

АНОО "Физтех-лицей" им. П. Л. Капицы

**Техническая документация
к транслятору**

Содержание

I	Введение и информация об авторах	2
II	Руководство программиста	2
1	Теоретическое описание	2
2	Архитектура	3
3	Система исключений (обработка ошибок)	4
4	Описание функций и методов классов	5
III	Руководство пользователя	10
5	Аннотация	10
6	Ввод кода программы	10
7	Структура программы	11
8	Переменные и массивы	12
9	Операторы	12
10	Функции	13
11	Выражения	13
12	Возможные проблемы	14
13	FAQ	14
IV	Контакты	15

Часть I

Введение и информация об авторах

Мы рады представить Вам транслятор, грамматика которого основана на языке C++. Он разработан Севрюковым Никитой и Балдиной Анастасией в рамках финального практикума для учеников 11 класса информационно-технологического профиля Физтех-лицея им. П. Л. Капицы, г. Долгопрудный. В руководстве подробно описывается работа данного транслятора как для программистов, так и для обычных пользователей.

Часть II

Руководство программиста

1 Теоретическое описание

Язык, по которому происходит интерпретация вводимого кода, соответствует грамматике по нотационным формам Бэкуса-Наура:

```
<программа> ::= {<описание>; | <функция>} int main() <составной оператор>
<составной оператор> ::= {<оператор>}
<оператор> ::= <оператор ввода> | <оператор вывода> | <оператор возврата> | <описание> | <оператор выражения> | <оператор цикла> | <оператор условия> | <оператор вызова функции>

<оператор выражения> ::= <выражение>;
<const> ::= [<знак>]<цифра>{<цифра>} | [<знак>]<цифра>{<цифра>}.<цифра>{<цифра>}
<знак> ::= + | -
<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<bool> ::= true | false

<начальное объявление> ::= <const> | <bool> | !<объявление>
<объявление> ::= <начальное объявление> | <имя> | (<выражение>) | <имя>”[<выражение>]”

<приоритет 0> ::= [<знак>] <объявление> | <инкремент><объявление>
<приоритет 1> ::= <приоритет 0> | <приоритет 0><операции 1><приоритет 1>
<приоритет 2> ::= <приоритет 1> | <приоритет 1><операции 2><приоритет 2>
<приоритет 3> ::= <приоритет 2> | <приоритет 2><операции 3><приоритет 3>
<приоритет 4> ::= <приоритет 3> | <приоритет 3><операции 4><приоритет 4>
<операции 1> ::= * | / | %
<операции 2> ::= + | -
<операции 3> ::= < | > | <= | >=
<операции 4> ::= == | <>
<приоритет 5> ::= <приоритет 4> | <приоритет 4> & <приоритет 5>
<приоритет 6> ::= <приоритет 5> | <приоритет 5> | <приоритет 6>
```

```

<приоритет 7> ::= <приоритет 6> | <приоритет 6> ^ <приоритет 7>
<приоритет 8> ::= <приоритет 7> | <приоритет 7> && <приоритет 8>
<приоритет 9> ::= <приоритет 8> | <приоритет 8> || <приоритет 9>
<приоритет 10> ::= <приоритет 9> | <имя> = <приоритет 9> | <имя> [<выражение>] =
<приоритет 9>
<выражение> ::= <приоритет 10>

<оператор ввода> ::= in » <имя> ['<выражение>'] { » <имя> ['<выражение>'] };
<оператор вывода> ::= out « <выражение> | <строка> | <разделитель> « <выражение> |
| <строка> | <разделитель>;
<разделитель> ::= \n | ' '
<переменная> ::= <тип><имя> [ = <выражение> ]
<массив> ::= array <тип><имя> [<const>]
<имя> ::= _ | <буква> { _ | <буква> | <цифра> }
<оператор возврата> ::= return [<выражение>];
<оператор цикла> ::= <оператор for> | <оператор while>
<оператор for> ::= for(<выражение> | <тип><имя><выражение>; <выражение>; <выра-
жение>) <оператор>
<описание> ::= [<переменная> | <массив>] {, <переменная> | <массив>}
<оператор while> ::= while(<выражение>) <оператор>
<оператор условия> ::= if(<выражение>)<оператор>{else if(<выражение>)<оператор>}
[else<оператор>]
<функция> ::= <тип><имя>({<тип><имя>}) <составной оператор> | void<имя>(<тип><имя>)
<составной оператор>
<оператор вызова функции> ::= <имя>() | <имя>(<выражение>{, <выражение>});
<тип> ::= int | double | bool
<инкремент> ::= ++ | -
<строка> ::= "{<символ>}"
<комментарий> ::= // <строка>

```

Примечание: в коде программы каждое определение записано на английском языке.

2 Архитектура

Всего существует 7 независимых классов:

1. Lexeme - класс лексемы. Содержит в себе саму лексему, её тип и номер строки в коде, на которой она находится.
2. LexicalAnalyzator - класс лексического анализатора. Содержит методы для обработки лексем разных типов и их последующего сохранения в массив, с которым будут работать следующие этапы. Главный метод - Analiz().
3. SyntaxAnalyzator - класс синтаксического анализатора. Анализирует текст программы с точки зрения грамматики (см. "Теоретическое описание"). Главный метод - Check(), который возвращает true, в случае соответствия кода с грамматикой, и наоборот. Бросает

исключение в случае несоответствия, таким образом достигается **дифференцированная обработка ошибок**.

4. `SemanticsAnalyzer` - класс семантического анализатора. Главный метод - `Check()`, который возвращает `true`, если код семантически правильный, и наоборот. Как и синтаксическом анализаторе, есть обработка исключений. Используется **дерево идентификаторов** (отдельное для переменных и функций).
5. `Generator` - класс генератора. Переводит код программы в ПОЛИЗ и сохраняет его в массив.
6. `Executor` - класс исполнителя. Получает на вход массив, который создал генератор, и исполняет код.
7. `Compiler` - "движущий" класс компилятора. Метод `bool Check()` проверяет код на синтаксис и семантику. Если в коде есть ошибки, то выведет "!!!not OK!!!" и ошибку на следующей строке. Если все хорошо, то выведет "ОК". Метод `void Start()` - запускает проверку и исполнение программы (если код правильный).

Также присутствуют некоторые структуры:

1. `Variable` - структура переменной для дерева идентификаторов.
2. `Function` - структура функции для дерева идентификаторов.
3. `NodeV` - "вершина" дерева идентификаторов, в котором хранятся переменные (структура `Variable`) на текущем уровне.
4. `NodeF` - "вершина" дерева идентификаторов функций. По сути, двунаправленный список, в элементах которого хранятся функции (структура `Function`)
5. `TypeControl` - структура, которая используется для обработки выражений (в нём хранится лексема, номер её строки и какого она типа данных)

3 Система исключений (обработка ошибок)

Как и в любом компиляторе, в нашем есть обработка ошибок. Разделяются на:

1. Лексическая - ошибка ввода несуществующего символа.
2. Синтаксические - ошибки несоответствия грамматике.
3. Семантические - ошибки связанные напрямую со смысловым содержанием введенного кода программы: существование переменной, функции, соответствие типов в выражениях и т. д.

В функции `Analiz()` присутствует непосредственный вывод ошибки (ввод несуществующего). А в функциях `Check()` синтаксического и семантических анализаторов бросается `throw error`, где `error` - текст ошибки. Выглядит это так:

```

bool Check() {
    try {
        // обработка
    }
    catch (std::string error) {
        std::cout << "ERROR: " << error << '\n';
        return false;
    }
    return true;
}

```

4 Описание функций и методов классов

Здесь более подробно описаны все функции и их методы.

```

class Lexeme {
public:
    Lexeme() = default;
    ~Lexeme() = default;
    Lexeme(std::string text, int type, int num); // принимает текст лексемы, её тип
        // (не тип данных!), и номер строки
    std::string GetText();
    int GetType();
    int GetNum();
    void Show(); // выводит лексему
private:
    std::string text_; // текст лексемы
    int type_; // тип лексемы
    int num_; // номер строки
};

class LexicalAnalyzer {
public:
    LexicalAnalyzer() = default;
    ~LexicalAnalyzer() = default;

    // const char& a - передаваемый символ из функции Analiz()
    bool Letter(const char& a); // проверка символа на букву
    bool Number(const char& a); // проверка символа на константу
    bool NumAndLetAndUnderscore(const char& a); // проверка символа на
        // ниж. подчёркивание, букву или число
    bool Punctuation(const char& a); // проверка на пунктуационные символы
    bool Operation(const char& a); // проверка на операции
    bool DoubleOperation(const std::string& a); // проверка на операции,
        // состоящих из двух символов
    std::vector<Lexeme> Analiz(bool& fl); // функция обрабатывает код
        // введённой программы и возвращает список (массив) лексем

```

```
};
```

```
class SyntaxAnalyzer {
public:
    SyntaxAnalyzer() = default;
    ~SyntaxAnalyzer() = default;
    void GetLexemes(std::string file_path); // принимает путь в файл с лексемами и
    // сохраняет их в массив лексем
    void SetLexemes(std::vector<Lexeme> vec); // принимает массив лексем
    // и записывает их в поле lexemes_
    void Show(); // выводит список лексем
    bool Check(); // обрабатывает список лексем и в случае соответствия
    // возвращает true
    void CheckLexeme(std::string str); // принимает ОЖИДАЕМУЮ лексему по грамматике,
    // и, если текущая лексема ей не равна, бросает
    // исключение (ожидалось ..., получено ...)
    std::string GetLex(); // возвращает текущую лексему (в процессе
    // обработки лексем есть счётчик i, проходящий по списку лексем)
    int GetType();
    std::string GetNum();
    void Next(); // перемещает на следующую лексему (фактически, ++i)
    bool CheckOperator(); // проверка на служебные слова, идентификаторы
    // или операции (унарные и бинарные)

    // ! Последующие методы являются реализацией того или иного понятия из грамматики
    // Изначально в методе Check() вызывается Program() (нач. символ программы)
    // Далее по методу рекурсивного спуска вызываются остальные
    // Эти методы будут поясняться определением из грамматики (на русском языке).
    // Все из этих методов зовут GetLex().
    // Какие-то из этих методов вспомогательные и не присутствуют в грамматике
    void Program(); // программа
    void ProgramParameters(); // несколько функций или переменных в начале программы
    void DefinitionOperator(); // оператор описания
    void DefinitionParameters(); // несколько переменных
    void Definition(); // описание
    void FunctionParameters(); // несколько параметров функции
    void Function(); // функция
    void CompoundOperatorParameters(); // несколько операторов
    void CompoundOperator(); // составной оператор
    void Operator(); // оператор
    void InputParameters(); // параметры для оператора ввода
    void InputOperator(); // оператор ввода
    void OutputOperator(); // оператор вывода
    void OutputOperatorParameters(); // несколько выводимых значений
    void ReturnOperator(); // оператор возврата
    void ExpressionOperator(); // оператор выражения
    void CycleOperator(); // оператор цикла

```

```

void ElseIf(); // аналог else if
void IfOperator(); // условный оператор
void FunctionCallOperatorParameters(); // параметры для оператора вызова функции
void FunctionCallOperator(); // оператор вызова функции
void Expression(); // выражение
void Const(); // константа (число)
void Sign(); // знак
void Bool(); // bool
void StartAdding(); // начальное объявление
void AddingFunctionParameters(); // ввод параметров вызова функции
void Adding(); // объявление
void Name(); // идентификатор
void Priority0(); // приоритет 0
void Priority1(); // приоритет 1
void Priority2(); // приоритет 2
void Priority3(); // приоритет 3
void Priority4(); // приоритет 4
void Priority5(); // приоритет 5
void Priority6(); // приоритет 6
void Priority7(); // приоритет 7
void Priority8(); // приоритет 8
void Priority9(); // приоритет 9
void Priority10(); // приоритет 10
void Operation1(); // операции 1
void Operation2(); // операции 2
void Operation3(); // операции 3
void Operation4(); // операции 4
void Variable(); // переменная
void Array(); // массив
void ForOperator(); // оператор for
void WhileOperator(); // оператор while
void Type(); // тип
void Increment(); // инкремент
void String(); // строка

private:
    std::vector<Lexeme> lexemes_; // список лексем
};

struct Variable {
    std::string id; // название переменной
    std::string type; // тип переменной
    bool is_init = false; // инициализирована ли переменная
};

struct Function {
    std::string id; // название функции
    std::string type; // тип функции

```



```

    std::vector<std::string> type_par; // массив типов параметров функции
};

struct NodeV {
    std::vector<Variable> vars; // переменные на текущем уровне дерева идентификаторов
    NodeV* next = nullptr; // указатель на следующую вершину
    NodeV* prev = nullptr; // указатель на предыдущую вершину
};

struct NodeF {
    Function function; // информация о функции, которая лежит в этом элементе списка
    NodeF* next = nullptr; // указатель на следующий элемент
    NodeF* prev = nullptr; // указатель на предыдущий элемент
};

// эту структуру использует стек типов в выражениях
struct TypeControl {
    std::string type; // тип "подвыражения"
    std::string id; // поле пустое, если производились какие-то операции
    // либо записан идентификатор, к которому относится эта лексема
    int line; // номер строки для обработки ошибок в выражениях
};

// в дальнейшем будем называть дерево идентификаторов TID
class SemanticsAnalyzer {
public:
    SemanticsAnalyzer() = default;
    ~SemanticsAnalyzer() = default;
    void SetLexemes(std::vector<Lexeme> l); // принимает список лексем
    // и записывает в lexemes_

    bool Check(); // проверяет семантику, и, если все соответствует,
    // возвращает true, иначе возвращает false и бросает исключение
    bool FindVar(NodeV*& cur,
        std::string type, std::string id,
        bool init, int line); // находит в TID переменную
    // проверяет её на существование, инициализацию и тип.
    // Бросает исключение и возвращает false в плохом случае
    bool FindFunc(NodeF*& cur,
        std::string type, std::string id,
        std::vector<std::string> p, int line); // находит в TID функцию,
    // проверяет на существование, тип и типы переменных
    Variable SearchVar(NodeV*& cur, std::string id, int line); // находит переменную
    // в списке TID и возвращает её или бросает исключение, если переменной не сущ.
    NodeF* SearchFunc(NodeF*& cur, std::string id, int line); // аналогично с функцией

    void CheckBin(std::stack<TypeControl>& type_control); // берёт последние три

```

```

// элемента стека, обрабатывает их и кладёт в стек тип, который получится, если
// произвести операцию (второй элемент - операция)
// бросает исключение в случае несоответствия типов
void CheckUno(std::stack<TypeControl>& type_control); // аналогично, только берёт
// два последних элемента и обрабатывает унарные операции

// ! Последующие методы - обработка выражений и добавление операндов
// и операций в стек !
// К какому определению относится тот или иной метод,
// написано в описании класса SyntaxAnalyzator
void Priority0(std::stack<TypeControl>& type_control, int& i);
void Priority1(std::stack<TypeControl>& type_control, int& i);
void Priority2(std::stack<TypeControl>& type_control, int& i);
void Priority3(std::stack<TypeControl>& type_control, int& i);
void Priority4(std::stack<TypeControl>& type_control, int& i);
void Priority5(std::stack<TypeControl>& type_control, int& i);
void Priority6(std::stack<TypeControl>& type_control, int& i);
void Priority7(std::stack<TypeControl>& type_control, int& i);
void Priority8(std::stack<TypeControl>& type_control, int& i);
void Priority9(std::stack<TypeControl>& type_control, int& i);
void Priority10(std::stack<TypeControl>& type_control, int& i);
void Operation1(std::stack<TypeControl>& type_control, int& i);
void Operation2(std::stack<TypeControl>& type_control, int& i);
void Operation3(std::stack<TypeControl>& type_control, int& i);
void Operation4(std::stack<TypeControl>& type_control, int& i);
void Expression(std::stack<TypeControl>& type_control, int& i);
void StartAdding(std::stack<TypeControl>& type_control, int& i);
void AddingFunctionParameters(std::stack<TypeControl>& type_control,
    int& i, Function f, int& ind_par);
void Adding(std::stack<TypeControl>& type_control, int& i);
private:
    std::vector<Lexeme> lexemes_; // список лексем
};

void AddVar(std::string id, std::string type, bool is_init);
// добавляет переменную в TID. Вызывает Check()

void AddFunc(std::string id, std::string type, std::vector<std::string> pars);
// добавляет функцию в TID. Вызывает Check()

bool CheckEqualFunctions(NodeF*& cur, std::string id, std::vector<std::string> par);
// проверка на равенство функции в TID и принимаемой в качестве аргумента
// Вызывает FuncTID()

void DeleteFunc(); // удаляет последнюю функцию из TID

void NewVarScope(); // создаёт новое пространство имён для переменных

```

```

// вызывает - Check()

void DeleteVarScope(); // удаляет последнее пространство имён из TID
// вызывает - Check()

// Следующие три метода вызываются методами CheckBin() и CheckUno()
bool IntOperation(std::string s); // проверка строки на операции,
// связанные с целыми числами

bool DoubleOperation(std::string s); // проверка строки на операции,
// связанные с вещественными числами

bool LogicalOperation(std::string s); // проверка строки на операции,
// связанные с логическим типом

```

Часть III

Руководство пользователя

5 Аннотация

Вы читаете руководство пользователя для использования транслятора по назначению. В этой части технической документации приведено подробное описание устройства компилятора и рассказано, как обычному пользователю с ним работать.

6 Ввод кода программы

Для ввода кода программы необходимо занести её текст в файл "inputLexical.txt", содержащийся в каталоге "...\\art_of_compilation\\art_of_compilation\\inputLexical.txt" (лежит в одной папке вместе с файлом "main.cpp"):

art_of_compilation > art_of_compilation >					Поиск в: art_of_compilation
Имя	Дата изменения	Тип	Размер		
Debug	24.03.2023 9:24	Папка с файлами			
Release	06.02.2023 23:47	Папка с файлами			
art_of_compilation.vcxproj	13.03.2023 22:12	VC++ Project	8 КБ		
art_of_compilation.vcxproj.filters	13.03.2023 22:12	VC++ Project Filte...	3 КБ		
art_of_compilation.vcxproj.user	06.02.2023 12:54	Per-User Project O...	1 КБ		
Compiler.cpp	23.03.2023 16:28	Файл "CPP"	1 КБ		
Compiler.h	28.02.2023 20:48	JetBrains CLion	1 КБ		
inputLexical.txt	24.03.2023 9:38	Текстовый докум...	1 КБ		
Lexeme.cpp	08.02.2023 22:54	Файл "CPP"	1 КБ		
Lexeme.h	08.02.2023 21:48	JetBrains CLion	1 КБ		
LexicalAnalyzator.cpp	08.02.2023 22:57	Файл "CPP"	4 КБ		
LexicalAnalyzator.h	08.02.2023 14:27	JetBrains CLion	1 КБ		
lexicalSettings.txt	10.01.2023 20:38	Текстовый докум...	1 КБ		
main.cpp	02.03.2023 19:03	Файл "CPP"	1 КБ		
officialWords.txt	09.03.2023 14:26	Текстовый докум...	1 КБ		
outputLexical.txt	08.02.2023 13:35	Текстовый докум...	1 КБ		
SemanticsAnalyzator.cpp	23.03.2023 16:14	Файл "CPP"	46 КБ		

7 Структура программы

Главная движущая функция - `int main()` (как и в языке C++). Можно добавлять функции и глобальные переменные, но только перед функцией `main()` в произвольном порядке. Например, можно написать вот так:

```
int main() {
    int a = 9;
    double b = 4;
    if (a <> 5 && b > 3) {
        out << 4;
    }
    return 0;
}
```

Или так:

```
int count(int a) {
    return a + 1;
}

int c = 5, v;
array int arr[10];

double Weight(bool is, int w) {
    if (is) {
        return w;
    }
}
```

```

    return w / 2;
}

int main() {
    int a = 9;
    double b = 4;
    if (a <> 5 && b > 3) {
        out << 4;
    }
    return 0;
}

```

На данном этапе достаточно знать, что сначала идут функции и глобальные переменные, а только потом - главная функция `int main()`.

8 Переменные и массивы

Переменные в языке бывают трёх типов: `int` (целочисленный), `double` (вещественный) и `bool` (логический). Переменные можно объявлять как в глобальной области (вне всяких функций, в том числе `main`), так и внутри функций. Стоит отметить, что их можно сразу инициализировать соответствующим значением. Имеют следующую структуру:

<тип> <имя> = <выражение>.

Массив может быть создан путём написания служебного слова **array**:

array <тип> <имя>[<число>], причём число должно быть **константным значением**!

Обращение к элементу массива:

<имя массива>[<выражение>].

9 Операторы

Как и в любом другом языке, в грамматике существуют следующие операторы:

1. Оператор ввода **in**:

in » <имя переменной или элемент массива> » ... ;

Разумеется, оператор ввода принимает только то, куда можно что-то записать (переменные или элемент массива), так что если вы случайно введёте выражение, то компилятор вас предупредит об ошибке. Осуществляется через стандартное окно ввода C++ (скорее всего, чёрное окошко на вашем компьютере).

2. Оператор вывода **out**:

out « <выражение или строка> « ... ;

В этом случае неважно, вводите ли вы переменную, элемент массива или сложно математическое выражение - программа засчитает всех их как элемент для вывода. Как и ввод, осуществляется через стандартное окно ввода C++.

3. Условный оператор **if (else)**:

if (<выражение>){<какие-то операции>}

else if (<выражение>) {<какие-то операции>}

...

else {<какие-то операции>}

Все выражение должны быть логического типа bool. В случае ошибки, компилятор выдаст ошибку несоответствия типов. Случаи с *else if* и *else* необязательны (подробнее см. "Теоретическое описание" в руководство программиста).

4. Оператор цикла **while**:

while (<выражение>) {<какие-то операции>}

Как и в условном операторе, необходимо, чтобы выражение было типа bool. Построен наподобие цикла *while* в C++.

5. Оператор цикла **for**:

for (<выражение или создание и объявление переменной>; <логическое выражение>; <выражение>) { <какие-то операции> }. Все выражения могут быть пустыми!

Как и цикл *while*, построен наподобие цикла *for* в C++;

6. Оператор возврата **return**:

return <выражение>

Выражение, стоящее после служебного слова, должно совпадать с типом функции, к которой этот оператор относится. Выражение может быть пустым!

10 Функции

Их грамматика устроена следующим образом:

```
<тип функции> <имя>(<тип> <имя> или array <тип> <имя>, ...) {  
<какие-то операторы>  
}
```

В этом случае тип функции может быть *void* - тип, который ничего не возвращает. Стоит учесть, что тут мы не будем писать *return* с выражением после него (либо не писать его вовсе). В функцию также можно передавать массивы.

11 Выражения

При записи выражений можно использовать переменные, элементы массива, числа, логические значения (*true* или *false*) и вызовы функций. Перечислены все операции с их приоритетами (взятие переменной/элемента массива считается операцией с приоритетом 0):

1	()	Круглые и квадратные скобки
2	!	Отрицание
3	+, -, ++, -	Унарный плюс и минус, инкременты
4	*, /, %	Умножить, разделить и взять остаток
5	+, -	Сложение и умножение
6	<, >, <=, >=	Больше, меньше, больше или равно, меньше или равно
7	==, <>	Равенство, неравенство
8	&	Битовая конъюнкция
9		Битовая дизъюнкция
10	^	Битовое "исключающее или"
11	&&	Логическая конъюнкция
12		Логическая дизъюнкция
13	=	Приравнивание

В языке всё строго: если необходим тип `int`, значит вы должны дать тип `int`, однако если требуется тип `double`, а выражение имеет тип `int`, то это возможно, компилятор преобразует целое значение в вещественное.

12 Возможные проблемы

1. Если не можете сохранить строку: в нашем языке нельзя сохранять строки, их можно только выводить.
2. У нас нет косвенной рекурсии, так как мы считаем, что это может усложнить процесс программирования для начинающих.

13 FAQ

1. Есть ли многомерные массивы?
Нет, наш язык учитывает только одномерные массивы с константным значением размера.
2. Можно ли создавать структуры и классы?
Нет, наш язык предназначен для изучения программирования с самых простых понятий.

Часть IV

Контакты

Балдина Анастасия: baldinaanastasia00@gmail.com

Севрюков Никита: sevryukovnikita@yandex.ru