Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования «Нижегородский государственный университет  
 им. Н.И. Лобачевского»

Институт информационных технологий, механики и математики

**Отчет по лабораторной работе**

**АЛГОРИТМЫ ПОИСКА НАИЛУЧШЕГО ХОДА   
В ИГРЕ«РУССКИЕ ШАШКИ»**

**Выполнили**:

студент группы 0823-1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Грачева Е. А.

студент группы 0826-1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Панов А. А.

**Руководитель**:

доц. каф. МОСТ ИИТММ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Мееров И.Б.

Нижний Новгород

2017

СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 2](#_Toc482532304)

[ВВЕДЕНИЕ 3](#_Toc482532305)

[1 Постановка задачи 4](#_Toc482532306)

[2 Обзор методов поиска лучшего хода 5](#_Toc482532307)

[2.1 Алгоритм полного перебора 5](#_Toc482532308)

[2.2 Оценочная функция 7](#_Toc482532309)

[2.3 Поиск с отсечениями 9](#_Toc482532310)

[2.4 Форсированные варианты 11](#_Toc482532311)

[3 Программная реализация 13](#_Toc482532312)

[3.1 Алгоритмическое ядро (С++) 14](#_Toc482532313)

[3.1.1 Описание структур данных 14](#_Toc482532314)

[3.1.2 Описание алгоритмов 16](#_Toc482532315)

[3.2 Графический интерфейс (C#) 18](#_Toc482532316)

[4 Апробация 19](#_Toc482532317)

[5 Сбор статистических данных 22](#_Toc482532318)

[6 Дальнейшие пути развития 23](#_Toc482532319)

[ЗАКЛЮЧЕНИЕ 23](#_Toc482532320)

[ЛИТЕРАТУРА 23](#_Toc482532321)

[ПРИЛОЖЕНИЕ. Фрагменты исходного кода программы 23](#_Toc482532322)

ВВЕДЕНИЕ

В настоящее время в программировании популярно направление, связанное с компьютеризацией различных логических игр. Мотивируется это тем, что, во-первых, компьютерные игры достаточно востребованы; во-вторых, логические игры представляют собой хороший материал для исследования различных алгоритмов поиска наилучшего хода.

# Постановка задачи

Были поставлены следующие задачи:

* исследовать различные алгоритмы поиска лучшего хода в игровой программе;
* реализовать логику игры "Русские шашки";
* создать виртуального игрока, способного оценивать ситуацию на доске и определять наилучший ход;
* реализовать пользовательский интерфейс игры;
* оценить качество игры бота в сравнении с другими известными игровыми программами, такими как "Тундра" или "Аврора".

# Обзор методов поиска лучшего хода

Представим, что у нас есть компьютер с практически бесконечными вычислительными ресурсами. Определим функцию ***f:P→Z***, которая ставит в соответствие позиции **pos** из множества всевозможных позиций ***P*** целое число, отражающее «выгодность» этой позиции для текущего игрока***.*** Такая функция называется *оценочной функцией*[1]*.* На подобном компьютере мы могли бы вычислить ее следующим образом:

* присвоим всем финальным позициям значения –1, 0, 1, в зависимости от исхода игры (проигрыш, ничья, победа);
* применим рекурсивное правило где **pos** соответствует ходу текущего игрока, **pos‘** соответствует ходу противника, а  **pos→pos‘**обозначает все допустимые ходы из позиции **pos**.

Чем больше значение функции, тем выгодней позиция. Таким образом, ставя каждому ходу в соответствие его оценку, мы бы могли однозначно определить лучший ход.

Количество [возможных позиций](http://en.wikipedia.org/wiki/Shannon_number) в шашках составляет около 5∙1020. Вычисления такого масштаба выполнить практически невозможно (канадским ученым потребовалось 200 компьютеров и 20 лет вычислений)[2]. Следовательно, приходится искать другие пути для нахождения лучшего хода, которые, возможно, не будут такими точными, но зато будут не столь затратны по времени.

## Алгоритм полного перебора

Самым простым способом получить если не лучший, то хотя бы осмысленный ход, является метод полного перебора с возвратом в соединении с рекурсией. Если сгенерировать все возможные ходы из данной позиции для белых, затем для каждого такого хода повторить те же действия для черных, потом снова для белых и т.д., получится простой и понятный рекурсивный алгоритм, основанный на переборе позиций. Пример дерева игры, возникающего таким образом, представлен на рис.1.

…

c5-a3

…

…

…

…

…

c7-c3

c5-a7

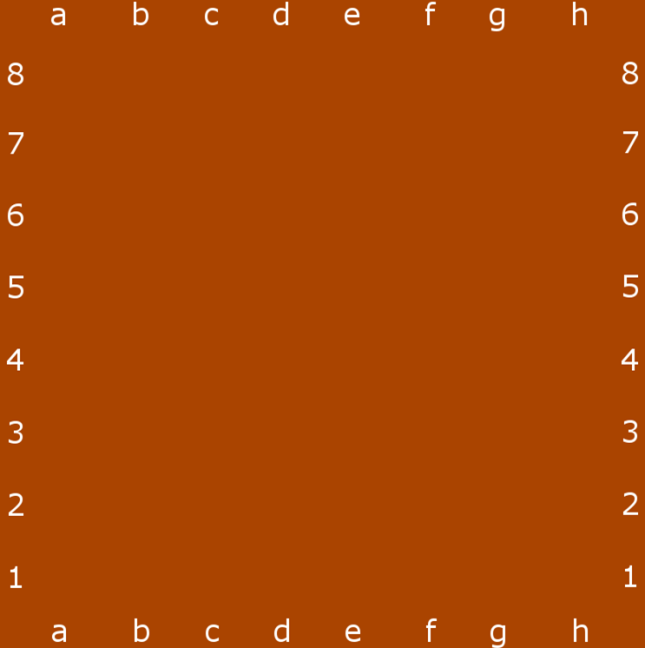


Рисунок . Пример дерева игры. а – позиция на доске, б – построенное для нее дерево

а) б)

Однако, как уже было сказано, рассмотреть все возможные варианты до конца невозможно, поэтому приходится ограничивать глубину рекурсии. Пусть на каждом шаге известна текущая глубина рекурсии – параметр ***depth***, первоначально равный максимальной заданной глубине. Тогда если ***depth***≠0, то рекурсия будет вызываться с параметром ***depth-1***, в противном случае вычисления следует прекратить и вызвать *оценочную функцию[[1]](#footnote-2)*.

Блок-схема алгоритма полного перебора представлена на рис.2.

depth=0?

да

нет

Вызови оценочную функцию

score:=tmp

score:=-INFINITY

Есть еще ходы?

нет

да

Сгенерируй все ходы

Сделай ход

tmp:=-Search(depth-1)

Отмени ход

score<tmp?

нет

да

Конец

Рисунок . Блок-схема алгоритма полного перебора

## Оценочная функция

Пусть P– множество всевозможных позиций на доске. Как уже было сказано, функция ***f:P→Z***, ставящая в соответствие некоторой позиции из множества ***P*** целое число, отражающее «выгодность» этой позиции для текущего игрока, называется *оценочной функцией*.

Простейшая оценочная функция считает шашки на доске, т.е. суммирует вес всех белых шашек и вычитает из полученного результата сумму всех черных шашек. Если шашек поровну, функция вернет 0.

Установим вес шашек следующим образом:

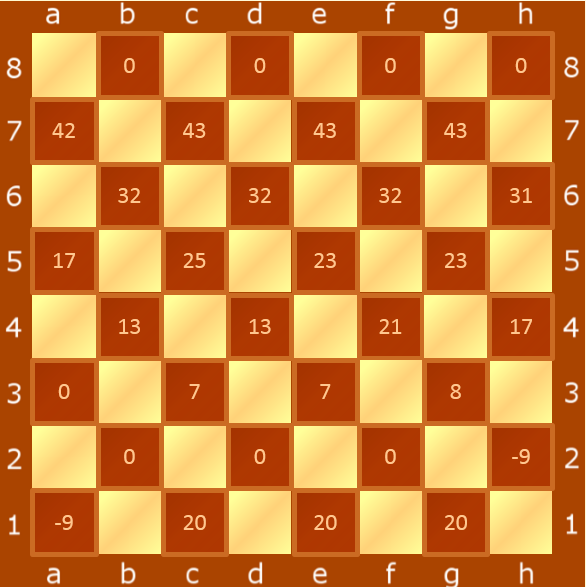
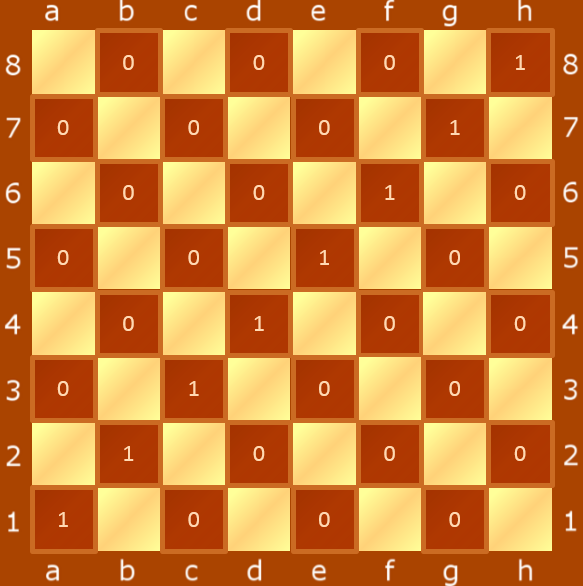
* простая шашка – 50;
* дамка – 350.

Таким образом, простая оценочная функция при расстановке шашек как на рис.1а для белого игрока вернет число 250, для черного игрока -250.

Более сложная оценочная функция учитывает ценность полей доски, которая определяется следующими принципами:

* надо стараться по возможности подвигаться вперед, т.к. чем ближе к последней горизонтали, тем сила полей больше;
* поскольку ценность шашек a1 и h2 является наименьшей, то их надо по возможности быстро ввести в игру;
* учитывая, что с полей c5 и f4 удобно препятствовать развитию сил противника, следует бороться за овладением центральными полями c5 и f4, стремиться к захвату центра;
* следует придерживаться принципа пропорционального, равномерного распределения шашек по обоим флангам;
* шашки с полей c1, e1, g1 лучше без особой надобности не сдвигать, чтобы не открывать поля последнего ряда;
* дамке по возможности стоит контролировать главную диагональ, т.к. она будет препятствовать продвижению противника в дамки.

Согласно этим правилам, ценность полей доски была обозначена следующим образом (рис.3):



а) б)

Рисунок . Оценка клеток доски. а – для белой шашки, б – для дамки

На практике оказалось, что компьютерный игрок, использующий оценочную функцию, считающую просто материал, проигрывает компьютерному игроку, использующему более сложную оценочную функцию, что и следовало ожидать.

## Поиск с отсечениями

Рассмотрим основной алгоритм оптимизации перебора –­­ алгоритм *alpha-beta*. Суть его в том, что для получения оценки такой же точности, как и при полном переборе, совершенно не обязательно просматривать все варианты. Для определения отсекаемых вариантов не требуется знать особенностей данной игры.

В рекурсивной функции мы должны ввести две новые переменные — максимум для белых (alpha) и максимум для черных (beta). При начальном вызове обе эти величины равны минимально возможному значению. Если в какой-то позиции, например, для белых, мы получили результат, превышающий максимум для белых, достигнутый до этого, мы увеличиваем этот максимум.

Теперь рассмотрим некоторую строку игры. Представим, что где-то в глубине дерева перебора мы обнаружили, что максимумы черных и белых сравнялись. Допустим, мы просчитываем позицию для черных. Если мы продолжим перебирать в данной позиции, то максимум для черных может еще увеличиться, а может остаться прежним, но он уже сравнялся с максимумом белых. Это значит, что когда программа поднимется на уровень выше (рекурсивно), результат не будет записан, т. к. он не превышает максимума для белых в этой позиции, т.е. в данной позиции для черных мы можем прекратить перебор и вернуть полученный результат досрочно, поскольку дальше считать нет смысла[3].

Проиллюстрируем вышесказанное на примере (рис.4). Пусть на каком-либо шаге рекурсии максимум для белых (alpha) равен 4. Сделаем какой-либо ход и просчитаем возможные ходы для черных (опустимся на уровень ниже). Пусть при дальнейших вычислениях для одного из ходов результат равен 1 (beta=1). Если впоследствии найдется ход, оценка которого будет меньше 1, то он будет записан в beta, но не будет записан в alpha, и наоборот, если найдется ход, оценка которого будет больше 1, то он не будет записан даже в beta, следовательно, и не будет записан в alpha. Получается, дальнейшие вычисления бессмысленны.

β=1<α

4

Рисунок . Пример работы алгоритма alpha-beta

Блок-схема алгоритма поиска с отсечениями представлена на рис.5.

Рисунок . Блок-схема алгоритма поиска с отсечениями

depth=0?

да

нет

Вызови оценочную функцию

alpha:=tmp

Есть еще ходы?

нет

да

Сгенерируй все ходы

Сделай ход

tmp:=-AlphaBeta(depth-1, -beta, -alpha)

Отмени ход

alpha<tmp?

нет

да

Конец

alpha>=beta?

нет

верни alpha

да

При сборе статистики[[2]](#footnote-3) выяснилось, что при использовании алгоритма *alpha-beta* мы просматриваем примерно в 10 раз меньше позиций, чем при обычном поиске.

## Форсированные варианты

В некоторых ситуациях, например, в случае размена, прекращение вычислений по достижении максимальной глубины рекурсии может привести к крайне неверной оценке позиции (*эффект горизонта*). Рассмотрим пример (рис.6). Пусть максимальная глубина рекурсии равна 2. Ход белых. Даже не очень опытному игроку очевидно, что лучший ход – *d8-h8*, поскольку он приводит к полному уничтожению противника. Однако бот не увидит преимуществ такого хода и сходит *h2-b8*. Чтобы избежать подобных ситуаций, отдельные ветки стоит просчитывать на большую глубину.

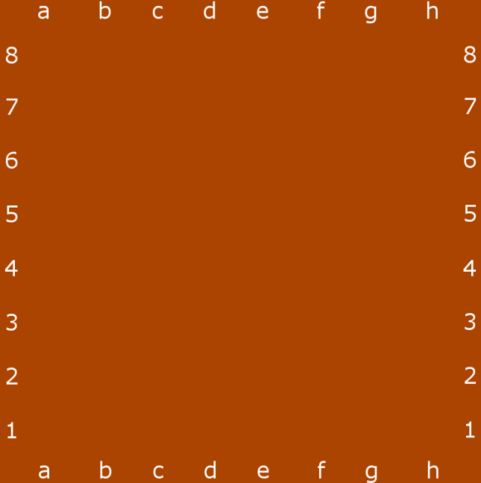


Рисунок . Пример позиции, для которой необходимо форсирование

В шашках форсированными вариантами являются взятия. Поэтому в конце каждой ветки рекурсии вызывается упрощенная функция поиска, рассматривающая только взятия. По завершению размена вызывается оценочная функция. Если же взятий не было, оценочная функция вызывается сразу.

# Программная реализация

Проект состоит из двух частей — вычислительного ядра и графического интерфейса (рис.7). В вычислительной части написаны различные функции поиска лучшего хода, то есть искусственный интеллект для бота. Графическая оболочка предоставляет визуализацию игры, предоставляет настройки компьютерного игрока и дает возможность ходить реальному игроку.

Вычислительное ядро

Графическая оболочка

позиция

лучший ход

Рисунок . Схема взаимодействия графической оболочки и вычислительного ядра

Нахождение наилучшего хода требует большого объема вычислений и для обеспечения наилучшей производительности вычислительное ядро написано в виде динамической библиотеки на C++.

Графическая оболочка, написана на C#, так как данный язык предоставляет удобный графический инструментарий.

Итоговое приложение отображает игру с помощью графической оболочки и находит лучший ход с помощью вычислительного ядра.

## Алгоритмическое ядро (С++)

### Описание структур данных

Далее приведены основные классы, необходимые для организации компьютерного игрока.

#### Class Checker

Класс *Checker* описывает структуру хранения шашки. В нем кроме идентификатора шашки, представляющего собой целое число от 1 до 12, присутствуют еще 2 типа полей. Первый тип описывает основные признаки шашки, такие как:

* цвет;
* тип (шашка или дамка);
* координаты клетки, на которой стоит шашка.

Второй тип полей класса необходим для организации двухсвязного списка на массиве[[3]](#footnote-4). В эту группу входят следующие поля:

* номер следующей по списку шашки;
* номер предыдущей по списку шашки.

#### Class ListOfCheckers

Класс *ListOfCheckers* является типом, представляющим собой список на массиве. Необходимость его создания обусловлена тем, что в программе необходимо как обращаться к шашке по индексу, так и часто добавлять и удалять шашки. Единственное поле класса – массив *Checker[[4]](#footnote-5)* на 13 элементов (12 шашек + голова). Благодаря особой структуре класса *Checker* возможно организовать список.

Также для удобства был создан класс *LChIterator*, представляющий собой итератор для класса *ListOfCheckers*.

#### Class Board

Доска представляет собой одномерный массив из указателей на *Checker*. Это необходимо для того, чтобы была возможность легко находить шашку, зная ее координаты.

#### Class Move

При реализации алгоритмов появляется необходимость хранить ход шашки. Данный класс предоставляет все необходимые для этого поля и методы. Ход характеризуют следующие признаки:

* идентификатор шашки, которая ходит;
* первоначальные координаты этой шашки;
* координаты шашки после того, как был сделан ход;
* съеденные шашки;
* стала ли шашка дамкой в процессе хода.

#### Class Cache

На каждом шаге рекурсии необходимо хранить возможные ходы. Можно было бы создавать на каждом шаге рекурсии массив ходов, но, во-первых, это может привести к дополнительным накладным расходам, во-вторых, выделять место пришлось бы с большим запасом, поскольку неизвестно, сколько ходов может быть сгенерировано для определенной позиции. Поэтому мы выделяем место для хранения ходов в виде большого массива один раз в самом начале, а во время работы рекурсии записываем туда ходы. Очищение массива происходит посредством перемещения указателя на конец массива в сохраненную точку. Данный массив представлен классом *Cache*.

Сбор статистики[[5]](#footnote-6) позволяет примерно оценить размеры данного массива. Так при глубине вычислений 12 его размер приблизительно равен … элементов.

### Описание алгоритмов

Основные алгоритмы, используемые в программе (полный перебор[[6]](#footnote-7), перебор с отсечениями[[7]](#footnote-8), форсирование вариантов[[8]](#footnote-9), использование различных оценочных функций[[9]](#footnote-10)) описаны в соответствующем разделе[[10]](#footnote-11). Здесь же мы представим вспомогательные алгоритмы, необходимые для реализации поиска лучшего хода.

Реализация данных алгоритмов представлена в приложении …

#### Генерация всевозможных ходов

Генерация ходов для позиции в шашках не является тривиальной задачей, поскольку, во-первых, возможны взятия сразу нескольких шашек, во-вторых, передвижения дамки (особенно с взятиями) требуют внимательного изучения правил игры. Алгоритм взятий для шашек можно описать следующим образом:

1. исследуем все четыре направления (по диагонали влево-вверх, вправо-вверх, вправо-вниз и влево-вниз) на наличие взятий (рассматриваются только соседние клетки);
2. если возможно взятие, «перепрыгнем» через битую шашку и поменяем ее цвет на противоположный (это необходимо, поскольку брать шашку дважды и убирать ее с доски до окончания хода нельзя по правилам), иначе завершим поиск взятий;
3. если текущая позиция находится на последнем ряду, изменим тип шашки на дамку и применим алгоритм взятий для дамки, начиная с шага 3 (см. далее);
4. в противном случае исследуем все вышеперечисленные направления, кроме того, откуда пришла шашка, на наличие взятий;
5. повторим алгоритм, начиная с шага 2.

Алгоритм взятий для дамок немного сложнее:

1. исследуем все четыре направления на наличие взятий (рассматриваются не только соседние клетки);
2. если возможно взятие, «перепрыгнем» через битую шашку (поменяв ее цвет) и поставим дамку на первую свободную клетку, иначе завершим поиск взятий;
3. исследуем направления, перпендикулярные направлению, с которого пришла дамка, на наличие взятий;
4. если в этих направлениях есть взятия, повторим алгоритм, начиная с шага 2, и вернемся в эту же точку;
5. в противном случае передвинем шашку на 1 клетку в первоначальном направлении (если это невозможно, то заверши поиск взятий);
6. повторим алгоритм, начиная с 2, до тех пор, пока впереди не возникнет преграда (шашка или граница доски);
7. если впереди возможно взятие, повторим алгоритм, начиная с шага.

Функция поиска ходов[[11]](#footnote-12) выполняет следующий алгоритм:

1. сохраним текущее состояние массива ходов[[12]](#footnote-13);
2. рассмотрим шашку текущего игрока;
3. проверим, может ли она взять какую-нибудь шашку противника;
4. если нет, то вызовем функцию поиска хода для шашки (учитывая, что она может быть как простой шашкой, так и дамкой) и повторим алгоритм, начиная с шага 2;
5. иначе вернем массив ходов в первоначальное состояние;
6. выполним функцию поиска взятий для шашки (учитывая ее тип);
7. для следующей шашки повторим алгоритм, начиная с шага 6;

#### Совершение и отмена хода

Также для выполнения рекурсивного поиска необходимо уметь делать ход и отменять его. Ход совершается по следующему алгоритму:

1. удалим все съеденные шашки с доски;
2. удалим основную шашку с доски;
3. поставим шашку на новое место;
4. если необходимо, изменим тип шашки.

Отмена хода выполняется в обратном порядке.

## Графический интерфейс (C#)

Для удобства графическая часть выполнена по схеме MVC (Module, View, Controller) (рис.8):

View

Controller

Model

I’m update!

User’s change

Change data

Рисунок . Схема MVC

# Руководство пользователя

…немного рекламы…

Приложение «Русские шашки» позволяет играть в следующих нижеперечисленных режимах.

Скриншоты игры представлены в приложении …

## Bot VS Bot

Данный режим игры позволяет тестировать эффективность различных алгоритмов поиска лучшего хода. Приложение позволяет настраивать различные параметры игры двух компьютерных игроков[[13]](#footnote-14) и запускать их в качестве противников.

## One Player

Режим игры «один игрок» предусмотрен для тех пользователей, которые жалуются на отсутствие противника по силам. Игрок может настраивать способности компьютерного игрока желаемым образом[[14]](#footnote-15). Сложность может варьироваться от самой простой (противник ходит практически случайным образом) до наибольшей, соответствующей максимальным возможностям программы.

## Two Players

Режим «два игрока» позволяет пользователям играть друг с другом с одного ПК.

## Constructor

Конструктор предназначен для генерации различных первоначальных позиций. Он дает возможность тестировать компьютерного игрока, решать различные логические задачи и просто начинать игру не со стандартной расстановки шашек. Интерфейс, используемый в данном режиме игры, достаточно естественный и не требует пояснений.

## Настройки игры

Окно настроек вызывается каждый раз перед началом игры, кроме случая, когда выбран режим «два игрока». Оно позволяет выбирать цвет шашек для игроков, в режиме «конструктор» определять, кем является каждый из игроков – компьютером или человеком, и настраивать сложность компьютерного игрока. Компьютерный игрок характеризуется следующими критериями:

* глубиной вычислений (чем больше, тем сложнее);
* видом поиска (обычный поиск, поиск с отсечениями, форсирование[[15]](#footnote-16));
* типом оценочной функции (простая или сложная[[16]](#footnote-17)).

Пример окна настроек для режима «конструктор» представлен в приложении ….

# Апробация

# Сбор статистических данных

# Дальнейшие пути развития

Дальнейшая разработка программы будет продолжена в нескольких направлениях. Во-первых, будут совершенствоваться алгоритмы. Предполагаются следующие улучшения:

* создание базы дебютов;
* добавление различных алгоритмов для ведения игры на стадии эндшпиля (три дамки против одной и др.);
* добавление возможность сведения игры вничью;
* отмена циклических повторов позиций;
* и др.

Во-вторых, предусматриваются следующие нововведения:

* добавление возможности играть онлайн с другого ПК;
* создание приложения «Русские шашки» для Android;
* усовершенствование интерфейса (добавление базы позиций, анализ проведенной игры и многое другое).

ЗАКЛЮЧЕНИЕ

В данной лабораторной работе вычислялись значения арифметических выражений. Для этого выражения разбивались на *лексемы* и проверялись на *корректность* ввода с помощью простейшего конечного автомата. Затем выполнялась необходимая сортировка лексем и итоговое вычисление.

ЛИТЕРАТУРА

* Гергель В. ШАГА, Лабутина А. А. Учебно-образовательный комплекс по методам программирования //Нижний Новгород: ННГУ им. НИ Лобачевского. – 2007.

ПРИЛОЖЕНИЕ. Фрагменты исходного кода программы

1. См. п.2.2 [↑](#footnote-ref-2)
2. См. п.6 [↑](#footnote-ref-3)
3. См. п.3.1.1.2 [↑](#footnote-ref-4)
4. См. п.3.1.1.1 [↑](#footnote-ref-5)
5. См. п.6 [↑](#footnote-ref-6)
6. См. п.2.1 [↑](#footnote-ref-7)
7. См. п.2.2 [↑](#footnote-ref-8)
8. См. п.2.3 [↑](#footnote-ref-9)
9. См. п.2.4 [↑](#footnote-ref-10)
10. См. п.2 [↑](#footnote-ref-11)
11. Реализация см. Приложение… [↑](#footnote-ref-12)
12. См. п.3.1.1.5 [↑](#footnote-ref-13)
13. См. п.4.5 [↑](#footnote-ref-14)
14. См. п.4.5 [↑](#footnote-ref-15)
15. См. п.2.1 – 2.3 [↑](#footnote-ref-16)
16. См. п.2.4 [↑](#footnote-ref-17)