МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФГБОУ ВПО «Пермский государственный национальный исследовательский университет»

ОТЧЕТ

О ВЫПОЛНЕНИИ ЗАДАНИЯ ПО ДИСЦИПЛИНЕ

«Формальные грамматики и методы трансляции»

по теме:

«Компилятор языка PASCAL»

Работу выполнил

Студент гр. ПМИ-2, 2021,

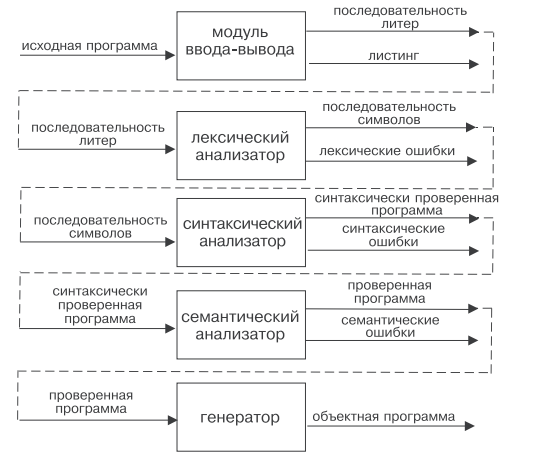
3 курс

Решетников М.А.

# **Модуль 1. Лексический анализатор.**

## **Теория и общие сведения.**

Структура компилятора:



Модуль ввода-вывода в моей программе соединен с лексическим анализатором.

Теория:

Необходимо разбить текстовый файл на токены языка паскаль — идентификаторы, ключевые слова, разделители, числа.

Токены должны браться из файла по требованию командой GetNextToken;

Пример: x: = 5;

Должно разбиваться на последовательность токенов:

X – Ident Token

: = - Special Symbol

5 – Constant Token

## **Моделирование.**

Для удобного ориентирования и написания кода должны быть расписаны все варианты токенов:

Token – базовый класс токена все остальные наследуют от него

Ident Token – токен-идентификатор

Keyword Token – токен для ключевого слова

Constant Token – токен для констант

Special Symbols Token – токен для специального символа

Удобно создать перечисления для каждой группы токенов

Пример:

eSpecialSymbols

{

comma,

dot,

RightCurveBrascet,

LeftCurveBrascet,

И т.д,

}

Примерная модель иерархии токенов:

class Token {

eTokenType type;

};

class IdentToken: public Token {

string ident;

}

class KeyWordToken: public Token {

eKeyWords kw;

}

class ConstToken: public Token {

variant<int, float, string, bool> data;

}

class SpecialSymbols: public Token {

eSpecialSymbols ss;

}

Лексический анализатор должен считывать информацию из файла по требованию поэтому в нем должен быть открыт файл с кодом языка Pascal

Lexer() {

position = 0;

row = 1;

file.open("D:\\myCompiler\\PascalCode.txt");

getline(file, input);

}

В конструкторе будет открываться файл для чтения, а также будут инициализированы поля под местоположения текущего просматриваемого элемента.

Примерный вид класса Lexer:

class Lexer {

private:

size\_t position;

size\_t row;

ifstream file;

string exceptions = "";

string input;

void Read\_input() {

if (position >= input.size()) {

// read from file

}

}

public:

Lexer() {

position = 0;

row = 1;

file.open("D:\\myCompiler\\PascalCode.txt");

getline(file, input);

}

~Lexer() {

file.close();

}

Как идет преобразование в токен:

GetNextToken – функция возвращающая текущий токен

Примерная реализация GetNextToken:

Мы должны прочитать строчку из файла

Читаем строку из файла

Пропустить пробелы

Если файл кончился вернуть null

Начинаем считывать символы из строки

Если является спец символом возвращаем его

Проверяет если буква, то это либо keyword, либо ident

Формируем слово

Если это keyword возвращаем его

Если дошли до конца это ident

Если была цифра, то это константа

Считываем константу

Перечисления для всех групп токенов.

enum eTokenType {

ttConstants, // Constants int float ...

ttIdentifier, // such as abc

ttKeywords, // such as if

ttSpecialSymbols, // such as + - \* / ( ) [ ] { } ...

UNKNOWN,

};

enum eKeyWords {

kwIf,

kwThen,

kwBegin,

kwProgram,

kwEnd,

kwWriteln,

...

};

enum eSpecialSymbols {

ssComma,

ssDot,

ssRightCurveBrascet,

ssLeftCurveBrascet,

ssEqual,

ssSemicolon,

...

};

enum eVariantType {

vtInt,

vtReal,

vtString,

vtBool,

None

};

Реализация всех классов для всех групп токенов.

class Token {

public:

eTokenType type;

virtual void Print() { };

};

class IdentToken : public Token {

public:

string ident;

void Print( ) {

};

};

class KeyWordToken : public Token {

public:

eKeyWords kw;

void Print() {

switch (kw)

{

};

};

class ConstToken : public Token {

public:

std::variant<int, float, std::string, bool> data;

int Get\_Int() {

if (data.index() == vtInt) {

return get<int>(data);

}

};

float Get\_Float() {

if (data.index() == vtInt) {

return get<float>(data);

}

};

void Print() {

if (data.index() == vtInt) {

cout << std::get<int>(data) << ' ';

}

else if (data.index() == vtReal) {

cout << std::get<float>(data) << ' ';

}

};

};

class SpecialSymbols : public Token {

public:

eSpecialSymbols ss;

void Print() {

switch (ss)

{

case ssComma:

cout << "ssComma" << ' ';

break;

case ssDot:

cout << "ssDot" << ' ';

break;

case ssRightCurveBrascet:

cout << "ssRightCurveBrascet" << ' ';

break;

}

}

};

Благодаря такой удобной иерархии код становиться легче читать и писать.

Это наиболее интересная вещь в лексическом анализаторе для меня, т.к раньше я и не задумывался как удобно могут использоваться перечисления и вообще ООП в C++.

## **Реализация.**

class Lexer {

private:

size\_t position;

size\_t row;

ifstream file;

string exceptions = "";

string input;

void Read\_input() {

if (position >= input.size()) {

input = "";

getline(file, input);

position = 0;

row++;

}

}

public:

Lexer() {

position = 0;

row = 1;

file.open("D:\\myCompiler\\PascalCode.txt");

getline(file, input);

}

~Lexer() {

file.close();

}

void print\_lex\_errors() {

cout << exceptions;

}

pair<int, int> get\_token\_cords()

{

return pair<int, int>(row, position);

}

std::unique\_ptr<Token> getNextToken() {

Read\_input();

while (position < input.size() && isspace(input[position])) position++; // skip space

if (position >= input.size()) {

return nullptr;

}

char currentChar = input[position];

if (currentChar == ':') { // special simbols

position++;

if (position < input.size() && input[position] == '=') {

position++;

auto token = make\_unique<SpecialSymbols>();

token->type = ttSpecialSymbols;

token->ss = ssAssigment;

return token;

}

else {

auto token = make\_unique<SpecialSymbols>();

token->type = ttSpecialSymbols;

token->ss = ssColon;

return token;

}

}

else if (isalpha(currentChar)) { // kw, ident

string identifier;

auto kwToken = make\_unique<KeyWordToken>();

kwToken->type = ttKeywords;

while (position < input.size() && (isalpha(input[position]) || isdigit(input[position])))

identifier += input[position++];

if (identifier == "program") {

kwToken->kw = kwProgram;

return kwToken;

}

auto idToken = make\_unique<IdentToken>();

idToken->type = ttIdentifier;

idToken->ident = identifier;

return idToken;

}

else if (isdigit(currentChar) || currentChar == '\'') { // constatnts

string number;

try

{

while (position < input.size() && (isdigit(input[position]) || input[position] == '.')) {

number += input[position++];

}

}

catch (const std::exception& exp)

{

exceptions += exp.what();

exceptions += " Row position: " + to\_string(row) + " Position in line: " + to\_string(position) + '\n';

}

if (number.find('.') != string::npos) {

try

{

auto realToken = make\_unique<ConstToken>(); // real

if (count(number.begin(), number.end(), '.') > 1) throw exception("Lex error in code!");

else {

realToken->type = ttConstants;

realToken->data = stof(number);

return realToken;

}

}

catch (const std::exception& exp)

{

...

}

}

else if (currentChar == '\'') { // string

position++;

while (position < input.size() && input[position] != '\'')

number += input[position++];

try

{

if (position == input.size() && input[position] != '\'')

throw exception("Lex error in code!");

}

auto stringToken = make\_unique<ConstToken>(); // string

stringToken->type = ttConstants;

stringToken->data = number;

position++;

return stringToken;

}

else {

auto intToken = make\_unique<ConstToken>(); // int

intToken->type = ttConstants;

intToken->data = stoi(number);

return intToken;

}

}

auto token = make\_unique<Token>();

token->type = UNKNOWN;

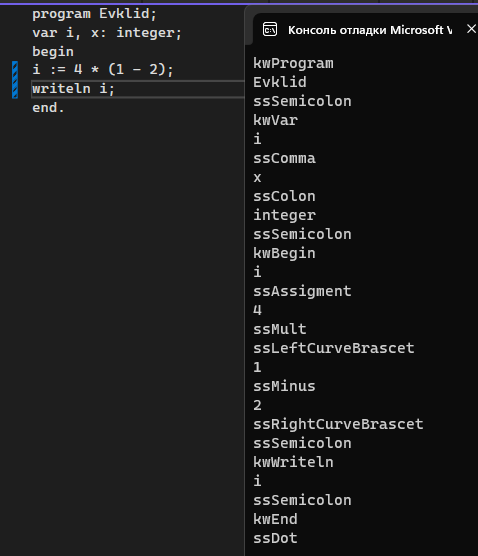
return token;

}

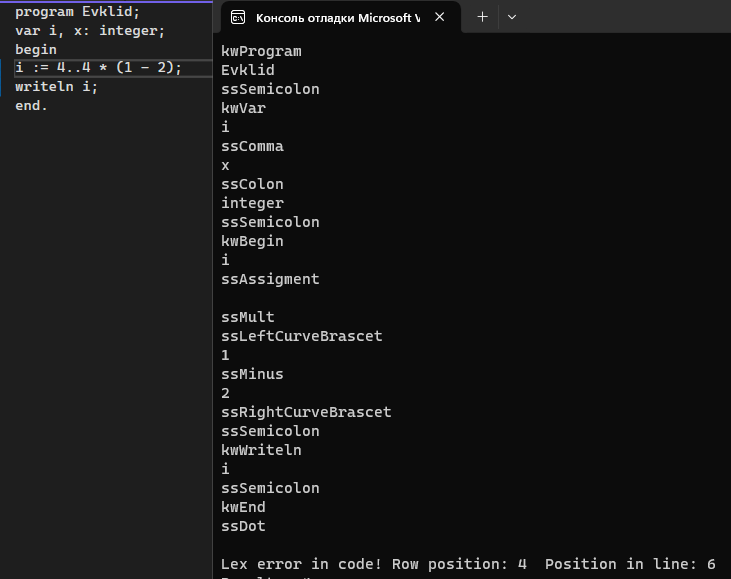
};

## **Тестирование.**

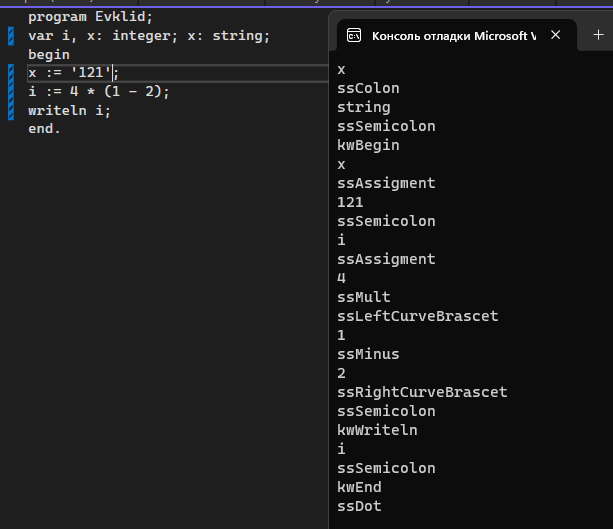
1.



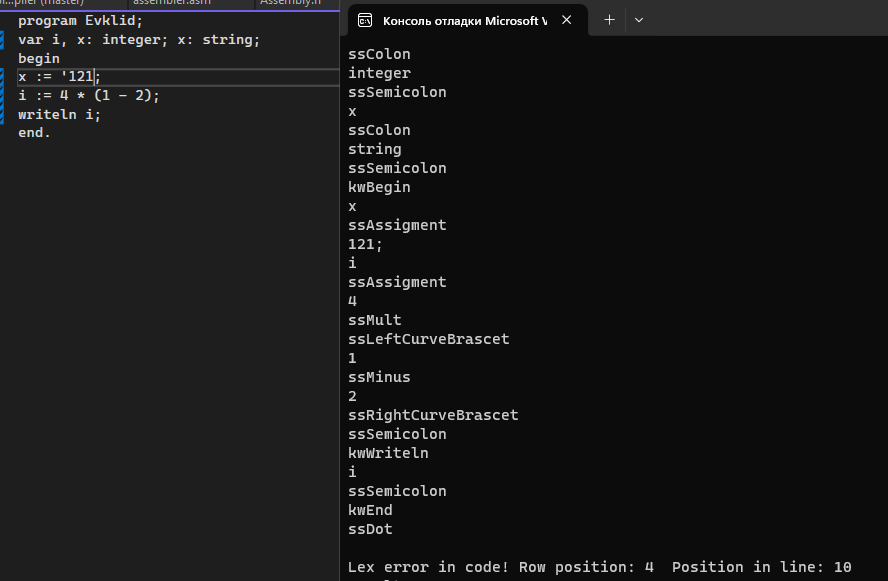
2.



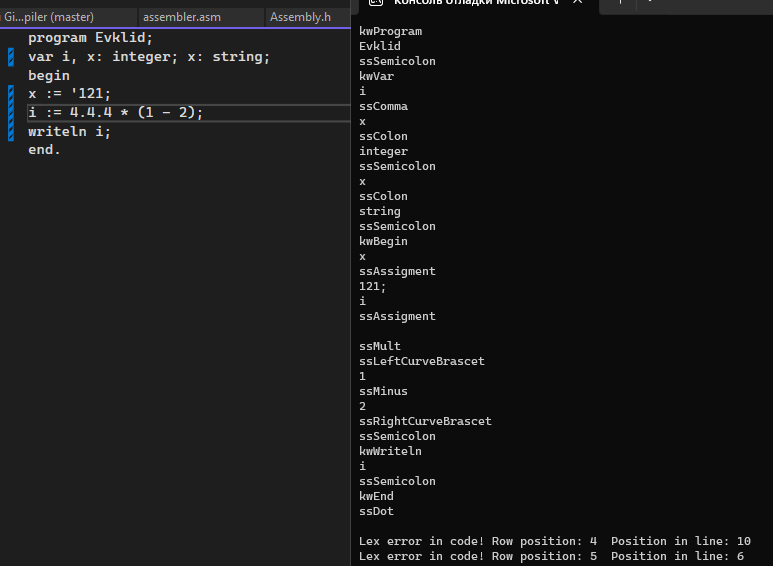
3.



4.



5.

****

# **Модуль 2. Синтаксический анализатор.**

## **Теория и общие сведения.**

В синтаксическом анализаторе производится анализ синтаксических конструкций по БНФ из книжки Залоговой.

Форма Бэкуса — Наура (сокр. БНФ) — формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории.

Все правила, описывающие синтаксис языка, имеют вид: <S>: = L где S — обозначение синтаксической конструкции, а L может состоять из символов языка, синтаксических конструкций и метасимволов «|» и «{}».

Для каждого правила необходимо описать функцию, тело которой является результатом некоторого преобразования правой части этого правила: procs () {T (L)}

При выходе из конструкции мы должны гарантировать, что находимся на токене следующей конструкции.

Нейтрализация ошибок.

При нарушении синтаксической конструкции мы пропускаем токены, пока не встретим знакомый токен другой конструкции.

Примерный вид:

if (!belong (token, starters))

{

error ( ... ); skipto (starters);

}

Здесь функция belong проверяет принадлежность token множеству starters. Функция skipto пропускает token входного потока, пока не встретится один из token множества, указанного в качестве параметра.

## **Моделирование.**

В лексическом анализаторе уже находится функция возвращающая следующий токен, поэтому проверка правильности конструкций происходит довольно просто.

Мы расписываем все БНФ как отдельные блоки при возникновении ошибки пропускам токены пока не встретим знакомый.

Анализ происходит путем нисходящей рекурсии по блокам БНФ.

Пример описания конструкции:

void block( ) // block

{

try

{

var\_block();

op\_block();

}

catch (const std::exception& exp) { skipto(starters\_type { kwVar, kwBegin }); cout << exp.what(); }

}

## **Реализация.**

class Syntax

{

public:

Syntax(shared\_ptr<Lexer> lexer) { this->lexer = lexer; }

void run\_syntax() {

program();

}

private:

shared\_ptr<Token> curToken;

shared\_ptr<Lexer> lexer;

void error(int index) {

switch (index)

{

case 1:

cout << "ошибка в разделе описаний\n";

break;

}

}

void getNext() {

this->curToken = this->lexer->getNextToken();

}

eSpecialSymbols get\_spec() {

try

{

return dynamic\_cast<SpecialSymbols\*>(curToken.get())->ss;

}

catch (const std::exception& exp)

{

cout << exp.what();

}

}

}

void accept(eSpecialSymbols ss)

{

// For SpecialSumbols

try

{

if (curToken == NULL || curToken->type != ttSpecialSymbols || get\_spec() != ss) throw exception("expected spec symbol op\n");

else getNext();

}

}

void accept(eKeyWords kw)

{

// For KeyWords

try

{

if (curToken == NULL || curToken->type != ttKeywords || get\_keyword() != kw) throw exception("expected another key word\n");

else getNext();

}

}

bool belong(unordered\_set<eKeyWords> starters)

{

for (const auto& starter : starters) {

if (curToken != NULL && curToken->type == ttKeywords && get\_keyword() == starter) {

return true;

}

}

return false;

}

void skipto(unordered\_set<eKeyWords> followers)

{

while (!belong(followers))

{

getNext();

}

}

void program() {

unordered\_set<eKeyWords>followers = { };

getNext();

accept(kwProgram);

accept\_ident();

accept(ssSemicolon);

block(followers);

accept(ssDot);

}

void block(unordered\_set<eKeyWords> followers) // block

{

var\_block(followers);

op\_block(followers);

}

void var\_block(unordered\_set<eKeyWords> followers) /\* анализ конструкции <раздел переменных> \*/

{

if (!belong(st\_var))

{

error(1);

skipto2(st\_var, followers);

}

if (curToken -> type == ttKeywords && get\_keyword() == kwVar && belong(st\_var))

{

accept(kwVar);

same\_var(followers);

accept(ssSemicolon);

}

}

void same\_var(unordered\_set<eKeyWords>followers) /\* анализ конструкции <описание однотипных переменных> \*/

{

accept\_ident();

while (get\_spec() == ssComma)

{

accept(ssComma);

accept\_ident();

}

accept(ssColon);

type(followers);

}

void type(unordered\_set<eKeyWords>followers)

{

accept\_ident();

}

void op\_block(unordered\_set<eKeyWords>followers)

{

compoundstatement(followers);

}

void compoundstatement(unordered\_set<eKeyWords>followers) /\* анализ конструкции <составной оператор> \*/

{

accept(kwBegin);

statement(followers);

while (curToken->type == ttSpecialSymbols && get\_spec() == ssSemicolon)

{

accept(ssSemicolon);

statement(followers);

}

accept(kwEnd);

}

void whilestatement(unordered\_set<eKeyWords>followers) /\* анализ конструкции <цикл с предусловием> \*/

{

accept(kwWhile);

expression();

accept(kwDo);

statement(followers);

}

void variable() /\* анализ конструкции <переменная> \*/

{

accept\_ident();

while (get\_spec() == ssLeftCurveBrascet || get\_spec() == ssDot)

{

switch (get\_spec())

{

case ssLeftCurveBrascet:

accept(ssLeftCurveBrascet);

expression();

while (get\_spec() == ssComma)

{

accept(ssComma);

expression();

}

accept(ssRightCurveBrascet);

break;

}

}

}

void ifstatement(unordered\_set<eKeyWords>followers)

{

accept(kwIf);

expression();

accept(kwThen);

statement(followers);

if (curToken->type == ttKeywords && get\_keyword() == kwElse)

{

accept(kwElse);

statement(followers);

}

}

void forstatement(unordered\_set<eKeyWords>followers {

}

void statement(unordered\_set<eKeyWords>followers) {

switch (curToken -> type)

{

case ttIdentifier:

assignment();

break;

case ttKeywords:

switch (get\_keyword())

{

case kwBegin:

compoundstatement(followers); break;

case kwIf:

ifstatement(followers); break;

}

}

void assignment()

{

variable();

accept(ssAssigment);

expression();

}

void expression()

{

eSpecialSymbols oper;

simple\_expression();

if (curToken->type == ttSpecialSymbols && realation\_operator())

{

simple\_expression();

}

}

void simple\_expression()

{

term();

while (curToken->type == ttSpecialSymbols && additive\_op())

{

term();

}

}

void term() // slogaemoe

{

factor();

if (curToken -> type == ttSpecialSymbols)

{

while (mult\_op())

{

factor();

}

}

}

void factor() // mnozitel

{

if (curToken->type == ttSpecialSymbols)

{

accept(ssLeftCurveBrascet);

expression();

accept(ssRightCurveBrascet);

}

else

{

switch (curToken->type) {

case ttConstants:

getNext(); // ПОЛУЧИЛИ Константу

break;

case ttIdentifier:

getNext();

break;

}

}

}

bool mult\_op() // mult op

{

eSpecialSymbols op;

op = get\_spec();

switch (op)

{

case ssDiv:

accept(ssDiv);

return true;

break;

case ssMult: }

return false;

}

bool additive\_op() {

eSpecialSymbols op;

op = get\_spec();

switch (op)

{

case ssMinus:

accept(ssMinus);

return true;

break;

case ssPlus:

}

return false;

}

bool realation\_operator() {

eSpecialSymbols op;

op = get\_spec();

switch (op)

{

case ssEqual:

accept(ssEqual);

return 1;

break;

case ssNoEqual:

case ssGreater:

case ssLesser:

}

return 0;

}

};

Каждый блок кода описывает свою БНФ.

Также есть функции проверки соответствия или получения значений токенов: realation\_operator, additive\_op, mult\_op, get\_keyword, get\_spec, get\_constant.

realation\_operator – проверка является ли оператор – оператором отношений: and or …

additive\_op - проверка является ли оператор – аддитивным оператором: + -

mult\_op - проверка является ли оператор – мультипликативным оператором: \* /

get\_keyword – получить значение keyword кинуть ошибку если не keyword

get\_spec – получить значение specsymbol кинуть ошибку если не specsymbol

get\_constant – получить значение constant кинуть ошибку если не constant

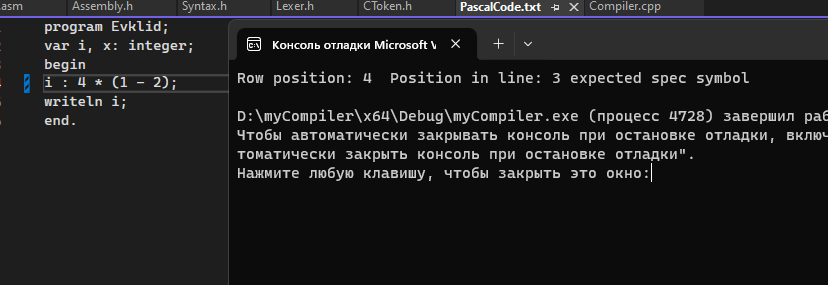
Также есть функция skipto для нейтрализации синтаксических ошибок, пропускает символы пока не встретит знакомый follower.

Функция accept – проверяет соответствие ожидаемого токена,

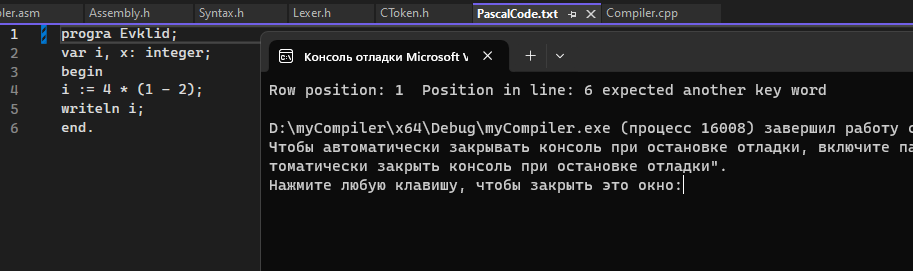
Если совпадает запрашивает следующий, иначе происходит нейтрализация ошибки – пропуск токенов пока не встретит знакомый.

## **Тесты.**

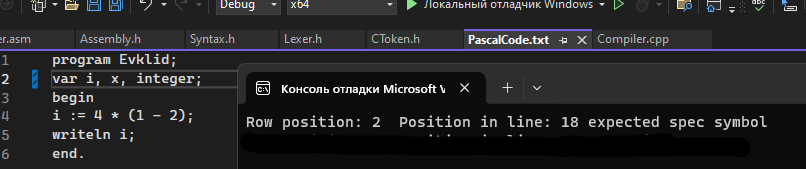
1.



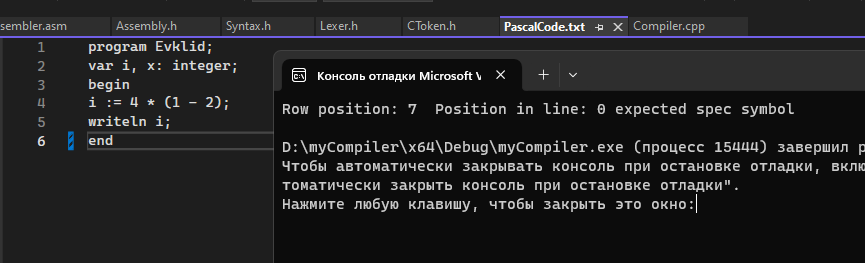
2.



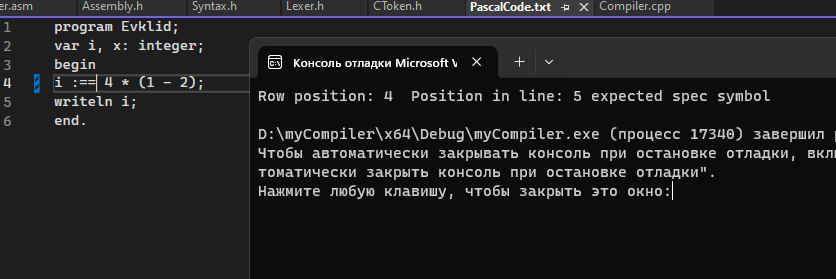
3.



4.



5.



# **Модуль 3. Семантический анализатор.**

## **Теория и общие сведения.**

Формальные правила описания синтаксиса языка программирования (формы Бэкуса—Наура, а также синтаксические диаграммы) служат основой для построения синтаксического анализатора, однако они дают неполное определение языка. Язык определяется с помощью формальных и неформальных описаний. Синтаксические правила языка программирования, которые задаются с помощью естественного языка (неформально), называются контекстными условиями.

Наличие контекстных условий в языке Паскаль связано с:

* идентификаторы используются для именования различных конструкций — типов, констант, переменных, процедур, функций.
* в программах один и тот же идентификатор может использоваться для обозначения различных типов.
* идентификаторы, описанные в некоторой области действия, могут использоваться только в соответствии с описанием внутри этой области.

Сформулируем контекстные условия, которые необходимо проверять при анализе программ.

* В любой области действия без внутренних по отношению к ней областей действия никакой идентификатор не может быть описан более одного раза.
* Каждому прикладному вхождению нестандартного идентификатора (стандартные идентификаторы — integer, boolean, real, true, false и др.) должно найтись соответствующее ему определяющее вхождение.
* Контекстные условия предполагают также проверку соответствия типов величин, входящих в синтаксические конструкции программ.

## **Моделирование.**

Для того чтобы отслеживать переменные в области видимости нужно где-то хранить их присутствие для этого создадим специальный map.

typedef map<string, pair<int, eVariantType>> var;

В нем будем хранить название переменной, в какой области видимости она находится, какого она типа.

Также будут храниться переменные нулевой области видимости такие как:

variables["INT\_MAX"] = pair(0, vtInt);

variables["INT\_MIN"] = pair(0, vtInt);

variables["PI"] = pair(0, vtReal);

variables["EXP"] = pair(0, vtReal);

Переменные заносятся в map в конструкции <описание однотипных переменных>, когда мы в этом блоке получаем идентификатор заносим переменную в текущую область видимости с None типом. Чтобы переменная получила свой тип мы должны будем дойти до конструкции <тип> где для всех переменных с None типом мы выдаем им их тип.

void check\_type()

{

eVariantType type;

if (curToken == NULL || curToken->type != ttIdentifier) throw exception(ident\_err);

else

{

if (get\_ident() == "integer") type = vtInt;

else if (get\_ident() == "bool") type = vtBool;

else if (get\_ident() == "string") type = vtString;

else if (get\_ident() == "real") type = vtReal;

else throw exception("unknown type");

}

Для того чтобы определить переменные на определенном уровне области видимости будем использовать функцию, которая будет возвращать map для элементов в текущей области видимости.

var variable\_level()

{

var level\_map;

for (const auto& element : variables) {

if (element.second.first <= cur\_scope) {

level\_map[element.first] = (element.first, pair(cur\_scope, element.second.second));

}

}

return level\_map;

}

Для того чтобы проверить возможна ли операция между переменными или выражениями мы будем проверять возможна ли операция между их типами.

eVariantType exist\_op(eVariantType type1, eVariantType type2, eKeyWords op)

Для булевских операций.

В фукнциях eVariantType exist\_op(eVariantType type1, eVariantType type2, eSpecialSymbols operation)

Для остальных операций.

Также нужно проверять приводимость типа выражения или переменной.  
В переменную типа bool нельзя положить real значение поэтому будет ошибка приводимости типов, но также к типу integer мы можем присвоить значение типа real (просто отбрасываем дробную часть).

## **Реализация.**

bool compatible\_to(eVariantType type1, eVariantType type2) // приведение типов функция compatible to

{

if (type1 == type2) {

return true;

}

else if (type1 == vtInt && type2 == vtReal) {

return true;

}

else if (type1 == vtReal && type2 == vtReal) {

return true;

}

return false;

}

eVariantType exist\_op(eVariantType type1, eVariantType type2, eSpecialSymbols operation) // возможна ли операция между переменными

{

switch (operation)

{

case ssEqual:

case ssNoEqual:

if (type1 == type2) {

return vtBool;

}

else {

return None;

}

break;

case eSpecialSymbols::ssPlus:

case eSpecialSymbols::ssMinus:

case eSpecialSymbols::ssMult:

if (type1 == vtInt && type2 == vtInt) {

return vtInt;

}

else if ((type1 == vtReal || type1 == vtInt) &&

(type2 == vtReal || type2 == vtInt)) {

return vtReal;

}

return None;

case eSpecialSymbols::ssDiv:

if ((type1 == vtReal || type1 == vtInt) &&

(type2 == vtReal || type2 == vtInt)) {

return vtReal;

}

return None;

case eSpecialSymbols::ssGreater:

if ((type1 == vtReal || type1 == vtInt) &&

(type2 == vtReal || type2 == vtInt)) {

return vtBool;

}

else if (type1 == vtString && type2 == vtString) {

return vtBool;

}

return None;

}

}

eVariantType exist\_op(eVariantType type1, eVariantType type2, eKeyWords op) // возможна ли операция между переменными

{

// Логические операции для типа bool

if ((op == kwAnd || op == kwOr|| op == kwNot) &&

(type1 == vtBool && type2 == vtBool)) {

return vtBool;

}

no\_op\_between;

return None;

}

void zero\_level\_variables() {

variables["INT\_MAX"] = pair(0, vtInt);

variables["INT\_MIN"] = pair(0, vtInt);

variables["PI"] = pair(0, vtReal);

variables["EXP"] = pair(0, vtReal);

}

var variable\_level()

{

var level\_map;

for (const auto& element : variables) {

if (element.second.first <= cur\_scope) {

level\_map[element.first] = (element.first, pair(cur\_scope, element.second.second));

}

}

return level\_map;

}

void check\_variables()

{

var cur\_level\_variables = variable\_level();

if (curToken == NULL || curToken->type != ttIdentifier) throw exception(ident\_err);

else if (cur\_level\_variables.find(get\_ident()) != cur\_level\_variables.end()) throw exception("variable already named\n");

else { variables[get\_ident()] = pair(cur\_scope, None); }// map <string, CType>

}

void check\_type()

{

eVariantType type;

if (curToken == NULL || curToken->type != ttIdentifier) throw exception(ident\_err);

else

{

if (get\_ident() == "integer") type = vtInt;

...

else throw exception("unknown type");

}

}

}

void var\_block() /\* анализ конструкции <раздел переменных> \*/

{

try

{

if (kw\_check == kwVar)

{

accept(kwVar);

cur\_scope++;

do

{

same\_var();

accept(ssSemicolon);

} while (curToken->type == ttIdentifier);

}

}

void same\_var() /\* анализ конструкции <описание однотипных переменных> \*/

{

try

{

check\_variables();

accept\_ident();

while (spec\_check == ssComma)

{

accept(ssComma);

check\_variables();

accept\_ident();

}

accept(ssColon);

type();

}

}

void type()

{

check\_type();

accept\_ident();

}

void assignment()

{

try

{

auto vartype = get\_variable\_type();

auto var = variable();

accept(ssAssigment);

auto exptype = expression();

asm\_file.assigment(var, vartype, exptype);

if (!compatible\_to(vartype, exptype)) compatible\_error;

}

eVariantType expression()

{

eSpecialSymbols res = ssNone;

auto ex1type = simple\_expression();

if (curToken->type == ttSpecialSymbols)

{

res = realation\_operator();

if (res != ssNone)

{

auto ex2type = simple\_expression();

ex1type = exist\_op(ex1type, ex2type, res);

}

}

return ex1type;

}

eVariantType simple\_expression()

{

eSpecialSymbols res = ssNone;

auto ex1type = term();

...

while (res != ssNone)

{

auto ex2type = term();

eVariantType temp = ex1type;

ex1type = exist\_op(ex1type, ex2type, res);

...

}

return ex1type;

}

eVariantType term() // slogaemoe

{

eSpecialSymbols res = ssNone;

auto ex1type = factor();

if (curToken->type == ttSpecialSymbols) res = mult\_op();

while (res != ssNone)

{

auto ex2type = factor();

eVariantType temp = ex1type;

ex1type = exist\_op(ex1type, ex2type, res);

...

}

return ex1type;

}

eVariantType factor() // mnozitel

{

eVariantType exptype = None;

if (curToken->type == ttSpecialSymbols)

{

accept(ssLeftCurveBrascet);

exptype = expression();

accept(ssRightCurveBrascet);

}

else

{

switch (curToken->type) {

case ttConstants:

switch (dynamic\_cast<ConstToken\*>(curToken.get())->data.index())

{

case 0:

exptype = vtInt;

break;

case 1:

exptype = vtReal;

break;

case 2:

exptype = vtString;

break;

case 3:

exptype = vtBool;

break;

default: /\* ошибка \*/

exptype = None;

}

accept(ttConstants);

break;

case ttIdentifier:

// check that it is variable

exptype = get\_variable\_type();

switch (exptype)

{

case vtInt:

case vtReal:

default:

break;

}

accept(ttIdentifier);

break;

}

}

return exptype;

}

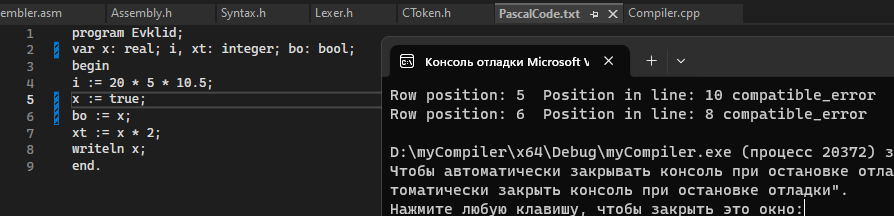
В каждом из блоков вычисления выражений возвращается тип полученного значения.

check\_type () – проверяет есть ли такой тип.

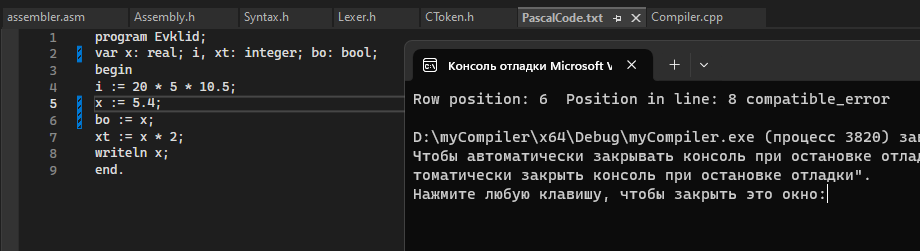
eVariantType exist\_op (eVariantType type1, eVariantType type2, eKeyWords op) – проверяет существует ли операция между переменными или выражениями (их типами).

## **Тесты.**

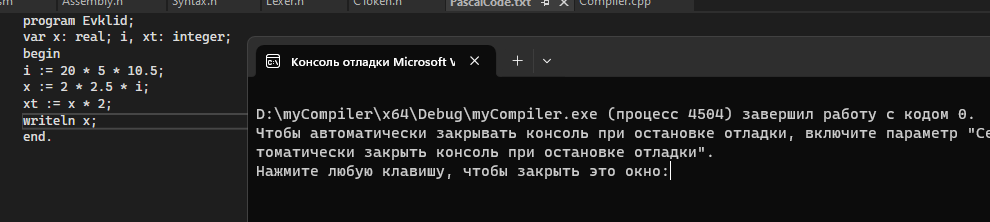
1)

****

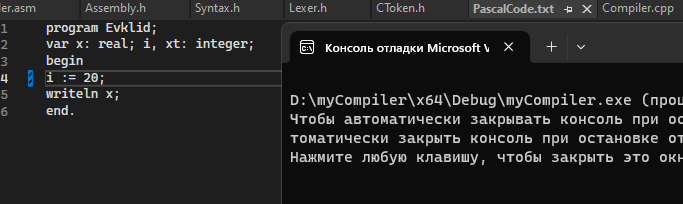
2)



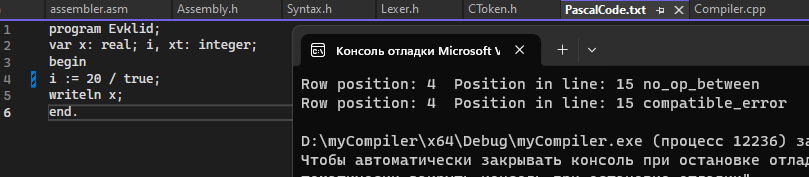
3)



4)



5)



# **Модуль 4. Генерация кода.**

## **Теория и общие сведения.**

Генерация кода — это машинно-зависимая часть компилятора, так как она определяется архитектурой конкретной вычислительной машины. Понятие архитектуры включает в себя все то, что предоставляет компьютер пользователю, программирующему на уровне машинных команд. Другими словами, архитектура определяет структуру оперативной памяти, организацию памяти процессора (регистры), способы адресации (определения местоположения операндов), способы представления данных и команды. Таким образом, согласно классической схеме компиляции, для реализации языка программирования на N вычислительных машинах с различной архитектурой необходимо создать один анализатор и N генераторов кода.

Результат генерации кода — объектная программа — может представлять собой последовательность машинных команд или программу на языке ассемблера.

Регистр процессора — сверхбыстрая память внутри процессора, предназначенная для хранения адресов и промежуточных результатов вычислений (регистр общего назначения/регистр данных) или данных, необходимых для работы самого процессора.

Регистры XMM (XMM0-XMM15) могут хранить одно 32-разрядное значение с плавающей точкой (то есть значение типа real4) или четыре значения с плавающей точкой одинарной точности (то есть вектор из 4-х значений.

Инструкции языка ассемблер:

Инструкция add выполняет сложение двух операндов, а инструкция sub - вычитание.

Инструкции mul и imul умножает два целых числа. imul умножает числа со знаком, а mul - беззнаковые числа.

Для деления чисел в архитектуре x86-64 предназначены инструкции div и idiv. idiv делит два числа со знаком, а div - беззнаковые числа.

В моем случае на языке assembler MASM x64.Visual Studio включает как 32-разрядную, так и 64-разрядную размещенные версии MASM (ассемблера макросов Microsoft) для работы с кодом x64. Названный ml64.exe, это ассемблер, который принимает язык ассемблера x64.

## **Моделирование.**

Для того чтобы перевести программу с языка Pascal на язык ассемблера мы будем основываться в каком месте анализа программы мы находимся и будем переводить в код ассемблера прямо на лету.

Для вывода информации на экран консоли используется библиотека C, из который мы будем пользоваться командой printf.

fmtStr2 byte 'Result is: %f', 10, 0 Шаблон строки для вывода результата.

Несложно перевести объявление переменных из паскаля в assembler, просто будем записывать переменную и ее тип в функции check\_type.

Для того чтобы разграничить взаимодействие с ассемблер файлом будем хранить специальный класс Assembly, где будут описаны специальные функции для перевода с языка паскаль на язык ассемблера.

Вид класса:

class Assembly

{

public:

Assembly() {

out.open(path);

out << "includelib ucrt.lib\nincludelib legacy\_stdio\_definitions.lib\nEXTERN printf : PROC\n.data\ntemp real4 ?\nfmtStr byte 'Result is: %d', 10, 0\nfmtStr2 byte 'Result is: %f', 10, 0\n";

}

~Assembly() {

out.close();

}

}

Есть определенная сложность работы с real типом в ассемблере, т.к объявлять их прямо в блоке .code нельзя, для этого я решил переводить число в real использую 2 целых числа из кода на Pascal для этого я передаю целую часть числа, а также дробную с определенным количеством 10^m, где m – количество чисел после запятой. Так мне удается записывать real значения прямо в блоке .code, используя вышеописанное и регистр xmm для работы с real значениями.

## **Реализация.**

class Assembly

{

public:

Assembly() {

out.open(path);

out << "includelib ucrt.lib\nincludelib legacy\_stdio\_definitions.lib\nEXTERN printf : PROC\n.data\ntemp real4 ?\nfmtStr byte 'Result is: %d', 10, 0\nfmtStr2 byte 'Result is: %f', 10, 0\n";

}

~Assembly() {

out.close();

}

void value\_to\_int\_stack(string data) {

string code = "\nmov rdx," + data + "\npush rdx";

out << code;

}

void value\_to\_float\_stack\_const(string floatnum) {

int tens = 1;

string code = "";

string int\_part = "";

string float\_part = "";

for (size\_t i = 0; i < floatnum.length(); i++)

{

if (isdigit(floatnum[i])) int\_part += floatnum[i];

if (floatnum[i] == '.')

{

for (size\_t j = i + 1; j < floatnum.length(); j++)

{

if (floatnum[j] != '0') float\_part += floatnum[j];

tens \*= 10;

}

break;

}

}

if (int\_part == "0")

{

code = "\nmov rcx," + float\_part + "\ncvtsi2ss xmm1, rcx" + "\nmovd temp, xmm1" + "\nfld temp";

code += "\nmov rcx," + to\_string(tens) + "\ncvtsi2ss xmm1, rcx" + "\nmovd temp, xmm1" + "\nfld temp";

code += "\nfdiv";

}

else if(float\_part != "")

{

code = "\nmov rcx," + float\_part + "\ncvtsi2ss xmm1, rcx" + "\nmovd temp, xmm1" + "\nfld temp";

code += "\nmov rcx," + to\_string(tens) + "\ncvtsi2ss xmm1, rcx" + "\nmovd temp, xmm1" + "\nfld temp";

code += "\nfdiv";

code += "\nmov rcx," + int\_part + "\ncvtsi2ss xmm1, rcx" + "\nmovd temp, xmm1" + "\nfld temp" + "\nfadd";

}

else

{

code += "\nmov rcx," + int\_part + "\ncvtsi2ss xmm1, rcx" + "\nmovd temp, xmm1" + "\nfld temp";

}

out << code;

}

void value\_to\_float\_stack\_ident(string data) {

string code = "\nfld " + data;

out << code;

}

void add(eVariantType vt1, eVariantType vt2){

if (vt1 == vtInt && vt2 == vtInt)

{

out << "\npop rdx\npop rax\nadd rax, rdx\npush rax";

}

else if(vt1 == vtReal && vt2 == vtReal)

{

out << "\nfadd";

}

else // случ int - float | float - int

{

out << "\npop rax\ncvtsi2ss xmm1, rax\nmovd temp, xmm1\nfld temp\nfadd";

}

}

void substract(eVariantType vt1, eVariantType vt2) {

if (vt1 == vtInt && vt2 == vtInt)

{

out << "\npop rdx\npop rax\nsub rax, rdx\npush rax";

}

else if (vt1 == vtReal && vt2 == vtReal)

{

out << "\nfsub";

}

else // случ int - float | float - int

{

out << "\npop rax\ncvtsi2ss xmm1, rax\nmovd temp, xmm1\nfld temp\nfsub";

}

}

void mult(eVariantType vt1, eVariantType vt2) {

if (vt1 == vtInt && vt2 == vtInt)

{

out << "\npop rdx\npop rax\nmul rdx\npush rax";

}

else if (vt1 == vtReal && vt2 == vtReal)

{

out << "\nfmul";

}

else // случ int - float | float - int

{

out << "\npop rax\ncvtsi2ss xmm1, rax\nmovd temp, xmm1\nfld temp\nfmul";

}

}

void division(eVariantType vt1, eVariantType vt2) {

if (vt1 == vtInt && vt2 == vtInt)

{

out << "\npop rbx\npop rax\ncqo\nidiv rbx\npush rax";

}

else if (vt1 == vtReal && vt2 == vtReal)

{

out << "\nfdiv";

}

else // случ int - float | float - int

{

out << "\npop rax\ncvtsi2ss xmm1, rax\nmovd temp, xmm1\nfld temp\nfdiv";

}

}

void assigment(string var, eVariantType vt1, eVariantType vt2) {

string code = "";

if (vt2 == vtInt && vt1 == vtInt)

{

code = "\npop rax\nmov " + var + string(", rax");

out << code;

}

else if (vt1 == vtReal && vt2 == vtReal)

{

code = "\nfstp " + var;

out << code;

}

else if(vt1 == vtInt)

{

code = "\nfistp " + var;

out << code;

}

}

void write\_variable(string var, eVariantType vt) {

switch (vt)

{

case vtInt:

out << var + " QWORD ?\n";

break;

case vtReal:

out << var + " real4 ?\n";

break;

case None:

break;

}

}

void prossed() {

string code = "\n.code\nprint proc\nsub rsp, 20h\nlea rcx, fmtStr; param 1: format string\nmov rdx, rax\ncall printf\nadd rsp, 20h; Restore the stack\nret\nprint endp\n" +

string ("\nprintFloat proc\nsub rsp, 20h\nmovss xmm0, [temp]\ncvtss2sd xmm1, xmm0\nmovd rdx, xmm1\nlea rcx, fmtStr2\ncall printf\nadd rsp, 20h\nret\nprintFloat endp\nexpression proc\n");

out << code;

}

void ending() {

out << "\nret\nexpression ENDP\nend";

}

void print\_to\_console(string var, eVariantType vt) {

switch (vt)

{

case vtInt:

out << "\nmov rax," + var + "\ncall print";

break;

case vtReal:

out << "\nfld " + var + "\nfstp temp" + "\ncall printfloat";

break;

default:

break;

}

}

private:

ofstream out;

string path = "assembler.asm";

};

value\_to\_int\_stack – записывает целочисленное значение в стек

value\_to\_float\_stack\_const – записывает real значение в стек

value\_to\_float\_stack\_ident - записывает real из переменной значение в стек

add – записывает выполнение операции сложения

substract – записывает выполнение операции вычитания

mult – записывает выполнение операции умножения

division – записывает выполнение операции деления

assigment – записывает выполнение операции присваивания

write\_variable – записывает переменную и ее тип

prossed – записывает продолжение файла начало блока .code,

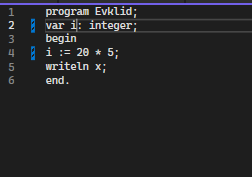
а также функции print и printfloat на ассемблере

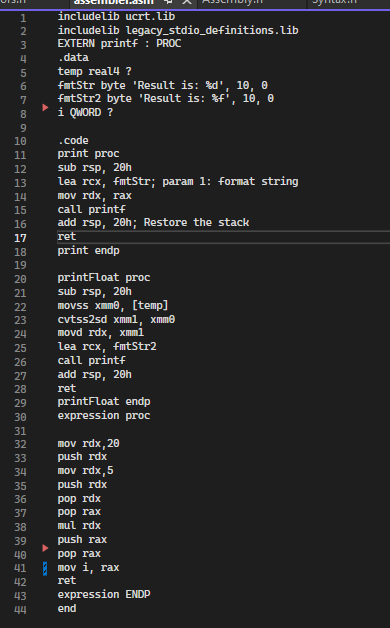
ending – записывает концовку файла

print\_to\_console – записывает вывод переменной на экран

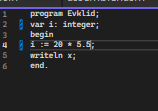
## **Тесты.**

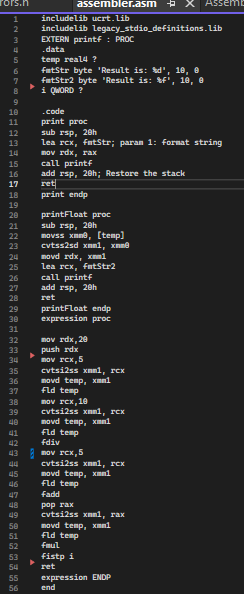
1)



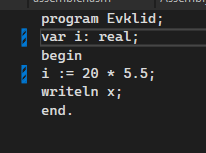


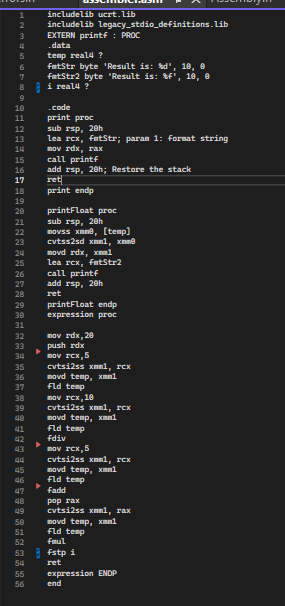
2)





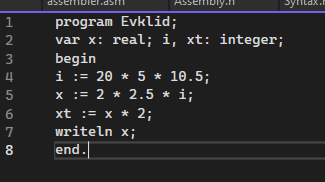
3)

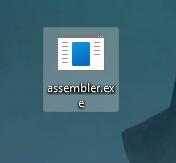


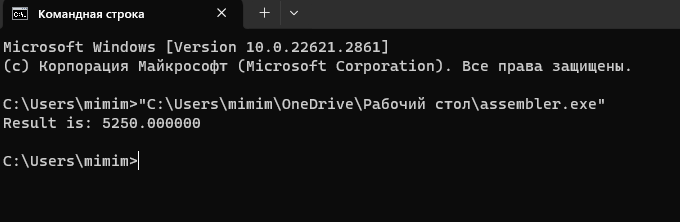


Нормальный тест программы.

Для того чтобы не копировать большой скрин в файл покажу только результат уже скомпонованного в exe шник программы.







Считает все выражения корректно.

# **Общая часть кода.**

В моем коде присутствуют директивы #define для лучшей читаемости кода и для лучшего понимания.

#define starters\_type vector<variant<eSpecialSymbols, eKeyWords>>

#define kw\_check curToken -> type == ttKeywords && get\_keyword()

#define spec\_check curToken -> type == ttSpecialSymbols && get\_spec()

Также некоторые повторяющиеся ошибки вынесены в отдельный класс Errors для удобства и лучшей читаемости.

#define keyword\_err ("Row position: " + to\_string(lexer->get\_token\_cords().first) + " Position in line: " + to\_string(lexer->get\_token\_cords().second) + " expected another key word\n").c\_str()

#define spec\_symbol\_err ("Row position: " + to\_string(lexer->get\_token\_cords().first) + " Position in line: " + to\_string(lexer->get\_token\_cords().second) + " expected spec symbol\n").c\_str()

#define token\_type\_err ("Row position: " + to\_string(lexer->get\_token\_cords().first) + " Position in line: " + to\_string(lexer->get\_token\_cords().second) + " expected another type\n").c\_str()

#define ident\_err ("Row position: " + to\_string(lexer->get\_token\_cords().first) + " Position in line: " + to\_string(lexer->get\_token\_cords().second) + " expected ident\n").c\_str()

#define compatible\_error cout << "Row position: " + to\_string(lexer->get\_token\_cords().first) + " Position in line: " + to\_string(lexer->get\_token\_cords().second) + " compatible\_error\n";

#define no\_op\_between cout << "Row position: " + to\_string(lexer->get\_token\_cords().first) + " Position in line: " + to\_string(lexer->get\_token\_cords().second) + " no\_op\_between\n";

Оглавление

[**Модуль 1. Лексический анализатор.** 2](#_Toc153706580)

[**Теория и общие сведения.** 2](#_Toc153706581)

[**Моделирование.** 2](#_Toc153706582)

[**Реализация.** 6](#_Toc153706583)

[**Тестирование.** 8](#_Toc153706584)

[**Модуль 2. Синтаксический анализатор.** 11](#_Toc153706585)

[**Теория и общие сведения.** 11](#_Toc153706586)

[**Моделирование.** 12](#_Toc153706587)

[**Реализация.** 12](#_Toc153706588)

[**Тесты.** 17](#_Toc153706589)

[**Модуль 3. Семантический анализатор.** 18](#_Toc153706590)

[**Теория и общие сведения.** 18](#_Toc153706591)

[**Моделирование.** 19](#_Toc153706592)

[**Реализация.** 20](#_Toc153706593)

[**Тесты.** 24](#_Toc153706594)

[**Модуль 4. Генерация кода.** 26](#_Toc153706595)

[**Теория и общие сведения.** 26](#_Toc153706596)

[**Моделирование.** 27](#_Toc153706597)

[**Реализация.** 27](#_Toc153706598)

[**Тесты.** 31](#_Toc153706599)

[**Общая часть кода.** 34](#_Toc153706600)