

**Министерство науки и высшего образования Российской Федерации**  
**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ**  
**УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**  
**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**  
**ITMO University**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**  
**GRADUATION THESIS**

**Автоматизация построения программных компонентов при подготовке задач по  
спортивному программированию**

**Обучающийся / Student** Назаров Георгий Дмитриевич  
**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет информационных технологий и программирования  
**Группа/Group** М34371  
**Направление подготовки/ Subject area** 01.03.02 Прикладная математика и информатика  
**Образовательная программа / Educational program** Информатика и программирование 2019  
**Язык реализации ОП / Language of the educational program** Русский  
**Статус ОП / Status of educational program**  
**Квалификация/ Degree level** Бакалавр  
**Руководитель ВКР/ Thesis supervisor** Корнеев Георгий Александрович, кандидат технических наук, Университет ИТМО, факультет информационных технологий и программирования, доцент (квалификационная категория "ординарный доцент")  
**Консультант/ Consultant** Мирзаянов Михаил Расихович, ФИТиП, программист, неосн по совм.

Обучающийся/Student

Документ подписан	
Назаров Георгий Дмитриевич	
15.05.2023	

(эл. подпись/ signature)

Назаров  
Георгий  
Дмитриевич

(Фамилия И.О./ name  
and surname)

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Корнеев Георгий Александрович	
15.05.2023	

(эл. подпись/ signature)

Корнеев  
Георгий  
Александрович

(Фамилия И.О./ name  
and surname)

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University**

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /  
OBJECTIVES FOR A GRADUATION THESIS**

**Обучающийся / Student** Назаров Георгий Дмитриевич

**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет информационных технологий и программирования

**Группа/Group** М34371

**Направление подготовки/ Subject area** 01.03.02 Прикладная математика и информатика

**Образовательная программа / Educational program** Информатика и программирование 2019

**Язык реализации ОП / Language of the educational program** Русский

**Статус ОП / Status of educational program**

**Квалификация/ Degree level** Бакалавр

**Тема ВКР/ Thesis topic** Автоматизация построения программных компонентов при подготовке задач по спортивному программированию

**Руководитель ВКР/ Thesis supervisor** Корнеев Георгий Александрович, кандидат технических наук, Университет ИТМО, факультет информационных технологий и программирования, доцент (квалификационная категория "ординарный доцент")

**Консультант/ Consultant** Мирзаянов Михаил Расихович, ФИТиП, программист, неосн по совм.

**Основные вопросы, подлежащие разработке / Key issues to be analyzed**

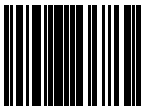
Требуется разработать систему, автоматизирующую процесс построения программных компонентов при подготовке (разработке) задач по спортивному программированию, в том числе:

1. Спроектировать язык описания входных и выходных данных, требуемых в задаче по спортивному программированию.
2. Реализовать систему кодогенерации, которая, на основании описания входных и выходных данных, будет генерировать исходный код программных компонентов задачи, в том числе:
  - 2.1. Программы, проверяющей ответ участника (чекера);
  - 2.2. Программы, проверяющей корректность входных данных (валидатора);
  - 2.3. Компонентов ввода-вывода пользовательских решений задачи (грейдеров).
3. Внедрить полученную систему в сервис подготовки задач по спортивному программированию «Polygon».

**Форма представления материалов ВКР / Format(s) of thesis materials:**

программный код, презентация, пояснительная записка

**Дата выдачи задания / Assignment issued on:** 10.04.2023**Срок представления готовой ВКР / Deadline for final edition of the thesis** 15.05.2023**Характеристика темы ВКР / Description of thesis subject (topic)****Тема в области прикладных исследований / Subject of applied research:** да / yes**СОГЛАСОВАНО / AGREED:**Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Корнеев Георгий Александрович	
21.04.2023	

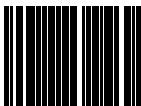
(эл. подпись)

Корнеев  
Георгий  
АлександровичЗадание принял к  
исполнению/ Objectives  
assumed BY

Документ подписан	
Назаров Георгий Дмитриевич	
21.04.2023	

(эл. подпись)

Назаров  
Георгий  
ДмитриевичРуководитель ОП/ Head  
of educational program

Документ подписан	
Станкевич Андрей Сергеевич	
22.05.2023	

(эл. подпись)

Станкевич  
Андрей  
Сергеевич

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University**

**АННОТАЦИЯ  
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ  
SUMMARY OF A GRADUATION THESIS**

**Обучающийся / Student** Назаров Георгий Дмитриевич  
**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет информационных технологий и программирования  
**Группа/Group** М34371  
**Направление подготовки/ Subject area** 01.03.02 Прикладная математика и информатика  
**Образовательная программа / Educational program** Информатика и программирование 2019  
**Язык реализации ОП / Language of the educational program** Русский  
**Статус ОП / Status of educational program**  
**Квалификация/ Degree level** Бакалавр  
**Тема ВКР/ Thesis topic** Автоматизация построения программных компонентов при подготовке задач по спортивному программированию  
**Руководитель ВКР/ Thesis supervisor** Корнеев Георгий Александрович, кандидат технических наук, Университет ИТМО, факультет информационных технологий и программирования, доцент (квалификационная категория "ординарный доцент")  
**Консультант/ Consultant** Мирзаянов Михаил Расихович, ФИТиП, программист, неосн по совм.

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ  
DESCRIPTION OF THE GRADUATION THESIS**

**Цель исследования / Research goal**

Разработка системы, автоматизирующей процесс построения программных компонентов при подготовке (разработке) задач по спортивному программированию.

**Задачи, решаемые в ВКР / Research tasks**

1. Проектирование языка описания входных и выходных данных, требуемых в задаче по спортивному программированию. 2. Реализация системы кодогенерации, которая, на основании описания входных и выходных данных, будет генерировать исходный код программных компонентов задачи, в том числе: 2.1. Программы, проверяющей ответ участника (чекера); 2.2. Программы, проверяющей корректность входных данных (валидатора); 2.3. Компонентов ввода-вывода пользовательских решений задачи (грейдеров). 3. Внедрение полученной системы в сервис подготовки задач по спортивному программированию «Polygon».

**Краткая характеристика полученных результатов / Short summary of results/findings**

Был спроектирован язык описания входных и выходных данных для задач по спортивному программированию. Реализована система кодогенерации, которая, на основании описания входных и выходных данных частично генерирует исходный код чекера, валидатора и

грейдеров. Полученная система внедрена в сервис подготовки задач по спортивному программированию «Polygon».

Обучающийся/Student

Документ подписан	
Назаров Георгий Дмитриевич	
15.05.2023	

(эл. подпись/ signature)

Назаров  
Георгий  
Дмитриевич

(Фамилия И.О./ name  
and surname)

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Корнеев Георгий Александрович	
15.05.2023	

(эл. подпись/ signature)

Корнеев  
Георгий  
Александрович

(Фамилия И.О./ name  
and surname)

## СОДЕРЖАНИЕ

Введение .....	6
1. Подготовка и проведение соревнований по спортивному программированию .....	8
1.1. Соревнования с точки зрения участников .....	8
1.2. Процесс тестирования .....	8
1.3. Подготовка задач .....	9
1.4. Примеры программных компонентов задачи .....	10
1.5. Цели и задачи ВКР .....	13
Выводы по главе 1 .....	14
2. Проектирование языка описания входных и выходных данных .....	15
2.1. Требования к языку описания .....	15
2.2. Концептуальное описание языка .....	15
2.3. Подязык выражений .....	18
2.4. Языковые конструкции .....	21
2.5. Система типов .....	23
Выводы по главе 2 .....	24
3. Разбор и семантика языка описания входных и выходных данных .....	25
3.1. Обработка языка описания .....	25
3.2. Области видимости .....	26
3.3. Реализация системы типов .....	27
3.4. Именованные сущности и символы .....	29
3.5. Конструкторы .....	30
3.6. Описания переменных .....	31
3.7. Язык записи выражений .....	33
3.8. Внутреннее представление .....	35
Выводы по главе 3 .....	37
4. Кодогенерация и интеграция в систему подготовки задач «Polygon» ...	38
4.1. Кодогенерация .....	38
4.1.1. Генерация объявлений структур .....	39
4.1.2. Генерация выражений .....	40
4.1.3. Генерация кода ввода .....	41
4.1.4. Особенности генерации валидатора и чекера .....	42
4.1.5. Генерация кода вывода .....	43

4.1.6. Шаблоны генерации кода.....	45
4.2. Интеграция в «Polygon» .....	46
4.2.1. Структура запросов и ответов .....	46
4.2.2. Поддержка в «Polygon».....	47
4.2.3. Сервис кодогенерации.....	48
4.3. Тестирование.....	48
Выводы по главе 4 .....	49
Заключение.....	50
Список использованных источников.....	51
Приложение А. Условие задачи-примера «Копия копии копии» .....	54
Приложение Б. Грамматика языка описания .....	56
Приложение В. Язык записи выражений .....	60
Приложение Г. Структура запросов и ответов сервиса генерации.....	61
Приложение Д. Шаблоны генерируемого кода .....	62
Приложение Е. Примеры генерируемого кода.....	69

## ВВЕДЕНИЕ

Соревнования по спортивному программированию проходят с использованием автоматизированных систем тестирования [18], например, PCMS2 [3], Ejudge [7], Яндекс.Контест [8], Codeforces [1]. Участники отсылают на проверку исходный код программы-решения, которая автоматически проверяется на наборе тестов [22].

Для того, чтобы задачу по спортивному программированию можно было использовать в автоматизированной тестирующей системе, авторы задачи должны предварительно подготовить «пакет» задачи, состоящий из условий задачи, тестов и набора программ, участвующих в процессе тестирования [26]. Подготовка каждой задачи — трудоемкий ручной процесс, легко подверженный ошибкам.

Целью данной выпускной квалификационной работы является разработка системы, частично автоматизирующей процесс реализации программных компонентов при подготовке задач по спортивному программированию.

В работе спроектирован язык описания входных и выходных данных для задач по спортивному программированию, позволяющий описывать структуру входных данных и ожидаемых ответов, а также указывать ограничения на переменные во входных и выходных данных. Реализована система частичной генерации некоторых программных компонентов задачи. Полученная система внедрена в сервис для подготовки задач по спортивному программированию «Polygon» [27]. На работу получен акт о внедрении.

Первая глава работы подробно описывает предметную область — описывается процесс подготовки соревнований по спортивному программированию и процесс их проведения. Первая глава содержит описания компонентов задачи, вводятся термины, которые будут использованы в последующих главах.

Во второй главе формулируются и требования к языку описания входных и выходных данных, приводится концептуальное описание спроектированного языка. Вторая глава содержит наглядное описание языковых конструкций на примере задачи с одного из соревнований на платформе Codeforces.

В третьей главе подробно изложены детали реализации разбора языка, более подробно описывается семантика языковых конструкций.



Четвертая глава содержит детали реализации системы генерации исходного кода программных компонентов задачи по описанию входных и выходных данных, описан способ интеграции полученной системы в сервис подготовки задач по спортивному программированию «Polygon», а также приводится описание процесса тестирования полученной системы.

## **ГЛАВА 1. ПОДГОТОВКА И ПРОВЕДЕНИЕ СОРЕВНОВАНИЙ ПО СПОРТИВНОМУ ПРОГРАММИРОВАНИЮ**

В данной главе приводятся описания существующего процесса подготовки задач по спортивному программированию и процесса участия в соревнованиях по спортивному программированию с точки зрения участника. В дальнейшей работе будут использоваться термины и определения, вводимые в этой главе.

### **1.1. Соревнования с точки зрения участников**

На соревнованиях по спортивному программированию участникам предлагается к решению набор задач. За ограниченное количество времени участники должны решить как можно больше задач и как можно точнее (определение «точности» зависит от правил конкретного соревнования).

К каждой задаче прилагается условие задачи — текстовое человекочитаемое описание требований к решению участника. Решением участника является исходный код на одном из доступных в автоматизированной системе тестирования языков программирования. Решение задачи может содержать, в зависимости от правил соревнования, как весь исходный код программы-решения, так и его часть — реализацию заданного в условии задачи интерфейса или функции с необходимой сигнатурой.

После проверки решения, участнику сообщается вердикт проверки — строка с информацией о корректности решения, возможно, содержащая дополнительную информацию, например номер теста, на котором решение участника выдало неправильный ответ [4], [24].

### **1.2. Процесс тестирования**

После получения решения участника автоматизированная тестирующая система выполняет его обработку, состоящую из следующих основных шагов [6].

- а) Компиляция решения участника.
- б) Компоновка (линковка) решения участника с компонентами задачи (при необходимости).
- в) Генерация наборов входных данных (для генерируемых тестов).
- г) Валидация входных данных на соответствие условию задачи.

- д) Запуск исполняемого файла с решением на наборе тестов (входных данных).
- е) Запуск проверяющей программы на каждом из тестов и ответов программы-решения.

В зависимости от правил соревнования шагов может быть больше. Об ошибке на любом из шагов сообщается участнику.

*Тестами (входными данными)* являются файлы с определенной автором задачи структурой. В условии задачи описываются переменные и ограничения на них, а также описывается порядок их появления во входных данных.

В ответ на каждый тест программа-решение должна сгенерировать файл, называемый *выходными данными*. Выходные данные так же состоят из переменных, описанных в условии задачи.

### 1.3. Подготовка задач

Чтобы задачу можно было использовать в автоматизированной тестирующей системе, авторы задачи должны подготовить следующие компоненты задачи [26].

- а) Условия задачи, возможно, на нескольких языках.
- б) Валидатор (от англ. *validator*) — программу, принимающую на вход произвольный текстовый файл и выдающую вердикт, может ли быть этот файл использован в качестве теста в конкретной задаче. Валидатор должен проверять структуру файла и соответствие значений переменных ограничениям из условия.
- в) Чекер (от англ. *checker*, альтернативное название — проверяющая программа) — программу, принимающую на вход тест, ответ программы участника и ответ жюри (авторского решения), и выдающая вердикт о корректности ответа участника. Чекер должен проверять выполнение в ответе ограничений из условия задачи.
- г) Тесты (входные данные) — файлы, которые будут подаваться на вход решению участника. Могут быть заданы вручную или генерироваться во время тестирования.
- д) Генераторы — программы, принимающие в аргументах командной строки параметры, на основании которых генерируют (выводят) тест или набор тестов.

- е) Решение жюри (авторское решение) — исходный код эталонного решения задачи. Используется для получения эталонных ответов при запуске чекера.

На некоторых соревнованиях от участников не требуется полная реализация решения, а только лишь его части — функции с заданной сигнатурой или интерфейса. Подобные правила применяются, например, на IOI — международной олимпиаде по информатике [25]. В таком случае авторам задачи так же необходимо подготовить грейдеры (от англ. *grader*, «оценщик») — часть исходного кода решения, которая будет скомпонована (слинкована) с решением участника. Исходный код грейдеров должен содержать точку входа программы и реализацию ввода-вывода, специфичную для задачи. Дополнительно, грейдеры могут выполнять дополнительные действия перед передачей управления коду участника или после окончания исполнения кода участника. Грейдеры должны быть написаны на каждом из доступных участнику языков программирования [33].

#### 1.4. Примеры программных компонентов задачи

Рассмотрим подробнее программные компоненты на примере задачи «A + B».

##### Условие задачи

Дано два целых числа:  $a$  и  $b$ . Требуется вывести их сумму.

В единственной строке входных данных задано два целых числа  $a$  и  $b$  ( $-1000 \leq a, b \leq 1000$ ), разделенных пробелом.

В единственной строке выходных данных требуется вывести значение  $a + b$ .

Листинг 1 содержит исходный код валидатора на языке программирования C++ [20] с применением библиотеки `testlib.h` [5]. В первой строке функции `main()` вызывается функция инициализации библиотеки `testlib.h` в режиме валидатора. Затем считывается два целочисленных значения в интервале от  $-1000$  до  $1000$ , разделенных одним пробелом. Затем ожидается перевод строки и конец файла теста. Для считывания данных используется объект `inf` из библиотеки `testlib.h`, отвечающий за поток ввода данных.

## Листинг 1 – Пример валидатора

```
#include "testlib.h"

int main(int argc, char *argv[])
{
    registerValidation(argc, argv);
    inf.readInt(-1000, 1000, "a");
    inf.readSpace();
    inf.readInt(-1000, 1000, "b");
    inf.readEoln();
    inf.readEof();
    return 0;
}
```

Листинг 2 содержит исходный код чекера. В первой строке функции `main()` вызывается функция инициализации библиотеки `testlib.h` в режиме чекера. Затем считывается одно целочисленное значение в интервале от  $-2000$  до  $2000$  два раза: сначала ответ участника, затем ответ жюри (вывод авторского решения). Ответ участника считывается при помощи объекта `ouf`, ответ жюри — при помощи объекта `ans`. Если ответы совпали, участник получает вердикт «правильный ответ», в противном случае «неправильный ответ», на соответствующем тесте. Для выдачи вердикта используется функция `quitf()` из библиотеки `testlib.h`, осуществляющая форматированный вывод вердикта и завершающая исполнение чекера с кодом возврата, соответствующем вердикту. В случае правильного ответа код возврата равен `0`, в случае неправильного — `1`.

Тесты могут быть сгенерированы программно. Листинг 3 содержит исходный код генератора случайных тестов, который принимает на вход единственный аргумент командной строки — ограничение на абсолютную величину генерируемых значений. В первой строке функции `main()` вызывается функция инициализации библиотеки `testlib.h` в режиме генератора. Затем происходит разбор аргументов командной строки при помощи функции `opt<>()` из библиотеки `testlib.h`. Затем два случайных значения выводятся в стандартный поток вывода через пробел, затем следует перевод строки. Для генерации случайных значений используется объект `rnd` из библиотеки `testlib.h`.

## Листинг 2 – Пример чекера

```
#include "testlib.h"

int readAns(InStream& stream)
{
    return stream.readInt(-2000, 2000, "sum");
}

int main(int argc, char *argv[])
{
    registerTestlibCmd(argc, argv);
    int participant = readAns(ouf);
    int jury = readAns(ans);
    if (participant != jury)
        quitf(_wa, "Wrong answer");
    else
        quitf(_ok, "Ok");
}
```

## Листинг 3 – Пример генератора

```
#include <iostream>
#include "testlib.h"

int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);

    int N = opt<int>(1);
    std::cout << rnd.next(-N, N) << ' ';
    std::cout << rnd.next(-N, N) << std::endl;
    return 0;
}
```

Грейдеры являются написанной жюри (автором задачи) частью исходного кода, который будет скомпонован с реализацией участника для получения полного исполняемого файла с решением.

Листинг 4 содержит исходный код грейдера для языка программирования C++. Грейдеры для C++ обычно состоят из двух файлов: заголовочного и основного. В заголовочном файле определены вспомогательные функции, предоставляемые участнику, а также сигнатуры функций, которые участник должен реализовать. Основной файл содержит точку входа (функцию `main()`, с которой будет начато исполнение) и код, считывающий входные данные из стандартного потока ввода и выводящий ответ, полученный вызовом пользовательской реализации, в стандартный поток вывода.

#### Листинг 4 – Пример грейдера для языка C++

```
// aplusb.h

int sum_ab(int a, int b);

// grader.cpp

#include <iostream>
#include "apusb.h"

int main() {
    int a, b;
    std::cin >> a >> b;
    std::cout << sum_ab(a, b) << std::endl;
    return 0;
}
```

Листинг 5 содержит исходный код грейдера для языка программирования Python [30]. Так как Python — интерпретируемый язык программирования, понятие линковки для него не определено, поэтому грейдер напрямую импортирует и вызывает решение участника. Блок кода в середине содержит проверку версии языка, чтобы грейдер мог работать как на Python 2 [29], так и на Python 3 [30].

#### Листинг 5 – Пример грейдера для языка Python

```
import solution
import sys

if sys.version_info[0] < 3:
    _input = raw_input
else:
    _input = input

a, b = map(int, _input().split())
print(solution.sum_ab(a, b))
```

### 1.5. Цели и задачи ВКР

В предыдущих пунктах показано, что для подготовки даже простых задач по спортивному программированию авторам задачи необходимо проделать существенную работу, написав исходный код для, как минимум, трёх программных компонентов — валидатора, чекера и генератора, при этом ге-

нераторов может быть много, для того чтобы сгенерировать как можно более разнообразные тесты. При этом, в случае если задача подразумевает наличие грейдеров, суммарный объем их исходного кода пропорционален числу поддерживаемых тестирующей системой языков программирования.

Основная цель данной выпускной квалификационной работы — разработка системы, автоматизирующей процесс реализации программных компонентов при подготовке задач по спортивному программированию.

В результате анализа существующего процесса подготовки задач соревнований были выделены следующие задачи, решаемые в данной выпускной квалификационной работе.

- а) Необходимо спроектировать специальный язык описания входных и выходных данных, специфичных для задачи.
- б) Реализовать систему кодогенерации, которая будет принимать на вход описание входных и выходных данных, на основании которой будет частично сгенерирован исходный код валидатора, чекера и грейдеров.
- в) Внедрить реализованную систему в сервис подготовки задач по спортивному программированию «Polygon».

### **Выводы по главе 1**

В этой главе был описан процесс подготовки и проведения соревнований по спортивному программированию, приведен перечень программных компонентов задач. Было показано, что данная выпускная квалификационная работа актуальна, так как существующий процесс подготовки задач требует от авторов задач высокого уровня внимательности. В конце главы были сформулированы цели и задачи выпускной квалификационной работы.



## ГЛАВА 2. ПРОЕКТИРОВАНИЕ ЯЗЫКА ОПИСАНИЯ ВХОДНЫХ И ВЫХОДНЫХ ДАННЫХ

В этой главе формулируются требования к языку описания входных и выходных данных, и описывается разработанный язык описания входных и выходных данных.

### 2.1. Требования к языку описания

Были выделены следующие требования к возможностям языка описания.

- а) Должна быть возможность указывать название для каждой переменной. Это необходимо для генерации человекочитаемых сообщений об ошибках в валидаторе и чекере.
- б) Должна быть возможность указывать тип переменных, для корректной реализации ввода-вывода.
- в) Должна быть возможность описывать структуры — упорядоченный набор семантически связанных переменных.
- г) Должна быть возможность указывать места перевода строк.
- д) Должна быть возможность задавать ограничения на переменные — для проверки корректности данных.

Отдельно стоит отметить одно неформальное требование: язык описания должен иметь «низкий порог входа». Иными словами, «простые» входные и выходные данные должны с его помощью описываться «простыми» конструкциями.

Так же для удобства последующей реализации было принято решение ограничить возможный язык описания классом  $LL(1)$  [17].

### 2.2. Концептуальное описание языка

Входные и выходные данные представляют собой текстовые файлы, содержание которых удовлетворяет описанным в условии задачи ограничениям и формату. Разработанный язык описания представляет способ текстового машиночитаемого описания структуры входных и выходных данных, понятного человеку.

В этом разделе мы рассмотрим лексику и синтаксис языка описания входных и выходных данных на примере задачи «Копия копии копии» с соревнования «Codeforces Round 839 (Div. 3)» [2]. Условие задачи приведено в

Приложении А. Листинг 6 содержит описание входных и выходных данных для этой задачи.

Листинг 6 – Пример описания ввода-вывода

# <https://codeforces.com/contest/1772/problem/F>

```

picture (n: int32, m: int32) {
    s: string[i=1..n, sep='\n'] | len(s[i]) == m;
}

input {
    n: int32 | 3 <= n && n <= 30;
    m: int32 | 3 <= m && m <= 30;
    k: int32 | 0 <= k && k <= 100;
    eoln;
    eoln;

    pictures: picture(n, m)[i=1..k + 1, sep="\n\n"];
}

output {
    startPicIndex: int32 | 1 <= startPicIndex
                        && startPicIndex <= input.k + 1;
    eoln;
    q: int32 | 0 <= q && q <= 10 ** 7;
    eoln;
    ops: array[j=1..q, sep='\n'] of {
        op: int32;
        if (op == 1) {
            x: int32 | 2 <= x && x <= input.n - 1;
            y: int32 | 2 <= y && y <= input.m - 1;
        } else {
            i: int32 | 1 <= i && i <= input.k + 1;
        }
    };
}

```

Файл описания входных и выходных данных состоит из набора *структур* — объявлений семантически объединенных переменных, идущих во входных или выходных данных в указанном в описании порядке. Структуры можно рассматривать как функции, тело которых состоит из описания того, какие переменные нужно считывать или выводить и где ожидаются переводы строк. В примере можно видеть три структуры: `picture`, `input` и `output`. При этом, наличие структур `input` и `output` обязательно, так как они описывают непосредственно входные и выходные данные. Переменные внутри структуры могут представлять собой значения, массивы или другие структуры.

Любая структура, за исключением `input` и `output` может иметь *параметры*, семантически похожие на аргументы функции. В примере у структуры `picture` есть два параметра: `n` и `m` — размеры картинки.

Структуры могут содержать переменные, условные операторы и модификаторы ввода-вывода. В примере структура `input` содержит четыре переменные: `n`, `m`, `k` и `pictures`.

Объявление переменной содержит название соответствующей переменной, её тип и, возможно, ограничение на эту переменную, представленное логическим выражением. Рассмотрим объявление переменной `x` из листинга 7. Сначала записано «`x`» — название переменной, затем следует «`int32`» — её тип, и после следует «`2 <= x && x <= input.n - 1`» — ограничение на переменную.

Модификаторы ввода-вывода используются для изменения разделителей между переменными и указания необходимого форматирования во входных и выходных данных. В примере в структуре `input` есть два вхождения модификатора `eoln`. Такая запись означает, что переменные `n`, `m` и `k` находятся на одной строке во вводе, затем следует два перевода строки и затем переменная `pictures`. На данный момент поддерживается только модификатор `eoln`, означающий перевод строки.

Условные операторы используются для изменения набора переменных в структуре в зависимости от какого-либо условия. Листинг 7 содержит пример условного оператора. Когда считывание входных данных или вывод выходных данных доходит до условного оператора, вычисляется условие, в данном случае «`op == 1`». Если оно истинно, далее будут считаны (или выведены) переменные из первого (положительного) блока, в данном случае `x` и `y`. Если же условие ложно, то будут считаны переменные из второго (отрицательного) блока, в данном случае `i`. Поддерживается неполная форма записи `if`, не содержащая второй блок.

#### Листинг 7 – Пример условного оператора

```
if (op == 1) {
    x: int32 | 2 <= x && x <= input.n - 1;
    y: int32 | 2 <= y && y <= input.m - 1;
} else {
    i: int32 | 1 <= i && i <= input.k + 1;
}
```

Для построения лексического и синтаксического анализаторов языка описания был использован генератор нисходящих анализаторов для формальных языков ANTLR 4 [11]. Приложение Б содержит полное описание грамматики языка в формате ANTLR 4. Листинг Б.1 содержит определения типов токенов, листинг Б.2 содержит грамматику языка описания ввода-вывода, листинг Б.3 содержит грамматику подязыка для записи выражений. Далее в этой главе будут рассматриваться различные поддерживаемые языковые конструкции, поэтому, в тексте этой главы выражения «тип токена X» и «нетерминал X» следует рассматривать как ссылки на соответствующие определения из приложения Б.

Типы токенов `LINE_COMMENT`, `COMMENT` и `WS` отвечают за комментарии и пробельные символы. Такие токены должны впоследствии игнорироваться синтаксическим анализатором, что указано директивой «-> skip» после них. Однострочные комментарии начинаются с символа «#» и продолжаются до конца строки. Блочные комментарии обрамляются последовательностями символов «/\*» и «\*/». Листинг 8 содержит примеры комментариев.

#### Листинг 8 – Примеры комментариев

# Пример однострочного комментария (`LINE_COMMENT`)

```
/*
    Пример
    многострочного
    комментария
    (COMMENT)
*/
```

Тип токена `NAME` отвечает за идентификаторы. Листинг 7 содержит примеры идентификаторов: «or», «x», «y», «n», «m», «k» и «input». Для упрощения последующей кодогенерации, идентификаторы могут содержать только буквы английского алфавита, и символы «\_», «0» – «9».

### 2.3. Подязык выражений

В языке описания входных и выходных данных можно выделить подязык для записи выражений. Выражения можно использовать при записи ограничений на переменные, как показано в листинге 7, или при указании длин массивов. Листинг 9 содержит примеры выражений. Список всех поддерживаемых в выражениях операций содержится в приложении В в таблице В.1.

Язык записи выражений сделан максимально близким по виду и семантике к языку C++, чтобы облегчить его использование авторам задач, так как для соревнований на платформе Codeforces программные компоненты задач разрабатываются на языке программирования C++ с применением библиотеки `testlib.h`. Приоритеты операторов так же идентичны таковым из языка программирования C++, соответствующая таблица приведена в приложении В.

Листинг Б.3 содержит полную грамматику подязыка выражений в формате ANTLR 4. Стартовым нетерминалом является `plExpression`.

### Листинг 9 – Примеры выражений

```
# Примеры констант
'a' # Одиночный символ - токен CHAR
"abacaba" # Строка - токен STRING
true # Истина - токен TRUE
false # Ложь - токен FALSE
123 # Целое число - токен NUM_VALUE

# Составные выражения
2 <= x && x <= input.n - 1

0 <= q && q <= 10u ** 7

(op == 1 || op == 2) ? "1 or 2" : "false"
```

Типы токенов `CHAR` и `STRING` отвечают за символьные и строковые константы соответственно. Одиночные символы обрамляются машинописными апострофами («'»), строки обрамляются кавычками («"»).

Типы токенов `TRUE` («true») и `FALSE` («false») отвечают за логические константы «истина» и «ложь» соответственно.

Токены типа `NUM_VALUE` могут содержать целочисленные значения или числа с плавающей точкой (в том числе в экспоненциальной форме записи). Используя суффиксы, можно явно указать тип, в который будет транслироваться токен при кодогенерации.

- а) Целое значение без суффикса (например, «25») будет иметь тип `int32`.
- б) Целое значение с суффиксом «U» или «u» (например, «10u») будет иметь тип `uint32`.
- в) Целое значение с суффиксом «L» или «l» (например, «123L») будет иметь тип `int64`.

г) Целое значение с суффиксом «UL» или «ul» (например, «1234ul») будет иметь тип `uint64`.

Значение считается числом с плавающей точкой, если оно содержит точку, экспоненту или суффикс «f». Примерами записи чисел с плавающей точкой являются, «1.0», «1e9», «2f» и «1e2f». При наличии суффикса «f» считается, что значение имеет тип `float32`, иначе — `float64`.

Нетерминал `plValue` отвечает за запись константных значений, он раскрывается в один из токенов, описанных выше.

Типы токенов `LOGICAL_AND` («&&»), `LOGICAL_OR` («||») и `LOGICAL_NOT` («!») отвечают за операторы логической конъюнкции, дизъюнкции и отрицания.

За операции «Побитовое И», «Побитовое исключающее ИЛИ», «Побитовое ИЛИ» и «Побитовое отрицание» отвечают типы токенов `BITWISE_AND` («&»), `BITWISE_XOR` («^»), `PIPE` («|») и `BITWISE_NOT` («~») соответственно.

Типы токенов `EQUALS` («==»), `NOT_EQUALS` («!=»), `LESS_EQUAL` («<=»), `GREATER_EQUAL` («>=»), `LESS` («<») и `GREATER` («>») отвечают за операторы сравнения: равенство, неравенство, «меньше или равно», «больше или равно», «меньше» и «больше» соответственно.

Далее следует блок типов токенов, отвечающих за арифметические операции: `PLUS` («+»), `MINUS` («-»), `POW` («\*\*»), `MULTIPLICATION` («\*»), `DIVISION` («/»), `MODULO` («%») — сложение, вычитание, возведение в степень, умножение, деление и взятие остатка от деления соответственно.

Последняя строка листинга 9 содержит пример использования тернарного оператора, к которому относятся типы токенов `QUESTION` («?») и `COLON` («:»). Приложение В содержит таблицу приоритетов операторов.

Нетерминал `plVarBinding`, раскрывающийся в токен типа `NAME`, отвечает за переменные в выражениях. Операторы языка выражений и остальные нетерминалы представлены в таблице В.1. Стартовым нетерминалом грамматики подязыка выражений является `plExpression`. Так же в выражениях можно использовать круглые скобки для изменения приоритета выполнения операций, вызовы встроенных функций (например, функции `len` для массивов).

## 2.4. Языковые конструкции

Рассмотрим языковые конструкции языка описания ввода-вывода.

Определение каждой структуры начинается с токена с типом `NAME` — названия соответствующей структуры. Названия структур должны быть уникальными и не совпадать с названиями встроенных типов. Затем следует необязательные параметры структуры (нетерминал `namedStructParameters`) и после них описание структуры (нетерминал `struct`).

Параметры структуры состоят из списка объявлений параметров (нетерминалы `parameterDeclaration`), разделенных запятыми, обрамленного круглыми скобками (типы токенов `LPAR` и `RPAR`). Каждое объявление параметра состоит из имени параметра (`NAME`) и названия типа этого параметра, разделенных символом двоеточия.

Листинг 10 содержит пример объявления структуры. В этом примере «`picture`» является названием структуры, «`(n: int32, m: int32)`» — списком параметров структуры. Тело структуры находится между открывающей и закрывающей фигурными скобками (типы токенов `LBRACE` и `RBRACE`).

Листинг 10 – Пример простой структуры

```
picture (n: int32, m: int32) {
    s: string[i=1..n, sep='\n'] | len(s[i]) == m;
}
```

Тело структуры состоит из списка элементов описания (нетерминалы `structItem`), обрамленного фигурными скобками. Элементами описания могут быть условные операторы (нетерминал `conditionalAlternative`), объявления переменных (нетерминал `variableDeclaration`) или модификаторы ввода-вывода (нетерминал `ioModifier`). Листинг 11 содержит пример структуры, содержащий все возможные элементы. Строки «`startPicIndex: int32;`», «`q: int32;`» и другие подобные являются объявлениями переменных, строки «`eoln;`» — модификаторами ввода-вывода. Так же на примере показано использование условного оператора «`if (...) {...} else {...}`».

Объявления переменных состоят из названия переменной, затем следует символ двоеточия (токен `COLON`), после него пишется тип переменной

## Листинг 11 – Пример сложной структуры

```

output {
  startPicIndex: int32;
  eoln;
  q: int32;
  eoln;
  ops: array[j=1..q, sep='\n'] of {
    op: int32;
    if (op == 1) {
      x: int32;
      y: int32;
    } else {
      i: int32;
    }
  };
}

```

(нетерминал `variableType`). Затем, опционально, следует символ вертикальной черты (тип токена `PIPE`) и выражение, содержащее ограничение на переменную. Завершается описание переменной символом точки с запятой (тип токена `SEMICOLON`).

Тип переменной задается следующим образом. Сначала записывается название типа (предопределенное или название объявленной автором задачи структуры), затем, опционально, следуют параметры структуры (нетерминал `namedTypeParameters`), состоящие из списка выражений, после чего, опционально, следуют одно или несколько объявлений параметров массива (нетерминалы `arrayParameters`).

Листинг 12 содержит примеры записи типов для переменных. Первый пример — объявление переменной встроенного типа `int32`. Второй пример — объявление переменной, содержащей структуру. Можно заметить, что первый и второй примеры отличаются только названием типа.

Третий пример показывает объявление переменной, содержащей структуру с параметрами. Здесь «`picture`» — название структуры, «`n * 2`» и «`m + 1`» — выражения аргументов, значения которых будут переданы в структуру.

Рассмотрим четвертый и пятый примеры, в которых объявляются массивы. Чтобы объявить массив, после типа элементов массива записываются *параметры массива*. Параметры массива состоят из объявления переменной итерирования с указанием начального и конечного значений итерирования (нетерминал `arrayIterationRange`). Затем, через запятую, указывается раздели-



тель переменных в массиве — строка, состоящая из пробелов или экранированных переводов строк (нетерминал `sepValue`).

В примерах переменная итерирования — «*i*». В четвертом примере индексы массива пробегают значения от 1 до  $k + 1$  и между соседними элементами массива ожидается два перевода строки. В пятом примере индексы массива пробегают значения от 1 до  $n$  и между соседними элементами массива ожидается символ пробела.

Листинг 12 – Примеры записи типов переменных

```
q: int32; # Переменная типа int32

x: somestruct; # Структура без параметров

pic: picture(n * 2, m + 1); # Структура с параметрами

# Массив структур с параметрами
pictures: picture(n, m)[i=1..k + 1, sep="\n\n"];

s: string[i=1..n, sep=' ']; # Массив строк
```

Поддержана возможность определения массивов неименованных структур. Примером такого массива является переменная «*ops*» из листинга 11. Чтобы объявить массив неименованной структуры, в качестве типа переменной указывается конструкция «*array ... of*» (нетерминал `arrayOfUnnamedStruct`), после которой следует описание структуры. Вместо троеточия пишутся параметры массива.

## 2.5. Система типов

Язык описания входных и выходных данных имеет встроенные целочисленные, строковые, символьные, логические типы и тип вещественных чисел с плавающей точкой. Полный список встроенных типов приведен в таблице 1.

Также поддерживаются объявленные пользователем составные типы (структуры), состоящие из упорядоченного набора переменных других, возможно составных, типов.

К любому типу переменной при объявлении может быть дописан суффикс, являющийся параметрами массива. Такая конструкция объявляет массив указанного типа.

Язык описания ввода-вывода имеет статическую типизацию, то есть типы переменных задаются до этапа исполнения и не могут меняться. При этом,

Таблица 1 – Встроенные типы данных

Тип	Описание	Пример
<code>bool</code>	Логический тип	<code>true</code>
<code>char</code>	Символьный тип	<code>'x'</code>
<code>int32</code>	32-х битный целочисленный знаковый тип	<code>123</code>
<code>uint32</code>	32-х битный целочисленный беззнаковый тип	<code>123u</code>
<code>int64</code>	64-х битный целочисленный знаковый тип	<code>123l</code>
<code>uint64</code>	64-х битный целочисленный беззнаковый тип	<code>123ul</code>
<code>float32</code>	32-х битный тип чисел с плавающей точкой	<code>12.3f</code>
<code>float64</code>	64-х битный тип чисел с плавающей точкой	<code>12.3</code>
<code>string</code>	Строковый тип (массив символов)	<code>"abc"</code>

в выражениях допускается приведение типов для чисел, следующее правилам из языка C++ — от меньшего типа к большему.

### Выводы по главе 2

В этой главе были сформулированы требования к языку описания входных и выходных данных для задач по спортивному программированию. Было приведено концептуальное описание спроектированного языка, описаны языковые конструкции.

## **ГЛАВА 3. РАЗБОР И СЕМАНТИКА ЯЗЫКА ОПИСАНИЯ ВХОДНЫХ И ВЫХОДНЫХ ДАННЫХ**

В этой главе описываются алгоритмы обработки языка описания входных и выходных данных, а также реализация соответствующих алгоритмов. В ходе этой главы будут упоминаться классы из исходного кода реализации данной выпускной квалификационной работы.

### **3.1. Обработка языка описания**

Был реализован инструмент, принимающий на вход описание входных и выходных данных на описанном выше языке. Обработка происходит в несколько этапов.

- а) Лексический разбор.
- б) Синтаксический разбор.
- в) Поиск определений структур.
- г) Разрешение имен и типов, получение абстрактных синтаксических деревьев выражений и структур.
- д) Генерация кода.

Лексический разбор — процесс перевода текстового файла, поданного на вход, в список токенов. Синтаксический разбор — процесс перевода списка токенов в дерево разбора. Во время синтаксического разбора так же происходит проверка синтаксической корректности последовательности токенов. Эти два этапа выполняются при помощи сгенерированного ANTLR 4 из грамматики, приведенной в приложении Б, анализатора [11].

На следующем этапе выполняется поиск и регистрация имен структур данных, определенных пользователем. Это необходимо для того, чтобы впоследствии, при получении абстрактных синтаксических деревьев структур, знать полный набор именованных типов, существующих в описании. Этот этап выполняется частичным обходом дерева разбора.

Затем выполняется основной этап обработки, включающий в себя получение внутреннего представления, пригодного для последующей генерации кода. Внутреннее представление является набором абстрактных синтаксических деревьев для выражений и структур с дополнительной информацией в вершинах. Так же на этом этапе происходит разрешение имен переменных и типов. Этот этап выполняется обходом дерева разбора. Во время обхода проверяется семантическая корректность, существование используемых типов (в

том числе структур), существование переменных, ссылки на которых используются, количество переданных в конструкторы аргументов. Во время обхода поддерживается информация о текущей области видимости переменных. Для упрощения реализации, выражения (длины массивов и ограничения) обходятся отдельно.

Последним этапом является непосредственно генерация целевого кода чекера, валидатора и грейдеров. Для каждого генерируемого программного компонента задачи поочередно выполняется генерация определений структур, объявленных автором в файле описания входных и выходных данных, генерация кода ввода и вывода.

### 3.2. Области видимости

После выполнения лексического и синтаксического разборов получается абстрактное синтаксическое дерево, содержащее описание ввода-вывода в иерархическом виде. Для дальнейшей работы необходимо знать полный список имен типов, доступных для использования. Помимо стандартных типов это так же могут быть названия пользовательских структур. Полный список известных имен типов необходим для проверки описания ввода-вывода на корректность перед генерацией кода.

Как и в языках программирования [17], в языке разметки входных и выходных данных предусмотрены области видимости, в том числе вложенные. Каждая пользовательская структура является отдельной областью видимости.

Чтобы значение переменной можно было использовать внутри другой структуры, её необходимо передать в качестве параметра. Листинг 13 содержит пример такой передачи.

Листинг 13 – Пример передачи аргумента

```
pair(m: int32) {
    x: int32 | x <= m;
    y: int32 | y <= m;
}

input {
    n: int32;
    m: int32;
    eoln;
    p: pair(m);
}
```

Реализация областей видимости представлена в решении классом `Scope`. Он поддерживает информацию о всех именованных сущностях, которые могут быть использованы. Всего их четыре.

- а) Именованные типы — встроенные типы данных языка описания ввода-вывода и объявленные пользователем структуры.
- б) Конструкторы — объекты, связанные со структурами и инкапсулирующие в себе информацию о полях, параметрах структуры и модификаторах ввода-вывода в структуре.
- в) Переменные — поля структур.
- г) Функции — доступные для вызова встроенные функции (на данный момент присутствует только функция взятия длины массива или строки).

Существует корневая область видимости, содержащая встроенные типы и функции, а также ссылки на объявленные пользователем структуры и их конструкторы. Каждая область видимости, кроме корневой, содержит ссылку на родительскую. Таким образом, для каждой области видимости по цепочке можно восстановить полное множество известных имен и их типы.

### 3.3. Реализация системы типов

Все типы представлены реализациями интерфейса `Type`. У каждого типа есть набор характеристик, которые используются для различных проверок корректности использования каждого конкретного типа. Взаимосвязь характеристик типов представлена на рисунке 1. Наличие стрелки на рисунке свидетельствует о том, что характеристика типа требует наличия другой характеристики.

`Predefined` — маркер того, что тип встроен в язык описания входных и выходных данных, например, `int32` или `string`. `Literal` — маркер, что значения этого типа могут быть литералами, то есть, могут быть непосредственно записаны в описании ввода-вывода. `Named` — обозначает, что тип имеет собственное имя. `Can equal` — для значений типа определена операция проверки равенства. `Comparable` — значения типа полностью упорядочены, то есть для них определены операции сравнения: «больше», «меньше», «больше или равно» и «меньше или равно».

`Primitive` — маркер примитивных типов, прямой аналог которых встроен в языки программирования [14]. Язык описания входных и выходных

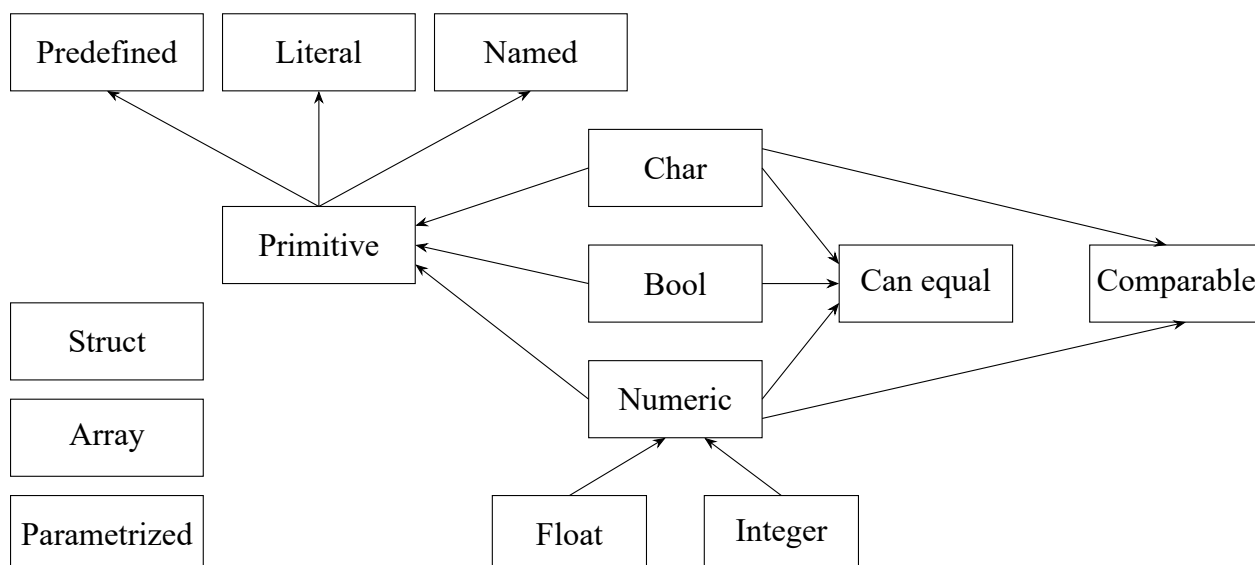


Рисунок 1 – Взаимосвязь характеристик типов

данных поддерживает восемь примитивных типов. Все типы, которые представлены в таблице 1 в разделе 2.5, за исключением, `string` являются примитивными.

`Char` — маркер символьного типа, `Bool` — маркер логического типа, `Numeric` — маркер числовых типов, которые, в свою очередь, подразделяются на `Integer` — целочисленные типы и `Float` — типы чисел с плавающей точкой.

Отдельно стоят характеристики `Struct`, `Array` и `Parametrized`. Характеристика `Struct` указывает, что тип является определенной пользователем структурой. `Parametrized` указывает, что конструктор типа параметризован, то есть имеет хотя бы один аргумент (параметр). В текущей реализации фактически характеристика `Parametrized` требует наличия характеристики `Struct`, но в реализации этот факт не используется. Характеристика `Array` указывает, что тип является массивом, то есть к значению типа может быть применена операция взятия элемента по индексу.

Примитивные типы представлены классом `PrimitiveType`. Реализация типов-массивов представлена классом `ArrayType`, содержащим информацию о длине массива (выражении, позволяющим вычислить его длину во время исполнения сгенерированного кода) и типе элементов массива. Строковый тип представлен классом `StringType` и является специализацией типа-массива для символьного типа элементов.

Типы задаваемых пользователем структур представлены классом `StructType`. Экземпляры класса хранят название пользовательской структуры. Если конструктор соответствующей структуры имеет аргументы, то так же хранится соответствующая характеристика типа (`Parametrized`). Диаграмма классов реализации типов приведена на рисунке 2.

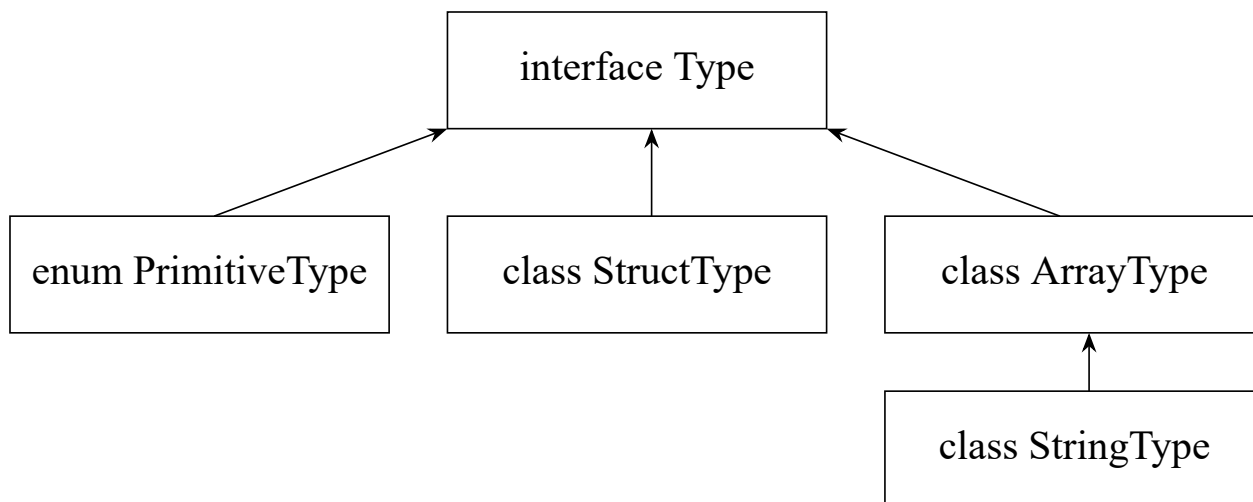


Рисунок 2 – Иерархия классов реализации типов

### 3.4. Именованные сущности и символы

В каждой области видимости доступна информация о всех именованных сущностях, которые доступны в ней для использования. Идентификаторы таких сущностей называются *символами* и представляются реализациями абстрактного класса `Symbol`. Всего реализация насчитывает пять видов символов: конструкторы, аргументы конструкторов, именованные типы, функции и переменные (поля структур).

Именованные типы представлены классом `NamedType`, являющимся оберткой над `Type`. При создании экземпляра класса `NamedType` выполняется проверка наличия характеристики `Named` у соответствующего типа.

Функции представлены классом `Function`, экземпляры которого хранят информацию о названии, типе возвращаемого значения и списке наборов требуемых характеристик типов для каждого из аргументов.

Конструкторы представлены классом `ConstructorWithBody`, экземпляры которого хранят информацию о названии, списке аргументов (параметров) конструктора, и списке элементов конструктора. Аргументы конструкторов представлены классом `ConstructorArgument`, экземпляры которого хранят информацию о названии аргумента и названии его типа, из чего следует,

что типы аргументов должны быть именованными. Подробнее про конструкторы изложено в разделе 3.5.

Переменные (поля структур) представлены классом `Variable`, экземпляры которого содержат информацию об её названии, заданных на переменную ограничениях и описании переменной. Описание переменной может быть не только типом (а, например, вызовом конструктора с аргументами), об этом будет изложено в разделе 3.5.

Иерархия классов реализации символов представлена на рисунке 3.

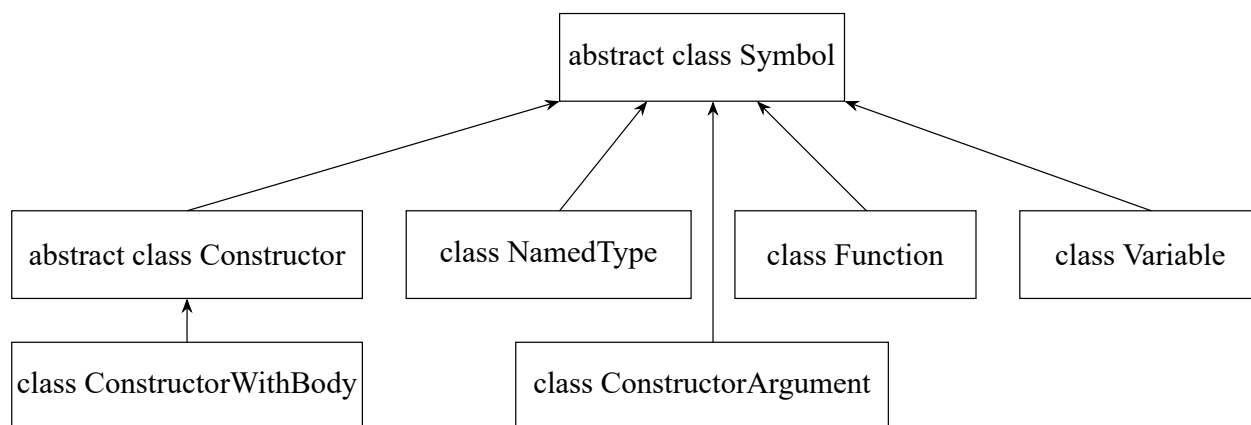


Рисунок 3 – Иерархия классов реализации символов

### 3.5. Конструкторы

Конструктором называется объект, ассоциированный с заданной пользователем именованной структурой. Конструкторы содержат информацию о своём названии (всегда совпадает с именем соответствующего типа), списке аргументов и списке элементов конструктора. Список элементов конструктора полностью определяет последовательность считываемых переменных (полей), а также разделители между этими переменными. Элементы конструктора представлены маркерным интерфейсом `ConstructorItem`, насчитывающим три реализации.

- а) Класс `Variable`, который предназначен для хранения информации о какой-либо переменной.
- б) Класс `IoModifier`, который предназначен для хранения информации о разделителе между переменными во входных или выходных данных.
- в) Класс `ConstructorIfAlt`, который предназначен для хранения информации об условном операторе, включая его тело.



В текущей реализации `IoModifier` может иметь только одно значение — `EOLN`, обозначающее перевод строки. Без указания разделителя явно будет использоваться символ пробела.

Класс `ConstructorIfAlt` отвечает за части конструкторов, которые будут задействованы или не задействованы в зависимости от какого-либо условия. Экземпляры этого класса хранят информацию о выражении, на основании которого будет выполнена положительная или отрицательная часть конструкции `if` и два списка элементов конструктора: для положительного и отрицательного случаев.

Листинг 14 содержит пример структуры. Конструктор этой структуры будет содержать три элемента.

- а) Экземпляр класса `Variable`, содержащий информацию о переменной, заданной строкой «`op: char;`».
- б) Экземпляр класса `IoModifier`, содержащий информацию о переводе строки («`eoln;`»).
- в) Экземпляр класса `ConstructorIfAlt`, содержащий информацию об условной конструкции («`if (op == '?') {...} else {...}`»).

Листинг 14 – Пример элементов структуры

```
operation {
  op: char;
  eoln;
  if (op == '?') {
    x: int32;
    y: int32;
  } else {
    i: int32;
  }
}
```

Иерархия классов реализации элементов конструкторов представлена на рисунке 4.

### 3.6. Описания переменных

Каждый экземпляр класса `Variable` хранит в себе описание переменной — информацию, необходимую для генерации кода ввода или вывода этой переменной. При этом, на этапе создания экземпляров класса `Variable` типы могут быть известны не полностью. Поэтому описания

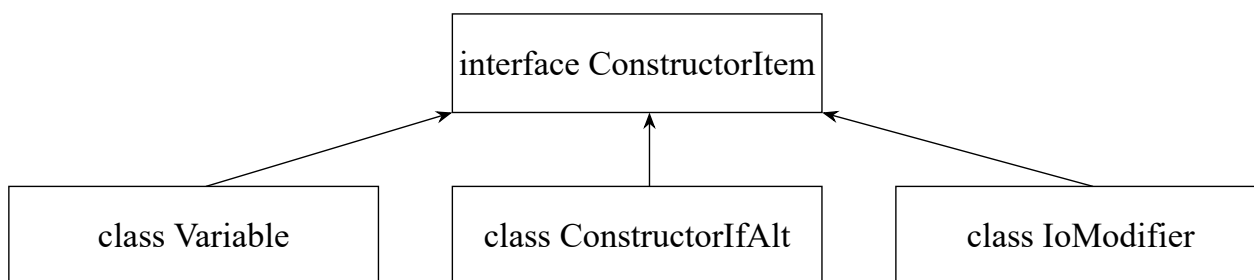


Рисунок 4 – Иерархия классов реализации элементов конструкторов

переменных представлены реализациями отдельного маркерного интерфейса `VariableDescription`. При этом любой тип может являться описанием переменной, то есть интерфейс `Type` является наследником интерфейса `VariableDescription`. Помимо этой, существует ещё три реализации описаний переменных: `NamedTypeArrayDescription`, `UnnamedStructArrayDescription` и `ParametrizedDescription`.

Листинг 15 содержит примеры объявления переменных. Для каждой переменной в комментарии указано, экземпляр какого класса будет хранить её описание.

Листинг 15 – Примеры объявления переменных

```

# Type
startPicIndex: int32;

# NamedTypeArrayDescription
s: string[i=1..n, sep='\n'];

# NamedTypeArrayDescription
pictures: picture(n, m)[i=1..k + 1, sep="\n\n"];

# ParametrizedDescription
one_picture: picture(n, m);

# UnnamedStructArrayDescription
ops: array[j=1..q, sep='\n'] of {
    x: int32;
    y: int32;
}
};
  
```

Классы `NamedTypeArrayDescription` и `UnnamedStructArrayDescription` отвечают за массивы именованных и неименованных типов соответственно. Каждый из них хранит в себе список параметров массива и описание типа элементов массива. Параметры массива

представлены классом `ArrayParameters` и содержат в себе информацию о переменной итерирования, выражении начала итерирования, выражении окончания итерирования и разделителе элементов массива во вводе и выводе.

Для именованных типов в экземплярах `NamedTypeArrayDescription` хранится описание типа элементов (экземпляр класса `VariableDescription`). Для неименованных типов в экземплярах класса `UnnamedStructArrayDescription` хранится список элементов конструктора, описывающий структуру элементов массива.

Класс `ParametrizedDescription` в некотором смысле «расширяет» интерфейс `Type`, храня дополнительно список выражений, которые будут вычислены и подставлены при вызове конструктора соответствующего типа.

Иерархия классов реализации описания переменных представлена на рисунке 5.

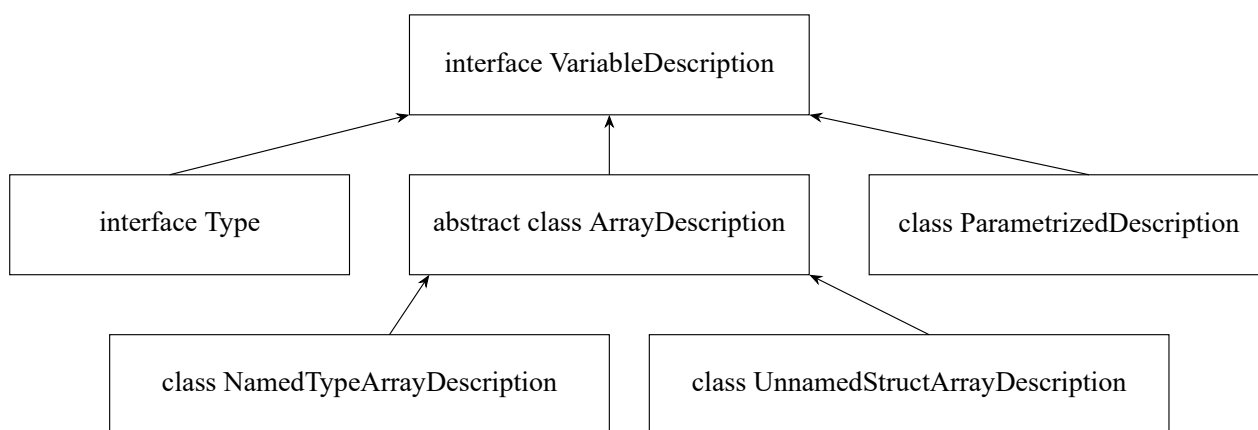


Рисунок 5 – Иерархия классов реализации описания переменных

### 3.7. Язык записи выражений

При записи вычисляемых выражений (например, при указании границ переменной итерирования при объявлении массивов и при записи логических ограничений на переменные) используется подязык выражений. Под выражением понимается языковая конструкция, которая может быть вычислена и определяет некоторое значение [10]. Синтаксис языка выражений представлен в главе 2.3, листинге Б.3 и в таблице В.1. Здесь же будет приведено описание реализации.

Любое выражение транслируется в экземпляр какого-либо наследника базового класса `P1Expression`. Всего таких наследников существует семь.

- а) **PlValue** — литерал (константа). Хранит фиксированное значение и его тип. Примеры: «1.2f», «true», «"abacaba"».
- б) **PlVarBinding** — связывание переменной. Хранит ссылку на переменную (поле структуры), которое будет подставлено по значению в выражение и вспомогательную информацию о месте связывания.
- в) **PlSubscript** — представление оператора взятия элемента массива по индексу. Хранит два выражения: выражение, возвращающее массив и выражение, возвращающее необходимый индекс. Пример: «a[i]».
- г) **PlFunctionCall** — оператор вызова функции. Хранит ссылку на вызываемую функцию и список выражений аргументов функции. Пример: «len(a)».
- д) **PlIfOperator** — тернарный оператор. Хранит три выражения: логическое, на основании которого будет выбираться какое из последующих выражений, будет возвращено в качестве результата, выражение для положительного случая и выражение для отрицательного случая. Пример: «x == 1 ? x : 0».
- е) **PlUnaryOperator** — унарный оператор. Всего представлено три унарных оператора: логическое отрицание (требует наличия характеристики **Bool** у аргумента), побитовое отрицание (требует наличия характеристики **Integer**) и унарный минус (требует **Numeric**). Примеры: «-x», «~x», «!x».
- ж) **PlBinaryOperator** — бинарный оператор. Всего представлено девятнадцать бинарных операторов. Все бинарные операторы представлены в таблице В.1. Примеры: «x + y», «x && y».

Каждый бинарный оператор хранит в себе список выражений — его аргументов. Проверка совместимости типов в выражениях выполняется на этапе построения абстрактного синтаксического дерева выражения. Операторы «больше», «меньше», «больше или равно» и «меньше или равно» не имеют ассоциативности, оператор возведения в степень имеет правую ассоциативность, остальные операторы — левую. Операторы «больше», «меньше», «больше или равно» и «меньше или равно» требуют наличия характеристики **Comparable** у типов их аргументов, операторы умножения, деления, сложения, вычитания и возведения в степень требуют наличия характеристики **Numeric**, операторы взятия остатка от деления, побитовые «И», «ИЛИ» и «ис-

ключающее ИЛИ», и операторы битовых сдвигов требуют наличия характеристики `Integer`. Операторы сравнения на равенство и не равенство требуют наличия характеристики `Can equal`. Логические операторы требуют `Bool`.

Полная иерархия классов реализации языка выражений представлена на рисунке 6.

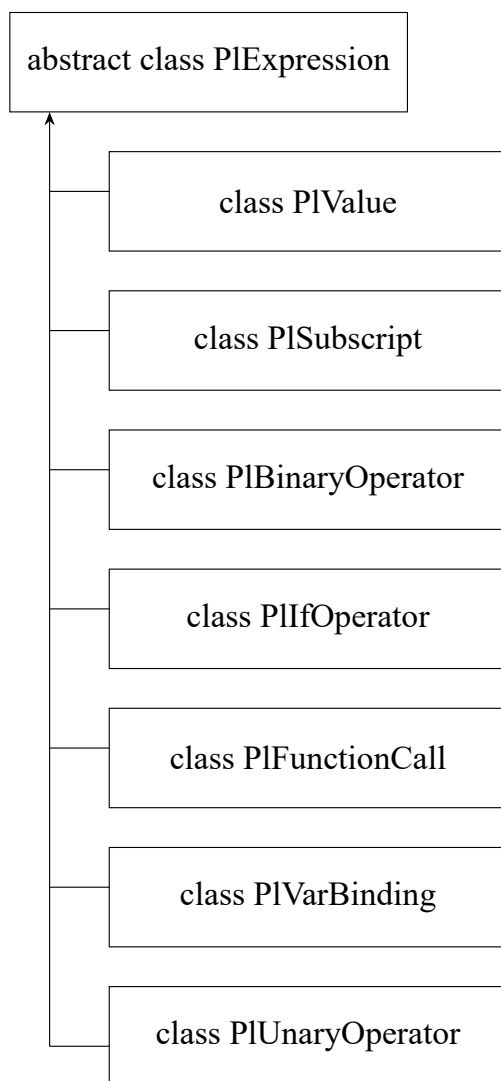


Рисунок 6 – Иерархия классов реализации языка выражений

### 3.8. Внутреннее представление

После выполнения лексического и синтаксического разбора составляется глобальная область видимости — специальный экземпляр класса `Scope`, содержащий, в том числе, информацию об именах всех объявленных пользователем структур. Эта информация получается обходом дерева разбора до глубины, достаточной, чтобы зарегистрировать имя каждой структуры. Реализация этого обхода выполнена с использованием стандартного шаблона проектирования «Посетитель» [19]. ANTLR 4 предоставляет стандартный интерфейс для

реализации этого шаблона проектирования. Реализация представлена классом `FindGlobalSymbolsVisitor`.

Затем происходит полный обход дерева разбора описания ввода-вывода, во время которого происходит непосредственно создание экземпляров внутреннего представления, описанных выше. Реализация обхода представлена классом `ResolveSymbolsVisitor`. Сначала происходит обход поддеревя, отвечающего за структуру входных данных (`input`), затем — обход остального дерева. Это нужно для поддержания корректности областей видимости, например, чтобы из выходных данных можно было ссылаться на переменные во входных данных.

Для каждого поддеревя, отвечающего целиком за объявленную пользователем структуру, создается экземпляр конструктора, в который помещается описание структуры. При этом соответствующий структуре тип уже существует в глобальной области видимости, так как создан на предыдущем этапе обработки.

Области видимости могут «вкладываться» друг в друга, образуя структуру данных «стек» [9]. Помимо стека областей видимости поддерживается стек названий структур и переменных итерирования для того, чтобы для каждой переменной можно было точно указать её местонахождение в иерархии структур. Кроме того, для каждого символа поддерживается список символов, от которых он зависит, то есть, список используемых им символов. Таким образом получается ациклический ориентированный граф [16] зависимостей символов. Циклические зависимости в текущей реализации не поддерживаются.

При обходе вершины, отвечающей за описание переменной происходит проверка, что ещё не было объявлено переменной с таким же именем, обновляется текущий путь в иерархии структур. Затем происходит обход поддеревя, отвечающего за описание переменной, после чего переменная помещается в текущую область видимости. Если переменная является массивом, то выполняется обход поддеревя с описанием массива. Если переменная имеет выражение-ограничение, то так же выполняется обход соответствующего поддеревя.

Если описание переменной является массивом неименованной структуры, создается новая область видимости для внутренней структуры массива и выполняется обход параметров массива и непосредственно внутренней струк-

туры. Иначе, описание переменной содержит именованный тип, проверяется его наличие в текущей области видимости. Если описание содержит аргументы конструктора, то выполняется проверка, что тип, соответствующий конструктору, имеет характеристику `Parametrized`. При обходе описаний массивов в любом случае создается новая область видимости, чтобы сделать доступными для использования переменные итерирования.

При обходе вершины, отвечающей за конструкцию `if` выполняется обход поддерева с логическим выражением и обходы положительной и отрицательной веток синтаксической конструкции. Каждая ветка при этом трактуется как отдельная неименованная структура с созданием соответствующей области видимости.

Обход поддеревьев, отвечающих за выражения, осуществляется экземплярами класса `PExpressionBuildVisitor`. Его устройство тривиально, потому что он решает стандартную задачу перевода дерева разбора в абстрактное синтаксическое дерево выражения.

### **Выводы по главе 3**

В этой главе был описан процесс обработки описания входных и выходных данных, изложены детали реализации внутреннего представления языка описания входных и выходных данных, описано устройство системы типов языка.

## ГЛАВА 4. КОДОГЕНЕРАЦИЯ И ИНТЕГРАЦИЯ В СИСТЕМУ ПОДГОТОВКИ ЗАДАЧ «POLYGON»

В этой главе описывается процесс генерации исходных кодов чекера, валидатора и грейдеров, а так же интеграция решения в систему подготовки задач по спортивному программированию «Polygon» и тестирование решения.

### 4.1. Кодогенерация

Последним этапом обработки языка описания входных и выходных данных является непосредственно генерация программных компонентов задачи. Текущая реализация может генерировать: валидаторы и чекеры с использованием библиотеки `testlib.h` на языке программирования C++, а также грейдеры и заготовки решений на языках программирования C++, Python и Java [13]. Эти языки были выбраны как наиболее популярные на платформе Codeforces. Подсчитанная статистика использования языков программирования на платформе Codeforces с января по март 2023 года приведена в таблице 2.

Таблица 2 – Статистика использования языков программирования на платформе Codeforces

Язык программирования	Количество посланных решений
C++	11 020 882
Python 3	984 533
Java	376 252
C	307 082
C#	38 644
Pascal	20 444
Rust	11 866
Go	11 587
Python 2	11 220
JavaScript	13 845
Kotlin	10 828
PHP	4 713
Haskell	1 845
Ruby	1 424
D	1 264
Perl	443
Scala	405
Ocaml	165



Для генерации кода достаточно получить содержимое глобальной области видимости, которое описано в предыдущей главе — объекты в ней содержат всю необходимую информацию. Так же, на этом этапе не требуется никаких дополнительных проверок, так как они выполнены на предыдущих этапах обработки.

Вне зависимости от целевого языка программирования или компонента, генерация кода состоит из нескольких этапов.

- а) Генерация объявлений структур.
- б) Генерация кода ввода.
- в) Генерация кода вывода (при необходимости).
- г) Подстановка сгенерированного кода в шаблон, в котором содержатся необходимые стандартные определения, точка входа (при необходимости), вызовы чтения выходных данных, записи выходных данных и т.д.

Генерация кода происходит построчно, то есть, во время генерации составляются списки сгенерированных строк, которые затем будут транслированы в полноценный файл с исходным кодом. Такой подход выбран для того, чтобы можно было обеспечить форматирование сгенерированного кода и сделать его «человекочитаемым».

Приложение Е содержит примеры сгенерированного кода.

#### **4.1.1. Генерация объявлений структур**

Генерация объявлений структур происходит следующим образом. Поочередно перебираются все конструкторы в глобальной области видимости, то есть все объявленные пользователем структуры, в порядке топологической сортировки графа зависимостей символов. Стоит отметить, что топологическая сортировка графа всегда существует, так как петли и циклы в этом графе запрещены. Таким образом, в сгенерированном коде структуры находятся строго после всех необходимых объявлений, то есть зависимости направлены снизу вверх. Направление зависимостей в исходном коде критично для таких языков программирования как C++.

Для генерации объявления каждой конкретной структуры происходит обход списка всех элементов конструктора за исключением модификаторов ввода-вывода, так как они не влияют на представление данных. Переменные транслируются тривиальным, зависимым от целевого языка программирова-

ния, образом — в сгенерированном коде объявляется соответствующее поле структуры (или класса) с соответствующим типом. Условные конструкции (`if`) «схлопываются» в «плоскую» структуру, то есть поля, объявленные в разных ветках будут транслироваться непосредственно в поля внешней структуры или класса.

Для массивов неименованных структур генерируются структуры или классы с названиями, соответствующими названиям переменных, поэтому процесс генерации кода определений для них не отличается от процесса генерации кода определений для именованных структур.

Реализации генерации объявлений структур для языков программирования C++, Java и Python представлены классами `CppStructDeclarationsTranslator`, `JavaStructDeclarationsTranslator` и `PythonStructDeclarationsTranslator` соответственно. В случае C++ генерируется набор структур (`struct`), в случаях Java и Python генерируется набор классов (`class`).

Листинг 16 содержит сгенерированный код определений структур для задачи «Копия копии копии». Условие задачи находится в приложении А, описание входных и выходных данных приведено в листинге 6 в разделе 2.2.

#### 4.1.2. Генерация выражений

Код для ввода и вывода может использовать выражения, например, при определении интервалов переменных итерирования или для проверки логических условий на переменные. Генерация кода выражений, для простоты реализации, выделена в отдельные классы: `CppPlExpressionTranslator`, `JavaPlExpressionTranslator` и `PythonPlExpressionTranslator` для языков программирования C++, Java и Python соответственно.

Для того, чтобы корректно провести трансляцию выражений необходимо знать местоположение выражения в описании входных и выходных данных и набор локальных переменных в текущей области видимости. Эта информация передается при создании экземпляра соответствующего класса-транслятора.

Трансляция происходит следующим образом: абстрактное синтаксическое дерево выражений преобразуется в конструкции соответствующего целевого языка программирования по заранее установленным правилам. Если в языке программирования нет прямых аналогов нужной функциональности,

Листинг 16 – Пример сгенерированного кода объявлений структур на языке C++

```
struct ops_t
{
    int op;
    int x;
    int y;
    int i;
};

struct picture_t
{
    std::vector<std::string> s;
};

struct input_t
{
    int n;
    int m;
    int k;
    std::vector<picture_t> pictures;
};

struct output_t
{
    int startPicIndex;
    int q;
    std::vector<ops_t> ops;
};
```

происходит трансляция в вызов вспомогательной функции, реализация которой будет предоставлена в шаблоне на соответствующем языке.

#### 4.1.3. Генерация кода ввода

Генерация кода ввода происходит следующим образом. Перебираются все конструкторы в глобальной области видимости, для каждого генерируется функция, которая будет считывать из потока ввода соответствующую структуру и возвращать её в качестве результата. При этом, если целевой язык — C++, то сигнатуры функций и их реализации генерируются по-отдельности, чтобы функции ссылались только на ранее объявленные в коде функции.

Для каждой структуры генерируется функция считывания, которая поочередно считывает поля соответствующей структуры или класса из стандартного потока ввода. Если очередное поле является встроенным в язык описания входных и выходных данных типом, то вызывается соответствующая кон-

структура целевого языка для считывания из ввода. Если же очередное считываемое поле является структурой, то происходит вызов сгенерированной функции считывания соответствующей структуры.

Для считывания массивов генерируется цикл на целевом языке программирования, в теле которого происходит считывание каждого элемента массива, которые поочередно добавляются в конец массива.

При наличии в структуре, для которой генерируется код ввода, условной конструкции `if`, генерируется условная конструкция на целевом языке программирования. Генерация веток условной конструкции происходит так же, как и генерация самих структур.

Для целевых языков программирования C++ и Java используются стандартные средства разбора потока ввода — `cin` [31] и `Scanner` [21] соответственно. Стоит отметить, что `Scanner` не является идеальным решением для разбора потока ввода ввиду его низкой производительности [23]. Этот недостаток можно исправить, если для ввода использовать методы, аналогичные примененным в библиотеке `testlib4j` [32]. Язык Python предоставляет ограниченные возможности для разбора стандартного потока ввода, поэтому, для него была написана собственная реализация разбора, которая встроена в шаблон для этого языка.

Реализация генерации кода ввода представлена классами `CppGraderReadTranslator`, `JavaGraderReadTranslator` и `PythonGraderReadTranslator`.

Листинг 16 содержит сгенерированный код для считывания структур `input` и `picture` для задачи «Копия копии копии». Условие задачи находится в приложении А, описание входных и выходных данных приведено в листинге 6 в разделе 2.2.

#### 4.1.4. Особенности генерации валидатора и чекера

Генерируемый код валидатора, по своей сути, является несколько модифицированным кодом ввода структуры `input` и её зависимостей. При этом сгенерированный код использует не стандартный способ разбора входных данных, а методы семейства «`readX(...)`» из библиотеки `testlib.h`.

При этом, между считыванием полей, происходит проверка, что разделитель соответствует заданному в описании ввода-вывода, вызовом метода `readSpace()` или `readEoln()`. Так же, после кода считывания каждого по-

Листинг 17 – Пример сгенерированного кода ввода

```

picture_t read_picture(int n, int m)
{
    picture_t picture;
    for (int i = 1; i <= n; i++)
    {
        std::string tmp;
        cin >> tmp;
        picture.s.push_back(tmp);
    }
    return picture;
}

input_t read_input()
{
    input_t input;
    cin >> input.n;
    cin >> input.m;
    cin >> input.k;
    for (int i = 1; i <= (input.k + 1); i++)
    {
        picture_t tmp;
        tmp = read_picture(input.n, input.m);
        input.pictures.push_back(tmp);
    }
    return input;
}

```

ля, при необходимости, вставляется код, проверяющий заданное в описании ввода-вывода логическое ограничение (вызов метода `ensuref(...)`).

В случае чекера, генерируется исходный код для считывания структур `input` и `output` и их зависимостей. В каждой генерируемой функции добавляется дополнительный аргумент `InStream& stream` — поток `testlib.h`, из которого будет происходить считывание. Всего в чекере существует три потока: поток входных данных, поток ответа участника и поток ответа жюри (ответа авторского решения). В чекере происходит считывание всех трёх потоков.

Реализация генерации кода валидатора и чекера представлена классами `CppTestlibValidatorTranslator` и `CppTestlibCheckerTranslator` соответственно. Листинг 18 содержит пример генерируемого кода для чекера.

#### 4.1.5. Генерация кода вывода

Генерация кода для вывода структур происходит аналогично генерации кода для ввода за исключением того, что методы, осуществляющие вывод не

Листинг 18 – Пример сгенерированного кода ввода в чекере

```
ops_t read_ops(InStream& stream)
{
    ops_t ops;
    ops.op = stream.readInt();
    if (ops.op == 1)
    {
        ops.x = stream.readInt();
        stream.ensuref((2 <= ops.x) && (ops.x <= input.n - 1));
        ops.y = stream.readInt();
        stream.ensuref((2 <= ops.y) && (ops.y <= input.m - 1));
    }
    else
    {
        ops.i = stream.readInt();
        stream.ensuref((1 <= ops.i) && (ops.yi <= input.k + 1));
    }
    return ops;
}

output_t read_output(InStream& stream)
{
    output_t output;
    output.startPicIndex = stream.readInt();
    stream.ensuref(1 <= output.startPicIndex &&
        output.startPicIndex <= input.k + 1);
    output.q = stream.readInt();
    stream.ensuref(0 <= output.q &&
        output.q <= iomarkup::pow(10, 7));
    for (int j = 1; j <= output.q; j++)
    {
        ops_t tmp;
        tmp = read_ops(stream);
        output.ops.push_back(tmp);
    }
    return output;
}
```

возвращают структуру, а принимают её в качестве аргумента. Так же, при выводе учитываются разделители полей в структурах и, при необходимости, в код вставляются операции вывода символов перевода строки. Во всех целевых языках используются стандартные конструкции для форматированного вывода данных (cout, print(...), PrintStream).

Реализация генерации кода вывода представлена классами CppGraderWriteTranslator, JavaGraderWriteTranslator и

PythonGraderWriteTranslator. Листинг 19 содержит пример генерируемого кода вывода.

Листинг 19 – Пример сгенерированного кода вывода

```
void write_ops(ops_t const& ops)
{
    cout << ops.op;
    cout << ' ';
    if (ops.op == 1)
    {
        cout << ops.x;
        cout << ' ';
        cout << ops.y;
    }
    else
    {
        cout << ops.i;
    }
}

void write_output(output_t const& output)
{
    cout << output.startPicIndex;
    cout << endl;
    cout << output.q;
    cout << endl;
    for (int j = 1; j <= output.q; j++)
    {
        write_ops(output.ops[j - 1]);
        cout << endl;
    }
}
```

#### 4.1.6. Шаблоны генерации кода

Выше описан процесс генерации кода, зависящего от описания входных и выходных данных задачи. Однако, часть результирующего исходного кода берется из заранее написанных шаблонов. Шаблоны содержат вспомогательные определения, необходимые для работы сгенерированного кода. Например, шаблон для грейдера на языке программирования Python содержит определение вспомогательного класса `Scanner` для разбора ввода.

После подстановки сгенерированного кода в шаблон получается один или несколько полноценных файлов с исходным кодом валидатора, чекера или грейдеров. После этого автору задачи остается лишь дополнить файлы специфичной для задачи логикой.

Приложение Д содержит исходные коды шаблонов для разных целевых языков и компонентов.

## 4.2. Интеграция в «Polygon»

Описанная в данной работе система была реализована на языке программирования Java версии 17. Эта версия была выбрана как последняя стабильная с долгим циклом поддержки. Так же, использование современной версии языка программирования Java позволило сократить исходный код реализации решения за счет использования последних возможностей этого языка.

Однако, интеграция непосредственно в систему подготовки задач «Polygon» при этом оказалось затруднительной, так как исходный код «Polygon» написан с использованием Java версии 8. Поэтому данная работа была реализована в качестве отдельного сервиса, которому «Polygon» при необходимости отправляет запросы на генерацию кода и в ответ получает готовые файлы.

Система кодогенерации упаковывается в исполняемый jar-файл и может использоваться как в качестве отдельного сервиса, так и как утилита командной строки. Точкой входа является класс `IoMarkupCli`.

При работе в режиме утилиты командной строки, считывается файл, указанный в аргументах командной строки, на языке описания входных и выходных данных и создаются сгенерированные файлы с исходным кодом запрошенных в аргументах командной строки целевых компонент задачи на целевых языках. После этого исполнение завершается.

При запуске в режиме сервиса (демона, от англ. *daemon*) слушается указанный в аргументах TCP-порт, на который будут поступать запросы. В силу особенностей существующей архитектуры системы подготовки задач по спортивному программированию «Polygon» протоколом обращения к сервису генерации кода, так же как и протоколом получения ответа, был выбран XML-RPC [12], содержащий сериализованные данные запроса в формате Protobuf [28]. Ответ так же содержит сериализованные данные в формате Protobuf.

### 4.2.1. Структура запросов и ответов

Определение структуры запросов и ответов в формате Protobuf приведено в приложении Г.



В запросе (`GenerateRequest`) содержится строка (поле `io_markup`) — описание входных и выходных данных и набор структур, описывающих целевой компонент и язык программирования для генерации кода (поле `target_file_description`). `TargetFileDescription` содержит целевой компонент (один из элементов перечисления `TargetComponent`) и язык генерации (один из элементов перечисления `TargetLanguage`).

В ответе (`GenerateResponse`) содержится список сгенерированных файлов в поле `result_file` и, возможно, сообщение об ошибке в поле `error_message`. Каждый сгенерированный файл (`ResultFile`) содержит компонент и язык программирования, название файла (поле `name`), содержимое файла в строковом виде (поле `content`) и необязательное сообщение об ошибке.

Все строки представляются в кодировке UTF-8 [15].

#### 4.2.2. Поддержка в «Polygon»

Поддержка со стороны исходного кода системы подготовки задач по спортивному программированию «Polygon» сводится к реализации интерфейса пользователя, отправки запросов, приема ответов и записи в репозиторий задачи сгенерированных файлов с исходным кодом. Процесс отправки запросов и приема ответов прозрачен для кода «Polygon» ввиду использования XML-RPC, то есть цикл «запрос-ответ» выглядит как один вызов удаленной процедуры. В случае наличия ошибок в ответе они будут выданы пользователю (автору задачи).

С точки зрения пользователя процесс выглядит следующим образом. В репозитории задачи в разделе ресурсов (Files — Resource Files) создается специальный файл на языке описания входных и выходных данных с именем «`iomarkup.txt`». В файле записывается описание ввода-вывода. В списке файлов появляется ссылка на страницу генерации (Generate). Страница со списком файлов показана на рисунке 7.

Resource Files			<a href="#">New File</a> <a href="#">Add Files</a>	
Name	Length	Modified	Actions	
<code>iomarkup.txt</code> <sup>Ⓢ</sup>	72	2023-04-30 17:18:48	<a href="#">Delete</a> <a href="#">Download</a> <a href="#">Edit</a> <a href="#">View</a> <a href="#">Generate</a>	<input type="checkbox"/>
<a href="#">Rename?</a>				

Рисунок 7 – Список ресурсов задачи

На странице генерации пользователь выбирает какие программные компоненты нужно сгенерировать и на каких языках программирования. Страница с выбором показана на рисунке 8.

[« Back to files page](#)

	SOLUTION	GRADER	VALIDATOR	CHECKER
CPP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
JAVA	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
PYTHON	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Рисунок 8 – Выбор параметров генерации

### 4.2.3. Сервис кодогенерации

В силу особенностей используемой библиотеки, реализующей протокол XML-RPC, реализация конечной точки обработки запроса на генерацию кода представлена реализацией интерфейса `com.codeforces.interop.iomarkup.InteropRpc` классом `InteropRpcImpl`. При получении запроса происходит десериализация тела запроса на генерацию из массива байт, затем обработка запроса по алгоритмам, описанным выше в работе и сериализация ответа с помощью `Protobuf` в массив байт. При возникновении ошибки обработка прекращается и ответ, содержащий описание ошибки, возвращается немедленно.

### 4.3. Тестирование

Тестирование предложенного в данной работе решения производилось на нескольких соревнованиях на платформе Codeforces. Для задач соревнований были вручную написаны описания входных и выходных данных и сгенерированы валидатор, чекер и грейдеры. Для каждого сгенерированного компонента на каждом языке проверялось, что сгенерированный код компилируется без ошибок (в случае Python проверялась корректности синтаксиса).

Рассмотрим множество всех возможных файлов. Валидатор для задачи должен выдавать положительный результат (вердикт «корректный тест») на подмножестве этого множества. При этом, сгенерированный валидатор должен выдавать положительный результат для подмножества множества всех возможных файлов, но, для надмножества множества корректных тестов, определяемых «настоящим» валидатором. Иными словами, не должно быть

такого файла, для которого «настоящий валидатор» выдаст вердикт «корректный тест», а сгенерированный — «некорректный тест».

Для каждой задачи при тестировании автоматически загружался пакет задачи из «Polygon». Для каждого теста валидатора, для которого валидатор из пакета задачи выдавал вердикт «корректный тест» проверялось, что сгенерированный валидатор выдавал такой же вердикт. Для всех наборов входных данных из пакета задачи проверялось, что сгенерированный валидатор выдает вердикт «корректный тест» на каждом из них.

Тестирование происходило в условиях, максимально приближенных к тем, в которых происходит тестирование на платформах «Codeforces» и «Polygon». Для этого проект «Invoker», который реализует полный цикл тестирования на вышеупомянутых платформах, был подготовлен для использования в качестве отдельной библиотеки и добавлен в качестве тестовой зависимости в реализацию решения в данной работе.

Сводная информация о результатах тестирования представлена в таблице 3. По столбцам — вердикты валидатора из пакета задачи, по строкам — вердикты сгенерированного валидатора. В ячейках таблицы указано количество тестов. Из таблицы видно, что сгенерированный валидатор не выдал вердикт «некорректный тест» ни для одного корректного теста.

Таблица 3 – Сводная информация о результатах тестирования

	Корректный тест	Некорректный тест
Корректный тест (сген. валидатор)	69	5
Некорректный тест (сген. валидатор)	0	15

## Выводы по главе 4

В этой главе был описан процесс частичной генерации исходного кода валидатора, чекера и грейдеров на основании обработанного описания входных и выходных данных. Описан способ интеграции в систему подготовки задач по спортивному программированию «Polygon». Описан процесс тестирования реализованной системы кодогенерации.

## ЗАКЛЮЧЕНИЕ

В данной выпускной квалификационной работе была разработана система, частично автоматизирующая процесс построения программных компонентов — чекера, валидатора и грейдеров — при подготовке задач по спортивному программированию.

Был спроектирован язык описания входных и выходных данных, позволяющий описывать структуру и формат файлов ввода и вывода для задачи. Реализована и описана система частичной генерации исходных кодов валидатора, чекера и грейдеров, включающих в себя реализацию ввода-вывода данных, описание структур, необходимых в задаче, а так же проверку логических условий, заданных автором. Система позволяет частично генерировать исходный код валидатора и чекера на языке программирования C++ с использованием библиотеки `testlib.h`, а так же генерировать исходный код грейдеров на языках программирования C++, Java и Python.

Реализованная система интегрирована в сервис для подготовки задач по спортивному программированию «Polygon», на работу получен акт о внедрении.

В дальнейшем планируется доработка системы: добавление возможности генерации исходных кодов генераторов и секций «входные данные» и «выходные данные» в условиях задач, а также поддержка рекурсивных структур.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 Codeforces [Электронный ресурс]. — 2023. — URL: [https : / / codeforces.com/](https://codeforces.com/).
- 2 Задача F. Копия копии копии [Электронный ресурс]. — 2022. — URL: <https://codeforces.com/contest/1772/problem/F>.
- 3 *Корнеев Г.* Игровой метод тестирования решений на соревнованиях по программированию и его реализация [Электронный ресурс]. — 2002. — URL: [https : / / www . kgeorgiy . info / theses / Korneev \\_ GA \\_ -- \\_Bachelor\\_thesis.pdf](https://www.kgeorgiy.info/theses/Korneev_GA_-_Bachelor_thesis.pdf).
- 4 *Мирзаянов М.* Правила соревнований Codeforces [Электронный ресурс]. — 2012. — URL: <https://codeforces.com/blog/entry/4088>.
- 5 *Мирзаянов М.* Коротко о testlib.h [Электронный ресурс]. — 2015. — URL: <https://codeforces.com/testlib>.
- 6 Требования к организации и проведению регионального этапа Всероссийской олимпиады школьников в 2022/2023 учебном году по информатике [Электронный ресурс]. — 2022. — URL: [https : / / edsoo . ru / download/1076/?hash=33c893065154553c8704a8a0cbe9860e](https://edsoo.ru/download/1076/?hash=33c893065154553c8704a8a0cbe9860e).
- 7 *Чернов А.* Ejudge contest management system [Электронный ресурс]. — 2023. — URL: <https://ejudge.ru/>.
- 8 Яндекс.Контест [Электронный ресурс]. — 2023. — URL: [https : / / contest.yandex.ru/](https://contest.yandex.ru/).
- 9 *Knuth D. E.* The art of computer programming. volume 1: Fundamental algorithms. volume 2: Seminumerical algorithms // Bull. Amer. Math. Soc. — 1997.
- 10 *Mitchell J. C., Apt K.* Concepts in programming languages. — Cambridge University Press, 2003.
- 11 *Parr T.* The definitive ANTLR 4 reference // The Definitive ANTLR 4 Reference. — 2013. — С. 1–326.
- 12 Programming Web Services with XML-RPC: Creating Web Application Gateways / S. S. Laurent [и др.]. — "O'Reilly Media, Inc.", 2001.
- 13 *Schildt H.* Java: a beginner's guide. — McGraw-Hill Education, 2022.

- 14 *Stone R. G., Cooke D. J.* Program construction. — Cambridge University Press, 1987.
- 15 The unicode standard / J. D. Allen [и др.] // Mountain view, CA. — 2012.
- 16 *Trudeau R. J.* Introduction to graph theory. — Courier Corporation, 2013.
- 17 Компиляторы: принципы, технологии и инструментарий / А. Ахо [и др.] // М.: Вильямс. — 2008. — Т. 1. — С. 257.
- 18 *Корнеев Г., Елизаров Р.* Автоматическое тестирование решений на соревнованиях по программированию // Телекоммуникации и информатизация образования. — 2003. — № 1. — С. 61–73.
- 19 Паттерны объектно-ориентированного проектирования / Г. Эрх [и др.]. — ”Издательский дом Питер”, 2020.
- 20 *Страуструп Б.* Язык программирования C++. — Бином, 2011.
- 21 Class Scanner (Java SE 19) [Электронный ресурс]. — 2022. — URL: [https://download.java.net/java/early\\_access/panama/docs/api/java.base/java/util/Scanner.html](https://download.java.net/java/early_access/panama/docs/api/java.base/java/util/Scanner.html).
- 22 Competitive programming [Электронный ресурс]. — 2023. — URL: [https://en.wikipedia.org/w/index.php?title=Competitive\\_programming&oldid=1145645404](https://en.wikipedia.org/w/index.php?title=Competitive_programming&oldid=1145645404).
- 23 Faster Input for Java [Электронный ресурс]. — URL: <https://www.cpe.ku.ac.th/~jim/java-io.html>.
- 24 ICPC Northern Eurasia Regional Contest Rules [Электронный ресурс]. — 2023. — URL: <https://neerc.ifmo.ru/information/contest-rules.html>.
- 25 IOI Competition Rules [Электронный ресурс]. — 2022. — URL: <https://ioi2022.id/competition-rules/>.
- 26 *Loc T. Q.* Polygon.Codeforces Tutorial - A Guide to Problem Preparation [Part 1] [Электронный ресурс]. — 2022. — URL: <https://quangloc99.github.io/2022/03/08/polygon-codeforces-tutorial.html>.
- 27 Polygon [Электронный ресурс]. — 2023. — URL: <https://polygon.codeforces.com/>.

- 28 Protocol Buffers [Электронный ресурс]. — 2023. — URL: <https://protobuf.dev/>.
- 29 Python 2.7.18 documentation [Электронный ресурс]. — 2020. — URL: <https://docs.python.org/2.7/>.
- 30 Python 3.11.3 documentation [Электронный ресурс]. — 2023. — URL: <https://docs.python.org/3.11/>.
- 31 std::cin, std::wcin [Электронный ресурс]. — 2021. — URL: <https://en.cppreference.com/w/cpp/io/cin>.
- 32 testlib4j [Электронный ресурс] / М. Буздалов [et al.]. — 2019. — URL: <https://github.com/mbuzdalov/testlib4j>.
- 33 *Мирзаянов М.* Polygon: расширенные свойства ресурсов (грейдеры) [Электронный ресурс]. — 2019. — URL: <https://codeforces.com/blog/entry/66916>.

## Приложение А. УСЛОВИЕ ЗАДАЧИ-ПРИМЕРА «КОПИЯ КОПИИ КОПИИ»

Все началось с черно-белой картинки, которую можно представить как матрицу  $n \times m$  такую, что все ее элементы равны 0 или 1. Строки пронумерованы от 1 до  $n$ , столбцы пронумерованы от 1 до  $m$ .

Над картинкой проделали несколько операций (возможно, ноль), каждая — одного из двух типов:

- выбрать ячейку такую, что она не находится на границе (строка не 1 и не  $n$ , столбец не 1 и не  $m$ ) и она окружена четырьмя ячейками противоположного цвета (четырьмя нулями, если она единица, и наоборот), и перекрасить ее саму в противоположный цвет;
- сделать копию текущей картинки.

Обратите внимание, что порядок операций мог быть произвольным, они могут не чередоваться.

Вам сообщили результат: все  $k$  копий, которые были сделаны. Кроме того, вам сообщили первоначальную картинку. Однако, все эти  $k + 1$  картинки были перемешаны.

Восстановите последовательность операций. Если существует несколько ответов, то выведите любой из них. Тесты построены из реальной последовательности операций, т. е. хотя бы один ответ всегда существует.

### Формат входных данных

В первой строке записаны три целых числа  $n, m$  и  $k$  ( $3 \leq n, m \leq 30$ ;  $0 \leq k \leq 100$ ) — количество строк, количество столбцов и количество сделанных копий, соответственно.

Затем следуют  $k + 1$  картинок —  $k$  копий и первоначальная. Их порядок произвольный.

Каждая картинка состоит из  $n$  строк, в каждой по  $m$  символов, каждый символ — это 0 или 1. **Перед каждой картинкой идет пустая строка.**

### Формат выходных данных

В первой строке выведите одно целое число — номер первоначальной картинки. Картинки пронумерованы от 1 до  $k + 1$  в порядке, в котором они заданы во входных данных.

Во второй строке выведите одно целое число  $q$  — количество операций.



В каждой из следующих  $q$  строк должна быть записана одна операция. Операции должны быть перечислены в том порядке, в котором они применялись. Каждая операция может быть одного из двух типов:

- $1 \ x \ y$  — перекрасить ячейку  $(x, y)$  ( $y$ -я ячейка в  $x$ -й строке, она не должна лежать на границу и должна быть окружена четырьмя клетками противоположного от себя цвета);
- $2 \ i$  — сделать копию текущей картинки и присвоить ей номер  $i$  (картинка под номером  $i$  должна совпадать с текущей картинкой).

Каждый номер от 1 до  $k + 1$  должен встречаться в выводе ровно один раз — один из них — это номер первоначальной картинки, остальные  $k$  — аргументы операций второго типа.

Если существует несколько ответов, то выведите любой из них. Тесты построены из реальной последовательности операций, т. е. хотя бы один ответ всегда существует.

### Пример

Входные данные	Выходные данные
3 3 1	
010	2
111	2
010	1 2 2
	2 1
010	
101	
010	

## Приложение Б. ГРАММАТИКА ЯЗЫКА ОПИСАНИЯ

Листинг Б.1 – Определения токенов в формате ANTLR 4

```
LINE_COMMENT: '#' .*? '\r'? ('\n' | EOF) -> skip;
COMMENT: '/*' .*? '*/' -> skip;
```

```
fragment ESCAPED_CHAR: '\\\n';
```

```
CHAR: '\'' ((~['\\r\n']) | ESCAPED_CHAR) '\'';
STRING: '"' ((~["\\r\n"]) | ESCAPED_CHAR)* '"';
TRUE: 'true';
FALSE: 'false';
LOGICAL_AND: '&&';
LOGICAL_OR: '||';
LOGICAL_NOT: '!';
BITWISE_AND: '&';
BITWISE_XOR: '^';
PIPE: '|';
BITWISE_NOT: '~';
EQUALS: '==';
NOT_EQUALS: '!=';
LESS_EQUAL: '<=';
GREATER_EQUAL: '>=';
LESS: '<';
GREATER: '>';
PLUS: '+';
MINUS: '-';
POW: '**';
MULTIPLICATION: '*';
DIVISION: '/';
MODULO: '%';
DOT: '.';
COLON: ':';
QUESTION: '?';
ASSIGN: '=';
COMMA: ',';
SEMICOLON: ';';
LBRACE: '{';
RBRACE: '}';
LBRACKET: '[';
RBRACKET: ']';
LPAR: '(';
```

```

RPAR: '>';
SEP: 'sep';
EOLN_MODIFIER: 'eoln';
IF: 'if';
ARRAY: 'array';
OF: 'of';
ELSE: 'else';

fragment U_NUM_LITERAL_MODIFIER: 'u' | 'U';
fragment L_NUM_LITERAL_MODIFIER: 'l' | 'L';
fragment F_NUM_LITERAL_MODIFIER: 'f' | 'F';
fragment EXPONENT: 'e' | 'E';
fragment DIGITS: [0-9][0-9'_]*;

fragment INTEGER_SUFFIX:
    U_NUM_LITERAL_MODIFIER? L_NUM_LITERAL_MODIFIER?;
fragment FLOAT_SUFFIX:
    ('.' DIGITS)? (EXPONENT DIGITS)? F_NUM_LITERAL_MODIFIER?;

NUM_VALUE: DIGITS INTEGER_SUFFIX FLOAT_SUFFIX;

NAME: [a-zA-Z_][a-zA-Z0-9_]*;
WS: [ \t\r\n]+ -> skip;

```

## Листинг Б.2 – Грамматика языка описания в формате ANTLR 4

```

ioMarkup: (namedStruct)* EOF;
namedStruct: NAME namedStructParameters? struct;
namedStructParameters:
    LPAR
    (parameterDeclaration (COMMA parameterDeclaration)*)?
    RPAR;
parameterDeclaration: NAME COLON namedType;
struct: LBRACE structItem* RBRACE;
structItem:
    conditionalAlternative |
    variableDeclaration |
    ioModifier;
conditionalAlternative:
    IF LPAR p1Expression RPAR struct (ELSE struct)?;
ioModifier: EOLN_MODIFIER SEMICOLON;
variableDeclaration:

```

```

NAME COLON variableType variableConstraint? SEMICOLON;
variableType:
    arrayOfUnnamedStruct |
    (namedType namedTypeParameters? arrayParameters*);
arrayOfUnnamedStruct: ARRAY arrayParameters+ OF struct;
variableConstraint: PIPE plExpression;
namedType: NAME;
namedTypeParameters:
    LPAR (plExpression (COMMA plExpression)*)? RPAR;
arrayParameters:
    LBRACKET
    arrayIterationRange (COMMA SEP ASSIGN sepValue)
    RBRACKET;
arrayIterationRange:
    NAME ASSIGN plExpression DOT DOT plExpression;
sepValue: STRING | CHAR;

```

Листинг Б.3 – Грамматика языка выражений в формате ANTLR 4

```

plExpression:
    plImplLogicalOr (QUESTION plImplLogicalOr COLON
    plImplLogicalOr)?;

plImplLogicalOr:
    plImplLogicalAnd (LOGICAL_OR plImplLogicalAnd)*;
plImplLogicalAnd:
    plImplBitwiseOr (LOGICAL_AND plImplBitwiseOr)*;
plImplBitwiseOr:
    plImplBitwiseXor (PIPE plImplBitwiseXor)*;
plImplBitwiseXor:
    plImplBitwiseAnd (BITWISE_XOR plImplBitwiseAnd)*;
plImplBitwiseAnd:
    plImplEquality (BITWISE_AND plImplEquality)*;
plImplEquality:
    plImplRelational (plImplEqualityOp plImplRelational)*;
plImplRelational:
    plImplBitwiseShift
    (plImplRelationalOp plImplBitwiseShift)?;
plImplBitwiseShift:
    plImplAdditiveBinary
    (plImplBitwiseShiftOp plImplAdditiveBinary)?;
plImplAdditiveBinary:

```

```

    plImplMultiplicativeBinary (plImplAdditiveOp
    plImplMultiplicativeBinary)*;
plImplMultiplicativeBinary:
    plImplPowBinary (plImplMultiplicativeBinaryOp
    plImplPowBinary)*;
plImplPowBinary: plImplPrefixUnary (POW plImplPrefixUnary)*;
plImplPrefixUnary: plImplPrefixUnaryOp* plImplHighestPriority;

plImplHighestPriority:
    (
        (LPAR plExpression RPAR) |
        plFunctionCall |
        plValue |
        plVarBinding
    ) plSubscript*;

plImplEqualityOp: EQUALS | NOT_EQUALS;
plImplRelationalOp: GREATER | LESS | GREATER_EQUAL | LESS_EQUAL;
plImplAdditiveOp: PLUS | MINUS;
plImplMultiplicativeBinaryOp: MULTIPLICATION | DIVISION | MODULO;
plImplPrefixUnaryOp: LOGICAL_NOT | BITWISE_NOT | MINUS;
plImplBitwiseShiftOp: LESS LESS | GREATER GREATER;

plFunctionCall: NAME LPAR plImplFunctionArgs? RPAR;
plImplFunctionArgs: plExpression (COMMA plExpression)*;

plVarBinding: NAME;

plSubscript: LBRACKET plExpression RBRACKET;

plValue: plNumValue | plBoolValue | plCharValue | plStringValue;
plBoolValue: TRUE | FALSE;
plCharValue: CHAR;
plStringValue: STRING;
plNumValue: NUM_VALUE;

```

## Приложение В. ЯЗЫК ЗАПИСИ ВЫРАЖЕНИЙ

Таблица В.1 – Операторы языка выражений в порядке уменьшения приоритета

Приоритет	Нетерминал	Оператор	Описание
1	pImplHighestPriority	f(x)	Вызов функции
1	pImplHighestPriority	x[y]	Взятие по индексу
2	pImplPrefixUnary	!x	Логическое отрицание
2	pImplPrefixUnary	~x	Инверсия битов
2	pImplPrefixUnary	-x	Унарный минус
3	pImplPowBinary	x ** y	Возведение в степень
4	pImplMultiplicativeBinary	x * y	Умножение
4	pImplMultiplicativeBinary	x / y	Деление
4	pImplMultiplicativeBinary	x % y	Остаток от деления
5	pImplAdditiveBinary	x + y	Сложение
5	pImplAdditiveBinary	x - y	Вычитание
6	pImplBitwiseShift	x >> y	Побитовый сдвиг вправо
6	pImplBitwiseShift	x << y	Побитовый сдвиг влево
7	pImplRelationalOp	>, <, >=, <=	Сравнение
8	pImplEquality	x == y	Равенство
8	pImplEquality	x != y	Неравенство
9	pImplBitwiseAnd	x & y	Побитовое «И»
10	pImplBitwiseXor	x ^ y	Побитовое исключающее «ИЛИ»
11	pImplBitwiseOr	x   y	Побитовое «ИЛИ»
12	pImplLogicalAnd	x && y	Конъюнкция
13	pImplLogicalOr	x    y	Дизъюнкция
14	pExpression	x ? y : x	Тернарный оператор

## Приложение Г. СТРУКТУРА ЗАПРОСОВ И ОТВЕТОВ СЕРВИСА ГЕНЕРАЦИИ

Листинг Г.1 – Структура запросов и ответов сервиса генерации в формате Protobuf

```
syntax = "proto2";

enum TargetLanguage {
    CPP = 0;
    JAVA = 1;
    PYTHON = 2;
}

enum TargetComponent {
    SOLUTION = 0;
    GRADER = 1;
    VALIDATOR = 2;
    CHECKER = 3;
}

message TargetFileDescription {
    required TargetComponent target_component = 1;
    required TargetLanguage target_language = 2;
    optional string name = 3;
    optional string previous_content = 4;
}

message ResultFile {
    required TargetComponent component = 1;
    required TargetLanguage language = 2;
    required string name = 3;
    required string content = 4;
    optional string error_message = 5;
}

message GenerateRequest {
    required string io_markup = 1;
    repeated TargetFileDescription target_file_description = 2;
}

message GenerateResponse {
    repeated ResultFile result_file = 1;
    optional string error_message = 2;
}
```

## Приложение Д. ШАБЛОНЫ ГЕНЕРИРУЕМОГО КОДА

Листинги в этом приложении содержат код с незначительными сокращениями.

Листинг Д.1 – Шаблон исходного кода валидатора

```
#include <iostream>
#include <vector>
#include "testlib.h"
namespace iomarkup
{
    int pow(int a, int b)
    {
        if (b == 0) return 1;
        if (b % 2 == 1) return a * pow(a, b - 1);
        return pow(a * a, b / 2);
    }

    long long pow(long long a, long long b)
    {
        if (b == 0) return 1ll;
        if (b % 2 == 1) return a * pow(a, b - 1ll);
        return pow(a * a, b / 2ll);
    }
}
using std::cin;
using std::cout;
using std::vector;

// Structure declarations
%s

input_t input;

// Functions for input
%s

int main(int argc, char *argv[])
{
    registerValidation(argc, argv);
    input = read_input();
    inf.readEoln();
```



```

    inf.readEof();

    // Write code for semantic input validation here.
    return 0;
}

```

Листинг Д.2 – Шаблон исходного кода чекера

```

#include <iostream>
#include <vector>
#include "testlib.h"
namespace iomarkup
{
    int pow(int a, int b)
    {
        if (b == 0) return 1;
        if (b % 2 == 1) return a * pow(a, b - 1);
        return pow(a * a, b / 2);
    }

    long long pow(long long a, long long b)
    {
        if (b == 0) return 1ll;
        if (b % 2 == 1) return a * pow(a, b - 1ll);
        return pow(a * a, b / 2ll);
    }
}
using std::cin;
using std::cout;
using std::vector;

// Structure declarations
%s
input_t input;

// Functions for input
%s

typedef output_t AnsType;
AnsType readAns(InStream& stream)
{
    output_t output = read_output(stream);
}

```

```

    // Write code for semantic check answer here.
    // For example, here you should check a certificate.
    // You should return a comparable representation of answer's
    quality.
    return output;
}
int main(int argc, char *argv[])
{
    registerTestlibCmd(argc, argv);
    input = read_input(inf);
    AnsType pa_answer = readAns(ouf);
    AnsType jury_answer = readAns(ans);
    // Check here that participant have found answer with the same
    quality as jury.

}

```

Листинг Д.3 – Шаблон исходного кода грейдера на языке C++

```

#include <iostream>
#include <vector>
namespace iomarkup
{
    int pow(int a, int b)
    {
        if (b == 0) return 1;
        if (b % 2 == 1) return a * pow(a, b - 1);
        return pow(a * a, b / 2);
    }

    long long pow(long long a, long long b)
    {
        if (b == 0) return 1ll;
        if (b % 2 == 1) return a * pow(a, b - 1ll);
        return pow(a * a, b / 2ll);
    }
}
using std::cin;
using std::cout;
using std::vector;

// Structure declarations

```

```

%s
input_t input;
// Functions for input
%s

// Functions for output
%s

int main(int argc, char *argv[])
{
    input = read_input();
    write_output(solve(input));
    return 0;
}

```

Листинг Д.4 – Шаблон исходного кода грайдера на языке Java

```

public abstract class Grader {
    private static class IoMarkup {
        private IoMarkup() {}
        int pow(int a, int b) {
            if (b == 0) return 1;
            if (b % 2 == 1) return a * pow(a, b - 1);
            return pow(a * a, b / 2);
        }
        long pow(long a, long b) {
            if (b == 0) return 1L;
            if (b % 2 == 1) return a * pow(a, b - 1L);
            return pow(a * a, b / 2L);
        }
    }

    // Structure declarations
    %s

    private final java.util.Scanner in;
    private final java.io.PrintStream out;
    private Input input;

    public Solution(java.util.Scanner in, java.io.PrintStream out)
    {
        this.in = in;
    }
}

```

```

        this.out = out;
    }

    // Input methods
    %s

    // Output methods
    %s

    public abstract Output solve(Input input);

    public static void main(java.lang.String[] args) {
        try (java.util.Scanner inputScanner =
            new java.util.Scanner(java.lang.System.in)) {
            Solution solution = new Solution(
                inputScanner, java.lang.System.out);
            solution.input = solution.readInput();
            solution.writeOutput(solution.solve(solution.input));
        }
    }
}

```

Листинг Д.5 – Шаблон исходного кода грейдера на языке Python

```

import solution

def iomarkup_div(x, y):
    return x / y if isinstance(x, float) or \
        isinstance(y, float) else x // y

class Scanner:
    def __init__(self, next_line_getter):
        self._next_line_getter = next_line_getter
        self._tokens = []
        self._i = 0
        self._j = None

    def _ensure_token(self):
        if self._i >= len(self._tokens):
            self._tokens.extend(
                self._next_line_getter().strip().split())

```

```

def next(self):
    self._ensure_token()
    next_token = self._tokens[self._i]
    if self._j is not None:
        next_token = next_token[self._j:]
    self._i += 1
    self._j = None
    return next_token

def next_int(self):
    return int(self.next())

def next_float(self):
    return float(self.next())

def next_char(self):
    self._ensure_token()
    next_token = self._tokens[self._i]
    if self._j is not None:
        if self._j >= len(next_token):
            self._i += 1
            self._j = None
            return self.next_char()
        next_token = next_token[self._j]
        self._j += 1
    else:
        self._j = 1
        next_token = next_token[0]
    return next_token

def next_bool(self):
    return self.next() == "true"

@staticmethod
def from_input():
    return Scanner(input)

_scanner = Scanner.from_input()

# Structure declarations
%s

```

```
input = None

# Input functions
%s

# Output functions
%s

if __name__ == '__main__':
    input = read_input()
    write_output(solution.solve(input))
```

## Приложение Е. ПРИМЕРЫ ГЕНЕРИРУЕМОГО КОДА

В этом приложении приведены сгенерированные программные компоненты задачи «Копия копии копии» из приложения А. Соответствующее описание входных и выходных данных приведено в разделе 2.2 в листинге 6. Здесь будут приведены части кода до вставки в шаблоны из приложения Д с некоторыми незначительными сокращениями.

Листинг Е.1 – Часть сгенерированного кода валидатора на языке C++

```
input_t read_input();
picture_t read_picture(int n, int m);

input_t read_input()
{
    input_t input;
    input.n = inf.readInt();
    ensuref((3 <= input.n) && (input.n <= 30));
    inf.readSpace();
    input.m = inf.readInt();
    ensuref((3 <= input.m) && (input.m <= 30));
    inf.readSpace();
    input.k = inf.readInt();
    ensuref((0 <= input.k) && (input.k <= 100));
    inf.readEoln();
    inf.readEoln();
    for (int i = 1; i <= (input.k + 1); i++)
    {
        if (i != 1)
        {
            inf.readEoln();
            inf.readEoln();
        }
        picture_t tmp;
        tmp = read_picture(input.n, input.m);
        input.pictures.push_back(tmp);
    }
    return input;
}

picture_t read_picture(int n, int m)
{
```

```

picture_t picture;
for (int i = 1; i <= n; i++)
{
    if (i != 1)
    {
        inf.readEoln();
    }
    std::string tmp;
    tmp = inf.readToken();
    picture.s.push_back(tmp);
    ensuref(picture.s[i - 1].size() == m);
}
return picture;
}

```

Листинг E.2 – Часть сгенерированного кода чекера на языке C++

```

output_t read_output(InStream& stream);
input_t read_input(InStream& stream);
picture_t read_picture(InStream& stream, int n, int m);
ops_t read_ops(InStream& stream);

output_t read_output(InStream& stream)
{
    output_t output;
    output.startPicIndex = stream.readInt();
    stream.ensuref(1 <= output.startPicIndex &&
        output.startPicIndex <= input.k + 1);
    output.q = stream.readInt();
    stream.ensuref(0 <= output.q &&
        output.q <= iomarkup::pow(10, 7));
    for (int j = 1; j <= output.q; j++)
    {
        ops_t tmp;
        tmp = read_ops(stream);
        output.ops.push_back(tmp);
    }
    return output;
}

input_t read_input(InStream& stream)
{

```



```

input_t input;
input.n = stream.readInt();
stream.ensuref(3 <= input.n && input.n <= 30);
input.m = stream.readInt();
stream.ensuref(3 <= input.m && input.m <= 30);
input.k = stream.readInt();
stream.ensuref(0 <= input.k && input.k <= 100);
for (int i = 1; i <= (input.k + 1); i++)
{
    picture_t tmp;
    tmp = read_picture(stream, input.n, input.m);
    input.pictures.push_back(tmp);
}
return input;
}

picture_t read_picture(InStream& stream, int n, int m)
{
    picture_t picture;
    for (int i = 1; i <= n; i++)
    {
        std::string tmp;
        tmp = stream.readToken();
        picture.s.push_back(tmp);
        stream.ensuref(picture.s[i - 1].size() == m);
    }
    return picture;
}

ops_t read_ops(InStream& stream)
{
    ops_t ops;
    ops.op = stream.readInt();
    if (ops.op == 1)
    {
        ops.x = stream.readInt();
        stream.ensuref((2 <= ops.x) && (ops.x <= input.n - 1));
        ops.y = stream.readInt();
        stream.ensuref((2 <= ops.y) && (ops.y <= input.m - 1));
    }
    else

```

```

{
    ops.i = stream.readInt();
    stream.ensuref((1 <= ops.i) && (ops.i <= input.k + 1));
}
return ops;
}

```

Листинг E.3 – Часть сгенерированного кода грейдера на языке C++

```

input_t read_input();
picture_t read_picture(int n, int m);

output_t read_output()
{
    output_t output;
    cin >> output.startPicIndex;
    cin >> output.q;
    for (int j = 1; j <= output.q; j++)
    {
        ops_t tmp;
        tmp = read_ops();
        output.ops.push_back(tmp);
    }
    return output;
}

input_t read_input()
{
    input_t input;
    cin >> input.n;
    cin >> input.m;
    cin >> input.k;
    for (int i = 1; i <= (input.k + 1); i++)
    {
        picture_t tmp;
        tmp = read_picture(input.n, input.m);
        input.pictures.push_back(tmp);
    }
    return input;
}

picture_t read_picture(int n, int m)

```

```

{
    picture_t picture;
    for (int i = 1; i <= n; i++)
    {
        std::string tmp;
        cin >> tmp;
        picture.s.push_back(tmp);
    }
    return picture;
}

void write_output(output_t const& output);
void write_ops(ops_t const& ops);

void write_output(output_t const& output)
{
    cout << output.startPicIndex;
    cout << endl;
    cout << output.q;
    cout << endl;
    for (int j = 1; j <= output.q; j++)
    {
        write_ops(output.ops[j - 1]);
        cout << endl;
    }
}

void write_ops(ops_t const& ops)
{
    cout << ops.op;
    cout << ' ';
    if (ops.op == 1)
    {
        cout << ops.x;
        cout << ' ';
        cout << ops.y;
    }
    else
    {
        cout << ops.i;
    }
}

```

```
}
```

Листинг Е.4 – Часть сгенерированного кода грейдера на языке Java

```
private static class Ops {
    private Integer op;
    private Integer x;
    private Integer y;
    private Integer i;
};

private static class Picture {
    private java.util.List<java.lang.String> s =
        new java.util.ArrayList<>();
};

private static class Input {
    private Integer n;
    private Integer m;
    private Integer k;
    private java.util.List<Picture> pictures =
        new java.util.ArrayList<>();
};

private static class Output {
    private Integer startPicIndex;
    private Integer q;
    private java.util.List<Ops> ops = new java.util.ArrayList<>();
};

private Input readInput(){
    Input input = new Input();
    input.n = in.nextInt();
    input.m = in.nextInt();
    input.k = in.nextInt();
    for (Integer i = 1; i <= input.k + 1; i++) {
        Picture tmp;
        tmp = readPicture(input.n, input.m);
        input.pictures.add(tmp);
    }
    return input;
}
```

```

private Picture readPicture(Integer n, Integer m){
    Picture picture = new Picture();
    for (Integer i = 1; i <= n; i++) {
        java.lang.String tmp;
        tmp = in.next();
        picture.s.add(tmp);
    }
    return picture;
}

```

```

private void writeOutput(Output output) {
    out.print(output.startPicIndex);
    out.println();
    out.print(output.q);
    out.println();
    for (Integer j = 1; j <= output.q; j++) {
        writeOps(output.ops.get(j - 1));
        out.println();
    }
}

```

```

private void writeOps(Ops ops){
    out.print(ops.op);
    out.print(' ');
    if (ops.op == 1) {
        out.print(ops.x);
        out.print(' ');
        out.print(ops.y);
    } else {
        out.print(ops.i);
    }
}

```

Листинг Е.5 – Часть сгенерированного кода грейдера на языке Python

```

class Ops:
    def __init__(self):
        self.op = None
        self.x = None
        self.y = None
        self.i = None

```

```

class Picture:
    def __init__(self):
        self.s = []

class Input:
    def __init__(self):
        self.n = None
        self.m = None
        self.k = None
        self.pictures = []

class Output:
    def __init__(self):
        self.startPicIndex = None
        self.q = None
        self.ops = []

def read_input():
    input = Input()
    input.n = _scanner.next_int()
    input.m = _scanner.next_int()
    input.k = _scanner.next_int()
    for i in range(1, (input.k + 1) + 1):
        tmp = read_picture(input.n, input.m)
        input.pictures.append(tmp)
    return input

def read_picture(n, m):
    picture = Picture()
    for i in range(1, (n) + 1):
        tmp = _scanner.next()
        picture.s.append(tmp)
    return picture

def write_output(output):
    print(output.startPicIndex, end='')
    print()
    print(output.q, end='')
    print()
    for j in range(1, (output.q) + 1):

```

```
        write_ops(output.ops[(j) - (1)])
    print()

def write_ops(ops):
    print(ops.op, end='')
    print(end=' ')
    if ops.op == 1:
        print(ops.x, end='')
        print(end=' ')
        print(ops.y, end='')
    else:
        print(ops.i, end='')

```