

**CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E
INGENIERÍAS (CUCEI)**

DIVISIÓN DE ELECTRÓNICA Y COMPUTACIÓN

DEPARTAMENTO DE CIENCIAS COMPUTACIONALES

Carrera: Ingeniería en Computación

Nombre Materia: Traductores de Lenguaje II

Profesor: Valdes López Julio Esteban

SECCIÓN: Do7

Nombre alumnos:

- López Arellano Ricardo David
- Valle Escoto Ariel Humberto



Practica 1

Fecha de entrega: 18/09/2023

Introducción:

Para el desarrollo de la práctica requerimos usar las reglas de producción utilizadas en la tarea

Expr	→ Expr + Term Expr - Term Term
Term	→ Term * Factor Term / Factor Factor
Factor	→ Dígito (Expr)

Sin embargo, estas reglas cuentan con recursividad por la izquierda, lo que causa ambigüedad e imposibilita su implementación con un analizador sintáctico de este tipo, por lo que fue necesario eliminar la recursividad a la izquierda, adaptando las reglas de la siguiente manera:

Expr	→ Term Expr'
Expr'	→ + Term Expr' - Term Expr' ε
Term	→ Factor Term'
Term'	→ * Factor Term' / Factor Term' ε
Factor	→ Dígito (Expr)

De este modo, ya sin recursividad fue posible comenzar el desarrollo de la práctica 1

Desarrollo:

Decidí seguir un desarrollo por etapas aunque no estaba seguro de lo que debía incluir el programa, ya que las especificaciones de la práctica no son precisas. Primero decidí utilizar el lenguaje C++ porque es el lenguaje con el que tengo mayor experiencia.

Hice una clase Token para administrar los caracteres individuales de la cadena de entrada

```
class Token
{
public:
    Token(char type, int val = 0) : tipo(type), valor(val) {}

    char getType()
    {
        return tipo;
    }

    int getVal()
    {
        return valor;
    }

private:
    char tipo;
    int valor;
};
```

Donde el tipo indica si es un símbolo cuál es ('+', '-', '*', '/', '()') o si se trataba de un número el carácter sería 'n' y su valor sería almacenado en el espacio correspondiente.

Después creé la clase “Lexer” para procesar la cadena y almacenar el número adecuado en caso necesario en la clase Token.

```
class Lexer
{
public:
    Lexer(string input) : entrada(input), posicion(0) {}

    Token getNextToken()
    {
        char currentChar = getCurrentChar();

        if (isdigit(currentChar))
        {
            int value = currentChar - '0';
            avanza();
            while (isdigit(getCurrentChar()))
            {
                value = value * 10 + (getCurrentChar() - '0');
                avanza();
            }
            return Token('n', value);
        }
        // Si el caracter actual es un operador o parentesis, devolver el token correspondiente
        else if (currentChar == '+')
        {
            avanza();
            return Token('+');
        }
        else if (currentChar == '-')
        {
            avanza();
            return Token('-');
        }
        else if (currentChar == '*')
        {
            avanza();
            return Token('*');
        }
    }
};
```

Ésta clase también se encarga de almacenar la posición del token relativa a la cadena y avanzar sobre ésta:

```
private:
    string entrada;
    int posicion;
    char getCurrentChar()
    {
        if (posicion >= entrada.length())
        {
            return 'x';
        }
        else
        {
            return entrada[posicion];
        }
    }
    void avanza()
    {
        posicion++;
    }
};
```

La clase “Parser” es donde tuve más dificultades pues no estuve seguro de cómo traducir la jerarquía de las reglas de producción en el código:

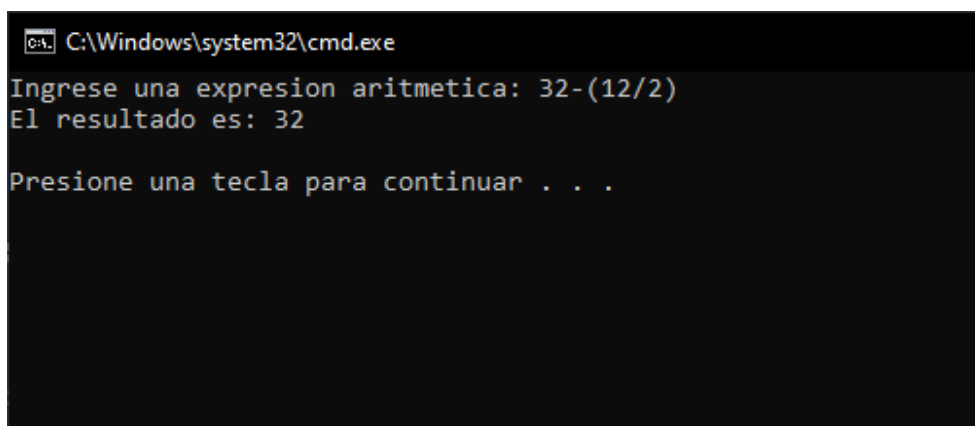
```
class Parser
{
public:
    Parser(Lexer &lexer)
        : lexer_(lexer), currentToken_(lexer.getNextToken())
    {
    }

    int expression()
    {
        int result = termino();

        while (currentToken_.getType() == '+' || currentToken_.getType() == '-')
        {
            if (currentToken_.getType() == '+')
            {
                consume('+');
                result += termino();
            }
            else
            {
                consume('-');
                result -= termino();
            }
        }

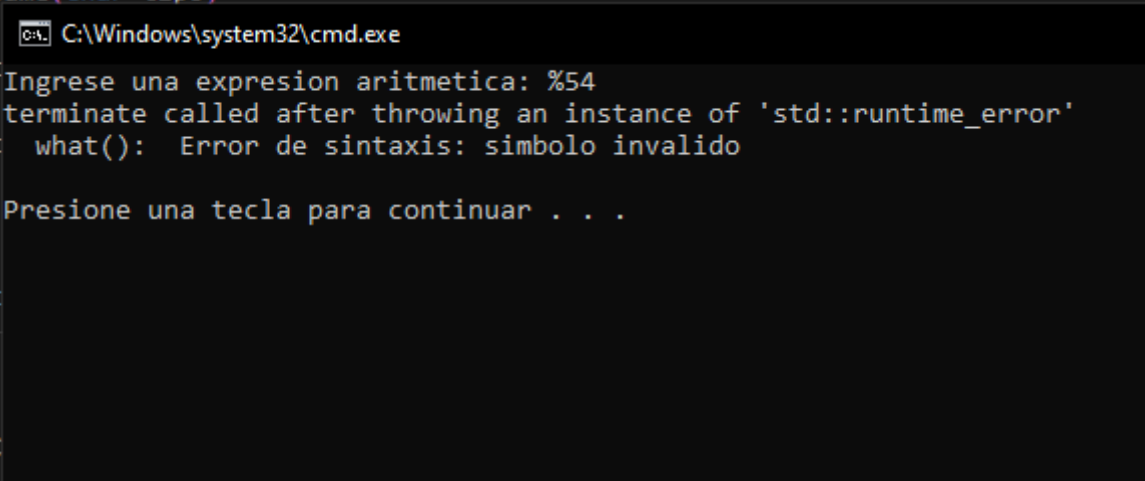
        return result;
    }
}
```

Intenté seguir el orden predeterminado y realizar las operaciones correspondientes pero el código tiene un error en alguna parte dentro de ésta clase que no pude identificar, por lo que el código no realiza las operaciones necesarias.



```
C:\Windows\system32\cmd.exe
Ingrese una expresion aritmetica: 32-(12/2)
El resultado es: 32
Presione una tecla para continuar . . .
```

Lo que sí hace correctamente es identificar cuando una cadena no es válida:



```
C:\Windows\system32\cmd.exe
Ingrese una expresion aritmetica: %54
terminate called after throwing an instance of 'std::runtime_error'
  what():  Error de sintaxis: simbolo invalido

Presione una tecla para continuar . . .
```

Código fuente:

Main.cpp:

```
#include <iostream>
#include <string>

using namespace std;

class Token
{
public:
    Token(char type, int val = 0) : tipo(type), valor(val) {}

    char getType()
    {
        return tipo;
    }

    int getVal()
    {
        return valor;
    }

private:
    char tipo;
    int valor;
};

class Lexer
{
public:
    Lexer(string input) : entrada(input), posicion(0) {}

    Token getNextToken()
    {
        char currentChar = getCurrentChar();

        if (isdigit(currentChar))
        {
            int value = currentChar - '0';
            avanza();
            while (isdigit(getCurrentChar()))
            {
                value = value * 10 + (getCurrentChar() - '0');
                avanza();
            }
            return Token('n', value);
        }
    }
}
```

```

        else if (currentChar == '+')
        {
            avanza();
            return Token('+');
        }
        else if (currentChar == '-')
        {
            avanza();
            return Token('-');
        }
        else if (currentChar == '*')
        {
            avanza();
            return Token('*');
        }
        else if (currentChar == '/')
        {
            avanza();
            return Token('/');
        }
        else if (currentChar == '(')
        {
            avanza();
            return Token('(');
        }
        else if (currentChar == ')')
        {
            avanza();
            return Token(')');
        }
        else
        {
            throw runtime_error("Error de sintaxis: simbolo invalido");
        }
    }

private:
    string entrada;
    int posicion;
    char getCurrentChar()
    {
        if (posicion >= entrada.length())
        {
            return 'x';
        }
        else
        {
            return entrada[posicion];
        }
    }
}

```



```

void avanza()
{
    posicion++;
}
};

class Parser
{
public:
    Parser(Lexer &lexer)
        : lexer_(lexer), currentToken_(lexer.getNextToken())
    {
    }

    int expresion()
    {
        int result = termino();

        while (currentToken_.getType() == '+' || currentToken_.getType()
== '-')
        {
            if (currentToken_.getType() == '+')
            {
                consume('+');
                result += termino();
            }
            else
            {
                consume('-');
                result -= termino();
            }
        }

        return result;
    }

    int termino()
    {
        int result = factor();

        while (currentToken_.getType() == '*' || currentToken_.getType()
== '/')
        {
            if (currentToken_.getType() == '*')
            {
                consume('*');
                result *= factor();
            }
            else
            {

```

```

        consume('/');
        result /= factor();
    }
}

return result;
}

int factor()
{
    if (currentToken_.getType() == '(')
    {
        consume('(');
        int result = expresion();
        consume(')');
        return result;
    }
    else if (currentToken_.getType() == 'n')
    {
        int result = currentToken_.getVal();
        consume('n');
        return result;
    }
    else
    {
        throw runtime_error("Factor invalido");
    }
}

private:
    void consume(char tipo)
    {
        if (currentToken_.getType() == tipo)
        {
            currentToken_ = lexer_.getNextToken();
        }
        else
        {
            throw runtime_error("Token inesperado: " +
currentToken_.getVal());
        }
    }
    Lexer lexer_;
    Token currentToken_;
};

int main()
{
    string input;
    cout << "Ingrese una expresion aritmetica: ";

```

```
getline(cin, input);

Lexer lexer(input);
Parser parser(lexer);

try
{
    int resultado = parser.expresion();
    cout << "El resultado es: " << resultado << endl;
}
catch (const exception &ex)
{
    cerr << "Error: " << ex.what() << endl;
}
return 0;
}
```

Conclusiones:

Este programa aún tiene mucho trabajo por delante, espero recibir indicaciones más detalladas para la próxima ocasión y así desarrollar un programa funcional.