Nombre: Dana Carolina Ramírez Velázquez

Código: 220286547

Materia: Traductores de Lenguaje 2

Actividad: Practica 4

Fecha: 06/11/22

Introducción:

Las indicaciones para elaborar esta práctica era, utilizando la gramática de la practica 2, mejorar el analizador léxico para que pudiera aceptar número enteros de más de un dígito y también identificadores.

En esta práctica implementé por primera vez mi clase de Analizador Léxico que reutilizaré en mis siguientes prácticas, adecuandolo a la gramática en cuestión.

Para elaborar mi Analizador Léxico me basé en una lista de Tokens, con lo que evaluaré si los datos introducidor por el usuario son válidos de mi gramática, aún si revisar si la sintaxis es la correcta.

Desarrollo:

En la clase AnalizadorLexico encontramos:

- Enum class TokenType, donde se encuentran los nombres de los tokens de mi gramática.
- La lista de Tokens que se irá llenando conforme vamos leyendo por primera vez la entrada del usuario.
- Un booleano que nos ayudará a despues aplicar la notación postfija si es que se encontró algún identificador o de lo contrario, realizar las operaciones matemáticas en caso de contar con solamente numeros y operadores.
- Tenemos las funciones que leen, evaluan y separan la información que fue recibida.

AnalizadorLexico.h > AnalizadorLexico > _token

```h
#ifndef ANALIZADORLEXICO_H
#define ANALIZADORLEXICO_H
#include <iostream>
#include <string>
#include <vector>
#pragma once
#include <fstream>

using namespace std;
enum class TokenType {
  NUMERO,
  IDENTIFICADOR,
  PARENTESIS,
  PARENTESIS_END,
  OPERADOR_TERM,
  OPERADOR_FACT,
};

class AnalizadorLexico {
 private:
  fstream Archivo;
  string linea;
  string _token;
  bool postfija = false;

  void leer();

  bool isNumber();
  void separate();
  void evaluate();

 public:
  struct Token {
    TokenType tipo;
    string value;
  };
  AnalizadorLexico();
  vector<Token> List;
  void start(string file);
  bool isPostfija() { return postfija; }
};

#endif
```

Las siguientes funciones son las básicas de iniciar, leer, separar, y verificar si un dato de entrada es numero.

```
5   void AnalizadorLexico::start(string file) {
6     Archivo.open(file, ios::in);
7
8     if (!Archivo.is_open()) {
9       throw std::runtime_error("tipo failed");
10    }
11    leer();
12  }
13
14  void AnalizadorLexico::leer() {
15    while (!Archivo.eof()) {
16      getline(Archivo, linea);
17      separate();
18    }
19  }
20
21  void AnalizadorLexico::separate() {
22    int i = 0;
23    while (linea[i] != '\000') {
24      while (isdigit(linea[i]) || isalpha(linea[i])) {
25        _token += linea[i];
26        i++;
27      }
28      if (_token == "") {
29        _token += linea[i];
30        i++;
31      }
32      evaluate();
33    }
34  }
35
36  bool AnalizadorLexico::isNumber() {
37    for (char const &c : _token) {
38      if (std::isdigit(c) == 0) return false;
39    }
40    return true;
41  }
```

Y ahora la más importante, la que evalúa con ayuda de las funciones anteriores.

```
43  void AnalizadorLexico::evaluate() {
44    Token nuevo;
45    if (_token == "(") {
46      nuevo.tipo = TokenType::PARENTESIS;
47    } else if (_token == ")") {
48      nuevo.tipo = TokenType::PARENTESIS_END;
49    } else if (_token == "+" || _token == "-") {
50      nuevo.tipo = TokenType::OPERADOR_TERM;
51    } else if (_token == "*" || _token == "/") {
52      nuevo.tipo = TokenType::OPERADOR_FACT;
53    } else if (isNumber()) {
54      nuevo.tipo = TokenType::NUMERO;
55    } else if (isalpha(_token[0])) {
56      nuevo.tipo = TokenType::IDENTIFICADOR;
57      postfija = true;
58    } else {
59      throw std::runtime_error("evaluation failed");
60    }
61
62    nuevo.value = _token;
63    List.insert(List.end(), nuevo);
64    _token.clear();
65  }
66
```

Podemos ver que desde aquí empezamos a mandar mensajes de error en caso de que la entrada no sea la información que se esperaba, asímismo podemos ver que por temas de practicidad, lo que recibe de entrada es un archivo de texto.

De parte del analizador sintáctico no hubo muchos cambios, más que separamos el análisis léxico que en prácticas anteriores lo teníamos fusionado.

C AnalizadorSintactico.h > AnalizadorSintactico > term()

```h
#ifndef ANALIZADORSINTACTICO_H
#define ANALIZADORSINTACTICO_H
#include <iostream>
#include <string>

#include "AnalizadorLexico.h"

#pragma once

using namespace std;

class AnalizadorSintactico {
 public:
  explicit AnalizadorSintactico(const string &cadena);
  int get_cadena() { return reglas; }

 private:
  AnalizadorLexico Tokens;
  int reglas;
  string m_cadena;
  string sem_cadena;
  bool postfija = false;
  int i = 0;

  void coincidir(const TokenType &token);
  string term();
  string fact();
  string Resto_expr(string resto_her);
  string Resto_term(string resto_her);
  string digito();
  string expr();
};

#endif

```

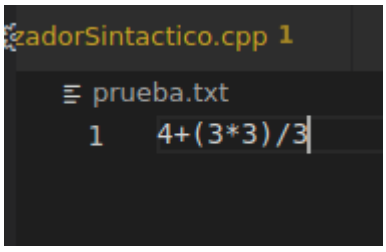
Uno de los cambios realizados fue que ahora en lugar de comparar con algún string, utilizamos el mismo enum class de TokenType que implementamos en el Analizador Léxico.

```
27  string AnalizadorSintactico::term() { return (Resto_term(fact())); }
28
29  string AnalizadorSintactico::Resto_expr(string resto_her) {
30    if (Tokens.List[i].tipo == TokenType::OPERADOR_TERM) {
31      string op = Tokens.List[i].value;
32      coincidir(TokenType::OPERADOR_TERM);
33      if (!postfija) {
34        if (op == "+") {
35          int r = stoi(resto_her) + stoi(term());
36          return (Resto_expr(to_string(r)));
37        } else if (op == "-") {
38          int r = stoi(resto_her) - stoi(term());
39          return (Resto_expr(to_string(r)));
40        }
41      } else {
42        return (Resto_expr(resto_her + term() + op));
43      }
44    } else {
45      return (resto_her);
46    }
47  }
48
49  string AnalizadorSintactico::Resto_term(string resto_her) {
50    if (Tokens.List[i].tipo == TokenType::OPERADOR_FACT) {
51      string op = Tokens.List[i].value;
52      coincidir(TokenType::OPERADOR_FACT);
53      if (!postfija) {
54        if (op == "/") {
55          int t = stoi(resto_her) / stoi(fact());
56          return (Resto_term(to_string(t)));
57        } else if (op == "*") {
58          int t = stoi(resto_her) * stoi(fact());
59          return (Resto_term(to_string(t)));
60        }
61      } else {
62        return (Resto_term(resto_her + fact() + op));
63      }
64    }
65    return (resto_her);
66  }
67
```

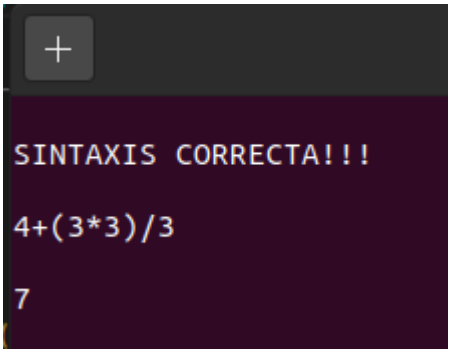
```
68  string AnalizadorSintactico::fact() {
69    string fact = "";
70    if (Tokens.List[i].tipo == TokenType::PARENTESIS) {
71      coincidir(TokenType::PARENTESIS);
72      fact = expr();
73      coincidir(TokenType::PARENTESIS_END);
74    } else {
75      fact = digito();
76    }
77    return fact;
78  }
79
80  string AnalizadorSintactico::digito() {
81    if (Tokens.List[i].tipo == TokenType::IDENTIFICADOR) {
82      coincidir(TokenType::IDENTIFICADOR);
83    } else {
84      coincidir(TokenType::NUMERO);
85    }
86    return (Tokens.List[i - 1].value);
87  }
88
89  string AnalizadorSintactico::expr() { return Resto_expr(term()); }
90
```

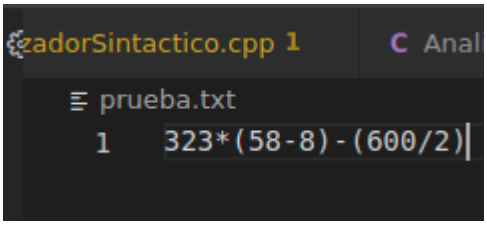
Resultados:

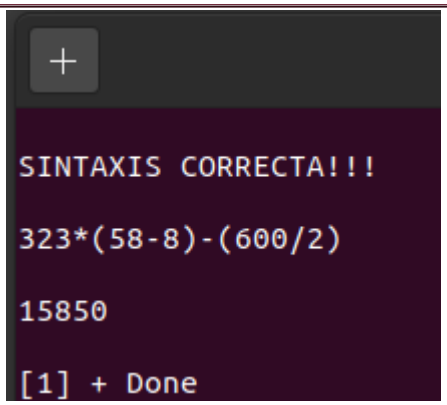
Intendando con la siguiente entrada:

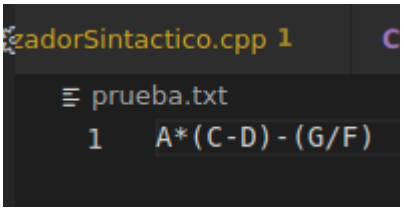


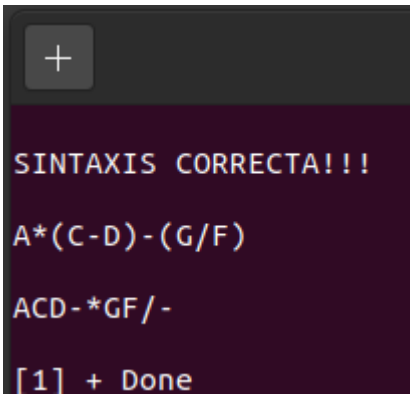
El resultado es el siguiente:

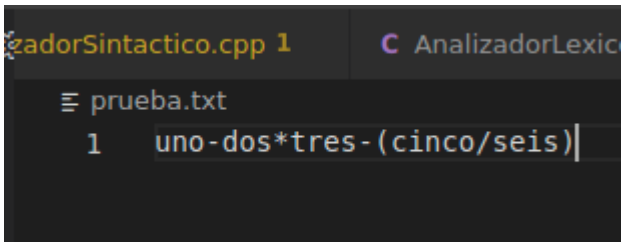


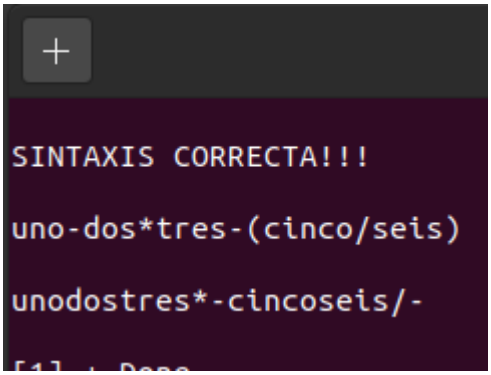






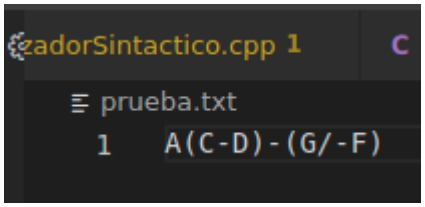


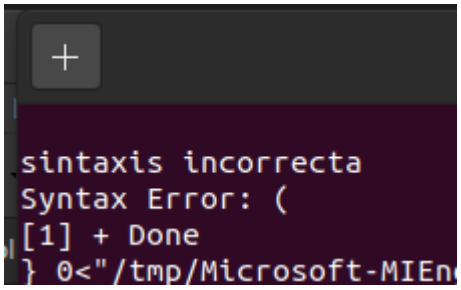


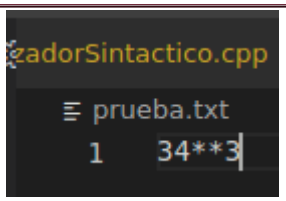


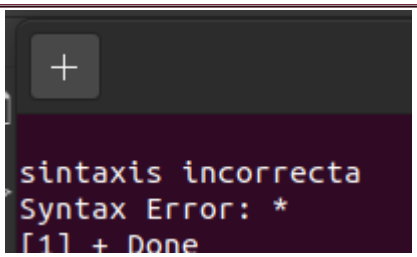
Entrada Erronea:

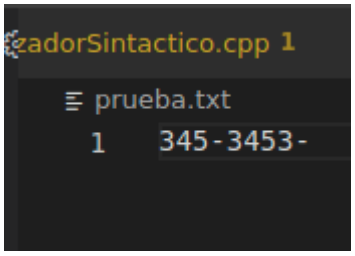
Mensaje resultado:

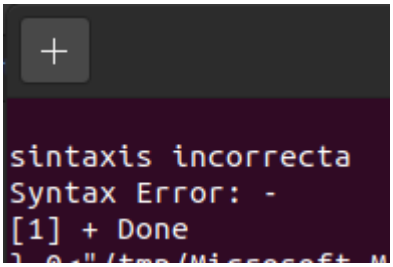












Conclusiones:

Cuando se tiene organizado el Analizador Léxico y el Analizador Sintáctico en dos partes distintas del programa, no facilita mucho la detección de errores y hay menos margen de error a la hora de programar un analizador para alguna gramática. También es importante resaltar la parte de la notación postfija, que con un indicador que le diga al programa cual es la situación, se pueda elegir las reglas semánticas con las que se debe trabajar.