Nombre: Dana Carolina Ramírez Velázquez

Código: 220286547

Materia: Traductores de Lenguaje 2

Actividad: Practica 5

Fecha: 14/11/22

Introducción:

Las indicaciones para elaborar esta práctica era, implementar la gramática de la gramática de la página 360 del libro del dragón en un analizador predictivo y generar el GDA como salida.

Desarrollo:

La gramática era la siguiente:

	PRODUCCIÓN	REGLAS SEMÁNTICAS
1)	$E \rightarrow E_1 + T$	$E.nodo = \textbf{new}\ Nodo('+', E_1.nodo, T.nodo)$
2)	$E \rightarrow E_1 - T$	$E.nodo = \textbf{new}\ Nodo('-', E_1.nodo, T.nodo)$
3)	$E \rightarrow T$	$E.nodo = T.nodo$
4)	$T \rightarrow (E)$	$T.nodo = E.nodo$
5)	$T \rightarrow \textbf{id}$	$T.nodo = \textbf{new}\ Hoja(\textbf{id}, \textbf{id}.entrada)$
6)	$T \rightarrow \textbf{num}$	$T.nodo = \textbf{new}\ Hoja(\textbf{num}, \textbf{num}.val)$

En la clase Analizador Lexico 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 > Valor(Token)

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

#include "Hoja.h"

using namespace std;
enum class TokenType {
  NUMERO,
  IDENTIFICADOR,
  MAS,
  MENOS,
  PARENTESIS,
  PARENTESIS_END,
};

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;
    int id;
  };

  int total = 0;
  AnalizadorLexico();
  vector<Hoja> Tabla;
  vector<Token> List;
  string Valor(Token _token);
  int LookTable(string valor);
  void start(string file);
  bool isPostfija() { return postfija; }
};
```

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.

```
53  void AnalizadorLexico::evaluate() {
54    Token nuevo;
55    if (_token == "(") {
56      nuevo.tipo = TokenType::PARENTESIS;
57      nuevo.id = -1;
58      total++;
59    } else if (_token == ")") {
60      nuevo.tipo = TokenType::PARENTESIS_END;
61      nuevo.id = -1;
62      total++;
63
64    } else if (_token == "+") {
65      nuevo.tipo = TokenType::MAS;
66      nuevo.id = -1;
67      total++;
68
69    } else if (_token == "-") {
70      nuevo.tipo = TokenType::MENOS;
71      nuevo.id = -1;
72      total++;
73
74    } else if (isNumber()) {
75      nuevo.tipo = TokenType::NUMERO;
76      nuevo.id = LookTable(_token);
77      total++;
78
79    } else if (isalpha(_token[0])) {
80      nuevo.tipo = TokenType::IDENTIFICADOR;
81      nuevo.id = LookTable(_token);
82      total++;
83      postfija = true;
84    } else {
85      throw std::runtime_error("not valid token found");
86    }
87
88    List.insert(List.end(), nuevo);
89    _token.clear();
90  }
```

En esta parte es donde empezamos con las partes de la tabla de simbolos, pues el id es la entrada en la tabla de simbolos, el -1 indica que el toquen no pertenece a la tabla, los demás numeros son el indice de nuestro arreglo “tabla de simbolos”. De esta manera,

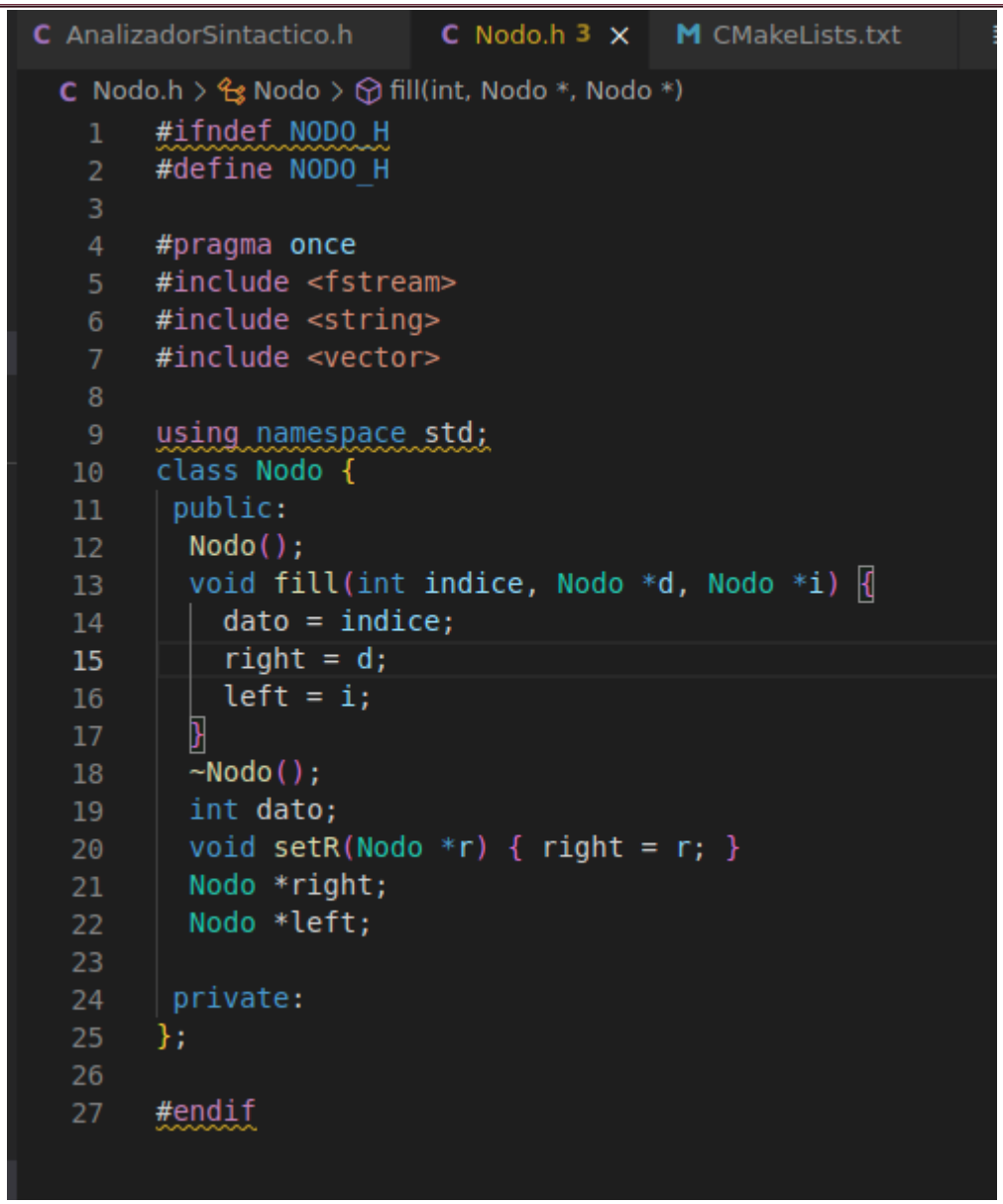
cuando termina de analizar todos los tokens, habrá realizado la tabla de simbolos también.

Mi tabla de simbolos es un vector de Hojas, esta es la clase “hoja”.

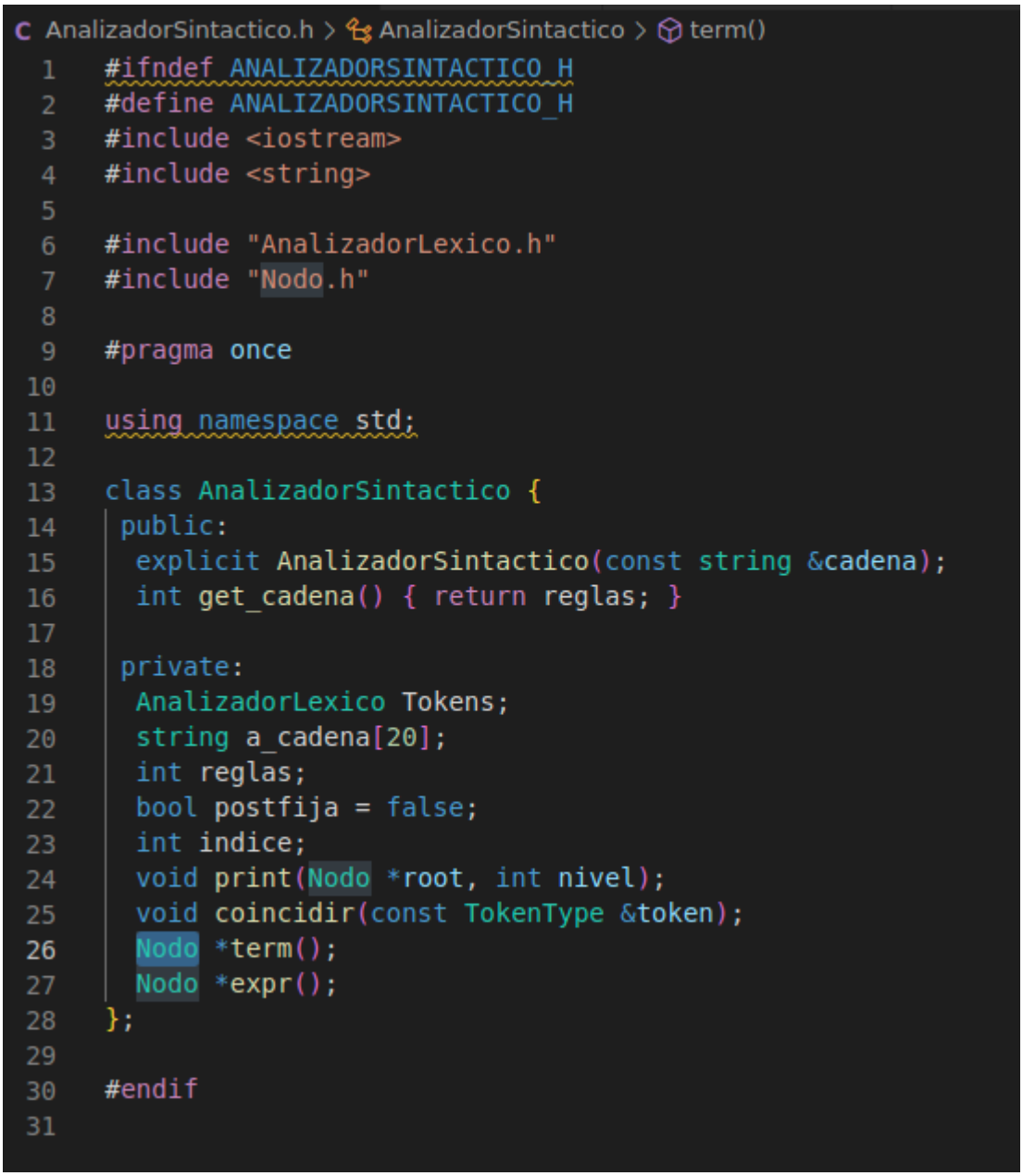
Hoja.h > ...

```
1   #ifndef HOJA_H
2   #define HOJA_H
3
4   #pragma once
5   #include <string>
6   #include <vector>
7
8   using namespace std;
9   class Hoja {
10   public:
11    Hoja(string id, int entrada) {
12      val = id;
13      index = entrada;
14    }
15    ~Hoja();
16    string val;
17    int index;
18
19   private:
20  };
21
22  #endif
```

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.



Esta es la clase Nodo que utilizaremos para el GDA al final del programa.



En la clase de Analizador sintáctico tenemos la declaración de las funciones de la gramática y el resto del código (las definiciones de las funciones) lo adjuntaré al final del documento.

prueba.txt

```
1   d+e-(a*c)+a
```

```
sintaxis incorrecta not valid token found
[1] + Done                    "/usr/bin/
nivel: 1 lado izquierdo: +
nivel: 2 lado izquierdo: +
nivel: 3 lado izquierdo: d
nivel: 3 lado derecho: e
nivel: 2 lado derecho: -
nivel: 3 lado izquierdo: a
nivel: 3 lado derecho: c
nivel: 1 lado derecho: a

nivel 0 +
nivel 1  + a
nivel 2  + -
nivel 3  d e a c

SINTAXIS CORRECTA!!!
```

Resultados:

Conclusiones:

Generar un GDA al final de una gramática es el paso principal para comenzar a crear el código intermedio de un compilador. En esta práctica se reutilizaron las clases de Analizador Léxico y Analizador sintáctico de la práctica anterior adecuando las partes que tienen que ser personalizadas para cada gramática, también se implementó la creación del árbol durante el proceso del análisis sintáctico.

Código identado: