Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

«Синтез исправлений для неверных решений олимпиадных задач по программированию»

Автор: Шовкопляс Григорий Филиппович _____
Направление подготовки (специальность): 01.03.02 Прикладная математика и информатика
Квалификация: Бакалавр
Руководитель: Буздалов М.В., канд. техн. наук _____

К защите допустить
Зав. кафедрой Васильев В.Н., докт. техн. наук, проф. _____

«___» _____ 20___ г.

Санкт-Петербург, 2017 г.

Студент Шовкопляс Г.Ф. **Группа** М3439 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования

Направленность (профиль), специализация Математические модели и алгоритмы разработки программного обеспечения

Квалификационная работа выполнена с оценкой _____

Дата защиты «___» _____ 20___ г.

Секретарь ГЭК _____ Принято: «___» _____ 20___ г.

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

УТВЕРЖДАЮ
Зав. каф. компьютерных технологий
докт. техн. наук, проф.
_____ Васильев В.Н.
«___» _____ 20___ г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Студент Шовкопляс Г.Ф. **Группа** М3439 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования **Руководитель** Буздалов Максим Викторович, канд. техн. наук, доцент кафедры КТ Университета ИТМО

1 Наименование темы: Синтез исправлений для неверных решений олимпиадных задач по программированию

Направление подготовки (специальность): 01.03.02 Прикладная математика и информатика

Направленность (профиль): Математические модели и алгоритмы разработки программного обеспечения

Квалификация: Бакалавр

2 Срок сдачи студентом законченной работы: «31» мая 2017 г.

3 Техническое задание и исходные данные к работе.

Требуется разработать способ анализа кода программ для решения олимпиадных задач, с целью применения автоматического исправления ошибок для повышения продуктивности обучения школьников.

4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)

Пояснительная записка должна демонстрировать актуальность данной задачи и подход к решению. Должна быть проведена оценка эффективности, а также оценены перспективы развития.

5 Перечень графического материала (с указанием обязательного материала)

Не предусмотрено

6 Исходные материалы и пособия

а) ANTLR 4 Documentation;
б) Томас Кормен и др. Алгоритмы: построение и анализ;
в) Earl T. Barr, Mark Harman и др. Automated Software Transplantation.

7 Календарный план

№№ пп.	Наименование этапов выпускной квалификационной работы	Срок выполнения этапов работы	Отметка о выполнении, подпись руков.
1	Ознакомление с областью задачи, поиск существующих решений	10.2017	
2	Разработка алгоритма для решения задачи	11.2017	
3	Реализация алгоритма	02.2017	
4	Тестирование и оценка эффективности	04.2017	
5	Написание пояснительной записки	05.2017	

8 Дата выдачи задания: «01» сентября 2017 г.

Руководитель _____

Задание принял к исполнению _____ «01» сентября 2017 г.

Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Студент: Шовкопляс Григорий Филиппович

Наименование темы работы: Синтез исправлений для неверных решений олимпиадных задач по программированию

Наименование организации, где выполнена работа: Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования: Исследование возможности синтеза исправлений для неверных решений олимпиадных задач по программированию.

2 Задачи, решаемые в работе:

а) Провести анализ актуальности;

б) Разработать алгоритм для решения;

в) Оценить эффективность алгоритма и возможности его улучшения.

3 Число источников, использованных при составлении обзора: _____

4 Полное число источников, использованных в работе: 2

5 В том числе источников по годам

Отечественных			Иностранных		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет

6 Использование информационных ресурсов Internet: _____

7 Использование современных пакетов компьютерных программ и технологий: Был использованы языки программирования Java 1.8, Python 3 и генератор ситаксических анализаторов ANTLR. Для Java была добавлена библиотека StructureGraphic для визуализации деревьев, а также библиотека для работы с ANTLR. Для разработки кода использовалась среда Intellij IDEA с плагином для запуска ANTLR для написания кода на Java и FAR manager для написания кода на Python.

8 Краткая характеристика полученных результатов: Реализованный метод синтеза покрывает достаточно большое количество типичных ошибок, а при дальнейшей доработке способен покрывать абсолютное большинство.

9 Гранты, полученные при выполнении работы:

10 Наличие публикаций и выступлений на конференциях по теме работы:

Выпусник: Шовкопляс Г.Ф. _____

Руководитель: Буздалов М.В. _____

«___» _____ 20___ г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время популярны и продолжают набирать популярность различные олимпиады по программированию. Особенностью данных олимпиад является использование автоматической системы тестирования, которая проверяет программу участника на заранее подготовленных тестах.

Ведут работу и крайне востребованы различные кружки, онлайн-курсы и другие обучающие занятия в данной области. Преподаватели постоянно ищут ошибки в программах учащихся в процессе обучения. Не всегда найти ошибку просто, особенно небольшую.

При этом одну и ту же задачу решает множество различных школьников, в разные моменты времени, но все посылки храняться на сервере с тестирующей системой в данном случае, в отличии от, например, математических задач, которые чаще всего проверяются устно, либо письменно, но на бумаге, которую не хранят.

В любой области обучения можно столкнуться с ошибками, которые совершают множество людей при обучении, в математике, например, это потеря корней при раскрытие модуля или деления на зависимую переменную. Программирование не является исключением, и ошибки при решении задач тоже повторяются, однако, как описано выше в рамках одной тестирующей системы хранится архив всех попыток сдач. Данную информацию как раз и хочется использовать, для повышения эффективности преподавания.

В данной работе будет рассмотрен способ синтеза исправлений для решения учащегося. Это поможет в некоторой степени автоматизировать процесс обучения, а следовательно снять не самую полезную нагрузку с преподавателя. Ведь, если в случае с локальным кружком на одного преподавателя приходится не более двадцати учащихся, но существуют онлайн-курсы, в которых количество учащихся в разы больше. На заочном кружке Петра Калинина в год написания работы обучалось 150 человек на одного преподавателя. Но эти цифры меркнут на фоне количества учащихся на онлайн-курсе Максима Буздалова «How to win coding competitions: secrets of champions» за предыдущие два года сум-

марно достигло 60 000 человек. При этом, даже на такое огромное количество человек приходится всего один преподаватель.

В **первой** главе вводятся основные определения и термины, используемые в работе и необходимые для понимания других определений и терминов. Также формулируется цель исследования, на основе которой приводится обзор работ из смежных областей, после чего окончательно формулируется задача исследования.

В **второй** главе описывается способ анализа текста программы, метод поиска исправления, а также сформулирован критерий «похожести», на основе которого разработан алгоритм применения исправления.

В **третьей** главе описываются результаты тестирования структуры данных для хранения дерева разбора, на искусственных и естественных данных и показана эффективность разработаного метода.

ГЛАВА 1. ОБЗОР ОБЛАСТИ, ПОСТАНОВКА ЗАДАЧИ

1.1. Термины и понятия

В данном разделе описаны основные термины и понятия, используемые в представленной работе.

1.1.1. Олимпиадное программирование

Задача олимпиадного программирования представляет из себя некоторое задание, которое требуется выполнить, написав программу на одном из языков программирования. Чтобы проверить корректность выполнения задания, текст программы отправляют на проверку в тестирующую систему.

Под **тестирующей системой** в данной работе будем подразумевать сервер, на который учащиеся отправляют свой код, для проверки его корректного выполнения на заранее подготовленных тестах.

Тест для олимпиадной задачи по программированию — некоторые входные данные, удовлетворяющие условию задачи. Как правило, представляет из себя текстовый файл. Для каждой задачи обычно существует свой набор тестов. В наборе может быть, как один, так и несколько тестов. Тесты составляются до проведения соревнования или занятия и обычно неизвестны учащемуся.

Решением олимпиадной задачи называют программу, которая считывает входные данные, находит ответ и выводит его. Также нередко в условии задачи присутствуют ограничения на время выполнения и количество используемой памяти, поэтому корректное решение должно укладываться в эти ограничения.

Вердикт тестирующей системы — ответ тестирующей системы после проверки некоторого решения. Может быть положительным или одним из отрицательных. В случае получения положительного ответа, считается, что задача решена правильно, иначе, что в решении присутствует ошибка. Как правило, вместе с вердиктом учащийся получает комментарий с номером теста, на котором решение работает некорректно.

1.1.2. Ошибки в олимпиадном программировании

Ошибкой будем называть причину, по которой решение получает отрицательный вердикт на одном из тестов.

Множество ошибок можно интуитивно разделить на несколько категорий:

а) **Идейные**. Данные ошибки совершаются по причине написания неправильной интерпритации условия, либо некорректного выбора алгоритма для решения. Простой пример — решение задачи динамического программирования о рюкзаке, используя метод жадного программирования.

б) **Неэффективный выбор алгоритма**. У некоторых алгоритмов бывают разные версии и модификации, поэтому бывает, что, например вместо реализации алгоритма Дейкстры с кучей, учащийся пишет реализацию алгоритма Дейкстры без кучи. В подобных случаях решение выдает правильный ответ, но может либо потреблять памяти больше, чем указанное в условии максимальное доступное количество памяти либо не успевает завершиться раньше указанного в условии максимального времени работы.

в) **Неаккуратная реализация** также является частой причиной нарушения ограничений работы. Например, в решении используется неоптимальный способ считывания входных данных, или неподходящая структура данных. Сюда же можно отнести обращения к незаалоцированной памяти и подобные ошибки. Основная причина таких ошибок, заключается в том, что учащийся придумал правильную идею решения, выбрал правильный алгоритм, но не смог его корректно реализовать.

г) **Нерассмотренные случаи** часто приводят к отрицательным вердиктам. Во многих задачах существуют крайние случаи, например, когда в массиве один элемент и в этом случае программа будет работать некорректно. Лечатся такие ошибки при помощи условного оператора и отдельного рассмотрения этих самых случаев.

д) **«Мелкие»** ошибки можно сравнить с помарками при решении математической задачи, на подобии потери знака при переносе. Такие ошибки легко допустить, но сложно заметить при самопроверке. В области олимпиадного программирования к таким ошибкам можно причислить неправильные типы, ошибки в индексации, неправильные размеры массивов. Особым отличием таких

ошибок является то, что для того, чтобы помочь учащемуся, достаточно просто указать область кода, где она совершена, либо ограничится фразой по типу: «У Вас ошибка в ограничениях в массиве», либо «Нужно использовать шестидесятичетырехбитный тип данных, вместо тридцатидвухбитного».

1.1.3. Теория графов

Граф — абстрактный математический объект, который характеризует пара $G = (V, E)$, где V — множество вершин, а $E \subset \{(v, u) : v, u \in V\}$ — множество ребер.

Связный граф — граф, в котором между любой парой вершин существует хотя бы один путь.

Цикл — последовательность вершин вида $v_1, v_2 \dots v_k$, где $v_i \in V$, $(v_i, v_{i+1}) \in E$ и $v_1 = v_k$.

Ациклический граф — граф, который не содержит в себе циклов.

Дерево — связный ациклический граф.

В данной работе рассматриваются **подвешенные** деревья. Это деревья, у которых для каждых двух смежных по ребру вершин выполняется отношение предок-потомок(по-другому родитель-ребенок).

Вершина, у которой нет потомков называется **листом**, в свою очередь вершина, у которой нет предков, называется **корнем**. Очевидно, что у любого дерева ровно один корень, при этом листьев может быть сколько угодно много.

Рассмотрим некоторое дерево $T = (V, E)$. **Поддеревом** данного дерева будет дерево $T' = (V', E')$, такое что $V' \subset V$ и $E' = \{(v, u) \in E : v, u \in V'\}$, при этом если $v \in V'$, то для любой вершины w — потомок v в дереве T, выполняется $w \in V'$

Разностью дерева $T = (V, E)$ и его поддерева $T' = (V', E')$ будем называть дерево $T'' = (V'', E'')$, такое что $V'' = V \setminus V'$ и $E'' = \{(v, u) \in E : v, u \notin V'\}$. Интуитивно можно представить, что от дерева T «отрезали» его поддерево T'. Обозначать будем $T'' = T \setminus T'$.

Поиск в глубину — один из наиболее популярных алгоритмов обхода графа, используемый для изучения строения.

1.1.4. Синтаксический анализ

Абстрактное синтаксическое дерево (дерево разбора) — структура данных, представляющая из себя ориентированное дерево, в котором каждая вершина сопоставляется с оператором языка программирования, а листья с соответствующими операндами.

Лексический анализ — процесс аналитического разбора входного текста на известные группы, с последующем получением на выходе идентифицированных последовательностей, называемых «токенами». Лексический анализ используется в компиляторах и интерпретаторах исходного кода языков программирования, и в различных парсерах слов естественных языков.

Лексическим анализатором (жарг. лексер от англ. $lexer$) называется программа, выполняющая лексический анализ текста.

Синтаксический анализ(жарг. парсинг от англ. $parsing$) — процесс сопоставления последовательности токенов(слов) формального языка с его формальной грамматикой. Результатом будет абстрактное синтаксическое дерево. Как правило, для получения токенов проводится лексический анализ.

Синтаксическим анализатором (жарг. парсер от англ. $parser$) называется программа выполняющая синтаксический анализ.

Контекстно-свободная грамматика — способ описания формального языка, в котором нет зависимости от контекста.

Генератор синтаксических анализаторов — программа, которая получает на вход контекстно-свободную грамматику некоторого языка, а на выход выдает синтезированный код лексического и синтаксического анализаторов для данного языка. Для большинства используемых в олимпиадном программировании языков программирования существуют стандартные уже реализованные грамматики.

1.2. Постановка цели исследования

1.2.1. Предпосылки

Рассмотрим некоторый кружок, онлайн-курс или что-то подобное по олимпиадному программированию. В нем обучается некоторое количество учащихся, которых, как правило, многократно больше, чем преподавателей.

Процесс обучения включает в себя практику, которая представляет из себя решение олимпиадных задач, с последующей отправкой решений на проверку в тестирующую систему. На каждую посылку решения, тестирующая система выдает некоторый вердикт. В том случае, если вердикт положительный задача считается решенной правильно, и учащийся с чистой совестью переходит к решению следующих задач, иначе же, учащийся будет пытаться самостоятельно найти ошибку.

Однако, как правило, большинство учащихся из-за лени, недостатка опыта, знаний или еще каких-либо причин не могут найти ошибку самостоятельно и обращаются к преподавателю за помощью. Чтобы помочь в поиске ошибки, преподавателю необходимо прочитать код учащегося, иногда не один раз, и потратить некоторое время. Когда подобных учащихся много, то и преподавательского времени тратиться непомерно много. Несложно предположить, что продуктивность преподавания можно повысить, если избавить преподавателя от подобных обязанностей путем автоматизации процесса.

Таким образом имеем базу предыдущих решений задачи с вердиктами, в том числе и с отрицательными, которые храняться в тестирующей системе. Хотим на основе этой информации, когда приходит новый запрос от учащегося на поиск ошибки, находить ее автоматически, если такая или подобная ей уже допускалась ранее другим участником.

1.2.2. Выбор фокусировки исследования

В представленой работе будем фокусироваться именно на «мелких», на что есть несколько достаточно веских причин:

— Ошибки, не являющееся «мелкими» непонятно как именно находить, и скорее всего задача по поиску таких является NP-полной.
— Также, даже если представить, что мы нашли не являющуюся «мелкой» ошибку автоматически, неясно как компьютер сможет объяснить участнику, что же у него не так. Естественно фраза «Замените вот тот код, на вот этот» с приложенным следом кодом сомнительна, как минимум потому, что с таким же успехом, можно сказать «вот правильный код, используйте его».

— Как было сказано выше, в случае «мелкой» ошибки, компьютер будет несложно научить давать комментарии, помогающие найти и исправить ее.
— В случае поиска «мелкой» ошибки лично преподавателем, нужно прочитать весь код, иногда несколько раз. Именно на поиски таких ошибок, обычно, тратится наибольшее количество преподавательского времени.
— «Мелкие» ошибки чаще всего встречаются в коде, который нужно посмотреть преподавателю на предмет ошибок, так как учащемуся проще их допустить, а также гораздо сложнее найти их самостоятельно.
— Такие ошибки также обладают свойством повторяться, ведь чем меньше размер кода представляющего ошибку, тем больше вероятность, что подобную повторит кто-либо другой.

1.3. Обзор смежной области

Существуют несколько работ в смежных областях, цели которых чем-то похожи на поставленную в этой работы.

1.3.1. Software transplantation

Работа, которая на первый взгляд должна помочь в достижении поставленной цели — это «Automated software transplantation» [1].

В данной работе авторы рассматривают возможность «пересадки» кода из одной программы в другую, с целью передачи необходимой функциональности, по аналогии с трансплантацией огранов живого организма.

Для того, чтобы сделать это берется весь необходимый код, смотряться его зависимости от окружения и других частей программы, все это «вырезается» и ставится в код назначения в указанное место.

Несмотря на то, что метод зарекомендовал себя очень хорошо, работая в подовляющем большинстве тестовых случаев, применимо к поставленной цели его использовать невозможно. Так как он попросту будет пересаживать код из правильного решения в неправильное и сообщать что-то наподобии «Я, конечно, не знаю, где у Вас ошибка, но если вы допишите в точности вот этот код, который раньше уже сдали в систему, то и Вы сдадите». Польза от данной информации, как уже было

отмечено ранее крайне сомнительна, да и в принципе ставит под вопрос необходимость использования таких сложных методов, лишь для того, чтобы показать учащемуся правильный код, что, кстати, обычно, не делают, ведь теряется смысл обучения.

1.3.2. Система поиска плагиата

Имеет место идея применения парсеров и деревьев разбора, для поиска плагиата в текстах программ [2].

Сравнения на плагиат в данной работе происходит ни непосредственно текста программы, а дерева разбора. Представленный подход хорош тем, что сравнивает структуры программ, поэтому устойчив к переименованиям и переписыванием кода с одного языка программирования на другой.

Также одной из особенностей работы является расширяемость за счет добавления новых грамматик на вход генератора синтаксических анализаторов. То есть, чтобы начать проверять новый язык программирования, надо просто предоставить его грамматику.

Однако антиплагиат умеет лишь сравнивать два кода на похожесть по запрограммированной метрике, однако он никак не может изменять программу, что для достижения поставленной цели в работе просто необходимо. Также крупной проблемой является то, что в данном случае ищется именно плагиат, а в случае олимпиадного программирования имеем один алгоритм реализованный двумя разными людьми, возможно, двумя абсолютно разными подходами.

Все же, возможность сравнивать отдельные части дерева разбора для сравнения их на похожесть — это идея, которая найдет отражение в решении поставленной задачи.

1.3.3. Выводы по обзору смежной области

К сожалению, ни одна из рассмотренных работ не помогает в достижении поставленной цели. Что подводит к необходимости разработки своей системы синтеза исправлений.

1.4. Постановка задачи для исследования

Основой исследования будут ранее сделанные изменения в коде решения, которые программа будет пробовать применить к текущему

решению, в котором требуется исправить ошибку. В этом разделе речь пойдет про решения по одной конкретной задаче.

Рассмотрим два решения некоторого участника, который ранее получил положительный вердикт, по интересующей нас задаче. При этом первое решение из рассматриваемых получило отрицательный вердикт, а второе либо положительный, либо отрицательный, но на более высоком номере теста. Важно, чтобы второе решение, было позже в хронологическом порядке, интересны именно изменения, которые привели к улучшению вердикта.

Затем рассмотрим деревья разбора для кода двух выбранных решений. Дерево разбора первого решения назовем A, второго — B. **Исправлением** назовем пару (C_1, C_2), где C_1 — поддерево дерева A, C_2 — поддерево дерева B, при этом $A \setminus C_1 = B \setminus C_2$. Интуитивно исправление представляется следующим образом: из первого решения удалили отрывок кода, который сопоставлялся дереву разбора C_1, а вместо него написали отрывок кода, сопоставляющийся C_2, получив таким образом второе решение. С точки зрения деревьев, одно поддерево было заменено на другое.

По определению исправление улучшает вердикт для решения задачи. Под **опытом** будем подразумевать известные исправления.

Пусть теперь мы получили новое решение учащегося, в котором нужно найти ошибку. Также знаем для него вердикт тестирующей системы. Рассмотрим все прошлые исправления (C_1, C_2) по данной задаче, после чего попробуем применить его уже к новому решению. Для этого в дереве разбора нового решения необходимо найти поддерево «похожее» на C_1, заменить его на поддерево «похожее» на C_2, а затем проверить подошло исправление или нет. Будем считать его подходящим, если при отправке получившегося кода в тестирующую систему вердикт будет лучше, чем полученный ранее.

Таким образом, можно по пунктам сформулировать окончательную задачу для исследования:

— Выбрать метод анализа деревьев разбора программ.
— Научиться находить исправление по двум разным версиям одного решения.

— Сформулировать критерий «похожести» двух поддеревьев.
— Научиться применять исправление.
— Проверить исправление на улучшение вердикта.
— Оценить эффективность и предложить пути улучшения.

Выводы по главе 1

а) Введены необходимые термины и определения.
б) Установлено, что существующие методы работы с кодом, не применимы для решения задачи синтеза исправлений в олимпиадной задаче по программированию.
в) Сформулирована цель и задача исследования выпускной квалификационной работы.

ГЛАВА 2. РЕШЕНИЕ ПОСТАВЛЕННОЙ ЗАДАЧИ

2.1. Анализ текста решения

2.1.1. ANTLR

Для синтаксического анализа решения предлагается использовать генератор анализаторов для формальных языков ANTLR. Чтобы получить парсер нужного языка, достаточно просто передать грамматику для данного языка в нужном формате. ANTLR зарекомендовал себя как стабильное и удобное средство для работы с синтаксическим анализом, при этом обладая крайне полезными для представленной работы приемуществами:

- Свободное программное обеспечение.
- Использование единой нотации для описания лексических и синтаксических анализаторов.
- В репозитории разработчиков есть примеры грамматик для многих популярных языков программирования.
- Множество плагинов для работы в различных средах разработки, в том числе и для Intellij IDEA, которая использовалась в данной работе.
- Возможность добавлять Java-код в грамматику с целью подстановки его напрямую в парсер. При помощи этого можно запрограммировать парсер возвращать какую-угодно информацию о тексте.
- Предоставление сообщений об ошибках и восстановление после них, в том числе корректная обработка отсутствующих узлов, с последующим продолжением работы над остальным текстом.

Основной сложностью при работе с ANTLR стало то, что он не умеет по дереву разбора восстанавливать исходный код программы, так как все пробелы и разделители убираются при работе лексера, и никаких обратных приобразователей не генерируется. Также проблемой стало отсутствие возможности как-либо модифицировать построенные деревья разбора, однако обе эти проблемы решаемы.

Чтобы решить возникшие проблемы, можно модифицировать стандартную реализованную грамматику из репозитория разработчиков добавив код, который будет строить дерево разбора, методы и структуру для которого можно реализовать отдельно на языке Java.

После модификации грамматики работа с ANTLR сводится к нажатию пары кнопок для генерации лексера и парсера.

2.1.2. Выбор языка программирования

Так как работать планируется со структурой кода при помощи деревьев разбора, можно не умаляя общности рассматривать один конкретный язык программирования. Для добавления другого языка нужно будет лишь использовав новую грамматику получить новые лексер и парсер, а также запрограммировать элементы структуры дерева разбора.

В данной работе будет рассматриваться язык программирования Паскаль. Это один из наиболее популярных языков для олимпиад по программированию, который долгое время использовался для начального обучения школьников и студентов. В момент написания работы, язык программирования сдал лидирующие позиции, однако все еще популярен и используется даже на всероссийской олимпиаде школьников по программированию (РОИ). Язык дорабатывается по сей день в рамках языков Pascal ABC и Delphi.

Паскаль имеет следующие оссобенности:

— Строгая типизация.
— Структурное программирование.
— Текстовая простота.

При программировании на Паскале может сложиться ощущение, что вы программируете на английском языке, настолько синтаксис оптимизирован для обучения.

В представленной работе будет рассмотрено избыточное подмножество изначального стандарта языка «Pascal Standard», принятого в 1974 году.

2.1.3. Дерево разбора

Для работы с деревьями разбора по причинам, описанным ранее, необходимо было разработать собственную структуру данных, представляющую дерево разбора, при этом позволяющую изменять свою структуру и получать код программы, которой это дерево отвечает.

Разработка данной структуры проводилась на языке программирования Java. Этот язык объектно ориентированный, что положительно

Листинг 1 – Пример программы на языке Паскаль

```
program tmp;
const
  Author = 'Grigory';
var
  arr: array[1..2] of string;
begin
  writeln('Hi')
end.
```

сказалось на дизайне кода и удобстве его написания, а также ANTLR, как уже отмечалось, отлично с ним совместим.

Структура данных наследуется из основного общего класса `ASTNode`, который представляет из себя вершину дерева. Основные типы вершин это:

- переменная;
- константа;
- функция;
- процедура;
- условие;
- цикл с предусловием;
- цикл с постусловием;
- цикл со счетчиком;
- перечисление (код между `begin` и `end`);
- текст;
- тип.

Также есть вспомогательные:

- двоичная операция;
- унарная операция;
- скобки;
- строка;
- универсальная.

Универсальная вершина разработана для удобства, через вершину данного типа можно разработать любую вершину, у которой есть дети. От нее наследуются, например, вершины двоичной и унарной операций, а также вершиана перечислений.

Данная структура предназначена для представления конкретно выбранного языка, но несложно дополняется для поддержки дополнительной функциональности языка, либо другого языка.

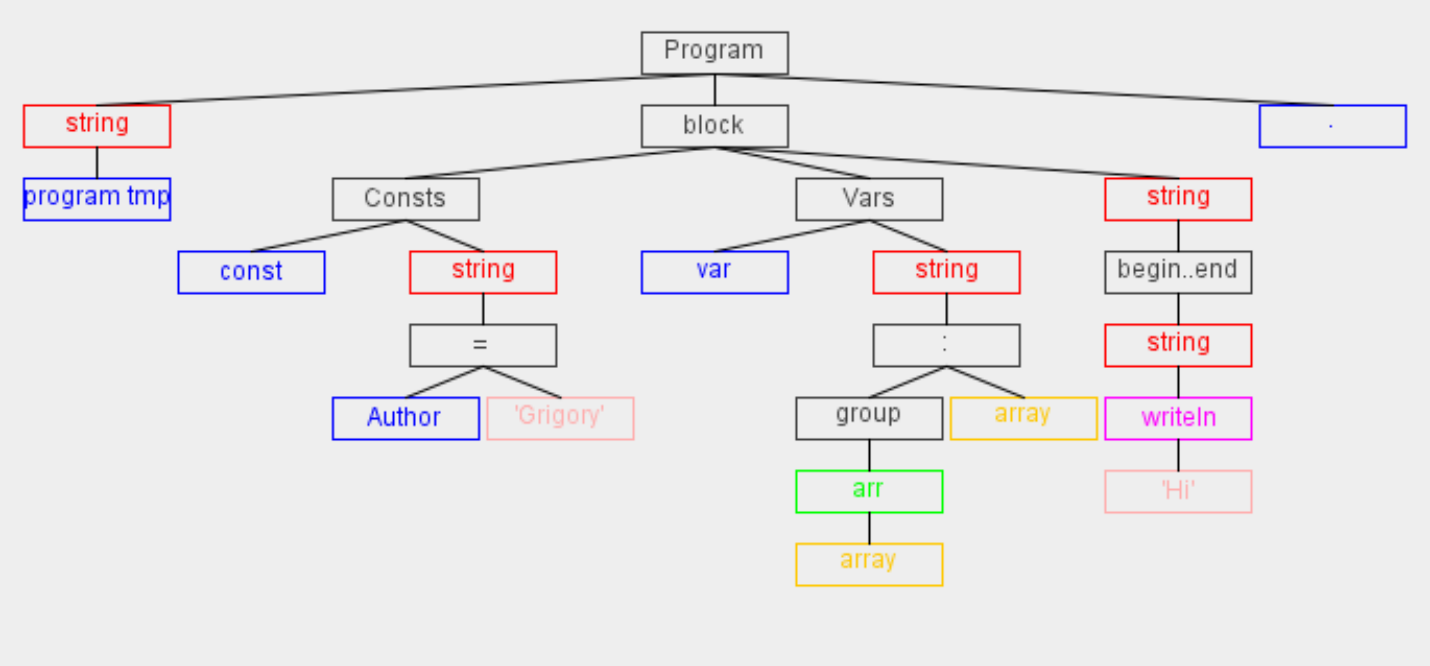

Рисунок 1 – Пример дерева разбора для программы из листинга 1

Как было отмечено ранее, ANTLR поддерживает вставки Java-кода, поэтому стандартная грамматика была модифицирована для поддержки разработанной структуры данных. Пример приведен в листинге 2.

В представленной структуре для каждого поддерева реализован метод `toString`, который сопоставляет ему корректный код программы, который репрезентует данное поддерево. Также реализовано множество методов, которые будут использованы по ходу продолжения работы.

2.2. Работа с деревьями разбора

2.2.1. Структура исправления

Имеется два решения участника. Построим для них деревья разбора, представленные структурой данных, описанной выше, с помощью имеющегося парсера.

Исследование фокусируется на «мелких» ошибках. Проанализируем, как выглядит поддерево дерева разбора для таких ошибок. Утверждается, что это почти всегда лист. Действительно, если это не так, то

Листинг 2 – Пример грамматики ANTLR с вставками Java-кода

```
constant returns [ASTNode ast]
 :unsignedNumber {
   $ast = new ConstNode($unsignedNumber.text, "num");
  }
 |sign unsignedNumber {
   $ast = new ConstNode($sign.text + $unsignedNumber.text, "sNum");
  }
 |identifier {
   $ast = new ConstNode($identifier.text, "id");
  }
 |sign identifier {
   $ast = new ConstNode($sign.text + $identifier.text, "sId");
  }
 |string {
   $ast = new ConstNode($string.text, "str");
  }
 ;
```

ошибка была где-то в структуре программы, что протеворечит определению «мелкой» ошибки. Единственное исключение — это структура арифметических формул. Такие формулы сами по себе могут сопоставляться сложным поддеревьям, но этот случай можно рассмотреть отдельно, с учетом приоритетов, симметричности и других свойств операций. В представленной работе он не освещается.

Теперь рассмотрим структуру исправления. Исправление есть пара поддеревьев с отношением было-стало. Как было рассмотрено, первое поддерево это всегда лист, второе же поддерево, как правило, тоже лист, но может обладать и более сложной структурой, однако это не так важно, потому что основной задачей видеться именно найти место в новом решении, куда именно нужно попробовать подставить код, чем сама подстановка кода.

Также стоит отдельно рассмотреть зависимости кода изменения от остального кода программы. Например, когда была использована неправильная переменная, и ее заменили на другую. Однако поиск зависимостей осложняется по следующим причинам:

— Для решения данной задачи нужен анализ решения на более высоком уровне чем лексер-парсер, как реализовано в данной работе, а именно уровень семантики языка. Уровень, для реализации кото-

рого требуется написать подмножество компилятора, что является далеко не простой задачей.

— Для определения переменной, которая является такой же, что и заменяемая в другом решении, нужно знать все объявленные переменные, включая заголовки, подключенные библиотеки и модулю. В Паскале примерами подобной переменной, могут послужить `input` и `output`.
— Так как не всегда умеем парсить абсолютно все конструкции языка, внутри этих конструкций, тоже может что-то неожиданное произойти.

Все вышеперечисленное требует того, чтобы мы умели в анализе переменных в случае чего «деградировать» до поведения, в котором неизвестно, одно и тоже обозначают две переменные с одним именем, или же нет. Которое, в общем-то, как раз и реализовано. При этом вероятность такой деградации, в случае поддержки подмножества языка, довольно велика.

2.2.2. Поиск исправления

Есть два дерева разбора, нужно найти в них «отличия». Про искомое поддерево в первом дереве известно, что оно лист. При этом, все остальные узлы деревьев должны быть одинаковыми по определению исправления. Случай, когда таких листьев несколько можно свести к последовательному применению двух исправлений.

Используем алгоритм обхода графа поиском в глубину, состоянием в котором будет пара вершин: в первом и втором дереве.

— Рекурсивно перебирая детей вершины, спускаемся глубже.
— Проверяем вершину первого дерева на наличие детей, если их нет, то это лист и ответом является текущее состояние.
— Проверяем на равенство структуру вершин. Если структуры разные, то ошибка не «мелкая» и можно терминировать алгоритм с информацией об этом.

В случае, если хотим найти все мелкие ошибки, не нужно терминировать алгоритм, а просто добавлять ответы в некоторую структуру данных, например лист.

Также стоит отметить, что функциональность разработанной структуры данных позволяет сравнивать два узла на равенство.

2.3. Применение исправления

2.3.1. Критерий «похожести»

Для того, чтобы применять исправления в решении, написанном другим автором, необходимо сформулировать критерий «похожести» для двух поддеревьев. В дальнейшем, именно модификация этого критерия позволит находить более сложные ошибки чем «мелкие», на подобии нерассмотренных случаев.

В данной работе, можно сказать, что два поддерева **похожи**, если:

— У них одинаковая структура.

— Переменные одного типа, но возможно, по-разному называются.

— Если в поддеревьях есть константы, они могут отличаться.

Последнее полезно для типов, так как `array[1..1000] of integer` и `array[1..1001] of integer` вполне себе похожи, однако не равны.

Таким образом, возможно изменение констант и переиспользование переменных.

2.3.2. Непосредственно применение

Имеем исправления, найденные в имеющейся базе проверок решений тестирующей системы. Получаем запрос на поиск ошибки в некотором решении, про которое известен вердикт тестирующей системы. После чего ищем все исправления в базе, в которых первое поддерево вырезано из решения с таким же вердиктом, как и у новоприбывшего решения. Остается, только «примерить» данное исправление.

С помощью поиска в глубину найдем все похожие на первое поддерево узлы в новом решении, после чего, перебирая их по очереди, будем подставлять вместо них второе поддерево из исправления. Простая подстановка будет корректна, так как в работе, по рассмотренным ранее причинам, не рассматриваются зависимые от остального кода вершины.

В случае, если первая часть изменения — переменная, предварительно найдем ее во второй части и при подстановке, заменим узел, на тот, что используется в рассматриваемом решении. Данный случай, на первый взгляд, является частью зависимых случаев, однако это не

совсем правда, так как при подстановке мы не используем каких либо переменных из внешнего окружения. В прочем, если читателю сложно представить пример такой подстановки, то он довольно прост и популярен — ошибка в индексе. Например, было `a[i]`, а стало `a[i + 1]`. Довольно частая ошибка при переходе от ноль-индексации к один-индексации, либо обратно.

Возникает последний вопрос: «что делать, если одно исправление надо применить в нескольких местах?». Сложно представить, чтобы в решении задачи олимпиадного программирования было более двадцати различных мест, с одинаковой ошибкой, поэтому предлагается, просто, попробовать подстановку в одно место, потом во все сразу, и если не поможет, то пробовать перебирать в какие вхождения подставлять, а в какие нет.

Подставлять во все места сразу, кажеться неплохой идеей, так как немалое количество исправлений часто безвредно. Например, переход от тридцатидвухбитного типа данных к шестидесятичетырехбитному, либо увеличение всех рамерностей массивов. То есть вероятность того, что можно исправить все, в случае, если исправить ровно одно место недостаточно, довольно велика, однако ассимптотически ничего не портит.

Также стоит отметить, что в случае синтеза исправлений автоматизация гораздо важнее времени выполнения, если оно пристойно, поэтому подбирать какие-то более сложные схемы приоритезации попросту бессмысленно.

Выводы по главе 2

а) Описан способ анализа текста решения, аргументирован выбор генератора синтаксических анализаторов и языка для исследования.
б) Описан поиск исправления по двум решениям одного учащегося, где у второго вердикт лучше чем у первого.
в) Сформулирован критерий «похожести», на основе которого разработан алгоритм применения исправления.

ГЛАВА 3. ТЕСТИРОВАНИЕ

3.1. Структура данных для дерева разбора

Для проверки того, что код корректно приобразуется в дерево разбора, а потом обратно в исходный код, а также для проверки достаточного покрытия программ олимпиадного проограммирования реализованным подмножеством языка, были взяты случайные решения из архива PCMS2 — тестирующая система, используемая в Университете ИТМО для обучения школьников и студентов, а также проведения различных соревнований вплоть до всероссийской командной олимпиады школьников по программированию и полуфиналу чемпионата мира по программированию ACM ICPC.

На 47 из 50 выбранных программ структура отработала корректно. В остальных трех сказалась неполнота подмножества языка:

а) В первой программе использовалось объявление типа через `record`.
б) Во второй условный оператор `case`.
в) В третьей конструкция `writeln(a:5:1)`.

Все три причины не входят в подмножество языка, однако при добавлении их в грамматику, в дальнейшем смогут обрабатываться корретно. Третья конструкция и вовсе не предусмотрена стандартной грамматикой языка Паскаль, реализованной разработчиками ANTLR, но вполне может быть дописана на языке ANTLR, как отмечалось ранее, в грамматику можно дописать все, что душе угодно.

3.2. Придуманные тесты

3.2.1. Различные популярные ошибки

Для первоначальной проверки работы были взяты случайные решения из архива PCMS2, в которые были добавлены ошибки взятые из опыта работы в учебной сфере автора работы:

— Неправильные константы.
— Потерянные ± 1 в индексах массивов. В том числе и многомерных.
— Неправильные типы переменных.
— Неправильные размерности массивов.
— Орфографические ошибки в модификаторах, например написать `vor` вместо `var`.

Для всех этих случаев, все работало безотказно. Однако, это сомнительный показатель, так как все-таки исправление, которое ищется в той же самой программе. Именно поэтому перед тестами на реальных данных будет приведено последнее внутреннее тестирование.

3.2.2. Обфускация

Обфускация (от англ. obfuscate — делать неочевидным, запутанным, сбивать с толку) — приведение исходного текста программы к виду, сохраняющему её функциональность, но затрудняющему анализ, понимание алгоритмов работы и модификацию кода. В промышленности данный прием применяют, наприемр, для сокрытия корпоративных тайн, в случае, когда надо открыть код.

По сути обфускация, это «загрязнение кода»: переименование пременных, добавление нейтральных строк, изменение форматирования и т.д. Модифицируем предыдущий метод тестирования, применив обфускацию, которая:

— переименовывает переменные в строки из десяти и больше случайных латинских букв;
— убирает отступы;
— случайно переставляет объявление функций, констант и переменных между собой;
— добавляет лишние переменные и константы;
— добавляет нейтральные строки, вида присвоения переменной, которой раньше не было какого-то значения.

После обфускации запускаем реализованный алгоритм, затем выясняется, что он работает абсолютно также, как и в предыдущем случае. То есть он находит исправления в таком же коде, реализованном по-другому, игнорируя другие имена переменных и изменения общей структуры. Данный результат был довольно предсказуем, исходя из описания алгоритма, однако, тот факт, что работа оправдывает возложенные ожидания, не может не радовать.

3.3. Реальные данные

Ну тут что-то появиться. Со временем. Надеюсь. Очень. Пожалуйста.

Выводы по главе 3

а) Проведено тестирование структуры данных.
б) Проведено тестирование на искусственных и естественных тестах.
в) Показаны результаты работы.

ЗАКЛЮЧЕНИЕ

В результате выпускной квалификационной работы получены следующие результаты:

а) Проведен обзор смежной области задачи.

б) Придложен метод синтеза исправлений для неверных решений олимпиадных задач по программированию.

в) Проведено комплексное тестирование предложенного метода.

По ходу работы было замечено, что многие исправления применимы не только в рамках одной конкретной задачи и вердикта, но можно считать универсальными и применять в других задачах или при других вердиктах. Для модификации метода, используя этот факт, предлагается для каждого исправления хранить статистику насколько оно универсально. Тогда, в случае когда не подошло ни одно исправление по данной конкретно задаче, можно последовательно начать применять наиболее популярные исправления, естественно поставив какое-то ограничение по времени. Тут же надо заметить, что нужно будет сформулировать критерий равенства для исправлений, чтобы не хранить одно и то же дважды, сокращая таким образом время работы.

Данная работа является лишь малой частью рассмотрения обширной темы генерации исправлений. В будущем возможен ряд серьезных модификаций:

— Разработка плагина для тестирующей системы `PCMS2` с целью внедрения описанного метода в онлайн-курс Максима Буздалова.

— Добавление поддержки нескольких популярных языков.

— Добавление поддержки зависимостей.

— Формулировка критерия «похожести» для поддеревьев, не являющихся листами, для дополнительной обработки ошибок из категории «нерассмотренный случай».

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Automated Software Transplantation / E. T. Barr [et al.] // ISSTA 2015 Proceedings of the 2015 International Symposium on Software Testing and Analysis. — 2015. — P. 257–269.

2 *Логинов А. И.* Система поиска плагиата // Сборник работ 70-ой научной конференции студентов и аспирантов Белорусского государственного университета. Т. 1. — 2013. — С. 207–211.