

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ  
ФГБОУ ВПО «ПЕРМСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ  
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»

ОТЧЕТ  
по реализации компилятора для подмножества языка  
Паскаль

Работу выполнили  
студент гр. ПМИ-1,2  
Захаров И.В. \_\_\_\_\_  
«\_\_» \_\_\_\_\_ 2021

Проверил  
ассистент кафедры МОВС  
Пономарев Ф.А. \_\_\_\_\_  
«\_\_» \_\_\_\_\_ 2021

Пермь 2021

## СОДЕРЖАНИЕ

1	Постановка задачи .....	5
2	Модуль ввода-вывода.....	7
2.1	Описание .....	7
2.2	Проектирование.....	7
2.3	Реализация .....	8
2.4	Тестирование .....	11
2.4.1	Создание из строки исходного кода.....	11
2.4.2	Создание из файла.....	11
2.4.3	Получение текущего символа, позиции.....	12
2.4.4	Вывод сообщения об ошибке.....	12
3	Лексический анализатор .....	13
3.1	Описание .....	13
3.2	Проектирование.....	13
3.3	Реализация .....	14
3.3.1	Лексический анализатор.....	14
3.3.2	Токен .....	18
3.3.3	Токен-константа .....	18
3.3.4	Токен-идентификатор.....	18
3.3.5	Токен-оператор.....	19
3.3.6	Хэш-таблицы и перечислимый тип для операторов.....	19
3.4	Тестирование .....	20

4	Синтаксический анализатор .....	22
4.1	Описание .....	22
4.2	Проектирование.....	22
4.3	Реализация .....	23
4.3.1	Функция next_token.....	23
4.3.2	Функция ассерт .....	23
4.3.3	Функции реализующие БНФ.....	25
4.3.4	Полное описание класса .....	30
4.4	Тестирование .....	38
5	Семантический анализатор.....	40
5.1	Описание .....	40
5.2	Проектирование.....	40
5.3	Реализация .....	40
5.3.1	Функция добавления переменной .....	40
5.3.2	Функция приведения типов.....	41
5.3.3	Полное описание класса модуля семантического анализатора .....	42
5.3.4	Описание класса типов .....	52
5.4	Тестирование .....	54
6	Транслятор.....	57
6.1	Описание .....	57
6.2	Проектирование.....	57
6.3	Реализация .....	58

6.3.1	Функция добавления переменной .....	58
6.3.2	Функция для вызова инструкции исходя из переданного оператора .....	59
6.3.3	Функция вызова метода вывода в консоль(writeln).....	60
6.3.4	Функция вызова метода ввода из консоли(readln).....	60
6.3.5	Функция для смены знака значения на стеке .....	61
6.3.6	Функция для загрузки константного значения в стек .....	61
6.3.7	Полный код модуля транслятора.....	61
6.4	Тестирование .....	76
6.4.1	Ввод и вывод числа .....	76
6.4.2	Проверка введенного числа на простоту .....	77
6.4.2	Нахождение корня для полинома третьей степени.....	79

## 1 Постановка задачи

Глобальное задание: написать компилятор для подмножества языка Паскаль.

Задание разбивается на отдельные этапы:

1. Модуль ввода-вывода (8 баллов, оценивается совместно с лексическим анализатором).
2. Лексический анализатор (12 баллов, оценивается совместно с модулем ввода-вывода).
3. Синтаксический анализатор (12 баллов) с нейтрализацией синтаксических ошибок (8 баллов).
4. Семантический анализатор с нейтрализацией семантических ошибок (20 баллов).
5. Генерация кода (25 баллов).

Реализовать следующие разделы программы:

Основные разделы программы: раздел описания переменных, раздел операторов. Переменные стандартных типов (Boolean, integer, real, char). Числовые константы. Арифметическое выражение (в выражении допустимы только константы, переменные, операции +, -, \*, / и скобки). Оператор присваивания и составной оператор.

Раздел описания типов. Выражение (полностью, включая арифметические, логические операции, сравнения и т.д., но только над константами и простыми переменными (не индексированные, не поля записи, не указатели)). Условный оператор (if). Оператор цикла с предусловием (while).

Описание функций. Вызов функции в выражении. Операторы циклов repeat, for.

Описание функций. Вызов функции в выражении. Оператор выбора (case).

Описание процедур. Операторы вызова процедуры, циклов repeat, for.

Описание процедур. Операторы вызова процедуры, выбора (case).

Пользовательские скалярные типы (перечислимый, интервальный). Описание массивов. Индексированные переменные в выражении. Операторы циклов repeat, for.

Раздел описания констант. Пользовательские скалярные типы (перечислимый, интервальный). Описание массивов. Индексированные переменные в выражении.

Описание записей (без вариантной части). Переменные - поля записи в выражении.

Операторы присоединения (with) , циклов repeat, for.

Описание записей (без вариантной части). Переменные - поля записи в выражении.

Операторы присоединения (with), выбора (case).

Интервальный тип. Описание массивов. Описание записей (без вариантной части).

Индексированные переменные и поля записи в выражении. Оператор присоединения

(with).

Интервальный тип. Описание массивов. Индексированные переменные в выражении.

Описание процедур. Оператор вызова процедуры.

Ссылочные типы данных. Описание функций. Вызов функции и указатели в выражении.

Раздел описания констант. Ссылочные типы данных. Указатели в выражениях.

Оператор выбора (case).

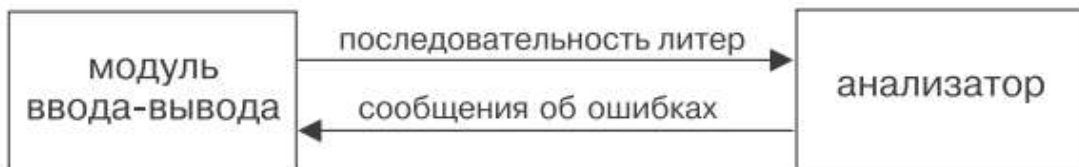
Описание записей. Переменные - поля записи в выражении. Оператор присоединения

(with).

## 2 Модуль ввода-вывода

### 2.1 Описание

Модуль ввода-вывода считывает последовательность литер исходной программы с внешнего устройства и передает их анализатору. Анализатор проверяет, удовлетворяет ли эта последовательность литер правилам описания языка, и формирует(в случае необходимости) сообщения об ошибках. Такое взаимодействие между модулем ввода-вывода и анализатором можно представить в виде схемы:



Для печати сообщений об ошибках будем хранить для каждого символа его позицию.

Сделаем ввод программы доступным в двух вариантах: по пути к файлу программы, либо же по строке в которой хранится код программы.

### 2.2 Проектирование

В модуле ввода-вывода должны быть определены следующие методы:

- Получение следующего символа
- Получение текущей позиции
- Генерация текста кода программы с указанием ошибок

Дополнительно для удобства создадим перечислимый тип `OpenState` – принимающий два состояния – открытие с помощью пути к файлу программы, либо же состояние, когда код программы был передан строкой.

Для поддержания текущей позиции заведем специальную переменную `position`.

В случае когда исходный код был получен с помощью файла, то текст кода с сгенерированными ошибками будем сохранять по тому же пути что и исходный, но с определенным суффиксом в конце имени файла. Для суффикса заведем строку-константу для удобства.

В функцию вывода ошибок передается указатель на экземпляр класса обработчика ошибок и вызывается соответствующая функция для получения всех ошибок.

После этого мы будем идти построчно, а затем посимвольно и сопоставлять текущую позицию с позициями ошибок(ошибки отсортированы по возрастанию позиции в которой находится ошибка).

Если в текущей позиции есть ошибка, то добавим ее в вектор ошибок текущей строки. После обработки всей строки дополнительно выведем информацию об ошибках и их позициях следующим образом:

Допустим у нас в строке нашлись следующие ошибки(позиции ошибок указаны относительно текущей строки): {2, 'Вычисленное выражение имеет другой тип в отличие от переменной'}, {7, 'Данную операцию нельзя применить к этим операндам'}

А текущая рассматриваемая строка такая: " $x := y + I;$ "

То текст с ошибками будет сгенерирован следующим образом:

$x := y + I;$

^     ^

/     *Данную операцию нельзя применить к этим операндам*

*Вычисленное выражение имеет другой тип в отличие от переменной*

## 2.3 Реализация

Ниже представлено описание класса и перечислимого типа в заголовочном файле

IO\_Module.h

```
enum OpenState {
    ByFile,
    ByString
};

class IO_Module
{
private:
    const string OUTPUT_PREFIX = "ANALYZED";
    string path_to_file;
    fstream input_stream;
    string code;

    OpenState open_state;

    int position = -1;

    string change_filename(string& path);
public:
    IO_Module() {};
    IO_Module(const string& input);

    char get_next_char();
    void write_errors(ErrorHandler* error_handler);
    int get_current_position();

    ~IO_Module();
};
```

Определим все методы в файле-источнике IO\_Module.cpp

```
IO_Module::IO_Module(const string& input)
{
```



```

        try
        {
            input_stream.open(input);
            path_to_file = input;
            open_state = ByFile;
        }
        catch (exception e)
        {
            code = input;
            open_state = ByString;
        }
    }

IO_Module::~~IO_Module()
{
    input_stream.close();
}

char IO_Module::get_next_char()
{
    position++;

    char c;
    if (input_stream.is_open())
    {
        c = (char)input_stream.get();
    }
    else
    {
        if (position < code.size())
            c = code[position];
        else
            c = EOF;
    }

    if (c == EOF)
        input_stream.close();
    return c;
}

int IO_Module::get_current_position()
{
    return position;
}

void IO_Module::write_errors(ErrorHandler* error_handler)
{
    vector<Error*> errors = error_handler->get_errors();
    IO_Module* io_helper;

    string result;

    if (open_state == ByFile)
        io_helper = new IO_Module(path_to_file);
    else
        io_helper = new IO_Module(code);
}

```

```

int current_error = 0;
char c;
do
{
    // Построчно будем искать позиция с ошибками
    int current_position_in_line = 0;
    ErrorHandler* error_handler_in_line = new
ErrorHandler();

    string first_line;

    do
    {
        c = io_helper->get_next_char();

        // Если еще есть ошибки и на текущей позиции
        есть ошибка, тогда добавим ее
        if (current_error < errors.size() &&
            errors[current_error]->position ==
io_helper->get_current_position())
        {
            error_handler_in_line-
>add_error(errors[current_error]->info, current_position_in_line);
            current_error++;
        }

        current_position_in_line++;
        first_line += (c == '\\t' ? '\\t' : ' ');
        result += (c == EOF ? '\\n' : c);

    } while (c != EOF && c != '\\n');

    vector<Error*> errors_in_line = error_handler_in_line-
>get_errors();

    int errors_count_in_line = errors_in_line.size();
    if (errors_count_in_line == 0) continue;
    // Если в строке были ошибки то создадим первую строку
типа:
    // ^           ^           ^
    // Указатели на позиции ошибок

    for (auto e : errors_in_line)
        first_line[e->position] = '^';

    first_line += '\\n';
    result += first_line;

    for (int i = errors_count_in_line - 1; i >= 0; i--)
    {
        // Обрежем первую строку до нужной позиции,
        т.к. на предыдущих итерациях мы уже обработали некоторые ошибки
        string line = first_line.substr(0,
errors_in_line[i]->position);
        // заменим на палочки, так красивее....
        for (int j = 0; j < line.size(); j++)
        {
            if (line[j] == '^')

```

```

        line[j] = '|';
    }

    // текущая ошибка всегда в конце строки,
    // поэтому просто добавим описание ошибки в конец строки
    line += errors_in_line[i]->info;
    line += '\n';

    result += line;
}

result += '\n';

delete error_handler_in_line;
} while (c != EOF);

if (open_state == ByFile)
{
    string path_to_result = change_filename(path_to_file);
    ofstream fout(path_to_result);
    fout << result;
    fout.close();
}
else
{
    cout << result;
}

delete io_helper;
}

string IO_Module::change_filename(string& path)
{
    // попытаемся найти где начинается расширение
    string result;
    auto pos = path.rfind(".");

    if (pos == string::npos) // если у файла нет расширения
    {
        return path + OUTPUT_PREFIX;
    }

    result = path.substr(0, pos) + "_" + OUTPUT_PREFIX +
path.substr(pos);

    return result;
}

```

## 2.4 Тестирование

### 2.4.1 Создание из строки исходного кода

```

const string PATH_INPUT_FILE = "C:\\Users\\user\\pascal.txt";
IO_Module* io = new IO_Module(PATH_INPUT_FILE);

```

### 2.4.2 Создание из файла

```

const string SOURCE_CODE = "program test; var x:integer; begin x:=5;
end.";

```



## **3 Лексический анализатор**

### **3.1 Описание**

Лексический анализатор формирует так называемые токены, объекты, созданные на основе последовательности символов(т.е. определенных слов заложенных в компиляторе).

Было выбрана реализация при которой лексический анализатор получает на вход объект модуля ввода-вывода и строит с помощью него последовательность этих токенов.

### **3.2 Проектирование**

В модуле лексического анализатора были определены следующие методы:

- Получение токена
- Функция проверки программы – говорит о том были ли обнаружены какие-либо ошибки на этапе лексического анализа
- Функция получения всех токенов исходной программы

В сам конструктор класса нашего модуля должны быть переданы указатель на модуль ввода-вывода и указатель на обработчик ошибок.

Главной функцией безусловно является функция получения очередного токена, поэтому опишем ее более подробно.

Пропускаем все пробелы, знаки табуляции, переводы на новую строку, они нам не нужны, запоминаем текущую позицию – это будет позицией нашего будущего, еще несформированного токена.

Сделаем вывод каким будет токен исходя из его первого символа:

- Если символ – число, то токеном будет являться численная константа
- Если символ – буква, то токеном является какое-либо ключевое слово, либо идентификатор, либо это булевская константа(true/false). Для определения является ли слово ключевым, либо булевской константой будем использовать хэш-таблицы для быстрого поиска соответствия. Если соответствия не нашлось – то слово является идентификатором.
- Если символ – ‘, то это должна быть константа типа Char.
- Если символ – “, то это должна быть константа типа String.
- Иначе это какой-либо оператор, например присваивания, сложения и т.п.

Будем продолжать генерировать и сохранять токены до тех пор, пока не дойдем до конца файла.

Токены разделим на три типа – токен-оператор, токен-идентификатор, токен-константа. Каждый токен должен хранить позицию начала данного токена.

Создадим абстрактный класс токен и будем наследоваться от него во всех дочерних классах токен-оператор, токен-идентификатор, токен-константа.

Класс токен-константа будет классом шаблоном и хранить значение константы, то есть использовать template, так как константы в нашем случае могут быть пяти типов – Int, Double, Bool, Char и String.

Класс токен-идентификатор будет хранить имя идентификатора.

Класс токен-оператор будет хранить тип оператора.

### 3.3 Реализация

#### 3.3.1 Лексический анализатор

Ниже представлено описание класса в заголовочном файле LexicalAnalyzer.h

```
class LexicalAnalyzer
{
public:
    IO_Module* io;
    ErrorHandler* error_handler;

    LexicalAnalyzer(IO_Module* _io, ErrorHandler*
_error_handler);

    bool check();
    vector<Token*> get_tokens();

    ~LexicalAnalyzer();
private:
    Token* get_token();

    vector<Token*> tokens;
    int position = 0;
    char c;
};
```

В файле LexicalAnalyzer.cpp представлены определения этих методов:

```
LexicalAnalyzer::LexicalAnalyzer(IO_Module* _io, ErrorHandler*
_error_handler)
{
    error_handler = _error_handler;

    io = _io;
    c = io->get_next_char();
}

Token* LexicalAnalyzer::get_token()
{
    while (c == ' ' || c == '\n' || c == '\r' || c == '\t')
        c = io->get_next_char();

    position = io->get_current_position();

    if (c == EOF)
    {
        return nullptr;
    }
}
```

```

// Парсинг чисел
else if (isdigit(c))
{
    string lexem(1, c);

    c = io->get_next_char();
    while (isdigit(c))
    {
        lexem += c;
        c = io->get_next_char();
    }

    // если число слишком длинное - все плохо

    if (c == '.') // типа real?
    {
        lexem += c;
        c = io->get_next_char();
        while (isdigit(c))
        {
            lexem += c;
            c = io->get_next_char();
        }

        return new ConstToken<double>(ttConst,
stod(lexem), dtReal, position);
    }
    else
    {
        return new ConstToken<int>(ttConst,
stoi(lexem), dtInt, position);
    }
}
// Парсинг идентификаторов/операторов
else if (isalpha(c))
{
    string lexem(1, c);
    c = io->get_next_char();

    while (isdigit(c) || isalpha(c))
    {
        lexem += c;
        c = io->get_next_char();
    }

    if (OperatorKeyWords.find(lexem) ==
OperatorKeyWords.end())
    {
        return new IdentificatorToken(ttIdentificator,
lexem, position);
    }
    else
    {
        OperatorType ot = OperatorKeyWords.at(lexem);
        if (ot == otTrue || ot == otFalse)
        {
            return new ConstToken<bool>(ttConst, ot,
dtBool, position);

```

```

        }
        else
        {
            return new OperatorToken(ttOperator, ot,
position);
        }
    }
}
// Парсинг символов
else if (c == '\\')
{
    char lexem = io->get_next_char();
    c = io->get_next_char();
    if (c != '\\') // ошибка
    {
        string error_text = "Ожидался символ '";
        error_handler->add_error(error_text, io-
>get_current_position());
    }
    else
    {
        c = io->get_next_char();
        return new ConstToken<char>(ttConst, lexem,
dtChar, position);
    }
}
// Парсинг строк
else if (c == '"')
{
    string lexem = "";
    c = io->get_next_char();
    // если закрытия строки не будет - ошибку как-то
    while (c != '"')
    {
        lexem += c;
        c = io->get_next_char();
        if (c == '\\n' || c == EOF)
            break;
    }

    c = io->get_next_char();
    return new ConstToken<string>(ttConst, lexem,
dtString, position);
}
// Парсинг небуквенных операторов
else
{
    string lexem(1, c);
    OperatorType ot = otError;
    if (OperatorSymbols.find(lexem) !=
OperatorSymbols.end())
        ot = OperatorSymbols.at(lexem);

    /*
    .. | := | >= | <= | <>
    */
    switch (ot)
    {

```



```

        case otDot:
            c = io->get_next_char();
            if (c == '.')
            {
                ot = otDots;
                c = io->get_next_char();
            }
            break;
        case otColon:
            c = io->get_next_char();
            if (c == '=')
            {
                ot = otAssign;
                c = io->get_next_char();
            }
            break;
        case otLess:
            c = io->get_next_char();
            if (c == '=')
            {
                ot = otLessEqual;
                c = io->get_next_char();
            }
            else if (c == '>')
            {
                ot = otLessGreater;
                c = io->get_next_char();
            }
            break;
        case otGreater:
            c = io->get_next_char();
            if (c == '=')
            {
                ot = otGreaterEqual;
                c = io->get_next_char();
            }
            break;
        default:
            c = io->get_next_char();
    }

    if (ot == otError)
        return new Token(ttUndefined, position);

    return new OperatorToken(ttOperator, ot, position);
}

}

bool LexicalAnalyzer::check()
{
    int errors_count = 0;
    Token* new_token = get_token();
    do
    {
        tokens.push_back(new_token);
        new_token = get_token();
    } while (new_token != nullptr);
}

```

```

        tokens.push_back(new Token(ttUndefined, io-
>get_current_position()));

        return error_handler->get_errors_count() == errors_count;
    }

vector<Token*> LexicalAnalyzer::get_tokens()
{
    return tokens;
}

LexicalAnalyzer::~LexicalAnalyzer()
{
    delete io;
}

```

### 3.3.2 Токен

```

enum TokenType {
    ttIdentificator,
    ttOperator,
    ttConst,
    ttUndefined
};

class Token {
public:
    TokenType token_type;
    int position;
    Token(TokenType token_type, int position)
    {
        this->token_type = token_type;
        this->position = position;
    }
    virtual ~Token() = default;
};

```

### 3.3.3 Токен-константа

```

template<typename T>
class ConstToken : public Token {
public:
    T value;
    DataType data_type;
    ConstToken(TokenType token_type, T value, DataType data_type,
int position) : Token(token_type, position)
    {
        this->value = value;
        this->data_type = data_type;
    }
};

```

### 3.3.4 Токен-идентификатор

```

class IdentificatorToken : public Token {

```

```

public:
    string name;
    IdentificatorToken(TokenType token_type, string name, int
position) : Token(token_type, position)
    {
        this->name = name;
    }
};

```

### 3.3.5 Токен-оператор

```

class OperatorToken : public Token {
public:
    OperatorType operator_type;
    OperatorToken(TokenType token_type, OperatorType
operator_type, int position) : Token(token_type, position)
    {
        this->operator_type = operator_type;
    }
};

```

### 3.3.6 Хэш-таблицы и перечислимый тип для операторов

```

enum OperatorType {
    otError,
    otPlus, // +
    otMinus, // -
    otSlash, // /
    otStar, // *
    otDot, // .
    otDots, // ..
    otEqual, // =
    otLess, // <
    otGreater, // >
    otAssign, // :=
    otComma, // ,

    ...

    otInteger, // integer
    otBool, // bool
    otReal, // real
    otString, // string
    otChar, // char

    otTrue, // true
    otFalse // false
};

const map<string, OperatorType> OperatorKeyWords = {
    {"if", otIf},
    {"do", otDo},
    {"of", otOf},
    {"or", otOr},
    {"in", otIn},
    {"to", otTo},
    ...
    {"integer", otInteger},
    {"bool", otBool},
    {"real", otReal},
    {"string", otString},
    {"char", otChar},
};

```

```

        {"true", otTrue},
        {"false", otFalse}
    };

    const map<string, OperatorType> OperatorSymbols = {
        {"+", otPlus},
        {"-", otMinus},
        {"/", otSlash},
        {"*", otStar},
        {".", otDot},
        {"..", otDots},
        {"=", otEqual},
        {"<", otLess},
        {">", otGreater},
        {":=", otAssign},
        {"", otComma},
        {">=", otGreaterEqual},
        {"<=", otLessEqual},
        {";", otSemiColon},
        {":", otColon},
        {"(", otLeftParenthesis},
        {")", otRightParenthesis},
        {"[", otLeftBracket},
        {"]", otRightBracket},
        {"<>", otLessGreater}
    };

    const map<OperatorType, string> KeyWordByOperator = {
        {otIf, "if"},
        {otDo, "do"},
        {otOf, "of"},
        {otOr, "or"},
        {otIn, "in"},
        {otTo, "to"},
        ...
        {otLeftBracket, "["},
        {otRightBracket, "]"},
        {otLessGreater, "<>"},
        {otTrue, "true"},
        {otFalse, "false"}
    };

```

### 3.4 Тестирование

Код программы:

```

program mfdсain;
var x:integer; y:char;
begin
    while not (x > -5) do
    begin
        x := -x * 5;
        x := y + 1;
    end;
end

```

Сгенерированные токены:

0x000002b995680390 {operator_type=otProgram (53) }	Token * {OperatorToken}
0x000002b9956992b0 {name="mfd sain" }	Token * {IdentificatorToken}
0x000002b9956808f0 {operator_type=otSemiColon (14) }	Token * {OperatorToken}
0x000002b995680950 {operator_type=otVar (28) }	Token * {OperatorToken}
0x000002b995699330 {name="x" }	Token * {IdentificatorToken}
0x000002b9956806b0 {operator_type=otColon (15) }	Token * {OperatorToken}
0x000002b995680870 {operator_type=otInteger (56) }	Token * {OperatorToken}
0x000002b9956803f0 {operator_type=otSemiColon (14) }	Token * {OperatorToken}
0x000002b995698ff0 {name="y" }	Token * {IdentificatorToken}
0x000002b995680970 {operator_type=otColon (15) }	Token * {OperatorToken}
0x000002b995680710 {operator_type=otChar (60) }	Token * {OperatorToken}
0x000002b9956804f0 {operator_type=otSemiColon (14) }	Token * {OperatorToken}
0x000002b995680790 {operator_type=otBegin (43) }	Token * {OperatorToken}
0x000002b995680410 {operator_type=otWhile (44) }	Token * {OperatorToken}
0x000002b9956809b0 {operator_type=otNot (31) }	Token * {OperatorToken}
0x000002b995680450 {operator_type=otLeftParenthesis (16) }	Token * {OperatorToken}
0x000002b995699030 {name="x" }	Token * {IdentificatorToken}
0x000002b995680a50 {operator_type=otGreater (9) }	Token * {OperatorToken}
0x000002b995680550 {operator_type=otMinus (2) }	Token * {OperatorToken}
0x000002b995680890 {value= 5 data_type= dtInt (0) }	Token * {ConstToken<int> }
0x000002b995680670 {operator_type=otRightParenthesis (17) }	Token * {OperatorToken}
0x000002b995680570 {operator_type=otDo (22) }	Token * {OperatorToken}
0x000002b995680730 {operator_type=otBegin (43) }	Token * {OperatorToken}
0x000002b9956990f0 {name="x" }	Token * {IdentificatorToken}
0x000002b9956805b0 {operator_type=otAssign (10) }	Token * {OperatorToken}
0x000002b995680a70 {operator_type=otMinus (2) }	Token * {OperatorToken}
0x000002b995699070 {name="x" }	Token * {IdentificatorToken}
0x000002b995680750 {operator_type=otStar (4) }	Token * {OperatorToken}
0x000002b9956809d0 {value= 5 data_type= dtInt (0) }	Token * {ConstToken<int> }
0x000002b9956807f0 {operator_type=otSemiColon (14) }	Token * {OperatorToken}
0x000002b9956991b0 {name="x" }	Token * {IdentificatorToken}
0x000002b995680770 {operator_type=otAssign (10) }	Token * {OperatorToken}
0x000002b995699130 {name="y" }	Token * {IdentificatorToken}
0x000002b995680350 {operator_type=otPlus (1) }	Token * {OperatorToken}
0x000002b995680370 {value= 1 data_type= dtInt (0) }	Token * {ConstToken<int> }
0x000002b9956803b0 {operator_type=otSemiColon (14) }	Token * {OperatorToken}
0x000002b9956805f0 {operator_type=otEnd (27) }	Token * {OperatorToken}
0x000002b99569ccc0 {operator_type=otSemiColon (14) }	Token * {OperatorToken}
0x000002b99569ca40 {operator_type=otEnd (27) }	Token * {OperatorToken}

## 4 Синтаксический анализатор

### 4.1 Описание

Синтаксический анализатор проверяет, удовлетворяет ли программа формальным правилам.

Для задания синтаксиса широко применяются формальные правила, записанные в формах Бэкуса-Наура(БНФ), а также синтаксические диаграммы.

### 4.2 Проектирование

Для начала мы должны определиться какие формальные правила мы будем реализовывать, то есть нужно выписать все формы Бэкуса-Наура которые нам понадобятся:

```
<программа>::=program <имя>;<блок>.
<блок>::=<раздел переменных><раздел операторов>
<описание однотипных переменных>::=<имя>{,<имя>}:<тип>
<тип>::=integer|real|string|char
<раздел переменных>::= var <описание однотипных переменных>;{<описание однотипных переменных>;} | <пусто>
<раздел операторов>::= <составной оператор>
<оператор>::=<простой оператор>|<сложный оператор>
<простой оператор>::=<переменная>:=<выражение>
<переменная>::=<имя>
<выражение>::=<простое выражение>|<простое выражение><операция отношения><простое выражение>
<операция отношения>::= =|<|<=>|=|>
<простое выражение>::=<слагаемое>{<аддитивная операция><слагаемое>}
<аддитивная операция>::= +|-|or
<слагаемое>::=<множитель>{<мультипликативная операция><множитель>}
<мультипликативная операция>::= *|/|div|mod|and
<множитель>::=[<знак>]<переменная>[<знак>]<константа>[<знак>](<выражение>)|not <множитель>
<знак>::= +|-
<сложный оператор>::=<составной оператор>|<выбирающий оператор>|<оператор цикла>
<составной оператор>::= begin <оператор>{;<оператор>} end
<выбирающий оператор>::= if <выражение> then <оператор>|if <выражение> then <оператор> else <оператор>
```

<оператор цикла>::= while <выражение> do <оператор>

Для каждого описанного выше правила должны быть описана функция, тело которой есть результат преобразования правой части.

Наш синтаксический анализатор будет строиться по принципу детерминированного рекурсивного нисходящего алгоритма.

В модуле синтаксического анализатора были определены следующие методы:

- Функция пытающаяся “принять” очередной токен, то есть подается тип токена который может/должен идти следующим и эта функция проверяет выполняется ли это условие или нет. Эта же функция будет добавлять различные ошибки в обработчик ошибок.
- Функция переходящая на следующий токен
- Функция проверки программы – говорит о том является ли программа синтаксически правильной
- И все функции реализующие вышеописанные формы Бэкуса-Наура

Также были созданы различные переменные – вектор токенов, указатель на обработчик ошибок, указатель на текущий токен.

## 4.3 Реализация

### 4.3.1 Функция next\_token

```
void SyntaxAnalyzer::next_token()
{
    if (current_token_position == tokens.size())
        return;

    current_token = tokens[current_token_position];
    current_token_position++;
}
```

### 4.3.2 Функция accept

```
bool SyntaxAnalyzer::accept(TokenType token_type, bool
is_necessarily)
{
    bool result = true;
    if (current_token->token_type != token_type)
        result = false;
    if (result)
        next_token();
    else if (is_necessarily) // выводим ошибку
    {
        string error_text = "";
        switch (current_token->token_type)
        {
            case ttIdentifier:
```

```

        error_text = "Ожидалось имя
идентификатора";
        break;
    case ttOperator:
        error_text = "Ожидалось имя оператора";
        break;
    case ttConst:
        error_text = "Ожидалась константа";
        break;
    }

    error_handler->add_error(error_text, current_token-
>position);

    next_token();
}
return result;
}

bool SyntaxAnalyzer::accept(OperatorType operator_type, bool
is_necessarily)
{
    bool result = true;
    if (current_token->token_type != ttOperator)
        result = false;

    if (((OperatorToken*)current_token)->operator_type !=
operator_type)
        result = false;

    if (result)
    {
        next_token();
    }
    else if (is_necessarily) // ошибка вышла
    {
        string error_text = "Ожидался оператор: " +
KeywordByOperator.find(operator_type)->second;

        error_handler->add_error(error_text, current_token-
>position);

        next_token();
    }

    return result;
}

bool SyntaxAnalyzer::accept(vector<OperatorType> operator_types,
bool is_necessarily)
{
    bool result = false;
    if (current_token->token_type == ttOperator)
    {
        OperatorType current_type =
((OperatorToken*)current_token)->operator_type;

        for (OperatorType operator_type : operator_types)

```



```

        {
            if (operator_type == current_type)
            {
                result = true;
                break;
            }
        }
    }

    if (result)
    {
        next_token();
    }
    else if (is_necessarilly)
    {
        string error_text = "Ожидался один из операторов: ";

        for (OperatorType operator_type : operator_types)
            error_text +=
KeywordByOperator.find(operator_type)->second + ", ";

        error_handler->add_error(error_text, current_token-
>position);

        next_token();
    }

    return result;
}

```

### 4.3.3 Функции реализующие БНФ

```

void SyntaxAnalyzer::program() // <программа>::=program <имя>(<имя
файла>{,<имя файла>});<блок>.
{
    if (!accept(otProgram, true)) return;
    if (!accept(ttIdentificator, true)) return;
    if (!accept(otSemiColon, true)) return;

    block();

    if (!accept(otDot, true)) return;
}

void SyntaxAnalyzer::block() // <блок>::=<раздел констант><раздел
типов><раздел переменных><раздел процедур и функций><раздел
операторов>
{
    //constants_section();
    vars_section();
    //functions_section();
    operators_section();
}

// ===== Раздел переменных =====

```

```

bool SyntaxAnalyzer::single_var_definition() // <описание однотоипных
переменных> ::= <имя> { , <имя> } : <тип>
{
    if (!accept(ttIdentificator))
        return false;

    while (accept(otComma))
    {
        if (!accept(ttIdentificator, true)) return false;
    }

    if (!accept(otColon, true)) return false;

    type();

    return true;
}

void SyntaxAnalyzer::type() // <тип> ::= integer | real | string | char
{
    accept({ otInteger, otReal, otString, otChar }, true);
}

void SyntaxAnalyzer::vars_section() // <раздел переменных> ::= var
<описание однотоипных переменных>; { <описание однотоипных переменных>; }
| <пусто>
{
    if (!accept(otVar))
        return;

    single_var_definition();
    if (!accept(otSemiColon, true)) return;

    while (single_var_definition())
    {
        if (!accept(otSemiColon, true)) return;
    }
}

// ===== Раздел операторов =====
// <раздел операторов> ::= <составной оператор>

void SyntaxAnalyzer::operators_section()
{
    neccessary_compound_operator();
}

// <оператор> ::= <простой оператор> | <сложный оператор>
void SyntaxAnalyzer::operator_()
{
    if (!simple_operator())
        complex_operator();
}

// <простой оператор> ::= <переменная> := <выражение>
bool SyntaxAnalyzer::simple_operator() // *
{
    if (!var())

```

```

        return false;

        if (!accept(otAssign, true)) return false;
        expression();

        return true;
    }

    //<переменная>::=<имя>
    bool SyntaxAnalyzer::var()    // *
    {
        return accept(ttIdentificator);
    }

    //<выражение>::=<простое выражение>|<простое выражение><операция
    отношения><простое выражение>
    void SyntaxAnalyzer::expression()
    {
        simple_expression();
        if (relation_operation())
            simple_expression();
    }

    //<операция отношения>::= =|<>|<|<=>|=|>
    bool SyntaxAnalyzer::relation_operation()    // *
    {
        return accept({ otEqual, otLessGreater, otLessEqual,
            otGreaterEqual, otGreater });
    }

    //<простое выражение>::=<слагаемое>{<аддитивная
    операция><слагаемое>}
    void SyntaxAnalyzer::simple_expression()
    {
        term();
        while (additive_operation())
            term();
    }

    //<аддитивная операция>::= +|-|or
    bool SyntaxAnalyzer::additive_operation()    // *
    {
        return accept({ otPlus, otMinus, otOr });
    }

    //<слагаемое>::=<множитель>{<мультипликативная операция><множитель>}
    void SyntaxAnalyzer::term()
    {
        factor();
        while (multiplicative_operation())
            factor();
    }

    //<мультипликативная операция>::= *|/|div|mod|and
    bool SyntaxAnalyzer::multiplicative_operation()    // *
    {
        return accept({ otStar, otSlash, otDiv, otMod, otAnd });
    }

```

```

//<множитель>::=[<знак>]<переменная>| [<знак>]<константа>| [<знак>] (<в
ыражение>)|not <множитель>
void SyntaxAnalyzer::factor()
{
    if (sign())
    {
        //...
    }

    if (var())
    {
        //...
    }
    else if (accept(ttConst))
    {
        //...
    }
    else if (accept(otLeftParenthesis))
    {
        expression();
        accept(otRightParenthesis, true);
    }
    else if (accept(otNot))
    {
        factor();
    }
    else
    {
        string error_text = "Ожидалась
переменная/константа/выражение";

        error_handler->add_error(error_text, current_token-
>position);
    }
}

//<знак>::= +|-
bool SyntaxAnalyzer::sign() // *
{
    return accept({ otPlus, otMinus });
}

//<сложный оператор>::=<составной оператор>|<выбирающий
оператор>|<оператор цикла>
void SyntaxAnalyzer::complex_operator()
{
    if (compound_operator())
    {
        //...
    }
    else if (if_operator())
    {
        //...
    }
    else if (while_operator())
    {

```

```

        //...
    }
}

//<составной оператор>::= begin <оператор>{;<оператор>} end
bool SyntaxAnalyzer::compound_operator() // *
{
    if (!accept(otBegin))
        return false;

    operator_();

    while (accept(otSemiColon))
    {
        operator_();
    }

    if (!accept(otEnd, true)) return false;

    return true;
}

//<обязательный составной оператор>::= begin <оператор>{;<оператор>}
end
void SyntaxAnalyzer::neccessary_compound_operator() // *
{
    if (!accept(otBegin, true)) return;

    operator_();

    while (accept(otSemiColon))
    {
        operator_();
    }

    if (!accept(otEnd, true)) return;
}

//<выбирающий оператор>::= if <выражение> then <оператор>|if
<выражение> then <оператор> else <оператор>
bool SyntaxAnalyzer::if_operator() // *
{
    if (!accept(otIf))
        return false;

    expression();
    accept(otThen);
    operator_();

    if (accept(otElse))
        operator_();

    return true;
}

//<оператор цикла>::= while <выражение> do <оператор>
bool SyntaxAnalyzer::while_operator() // *
{

```

```

        if (!accept(otWhile))
            return false;

        expression();
        if (!accept(otDo, true)) return false;
        operator_();

        return true;
}

```

#### 4.3.4 Полное описание класса

Ниже представлено описание класса в заголовочном файле SyntaxAnalyzer.h

```

class SyntaxAnalyzer
{
public:
    SyntaxAnalyzer(vector<Token*> _tokens, ErrorHandler*
_error_handler);

    bool check();

    ~SyntaxAnalyzer();
private:
    vector<Token*> tokens;
    ErrorHandler* error_handler;

    int current_token_position;
    Token* current_token;

    void next_token();

    bool accept(TokenType token_type, bool is_necessarily =
false);
    bool accept(OperatorType operator_type, bool is_necessarily =
false);
    bool accept(vector<OperatorType> operator_types, bool
is_necessarily = false);

    void program();

    void block();

    void vars_section();
    void operators_section();

    bool single_var_definition();

    void type();

    void operator_();
    bool simple_operator();
    bool var();
    void expression();
    bool relation_operation();
    void simple_expression();
    bool additive_operation();
    void term();

```

```

    bool multiplicative_operation();
    void factor();
    bool sign();
    void complex_operator();
    void necessary_compound_operator();
    bool compound_operator();
    bool if_operator();
    bool while_operator();
};

```

Определим все методы в файле-источнике SyntaxAnalyzer.cpp

```

SyntaxAnalyzer::SyntaxAnalyzer(vector<Token*> _tokens, ErrorHandler*
_error_handler)
{
    error_handler = _error_handler;
    tokens = _tokens;
    current_token_position = 0;
    next_token();
}

SyntaxAnalyzer::~SyntaxAnalyzer()
{
    delete current_token;
}

bool SyntaxAnalyzer::check()
{
    int erros_count = error_handler->get_errors_count();

    program();

    return error_handler->get_errors_count() == erros_count;
}

void SyntaxAnalyzer::next_token()
{
    if (current_token_position == tokens.size())
        return;

    current_token = tokens[current_token_position];
    current_token_position++;
}

bool SyntaxAnalyzer::accept(TokenType token_type, bool
is_necessarily)
{
    bool result = true;
    if (current_token->token_type != token_type)
        result = false;
    if (result)
        next_token();
    else if (is_necessarily) // выводим ошибку
    {
        string error_text = "";
        switch (current_token->token_type)
        {
            case ttIdentificator:

```

```

        error_text = "Ожидалось имя
идентификатора";
        break;
    case ttOperator:
        error_text = "Ожидалось имя оператора";
        break;
    case ttConst:
        error_text = "Ожидалась константа";
        break;
    }

    error_handler->add_error(error_text, current_token-
>position);

    next_token();
}
return result;
}

bool SyntaxAnalyzer::accept(OperatorType operator_type, bool
is_necessarily)
{
    bool result = true;
    if (current_token->token_type != ttOperator)
        result = false;

    if (((OperatorToken*)current_token)->operator_type !=
operator_type)
        result = false;

    if (result)
    {
        next_token();
    }
    else if (is_necessarily) // ошибка вышла
    {
        string error_text = "Ожидался оператор: " +
KeywordByOperator.find(operator_type)->second;

        error_handler->add_error(error_text, current_token-
>position);

        next_token();
    }

    return result;
}

bool SyntaxAnalyzer::accept(vector<OperatorType> operator_types,
bool is_necessarily)
{
    bool result = false;
    if (current_token->token_type == ttOperator)
    {
        OperatorType current_type =
((OperatorToken*)current_token)->operator_type;

        for (OperatorType operator_type : operator_types)

```



```

        {
            if (operator_type == current_type)
            {
                result = true;
                break;
            }
        }

    }

    if (result)
    {
        next_token();
    }
    else if (is_necessarily)
    {
        string error_text = "Ожидался один из операторов: ";

        for (OperatorType operator_type : operator_types)
            error_text +=
KeywordByOperator.find(operator_type)->second + ", ";

        error_handler->add_error(error_text, current_token-
>position);

        next_token();
    }

    return result;
}

void SyntaxAnalyzer::program() // <программа>::=program <имя>(<имя
файла>{,<имя файла>});<блок>.
{
    if (!accept(otProgram, true)) return;
    if (!accept(ttIdentificator, true)) return;
    if (!accept(otSemiColon, true)) return;

    block();

    if (!accept(otDot, true)) return;
}

void SyntaxAnalyzer::block() // <блок>::=<раздел констант><раздел
типов><раздел переменных><раздел процедур и функций><раздел
операторов>
{
    //constants section();
    vars_section();
    //functions_section();
    operators_section();
}

// ===== Раздел переменных =====
bool SyntaxAnalyzer::single_var_definition() // <описание однотипных
переменных>::=<имя>{,<имя>}:<тип>
{

```

```

        if (!accept(ttIdentificator))
            return false;

        while (accept(otComma))
        {
            if (!accept(ttIdentificator, true)) return false;
        }

        if (!accept(otColon, true)) return false;

        type();

        return true;
    }

void SyntaxAnalyzer::type() // <тип>::=integer|real|string|char
{
    accept({ otInteger, otReal, otString, otChar }, true);
}

void SyntaxAnalyzer::vars_section() // <раздел переменных>::= var
<описание однотипных переменных>;{<описание однотипных переменных>;}
| <пусто>
{
    if (!accept(otVar))
        return;

    single_var_definition();
    if (!accept(otSemiColon, true)) return;

    while (single_var_definition())
    {
        if (!accept(otSemiColon, true)) return;
    }
}

// ===== Раздел операторов =====
// <раздел операторов>::= <составной оператор>

void SyntaxAnalyzer::operators_section()
{
    neccessary_compound_operator();
}

//<оператор>::=<простой оператор>|<сложный оператор>
void SyntaxAnalyzer::operator_()
{
    if (!simple_operator())
        complex_operator();
}

//<простой оператор>::=<переменная>:=<выражение>
bool SyntaxAnalyzer::simple_operator() // *
{
    if (!var())
        return false;

    if (!accept(otAssign, true)) return false;

```

```

        expression();

        return true;
    }

    //<переменная>::=<имя>
    bool SyntaxAnalyzer::var()    // *
    {
        return accept(ttIdentificator);
    }

    //<выражение>::=<простое выражение>|<простое выражение><операция
    отношения><простое выражение>
    void SyntaxAnalyzer::expression()
    {
        simple_expression();
        if (relation_operation())
            simple_expression();
    }

    //<операция отношения>::= =|<|<|<|=|>
    bool SyntaxAnalyzer::relation_operation()    // *
    {
        return accept({ otEqual, otLessGreater, otLessEqual,
otGreaterEqual, otGreater });
    }

    //<простое выражение>::=<слагаемое>{<аддитивная
    операция><слагаемое>}
    void SyntaxAnalyzer::simple_expression()
    {
        term();
        while (additive_operation())
            term();
    }

    //<аддитивная операция>::= +|-|or
    bool SyntaxAnalyzer::additive_operation()    // *
    {
        return accept({ otPlus, otMinus, otOr });
    }

    //<слагаемое>::=<множитель>{<мультипликативная операция><множитель>}
    void SyntaxAnalyzer::term()
    {
        factor();
        while (multiplicative_operation())
            factor();
    }

    //<мультипликативная операция>::= *||div|mod|and
    bool SyntaxAnalyzer::multiplicative_operation()    // *
    {
        return accept({ otStar, otSlash, otDiv, otMod, otAnd });
    }

    //<множитель>::=[<знак>]<переменная>| [<знак>]<константа>| [<знак>] (<в
    ыражение>)|not <множитель>

```

```

void SyntaxAnalyzer::factor()
{
    if (sign())
    {
        //...
    }

    if (var())
    {
        //...
    }
    else if (accept(ttConst))
    {
        //...
    }
    else if (accept(otLeftParenthesis))
    {
        expression();
        accept(otRightParenthesis, true);
    }
    else if (accept(otNot))
    {
        factor();
    }
    else
    {
        string error_text = "Ожидалась
переменная/константа/выражение";

        error_handler->add_error(error_text, current_token-
>position);
    }

}

//<знак>::= +|-
bool SyntaxAnalyzer::sign() // *
{
    return accept({ otPlus, otMinus });
}

//<сложный оператор>::=<составной оператор>|<выбирающий
оператор>|<оператор цикла>
void SyntaxAnalyzer::complex_operator()
{
    if (compound_operator())
    {
        //...
    }
    else if (if_operator())
    {
        //...
    }
    else if (while_operator())
    {
        //...
    }
}

```

```

//<составной оператор>::= begin <оператор>{;<оператор>} end
bool SyntaxAnalyzer::compound_operator() // *
{
    if (!accept(otBegin))
        return false;

    operator_();

    while (accept(otSemiColon))
    {
        operator_();
    }

    if (!accept(otEnd, true)) return false;

    return true;
}

//<обязательный составной оператор>::= begin <оператор>{;<оператор>}
end
void SyntaxAnalyzer::neccessary_compound_operator() // *
{
    if (!accept(otBegin, true)) return;

    operator_();

    while (accept(otSemiColon))
    {
        operator_();
    }

    if (!accept(otEnd, true)) return;
}

//<выбирающий оператор>::= if <выражение> then <оператор>|if
<выражение> then <оператор> else <оператор>
bool SyntaxAnalyzer::if_operator() // *
{
    if (!accept(otIf))
        return false;

    expression();
    accept(otThen);
    operator_();

    if (accept(otElse))
        operator_();

    return true;
}

//<оператор цикла>::= while <выражение> do <оператор>
bool SyntaxAnalyzer::while_operator() // *
{
    if (!accept(otWhile))
        return false;

```

```

        expression();
        if (!accept(otDo, true)) return false;
        operator_();

        return true;
}

```

#### 4.4 Тестирование

Код программы:

```

program mfdсain;
var x:integer; y:char;
begin
    while not (x > -5) do
    begin
        x := -x * 5;
        x := x + 1;

    end;
end

```

Вывод:

```

program mfdсain;
var x:integer; y:char;
begin
    while not (x > -5) do
    begin
        x := -x * 5;
        x := x + 1;

    end;
end

```

^

Ожидался оператор: .

Код программы:

```

program mfdсain;
var x:integer; y:char;
begin
    not (x > -5) do
    begin
        x := -x * 5;
        x := x + 1;

    end;
end

```

Вывод:

```

program mfdсain;
var x:integer; y:char;
begin
    not (x > -5) do
    ^      ^
    |      Ожидался оператор: .
    Ожидался оператор: end

begin

```

```
        x := -x * 5;  
        x := x + 1;  
    end;  
end
```

## 5 Семантический анализатор

### 5.1 Описание

Семантический анализатор получает на вход синтаксически проверенную программу и проверяет не нарушаются ли неформальные правила описания языка.

Реализованные проверки неформальных правил:

- Переменная может быть только одного типа
- Использование необъявленных переменных
- Использование переменных с неприводимыми типами в одном выражении
- Попытка присвоить значение выражения другого типа в отличие от переменной
- Использование неприводимого выражения к типу Boolean в выбирающем операторе и операторе цикла

### 5.2 Проектирование

Для типов были спроектированы и реализованы специальные классы, для каждого типа свой класс. В каждом таком классе была определена функция – приводим ли этот тип к другому типу?

Для этого был написан абстрактный класс Type и дочерние классы – IntegerType, BoolType, RealType, CharType, StringType

В модуле семантического анализатора были созданы следующие хэш-таблицы:

- Переменные {название, указатель на экземпляр типа}
- Доступные типы {тип данных(перечислимый тип из модуля с токенами), указатель на экземпляр типа}

При вызове конструктора добавим все пять типов в хэш-таблицу доступных типов.

При попытке добавления новой переменной сначала проверим есть ли переменная с таким именем в таблице, если есть – ошибка, иначе добавим эту пару <переменная, тип> в мэпу.

Также одной из важнейших функций является функция приведения типов – она получает два типа и пытается их привести к какому-то, например: [Integer, Real] -> можно привести к типу Real, этот тип и вернет функция.

### 5.3 Реализация

#### 5.3.1 Функция добавления переменной

```
void SemanticAnalyzer::add_var(VarName name, Type* dt)
{
    if (variables.find(name) == variables.end())
        variables[name] = dt;
```



```

else
{
    string error_text = "Переменная с именем `" + name +
" уже была объявлена";
    error_handler->add_error(error_text, current_token-
>position);
}
}

```

### 5.3.2 Функция приведения типов

```

Type* SemanticAnalyzer::derive(Type* left, Type* right,
OperatorType last_operation, int position_for_error)
{
    Type* result = new Type();
    if (left->can_cast_to(right))
        result = right;
    if (right->can_cast_to(left))
        result = left;

    bool is_string = result-
>can_cast_to(available_types[dtString]);

    /*
    =|<>|<|<=|>=|>
    accept({ otEqual, otLessGreater, otLessEqual, otGreaterEqual,
otGreater });
    +|-|or
    accept({ otPlus, otMinus, otOr });
    *|/|div|mod|and
    accept({ otStar, otSlash, otDiv, otMod, otAnd });
    */

    string error_text;

    if (is_string)
    {
        // только +, =, <>
        if (last_operation != otPlus && last_operation !=
otEqual && last_operation != otLessGreater)
        {
            error_text = "Данную операцию нельзя применить
к этим операндам";
            error_handler->add_error(error_text,
position_for_error);
        }
    }
    else
    {
        if (left->can_cast_to(available_types[dtString]) ||
right->can_cast_to(available_types[dtString])) // хотя бы один
операнд - строковый
        {
            error_text = "Данную операцию нельзя применить
к этим операндам";
            error_handler->add_error(error_text,
position_for_error);
        }
    }
}

```

```

        if (last_operation == otEqual || last_operation ==
otLessGreater || last_operation == otLessEqual ||
            last_operation == otGreaterEqual || last_operation ==
otGreater || last_operation == otLess ||
            last_operation == otOr || last_operation == otAnd)
            return available_types[dtBool];

    return result;
}

```

### 5.3.3 Полное описание класса модуля семантического анализатора

Описание в заголовочном файле SemanticAnalyzer.h

```

class SemanticAnalyzer
{
public:
    SemanticAnalyzer(vector<Token*> _tokens, ErrorHandler*
_error_handler);
    ~SemanticAnalyzer();

    bool check();

private:
    int current_token_position;
    Token* current_token;

    vector<Token*> tokens;
    ErrorHandler* error_handler;

    map<VarName, Type*> variables;
    map<DataType, Type*> available_types;
    map<OperatorType, int> get_last_position_of_operator;
    OperatorType lastOp;

    void next_token();

    Type* derive(Type* left, Type* right, OperatorType
last_operation, int position_for_error);

    void add_var(VarName name, Type* dt);
    VarName get_var_name_from_token(Token* token);
    Type* get_type_from_const_token(Token* token);

    bool accept(TokenType token_type);
    bool accept(OperatorType operator_type);
    bool accept(vector<OperatorType> operator_types);

    void program();

    void block();

    void vars_section();
    void operators_section();

    bool single_var_definition();

```

```

Type* type();

void operator_();
bool simple_operator();
Type* expression();
bool relation_operation();
Type* simple_expression();
bool additive_operation();
Type* term();
bool multiplicative_operation();
Type* factor();
bool sign();
void complex_operator();
void neccessary_compound_operator();
bool compound_operator();
bool if_operator();
bool while_operator();
};

```

Реализация описанных методов в файле-источнике SemanticAnalyzer.cpp:

```

SemanticAnalyzer::SemanticAnalyzer(vector<Token*> _tokens,
ErrorHandler* _error_handler)
{
    tokens = _tokens;
    error_handler = _error_handler;

    current_token_position = 0;
    next_token();

    available_types[dtInt] = new IntegerType();
    available_types[dtReal] = new RealType();
    available_types[dtString] = new StringType();
    available_types[dtBool] = new BoolType();
    available_types[dtChar] = new CharType();
}

SemanticAnalyzer::~SemanticAnalyzer()
{
    delete error_handler;
}

bool SemanticAnalyzer::check()
{
    int errors_count = error_handler->get_errors_count();
    program();
    return error_handler->get_errors_count() == errors_count;
}

void SemanticAnalyzer::next_token()
{
    if (current_token_position == tokens.size())
        return;

    current_token = tokens[current_token_position];
    current_token_position++;

    if (current_token->token_type == ttOperator)

```

```

        get_last_position_of_operator[((OperatorToken*)current_token)
->operator_type] = current_token->position;
    }

Type* SemanticAnalyzer::derive(Type* left, Type* right, OperatorType
last_operation, int position_for_error)
{
    Type* result = new Type();
    if (left->can_cast_to(right))
        result = right;
    if (right->can_cast_to(left))
        result = left;

    bool is_string = result-
>can_cast_to(available_types[dtString]);

    /*
    |=<>|<|<=|>=|>
    accept({ otEqual, otLessGreater, otLessEqual, otGreaterEqual,
otGreater });
    +|-|or
    accept({ otPlus, otMinus, otOr });
    */div|mod|and
    accept({ otStar, otSlash, otDiv, otMod, otAnd });
    */

    string error_text;

    if (is_string)
    {
        // ТОЛЬКО +, =, <>
        if (last_operation != otPlus && last_operation !=
otEqual && last_operation != otLessGreater)
        {
            error_text = "Данную операцию нельзя применить
к этим операндам";
            error_handler->add_error(error_text,
position_for_error);
        }
    }
    else
    {
        if (left->can_cast_to(available_types[dtString]) ||
right->can_cast_to(available_types[dtString])) // хотя бы один
операнд - строковый
        {
            error_text = "Данную операцию нельзя применить
к этим операндам";
            error_handler->add_error(error_text,
position_for_error);
        }
    }

    if (last_operation == otEqual || last_operation ==
otLessGreater || last_operation == otLessEqual ||
        last_operation == otGreaterEqual || last_operation ==
otGreater || last_operation == otLess ||

```

```

        last_operation == otOr || last_operation == otAnd)
        return available_types[dtBool];

    return result;
}

void SemanticAnalyzer::add_var(VarName name, Type* dt)
{
    if (variables.find(name) == variables.end())
        variables[name] = dt;
    else
    {
        string error_text = "Переменная с именем `" + name +
" уже была объявлена";
        error_handler->add_error(error_text, current_token-
>position);
    }
}

VarName SemanticAnalyzer::get_var_name_from_token(Token* token)
{
    if (token->token_type != ttIdentificator)
        return "";

    return ((IdentificatorToken*)token)->name;
}

Type* SemanticAnalyzer::get_type_from_const_token(Token* token)
{
    if (auto ct = dynamic_cast<ConstToken<int>*>(token)) {
        return available_types[ct->data_type];
    }
    else if (auto ct = dynamic_cast<ConstToken<double>*>(token))
    {
        return available_types[ct->data_type];
    }
    else if (auto ct = dynamic_cast<ConstToken<string>*>(token))
    {
        return available_types[ct->data_type];
    }
    else if (auto ct = dynamic_cast<ConstToken<char>*>(token)) {
        return available_types[ct->data_type];
    }
    else if (auto ct = dynamic_cast<ConstToken<bool>*>(token)) {
        return available_types[ct->data_type];
    }

    return new Type();
}

bool SemanticAnalyzer::accept(TokenType token_type)
{
    bool result = true;
    if (current_token->token_type != token_type)
        result = false;

    if (result)
        next_token();
}

```

```

        return result;
    }

bool SemanticAnalyzer::accept(OperatorType operator_type)
{
    bool result = true;
    if (current_token->token_type != ttOperator)
        result = false;

    if (((OperatorToken*)current_token)->operator_type !=
operator_type)
        result = false;

    if (result)
    {
        lastOp = operator_type;
        next_token();
    }

    return result;
}

bool SemanticAnalyzer::accept(vector<OperatorType> operator_types)
{
    bool result = false;
    if (current_token->token_type == ttOperator)
    {
        OperatorType current_type =
((OperatorToken*)current_token)->operator_type;

        for (OperatorType operator_type : operator_types)
        {
            if (operator_type == current_type)
            {
                lastOp = operator_type;
                result = true;
                break;
            }
        }
    }

    if (result)
        next_token();

    return result;
}

void SemanticAnalyzer::program() // <программа> ::= program <имя> (<имя
файла>{, <имя файла>}); <блок>.
{
    accept(otProgram);
    accept(ttIdentificator);
    accept(otSemiColon);

    block();

    accept(otDot);
}

```

```

}

void SemanticAnalyzer::block() // <блок>::=<раздел констант><раздел
типов><раздел переменных><раздел процедур и функций><раздел
операторов>
{
    vars_section();
    operators_section();
}

// ===== Раздел переменных =====
bool SemanticAnalyzer::single_var_definition() // <описание
однотипных переменных>::=<имя>{,<имя>}:<тип>
{
    vector<VarName> variableNames;

    variableNames.push_back(get_var_name_from_token(current_token
));
    if (!accept(ttIdentificator))
        return false;

    while (accept(otComma))
    {
        variableNames.push_back(get_var_name_from_token(current_token
));
        accept(ttIdentificator);
    }

    accept(otColon);

    Type* varType = type();

    for (VarName name : variableNames)
    {
        add_var(name, varType);
    }

    return true;
}

Type* SemanticAnalyzer::type() // <тип>::=integer|real|string|char
{
    if (accept(otInteger))
        return available_types[dtInt];
    if (accept(otReal))
        return available_types[dtReal];
    if (accept(otString))
        return available_types[dtString];
    if (accept(otChar))
        return available_types[dtChar];
}

void SemanticAnalyzer::vars_section() // <раздел переменных>::= var
<описание однотипных переменных>;{<описание однотипных переменных>;}
| <пусто>
{
    accept(otVar);
}

```

```

    single_var_definition();
    accept(otSemiColon);

    while (single_var_definition())
    {
        accept(otSemiColon);
    }
}

// ===== Раздел операторов =====
// <раздел операторов>::= <составной оператор>

void SemanticAnalyzer::operators_section()
{
    neccessary_compound_operator();
}

//<оператор>::=<простой оператор>|<сложный оператор>
void SemanticAnalyzer::operator_()
{
    if (!simple_operator())
        complex_operator();
}

//<простой оператор>::=<переменная>:=<выражение>
bool SemanticAnalyzer::simple_operator() // *
{
    int mem_position = current_token->position;
    VarName name = get_var_name_from_token(current_token);

    if (!accept(ttIdentificator))
        return false;

    accept(otAssign);
    Type* t = expression();

    if (variables.find(name) == variables.end())
    {
        string error_text = "Переменная не была объявлена";
        error_handler->add_error(error_text, mem_position);
    }
    else
    {
        if (!t->can_cast_to(variables[name]))
        {
            // TODO: вывод ошибки
            string error_text = "Вычисленное выражение
имеет другой тип в отличие от переменной";
            error_handler->add_error(error_text,
get_last_position_of_operator[otAssign]);
        }
    }

    return true;
}

```



```

//<выражение>::=<простое выражение>|<простое выражение><операция
отношения><простое выражение>
Type* SemanticAnalyzer::expression()
{
    Type *t1, *t2;
    t1 = simple_expression();
    if (relation_operation())
    {
        OperatorType last_operation = lastOp;
        int position =
get_last_position_of_operator[last_operation];
        t2 = simple_expression();
        t1 = derive(t1, t2, last_operation, position);
    }

    return t1;
}

//<операция отношения>::= =|<|<|<=|>=|>
bool SemanticAnalyzer::relation_operation() // *
{
    return accept({ otEqual, otLessGreater, otLessEqual,
otGreaterEqual, otGreater });
}

//<простое выражение>::=<слагаемое>{<аддитивная
операция><слагаемое>}
Type* SemanticAnalyzer::simple_expression()
{
    Type* t1, * t2;
    t1 = term();
    while (additive_operation())
    {
        OperatorType last_operation = lastOp;
        int position =
get_last_position_of_operator[last_operation];
        t2 = term();
        t1 = derive(t1, t2, last_operation, position);
    }
    return t1;
}

//<аддитивная операция>::= +|-|or
bool SemanticAnalyzer::additive_operation() // *
{
    return accept({ otPlus, otMinus, otOr });
}

//<слагаемое>::=<множитель>{<мультипликативная операция><множитель>}
Type* SemanticAnalyzer::term()
{
    Type* t1, * t2;
    t1 = factor();
    while (multiplicative_operation())
    {
        OperatorType last_operation = lastOp;
        int position =
get_last_position_of_operator[last_operation];

```

```

        t2 = factor();
        t1 = derive(t1, t2, last_operation, position);
    }

    return t1;
}

//<мультипликативная операция>::=*||div|mod|and
bool SemanticAnalyzer::multiplicative_operation() // *
{
    return accept({ otStar, otSlash, otDiv, otMod, otAnd });
}

//<множитель>::=[<знак>]<переменная>| [<знак>]<константа>| [<знак>] (<в
ыражение>)|not <множитель>
Type* SemanticAnalyzer::factor()
{
    // TODO: сделать что-то со знаком...
    if (sign())
    {
        //...
    }

    VarName name = get_var_name_from_token(current_token);
    Type* const_type = get_type_from_const_token(current_token);
    int mem_position = current_token->position;
    if (accept(ttIdentificator))
    {
        if (variables.find(name) == variables.end())
        {
            string error_text = "Переменная не была
объявлена";
            error_handler->add_error(error_text,
mem_position);
            return new Type();
        }
        return variables[name];
    }
    else if (accept(ttConst))
    {
        return const_type;
    }
    else if (accept(otLeftParenthesis))
    {
        Type* t = expression();
        accept(otRightParenthesis);

        return t;
    }
    else if (accept(otNot))
    {
        Type* t = factor();

        if (!t->can_cast_to(available_types[dtBool]))
        {
            string error_text = "Выражение должно иметь тип
Bool";

```

```

        error_handler->add_error(error_text,
get_last_position_of_operator[otNot]);
        // TODO: ошибка
    }

    return available_types[dtBool];
}

}

//<знак>::= +|-
bool SemanticAnalyzer::sign()    // *
{
    return accept({ otPlus, otMinus });
}

//<сложный оператор>::=<составной оператор>|<выбирающий
оператор>|<оператор цикла>
void SemanticAnalyzer::complex_operator()
{
    if (compound_operator())
    {
        //...
    }
    else if (if_operator())
    {
        //...
    }
    else if (while_operator())
    {
        //...
    }
}

//<составной оператор>::= begin <оператор>{;<оператор>} end
bool SemanticAnalyzer::compound_operator()    // *
{
    if (!accept(otBegin))
        return false;

    operator_();

    while (accept(otSemiColon))
    {
        operator_();
    }

    accept(otEnd);

    return true;
}

//<обязательный составной оператор>::= begin <оператор>{;<оператор>}
end
void SemanticAnalyzer::neccessary_compound_operator()    // *
{
    accept(otBegin);

```

```

        operator_();

        while (accept(otSemiColon))
        {
            operator_();
        }

        accept(otEnd);
    }

    //<выбирающий оператор>::= if <выражение> then <оператор>|if
    <выражение> then <оператор> else <оператор>
    bool SemanticAnalyzer::if_operator() // *
    {
        if (!accept(otIf))
            return false;

        Type* t = expression();

        if (!t->can_cast_to(available_types[dtBool]))
        {
            string error_text = "Выражение должно иметь тип Bool";
            error_handler->add_error(error_text, current_token-
>position);
            // TODO: ошибка
        }

        accept(otThen);
        operator_();

        if (accept(otElse))
            operator_();

        return true;
    }

    //<оператор цикла>::= while <выражение> do <оператор>
    bool SemanticAnalyzer::while_operator() // *
    {
        if (!accept(otWhile))
            return false;

        Type* t = expression();
        if (!t->can_cast_to(available_types[dtBool]))
        {
            string error_text = "Выражение должно иметь тип Bool";
            error_handler->add_error(error_text, current_token-
>position);
            // TODO: ошибка
        }

        accept(otDo);
        operator_();

        return true;
    }

```

### 5.3.4 Описание класса типов

```

enum EType
{
    et_intger,
    et_bool,
    et_real,
    et_char,
    et_string,
    et_undefined
};

class Type
{
public:
    EType type = et_undefined;
    Type() {};
    virtual bool can_cast_to(Type* another_type) { return false;
};
};

class IntegerType : public Type
{
public:
    IntegerType() { type = et_intger; }
    bool can_cast_to(Type* another_type)
    {
        return another_type->type == et_real ||
            another_type->type == type;
    }
};

class BoolType : public Type
{
public:
    BoolType() { type = et_bool; }
    bool can_cast_to(Type* another_type)
    {
        return another_type->type == et_real ||
            another_type->type == et_intger ||
            another_type->type == et_real ||
            another_type->type == type;
    }
};

class RealType : public Type
{
public:
    RealType() { type = et_real; }
    bool can_cast_to(Type* another_type)
    {
        return another_type->type == type;
    }
};

class CharType : public Type
{
public:
    CharType() { type = et_char; }
    bool can_cast_to(Type* another_type)

```



операндам | Данную операцию нельзя применить к этим  
 Вычисленное выражение имеет другой тип в  
 отличие от переменной

```
f := 5
^
```

Переменная не была объявлена

```
end;
x := -x * 5;
x := y + 1;
^ ^
```

операндам | Данную операцию нельзя применить к этим  
 Вычисленное выражение имеет другой тип в отличие от  
 переменной

```
f := 5
^
```

Переменная не была объявлена

```
end;
if (1 > (x + (test + (1 + 'c')))) then
^ ^
```

этим операндам | Данную операцию нельзя применить к  
 Переменная не была объявлена

```
begin
```

```
z := 5
^
```

Переменная не была объявлена

```
end;
```

```
end.
```

Код программы:

```
program mfdсain;
var x:integer; y:char;
begin
  if (a + b + c > 0) then
  begin
    if (x > 0) then
    begin
      if (y > x) then
      begin
        z := 5
      end;
    end;
  end;
end.
```

Вывод:

```
program mfdсain;
var x:integer; y:char;
begin
```

```

if (a + b + c > 0) then
  ^   ^   ^
  |   |   Переменная не была объявлена
  |   Переменная не была объявлена
  Переменная не была объявлена

begin
  if (x > 0) then
  begin
    if (y > x) then
      ^
      Данную операцию нельзя применить к этим
операндам

      begin
        z := 5
        ^
        Переменная не была объявлена

      end;

    end;

  end;

end.

```



## 6 Транслятор

### 6.1 Описание

Требуется написать генератор кода для программы Паскаль, генерировать код будем на языке CIL(MSIL). Common Intermediate Language (сокращённо CIL) — «высокоуровневый ассемблер» виртуальной машины .NET. Промежуточный язык, разработанный фирмой Microsoft для платформы .NET Framework. JIT-компилятор CIL является частью CLR (англ. common language runtime) — общей среды выполнения программ, написанных на языках .NET.

### 6.2 Проектирование

Создадим класс Translator, в конструктор которого будет передаваться синтаксически и семантически правильная в виде вектора токенов. Сразу после этого будет генерировать для нее код с помощью модуля System::Reflection::Assembly, для этого пришлось перенести проект на C++/CLI, чтобы воспользоваться модулями из C#.

Создадим приватную переменную – il\_generator. Она будет использоваться для создания тела методов и конструкторов в динамических сборках.

Главной функцией у il\_generator является функция Emit, с помощью нее можно положить инструкцию в поток инструкций, инструкции представлены в классе OpCodes.

В модуле транслятора создадим следующие методы:

- Добавление переменной
- Вызов инструкции исходя из оператора
- Вызов функции вывода в консоль(writeln)
- Вызов функции ввода из консоли(readln)
- Смену знака для значения на стеке
- Функция для загрузки значения константы в стек инструкций

Как видно, мы дополнительно ввели функции writeln и readln, в связи с этим пришлось дополнить синтаксический и семантический анализатор следующими БНФ:

- <readln>::= readln(<переменная>)
- <writeln>::= writeln(<выражение>)

Опкоды которые нам понадобятся:

Название	Описание
Call	Вызывает метод, на который ссылается переданный дескриптор метода.
Stloc	Извлекает верхнее значение в стеке вычислений и сохраняет его в списке локальных переменных с заданным индексом.

Ret	Выполняет возврат из текущего метода, помещая возвращаемое значение (если имеется) из стека вычислений вызываемого метода в стек вычислений вызывающего метода.
Add	Складывает два значения и помещает результат в стек вычислений.
Sub	Вычитает одно значение из другого и помещает результат в стек вычислений.
Div	Делит одно значение на другое и помещает результат в стек вычислений как число с плавающей запятой (с типом F) или как частное (с типом int32).
Mul	Умножает два значения и помещает результат в стек вычислений.
Rem	Делит одно значение на другое и помещает остаток в стек вычислений.
And	Вычисляет побитовое И двух значений и помещает результат в стек вычислений.
Or	Вычисляет побитовое дополнение двух целочисленных значений, находящихся на вершине стека, и помещает результат в стек.
Not	Вычисляет побитовое дополнение целочисленного значения, находящегося на вершине стека, и помещает результат в стек с тем же типом.
Ceq	Сравнивает два значения. Если они равны, целочисленное значение 1 ((int32) помещается в стек вычислений; в противном случае в стек вычислений помещается 0 (int32).
Clt	Сравнивает два значения. Если первое значение меньше второго, целочисленное значение 1 ((int32) помещается в стек вычислений; в противном случае в стек вычислений помещается 0 (int32).
Cgt	Сравнивает два значения. Если первое значение больше второго, целочисленное значение 1 ((int32) помещается в стек вычислений; в противном случае в стек вычислений помещается 0 (int32).
Ldc_I4	Помещает переданное значение с типом int32 в стек вычислений как int32.
Ldc_R8	Помещает переданное значение с типом float64 в стек вычислений как F (число с плавающей запятой).
Ldstr	Помещает в стек ссылку на новый объект, представляющий строковой литерал, хранящийся в метаданных.
Ldloc	Загружает в стек вычислений локальную переменную с указанным индексом.
Brfalse	Передаёт управление конечной инструкции, если значением value является false, пустая ссылка (Nothing в Visual Basic) или ноль.
Br	Обеспечивает безусловную передачу управления конечной инструкции.

## 6.3 Реализация

### 6.3.1 Функция добавления переменной

```
Dictionary<String^, LocalBuilder^> variable_lb;
Dictionary<String^, System::Type^> variable_type;

void Translator::add_var(String^ name, Type^ type)
{
```

```

        variable_lb[name] = il_generator->DeclareLocal(type);
        variable_type[name] = type;
    }

```

### 6.3.2 Функция для вызова инструкции исходя из переданного оператора

```

void Translator::emit_by_operation(OperatorType op)
{
    /*
        {"+", otPlus},
        {"-", otMinus},
        {"/", otSlash},
        {"*", otStar},
        {"=", otEqual},
        {"<", otLess},
        {">", otGreater},
        {">=", otGreaterEqual},
        {"<=", otLessEqual},
        {"<>", otLessGreater}*/
    /*
        otDiv,                // div
        otAnd,                 // and
        otNot,                 // not
        otMod,                 // mod
        otOr
    */
    switch (op)
    {
        case otPlus:                il_generator-
>Emit(OpCodes::Add); break;
        case otMinus:              il_generator-
>Emit(OpCodes::Sub); break;
        case otSlash:              il_generator-
>Emit(OpCodes::Div); break;
        case otStar:               il_generator-
>Emit(OpCodes::Mul); break;
        case otDiv:                il_generator-
>Emit(OpCodes::Div); break;
        case otMod:                il_generator-
>Emit(OpCodes::Rem); break;

        case otAnd:                il_generator-
>Emit(OpCodes::And); break;
        case otOr:                 il_generator-
>Emit(OpCodes::Or); break;
        case otNot:                il_generator-
>Emit(OpCodes::Not); break;

        case otEqual:              il_generator-
>Emit(OpCodes::Ceq); break;
        case otLess:               il_generator-
>Emit(OpCodes::Clt); break;
        case otGreater:            il_generator-
>Emit(OpCodes::Cgt); break;

        case otGreaterEqual:
            il_generator->Emit(OpCodes::Clt);
            il_generator->Emit(OpCodes::Ldc_I4_0);

```

```

        il_generator->Emit(OpCodes::Ceq);
        break;
    case otLessEqual:
        il_generator->Emit(OpCodes::Cgt);
        il_generator->Emit(OpCodes::Ldc_I4_0);
        il_generator->Emit(OpCodes::Ceq);
        break;
    case otLessGreater:
        il_generator->Emit(OpCodes::Ceq);
        il_generator->Emit(OpCodes::Ldc_I4_0);
        il_generator->Emit(OpCodes::Ceq);
        break;
    }
}

```

### 6.3.3 Функция вызова метода вывода в консоль(writeln)

```

void Translator::call_writeln(System::Type^ type)
{
    auto writeln_method = Console::typeid->GetMethod("WriteLine",
gcnew cli::array<Type^>{type});
    il_generator->Emit(OpCodes::Call, writeln_method);
}

```

### 6.3.4 Функция вызова метода ввода из консоли(readln)

```

void Translator::call_readln(Token* token)
{
    String^ name          = get_var_name_from_token(token);
    Type^ type            = variable_type[name];
    LocalBuilder^ lb      = variable_lb[name];

    MethodInfo^ readln_method = Console::typeid-
>GetMethod("ReadLine", gcnew cli::array<Type^>{});
    il_generator->Emit(OpCodes::Call, readln_method);

    MethodInfo^ convert_method = nullptr;

    if (type == int::typeid)
    {
        convert_method = Convert::typeid->GetMethod("ToInt32",
gcnew cli::array<Type^>{string::typeid});
    }
    else if (type == double::typeid)
    {
        convert_method = Convert::typeid-
>GetMethod("ToDouble", gcnew cli::array<Type^>{string::typeid});
    }
    else if (type == bool::typeid)
    {
        convert_method = Convert::typeid-
>GetMethod("ToBoolean", gcnew cli::array<Type^>{string::typeid});
    }
    else if (type == char::typeid)
    {

```

```

        convert_method = Convert::typeid->GetMethod("ToChar",
gcnew cli::array<Type^>{string::typeid});
    }

    if (convert_method != nullptr)
        il_generator->Emit(OpCodes::Call, convert_method);
    il_generator->Emit(OpCodes::Stloc, lb);
}

```

### 6.3.5 Функция для смены знака значения на стеке

```

void Translator::change_sign(System::Type^ type)
{
    if (type == int::typeid) {
        il_generator->Emit(OpCodes::Ldc_I4, -1);
        il_generator->Emit(OpCodes::Mul);
    }
    else if (type == double::typeid) {
        il_generator->Emit(OpCodes::Ldc_R8, -1);
        il_generator->Emit(OpCodes::Mul);
    }
}

```

### 6.3.6 Функция для загрузки константного значения в стек

```

void Translator::load_const(Token* token)
{
    if (auto ct = dynamic_cast<ConstToken<int>*>(token)) {
        il_generator->Emit(OpCodes::Ldc_I4, ct->value);
    }
    else if (auto ct = dynamic_cast<ConstToken<double>*>(token))
    {
        il_generator->Emit(OpCodes::Ldc_R8, ct->value);
    }
    else if (auto ct = dynamic_cast<ConstToken<string>*>(token))
    {
        il_generator->Emit(OpCodes::Ldstr,
convert_to_managed_string(ct->value));
    }
    else if (auto ct = dynamic_cast<ConstToken<char>*>(token)) {
        il_generator->Emit(OpCodes::Ldstr,
convert_to_managed_string(string(1, ct->value)));
    }
    else if (auto ct = dynamic_cast<ConstToken<bool>*>(token)) {
        il_generator->Emit(OpCodes::Ldc_I4, ct->value);
    }
}

```

### 6.3.7 Полный код модуля транслятора

Заголовочный файл Translator.h:

```

#pragma once
#include "Token.h"
#include <string>
#include <vector>
#include <map>
#include <msclr\marshal_cppstd.h>

```

```

using namespace std;

using namespace System;
using namespace System::Reflection::Emit;
using namespace System::Reflection;
using System::Reflection::AssemblyName;
using namespace System::Threading;
using namespace msclr::interop;
using namespace System::Collections::Generic;

ref class Translator
{
private:
    int variable_counter = 0;

    ILGenerator^ il_generator;
    Dictionary<String^, LocalBuilder^> variable_lb;
    Dictionary<String^, System::Type^> variable_type;

    vector<Token*>* tokens = new vector<Token*>();
    int current_token_position;
    Token* current_token;
    OperatorType previous_operator;

    bool accept(TType token_type);
    bool accept(OperatorType operator_type);
    bool accept(vector<OperatorType> operator_types);

    void next_token();

    System::Type^ get_type_from_token(Token* token);
    String^ get_var_name_from_token(Token* token);
    void add_var(String^ name, System::Type^ type);
    void emit_by_operation(OperatorType op);

    String^ convert_to_managed_string(string str);

    void change_sign(System::Type^ type);
    void load_const(Token* token);
    void call_writeln(System::Type^ type);
    void call_readln(Token* token);

    void program();

    void block();

    void vars_section();
    void operators_section();

    bool single_var_definition();

    System::Type^ type();

    void operator_();
    bool simple_operator();
    System::Type^ expression();
    bool relation_operation();
    System::Type^ simple_expression();

```

```

    bool additive_operation();
    System::Type^ term();
    bool multiplicative_operation();
    System::Type^ factor();
    bool sign();
    void complex_operator();
    void necessary_compound_operator();
    bool compound_operator();
    bool if_operator();
    bool while_operator();
    bool writeln();
    bool readln();
public:
    Translator() {};
    Translator(vector<Token*> _tokens);

    ~Translator();
};

```

Исходный файл Translator.cpp:

```

#include "pch.h"
#include "Translator.h"

Translator::Translator(vector<Token*> _tokens)
{
    tokens = &_tokens;

    current_token_position = 0;
    previous_operator = otError;
    next_token();

    String^ assemblyName = "MyProgram";
    String^ modName = "MyProgram.exe";
    String^ typeName = "MyProgram";
    String^ methodName = "Main";

    //Уникальный идентификатор сборки
    AssemblyName^ name = gcnew AssemblyName(assemblyName);

    //Домен приложения
    AppDomain^ domain = Threading::Thread::GetDomain();

    //Определяет и представляет динамическую сборку
    AssemblyBuilder^ builder = domain-
>DefineDynamicAssembly(name, AssemblyBuilderAccess::RunAndSave);

    //Определяет и представляет модуль в динамической сборке
    ModuleBuilder^ module = builder->DefineDynamicModule(modName,
true);

    //Определяет и создает новые экземпляры классов во время
    выполнения
    TypeBuilder^ typeBuilder = module->DefineType(typeName,
TypeAttributes::Public);

    //Определяет и представляет метод для динамического класса

```

```

        MethodBuilder^ methodBuilder = typeBuilder-
>DefineMethod(methodName, MethodAttributes::Static
            | MethodAttributes::Public, void::typeid, gcnew
cli::array<Type^>{});

        il_generator = methodBuilder->GetILGenerator();

        // generate

        /*
        il_generator->DeclareLocal(int::typeid);
        il_generator->Emit(OpCodes::Ldc_I4, 10);
        il_generator->Emit(OpCodes::Stloc, 0);

        auto x = il_generator->DefineLabel();
        auto y = il_generator->DefineLabel();

        il_generator->MarkLabel(x);
        il_generator->Emit(OpCodes::Ldloc, 0);
        il_generator->Emit(OpCodes::Ldc_I4, 100);
        il_generator->Emit(OpCodes::Clt);
        il_generator->Emit(OpCodes::Brfalse, y);

        il_generator->Emit(OpCodes::Ldloc, 0);
        il_generator->Emit(OpCodes::Ldc_I4, 10);
        il_generator->Emit(OpCodes::Add);
        il_generator->Emit(OpCodes::Stloc, 0);
        il_generator->Emit(OpCodes::Ldloc, 0);
        auto writeln_test = Console::typeid->GetMethod("WriteLine",
gcnew cli::array<Type^>{int::typeid});
        il_generator->Emit(OpCodes::Call, writeln_test);
        il_generator->Emit(OpCodes::Br, x);
        il_generator->MarkLabel(y);

        //typeof(Console).GetMethod("WriteLine", new Type[] {
typeof(string) })
        auto writeln_test = Console::typeid->GetMethod("ReadLine",
gcnew cli::array<Type^>{});
        il_generator->Emit(OpCodes::Call, writeln_test);
        auto lb = il_generator->DeclareLocal(int::typeid);
        il_generator->Emit(OpCodes::Stloc, lb);
        */

        program();
        // end generate

        // чтобы консоль не сразу закрывалась
        il_generator->EmitWriteLine(convert_to_managed_string("Press
any button..."));
        auto readln_method = Console::typeid->GetMethod("ReadLine",
gcnew cli::array<Type^>{});
        il_generator->Emit(OpCodes::Call, readln_method);
        auto lb = il_generator->DeclareLocal(String::typeid);
        il_generator->Emit(OpCodes::Stloc, lb);
        //

        il_generator->Emit(OpCodes::Ret);

```



```

        builder->SetEntryPoint(methodBuilder);

        Type^ myClass = typeBuilder->CreateType();

        //Сохраняем в *.exe
        builder->Save(modName);
    }

    Translator::~Translator()
    {
        delete il_generator;
    }

    bool Translator::accept(TType token_type)
    {
        bool result = true;
        if (current_token->token_type != token_type)
            result = false;

        if (result)
            next_token();

        return result;
    }

    bool Translator::accept(OperatorType operator_type)
    {
        bool result = true;
        if (current_token->token_type != ttOperator)
            result = false;

        if (((OperatorToken*)current_token)->operator_type !=
operator_type)
            result = false;

        if (result)
            next_token();

        return result;
    }

    bool Translator::accept(vector<OperatorType> operator_types)
    {
        bool result = false;
        if (current_token->token_type == ttOperator)
        {
            OperatorType current_type =
((OperatorToken*)current_token)->operator_type;

            for (OperatorType operator_type : operator_types)
            {
                if (operator_type == current_type)
                {
                    result = true;
                    break;
                }
            }
        }
    }

```

```

    }

    if (result)
        next_token();

    return result;
}

void Translator::next_token()
{
    if (current_token_position == tokens->size())
        return;

    if (current_token != NULL && current_token->token_type ==
ttOperator)
    {
        previous_operator = ((OperatorToken*)current_token)-
>operator_type;
    }

    current_token = (*tokens)[current_token_position];
    current_token_position++;
}

String^ Translator::get_var_name_from_token(Token* token)
{
    string result = "";
    if (token->token_type == ttIdentificator)
        result = ((IdentificatorToken*)token)->name;

    return convert_to_managed_string(result);
}

Type^ Translator::get_type_from_token(Token* token)
{
    if (auto ct = dynamic_cast<ConstToken<int>*>(token)) {
        return int::typeid;
    }
    else if (auto ct = dynamic_cast<ConstToken<double>*>(token))
    {
        return double::typeid;
    }
    else if (auto ct = dynamic_cast<ConstToken<string>*>(token))
    {
        return string::typeid;
    }
    else if (auto ct = dynamic_cast<ConstToken<char>*>(token)) {
        return char::typeid;
    }
    else
        return bool::typeid;
}

void Translator::add_var(String^ name, Type^ type)
{
    variable_lb[name] = il_generator->DeclareLocal(type);
    variable_type[name] = type;
}

```

```

void Translator::emit_by_operation(OperatorType op)
{
    /*
        {"+", otPlus},
        {"-", otMinus},
        {"/", otSlash},
        {"*", otStar},
        {"=", otEqual},
        {"<", otLess},
        {">", otGreater},
        {">=", otGreaterEqual},
        {"<=", otLessEqual},
        {"<>", otLessGreater}*/
    /*
        otDiv,                // div
        otAnd,                 // and
        otNot,                 // not
        otMod,                 // mod
        otOr
    */
    switch (op)
    {
        case otPlus:                il_generator-
>Emit(OpCodes::Add); break;
        case otMinus:              il_generator-
>Emit(OpCodes::Sub); break;
        case otSlash:              il_generator-
>Emit(OpCodes::Div); break;
        case otStar:               il_generator-
>Emit(OpCodes::Mul); break;
        case otDiv:                il_generator-
>Emit(OpCodes::Div); break;
        case otMod:                il_generator-
>Emit(OpCodes::Rem); break;

        case otAnd:                il_generator-
>Emit(OpCodes::And); break;
        case otOr:                 il_generator-
>Emit(OpCodes::Or); break;
        case otNot:                il_generator-
>Emit(OpCodes::Not); break;

        case otEqual:              il_generator-
>Emit(OpCodes::Ceq); break;
        case otLess:               il_generator-
>Emit(OpCodes::Clt); break;
        case otGreater:            il_generator-
>Emit(OpCodes::Cgt); break;

        case otGreaterEqual:
            il_generator->Emit(OpCodes::Clt);
            il_generator->Emit(OpCodes::Ldc_I4_0);
            il_generator->Emit(OpCodes::Ceq);
            break;
        case otLessEqual:
            il_generator->Emit(OpCodes::Cgt);
            il_generator->Emit(OpCodes::Ldc_I4_0);

```

```

        il_generator->Emit(OpCodes::Ceq);
        break;
    case otLessGreater:
        il_generator->Emit(OpCodes::Ceq);
        il_generator->Emit(OpCodes::Ldc_I4_0);
        il_generator->Emit(OpCodes::Ceq);
        break;
    }
}

void Translator::program() // <программа>::=program <имя>(<имя
файла>{,<имя файла>});<блок>.
{
    accept(otProgram);
    accept(ttIdentificator);
    accept(otSemiColon);

    block();

    accept(otDot);
}

String^ Translator::convert_to_managed_string(string str)
{
    return marshal_as<String^>(str);
}

void Translator::block() // <блок>::=<раздел констант><раздел
типов><раздел переменных><раздел процедур и функций><раздел
операторов>
{
    vars_section();
    operators_section();
}

// ===== Раздел переменных =====
bool Translator::single_var_definition() // <описание однотипных
переменных>::=<имя>{,<имя>}<тип>
{
    List<String^> variableNames;

    variableNames.Add(get_var_name_from_token(current_token));
    if (!accept(ttIdentificator))
        return false;

    while (accept(otComma))
    {
        variableNames.Add(get_var_name_from_token(current_token));
        accept(ttIdentificator);
    }

    accept(otColon);

    Type^ varType = type();

    for (int i = 0; i < variableNames.Count; i++)

```

```

        {
            String^ name = variableNames[i];
            add_var(name, varType);
        }

        return true;
    }

void Translator::call_writeln(System::Type^ type)
{
    auto writeln_method = Console::typeid->GetMethod("WriteLine",
gcnew cli::array<Type^>{type});
    il_generator->Emit(OpCodes::Call, writeln_method);
}

void Translator::call_readln(Token* token)
{
    String^ name          = get_var_name_from_token(token);
    Type^ type            = variable_type[name];
    LocalBuilder^ lb      = variable_lb[name];

    MethodInfo^ readln_method = Console::typeid-
>GetMethod("ReadLine", gcnew cli::array<Type^>{});
    il_generator->Emit(OpCodes::Call, readln_method);

    MethodInfo^ convert_method = nullptr;

    if (type == int::typeid)
    {
        convert_method = Convert::typeid->GetMethod("ToInt32",
gcnew cli::array<Type^>{string::typeid});
    }
    else if (type == double::typeid)
    {
        convert_method = Convert::typeid-
>GetMethod("ToDouble", gcnew cli::array<Type^>{string::typeid});
    }
    else if (type == bool::typeid)
    {
        convert_method = Convert::typeid-
>GetMethod("ToBoolean", gcnew cli::array<Type^>{string::typeid});
    }
    else if (type == char::typeid)
    {
        convert_method = Convert::typeid->GetMethod("ToChar",
gcnew cli::array<Type^>{string::typeid});
    }

    if (convert_method != nullptr)
        il_generator->Emit(OpCodes::Call, convert_method);
    il_generator->Emit(OpCodes::Stloc, lb);
}

Type^ Translator::type() // <тип>:=integer|real|string|char
{
    if (accept(otInteger))
        return int::typeid;
    if (accept(otReal))

```

```

        return double::typeid;
    if (accept(otString))
        return String::typeid;
    if (accept(otChar))
        return Char::typeid;
}

void Translator::vars_section() // <раздел переменных> ::= var
<описание однотипных переменных>; {<описание однотипных переменных>;}
| <пусто>
{
    accept(otVar);

    single_var_definition();
    accept(otSemiColon);

    while (single_var_definition())
    {
        accept(otSemiColon);
    }
}

// ===== Раздел операторов =====
// <раздел операторов> ::= <составной оператор>

void Translator::operators_section()
{
    neccessary_compound_operator();
}

void Translator::change_sign(System::Type^ type)
{
    if (type == int::typeid) {
        il_generator->Emit(OpCodes::Ldc_I4, -1);
        il_generator->Emit(OpCodes::Mul);
    }
    else if (type == double::typeid) {
        il_generator->Emit(OpCodes::Ldc_R8, -1);
        il_generator->Emit(OpCodes::Mul);
    }
}

void Translator::load_const(Token* token)
{
    if (auto ct = dynamic_cast<ConstToken<int>*>(token)) {
        il_generator->Emit(OpCodes::Ldc_I4, ct->value);
    }
    else if (auto ct = dynamic_cast<ConstToken<double>*>(token))
    {
        il_generator->Emit(OpCodes::Ldc_R8, ct->value);
    }
    else if (auto ct = dynamic_cast<ConstToken<string>*>(token))
    {
        il_generator->Emit(OpCodes::Ldstr,
convert_to_managed_string(ct->value));
    }
    else if (auto ct = dynamic_cast<ConstToken<char>*>(token)) {

```

```

        il_generator->Emit(OpCodes::Ldstr,
convert_to_managed_string(string(1, ct->value)));
    }
    else if (auto ct = dynamic_cast<ConstToken<bool>*>(token)) {
        il_generator->Emit(OpCodes::Ldc_I4, ct->value);
    }
}

//<оператор>::=<простой оператор>|<сложный оператор>
void Translator::operator_()
{
    if (!simple_operator())
        complex_operator();
}

//<простой оператор>::=<переменная>:=<выражение>
bool Translator::simple_operator() // *
{
    int mem_position = current_token->position;
    String^ name = get_var_name_from_token(current_token);

    if (!accept(ttIdentificator))
        return false;

    accept(otAssign);
    Type^ t = expression();

    // кладем значение
    il_generator->Emit(OpCodes::Stloc, variable_lb[name]);

    return true;
}

//<выражение>::=<простое выражение>|<простое выражение><операция
отношения><простое выражение>
Type^ Translator::expression()
{
    Type^ t1, ^ t2;
    t1 = simple_expression();
    if (relation_operation())
    {
        OperatorType mem_op = previous_operator;
        t2 = simple_expression();
        emit_by_operation(mem_op);
    }

    return t1;
}

//<операция отношения>::= =|<>|<|<=|>=|>
bool Translator::relation_operation() // *
{
    return accept({ otEqual, otLessGreater, otLess, otLessEqual,
otGreaterEqual, otGreater });
}

//<простое выражение>::=<слагаемое>{<аддитивная
операция><слагаемое>}

```

```

Type^ Translator::simple_expression()
{
    Type ^t1, ^t2;
    t1 = term();
    while (additive_operation())
    {
        OperatorType mem_op = previous_operator;
        t2 = term();
        emit_by_operation(mem_op);
    }
    return t1;
}

//<аддитивная операция>::= +|-|or
bool Translator::additive_operation() // *
{
    return accept({ otPlus, otMinus, otOr });
}

//<слагаемое>::=<множитель>{<мультипликативная операция><множитель>}
Type^ Translator::term()
{
    Type^ t1, ^t2;
    t1 = factor();
    while (multiplicative_operation())
    {
        OperatorType mem_op = previous_operator;
        t2 = factor();
        emit_by_operation(mem_op);
    }

    return t1;
}

//<мультипликативная операция>::=*||div|mod|and
bool Translator::multiplicative_operation() // *
{
    return accept({ otStar, otSlash, otDiv, otMod, otAnd });
}

//<множитель>::=[<знак>]<переменная>| [<знак>]<константа>| [<знак>] (<в
ыражение>)|not <множитель>
Type^ Translator::factor()
{
    // TODO: сделать что-то со знаком...
    OperatorType mem_op = otPlus;
    if (sign())
    {
        mem_op = previous_operator;
    }

    Token* mem_token = current_token;

    if (accept(ttIdentificator))
    {
        String^ name = get_var_name_from_token(mem_token);

        il_generator->Emit(OpCodes::Ldloc, variable_lb[name]);
    }
}

```



```

        auto type = variable_type[name];

        if (mem_op == otMinus)
            change_sign(type);

        return type;
    }
    else if (accept(ttConst))
    {
        Type^ const_type = get_type_from_token(mem_token);

        load_const(mem_token);

        if (mem_op == otMinus)
            change_sign(const_type);

        return const_type;
    }
    else if (accept(otLeftParenthesis))
    {
        Type^ type = expression();
        accept(otRightParenthesis);

        if (mem_op == otMinus)
            change_sign(type);

        return type;
    }
    else if (accept(otNot))
    {
        Type^ t = factor();

        emit_by_operation(otNot);

        return bool::typeid;
    }
}

//<знак>::= +|-
bool Translator::sign() // *
{
    return accept({ otPlus, otMinus });
}

//<сложный оператор>::=<составной оператор>|<выбирающий
оператор>|<оператор цикла>|<вraithлайн>
void Translator::complex_operator()
{
    if (compound_operator())
    {
        //...
    }
    else if (if_operator())
    {
        //...
    }
    else if (while_operator())

```

```

        {
            //...
        }
        else if (writeln())
        {
            //...
        }
        else if (readln())
        {
            //...
        }
    }
}

//<составной оператор>::= begin <оператор>{;<оператор>} end
bool Translator::compound_operator() // *
{
    if (!accept(otBegin))
        return false;

    operator_();

    while (accept(otSemiColon))
    {
        operator_();
    }

    accept(otEnd);

    return true;
}

//<обязательный составной оператор>::= begin <оператор>{;<оператор>}
end
void Translator::necessary_compound_operator() // *
{
    accept(otBegin);

    operator_();

    while (accept(otSemiColon))
    {
        operator_();
    }

    accept(otEnd);
}

//<выбирающий оператор>::= if <выражение> then <оператор>|if
<выражение> then <оператор> else <оператор>
bool Translator::if_operator() // *
{
    /*
    il_generator->DeclareLocal(int::typeid);
    il_generator->Emit(OpCodes::Ldc_I4, 10);
    il_generator->Emit(OpCodes::Stloc, 0);

    auto x = il_generator->DefineLabel();
    auto y = il_generator->DefineLabel();
    */
}

```

```

    il_generator->MarkLabel(x);
    il_generator->Emit(OpCodes::Ldloc, 0);
    il_generator->Emit(OpCodes::Ldc_I4, 100);
    il_generator->Emit(OpCodes::Clt);
    il_generator->Emit(OpCodes::Brfalse, y);

    il_generator->Emit(OpCodes::Ldloc, 0);
    il_generator->Emit(OpCodes::Ldc_I4, 10);
    il_generator->Emit(OpCodes::Add);
    il_generator->Emit(OpCodes::Stloc, 0);
    il_generator->Emit(OpCodes::Ldloc, 0);
    auto writeln_test = Console::typeid->GetMethod("WriteLine",
gcnew cli::array<Type^>{int::typeid});
    il_generator->Emit(OpCodes::Call, writeln_test);
    il_generator->Emit(OpCodes::Br, x);
    il_generator->MarkLabel(y);*/
    if (!accept(otIf))
        return false;

    auto then_pointer = il_generator->DefineLabel();
    auto after_pointer = il_generator->DefineLabel();

    Type^ t = expression();
    il_generator->Emit(OpCodes::Brfalse, then_pointer);

    accept(otThen);
    operator_();
    il_generator->Emit(OpCodes::Br, after_pointer);

    il_generator->MarkLabel(then_pointer);
    if (accept(otElse))
    {
        operator_();
    }
    il_generator->MarkLabel(after_pointer);

    return true;
}

//<оператор цикла>::= while <выражение> do <оператор>
bool Translator::while_operator() // *
{
    if (!accept(otWhile))
        return false;

    auto condition_pointer = il_generator->DefineLabel();
    auto after_pointer = il_generator->DefineLabel();

    il_generator->MarkLabel(condition_pointer);
    Type^ t = expression();
    il_generator->Emit(OpCodes::Brfalse, after_pointer);

    accept(otDo);

    operator_();
    il_generator->Emit(OpCodes::Br, condition_pointer);

```

```

        il_generator->MarkLabel(after_pointer);

        return true;
    }

bool Translator::writeln()
{
    if (!accept(otWriteLn))
        return false;

    accept(otLeftParenthesis);
    Type^ t = expression();
    accept(otRightParenthesis);

    call_writeln(t);

    return true;
}

bool Translator::readln()
{
    if (!accept(otReadLn))
        return false;

    accept(otLeftParenthesis);
    Token* mem_token = current_token;
    accept(ttIdentificator);

    call_readln(mem_token);

    accept(otRightParenthesis);

    return true;
}

```

## 6.4 Тестирование

Для просмотра инструкций сгенерированной программы воспользуемся инструментом дизассемблирования программы – ildasm.

### 6.4.1 Ввод и вывод числа

Исходный код программы:

```

program test;
var x:integer;
begin
    readln(x);
    writeln("Vvedennoe chislo:");
    writeln(x)
end.

```

Работа программы:

```
C:\Users\ilyar\source\repos\SimpleCompiler_with_Gen\SimpleCompiler_with_Gen\build.exe
1337
Vvedennoe chislo:
1337
Press any button...
```

Код на языке MSIL:

```
build::main : void()
Найти Найти далее
.method public static void main() cil managed
{
    .entrypoint
    // Размер кода:      44 (0x2c)
    .maxstack 1
    .locals init ([0] int32 V_0,
                  [1] string V_1)
    IL_0000: call      string [mscorlib]System.Console::ReadLine()
    IL_0005: call      int32 [mscorlib]System.Convert::ToInt32(object)
    IL_000a: stloc.0
    IL_000b: ldstr      "Vvedennoe chislo:"
    IL_0010: call      void [mscorlib]System.Console::WriteLine(object)
    IL_0015: ldloc.0
    IL_0016: call      void [mscorlib]System.Console::WriteLine(int32)
    IL_001b: ldstr      "Press any button..."
    IL_0020: call      void [mscorlib]System.Console::WriteLine(string)
    IL_0025: call      string [mscorlib]System.Console::ReadLine()
    IL_002a: stloc.1
    IL_002b: ret
} // end of method build::main
```

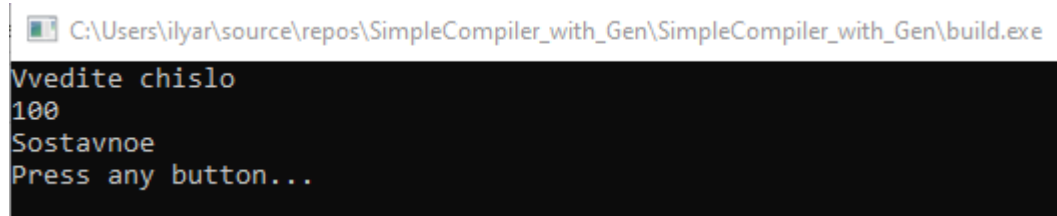
## 6.4.2 Проверка введенного числа на простоту

Исходный код программы:

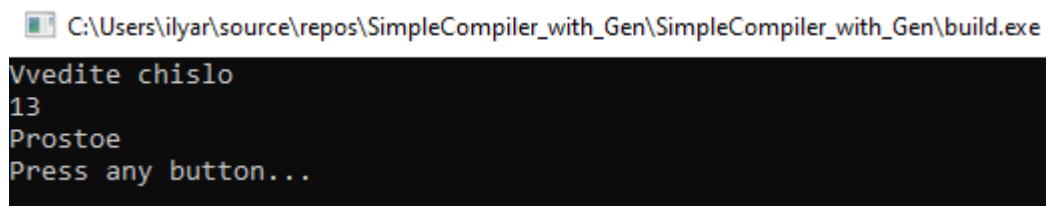
```
program isprime;
var x,i,c:integer;
begin
    writeln("Vvedite chislo");
    readln(x);
    c:=0;
    i:=2;
    while (i < x) do
    begin
        if (x mod i = 0) then
```

```
                c := c + 1;
            i := i+1
        end;
    if (c > 0) then
        writeln("Sostavnoe")
    else
        writeln("Prostoe")
    end.
end.
```

Результат работы программы:



```
C:\Users\ilyar\source\repos\SimpleCompiler_with_Gen\SimpleCompiler_with_Gen\build.exe
Vvedite chislo
100
Sostavnoe
Press any button...
```



```
C:\Users\ilyar\source\repos\SimpleCompiler_with_Gen\SimpleCompiler_with_Gen\build.exe
Vvedite chislo
13
Prostoe
Press any button...
```

Код на языке MSIL:

```

.method public static void main() cil managed
{
    .entrypoint
    // Размер кода:      138 (0x8a)
    .maxstack 7
    .locals init ([0] int32 V_0,
                  [1] int32 V_1,
                  [2] int32 V_2,
                  [3] string V_3)
    IL_0000: ldstr      "Uvedite chislo"
    IL_0005: call       void [mscorlib]System.Console::WriteLine(object)
    IL_000a: call       string [mscorlib]System.Console::ReadLine()
    IL_000f: call       int32 [mscorlib]System.Convert::ToInt32(object)
    IL_0014: stloc.0
    IL_0015: ldc.i4      0x0
    IL_001a: stloc.2
    IL_001b: ldc.i4      0x2
    IL_0020: stloc.1
    IL_0021: ldloc.1
    IL_0022: ldloc.0
    IL_0023: clt
    IL_0025: brfalse   IL_0053
    IL_002a: ldloc.0
    IL_002b: ldloc.1
    IL_002c: rem
    IL_002d: ldc.i4      0x0
    IL_0032: ceq
    IL_0034: brfalse   IL_0046
    IL_0039: ldloc.2
    IL_003a: ldc.i4      0x1
    IL_003f: add
    IL_0040: stloc.2
    IL_0041: br       IL_0046
    IL_0046: ldloc.1
    IL_0047: ldc.i4      0x1
    IL_004c: add
    IL_004d: stloc.1
    IL_004e: br       IL_0021
    IL_0053: ldloc.2
    IL_0054: ldc.i4      0x0
    IL_0059: cgt
    IL_005b: brfalse   IL_006f
    IL_0060: ldstr      "Sostavnoe"
    IL_0065: call       void [mscorlib]System.Console::WriteLine(object)
    IL_006a: br       IL_0079
    IL_006f: ldstr      "Prostoe"
    IL_0074: call       void [mscorlib]System.Console::WriteLine(object)
    IL_0079: ldstr      "Press any button..."
    IL_007e: call       void [mscorlib]System.Console::WriteLine(string)
    IL_0083: call       string [mscorlib]System.Console::ReadLine()
    IL_0088: stloc.3
    IL_0089: ret
} // end of method build::main

```

## 6.4.2 Нахождение корня для полинома третьей степени

Исходный код:

```

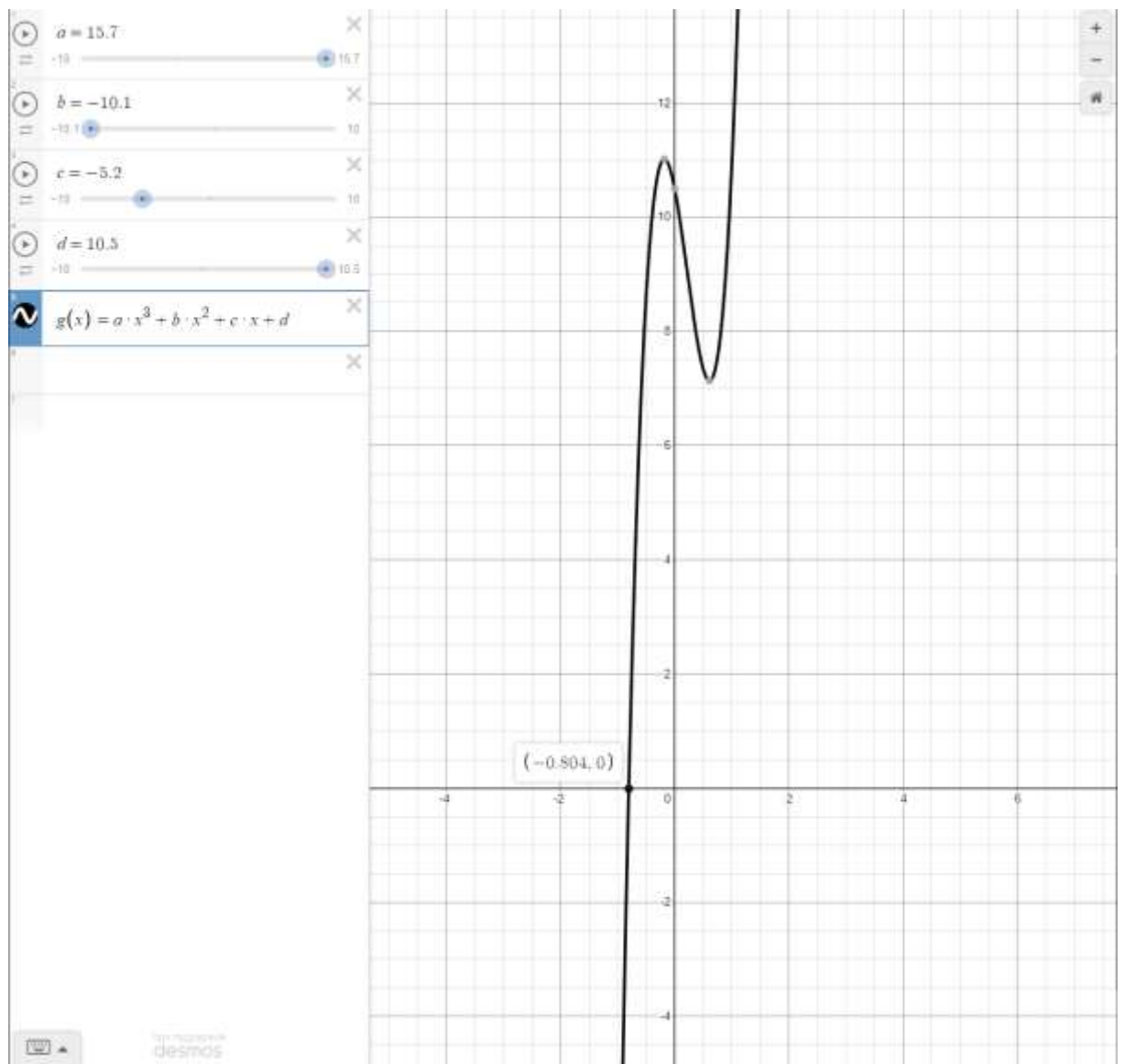
program root_p3;
var x,a,b,c,d,res,eps,l,r,m,fl,fr,fm:real;
begin
    writeln("Vvedite koeficienti: a,b,c,d");
    readln(a);
    readln(b);
    readln(c);
    readln(d);
    eps := 0.00000001;
    l:=-1000.0;
    r:=1000.0;

    while (r - l > eps) do
    begin
        m := (l+r) / 2.0;
        fm:= a * m * m * m + b * m * m + c * m + d;
        fr:= a * r * r * r + b * r * r + c * r + d;
        if (fr * fm < 0.0) then
            l := m
        else
            r := m
        end;
        writeln((l+r)/2.0)
    end.
end.

```

Введем коэф-ы следующей функции:





Вывод программы:

```
C:\Users\ilyar\source\repos\SimpleCompiler_with_Gen\SimpleCompiler_with_Gen\build.exe
Vvedite koeficienti: a,b,c,d
15,7
-10,1
-5,2
10,5
-0,803817220003111
Press any button...
```

Код программы на язык MSIL:

```

.method public static void main() cil managed
{
    .entrypoint
    // Размер кода:      262 (0x106)
    .maxstack 6
    .locals init ([0] float64 U_0,
        [1] float64 U_1,
        [2] float64 U_2,
        [3] float64 U_3,
        [4] float64 U_4,
        [5] float64 U_5,
        [6] float64 U_6,
        [7] float64 U_7,
        [8] float64 U_8,
        [9] float64 U_9,
        [10] float64 U_10,
        [11] float64 U_11,
        [12] float64 U_12,
        [13] string U_13)
    IL_0000: ldstr      "Uvedite koeficienti: a,b,c,d"
    IL_0005: call       void [mscorlib]System.Console::WriteLine(object)
    IL_000a: call       string [mscorlib]System.Console::ReadLine()
    IL_000f: call       float64 [mscorlib]System.Convert::ToDouble(object)
    IL_0014: stloc.1
    IL_0015: call       string [mscorlib]System.Console::ReadLine()
    IL_001a: call       float64 [mscorlib]System.Convert::ToDouble(object)
    IL_001f: stloc.2
    IL_0020: call       string [mscorlib]System.Console::ReadLine()
    IL_0025: call       float64 [mscorlib]System.Convert::ToDouble(object)
    IL_002a: stloc.3
    IL_002b: call       string [mscorlib]System.Console::ReadLine()
    IL_0030: call       float64 [mscorlib]System.Convert::ToDouble(object)
    IL_0035: stloc.s   U_4
    IL_0037: ldc.r8     1.e-008
    IL_0040: stloc.s   U_6
    IL_0042: ldc.r8     1000.
    IL_004b: ldc.r8     -1.
    IL_0054: mul
    IL_0055: stloc.s   U_7
    IL_0057: ldc.r8     1000.
    IL_0060: stloc.s   U_8
    IL_0062: ldloc.s   U_8
    IL_0064: ldloc.s   U_7
    IL_0066: sub
    IL_0067: ldloc.s   U_6
    IL_0069: cgt
    IL_006b: brfalse    IL_00e0
    IL_0070: ldloc.s   U_7
    IL_0072: ldloc.s   U_8
    IL_0074: add
    IL_0075: ldc.r8     2.
    IL_007e: div
    IL_007f: stloc.s   U_9
    IL_0081: ldloc.1
    IL_0082: ldloc.s   U_9
    IL_0084: mul
    IL_0085: ldloc.s   U_9
    IL_0087: mul
    IL_0088: ldloc.s   U_9
    IL_008a: mul
    IL_008b: ldloc.2
    IL_008c: ldloc.s   U_9
    IL_008e: mul
    IL_008f: ldloc.s   U_9
    IL_0091: mul
    IL_0092: add
    IL_0093: ldloc.3
    IL_0094: ldloc.s   U_9
    IL_0096: mul
    IL_0097: add
    IL_0098: ldloc.s   U_4
    IL_009a: add
    IL_009b: stloc.s   U_12
    IL_009d: ldloc.1
    IL_009e: ldloc.s   U_8
    IL_00a0: mul
    IL_00a1: ldloc.s   U_8
    IL_00a3: mul
    IL_00a4: ldloc.s   U_8
    IL_00a6: mul
    IL_00a7: ldloc.2
    IL_00a8: ldloc.s   U_8
    IL_00aa: mul

```

```

IL_00ab: ldloc.s    V_8
IL_00ad: mul
IL_00ae: add
IL_00af: ldloc.s    V_8
IL_00b0: ldloc.s    V_8
IL_00b2: mul
IL_00b3: add
IL_00b4: ldloc.s    V_4
IL_00b6: add
IL_00b7: stloc.s    V_11
IL_00b9: ldloc.s    V_11
IL_00bb: ldloc.s    V_12
IL_00bd: mul
IL_00be: ldc.r8     0.0
IL_00c7: clt
IL_00c9: brfalse   IL_00d7
IL_00ce: ldloc.s    V_9
IL_00d0: stloc.s    V_7
IL_00d2: br        IL_00db
IL_00d7: ldloc.s    V_9
IL_00d9: stloc.s    V_8
IL_00db: br        IL_0062
IL_00e0: ldloc.s    V_7
IL_00e2: ldloc.s    V_8
IL_00e4: add
IL_00e5: ldc.r8     2.
IL_00ee: div
IL_00ef: call      void [mscorlib]System.Console::WriteLine(float64)
IL_00f4: ldstr     "Press any button..."
IL_00f9: call      void [mscorlib]System.Console::WriteLine(string)
IL_00fe: call      string [mscorlib]System.Console::ReadLine()
IL_0103: stloc.s    V_13
IL_0105: ret
} // end of method build::main

```