Министерство науки и высшего образования Российской Федерации

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Лабораторная работа №3

по дисциплине

«Языки программирования и методы трансляции»

Факультет прикладной математики и информатики

Группа ПМ-63

Студенты Майер В.

Шепрут И.

Преподаватели Еланцева И.Л.

Вариант 8

Новосибирск 2019

1. Цель работы

Изучить табличные методы синтаксического анализа. Получить представление о методах диагностики и исправления синтаксических ошибок. Научиться проектировать синтаксический анализатор на основе табличных методов.

1. Задание

Подмножество языка С++ включает:

* данные типа int, float, struct из элементов указанных типов
* инструкции описания переменных
* операторы присваивания в любой последовательности
* операции +, -, \*, = =, !=, <, >.

В соответствии с выбранным вариантом заданий к лабораторным работам реализовать синтаксический анализатор с использованием одного из табличных методов (LL-, LR – метод, метод предшествования).

Этапы проектирования синтаксического анализатора:

1. сконструировать КС-грамматику в соответствии с вариантом задания;
2. в случае несоответствия постоянной грамматики требованиям выбранного табличного метода разбора следует провести эквивалентные преобразования грамматики либо выбрать другой метод разбора;
3. построить таблицу разбора и запрограммировать драйвер, реализующий работу с этой таблицей.

Исходные данные – файл токенов таблицы лексем.

Результатом работы синтаксического анализатора является:

* синтаксическое дерево или постфиксная запись;
* файл сообщений об ошибках. В лабораторной работе необходимо реализовать возможности табличного метода по диагностике и исправлению синтаксических ошибок в исходной программе.
* Так же результатом работы программы будет файл предуплеждений.

1. Грамматика языка

*Начальный символ*

S =: <Struct>, “void”, “main”, ”(“, ”)”, ”{“, <program>,”}”

*Типы*

type =: “int” | “float” | indS // indS – Название структуры, ind – индификатор

*Операции*

oper =: “+” | “-” | “\*” | “!=” | “==” | “<” | “>”

*Структура*

Struct =: {“struct”, indS, “{”, <annou\_struct>, “}”, “;”, <Struct> } | eps // eps – ничего

*Объявление полей структуры*

annou\_struct =: {“type”, ind, <nameS>, “;”, <annou\_struct> } | eps

*Случай если через запятую*

nameS =: {“,”, “ind”, <nameS>} | eps

*Программа main*

program =: {“type”, <name>, “;”, <program>}| {“<name>”, “;”,<program>} | eps

*Объявление переменной*

name =: “ind”, <eq>, {“,”, <name>} | eps

*Задание значение переменной (либо задается либо нет)*

<eq> =: <eq1> |eps

*Задается значение*

<eq1> =: “=”, {“ind”, <continue> | <eq1>} | {“const”| <continue>} | {“(”, <expr>, “}”, continue}

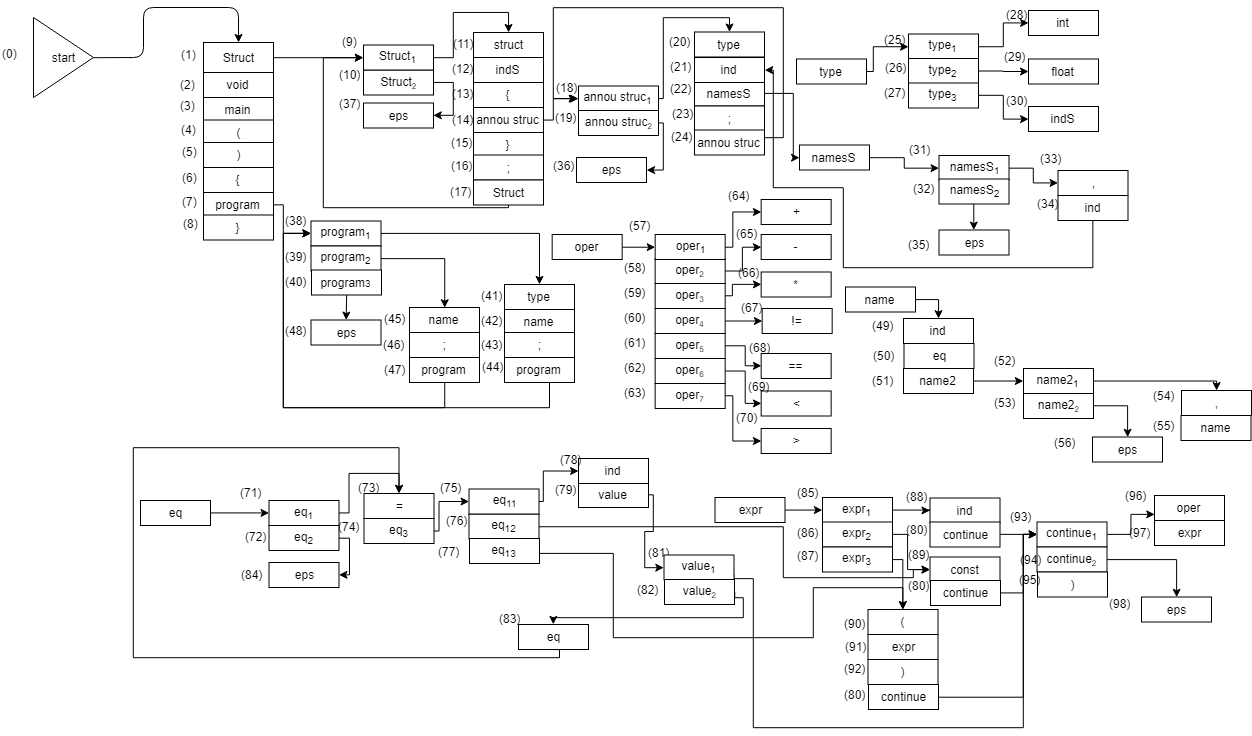
*Продолжение составного выражения*

continue =: <oper>, <expr> | eps

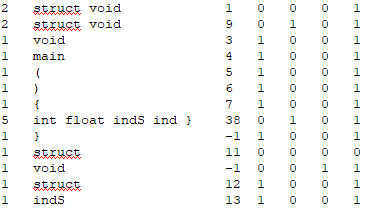
*Выражение*

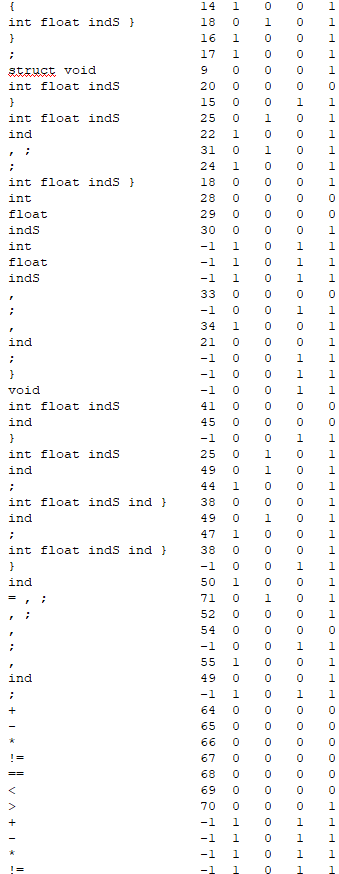
expr =: {““ind”, <continue>”} | {“const”| <continue>} | {“(”, <expr>, “}”, continue}

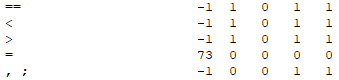
1. **Схема разбора**

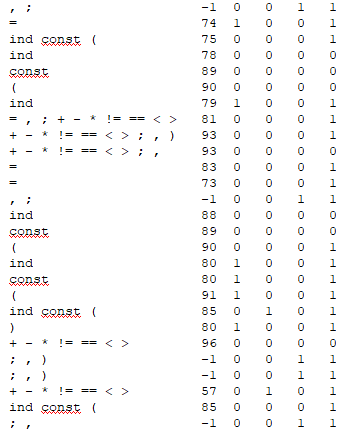


1. **Таблица разбора**

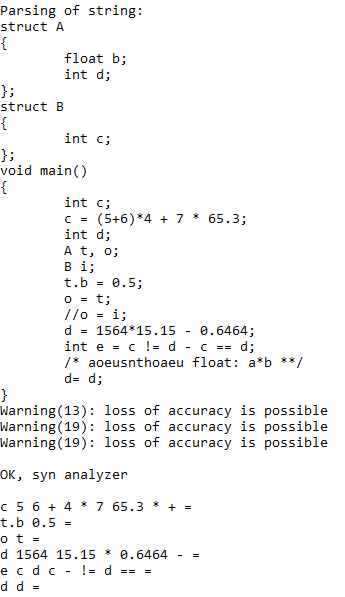




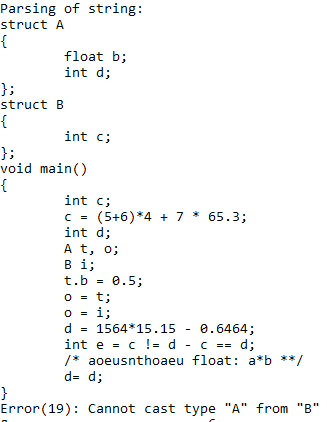




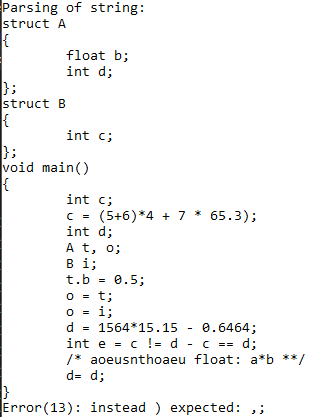
1. **Тесты**
2. *Верный исходный код*



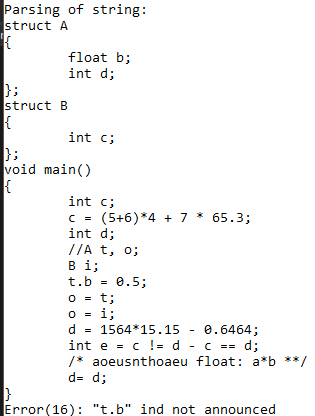
1. *Прирванивание разных структур*



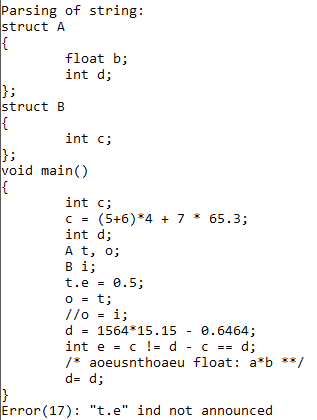
1. *Лишняя закрывающаяся скобка*



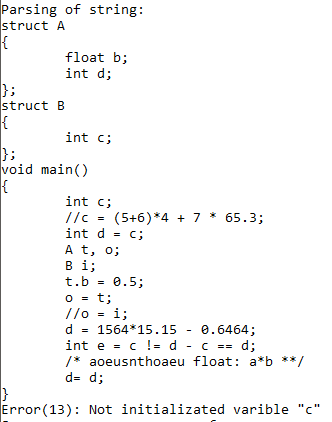
1. *Использование необъявленной переменной*



1. *Использование необъявленного поля структуры*



1. *Использование не проинцелезированной переменной*



1. **Текст программы**

/\*test\_syn\_analyzer.cpp\*/

#include <iostream>

#include <iomanip>

#include <lex\_analyzer.h>

#include <syn\_analyzer.h>

int main() {

std::string file = "struct A\n{\n\tfloat b;\n\tint d;\n};\nstruct B\n{\n\tint c;\n};\nvoid main()\n{\n\tint c;\n\tc = (5+6)\*4 + 7 \* 65.3;\n\tint d;\n\tA t, o;\n\tB i;\n\tt.b = 0.5;\n\to = t;\n\t//o = i;\n\td = 1564\*15.15 - 0.6464;\n\tint e = c != d - c == d;\n\t/\* aoeusnthoaeu float: a\*b \*\*/\n\td= d;\n}";

file += ' ';

std::cout << "Parsing of string: \n" << file << std::endl;

try

{

Tables tables;

auto tokens = parse(file, tables);

string path = "syn\_table.txt";

auto result = Analyzer(tokens, tables, path);

std::cout << std::endl << "OK, syn analyzer" << std::endl << std::endl;

for (auto& i : result)

{

for (auto& j : i)

std::cout << tables.getStr(j) << " ";

std::cout << std::endl;

}

}

catch (string &e)

{

std::cerr << e << endl;

}

system("pause");

}

/\*syn\_analyzer.h\*/

#include <vector>

#include <string>

#include <tables.h>

std::vector<std::vector<Token>> Analyzer(const std::vector<Token>& tokens, Tables& tables, string path);

/\*syn\_analyzer.cpp\*/

#include <string>

#include <syn\_analyzer.h>

#include <fstream>

#include <iostream>

#include <optional>

#include <stack>

class SynTable

{

public:

SynTable(const std::string& path, Tables& tables); //Конструктор

bool /\*std::optional<vector<Token>>\*/ Next(Token); //Обработака следующего токена

std::vector<Token> ret;

std::vector<string> warning;

private:

struct Row //Строка лексической таблицы

{

std::vector<string> term;

int jmp;

bool act;

bool stk;

bool ret;

bool err;

};

string nameStr; //Имя структуры, которая обрабатывается

bool CheckTrashToken(Token token); //Проверка ненужных токенов

int nstr; //Номер строки у токенов

void Error(std::string);

void Warning(std::string);

bool treestart; //Число элементов в дереве

Token struc;

Type type; //Тип следующих функций

Tables& tables; //Хэш таблицы

int state; //Текущее состояние

std::vector<int> stack\_state; //Стек следующих состояний

std::vector<Row> syn\_table; //Синтаксическая таблица

std::stack<Token> oper; //Операции

int GetPriorOper(Token token) //Получить преоритет опирации

{

auto str = tables.getStr(token);

if (str == "\*")

return 8;

if (str == "+" || str == "-")

return 7;

if (str == "==" || str == "!=" || str == "<" || str == ">")

return 6;

if (str == "(")

return 2;

if (str == "=")

return 1;

}

};

//=============================================================================

//=============================================================================

//=============================================================================

//Считываем таблицу и задаем начальные значения

SynTable::SynTable(const std::string& path, Tables& tables) : tables(tables) {

ifstream itable(path);

if (!itable) Error("Error open file " + path);

int n, m;

int num;

itable >> n;

syn\_table.resize(n);

stack\_state.clear();

state = 0;

type = TYPE\_NONE;

nstr = 1;

treestart = false;

for (size\_t i = 0; i < n; i++)

{

itable >> m;

syn\_table[i].term.resize(m);

for (size\_t j = 0; j < m; j++)

itable >> syn\_table[i].term[j];

itable >> syn\_table[i].jmp;

itable >> syn\_table[i].act;

itable >> syn\_table[i].stk;

itable >> syn\_table[i].ret;

itable >> syn\_table[i].err;

}

itable.close();

}

void SynTable::Error(std::string str)

{

throw "Error(" + std::to\_string(nstr) + "): " + str;

}

void SynTable::Warning(std::string str)

{

warning.push\_back("Warning(" + std::to\_string(nstr) + "): " + str);

}

bool SynTable::CheckTrashToken(Token token)

{

Token tab = tables.find("\t");

Token sce = tables.find(" ");

Token ent = tables.find("\n");

if (token == tab || token == sce) return true;

if (token == ent) { nstr++; return true; }

return false;

}

bool/\*std::optional<vector<Token>>\*/ SynTable::Next(Token token)

{

if (CheckTrashToken(token)) return false;// std::nullopt;

bool f = false;

for (auto& i : syn\_table[state].term)

{

if (i == "indS" && token.table == TABLE\_IDENTIFIERS)

{

auto b = tables.getStr(token); //доработать

if (b[0] >= 'A' && b[0] <= 'Z') { f = true; break; }

}

if (i == "ind" && token.table == TABLE\_IDENTIFIERS) { f = true; break; }

if (i == "const" && token.table == TABLE\_CONSTANTS) { f = true; break; }

if (token == tables.find(i)) { f = true; break; }

}

bool out = false;

int s;

//Если токен есть в списке возможных слов

if (f)

{

//Добавление в стек

if (syn\_table[state].stk)

stack\_state.push\_back(state + 1);

//Созранение положение

s = state;

//Обработка тикущего состояние

switch (state)

{

case 12: //Добавление тип структуры

nameStr = tables.getStr(token);

struc = tables.add(nameStr, TABLE\_STRUCTURES);

break;

case 21: //Добавление поля в таблицу структуры

{

auto& arg = tables.get<Structure>(struc);

Structure::StructElem elem;

string name = tables.getStr(token);

for (auto& i : arg.elems) //Ошибка одинаковые поля в структуре

if (i.name == name) Error("ind \"" + name + "\" was announced in struc \"" + nameStr + "\"");

elem.name = name;

elem.structToken = struc;

elem.type = type;

if (type == TYPE\_STRUCT)

elem.nameStruct = nameStr;

arg.elems.push\_back(elem);

}

break;

case 49: //Объевление переменной

{

auto& arg = tables.get<Identifier>(token);

if (type == TYPE\_STRUCT) //Если тип структура

{

auto& argStruc = tables.get<Structure>(struc); //То всем полям, которые найдем в таблице

for (auto& i : argStruc.elems) //Задаем тип из структуры

{

auto tok = tables.find(tables.getStr(token) + "." + i.name);

if (tok)

{

auto& argtok = tables.get<Identifier>(tok);

argtok.type = i.type;

}

}

arg.nameStruct = nameStr;

}

if (type != TYPE\_NONE)

{ //Задаем тип переменной

if (arg.type != TYPE\_NONE) Error("\"" + tables.getStr(token) + "\" ind was announced"); //Ошибка переменная уже была объявлена

arg.type = type;

}

else

if (tables.get<Identifier>(token).type == TYPE\_NONE) Error("\"" + tables.getStr(token) + "\" ind not announced"); //Ошибка переменная не былоа объявлена

else type = tables.get<Identifier>(token).type, nameStr = tables.get<Identifier>(token).nameStruct;

ret.push\_back(token);

}

break;

//Дерево

case 73: // =

treestart = true;

oper.push(token);

break;

//Структуры

case 88: //id продолжение

{

ret.push\_back(token);

auto& arg = tables.get<Identifier>(token);

if (arg.type == TYPE\_STRUCT) Error("Cannot cast type \"" + arg.nameStruct + "\" in expression");

}

break;

case 89: //Константа

{

auto& arg = tables.get<Constant>(token);

string name = tables.getStr(token);

Type b;

name.find(".") == -1 ? b = TYPE\_INT : b = TYPE\_FLOAT;

if (type == TYPE\_INT && b == TYPE\_FLOAT) Warning("loss of accuracy is possible");

arg.type = b;

ret.push\_back(token);

}

break;

case 90: // (

oper.push(token);

break;

case 92:// )

Token top = oper.top();

oper.pop();

Token scob = tables.find("(");

while (top != scob)

{

ret.push\_back(top);

top = oper.top();

oper.pop();

}

break;

case 93:// Продолжение выражения

if (type == TYPE\_STRUCT) Error("Cannot cast expression to \"" + nameStr + "\"");

break;

case 64:

case 65:

case 66:

case 67: //Операции

case 68:

case 69:

case 70:

while (oper.size() && GetPriorOper(oper.top()) >= GetPriorOper(token))

{

ret.push\_back(oper.top());

oper.pop();

}

oper.push(token);

break;

case 78:

{

//root->right = new Tree<Token>(token);

//treeptr = root->right;

ret.push\_back(token);

auto& arg = tables.get<Identifier>(token);

if (type == TYPE\_STRUCT && arg.type == TYPE\_STRUCT && nameStr != arg.nameStruct) Error("Cannot cast type \"" + nameStr + "\" from \"" + arg.nameStruct + "\"");

if (type == TYPE\_STRUCT && arg.type != TYPE\_STRUCT) Error("Cannot cast type \"" + nameStr + "\ from " + (arg.type == TYPE\_INT ? "\" int\"" : "\" float\""));

if (type != TYPE\_STRUCT && arg.type == TYPE\_STRUCT) Error("Cannot cast type \"" + type /\*(type == TYPE\_INT ? "int\ to" : "float\ to")\*/ + arg.nameStruct + "\"");

}

break;

case 28: //Объявление тип int

type = TYPE\_INT;

break;

case 29: //Объявление тип float

type = TYPE\_FLOAT;

break;

case 30: //Объявление тип структуры

type = TYPE\_STRUCT;

nameStr = tables.getStr(token);

struc = tables.find(nameStr , TABLE\_STRUCTURES);

break;

case 23:

case 46:

case 43: //окончание объявления

//case 53:

type = TYPE\_NONE;

case 54:

if (treestart)

{

treestart = false;

while (oper.size())

{

ret.push\_back(oper.top());

oper.pop();

}

out = true;

}

else

ret.clear();

break;

default:

break;

}

//Конец ветки, возвращение по стеку

if (syn\_table[state].ret)

{

//Если пустой то пишем ошибку(по идее она не случится)

if (!stack\_state.size()) Error("stack going lim"); //Ошибка выход из стека

state = stack\_state.back();

stack\_state.pop\_back();

}

else

state = syn\_table[state].jmp;

//Если это не терм то идем с этим токеном дальше

if (!syn\_table[s].act)

if (Next(token))

out = true;

}

else

//Если токен не найден среди возмодный слов

if (!syn\_table[state].err)//и это не ошибка

{

state++; //переходим на альтернативу

if (Next(token))

out = true;

}

else //Иначе формируем сообщение об ошибке

{

std::string error = "instead "+ tables.getStr(token) + " expected: ";

for (auto& i : syn\_table[state].term)

error += i;

Error(error); //Ошибка встретили не тот токен

}

return out;

}

//-----------------------------------------------------------------------------

std::vector<std::vector<Token>> Analyzer(const std::vector<Token>& tokens, Tables& tables, string path){

std::vector<std::vector<Token>> result;

SynTable syn\_table(path, tables);

for (auto i : tokens)

{

//std::cout << tables.getStr(i) << std::endl;

//Обрабатываем по одному токену

if (syn\_table.Next(i))

{

result.push\_back(syn\_table.ret);

syn\_table.ret.clear();

}

//auto elem = syn\_table.Next(i).value\_or(vector<Token>());

}

for (auto& i : syn\_table.warning) //Вывод предупреждений

std::cout << i << endl;

return result;

}

/\*lex\_analyzer.h\*/

#pragma once

#include <vector>

#include <tables.h>

std::vector<Token> parse(const std::string& str, Tables& tables);

/\*lex\_analyzer.cpp\*/

#include <string>

#include <lex\_analyzer.h>

enum AddType

{

ADD\_IDENTIFIER,

ADD\_CONSTANT,

ADD\_KEYWORD\_OR\_DELIMITER,

IGNORE,

ERROR,

};

enum StateType

{

Start, // Стартовое состояние

// Завершающие состояния, при достижении их, конечный автомат должен завершать свою работу

ADD, // Все считанные символы, начиная со Start, до текущего момента ялвяются лексемой

PUSH, // ADD + Поместить предыдущий считанный символ обратно в ввод

NEXT, // Данная лексема бесполезна для трансляции, и её можно игнорировать

ERR, // Ошибочное состояние

\_IDE, // PUSH, только для таблицы идентификаторов

\_CONS, // PUSH, только для таблицы констант

// Промежуточные состояния

Ide, Ide2,

Com1, Com2, Com3, Com4,

Num1, Num2,

Oper1, Oper2,

};

enum SymbolType

{

SYMBOL, // "\_a-zA-Z"

NUMBER, // "0-9"

DELIM, // " \t;,{}()"

OPER, // "+-<>"

DOT, // "."

EXCL, // "!"

EQUAL, // "="

DIV, // "/"

MUL, // "\*"

EOLN, // "\n"

OTHER,

};

SymbolType getType(char symbol);

// Здесь надо запрогать конечный автомат, который по текущему состоянию и считанному символу возвращает следующее состояние

StateType automatonNext(StateType state, SymbolType symbol);

class Automaton

{

public:

Automaton(const std::string& str);

bool next(void);

bool isEndState(void) const;

AddType getState(void) const;

std::string getStr(void) const;

int getPos(void) const;

int getLine(void) const;

private:

const std::string& str;

int i, n;

std::string sum;

StateType state;

};

//=============================================================================

//=============================================================================

//=============================================================================

//-----------------------------------------------------------------------------

SymbolType getType(char c) {

if (c == '.') return DOT;

if (c == '!') return EXCL;

if (c == '=') return EQUAL;

if (c == '/') return DIV;

if (c == '\*') return MUL;

if (c == '\n') return EOLN;

if (c == '\_' || (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))

return SYMBOL;

if (c >= '0' && c <= '9')

return NUMBER;

if (c == ' ' || c == '\t' || c == ';' || c == ',' || c == '{' || c == '}' || c == '(' || c == ')')

return DELIM;

if (c == '+' || c == '-' || c == '<' || c == '>')

return OPER;

return OTHER;

}

//-----------------------------------------------------------------------------

StateType automatonNext(StateType state, SymbolType sym) {

switch (state) {

case Start: {

if (sym == EQUAL) return Oper1;

if (sym == EXCL) return Oper2;

if (sym == SYMBOL) return Ide;

if (sym == DELIM) return ADD;

if (sym == OPER) return ADD;

if (sym == NUMBER) return Num1;

if (sym == DIV) return Com1;

if (sym == DOT) return ERR;

if (sym == MUL) return ADD;

if (sym == EOLN) return ADD;

return ERR;

} break;

case Ide: {

if (sym == SYMBOL) return Ide;

if (sym == NUMBER) return Ide;

if (sym == DOT) return Ide2;

return \_IDE;

} break;

case Ide2: {

if (sym == SYMBOL) return Ide;

return ERR;

} break;

case Com1: {

if (sym == DIV) return Com2;

if (sym == MUL) return Com3;

return PUSH;

} break;

case Com2: {

if (sym == EOLN) return NEXT;

return Com2;

} break;

case Com3: {

if (sym == MUL) return Com4;

return Com3;

} break;

case Com4: {

if (sym == DIV) return NEXT;

if (sym == MUL) return Com4;

return Com3;

} break;

case Num1: {

if (sym == NUMBER) return Num1;

if (sym == DOT) return Num2;

return \_CONS;

} break;

case Num2: {

if (sym == NUMBER) return Num2;

return \_CONS;

} break;

case Oper1: {

if (sym == EQUAL) return ADD;

return PUSH;

} break;

case Oper2: {

if (sym == EQUAL) return ADD;

return ERR;

} break;

case ADD:

case PUSH:

case ERR:

case \_IDE:

case \_CONS: {

throw std::exception("That is final state!");

} break;

}

}

//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------

Automaton::Automaton(const std::string& str) : str(str), i(0), n(1), state(Start) {

}

//-----------------------------------------------------------------------------

bool Automaton::next(void) {

if (state == PUSH || state == \_IDE || state == \_CONS)

if (str[--i] == '\n') n--;

if (isEndState()) {

state = Start;

sum = "";

}

char c = str[i];

//Надо либо automatonNext засовывать внуторь класса Automation, либо так

if (c == '\n') n++;

state = automatonNext(state, getType(c));

sum += c;

i++;

return i != str.size();

}

//-----------------------------------------------------------------------------

bool Automaton::isEndState(void) const {

return

state == ADD ||

state == PUSH ||

state == NEXT ||

state == ERR ||

state == \_IDE ||

state == \_CONS;

}

//-----------------------------------------------------------------------------

AddType Automaton::getState(void) const {

if (state == ADD || state == PUSH)

return ADD\_KEYWORD\_OR\_DELIMITER;

if (state == \_IDE)

return ADD\_IDENTIFIER;

if (state == \_CONS)

return ADD\_CONSTANT;

if (state == NEXT)

return IGNORE;

if (state == ERR)

return ERROR;

return ERROR;

}

//-----------------------------------------------------------------------------

std::string Automaton::getStr(void) const {

auto result = sum;

if (state == PUSH || state == \_IDE || state == \_CONS)

result.pop\_back();

return result;

}

//-----------------------------------------------------------------------------

int Automaton::getPos(void) const {

return i;

}

int Automaton::getLine(void) const {

return n;

}

//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------

std::vector<Token> parse(const std::string& str, Tables& tables) {

std::vector<Token> result;

Automaton automaton(str);

while (automaton.next()) {

if (automaton.isEndState()) {

auto to\_add = automaton.getStr();

switch (automaton.getState()) {

case ADD\_IDENTIFIER: {

auto token = tables.find(to\_add);

if (token) {

result.push\_back(token);

} else {

result.push\_back(tables.add(to\_add, TABLE\_IDENTIFIERS));

}

} break;

case ADD\_CONSTANT: {

auto token = tables.find(to\_add);

if (token) {

result.push\_back(token);

} else {

result.push\_back(tables.add(to\_add, TABLE\_CONSTANTS));

}

} break;

case ADD\_KEYWORD\_OR\_DELIMITER: {

auto token = tables.find(to\_add);

if (token) {

result.push\_back(token);

} else {

// Такого быть не может

throw std::exception();

}

} break;

case IGNORE: {} break;

case ERROR: {

// Надо как-то обработать эту ошибку

std::string error = "Error around symbol \"" + to\_add +

"\"\nIn " + std::to\_string(automaton.getLine()) + " line";

throw error; // std::exception();

break;}

}

}

}

return result;

}

/\*tables.h\*/

#pragma once

#include <vector>

#include <string>

#include <functional>

#include <hash\_table.h>

using namespace std;

enum TableType

{

TABLE\_NULL,

// Статические таблицы

TABLE\_KEYWORDS,

TABLE\_OPERATIONS,

TABLE\_DELIMITERS,

// Динамические таблицы

TABLE\_IDENTIFIERS,

TABLE\_CONSTANTS,

TABLE\_STRUCTURES,

};

struct Token

{

TableType table;

hash\_table\_pos pos;

bool operator ==(Token b) {

return table == b.table && pos.line == b.pos.line && pos.pos == b.pos.pos;

}

bool operator !=(Token b) {

return table != b.table || pos.line != b.pos.line || pos.pos != b.pos.pos;

}

operator bool() const {

return bool(pos);

}

};

std::ostream& operator<<(std::ostream& out, const Token& t);

enum Type

{

TYPE\_NONE,

TYPE\_INT,

TYPE\_FLOAT,

TYPE\_STRUCT // Используется только в StructElem

};

struct Identifier

{

Type type;

bool isInitialized;

std::string nameStruct; //Если тип структура

};

struct Constant

{

Type type;

union {

float f;

int i;

} value;

};

struct Structure

{

struct StructElem

{

Type type;

std::string nameStruct; //Если тип структура

std::string name;

Token structToken;

};

std::vector<StructElem> elems;

};

class Tables

{

public:

Tables();

// Удаление из таблицы идентификаторов нужно потому что на этапе лексического анализа невозможно отличить структуры от имен идентификаторов. И все имена, даже имена структур будут заноситься в таблицу идентификаторов

void deleteFromIdentifiers(const Token& token);

Token find(const std::string& str) const;

Token find(const std::string& str, const TableType& type) const;

Token add(const std::string& str, const TableType& type);

std::string getStr(const Token& token) const;

template<class T>

T& get(const Token& token);

template<class T>

const T& get(const Token& token) const;

private:

struct void\_struct {};

hash\_table<void\_struct> m\_keywords;

hash\_table<void\_struct> m\_delimiters;

hash\_table<void\_struct> m\_operations;

hash\_table<Identifier> m\_identifiers;

hash\_table<Constant> m\_constants;

hash\_table<Structure> m\_structures;

};

//=============================================================================

//=============================================================================

//=============================================================================

//-----------------------------------------------------------------------------

template<class T>

T& Tables::get(const Token& token) {

if constexpr (std::is\_same<T, Identifier>::value)

return m\_identifiers[token.pos];

if constexpr (std::is\_same<T, Constant>::value)

return m\_constants[token.pos];

if constexpr (std::is\_same<T, Structure>::value)

return m\_structures[token.pos];

}

//-----------------------------------------------------------------------------

template<class T>

const T& Tables::get(const Token& token) const {

if constexpr (std::is\_same<T, Identifier>::value)

return m\_identifiers[token.pos];

if constexpr (std::is\_same<T, Constant>::value)

return m\_constants[token.pos];

if constexpr (std::is\_same<T, Structure>::value)

return m\_structures[token.pos];

}

/\*tables.cpp\*/

#include <sstream>

#include <tables.h>

Tables::Tables() :

m\_keywords(100),

m\_delimiters(100),

m\_operations(100),

m\_identifiers(500),

m\_constants(500),

m\_structures(500) {

{

std::vector<string> add = {"int", "float", "struct", "return", "void", "main"};

for (auto& i : add)

m\_keywords.add(i, {});

}

{

std::vector<string> add = {" ", ";", "\n", "\t", "{", "}", "(", ")", ","};

for (auto& i : add)

m\_delimiters.add(i, {});

}

{

std::vector<string> add = {"=", "+", "-", "\*", "/", "==", "!=", "<", ">"};

for (auto& i : add)

m\_operations.add(i, {});

}

}

//-----------------------------------------------------------------------------

void Tables::deleteFromIdentifiers(const Token& token) {

if (token.table != TABLE\_IDENTIFIERS)

throw std::exception();

m\_identifiers.erase(token.pos);

}

//-----------------------------------------------------------------------------

Token Tables::find(const std::string& str) const {

{ auto token = m\_keywords.find(str); if (token) return {TABLE\_KEYWORDS, token}; }

{ auto token = m\_delimiters.find(str); if (token) return {TABLE\_DELIMITERS, token}; }

{ auto token = m\_operations.find(str); if (token) return {TABLE\_OPERATIONS, token}; }

{ auto token = m\_identifiers.find(str); if (token) return {TABLE\_IDENTIFIERS, token}; }

{ auto token = m\_constants.find(str); if (token) return {TABLE\_CONSTANTS, token}; }

{ auto token = m\_structures.find(str); if (token) return {TABLE\_STRUCTURES, token}; }

return {TABLE\_NULL, hash\_table\_pos::getNullPos()};

}

Token Tables::find(const std::string& str, const TableType& type) const {

switch (type)

{

case TABLE\_KEYWORDS: { auto token = m\_keywords.find(str); if (token) return { TABLE\_KEYWORDS, token }; } break;

case TABLE\_DELIMITERS: { auto token = m\_delimiters.find(str); if (token) return { TABLE\_DELIMITERS, token }; } break;

case TABLE\_OPERATIONS: { auto token = m\_operations.find(str); if (token) return { TABLE\_OPERATIONS, token }; } break;

case TABLE\_IDENTIFIERS: { auto token = m\_identifiers.find(str); if (token) return { TABLE\_IDENTIFIERS, token }; } break;

case TABLE\_CONSTANTS: { auto token = m\_constants.find(str); if (token) return { TABLE\_CONSTANTS, token }; } break;

case TABLE\_STRUCTURES: { auto token = m\_structures.find(str); if (token) return { TABLE\_STRUCTURES, token }; } break;

}

return { TABLE\_NULL, hash\_table\_pos::getNullPos() };

}

//-----------------------------------------------------------------------------

Token Tables::add(const std::string& str, const TableType& type) {

switch (type) {

case TABLE\_IDENTIFIERS: return {TABLE\_IDENTIFIERS, m\_identifiers.add(str, {})}; break;

case TABLE\_CONSTANTS: return {TABLE\_CONSTANTS, m\_constants.add(str, {})}; break;

case TABLE\_STRUCTURES: return {TABLE\_STRUCTURES, m\_structures.add(str, {})}; break;

}

}

//-----------------------------------------------------------------------------

std::string Tables::getStr(const Token& token) const {

switch (token.table) {

case TABLE\_KEYWORDS: return m\_keywords.str(token.pos); break;

case TABLE\_OPERATIONS: return m\_operations.str(token.pos); break;

case TABLE\_DELIMITERS: return m\_delimiters.str(token.pos); break;

case TABLE\_IDENTIFIERS: return m\_identifiers.str(token.pos); break;

case TABLE\_CONSTANTS: return m\_constants.str(token.pos); break;

case TABLE\_STRUCTURES: return m\_structures.str(token.pos); break;

}

}

//-----------------------------------------------------------------------------

std::ostream& operator<<(std::ostream& out, const Token& t) {

std::stringstream sout;

if (t) {

switch (t.table) {

case TABLE\_KEYWORDS: sout << "KEYWORDS."; break;

case TABLE\_OPERATIONS: sout << "OPERATIONS."; break;

case TABLE\_DELIMITERS: sout << "DELIMITERS."; break;

case TABLE\_IDENTIFIERS: sout << "IDENTIFIERS."; break;

case TABLE\_CONSTANTS: sout << "CONSTANTS."; break;

case TABLE\_STRUCTURES: sout << "STRUCTURES."; break;

}

sout << t.pos.line << "." << t.pos.pos;

} else {

sout << "not exists";

}

out << sout.str();

return out;

}

/\*hash\_table.h\*/

#pragma once

#include <functional>

#include <vector>

#include <list>

// Хеш-функция. https://sohabr.net/post/219139/

struct HashH37

{

size\_t operator()(const std::string& str) const {

size\_t hash = 2139062143;

for (auto& i : str)

hash = 37 \* hash + size\_t(i);

return hash;

}

};

// Позиция в хеш-таблице

struct hash\_table\_pos

{

size\_t line;

size\_t pos;

// Возвращает true, если позиция действительная, и false если такой позиции не существует

operator bool() const {

return !(line == std::numeric\_limits<size\_t>::max() && pos == line);

}

static hash\_table\_pos getNullPos() {

return {std::numeric\_limits<size\_t>::max(), std::numeric\_limits<size\_t>::max()};

}

};

// Хеш-таблица. Можно обращаться как по строке, так и по позиции.

//template<typename T, typename F = std::hash<std::string>>

template<typename T, typename F = HashH37>

class hash\_table

{

public:

hash\_table(int size) : m\_table(size), m\_size(size) {

}

hash\_table\_pos add(const std::string& str, const T& t) {

if (find(str))

throw std::exception();

size\_t line = f(str) % m\_size;

m\_table[line].push\_back({str, t});

return {line, m\_table[line].size() - 1};

}

hash\_table\_pos find(const std::string& str) const {

size\_t line = f(str) % m\_size;

auto& list = m\_table[line];

for (auto i = list.begin(); i != list.end(); ++i) {

if (i->first == str) {

size\_t pos = std::distance(list.begin(), i);

return {line, pos};

}

}

return hash\_table\_pos::getNullPos();

}

void erase(const hash\_table\_pos& pos) {

m\_table[pos.line].erase(std::next(m\_table[pos.line].begin(), pos.pos));

}

std::string str(const hash\_table\_pos& pos) const {

return std::next(m\_table[pos.line].begin(), pos.pos)->first;

}

T& operator[](const std::string& str) {

auto pos = find(str);

return operator[](pos);

}

const T& operator[](const std::string& str) const {

auto pos = find(str);

return operator[](pos);

}

T& operator[](const hash\_table\_pos& pos) {

return std::next(m\_table[pos.line].begin(), pos.pos)->second;

}

const T& operator[](const hash\_table\_pos& pos) const {

return std::next(m\_table[pos.line].begin(), pos.pos)->second;

}

private:

std::vector<std::list<std::pair<std::string, T>>> m\_table;

int m\_size;

F f;

};