Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий

Работа допущена к защите
зав. кафедрой
_____ В.М. Ицыксон
«_____» _____ 2022 г.

ВыпусКнАя КВАЛИфИКАцИоннАя рАБотА
рАБотА БАКАЛАВрА
ИнтЕрАКтИВныЙ ВИЗуАЛИЗАтор МоДЕЛЕЙ проГрАММ

по направлению 09.03.01 «Информатика и вычислительная техника»
по образовательной программе
09.03.01_01 Вычислительные машины, комплексы, системы и сети

Выполнил
студент гр. 3530901/80101 Д.А. Кучмин

Руководитель
к.т.н,
доцент, <*подпись*> В.М. Ицыксон

Консультант
по нормоконтролю <*подпись*> А.Г. Новопашенный

Санкт-Петербург
2022

ЁАнКт-ПЕтЕрБурГсКИЙ поЛИтЕхнИчЕсКИЙ унИВЕрсИтЕт
ПЕтрА ВЕЛИКоГо
Институт компьютерных наук и технологий

УТВЕРЖДАЮ
зав. кафедрой
_____ В.М. Ицыксон
«_____» _____ 2022г.

ЗАДАНИЕ
на выполнение выпускной квалификационной работы

студенту Кучмину Дмитрию Алексеевичу гр. 3530901/80101

1. Тема работы: Интерактивный визуализатор моделей программ.
2. Срок сдачи студентом законченной работы: дд.мм.202X.
3. Содержание работы (перечень подлежащих разработке вопросов):
   3.1. Изучение предметной области и аналогов.
   3.2. Выбор используемых технологий.
   3.3. Разработка интерактивного визуализатора моделей программ.
   3.4. Тестирование интерактивного визуализатора моделей программ.
4. Дата выдачи задания: дд.мм.202X.

Руководитель ВКР _____ В.М. Ицыксон
Задание принял к исполнению дд.мм.202X
Студент _____ Д.А. Кучмин

ÈЕфЕрАт

На 41 с., 27, 2

КЛючЕВыЕ сЛоВА: стАтИчЕсКИЙ АнАЛИЗ, JAVASCRIPT, JAVA, ВЕрИфИКАцИя И АнАЛИЗ проГрАММ, ДИнАМИчЕсКАя ВИЗуАЛИЗАцИя.

Тема выпускной квалификационной работы: «Интерактивный визуализатор моделей программ».

Данная работа представляет собой разработку интерактивного визуализатора моделей программ, который упростит подачу материала студентам и поможет разобраться в устройстве абстрактного синтаксического дерева, графа потока управления, графа зависимости по данным, графа зависимости программы, абстрактного семантического графа и представления на основе однократного статического присваивания. Результатом работы является клиент-серверное приложение, визуализирующее все вышеупомянутые модели. Приложение поддерживает программный код, написанный на Java.

ABSTRACT

41 pages, 27 figures, 2 appendices

KEYWORDS: STATIC ANALYSIS, JAVASCRIPT, JAVA, PROGRAM VERIFICATION AND ANALYSIS, DYNAMIC VISUALIZATION.

The subject of the graduate qualification work is «Interactive visualizer of program models».

This work is the development of an interactive visualizer of program models that will simplify the presentation of material to students and help to understand the structure of an abstract syntax tree, a control flow graph, a data dependency graph, a program dependency graph, an abstract semantic graph, and a representation based on a single static assignment. The result of the work is a client-server application that visualizes all the above models. The application supports program code written in Java.

ЁоДЕрЖАнИЕ

ВВЕДЕнИЕ

На ранних этапах обучения программированию студенты зачастую сталкиваются с непониманием программного кода. Оно, отчасти, происходит из-за отсутствия визуализации. Согласно исследованию, 65% информации человек воспринимает зрением, а остальные 35% всеми другими органами чувств. Визуализатор значительно упростит как изложение материала преподавателям, так и восприятие получаемой информации студентами. Визуализатор – вид программного обеспечения, предназначенный для преобразования разной информации в зрительные образы. Может быть или отдельным приложением, или плагином, или частью иного приложения. На данный момент существует множество таких утилит, например, визуализатор поведения многопоточных Java-программ, который демонстрирует выполнение написанного кода в виде схемы, в то время как без визуализации понять некоторые программы, в которых используется многопоточность, затруднительно. Это происходит из-за того, что исходный код, представленный в виде текста, достаточно плохо отображает ход выполнения программы. Такие области программирования, как графы и деревья, являются достаточно сложными в представлении и понимании, поэтому визуализатор, разрабатываемый в ходе данной выпускной работы, будет являться наглядным инструментом для демонстрации моделей программ, включающих в себя одну функцию или метод, написанный на языке программирования Java. Проблема визуализации графов весьма старая и для ее решения было разработано много средств, однако практически все найденные решения направлены на визуализацию одного типа графа, написанного на определенном языке программирования. Цель данной работы - создать интерактивный визуализатор моделей программ, то есть наглядного инструмента для демонстрации таких моделей программ, как абстрактного синтаксического дерева (AST), графа потока управления (CFG), графа зависимости по данным (DDG), графа зависимости программы (PDG) и абстрактного семантического графа (ASG) и представления в виде однократного статического присваивания (SSA). Визуализатор поддерживает код, написанный на Java. В данной выпускной квалификационной работе будут рассмотрены существующие программные решения для визуализации, проведен их обзор и анализ. Результатом работы является интерактивное клиент-серверное приложение, выполняющее все поставленные задачи. В конце работы проводится тестирование и анализ результатов.

ГЛАВА 1. АнАЛИЗ прЕДМЕтноЙ оБЛАстИ

В данной главе рассматривается предметная область и описываются основные понятия, которые с ней связаны, представлены требования к разрабатываемому программному средству, а также постановка цели и задач выпускной работы.

1.1. Описание предметной области, обоснование актуальности

На текущий момент компьютерная индустрия стремительно развивается, и это неразрывно связано с увеличением количества используемых программных продуктов и увеличением их сложности. В последние годы объем программного кода многих продуктов составляет миллионы строк. С ошибками в программном коде сталкивался каждый программист и именно по этой причине на данный момент актуальна задача поиска этих ошибок. Для поиска ошибок используются различные методики. Они могут быть как основаны на анализе только исходных кодов (методы статического анализа), так и на использовании информации времени выполнения (методы динамического анализа). Динамические методы (такие как тестирование) просты в реализации, не требуют больших вычислительных затрат, но при этом позволяют выявлять ошибки только для конкретных трасс исполнения программы. Методы статического анализа характеризуются высокой вычислительной сложностью, но при этом позволяют обнаруживать ошибки во всех возможных трассах исполнения.

Модели программ делятся на структурные, поведенческие и гибридные. Структурные модели основаны на синтаксисе исходного языка, а поведенческие дополнительно используют семантику конструкций языков программирования. К структурным моделям относят:

- Дерево разбора
- Абстрактное синтаксическое дерево

К поведенческим моделям относят:

- Граф потока управления

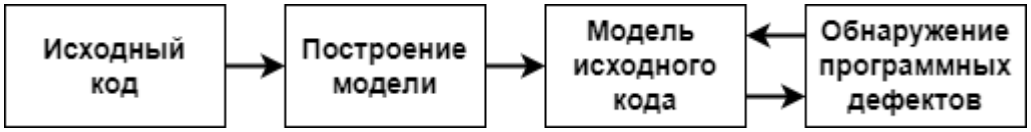


Рис.1.1. Общая схема статического анализа

- Статическое однократное присваивание
- Граф зависимостей по данным
- Граф зависимости программы

К гибридным моделям относят абстрактный семантический граф.

1.2. Описание моделей программ

Каждая из перечисленных моделей имеет свою область применения, связанную с целью проведения разбора (компиляция, оптимизация, распараллеливание, анализ и т. п.). Рассмотрим каждую из вышеупомянутых моделей подробнее и проведем анализ.

1.3. Абстрактное синтаксическое дерево

Абстрактное синтаксическое дерево (Abstract Syntax Tree, AST) - дерево, которое в абстрактном виде представляет структуру программы. AST создаётся парсером по мере синтаксического разбора программы, обрабатывается путём обхода при проверке семантических правил и проверке/определении типов, а затем, также, путём обхода AST, выполняется генерация кода. В современных компиляторах AST и список диагностик (ошибок, предупреждений) — это два результата вызова модуля синтаксического разбора. Данное дерево содержит полную синтаксическую модель программы без лишних деталей (таких, как пробельные символы или комментарии). Ниже представлен программный код и представлена визуализация AST. Данный код используется для всех последующих примеров в этой главе.

1.4. Граф потока управления

Граф потока управления (Control Flow Graph, CFG) — модель программы, представляющая в виде ориентированного графа поток управления в программе. В этой модели сохраняются только инструкции программы, а также информация о возможной передаче управления между инструкциями. Каждый узел (вершина) графа соответствует базовому блоку - прямолинейному участку кода, не содержащего в себе ни операций передачи управления, ни точек, на которое управление передается из других частей программы. Имеется лишь 2 исключения: точка, на

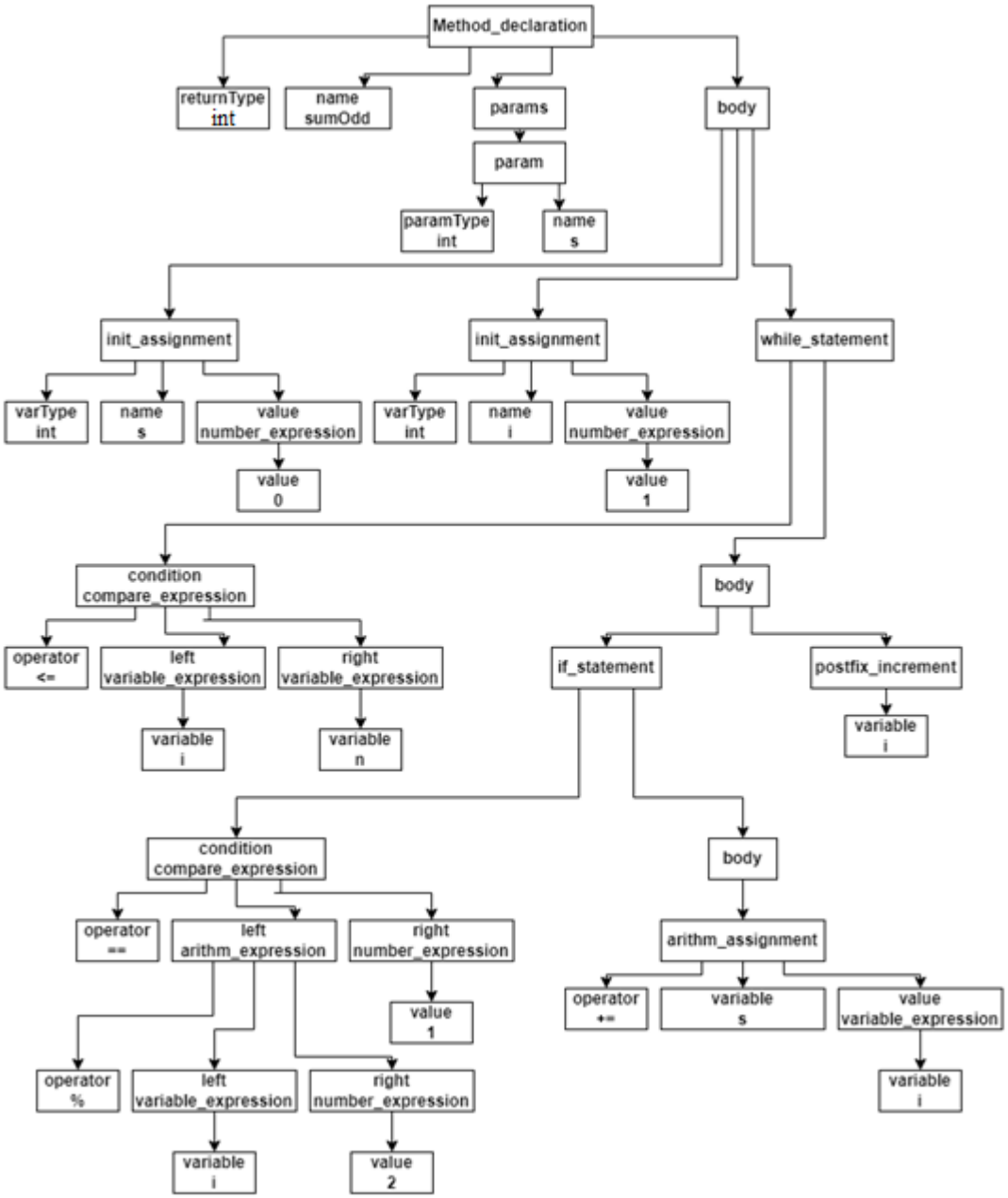


Рис.1.2. Общая схема статического анализа

которую выполняется переход, является первой инструкцией в базовом блоке, и базовый блок завершается инструкцией перехода. Направленные дуги используются в графе для представления инструкций перехода. Достижимость — одно из свойств графа, используемое при оптимизациях. Если блок или подграф не имеют путей до них от входного блока, то данная часть графа является недостижимой (мертвый код) при любых вариантах исполнения, и по этой причине он может быть удален из программы. Если же из данного подграфа нет путей до выходного блока, то значит этот подграф содержит бесконечный цикл. Граф потока управления используется для:

– Генерации кода

– Оптимизации программ
– Обнаружения ошибок в программе
– И т. п.

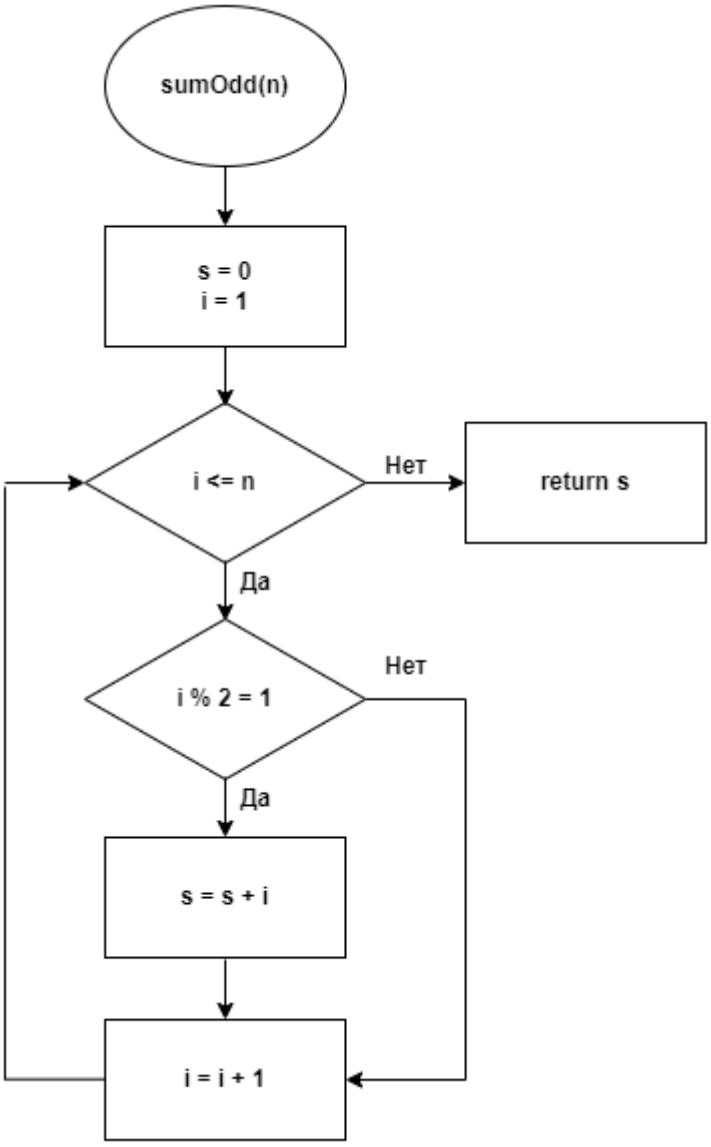


Рис.1.3. Общая схема статического анализа

1.5. Граф зависимости по данным

Граф зависимостей по данным (Data Dependency Graph, DDG) — модель программы, представляющая в виде направленных дуг зависимости по данным между узлами-инструкциями. Дуга связывает два узла тогда и только тогда, когда между соответствующими инструкциями есть зависимость по данным. Зависимость по данным может иметь один из трех типов: "запись-чтение "чтение-запись "запись-запись". DDG используется при решении задач оптимизации, а также задач

автоматизации распараллеливания выполнения. Особенностями данной модели являются относительно небольшое количество типов узлов и связей между ними, простота доступа и навигации. Исходя из всего вышесказанного, можно сделать вы-

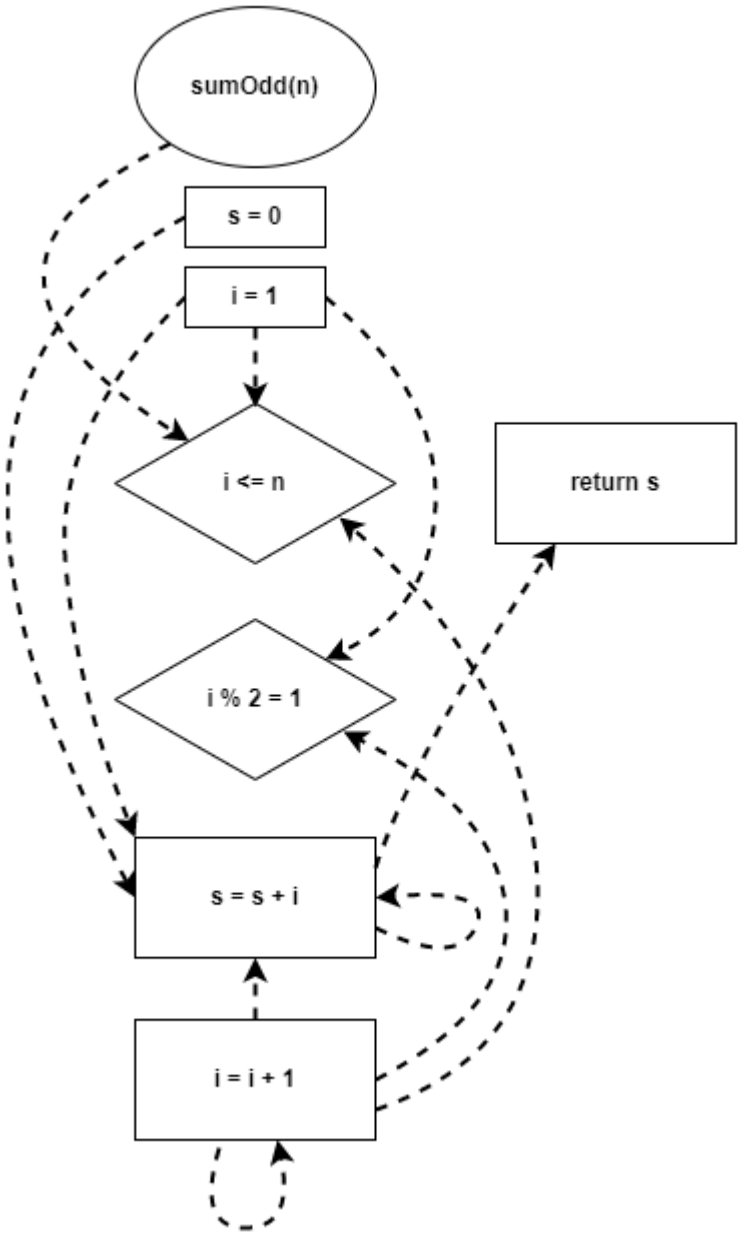


Рис.1.4. Общая схема статического анализа

вод, что модели на основе CFG и DDG применимы для отдельных аспектов анализа кода, отражая лишь часть имеющихся зависимостей, и не обладают необходимой полнотой для всестороннего анализа исходного кода. sectionГраф зависимости программы Граф зависимостей программы (Program Dependence Graph, PDG) – это объединенный граф потока данных и графа потока управления. Вершины PDG – это инструкции программы, а ребра – зависимости между ними. Есть две основных

типов ребер: ребра выражающие зависимости по данным и ребра выражающие зависимости по управлению. Граф зависимостей программы используется для:

- Оптимизации
- Генерации кода
- Анализа программ
- Верификации
- И т. п.

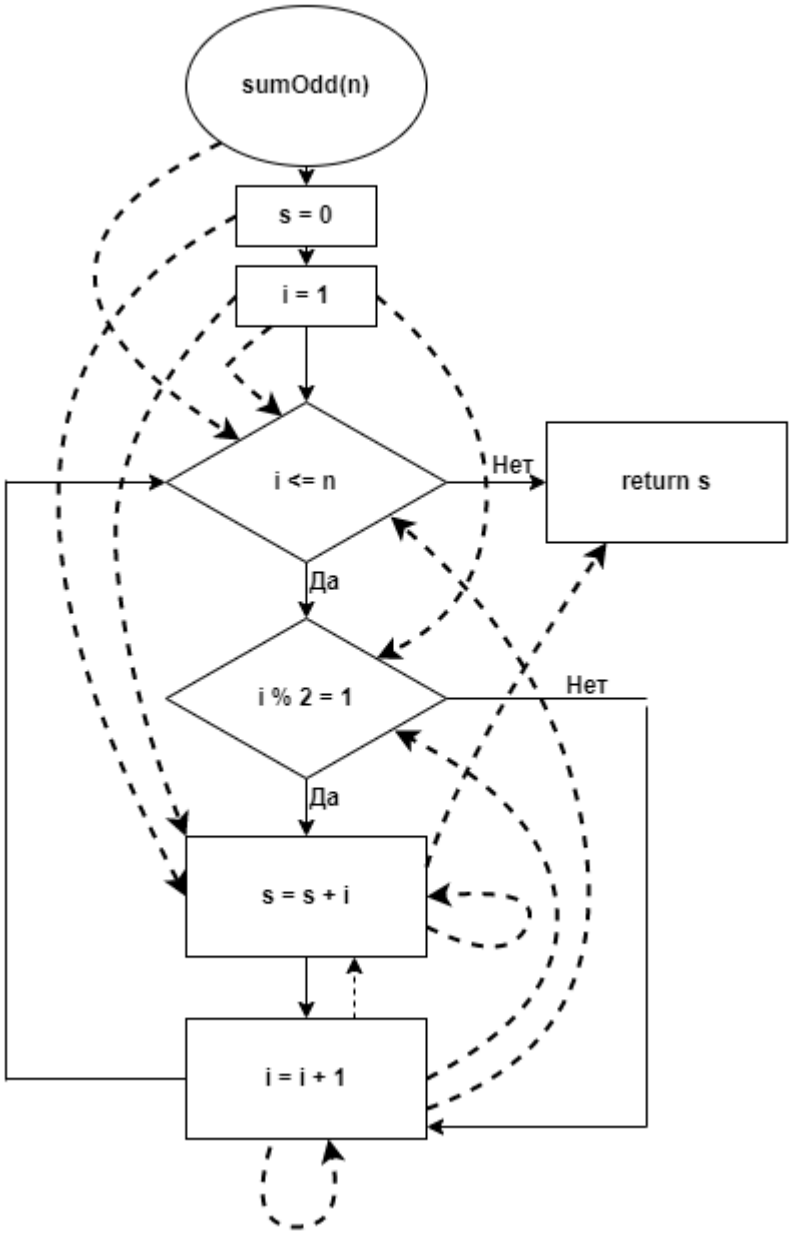


Рис.1.5. Общая схема статического анализа

1.6. Абстрактный семантический граф

Абстрактный семантический граф (Abstract Semantic Graph, ASG) является расширением AST. ASG по сравнению с AST дополнен различными семантическим дугами, отображающими соответствующие семантические свойства программы и упрощающими навигацию по исходному коду. Данная модель обладает необходимой полнотой для всестороннего анализа исходного кода программы. Для ряда методов статического анализа целесообразно применение упрощенных видов ASG. рассматриваемых как отдельные модели. Подобные модели концентрируются на определенных семантических аспектах, за счет чего размерность и сложность моделей сокращаются.

1.7. Представление на основе однократного статического присваивания

Представление на основе однократного статического присваивания (static single assignment, SSA) – это представление исходного кода программы, в котором:

- Каждой локальной переменной значение присваивается только один раз.
- Вводится версионирование для локальных переменных, которые в исходном коде имеют неоднократные присваивания.
- Для локальных переменных вводятся Фи-функции на выходе условных конструкций, объединяющие несколько ветвей программы и определяющие их окончательное значение
- Циклы заменяются инструкциями ветвления и безусловных переходов

Данное представление может быть изображено в CFG-форме.

1.8. Выводы по главе

В данной главе были изучены методики исправления ошибок в программном коде и проведен анализ предметной области. Для каждой модели приведен теоретический обзор и представлен один из вариантов визуализации. Модельное представление программы призвано упростить и формализовать разработку алгоритмов статического анализа. При этом выбранное модельное представление должно обладать определенными свойствами, обеспечивающими эффективность и наглядность алгоритмов статического анализа.

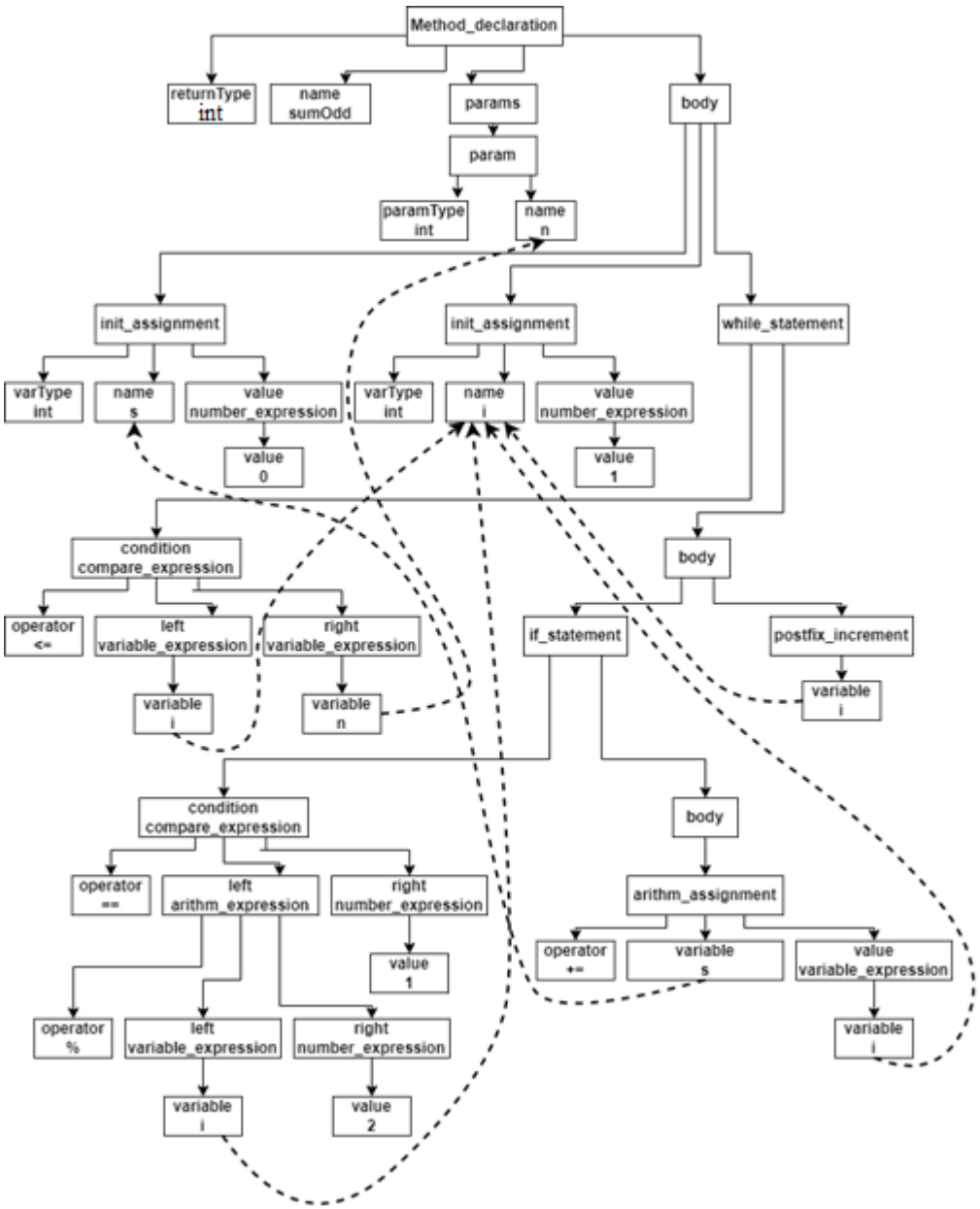


Рис.1.6. Общая схема статического анализа

ГЛАВА 2. БЗор сущЕстВующИх рЕшЕнИЙ

В данной главе будут рассмотрены существующие средства визуализации, определены их плюсы и минусы и указан краткий обзор.

2.1. Анализ существующих средств визуализации моделей программ

Существует множество вариантов архитектуры визуализатора. Это может быть как десктопное приложение, как мобильное, как клиент-серверное. Для анали-

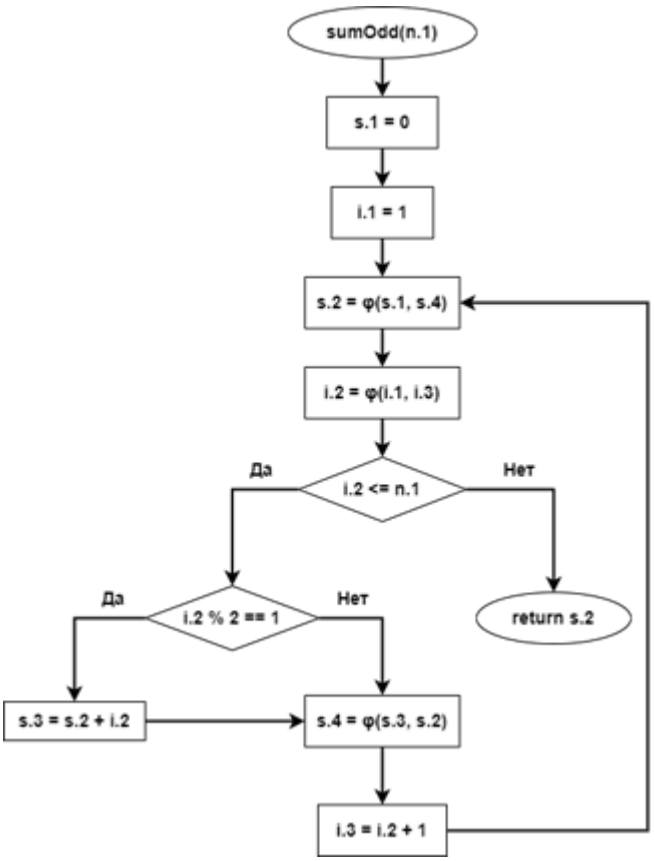


Рис.1.7. Общая схема статического анализа

за были выбраны визуализаторы, находящиеся в свободном доступе и максимально приближенные к желаемому результату.

2.2. Анализ существующих средств визуализации моделей программ

ANTLR (Another Tool for Language Recognition) – это мощный генератор парсеров. С его помощью можно сгенерировать парсер для любой заданной грамматики, в том числе для языков программирования. Для генерации парсера нужен файл грамматики, составленный по определённым правилам. Существует множество готовых грамматик для разных языков, в том числе для языка программирования Java. Кроме того, есть возможность вывести дерево парсинга в графическом интерфейсе. Сам по себе ANTLR не является готовым решением для построения и визуализации AST. Он лишь может сгенерировать код по шаблону (Visitor или Listener) на одном из поддерживаемых языков программирования, который можно дописать для получения желаемого результата. Дерево парсинга, которое получается при использовании ANTLR, содержит в себе все символы исходного текста. Чтобы из этого построить AST, необходимо убрать множество лишних элементов.

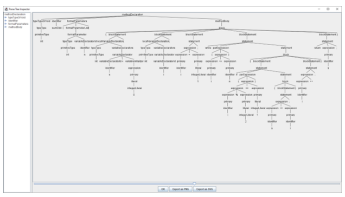


Рис.2.1. Графический интерфейс ANTLR

2.2.1. AST Explorer

AST Explorer – это инструмент для парсинга кода на разных языках, в том числе Java, который использует Node.js в качестве сервера и содержит ссылки на каждый пакет в репозитории npm, используемый в качестве парсера для того или иного языка. Из способов визуализации поддерживается только вывод в виде интерактивного иерархического текста, либо в виде кода JSON. И строится в результате не AST, а дерево парсинга, которое тоже содержит множество ненужных для построения AST узлов. Таким образом, его тоже нельзя назвать готовым решением для построения и визуализации AST для кода на языке Java.

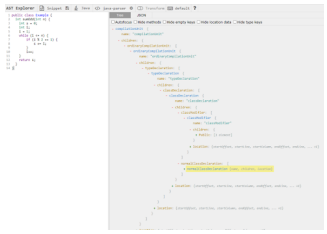


Рис.2.2. Результат работы AST Explorer

2.2.2. VisualDFA

VisualDFA – это сложный образовательный инструмент для визуализации анализа потоков данных с использованием Java/Jimple, который позволяет:

- Запускать 4 встроенных анализа: сворачивание констант (Constant Folding), биты констант (Constant Bits), достигающие определения (Reaching Definitions) и проверка на заражённость (Taint Analysis)
- Вводить Java-код, либо подключать его из файла
- Визуализировать код с помощью CFG
- Выполнять код пошагово, либо продолжить до следующей установленной точки остановки
- Видеть входные и выходные данные для любого блока или строки кода в любой момент времени

- Экспорт CFG в PNG-файл или вывод одной командой изображений всех шагов анализа
- Внедрять и запускать пользовательские анализы без перекомпиляции VisualDFA

Этот инструмент строит CFG в интерфейсе своего приложения. Внешний вид получаемого CFG не такой, как хотелось бы его видеть. У каждого блока вверху есть пустая область, за которую блок можно перетаскивать. При этом во время перетаскивания размещение стрелок происходит странным образом. Они пересекают другие блоки и частично остаются на начальных позициях. Так же блоки начала метода, условий и обычных операторов визуально не отличаются. От условий идут стрелки, смотря на которые непонятно, по какой из них переходить в случае, если условие истинно, и по какой в случае, если ложно. Создаются временные переменные там, где их нет в коде. Почему-то сами условия записываются противоположно тем, которые написаны в коде. Как итог, VisualDFA можно назвать готовым решением для построения и визуализации CFG, но результат оставляет желать лучшего.

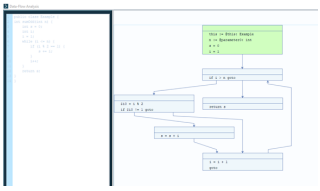


Рис.2.3. Результат работы VisualDFA

2.2.3. VisualControlFlowGraph4J

VisualControlFlowGraph4J – инструмент для построения и визуализации Java-кода в виде CFG, использующий ANTLR и GraphViz. Инструмент помогает строить CFG, но у него также есть ряд недостатков. Построение CFG нельзя назвать интерактивным. Он требует файла с исходным кодом в папке с ресурсами для того, чтобы строить модель по нему. Граф сохраняется в файловую систему в виде SVG-файла по завершению работы программы. В самом CFG присутствуют обозначения типов данных, что делает граф привязанным к языку программирования. Также наблюдается проблема с тем, что, смотря на стрелки, исходящие из блоков условий, непонятно, по какой из них переходить при истинном или ложном условии. Кроме того, никак не обозначены параметры метода, граф для

которого строится. VisualControlFlowGraph4J можно считать готовым решением для построения и визуализации CFG, но не лишённым своих недостатков.

Рис.2.4. Результат работы VisualControlFlowGraph4J

2.3. Выводы по главе

В данной главе проведен обзор и анализ существующих средств визуализации моделей программ. Были найдены лишь решения для AST и CFG, для визуализации остальных моделей программ не было найдено ни одного достойного решения. Также хочется отметить, что не существует ни одного сервиса, способного интерактивно визуализировать несколько разных моделей программ. На основе результата обзора можно сделать вывод о том, что тема выпускной квалификационной работы является актуальной.

ГЛАВА 3. ПостАноВКА ЗАДАчИ И рАЗрАБотКА трЕБоВАнИЙ К сИстЕМЕ

В данной главе производится постановка задачи и разработка требований к сервису интерактивной визуализации моделей программ.

3.1. Постановка задачи

Исходя из анализа существующих решений можно сделать вывод, что не существует интерактивного визуализатора, отображающего все заявленные модели программ, поэтому необходимо разработать систему для интерактивной визуализации моделей кода на языке Java, а именно:

- AST
- CFG
- DDG
- PDG
- ASG
- SSA

Для выполнения этой задачи требуется:

- Разработать требования к системе
- Спроектировать архитектуру сервиса
- Проанализировать и выбрать средства решения
- Разработать сервис в соответствии с установленными требованиями
- Протестировать и отладить разработанный сервис

3.2. Разработка требований к системе

Необходимо разработать сервис, позволяющий получать код метода на языке Java от пользователя и строить по этому коду визуализацию выбранной модели. Общие требования:

- Сервис должен быть интерактивным

Любое изменение кода должно приводить к изменению модели без дополнительных действий со стороны пользователя. Если введённый код некорректен, сервис должен сообщить пользователю об этом.

- Сервис должен работать на компьютерах, смартфонах и планшетах

Должна быть возможность воспользоваться сервисом с любых устройств, на которых есть браузер.

– Сервис должен позволять строить все обозначенные модели через единый интерфейс

Это AST, CFG, DDG, PDG, ASG и SSA. У пользователя должна быть возможность переключаться между моделями.

– Удобная навигация по большим моделям

Должна быть возможность изменения масштаба изображения модели, а также возможность его перемещения внутри области просмотра.

– Элементы графов должны размещаться детерминировано

Для одного и того же кода необходимо всегда получать одинаковую визуализацию всех моделей. Уточняющие требования по AST:

– В дереве должны присутствовать только семантически значимые узлы
– Должна быть возможность сворачивать и разворачивать поддеревья

Уточняющие требования по CFG:

– Необходимо использовать стандартные формы для обозначения узлов

Эллипс для обозначения начала или конца, прямоугольник для обычных операторов, ромб для узлов принятия решений.

– Метки на рёбрах из узлов принятия решений

Рёбра, выходящие из узлов принятия решений, должны иметь обозначения, дающие понять пользователю, по какому ребру переходить в случае, если условие истинно, а по какому в случае, если ложно.

– Граф не должен быть привязан к языку программирования

Нужно убрать типы данных из названий узлов, а также объявления переменных без инициализации. Конструкции if, for и while должны быть отображены как условия с необходимыми стрелками. Инструкции break и continue не должны появляться как самостоятельные узлы, а должны только влиять на расположение стрелок.

3.3. Выводы по главе

В этой главе была поставлена задача, которую необходимо решить в рамках выпускной квалификационной работы и определены требования для реализации сервиса интерактивной визуализации моделей программ.

ГЛАВА 4. ПроЕКтИроВАнИЕ АрхИтЕКтуры И ВыБор срЕДстВ рЕшЕнИя

Для реализации интерактивного визуализатора моделей программ необходимо решить следующие подзадачи:

- Получение из исходного кода метода на языке Java абстрактного синтаксического дерева
- Обеспечение отрисовки деревьев и графов в браузере
- Преобразование структуры AST в данные для отрисовки AST, CFG, DDG, PDG, ASG и SSA
- Создание интерфейса для взаимодействия с программой

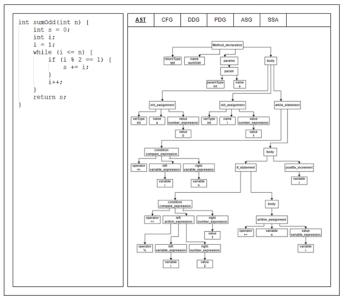


Рис.4.1. Схематичное представление интерфейса визуализатора

4.1. Парсинг кода

Для парсинга кода можно использовать существующие парсеры, либо же разработать свой.

- Существующие парсеры

Плюсы: не нужно тратить время на разработку, наверняка работает как ожидается. Минусы: обычно работает не так, как хотелось бы, очень сложно изменить что-то, обычно решение громоздкое, нужно тратить время, чтобы найти готовое решение, и разобраться в нём.

- Разработка своего парсера

Плюсы: можно разработать именно так, как нужно, достаточно компактно, без лишнего нагромождения функционала. Минусы: Процесс слишком долгий и трудоемкий. Из готовых решений хочется выделить ANTLR и JavaParser.

4.1.1. ANTLR

ANTLR. Универсальный парсер, в котором есть готовые грамматики почти под любой язык, в том числе Java. Результатом работы является parse tree. Если мы хотим избавиться от лишней для AST информации, то требуется дописать большое количество программного кода. Также у данного парсера нет сериализации в JSON.

4.1.2. JavaParser

JavaParser заточен под Java и выдаёт структуру только из семантически значимых элементов, что очень удобно. Реализована встроенная сериализация в JSON, но выдаётся куча вспомогательной информации, AST из такого строить неудобно, поэтому требуется писать упрощённый сериализатор. Исходя из всех вышеуказанных плюсов и минусов было принято решение воспользоваться JavaParser. Он не работает в браузере, поэтому нужен сервер. Таким образом, нужно использовать клиент-серверную архитектуру. Будет приложение в браузере, которое отправляет запросы на сервер, чтобы получить результат работы JavaParser.

4.2. Выбор фреймворка для сервера

Для Java существует множество различных http-фреймворков. Самый популярный из них Spring, так же существует очень простой фреймворк Spark.

- Spring - очень мощный и большой фреймворк, популярный и сложный. Избыточный для поставленных задач
- Spark - простой и минималистичный, вполне достаточный для наших задач

4.3. Выбор способа визуализации

Для выбора способа визуализации нужно определиться, использовать готовую библиотеку или писать свое решение. Существует множество библиотек для отрисовки графов, требуется выбрать наилучшую для реализации поставленных задач.

- Dracula Graph Library – размещает узлы хаотично, а так же сам вид визуализации не подходит для требуемого сервиса.

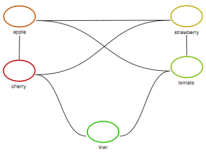


Рис.4.2. Пример результата, выдаваемого библиотекой Dracula Graph Library

– Vis.js – размещает узлы не так, как хотелось бы (по принципу “упругого” физического движка), а так же сам вид визуализации не подходит для требуемого сервиса.

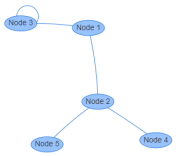


Рис.4.3. Пример результата, выдаваемого библиотекой Vis.js

– Cytoscape.js – так же размещает узлы хаотично.

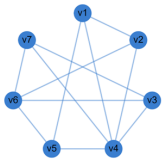


Рис.4.4. Пример результата, выдаваемого библиотекой Cytoscape.js

– JavaScript InfoVis Toolkit (JIT) - неплохо рисует деревья, но плохо рисует графы. Как и предыдущие библиотеки хаотично размещает узлы и ребра и не подходит по виду визуализации.

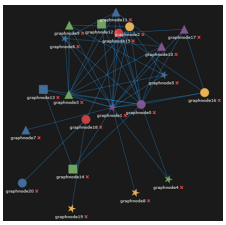


Рис.4.5. Пример результата, выдаваемого библиотекой JIT

– D3.js – позволяет визуализировать очень много всего, но не то, что нужно для нашего сервиса. Тоже имеет “упругое” размещение и не обладает нужным видом.
– Sigma.js - требует задания координат в XML и не обладает нужным видом.
– Graphviz - был скомпилирован для Javascript с использованием Emscripten в формат WebAssembly. Существует в виде плагина к библиотеке D3.js. Умеет визуализировать графы с настраиваемым размещением и видом узлов, а также, отображает графы, описанные на языке DOT – универсаль-

Рис.4.6. Пример результата, выдаваемого библиотекой D3.js

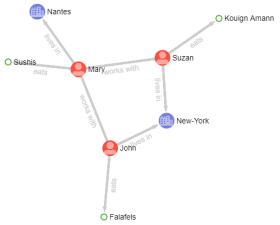


Рис.4.7. Пример результата, выдаваемого библиотекой Sigma.js

ном языке описания графов. Так же стоит рассмотреть вариант создания собственной визуализации.

– Плюсы: достаточно компактное решение, которое делает именно то, что нужно
– Минусы: сложная логика размещения и очень затратное по времени решение

Исходя из всего вышесказанного выбор пал на d3-graphviz, так как это наиболее подходящий вариант для нашего сервиса.

4.4. Интерфейс взаимодействия с программой

Требуется создать интерфейс для взаимодействия с программой, для этого нужно определиться использовать или не использовать фреймворк. Фреймворк – это инструмент для построения приложений. Обычно фреймворки определяют структуру приложения. Кроме того, они содержат множество уже готовых модулей, упрощающих разработку. В случае с JavaScript фреймворки помогают писать декларативный код, а не императивный. При использовании фреймворка обычно используются сборщики проекта. Как правило, настройка сборщика по умолчанию уже включает в себя компилятор babel, а также минификацию HTML, CSS и JavaScript кода. Минификация уменьшает размер и количество подключаемых файлов, что положительно сказывается на времени загрузки страниц. Babel позволяет писать код на новых версиях JavaScript, и он будет преобразован в JavaScript более старых версий, что увеличивает поддержку браузерами. Кроме этого, фреймворки добавляют плагины к babel, что позволяет использовать синтаксические

конструкции, которых нет в чистом JavaScript. Кроме того, фреймворки позволяют запускать сервер для разработки, благодаря которому происходит отслеживание изменений в коде в режиме реального времени, и все изменения видны в браузере сразу без перезагрузки страницы. Таким образом, можно сделать вывод, что фреймворки ускоряют и упрощают процесс разработки. Минус фреймворка заключается в том, что его код также включается в сборку, из-за чего JavaScript кода, который нужно загрузить в браузер, становится больше, что в некоторой степени замедляет первоначальную загрузку страницы. Было принято решение использовать фреймворк.

4.5. Выбор фреймворка

1. React
   - Плюсы: большое сообщество, множество дополнительных библиотек, самый популярный на сегодняшний день, использует Virtual DOM (что улучшает производительность)
   - Минусы: громоздкий синтаксис, относительная сложность освоения, плохо структурированная документация, нестрогая структура приложения
2. Angular
   - Плюсы: огромный, мощный фреймворк, встроенная строгая типизация, множество встроенных возможностей, строгая структура приложения, хорошая документация
   - Минусы: сложно освоить, не использует Virtual DOM, большой размер, излишний для небольших проектов
3. Vue.js
   - Плюсы: очень простой и лёгкий в освоении, использует Virtual DOM (что улучшает производительность), хорошая документация, достаточно популярный
   - Минусы: не очень развитое сообщество, нестрогая структура приложения
4. Svelte
   - Плюсы: минимальный размер выходного приложения, простой в освоении, максимально производительный код на выходе
   - Минусы: очень молодой, непопулярный проект, маленькое сообщество

Проанализировав плюсы и минусы различных фреймворков, было принято решение, что для создания визуализатора самым подходящим будет Vue.js. Для обращений

к серверу решено использовать библиотеку axios, как самое популярное решение для подобных задач, а в качестве готовых стилей для оформления интерфейса выбран CSS-фреймворк bulma.

4.6. Выводы по главе

В данной главе описана архитектура и были выбраны средства решения. При создании сервиса будет использована клиент-серверная архитектура с использованием фреймворка Spark на стороне сервера и фреймворка Vue.js на стороне клиента. В качестве дополнительных инструментов будут использоваться JavaParser на стороне сервера для парсинга кода, Gson для сериализации в JSON. На стороне клиента будут использоваться axios для сетевых запросов, bulma для оформления интерфейса, d3 и d3-graphviz для отрисовки деревьев и графов. Когда пользователь

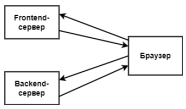


Рис.4.8. Схема работы сервиса

открывает сервис в браузере, он обращается к Frontend серверу, откуда получает статические файлы HTML, CSS, JavaScript и другие, далее браузер строит по этим файлам страницу. Когда пользователь вводит код в интерфейсе, происходит запрос к Backend-серверу (его адрес прописывается в JavaScript от Frontend-сервера). Ответ от сервера обрабатывается браузером при помощи JavaScript-кода с Frontend-сервера.

ГЛАВА 5. ÈАЗрАБотКА сЕрВИсА ИнтЕрАКтИВноГо ВИЗуАЛИЗАторА МоДЕЛЕЙ проГрАММ

В данном разделе описывается разработка сервиса интерактивного визуализатора моделей программ. Для разработки сервера выбран язык программирования Java, для разработки клиента – JavaScript.

5.1. Разработка сервера

Главная задача сервера – отвечать по протоколу HTTP на запросы от клиента. В нашем случае будет одна основная точка доступа, на вход которой подаётся исходный код на языке Java, а в качестве ответа ожидается JSON, в котором будет содержаться структура AST в двух видах – в облегчённом (для рисования) и в расширенном (с указанием связей с исходным кодом). Для реализации этой логики были созданы следующие классы:

- Main – содержит главный метод приложения. В нём описаны точки доступа и их поведение. В том числе точка доступа для обеспечения корректной работы кросс-доменных запросов (настройка CORS), и главная точка доступа приложения для преобразования кода в AST.
- JsonPrinter – класс для преобразования объектной структуры AST в облегчённую версию JSON. Этот класс был удалён из библиотеки JavaParser. Разработчики изменили способ сериализации, оставив только возможность генерации JSON расширенной версии AST. Расширенная версия включает в себя дополнительную информацию, в том числе номера начальных и конечных строк и столбцов для каждого узла, номера токенов, а также пустые ключи для всех синтаксически допустимых элементов, не встречающихся в коде (например, аннотаций для методов и переменных, бросаемые исключения и прочее). Поскольку облегченная версия всё-таки оказалась нужна, пришлось добавить этот класс.
- Response – структурный класс, содержащий поля status и data. Создан для стандартизации вида ответов сервера (status может иметь значение “success” или “error”, data – любые данные). Задаёт структуру ответа сервера при сериализации в JSON.
- AstResponse – структурный класс, содержащий поля simple и extended (для облегчённой и расширенной версии AST, соответственно). Используется в

основной точке доступа как значение поля data в объекте Response при ответе.

5.2. Разработка клиента

Клиент включает в себя практически всю логику приложения. Ниже описана разработка для каждой модели.

5.2.1. Построение AST

Для построения AST нужно использовать облегчённую версию структуры, полученной от сервера. Задача сводится к преобразованию этой иерархической структуры в код графа на языке DOT. Для этого необходимо описать каждый узел дерева и каждое ребро, соединяющее узлы. Кроме того, необходимо реализовать функционал сворачивания и разворачивания поддеревьев по клику на узел. Для получения массива узлов и массива рёбер нужно выполнить рекурсивный обход AST-структуры. В зависимости от типа полей требуется выполнять разные действия. Если тип поля – массив, нужно добавить узел с именем поля, на каждом элементе рекурсивно вызвать обход, добавляя рёбра от только что созданного узла к корневым узлам поддеревьев, созданных в каждом внутреннем обходе. Если тип поля – объект, нужно отделить поле с типом узла от остальных (значение этого поля установится как имя узла). Для всех остальных полей нужно запустить рекурсивный обход, добавляя необходимые рёбра. Если тип поля – примитив, то и название, и значение поля будут установлены в названии узла. Для реализации функционала сворачивания и разворачивания поддеревьев необходимо каким-то образом обозначить связи между отображаемыми узлами дерева, чтобы можно было определять, является ли тот или иной узел дерева потомком другого. Это можно делать по данным из массива рёбер, но такой алгоритм будет неэффективным. Поэтому было принято решение сделать у каждого узла ссылку на родительский узел. Пройдя циклом по цепочке родительских элементов, можно понять, является ли какой-то узел потомком другого. Далее создадим множество из индексов узлов для хранения информации о корнях свёрнутых пользователем поддеревьев. При клике на узел нужно получить индекс того узла, по которому произошёл клик. Если такой индекс есть во множестве, тогда его нужно удалить из множества, а если нет, то добавить. После этого пройти по всем узлам и рёбрам, и пометить скрытыми те

узлы, которые являются потомками хотя бы одного из узлов во множестве, и те рёбра, которые ведут на такие узлы (которые помечаем скрытыми). Для создания кода графа на языке DOT достаточно создать по одной строке для каждого не скрытого узла и для каждого не скрытого ребра, а затем объединить эти строки в соответствии с синтаксисом языка DOT.

5.2.2. Построение CFG

Для построения CFG нужно использовать расширенную версию структуры, полученной от сервера. Задача также сводится к преобразованию этих данных в код графа на языке DOT. Входные данные – AST-дерево, описывающее код. Нужно каким-то образом определять, какие элементы будут являться узлами графа, какой текст будет в них содержаться, и какие пары из них будут соединены ориентированными рёбрами. Кроме того, узлы должны иметь разные формы, в зависимости от их типа, и некоторые рёбра должны иметь подписи “да” и “нет”, в случае если они выходят из узла-условия. AST содержит множество типов узлов. Для удобства обхода этой структуры было решено воспользоваться адаптацией ООП-паттерна «Посетитель» (Visitor). В зависимости от типа узла, будут выполняться различные действия. В частности, необходимо описать различные действия на узлах объявления метода, объявления переменных, блоков кода, циклов while, циклов for, ветвлений if, инструкций return, break и continue. Для всех остальных типов узлов, вложенных непосредственно в перечисленные, будет создаваться отдельный узел графа, текстом для которого будет код, по которому был построен данный узел AST дерева. Была определена процедура добавления узла. Кроме того, что она создаёт узел и добавляет его в массив узлов, она ещё ставит все необходимые рёбра в этот узел. Это реализовано за счёт создания глобального списка частично определённых рёбер (для удобства обозначим аббревиатурой ЧОР, у которых указаны начало и тип, но не указан конец). Эта процедура извлекает все значения из этого списка, добавляет в каждое ЧОР конечным узлом себя, и добавляет каждое полученное ребро в массив рёбер. Также добавляет в список ЧОР единственным элементом ЧОР, в котором началом указан новый узел, а типом – ребро без подписи. Дальнейшие инструкции (после процедуры) могут исправлять добавленное ребро. Действия на узлах объявления метода: создание узла, добавление ЧОР. Имя узла собирается из имени метода и имён параметров метода через запятую в скобках. Действия на узлах объявления переменных: пройти по всем

объявляемым переменным. Для каждой переменной, если есть инициализирующее значение, добавляем узел процедурой. Название узла составляется из имени переменной, знака равно и кода выражения, инициализирующего переменную. Действия на узлах блоков кода: запустить обход для каждого оператора в блоке. Действия на узлах циклов while. Добавляется узел с помощью процедуры. Тип узла – условие, название – код условия. После этого изменяется тип пока единственного ЧОР на ребро с подписью “да”. Для корректной работы с инструкциями break и continue глобально был создан стек для циклов. Для текущего цикла создаётся объект, который будет содержать два массива ЧОР – ведущих в инструкцию break и ведущих в инструкцию continue. Этот объект добавляется в стек. Далее запускается обход тела цикла. При обходе заполняются массивы в объекте. Затем все ЧОР из массива для continue добавляются в список ЧОР. Далее во все ЧОР из списка добавляется узел условия конечным пунктом, и все они извлекаются из списка и записываются в массив рёбер. Потом в список ЧОР добавляется ЧОР из узла условия с подписью “нет”, а также все ЧОР из массива для break. Объект с массивами удаляется из стека. Действия на узлах циклов for. Происходит всё то же самое, что с while, только вначале обходятся узлы из блока инициализации, в блоке условия может не быть кода, в этом случае будет вписано true, и после добавления всех ЧОР из массива для continue добавляется обход секции обновления переменных. Действия на узлах ветвлений if. Добавляется узел с помощью процедуры. Тип узла – условие, название – код условия. После этого изменяется тип пока единственного ЧОР на ребро с подписью “да”. Обходом добавляется тело блока if. Затем из списка ЧОР извлекаются все ЧОР во временный список. Затем в основной список ЧОР добавляется ЧОР из узла с условием с подписью “нет”. Если у конструкции присутствует блок else, обходом добавляется его тело. После этого в любом случае все ЧОР из временного списка добавляются обратно в основной. Действия на узлах инструкций return. Добавляется узел с помощью процедуры. Тип узла – return, название – return и код выражения под оператором return. После добавления узла очищается список ЧОР. Действия на узлах инструкций break. Получаем массив для break из верхнего элемента стека. В него извлекаем все ЧОР из списка ЧОР. Действия на узлах инструкций continue. Аналогично break, только с массивом для continue. Действия на узлах остальных типов, вложенных непосредственно в узлы описанных выше типов. Добавляется узел с помощью процедуры. Тип – обычный, название – код узла из AST. Нужно запустить обход от корня AST дерева. Этот обход сформирует массивы из узлов и рёбер для отрисовки. При отрисовке

нужно узлы с типами объявление метода и return рисовать овалами, узлы-условия ромбами, а обычные – прямоугольниками.

5.2.3. Построение DDG и PDG

Для построения DDG и PDG тоже используется расширенная версия AST, полученная от сервера. Узлы и рёбра можно получить по тому же алгоритму, который использовался для построения CFG с небольшим дополнением. В каждый узел нужно вставить ссылку на элемент исходного AST. Это необходимо для определения переменных, которые были считаны и записаны в этом узле. Дополнительно нужно сформировать массив рёбер, соединяющих узлы, в которых происходит запись значений переменных, с узлами, в которых происходит возможное чтение этих значений. Назовём его массивом зависимостей. Чтобы сформировать массив зависимостей, нужно у каждого узла составить множество имен переменных, которые в нём получают новые значения, и множество переменных, которые читаются в этом узле. Для составления множества читаемых переменных по данному узлу, нужно сделать обход элемента AST, соответствующего узлу. Для такого обхода будем использовать подход, аналогичный использованному при построении CFG (с применением адаптации паттерна Visitor), только типы узлов и действия будут отличаться. В данном случае нужно обрабатывать все типы узлов, которые могут внутри себя иметь чтение переменных. Во всех таких узлах нужно запускать обходы на тех частях, где возможно чтение переменных. В итоге при попадании в узел с типом имя переменной, нужно добавить имя во множество. Аналогично можно действовать для получения множества записываемых переменных для каждого узла. Теперь нужно для каждого узла составить набор из переменных, доступных узлу (которым присвоено значение в цепочке до него), и для каждой такой переменной сделать множество из возможных источников значения этой переменной. Это можно сделать алгоритмом, похожим на рекурсивный обход графа в глубину от первого узла. Таким образом в графе будут пройдены все возможные пути, в том числе циклы. И на каждый шаг каждого пути будет передаваться объект контекста, ключами которого будут все записанные переменные, а значениями узлы, на которых произошла последняя запись. При попадании на узел во множества источников добавляются все источники из контекста. Если никаких изменений не произошло (все источники ранее уже были добавлены), и все соседние узлы посещены, то это сигнал к завершению рекурсии. Используя

построенные наборы, можно теперь составить массив зависимостей. Для этого нужно пройти по каждому узлу. В каждом узле по всем считанным переменным. Для каждой переменной перебрать множество источников. В зависимости нужно добавить ребро между источником и текущим узлом. Отрисовка будет работать аналогично CFG, только для DDG нужно скрыть рёбра из CFG, а для PDG не нужно. После этого нужно добавить рёбра из зависимостей пунктирными стрелками, не меняющими положение узлов (чтобы узлы располагались аналогично CFG).

5.2.4. Построение ASG

При построении ASG используется облегчённая версия структуры, полученной от сервера. Теперь к AST нужно добавить рёбра от узлов объявления переменных к узлам, где они используются. И ещё рёбра последовательности инструкций, как в CFG. Для получения узлов и рёбер можно использовать алгоритм, реализованный для отрисовки AST. Только теперь для каждого узла, созданного из объекта, нужно добавить ссылку на этот объект. Для получения рёбер последовательности инструкций можно использовать алгоритм, аналогичный CFG. Только в алгоритме построения CFG узлы графа создавались при обходе, а теперь необходимо использовать узлы, созданные алгоритмом построения AST. Соответствие по узлам можно установить, опираясь на добавленные ссылки в узлах AST. Чтобы это было проще, можно сделать map, в котором ключами будут объекты исходной структуры, а значениями индексы узлов AST, которым они соответствуют. Для этого достаточно сделать один проход по массиву узлов AST, поскольку в каждом таком узле содержится ссылка на исходный объект. Для получения рёбер, соединяющих узлы с объявлением переменных с узлами с использованием переменных, нужно для каждой объявленной переменной получить индекс узла с её объявлением, и индексы узлов с её использованием. Объекты с именами переменных в исходной структуре имеют определённый тип. По этому типу можно найти все упоминания переменных. Чтобы понять, является это упоминание объявлением или использованием, нужно посмотреть на тип родительского элемента этого объекта. Если родительский элемент имеет тип объявление переменной или параметр метода, упоминание считается объявлением, в противном случае использованием. Теперь достаточно для каждого найденного объявления переменной добавить по одному ребру к каждому использованию. При отрисовке ребра между соседними

инструкциями для наглядности обозначаются красными пунктирными стрелками, а рёбра между объявлениями и использованием переменных пунктирными синими.

5.2.5. Построение SSA

При построении SSA используется расширенная версия структуры, полученной от сервера. Для получения узлов и рёбер графа можно использовать алгоритм, аналогичный CFG. Только теперь названия узлов не указываются на этапе создания узлов. Они будут добавлены позже. После этого нужно добавить в каждый узел информацию о считанных и записанных переменных в этом узле, аналогично DDG и PDG. Далее нужно установить версии для объявлений и использований каждой переменной. Сложность заключается в том, что значения переменных могут устанавливаться в разных ветвях программы, и затем, когда ветви сходятся, по правилам построения SSA, необходимо добавить узел для каждой переменной, которая имеет разные версии на сходящихся ветвях. В этом узле нужно обозначить новую версию переменной, в которую записывается результат вызова Фи-функции, аргументами которой передаются все версии переменной на сходящихся ветвях. Кроме того, нумерация версий переменных должна происходить в порядке появления в коде, но с учётом добавленных Фи-определений. Для начала нужно определить узлы, в которых сходятся несколько ветвей программы. Для этого достаточно посчитать для каждого узла, в скольких рёбрах он является конечным. Если количество рёбер больше одного, то перед этим узлом могут появиться Фи-определения. Назовём такой узел узлом схождения. Для установки версий переменных можно воспользоваться алгоритмом, аналогичным рекурсивному обходу графа в глубину от первого узла. Таким образом в графе будут пройдены все возможные пути, в том числе циклы. В каждый рекурсивный вызов будет передаваться узел, на который попадаем, а также индекс узла, из которого попадаем в этот узел. При попадании в какой-либо узел, нужно определить контекст переменных, который приходит от предыдущего узла (какие переменные существуют, и какие их версии передаются). Контекст будет кэшироваться в каждом узле. Изначально у всех узлов зададим пустой контекст. При попадании на узел, входной контекст будет определяться как закэшированный контекст прошлого узла, на который накладываются версии записанных в том узле переменных. При попадании на узел схождения, для каждой переменной нужно проверить, отличается ли версия в закэшированном контексте от версии во входном контексте. В случае, если не отличается, ничего делать

не надо. В противном случае нужно сравнить версию этой переменной с той, которая была прислана в прошлый раз с этого же узла. Если это первый вход с этого узла, либо если версии отличаются, это сигнал к продолжению рекурсии, в противном случае к выходу из рекурсии. Далее, если количество различных версий переменной среди последних пришедших по каждой из ветвей равно единице, то достаточно переопределить версию переменной в закэшированном контексте на единственную полученную. В противном случае, если ранее это не было сделано, в закэшированном контексте необходимо установить новую версию для данной переменной. При попадании на узел, не являющийся узлом схождения, достаточно обновить закэшированный контекст на входной (переписать все версии из входного контекста в закэшированный). Такие узлы всегда продолжают рекурсию. Затем, вне зависимости от типа узла, если в нём присутствуют записи в переменные, и при этом узел ранее не был посещён, для каждой записываемой переменной создаём новую версию и сохраняем их в отдельный объект, который будет содержать версии всех записанных в узле переменных. Далее, если рекурсию нужно продолжать, функция запускается на всех узлах, в которые можно перейти из текущего. После этого обхода в каждом узле будет храниться информация о версиях используемых переменных, но пока эти версии будут стоять в неправильном порядке, поскольку устанавливались в порядке обхода в глубину, а не в порядке следования в коде. Далее, чтобы исправить порядок версий, нужно пройти по узлам в порядке следования в коде. При проходе нужно смотреть на порядок первых упоминаний версий переменных в закэшированных контекстах узлов и объектах с версиями записываемых переменных. Таким образом, для каждой переменной получится порядок упоминания её версий в коде, который может отличаться от естественного. Нужно поставить в соответствие каждой такой версии номер её упоминания, и исправить в закэшированных контекстах и объектах с версиями записываемых переменных версии на номера их упоминаний. Далее сложность заключается в том, что теперь каким-то образом нужно вставить версии переменных в названия узлов при отрисовке. При построении CFG и ему подобных графов для названий узлов использовался непосредственно исходный код. Он извлекался с помощью координат, указанных в каждом объекте структуры, полученной с сервера (указаны номер начальной строки, номер конечной строки, номер столбца в начальной строке и номер столбца в конечной строке). По этим данным можно было извлечь необходимый участок кода. Теперь так сделать не получится, поскольку в исходном коде версии переменных не указаны. Чтобы это исправить, исходный код был

преобразован в матрицу из строк по одному символу. Далее необходимо было пройти по всем узлам, и для каждой записанной переменной из узла в матрицу к последнему символу имени переменной дописать версию. Аналогично для чтения переменных. Таким образом, ранее описанные координаты будут определять участки матрицы, а не исходного кода, а в матрице некоторые элементы станут длиной более одного символа, что не влияет на нумерацию строк и столбцов. Теперь достаточно написать функцию, которая сможет по координатам взять участок матрицы и собрать из него строку, в которой уже будут присутствовать версии переменных. Когда это готово, можно пройти по каждому узлу и добавить название. Для некоторых типов узлов логика построения названия будет отличаться от CFG. Например, присваивания с коротким синтаксисом (с использованием операторов +=, -=, *= и т. п.), инкремент и декремент нужно записать в ином виде, нежели в коде, поскольку они и записывают, и читают переменную, причем разные её версии. При отрисовке узлов схождения для каждой переменной, у которой в итоге с разных ветвей идут разные версии, необходимо добавить узел с Фи-функцией. Это также меняет рёбра. Все рёбра, которые шли в этот узел схождения, должны идти в первый Фи-узел. Остальные Фи-узлы, если они есть, выстраиваются в цепочку за первым, и от последнего Фи-узла ребро должно идти к исходному узлу схождения. В остальном логика отрисовки аналогична CFG.

5.3. Выводы по главе

В данной главе были описаны общие детали реализации интерактивного визуализатора моделей программ, такие как структура сервера и процесс построения абстрактного синтаксического дерева, графа потока управления, графа зависимости по данным, графа зависимости программы, абстрактного семантического графа и представления на основе статического однократного присваивания.

ГЛАВА 6. ЂЕстИроВАнИЕ И АнАЛИЗ рЕЗуЛьтАтоВ

В данной главе будет рассмотрено тестирование сервера и клиента визуализатора. Так как сервис разработан в учебных целях, тестирование производится вручную за счет анализа методов, написанных на Java для каждой модели. Тестирование сервера производится с помощью утилиты Postman.

6.1. Тестирование сервера

Тестирование сервера производилось с помощью утилиты Postman. В нашем случае будет есть одна основная точка доступа, на вход которой подаётся исходный код на языке Java, а в качестве ответа ожидается JSON, в котором содержится структура AST в двух видах – в облегчённом (simple) и в расширенном (extended). В запросе мы подаем на вход исходный код, который требуется парсить, а на

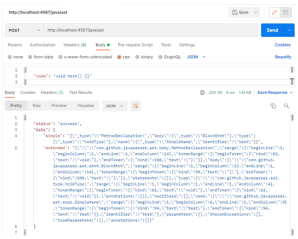


Рис.6.1. Тестирование сервера визуализатора

выходе получаем результат работы JavaParser в двух видах: simple (который потребуется для рисования) и extended (с указанием связей с исходным кодом и прочей информацией). Исходя из результата тестирования можно сделать вывод, что сервер работает так, как надо.

6.2. Тестирование клиента

Тестирование клиента проводится вручную в таких браузерах, как Яндекс Браузер, Google Chrome, Firefox, Opera, Edge за счет анализа ряда методов, написанных на Java для каждой модели. Ниже приведены рисунки с результатом работы визуализатора для одного из методов (sumOdd, упомянутый в первой главе) в каждом браузере. Как можно увидеть на рисунке 21, в дереве присутствуют только семантически значимые узлы. Если блок подсвечен синим цветом, то это означает, что все его наследники скрыты. На построенном CFG используются стандартные формы для обозначения узлов: эллипс для обозначения начала или

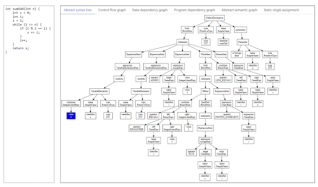


Рис.6.2. Построенное AST в браузере Edge

Рис.6.3. Построенный CFG в Яндекс браузере

конца, прямоугольник для обычных операторов, ромб для узлов принятия решений. Рёбра, выходящие из узлов принятия решений, имеют обозначения, дающие понять, по какому ребру переходить в случае, если условие истинно, а по какому в случае, если ложно. На рисунке 24 видно, что при отрисовке ребра между соседними

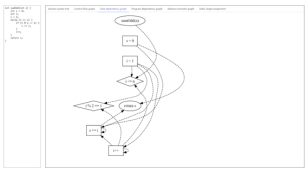


Рис.6.4. Построенный DDG в браузере Chrome

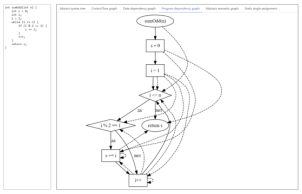


Рис.6.5. Построенный PDG в браузере Opera

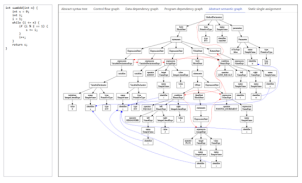


Рис.6.6. Построенный ASG в браузере Firefox

инструкциями для наглядности обозначаются красными пунктирными стрелками, а рёбра между объявлениями и использованием переменных пунктирными синими.

Если код написан неверно, то интерфейс подсвечивается красным цветом и отрисовка и изменение уже нарисованной модели не происходит.

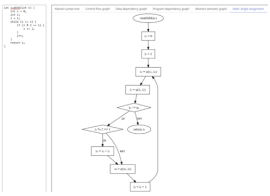
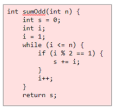


Рис.6.7. Построенное SSA в Яндекс браузере



```
int sumOdd(int n) {
    int s = 0;
    int i;
    i = 1;
    while (i <= n) {
        if (i % 2 == 1) {
            s += i;
        }
        i++;
    }
    return s;
```

Рис.6.8. Неправильно введенный код

6.3. Выводы по главе

В данной главе рассмотрено тестирование сервера и клиента визуализатора. Сервер протестирован с помощью утилиты Postman, а клиент тестировался в разных браузерах для разных методов, написанных на Java. В отчете приведен один из них. Из результатов тестирования можно сделать вывод, что сервис работает, так как надо.

ЗАКЛючЕнИЕ

В рамках выпускной квалификационной работы был разработан сервис интерактивной визуализации моделей программ, как средство для улучшения понимания устройства абстрактного синтаксического дерева, графа потока управления, графа зависимостей по данным, графа зависимости программы, абстрактного семантического графа и представления на основе однократного статического присваивания. Сперва были рассмотрены аналоги, их плюсы и минусы. Исходя из обзора был сделан вывод, что на данным момент не существует универсального и хорошего продукта, у всех есть какие-либо недостатки, так же не существует ни одного сервиса, который мог бы построить несколько заявленных моделей. После разбора были получены требования к системе, выбрана библиотека для рисования, парсер кода и фреймворк для сервера и клиента и была описана разработка. Итоговая версия визуализатора имеет клиент-серверную архитектуру, сервер написан на языке программирования Java, использует парсер JavaParser и фреймворк Spark. Клиент же написан на языке программирования JavaScript с использованием фреймворка Vue.js, библиотекой axios для сетевых запросов, фреймворка bulma для оформления интерфейса и использует библиотеку d3-graphviz для визуализации моделей. Сервис был протестирован за счет анализа ряда методов, написанных на Java. Ожидаемые деревья и графы совпали с теми, которые рисует визуализатор и сейчас им можно пользоваться в образовательных целях. В будущем планируется добавление других языков программирования и расширение функционала.

ЁпИсоК ИспоЛьЗоВАнных ИсточнИКоВ

Приложение 1

Краткие инструкции по настройке издательской системы LaTeX

В SPbPU-BCI-template *автоматически выставляются необходимые настройки и в исходном тексте шаблона приведены примеры оформления текстово-графических объектов*, поэтому авторам достаточно заполнить имеющийся шаблон текстом главы (статьи), не вдаваясь в детали оформления, описанные далее. Возможный «быстрый старт» оформления главы (статьи) под Windows следующий[П1.1]:

A. Установка полной версии MikTeX [**latex-miktex**]. В процессе установки лучше выставить параметр доустановки пакетов «на лету».
B. Установка TexStudio [**latex-texstudio**].
C. Запуск TexStudio и компиляция `my_chapter.tex` с помощью команды «Build&View» (например, с помощью двойной зелёной стрелки в верхней панели). *Иногда, для достижения нужного результата необходимо несколько раз скомпилировать документ.*
D. В случае, если не отобразилась библиография, можно
   - воспользоваться командой Tools → Commands → Biber, затем запустив Build&View;
   - настроить автоматическое включение библиографии в настройках Options → Configure TexStudio → Build → Build&View (оставить по умолчанию, если сборка происходит слишком долго): `txs:///pdflatex | txs:///biber | txs:///pdflatex | txs:///pdflatex | txs:///view-pdf`.

В случае возникновения ошибок, попробуйте скомпилировать документ до последних действий или внимательно ознакомьтесь с описанием проблемы в log-файле. Бывает полезным переход (по подсказке TexStudio) в нужную строку в pdf-файле или запрос с текстом ошибке в поисковиках. Наиболее вероятной проблемой при первой компиляции может быть отсутствие какого-либо установленного пакета LaTeX.

В случае корректной работы настройки «установка на лету» все дополнительные пакеты будут скачиваться и устанавливаться в автоматическом режиме. Если доустановка пакетов осуществляется медленно (несколько пакетов за один запуск

[П1.1] Внимание! Пример оформления подстрочной ссылки (сноски).

компилятора), то можно попробовать установить их в ручном режиме следующим образом:

1. Запустите программу: меню → все программы → MikTeX → Maintenance (Admin) → MiKTeX Package Manager (Admin).
2. Пользуясь поиском, убедитесь, что нужный пакет присутствует, но не установлен (если пакет отсутствует воспользуйтесь сначала MiKTeX Update (Admin)).
3. Выделив строку с пакетом (возможно выбрать несколько или вообще все неустановленные пакеты), выполните установку Tools → Install или с помощью контекстного меню.
4. После завершения установки запустите программу MiKTeX Settings (Admin).
5. Обновите базу данных имен файлов Refresh FNDB.

Для проверки текста статьи на русском языке полезно также воспользоваться настройками Options → Configure TexStudio → Language Checking → Default Language. Если русский язык «ru_RU» не будет доступен в меню выбора, то необходимо вначале выполнить Import Dictionary, скачав из интернета любой русскоязычный словарь.

Далее приведены формулы (П1.2), (П1.1), рис.П1.2, рис.П1.1, табл.П1.2, табл.П1.1.

$$\pi \approx 3{,}141. \tag{П1.1}$$

Рис.П1.1. Вид на гидробашню СПбПУ [**spbpu-gallery**]

Таблица П1.1

Представление данных для сквозного примера по ВКР [**Peskov2004**]

G	m_1	m_2	m_3	m_4	K
g_1	0	1	1	0	1
g_2	1	2	0	1	1
g_3	0	1	0	1	1
g_4	1	2	1	0	2
g_5	1	1	0	1	2
g_6	1	1	1	2	2

П1.1. Параграф приложения

П1.1.1. Название подпараграфа

Название подпараграфа оформляется с помощью команды `\subsection{...}`. Использование подподпараграфов в основной части крайне не рекомендуется.

П1.1.1.1. Название подподпараграфа

$$\pi \approx 3{,}141. \tag{П1.2}$$

Рис.П1.2. Вид на гидробашню СПбПУ [**spbpu-gallery**]

Таблица П1.2

Представление данных для сквозного примера по ВКР [**Peskov2004**]

G	m_1	m_2	m_3	m_4	K
g_1	0	1	1	0	1
g_2	1	2	0	1	1
g_3	0	1	0	1	1
g_4	1	2	1	0	2
g_5	1	1	0	1	2
g_6	1	1	1	2	2

Приложение 2

Некоторые дополнительные примеры

В приложении[П2.1] приведены формулы (П2.2), (П2.1), рис.П2.2, рис.П2.1, табл.П2.2, табл.П2.1

$$\pi \approx 3{,}141. \qquad \text{(П2.1)}$$

Рис.П2.1. Вид на гидробашню СПбПУ [**spbpu-gallery**]

Таблица П2.1

Представление данных для сквозного примера по ВКР [**Peskov2004**]

G	m_1	m_2	m_3	m_4	K
g_1	0	1	1	0	1
g_2	1	2	0	1	1
g_3	0	1	0	1	1
g_4	1	2	1	0	2
g_5	1	1	0	1	2
g_6	1	1	1	2	2

[П2.1] Внимание! Пример оформления подстрочной ссылки (сноски).

П2.1. Подраздел приложения

$$\pi \approx 3{,}141. \qquad \text{(П2.2)}$$

Рис.П2.2. Вид на гидробашню СПбПУ [**spbpu-gallery**]

Таблица П2.2

Представление данных для сквозного примера по ВКР [**Peskov2004**]

G	m_1	m_2	m_3	m_4	K
g_1	0	1	1	0	1
g_2	1	2	0	1	1
g_3	0	1	0	1	1
g_4	1	2	1	0	2
g_5	1	1	0	1	2
g_6	1	1	1	2	2