t”, v);)

G = {E → IP_aCOTCMVP_cS;
I → ‘printf’;
P_a → ‘(‘;
P_c → ‘)’,
C→ ‘“‘; O→ ‘%’;
T→ ‘d’ | ‘f’;
M→ ‘,’;
V → LV;
L → a…z;
S → ‘;’}

5) Gramática para la función principal (forma int main())

G = {E → ISM$P_a P_c$; I → int; S → ‘ ’; M → ‘main’; P_a → ‘(‘; P_c → ‘)’}

Código

Se definen variables locales, vectores de cadenas, para clasificar diferentes elementos del código.

```
vector<string> dataType = {"float","int","signed","unsigned"};

vector<string> keywords = {"auto","break","case","char","const","continue","default",
                           "do","double","else","enum","extern","for","goto",

"if","signed","sizeof","static","struct","switch","typedef","union",
                          "void","volatile","while"
                         };
```

```
vector<string> operators = {"+","-","*","/",">","<",">=","<=","||","&&", "!=", "="};
vector<string> numbers = {"0","1","2","3","4","5","6","7","8","9"};
vector<string> parenthesis = {"(",")","()"};
vector<string> brackets = {"[","]"};
vector<string> brace = {"{","}"};
vector<string> specialChar = {";", ",", ".", "%", "#", "!"};
```

En esta función myString.erase, se encargará de eliminar cualquier salto de línea analizado, myString verifica un tipo de dato, un identificador, operador, numero, paréntesis o algún carácter especial y en caso de que no sea ninguno de estos verifica si hay operadores y llama a la función "others" para empezar el análisis.

```
void lexer(string myString)
{
    myString.erase(std::remove(myString.begin(), myString.end(), '\n'), myString.end());
/// Eliminamos posibles saltos de linea
    char chars[] = {'=', '+', '-', '*', '/', '(', ')', ';', ','};
    size_t position = myString.find_first_of(chars);

    if(find(keywords.begin(), keywords.end(), myString) != keywords.end())
        cout<<myString<<" \t Palabra reservada\n";

    if(find(dataType.begin(), dataType.end(), myString) != dataType.end())
        cout<<myString<<" \t Tipo de dato\n";

    else if((isalpha(myString[0]) || myString[0] == '_') && position == string::npos)
/// Verificamos si es una variable
    {
        for (int i = 0; i < myString.length(); i++)
        {
            if (!isalnum(myString[i]) && myString[i] != '_')
            {
                return;
            }
        }
        cout<<myString<<" \t Identificador\n";
    }
```

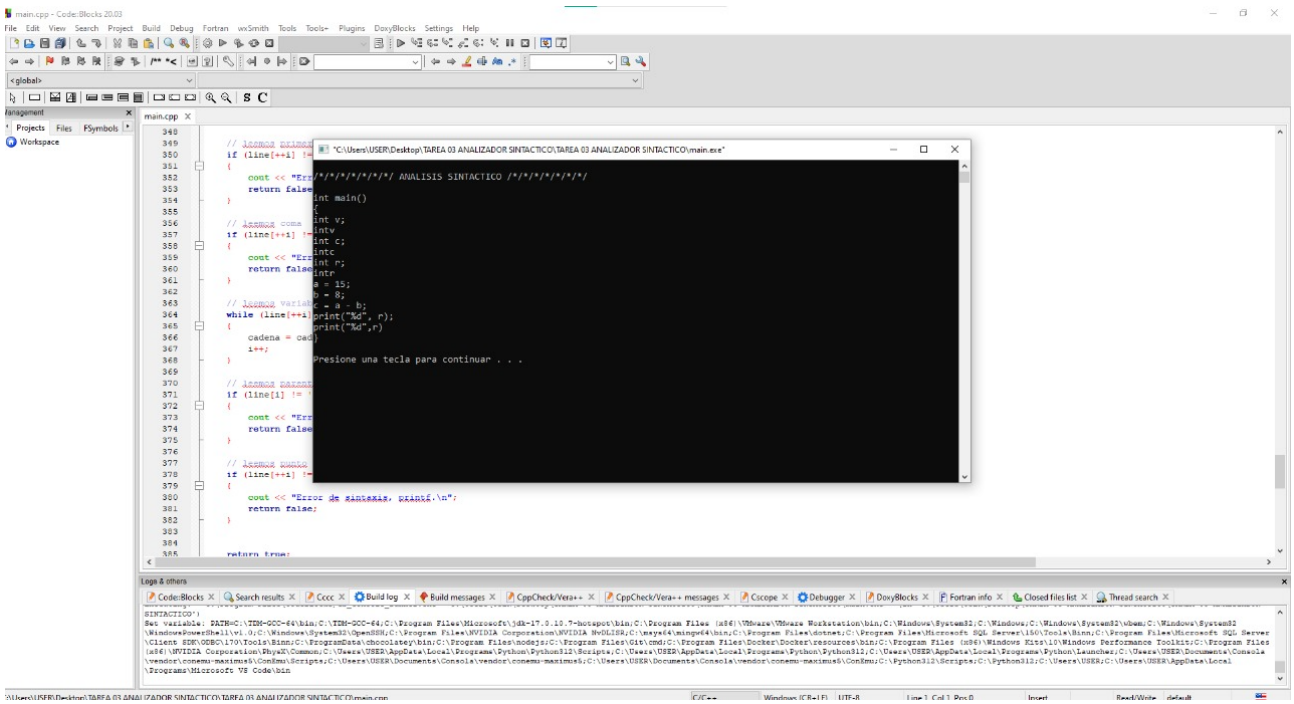
En la función "others" dividiremos la cadena que contiene el operador en "antes del operador " de lado izquierdo y derecho "después del operador, y se llamara a la función "leer" para analizar cada parte de la cadena.

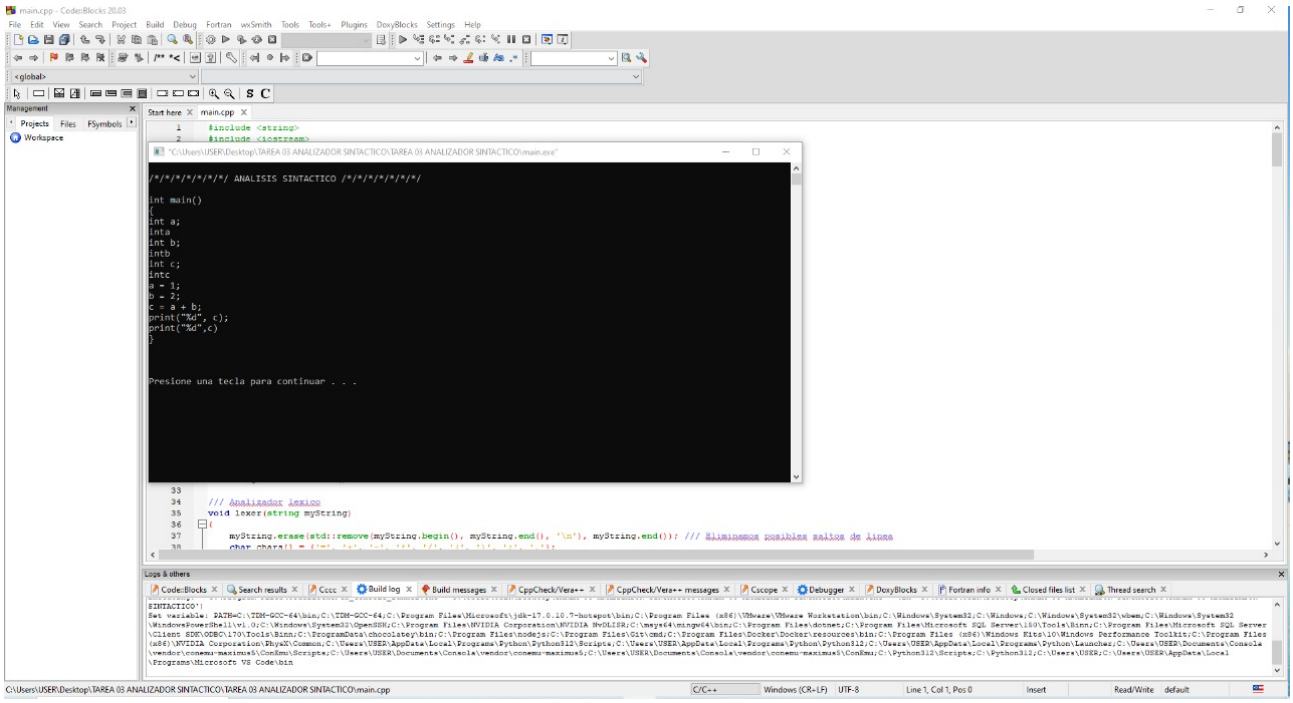
```
void others(string myString, size_t position)
{
```

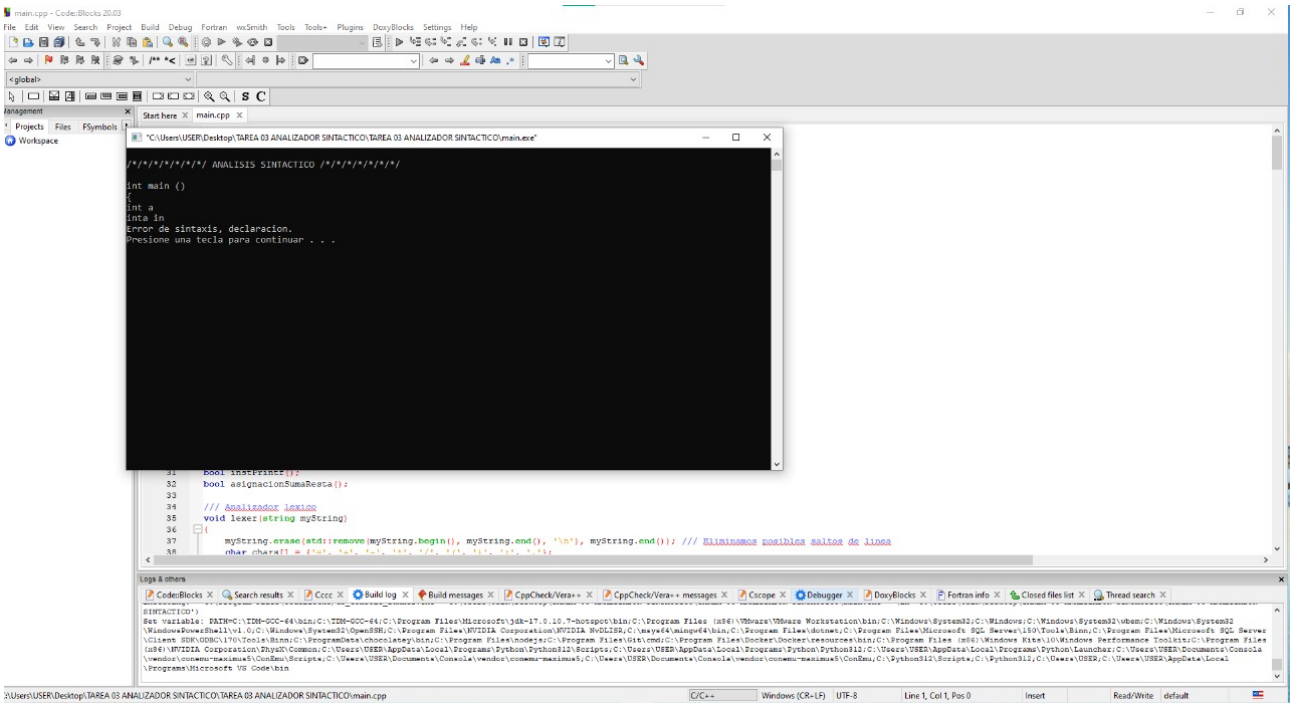
```
    if (position == 1)   /// Si solo hay un caracter antes del signo o caracter (parte izq)
    {
        string newString = string(1, myString[0]);
        lexer(newString);
    }
    else /// Si hay mas de un caracter antes del signo o caracter (parte izq)
        lexer(myString.substr(0, position));

    string newString2 = string(1, myString[position]); /// Tomamos en cuenta el signo o caracter (mid)
    lexer(newString2);

    if (myString.length() > position + 1)
        lexer(myString.substr(position + 1)); /// Tomamos en cuenta el lado derecho del signo o caracter (der)
}
```

En la función principal revisaremos los delimitadores y que la sintaxis sea correcta como el uso de los paréntesis.

3. Resultados




/*/*/*/*/*/*/ ANALISIS SINTACTICO /*/*/*/*/*/*/
Presione una tecla para continuar . . .


/*/*/*/*/*/*/ ANALISIS SINTACTICO /*/*/*/*/*/*/
Error de sintaxis, declaracion.
Presione una tecla para continuar . . .

4. Conclusión

Crear un analizador sintáctico en C++ para procesar fragmentos de código ha sido un paso importante para entender cómo funcionan los compiladores y el análisis de lenguajes de programación. Este analizador ayuda a identificar diferentes partes del código, como palabras clave, tipos de datos y operadores, lo que es crucial para interpretar el código correctamente.
En resumen, esta actividad nos ha permitido mejorar nuestras habilidades de programación y entender mejor los conceptos de análisis y traducción de lenguajes, lo que es valioso para nuestra formación en informática y ingeniería de software.

5. Bibliografía

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compiladores: Principios, técnicas y herramientas* (2da ed.). Pearson Education.