It.cpp

#include "stdafx.h"

namespace IT

{

IdTable::IdTable()

{

noname\_lexema\_count = 0;

maxsize = TI\_MAXSIZE;

size = 0;

table = new Entry[TI\_MAXSIZE];

}

Entry::Entry()

{

parrent\_function[0] = '\0';

id[0] = '\0';

firstApi = 0;

iddatatype = IT::IDDATATYPE::DEF;

idtype = IT::IDTYPE::D;

parmQuantity = 0;

}

Entry::Entry(const char\* parrent\_function, const char\* id, IDDATATYPE iddatatype, IDTYPE idtype, int first)

{

int i = 0;

if (parrent\_function)

for (i = 0; parrent\_function[i] != '\0'; i++)

this->parrent\_function[i] = parrent\_function[i];

this->parrent\_function[i] = '\0';

i = 0;

if (id)

for (i = 0; id[i] != '\0'; i++)

this->id[i] = id[i];

this->firstApi = first;

this->id[i] = '\0';

this->iddatatype = iddatatype;

this->idtype = idtype;

this->parmQuantity = 0;

}

Entry::Entry(const char\* parrent\_function, const char\* id, IDDATATYPE iddatatype, IDTYPE idtype, int first, int it)

{

int i = 0;

if (parrent\_function)

for (i = 0; parrent\_function[i] != '\0'; i++)

this->parrent\_function[i] = parrent\_function[i];

this->parrent\_function[i] = '\0';

i = 0;

if (id)

for (i = 0; id[i] != '\0'; i++)

this->id[i] = id[i];

this->firstApi = first;

this->id[i] = '\0';

this->iddatatype = iddatatype;

this->idtype = idtype;

this->parmQuantity = 0;

this->value.vint = it;

}

Entry::Entry(const char\* parrent\_function, const char\* id, IDDATATYPE iddatatype, IDTYPE idtype, int first, char\* ch)

{

int i = 0;

if (parrent\_function)

for (i = 0; parrent\_function[i] != '\0'; i++)

this->parrent\_function[i] = parrent\_function[i];

this->parrent\_function[i] = '\0';

i = 0;

if (id)

for (i = 0; id[i] != '\0'; i++)

this->id[i] = id[i];

this->firstApi = first;

this->id[i] = '\0';

this->iddatatype = iddatatype;

this->idtype = idtype;

this->parmQuantity = 0;

strcpy\_s(this->value.vstr.str, 255, ch);

this->value.vstr.len = strlen(ch);

}

Entry::Entry(const char\* parrent\_function, const char\* id, IDDATATYPE iddatatype, IDTYPE idtype, int first, const char\* ch)

{

int i = 0;

if (parrent\_function)

for (i = 0; parrent\_function[i] != '\0'; i++)

this->parrent\_function[i] = parrent\_function[i];

this->parrent\_function[i] = '\0';

i = 0;

if (id)

for (i = 0; id[i] != '\0'; i++)

this->id[i] = id[i];

this->firstApi = first;

this->id[i] = '\0';

this->iddatatype = iddatatype;

this->idtype = idtype;

this->parmQuantity = 0;

strcpy\_s(this->value.vstr.str, 255, ch);

this->value.vstr.len = strlen(ch);

}

Entry::Entry(char\* parrent\_function, char\* id, IDDATATYPE iddatatype, IDTYPE idtype)

{

int i = 0;

if (parrent\_function)

for (i = 0; parrent\_function[i] != '\0'; i++)

this->parrent\_function[i] = parrent\_function[i];

this->parrent\_function[i] = '\0';

i = 0;

if (id)

for (i = 0; id[i] != '\0'; i++)

this->id[i] = id[i];

this->id[i] = '\0';

this->iddatatype = iddatatype;

this->idtype = idtype;

this->parmQuantity = 0;

}

IdTable Create(int size)

{

IdTable id\_table;

id\_table.size = size;

id\_table.maxsize = TI\_MAXSIZE;

id\_table.table = new Entry[TI\_MAXSIZE];

return id\_table;

}

void IdTable::Add(Entry entry)

{

if (strlen(entry.id) > ID\_MAXSIZE && entry.idtype != IDTYPE::F)

throw ERROR\_THROW(65);

if(size >= maxsize)

throw ERROR\_THROW(66);

if (entry.idtype != IDTYPE::F)

entry.id[5] = '\0';

table[size] = entry;

switch (entry.iddatatype)

{

case IDDATATYPE::INT:

{

table[size].value.vint = TI\_INT\_DEFAULT;

break;

}

case IDDATATYPE::STR:

{

table[size].value.vstr.str[0] = TI\_STR\_DEFAULT;

table[size].value.vstr.len = 0;

break;

}

}

size++;

}

Entry IdTable::GetEntry(int n)

{

if (n >= 0 && n < maxsize)

return table[n];

}

int IdTable::IsId(const char id[ID\_MAXSIZE])

{

for (int i = 0; i < size; i++)

if (strcmp(table[i].id, id) == 0)

return i;

return TI\_NULLIDX;

}

int IdTable::IsId(const char id[ID\_MAXSIZE], const char parrent\_function[ID\_MAXSIZE + 5])

{

for (int i = 0; i < size; i++)

if ((strcmp(this->table[i].id, id) == 0) &&

(strcmp(this->table[i].parrent\_function, parrent\_function) == 0))

return i;

return TI\_NULLIDX;

}

void Delete(IdTable& idtable)

{

delete[] idtable.table;

idtable.table = nullptr;

}

char\* IdTable::GetLexemaName()

{

char buffer[5];

\_itoa\_s(noname\_lexema\_count, buffer, 10);

strcat\_s(buffer, 5, "\_l");

noname\_lexema\_count++;

return buffer;

}

void IdTable::PrintIdTable(const wchar\_t\* inFile)

{

ofstream idStream(inFile);

if (!idStream.is\_open())

throw ERROR\_THROW(64);

idStream << "---------------------Таблица идентификаторов:---------------------" << endl;

idStream << " Литералы:" << endl;

idStream << setw(15) << "Идентификатор:" << setw(17) << "Тип данных:" << setw(15) << "Значение:" << setw(27) << "Длина строки:" << setw(20) << "Первое вхождение:" << endl;

for (int i = 0; i < size; i++)

{

if (table[i].idtype == IT::IDTYPE::L)

{

cout.width(25);

switch (table[i].iddatatype)

{

case IDDATATYPE::INT:

idStream << " " << this->table[i].id << "\t\t\t\t\t" << "INT " << "\t\t\t " << table[i].value.vint << "\t\t\t\t\t\t " << "-\t\t\t\t\t" << table[i].firstApi << endl;

break;

case IDDATATYPE::STR:

idStream << " " << this->table[i].id << "\t\t\t\t\t" << "STR " << "\t " << table[i].value.vstr.str << setw(30 - strlen(table[i].value.vstr.str)) << (int)table[i].value.vstr.len << "\t\t\t\t\t" << table[i].firstApi << endl;

break;

}

}

}

idStream << "\n\n\n\n";

idStream << "Функции:" << endl;

idStream << setw(15) << "Идентификатор:" << setw(26) << "Тип данных возврата:" << setw(36) << "Количество переданных параметров:" << setw(22) << "Первое вхождение:" << endl;

for (int i = 0; i < size; i++)

{

if (this->table[i].idtype == IT::IDTYPE::F)

{

switch (table[i].iddatatype)

{

case IDDATATYPE::INT:

idStream << " " << table[i].id << setw(28 - strlen(table[i].id)) << "INT " << "\t\t\t\t\t\t\t\t" << table[i].parmQuantity << "\t\t\t\t\t\t\t" << table[i].firstApi << endl;

break;

case IDDATATYPE::STR:

idStream << " " << table[i].id << setw(28 - strlen(table[i].id)) << "STR " << "\t\t\t\t\t\t\t\t" << table[i].parmQuantity << "\t\t\t\t\t\t\t" << table[i].firstApi << endl;

break;

}

}

}

idStream << "\n\n\n\n";

idStream << "Переменные:" << endl;

idStream << setw(25) << "Имя родительского блока:" << setw(20) << "Идентификатор:" << setw(16) << "Тип данных:" << setw(24) << "Тип идентификатора:" << setw(21) << "Первое вхождение:" << endl;

for (int i = 0; i < size; i++)

{

if (table[i].idtype == IT::IDTYPE::V)

{

switch (table[i].iddatatype)

{

case IDDATATYPE::INT:

idStream << " " << table[i].parrent\_function << setw(35 - strlen(table[i].parrent\_function)) << table[i].id << setw(20) << "INT " << setw(15) << "\t" << "V" << "\t\t\t\t\t\t" << table[i].firstApi << endl;

break;

case IDDATATYPE::STR:

idStream << " " << table[i].parrent\_function << setw(35 - strlen(table[i].parrent\_function)) << table[i].id << setw(20) << "STR " << setw(15) << "\t" << "V" << "\t\t\t\t\t\t" << table[i].firstApi << endl;

break;

}

}

if (table[i].idtype == IT::IDTYPE::P)

{

switch (table[i].iddatatype)

{

case IDDATATYPE::INT:

idStream << " " << table[i].parrent\_function << setw(35 - strlen(table[i].parrent\_function)) << table[i].id << setw(20) << "INT " << setw(15) << "\t" << "P" << "\t\t\t\t\t\t" << table[i].firstApi << endl;

break;

case IDDATATYPE::STR:

idStream << " " << table[i].parrent\_function << setw(35 - strlen(table[i].parrent\_function)) << table[i].id << setw(20) << "STR " << setw(15) << "\t" << "P" << "\t\t\t\t\t\t" << table[i].firstApi << endl;

break;

}

}

}

idStream << "\n\n\n";

idStream.close();

}

}

It.h

#pragma once

// Из 2 Части 17 лабы(взято)

//для построения таблицы лексем

#define ID\_MAXSIZE 5 // максимальное количество символов в идентификаторе

#define TI\_MAXSIZE 4096 // максимальное количество строк в таблице идентификаторов

#define TI\_INT\_DEFAULT 0x00000000 // значение по умолчанию для типа integer

#define TI\_STR\_DEFAULT 0x00 // значение по умолчанию для типа string

#define TI\_NULLIDX 0xffffffff // нет элемента таблицы идентификаторов

#define TI\_STR\_MAXSIZE 255 // максимальная длина строки

#define PARM\_ID L".id.txt"

namespace IT // таблица идентификаторов

{

enum IDDATATYPE { DEF = 0, INT = 1, STR = 2 }; // типы данных идентификаторов: integer, string

enum IDTYPE { D = 0, V = 1, F = 2, P = 3, L = 4 }; // типы идентификаторов: переменная, функция, параметр, литерал

struct Entry // строка таблицы идентификаторов

{

char parrent\_function[ID\_MAXSIZE + 5]; // идентификатор родительской функции

int firstApi; // индекс первой строки в таблице лексем

char id[ID\_MAXSIZE + 5]; // идентификатор (автоматически усекается до ID\_MAXSIZE)

IDDATATYPE iddatatype; // тип данных

IDTYPE idtype; // тип идентификатора

union

{

int vint; // значение integer

char operation = '\0';

struct

{

unsigned char len; // количество символов в string

char str[TI\_STR\_MAXSIZE]; // символы string

} vstr; // значение string

} value; // значение идентификатора

int parmQuantity;

Entry();

Entry(const char\* parrentFunc, const char\* id, IDDATATYPE iddatatype, IDTYPE idtype, int first);

Entry(const char\* parrentFunc, const char\* id, IDDATATYPE iddatatype, IDTYPE idtype, int first, int it);

Entry(const char\* parrentFunc, const char\* id, IDDATATYPE iddatatype, IDTYPE idtype, int first, char\* str);

Entry(const char\* parrentFunc, const char\* id, IDDATATYPE iddatatype, IDTYPE idtype, int first, const char\* str);

Entry(char\* parrentFunc, char\* id, IDDATATYPE iddatatype, IDTYPE idtype);

};

struct IdTable // экземпляр таблицы идентификаторов

{

int noname\_lexema\_count; // счётчик неидентифицированных лексем

int maxsize; // емкость таблицы идентификаторов < TI\_MAXSIZE

int size; // текущий размер таблицы идентификаторов < maxsize

Entry\* table; // массив строк таблицы идентификаторов

Entry GetEntry( // получить строку таблицы идентификаторов

int n // номер получаемой строки

);

int IsId( // возврат: номер строки (если есть), TI\_NULLIDX(если нет)

const char id[ID\_MAXSIZE] // идентификатор

);

int IsId(const char id[ID\_MAXSIZE], const char parrent\_function[ID\_MAXSIZE + 5]);

void Add( // добавить строку в таблицу идентификаторов

Entry entry // строка таблицы идентификторов

);

void PrintIdTable(const wchar\_t\* in); // вывод таблицы идентификаторов

IdTable();

char\* GetLexemaName();

};

void Delete(IdTable& idtable); // удалить таблицу лексем (освободить память)

};

LT.CPP

#include "stdafx.h"

namespace LT

{

LT::LexTable::LexTable()

{

maxsize = LT\_MAXSIZE;

size = 0;

table = new Entry[LT\_MAXSIZE];

}

void Add(LexTable& lextable, Entry entry)

{

if (lextable.size >= lextable.maxsize)

throw ERROR\_THROW(60);

lextable.table[lextable.size++] = entry;

}

Entry LexTable::GetEntry(int n)

{

if (n >= 0 && n < maxsize)

return table[n];

}

void Delete(LexTable& lextable)

{

delete[] lextable.table;

lextable.table = nullptr;

}

void LexTable::PrintLexTable(const wchar\_t\* in)

{

ofstream lexStream(in);

if(!lexStream.is\_open())

throw ERROR\_THROW(63);

int num\_string = 0;

for (int i = 0; i < size;)

{

if (num\_string == table[i].sn) {

lexStream << table[i++].lexema;

continue;

}

lexStream << '\n' << ++num\_string << ".\t";

}

lexStream.close();

}

LT::Entry::Entry()

{

lexema = '\0';

sn = LT\_TI\_NULLXDX;

idxTI = LT\_TI\_NULLXDX;

}

LT::Entry::Entry(const char lex, int str\_n, int idxTI)

{

lexema = lex;

sn = str\_n;

this->idxTI = idxTI;

}

}

LT.H

// Из 2 Части 17 лабы(взято)

//для построения таблицы лексем

#pragma once

#include "IT.h"

#define LEXEMA\_FIXSIZE 1 // фиксированный размер лексемы

#define LT\_MAXSIZE 4096 // максимальное количество строк в таблице лексем

#define LT\_TI\_NULLXDX -1 // нет элемента таблицы идентификаторов

#define LEX\_DATATYPE 't' // лексема для типа данных

#define LEX\_INTEGER 't' // лексема для integer

#define LEX\_STRING 't' // лексема для string

#define LEX\_ID 'i' // лексема для идентификатора

#define LEX\_LITERAL 'l' // лексема для литерала

#define LEX\_FUNCTION 'f' // лексема для function

#define LEX\_DECLARE 'd' // лексема для declare

#define LEX\_RETURN 'r' // лексема для return

#define LEX\_PRINT 'p' // лексема для print

#define LEX\_MAIN 'm' // лексема для main

#define LEX\_STRLEN 'e' // лексема для strlen

#define LEX\_SEMICOLON ';' // лексема для ;

#define LEX\_COMMA ',' // лексема для ,

#define LEX\_LEFTBRACE '{' // лексема для {

#define LEX\_RIGHTBRACE '}' // лексема для }

#define LEX\_LEFTHESIS '(' // лексема для (

#define LEX\_RIGHTHESIS ')' // лексема для )

#define LEX\_PLUS 'v' // лексема для +

#define LEX\_MINUS 'v' // лексема для -

#define LEX\_STAR 'v' // лексема для \*

#define LEX\_DIRSLASH 'v' // лексема для /

#define LEX\_EQUAL\_SIGN '=' // лексема для =

#define PLUS '+'

#define MINUS '-'

#define STAR '\*'

#define DIRSLASH '/'

namespace LT // таблица лексем

{

struct Entry // строка таблицы лексем

{

char lexema; // лексема

int sn; // номер строки в исходном тексте

int idxTI; // индекс в таблице идентификаторов или LT\_TI\_NULLIDX

Entry();

Entry(const char lex, int str\_n, int idxTI);

};

struct LexTable // экзепляр таблицы лексем

{

int maxsize; // ёмкость таблицы лексем < LT\_MAXSIZE

int size; // текущий размер таблицы лексем < maxsize

Entry\* table; // массив строк таблицы лексем

Entry GetEntry( // получить строку таблицы лексем

int n // номер получаемой строки

);

void PrintLexTable(const wchar\_t\* in); // вывод таблицы лексем

LexTable();

};

void Add( // добавить строку в таблицу лексем

LexTable& lextable, // экземпляр таблицы лексем

Entry entry // строка таблицы лексем

);

void Delete(LexTable& lextable); // удалить таблицу лексем (освободить память)

};

LEXANALYSIS.CPP

#include "stdafx.h"

namespace LA

{

void LexicalAnalyzer(const In::IN& source, LT::LexTable& lexTable, IT::IdTable& idTable)

{

bool isWord = false; //было ли считано слово

int tempWordIndex = 0; // индекс для буфера с текущим словом

int lexemaLine = 1; // текущая строка

int lexemaPosition = 0; // позиция текущей лексемы в строке

In::IN inTable;

for (int i = 0; i < source.size; i++)

{

char\* tempWord = new char[token\_size] {};

// проверка слова

while (inTable.code[source.text[i]] == inTable.T && source.text[i] != '|')

{

tempWord[tempWordIndex++] = source.text[i++];

lexemaPosition++;

isWord = true;

}

// Если слово истинно

if (isWord)

{

if (AnalyzeToken(tempWord, lexemaLine, lexTable, idTable))

{

tempWordIndex = 0;

i--;

isWord = false;

continue;

}

else

throw ERROR\_THROW\_IN(90, lexemaLine, lexemaPosition); // слово не распознано

}

// Считывание строкового литерала

if (source.text[i] == '\'')

{

tempWord[tempWordIndex++] = source.text[i++];

while (source.text[i] != '\'')

{

if (tempWordIndex >= TI\_STR\_MAXSIZE)

throw ERROR\_THROW\_IN(91, lexemaLine, lexemaPosition); // строковый литерал слишком длинный

tempWord[tempWordIndex++] = source.text[i++];

}

tempWord[tempWordIndex] = source.text[i];

if (AnalyzeToken(tempWord, lexemaLine, lexTable, idTable))

{

tempWordIndex = 0;

continue;

}

else

throw ERROR\_THROW\_IN(92, lexemaLine, lexemaPosition); // строковый литерал не распознан автоматом

}

// Считывание символа сепаратора или оператора

if (inTable.code[source.text[i]] == inTable.S || inTable.code[source.text[i]] == inTable.O)

{

if (source.text[i] == ' ' || source.text[i] == '\t')

continue;

tempWord[tempWordIndex] = source.text[i];

if (!AnalyzeToken(tempWord, lexemaLine, lexTable, idTable))

throw ERROR\_THROW\_IN(123, lexemaLine, lexemaPosition); // символ операции или сепаратор не распознан

lexemaPosition++;

tempWord = 0;

continue;

}

if (source.text[i] == '|') //конец строки

{

lexemaLine++;

lexemaPosition = 0;

}

}

}

//анализ символа лексемы для записи в таблицу лексем

bool AnalyzeToken(char\* token, const int str\_number, LT::LexTable& lexTable, IT::IdTable& idTable)

{

static TypeOfVar flag\_type\_variable;

switch (token[0])

{

case LEX\_COMMA:

Add(lexTable, { LEX\_COMMA ,str\_number, LT\_TI\_NULLXDX });

return true;

case LEX\_LEFTBRACE:

Add(lexTable, { LEX\_LEFTBRACE,str\_number,LT\_TI\_NULLXDX });

return true;

case LEX\_RIGHTBRACE:

Add(lexTable, { LEX\_RIGHTBRACE,str\_number,LT\_TI\_NULLXDX });

return true;

case LEX\_LEFTHESIS:

Add(lexTable, { LEX\_LEFTHESIS,str\_number,LT\_TI\_NULLXDX });

return true;

case LEX\_RIGHTHESIS:

Add(lexTable, { LEX\_RIGHTHESIS,str\_number,LT\_TI\_NULLXDX });

return true;

case LEX\_SEMICOLON:

Add(lexTable, { LEX\_SEMICOLON,str\_number,LT\_TI\_NULLXDX });

return true;

case LEX\_EQUAL\_SIGN:

Add(lexTable, { LEX\_EQUAL\_SIGN, str\_number, LT\_TI\_NULLXDX });

return true;

case '+':

Add(lexTable, { LEX\_PLUS,str\_number,idTable.size - 1 });

return true;

case '-':

Add(lexTable, { LEX\_MINUS, str\_number, idTable.size - 1 });

return true;

case '\*':

Add(lexTable, { LEX\_STAR, str\_number, idTable.size - 1 });

return true;

case '/':

Add(lexTable, { LEX\_DIRSLASH, str\_number, idTable.size - 1 });

return true;

case 'f':

{

FST::FST function(FUNCTION(token));

if (FST::execute(function))

{

Add(lexTable, { LEX\_FUNCTION, str\_number, LT\_TI\_NULLXDX });

return true;

}

else

return AnalyzeIdentifier(token, str\_number, lexTable, idTable, flag\_type\_variable);

}

case 'i':

{

FST::FST integer(INTEGER(token));

if (FST::execute(integer))

{

Add(lexTable, { LEX\_INTEGER, str\_number, LT\_TI\_NULLXDX });

flag\_type\_variable.LT\_posititon = lexTable.size - 1;

flag\_type\_variable.type = TypeOfVar::INT;

return true;

}

else

return AnalyzeIdentifier(token, str\_number, lexTable, idTable, flag\_type\_variable);

}

case 's':

{

FST::FST string(STRING(token));

if (FST::execute(string))

{

Add(lexTable, { LEX\_STRING, str\_number, LT\_TI\_NULLXDX });

flag\_type\_variable.LT\_posititon = lexTable.size - 1;

flag\_type\_variable.type = TypeOfVar::STR;

return true;

}

else

return AnalyzeIdentifier(token, str\_number, lexTable, idTable, flag\_type\_variable);

}

case 'd':

{

FST::FST declare(DECLARE(token));

if (FST::execute(declare))

{

Add(lexTable, { LEX\_DECLARE, str\_number, LT\_TI\_NULLXDX });

return true;

}

else

return AnalyzeIdentifier(token, str\_number, lexTable, idTable, flag\_type\_variable);

}

case 'r':

{

FST::FST \_return(RETURN(token));

if (FST::execute(\_return))

{

Add(lexTable, { LEX\_RETURN, str\_number, LT\_TI\_NULLXDX });

return true;

}

else

return AnalyzeIdentifier(token, str\_number, lexTable, idTable, flag\_type\_variable);

}

case 'p':

{

FST::FST print(PRINT(token));

if (FST::execute(print))

{

Add(lexTable, { LEX\_PRINT, str\_number, LT\_TI\_NULLXDX });

return true;

}

else

return AnalyzeIdentifier(token, str\_number, lexTable, idTable, flag\_type\_variable);

}

case '\'':

{

FST::FST stringLiteral(STRING\_LITERAL(token));

if (FST::execute(stringLiteral))

{

int i = idTable.IsId(token);

if (i != LT\_TI\_NULLXDX)

Add(lexTable, { LEX\_LITERAL, str\_number, i });

else

{

idTable.Add({ "\0", (idTable.GetEntry(lexTable.GetEntry(lexTable.size - 2).idxTI).id[0] != '\0' && lexTable.GetEntry(lexTable.size - 1).lexema == '=') ? idTable.GetEntry(lexTable.GetEntry(lexTable.size - 2).idxTI).id : idTable.GetLexemaName() , IT::IDDATATYPE::STR, IT::IDTYPE::L,lexTable.size });

idTable.table[idTable.size - 1].value.vstr.len = 0;

int i = 0, j = 0;

for (; token[i] != '\0'; i++)

{

idTable.table[idTable.size - 1].value.vstr.str[j] = token[i];

idTable.table[idTable.size - 1].value.vstr.len++;

j++;

}

idTable.table[idTable.size - 1].value.vstr.str[j] = '\0';

Add(lexTable, { LEX\_LITERAL, str\_number,idTable.size - 1 });

}

return true;

}

}

default:

{

FST::FST numberLiteral(INTEGER\_LITERAL(token));

if (FST::execute(numberLiteral))// попытка распознать число

{

int i = idTable.IsId(token);

if (i != LT\_TI\_NULLXDX)

Add(lexTable, { LEX\_LITERAL, str\_number, i });

else

{

idTable.Add({ "\0", (idTable.GetEntry(lexTable.GetEntry(lexTable.size - 2).idxTI).id[0] != '\0' && lexTable.GetEntry(lexTable.size - 1).lexema == '=') ? idTable.GetEntry(lexTable.GetEntry(lexTable.size - 2).idxTI).id : idTable.GetLexemaName() , IT::IDDATATYPE::INT, IT::IDTYPE::L, lexTable.size });

idTable.table[idTable.size - 1].value.vint = atoi(token);

Add(lexTable, { LEX\_LITERAL,str\_number, idTable.size - 1 });

}

return true;

}

else //лексема является идентификатором

return AnalyzeIdentifier(token, str\_number, lexTable, idTable, flag\_type\_variable);

}

}

}

//анализ идентификаторов(переменных и функций)

bool AnalyzeIdentifier(char\* token, int str\_number, LT::LexTable& lexTable, IT::IdTable& idTable, TypeOfVar& flagTypeVariable)

{

FST::FST id(IDENTIFICATOR(token));

if (!FST::execute(id))

throw ERROR\_THROW\_IN(93, str\_number, -1);

// main

if (strcmp(token, "main") == 0)

{

for (int i = 0; i < lexTable.size; i++)

if (lexTable.GetEntry(i).lexema == LEX\_MAIN) // Проверка:был ли "main" добавлен в лексическую таблицу

throw ERROR\_THROW\_IN(94, str\_number, -1);

//"main" добавляется в таблицу лексем.

idTable.Add({ "\0", token, IT::IDDATATYPE::INT, IT::IDTYPE::F, lexTable.size });

Add(lexTable, { LEX\_MAIN, str\_number, idTable.size - 1 });

return true;

}

// Функция

if (lexTable.GetEntry(lexTable.size - 1).lexema == LEX\_FUNCTION

&& lexTable.GetEntry(lexTable.size - 2).lexema == LEX\_DATATYPE)

{

if(idTable.IsId(token) != -1)

throw ERROR\_THROW\_IN(95, str\_number, -1);

switch (flagTypeVariable.type) //указывается тип переменной

{

case TypeOfVar::INT:

idTable.Add({ "\0", token, IT::IDDATATYPE::INT, IT::IDTYPE::F, lexTable.size });

break;

case TypeOfVar::STR:

idTable.Add({ "\0", token, IT::IDDATATYPE::STR, IT::IDTYPE::F, lexTable.size });

break;

}

flagTypeVariable.type = TypeOfVar::DEF;

Add(lexTable, { LEX\_ID, str\_number, idTable.size - 1 });

return true;

}

// Переменная, если записан после declare | datatype

if (lexTable.GetEntry(lexTable.size - 1).lexema == LEX\_DATATYPE

&& lexTable.GetEntry(lexTable.size - 2).lexema == LEX\_DECLARE)

{

for (int i = lexTable.size - 1; i >= 0; i--)

{

// Поиск родительской функции

if ((lexTable.GetEntry(i).lexema == LEX\_ID || lexTable.GetEntry(i).lexema == LEX\_MAIN)

&& idTable.GetEntry(lexTable.GetEntry(i).idxTI).idtype == IT::IDTYPE::F)

{

if (idTable.IsId(token, idTable.GetEntry(lexTable.GetEntry(i).idxTI).id) != -1)

throw ERROR\_THROW\_IN(96, str\_number, -1);

switch (flagTypeVariable.type)

{

case TypeOfVar::INT:

idTable.Add({ idTable.GetEntry(lexTable.GetEntry(i).idxTI).id, token, IT::IDDATATYPE::INT, IT::IDTYPE::V, lexTable.size });

break;

case TypeOfVar::STR:

idTable.Add({ idTable.GetEntry(lexTable.GetEntry(i).idxTI).id, token, IT::IDDATATYPE::STR, IT::IDTYPE::V, lexTable.size });

break;

}

flagTypeVariable.type = TypeOfVar::DEF;

Add(lexTable, { LEX\_ID, str\_number, idTable.size - 1 });

return true;

}

}

}

// Параметр функции, если записан после function и после стоит "{"

if (lexTable.GetEntry(lexTable.size - 1).lexema == 't')

{

for (int i = lexTable.size - 1; i > 0; i--)

{

if (lexTable.GetEntry(i - 2).lexema == LEX\_DATATYPE

&& lexTable.GetEntry(i - 1).lexema == LEX\_FUNCTION

&& lexTable.GetEntry(i).lexema == LEX\_ID

&& idTable.GetEntry(lexTable.GetEntry(i).idxTI).idtype == IT::IDTYPE::F)

{

if (idTable.IsId(token, idTable.GetEntry(lexTable.GetEntry(i).idxTI).id) != -1)

throw ERROR\_THROW\_IN(96, str\_number, -1);

switch (flagTypeVariable.type)

{

case TypeOfVar::INT:

idTable.Add({ idTable.GetEntry(lexTable.GetEntry(i).idxTI).id, token, IT::IDDATATYPE::INT, IT::IDTYPE::P, lexTable.size });

idTable.table[(lexTable.GetEntry(i).idxTI)].parmQuantity++;

break;

case TypeOfVar::STR:

idTable.Add({ idTable.GetEntry(lexTable.GetEntry(i).idxTI).id, token, IT::IDDATATYPE::STR, IT::IDTYPE::P, lexTable.size });

idTable.table[(lexTable.GetEntry(i).idxTI)].parmQuantity++;

break;

}

flagTypeVariable.type = TypeOfVar::DEF;

Add(lexTable, { LEX\_ID, str\_number, idTable.size - 1 });

return true;

}

}

}

bool LeftBrace = false;

for (int i = lexTable.size - 1; i >= 0; i--)

{

if (lexTable.GetEntry(i).lexema == LEX\_LEFTBRACE)

LeftBrace = true;

if (LeftBrace && (lexTable.GetEntry(i).lexema == LEX\_ID || lexTable.GetEntry(i).lexema == LEX\_MAIN)

&& idTable.GetEntry(lexTable.GetEntry(i).idxTI).idtype == IT::IDTYPE::F)

{

int temp = idTable.IsId(token, idTable.GetEntry(lexTable.GetEntry(i).idxTI).id);

if (temp != -1)

{

Add(lexTable, { LEX\_ID,str\_number,temp });

return true;

}

else //поиск идентификатора в таблице идентификаторов

{

temp = idTable.IsId(token);

if (idTable.GetEntry(temp).idtype != IT::IDTYPE::F)

{

token[5] = '\0';

}

if (temp != -1 && idTable.GetEntry(temp).idtype == IT::IDTYPE::F)

{

Add(lexTable, { LEX\_ID,str\_number,temp });

return true;

}

else throw ERROR\_THROW\_IN(97, str\_number, -1);

}

}

}

return false;

}

}

LEXANALYSIS.H

#pragma once

#include "stdafx.h"

#define token\_size 256

namespace LA

{

struct TypeOfVar

{

int LT\_posititon = -1;

enum { DEF = 0, INT = 1, STR = 2 } type = DEF;

};

bool AnalyzeIdentifier(char\* token, const int strNumber, LT::LexTable& lexTable, IT::IdTable& idTable, TypeOfVar& flag\_type\_variable);

void LexicalAnalyzer(const In::IN& source, LT::LexTable& lexTable, IT::IdTable& idTable);

bool AnalyzeToken(char\* token, const int strNumber, LT::LexTable& lexTable, IT::IdTable& idTable);

}

MAIN

include "stdafx.h"

int \_tmain(int argc, \_TCHAR\* argv[])

{

argc = 1;

argv[0] = (\_TCHAR\*)L"-in:C:\\Users\\admin\\Documents\\SE\_Lab14\\SE\_Lab14\\in.txt";

setlocale(LC\_ALL, "Rus");

Log::LOG log = Log::INITLOG;

Out::OUT out = Out::INITOUT;

try

{

Parm::PARM parm = Parm::getparm(argc, argv);

log = Log::getlog(parm.log);

out = Out::getOut(parm.out);

Log::WriteLog(log);

Log::WriteParm(log, parm);

In::IN in = In::getin(parm.in); // возвращает структуру IN с обработанными данными.

Log:WriteIn(log, in); // протокол

Out::WriteOut(out, in); // выходной файл

LT::LexTable lexTable; // экзепляр таблицы лексем

IT::IdTable idTable; //экземпляр таблицы идентификаторов

LA::LexicalAnalyzer(in, lexTable, idTable);

lexTable.PrintLexTable(L"TableOfLexems.txt");

idTable.PrintIdTable(L"TableOfIdentificators.txt");

LT::Delete(lexTable);

IT::Delete(idTable);

Log::Close(log);

Out::Close(out);

}

catch (Error::ERROR e)

{

Log::WriteError(log, e);

Out::WriteError(out, e);

}

return 0;

}