Федеральное государственное автономное образовательное
учреждение высшего образования
«Южный федеральный университет»

Институт математики, механики и компьютерных наук
им. И.И. Воровича

На правах рукописи

Головешкин Алексей Валерьевич

Устойчивая алгоритмическая привязка к коду программы

2.3.5 — Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание учёной степени
кандидата технических наук

Научный руководитель:
к. ф.-м. н., доц. Михалкович Станислав Станиславович

Ростов-на-Дону — 2022

Оглавление

Введение

Актуальность темы исследования

Многочисленные эксперименты и наблюдения, описанные в научной литературе, показывают, что изучение кода является основной активностью программиста при работе над задачей [1–4] и занимает до 70% рабочего времени [2]. Изучая код, программист находит фрагменты, относящиеся к задаче, после чего наиболее активно перемещается по коду в пределах найденного набора фрагментов. Сохранить информацию о расположении и назначении участков кода, относящихся к задаче, важно в краткосрочной перспективе, чтобы не тратить время на повторные поиски в ходе дальнейшей работы над этой задачей, и в долгосрочной, чтобы не изучать код заново, если к задаче потребуется вернуться. Однако, поскольку речь идёт о программе, находящейся в разработке, важно помнить, что её код может быть существенно отредактирован, и однажды найденный фрагмент кода может сместиться относительно первоначального положения, а также может быть значительно изменён.

Совокупность фрагментов кода, решающих общую задачу, в научной литературе принято называть словом *функциональность* (concern) [5,6]. Функциональности, которые реализуются фрагментами кода, рассредоточенными между элементами приоритетной декомпозиции программы (классами, методами в случае ООП), называются *сквозными* или *прорезающими* (crosscutting) [7]. Прорезающие функциональности присутствуют в большинетве проектов. Например, при разработке компилятора реализация каждой языковой конструкции прорезает грамматику и классы, отвечающие за этапы компиляции, а при разработке веб-сайта реализация очередной возможности, предоставляемой пользователю, может прорезать представления, контроллеры и сервисы.

Результаты, полученные в настоящей диссертационной работе, позволяют сохранить информацию об участках кода, относящихся к задаче-прорезающей функциональности, связать с каждым из них пометку-закладку и по требова-

нию найти текущие версии этих участков в отредактированной программе

Ключевыми понятиями, вводимыми в работе, являются понятия привязки и переприязки. *Привязка* — это построение и сохранение описания участка кода в виде набора структур специального вида, называемых *контекстами*. *Переприязка* — поиск актуальной версии этого участка по его описанию, сохранённому в момент привязки, и повторная привязка к найденному участку. Посредством разработанного нами инструмента с участками кода, к которым произведена привязка, можно связать пометки-закладки. Так формируется *разметка* кода — совокупность пометок, связанных с участками кода, относящимися к прорезающим функциональностям. Пометки, относящиеся к одной функциональности, предполагается объединять в группу. С каждой пометкой и группой может быть связан некоторый комментарий. Таким образом разметка задаёт семантическое (смысловое) описание функциональностей, представленных в коде. Разметка хранится вместе с проектом, но отделена от текста программы. Требование *устойчивости* привязки означает устойчивость к редактированию кода: переприязка должна успешно производиться всегда независимо от того, как поменялся сам участок кода или его окружение.

Построение контекстов, описывающих участок, к которому производится привязка, осуществляется по легковесному абстрактному синтаксическому дереву (АСД) программы, захватывающему достаточный для построения контекстов структурный «каркас» программы — информацию о крупных синтаксических сущностях. Под синтаксической сущностью (синтаксическим элементом) будем понимать часть программы, соответствующую некоторому нетерминальному символу грамматики языка (например, в ООП-программе крупными синтаксическими сущностями будут пространства имён, классы, члены классов). Легковесное дерево можно построить в ходе *легковесного парсинга* [8–10] — варианта синтаксического анализа, при котором подробно анализируются только нужные разработчику парсера части программы. В настоящем диссертационном исследовании предлагается метод легковесного парсинга на основе *упро-*

щённых грамматик. Его актуальность связана не только с задачей устойчивой привязки к коду программы, но и с остальными задачами, в которых применяется легковесный парсинг (реверс-инжиниринг, обработка неправильных программ, обработка вставок кода на другом языке и др.).

В задаче устойчивой привязки легковесные парсеры могут быть быстро прототипированы для языков, к коду на которых необходимо привязаться, и даже для конкретных размечаемых проектов. Самостоятельная генерация легковесных парсеров позволяет работать с деревьями, имеющими единый языконезависимый формат, и формулировать языконезависимые модели и алгоритмы для привязки. Также легковесный парсинг позволяет разбирать неправильные программы, в том числе находящиеся в процессе редактирования, что важно для практической применимости инструмента разметки.

Степень разработанности темы исследования

Основополагающими работами отечественных учёных, посвящёнными вопросам привязки и прорезающих функциональностей, являются труды А. Л. Фуксмана [11] и М. М. Горбунова-Посадова [12–14]. В них предлагается разделять программу на некоторую базовую версию («основу») и сосредоточенно описываемые «расширяющие функции» или «вертикальные слои», инкапсулирующие в себе прорезающие функциональности. Оба автора считают, что прорезающие функциональности должны встраиваться в нужные места базовой программы при генерации интегрированной программы. Привязываться к этим местам предполагается по номеру строки и столбца, а также с помощью поиска подстрок, что абсолютно неустойчиво к редактированию основы.

На идее разделения кода основной программы и кода прорезающих функциональностей основан ряд прикладных подходов к разработке, развиваемых зарубежными исследователями. Наибольшую известность получила парадигма аспектно-ориентированного программирования, предложенная Г. Кичалесом [7, 15, 16]. Помимо неё существуют парадигмы функциональностно-ориентированного (Feature-Oriented) (К. Прехофер и др. [17, 18]) и дельта-ориенти-

рованного (И. Шефер и др. [19, 20]) программирования. Способы, которыми в данных парадигмах осуществляется привязка, также неустойчивы к редактированию: запоминаются имена синтаксических сущностей и сигнатуры методов.

На практике полностью избавиться от прорезающих функциональностей, вплетённых в код проекта, невозможно: многие из них слишком сложно (как идейно, так и физически) отделить от «основной» логики программы. Активно ведутся научные исследования того, как программисты работают с интегрированной программой и сохраняют знание о предназначении её участков [1–4, 21–27]. Полученные в них результаты обозначают ряд проблем, подтверждающих актуальность настоящей работы:

- программисты тратят большую часть рабочего времени на исследование кода, а не на редактирование [1, 2];
- в процессе работы программист вынужден переключаться между задачами, при этом для приостанавливаемой задачи важно сохранить «рабочее состояние», включающее в себя информацию о связанных с ней фрагментах кода [21];
- интенсивность и осмысленность навигации по коду напрямую влияют на успешность выполнения задачи разработчиком [22];
- программисты вынуждены информация о функциональностях постепенно «утекает» из проекта, что обусловлено и свойствами человеческой памяти, и уходом разработчиков, и нехваткой времени на актуализацию документации [23–25].

Разметка кода, основанная на устойчивой привязке, помогает справиться со всем перечисленным.

Устойчивая привязка к коду имеет свою специфику, но является родственной таким хорошо исследованным вопросам, как поиск клонов кода [28–40], поиск плагиата [41–47], восстановление сценариев редактирования программы [48–

53]. Неалгоритмическая (основанная на псевдокомментариях) привязка к произвольным многострочным фрагментам кода применяется в работах С. Ю. Калинина, И. А. Колоколова и А. Н. Литвиненко [54, 55], в CASE-системе Rational Rose (Rational Rhapsody) [56,57], в инструменте генерации кода Eclipse Modeling Framework [58]. В качестве базы в настоящем диссертационном исследовании используются работы М. С. Малёванного и С. С. Михалковича [59, 60], непосредственно посвящённые проблеме устойчивой привязки.

Предлагаемая в настоящем исследовании концепция упрощённых грамматик развивает подход к осуществлению легковесного парсинга, именуемый «островными грамматиками», описанный в работах Ари ван Дерсена и Леона Мунена [8, 61, 62], а также концепцию специального символа *Any* [63–65] и схожую с ней концепцию «ограниченных морей» [66]. При этом нами используются такие фундаментальные научные результаты, как подход к спецификации формальных языков, разработанный Ноамом Хомским [67, 68], метод нисходящего синтаксического анализа LL(1), описанный Филипом Льюисом и Ричардом Стирнзом [69], метод восходящего синтаксического анализа LR(1), введённый Дональдом Кнутом [70].

Объект и предмет исследования

Объектом исследования является структурированный текст, в частности, код на языке программирования. Предметом исследования являются методы его легковесного синтаксического анализа на основе островных грамматик и методы устойчивой алгоритмической привязки к такому тексту.

Цели и задачи работы

Целью работы является создание метода устойчивой алгоритмической привязки к коду на языке программирования и последующая разработка инструмента разметки кода на основе данного метода. Предполагается, что инструмент разметки кода будет использоваться программистами при разработке и сопровождении программного обеспечения, чтобы сократить время, необходимое для поиска участков кода, важных для текущей решаемой ими проблемы.

Для достижения цели решаются следующие задачи.

1. Разработка теории легковесных грамматик с символом Any и упрощённых грамматик.
2. Разработка и алгоритмическая реализация методов LL(1) и LR(1) синтаксического анализа, позволяющих осуществить разбор программы на основе упрощённой грамматики.
3. Разработка генератора легковесных парсеров и специализированного языка для описания упрощённых грамматик.
4. Разработка упрощённых грамматик нескольких языков программирования и экспериментальная проверка сгенерированных парсеров.
5. Разработка и реализация моделей и методов, позволяющих осуществлять привязку к синтаксическим сущностям программы и их последующий поиск в изменившемся коде.
6. Разработка и алгоритмическая реализация методов, позволяющих осуществлять привязку к произвольным участкам кода программы и их последующий поиск в изменившемся коде.
7. Разработка инструмента с графическим интерфейсом, использующего указанные модели и алгоритмы для предоставления программисту возможности разметить код программы.
8. Проведение экспериментов для оценки устойчивости привязки — успешности поиска участков, к которым ранее была произведена привязка, в изменённой версии кода.

Соответствие паспорту специальности

Цели и задачи диссертационного исследования соответствуют направлениям исследований, предусмотренным паспортом специальности 2.3.5 «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей». Область исследования настоящей диссертации включает

в себя модели, методы и алгоритмы анализа программ (пункт 1 паспорта специальности), языки программирования и семантику программ (пункт 2), модели, методы, архитектуры, алгоритмы, языки и программные инструменты организации взаимодействия программ и программных систем (пункт 3).

Методология и методы исследования

В процессе решения задачи легковесного парсинга автор опирается на методы теории алгоритмов и теории формальных языков и грамматик. При решении задачи устойчивой привязки используются методы теории информации. Предлагаемые модели хранения контекстов формализуются в терминах теории множеств. Для проверки полученных результатов проводятся вычислительные эксперименты. Доказательства завершимости и корректности предложенных алгоритмов проводятся с применением теории формальных языков и теории сложности алгоритмов.

Научная новизна

1. Предложен метод легковесного парсинга на основе упрощённых грамматик, развивающий метод островных грамматик путём задания строгой конструктивно сформулированной связи между «полной» грамматикой G и упрощённой грамматикой G_s. Наличие такой связи позволяет модифицировать для легковесного парсинга на основе упрощённых грамматик методы LL(1) и LR(1) синтаксического анализа. В отличие от других методов легковесного парсинга, для упрощённых грамматик доказано, что легковесный парсер, сгенерированный по грамматике G_s, упрощённой относительно LL(1) грамматики G, распознаёт любую правильную программу из языка $L(G)$. Для случая LR(1) сформулированы существенные условия успешного разбора.
2. Разработан метод устойчивой алгоритмической привязки к синтаксическим элементам кода, отличающийся от других методов «запоминания» места в коде тем, что предназначен для применения к изменяющейся про-

грамме и предлагает новые модели контекстов, описывающие элемент, к которому производится привязка, и новый алгоритм переприв язки, осуществляющий поиск элемента в отредактированной программе на основе ранее построенных для него контекстов. Поиск происходит успешно, даже если в программу внесено большое количество правок. По сравнению с ближайшим аналогом метод производит в 2 раза меньше ошибочных переприв язок и в 3 раза реже требует производить переприв язку вручную.

3. Предложен метод выделения многострочных фрагментов кода и встраивания соответствующих им узлов в АСД, сохраняющий корректность АСД. Он уточняет и формализует подход, применяемый в инструментах, полагающихся на использование псевдокомментариев для обрамления участков кода. Во множестве участков кода задаётся подмножество многострочных фрагментов, выделение которых согласуется с синтаксической структурой программы; метод устойчивой привязки к синтаксическим элементам программы обобщается на такие фрагменты.

Теоретическая значимость

Результаты, полученные при решении задачи легковесного синтаксического анализа, могут быть использованы для дальнейшего развития синтаксического анализа на основе островных грамматик, в частности, для построения более совершенных алгоритмов легковесного синтаксического анализа с доказанной корректностью.

Предложенные модели и алгоритмы привязки и переприв язки могут быть использованы в качестве базы для построения более специализированных моделей и алгоритмов, в том числе учитывающих информацию об отношениях между элементами функциональностей и информацию о конкретном сценарии редактирования кода.

Практическая значимость

Реализованные алгоритмы легковесного парсинга с символом Any, генера-

тор легковесных парсеров LanD, язык описания легковесных грамматик позволяют создавать легковесные грамматики языков, в 20–50 раз более короткие по сравнению с полными грамматиками. Генерируемые парсеры могут использоваться для широкого круга задач, таких как реверс-инженеринг, обработка вставок кода на другом языке, разбор незавершённого кода и кода с ошибками, собственно задача разметки кода.

Реализованный инструмент разметки кода, использующий предложенные модели и алгоритмы привязки и перепривязки, позволяет размечать код в реальных проектах, что способно более чем в 100 раз сократить время, необходимое для последующих переходов к помеченным фрагментам (по сравнению с ручным их поиском). В ходе разработки программы разметка позволяет осуществить долгосрочное документирование кода. При помощи разметки в ходе коллективной работы над проектом могут происходить обмен информацией в команде, работающей над одной задачей, и выдача заданий, связанных с модификацией кода или добавлением кода в существующие места программы. Также разметка кода может быть использована в процессе обучения программированию для выдачи заданий и последующей их проверки.

Реализованный инструмент разметки интегрирован в среды Visual Studio и YACC MC, результаты работы внедрены в четырёх организациях, занимающихся разработкой программного обеспечения (ООО «Кассир-софт», ООО «Чек-онлайн», ФГАНУ НИИ «Спецвузавтоматика», ИП Юрушкин Михаил Викторович), а также используются при разработке компилятора языка PascalABC.NET.

Степень достоверности и апробация результатов работы

Достоверность результатов исследования обеспечивается использованием формальных методов исследуемой области, математической строгостью изложения.

Для модифицированного алгоритма LL(1) синтаксического анализа доказано, что парсер, сгенерированный по упрощённой грамматике и использующий данный алгоритм, успешно разбирает программу, порождённую «полной»

грамматикой. Для алгоритма встраивания в АСД узлов, соответствующих произвольным фрагментам кода, доказано сохранение корректности АСД.

Практическая применимость предложенных алгоритмов синтаксического анализа и алгоритма перепривязки подтверждена результатами вычислительных экспериментов.

Основные результаты работы докладывались на следующих международных, всероссийских, региональных конференциях и семинарах:

- Всероссийская научная конференция «Современные информационные технологии: тенденции и перспективы развития» — Ростов-на-Дону, **2018**, **2019**, **2021**;
- Международная научная конференция молодых ученых по программной инженерии (Spring/Summer Young Researchers' Colloquium on Software Engineering) — Великий Новгород, **2018**; Саратов, **2019**;
- Семинар кафедры алгебры и дискретной математики Мехмата ЮФУ — Ростов-на-Дону, **2019**;
- Всероссийская научная конференция «Научный сервис в сети Интернет» — Новороссийск, **2019**;
- International Young Scientists Conference on Computational Science — Online, **2021**;
- Семинар о развитии открытой распараллеливающей системы (OPC), Мехмат ЮФУ — Ростов-на-Дону, **2021**;
- Семинар кафедры информатики и вычислительного эксперимента Мехмата ЮФУ — Ростов-на-Дону, **2021**;
- Национальный Суперкомпьютерный Форум — Переславль-Залесский, **2021**.

Публикации

Все результаты диссертации опубликованы в **10** работах [71–80]. Работы [74, 76, 79, 80] содержат основные результаты настоящего исследования. В

совместных работах научному руководителю С. С. Михалковичу принадлежат постановка задач, определение основных направлений исследования и общее руководство.

4 научные статьи [71,74,76,80] изданы в журналах, рекомендованных ВАК Минобрнауки РФ, из них 3 [74, 76, 80] — в журналах, включённых в Russian Science Citation Index на платформе Web of Science. **1** работа [79] опубликована в журнале, индексируемом в Scopus. **5** работ [72, 73, 75, 77, 78] опубликованы в сборниках тезисов, индексируемых в РИНЦ.

На реализованные программные инструменты получены **3** свидетельства о регистрации авторских прав [81–83].

Личный вклад автора

Все выносимые на защиту результаты получены лично автором диссертации.

Основные положения, выносимые на защиту

1. Доказано для случая LL(1) грамматик, что легковесный синтаксический анализатор, сгенерированный по грамматике G_s, упрощённой относительно грамматики G, разбирает все программы, удовлетворяющие грамматике G. Для случаев LL(1) и LR(1) доказано, что легковесная грамматика G' с символом Any является упрощённой для некоторого множества «полных» грамматик G.
2. Разработаны эффективные модели хранения контекстов и алгоритмы устойчивой привязки к меняющемуся коду, позволяющие осуществлять правильную перепривязку в изменённом коде крупных промышленных проектов с вероятностью, близкой к 100%.
3. Приведены формальные условия для привязки к многострочному фрагменту кода. Доказано, что при соблюдении этих условий узел, соответствующий многострочному фрагменту, можно корректно встроить в абстрактное синтаксическое дерево программы.

4. Разработаны программные комплексы для генерации легковесных парсеров по упрощённым грамматикам и для привязки и перепривязки к меняющемуся коду, позволяющие экспериментально подтвердить эффективность моделей хранения контекстов и алгоритмов устойчивой перепривязки.

Объём и структура работы

Работа состоит из введения, пяти глав, заключения и списка литературы. Полный объем работы составляет 170 страниц текста с 21 рисунком и 12 таблицами. Список литературы содержит 162 наименования.

В **первой** главе приводится обзор предметной области. Во **второй** главе вводятся и формально определяются для случаев LL(1) и LR(1) понятия легковесной грамматики со специальным терминальным символом Any и упрощённой грамматики. Предлагаются модификации методов LL(1) и LR(1) синтаксического анализа, осуществляющие разбор программы из «полного» языка $L(G)$ на основе упрощённой относительно G грамматики G_s. Для алгоритма LL(1) доказывается утверждение о том, что синтаксический анализатор, сгенерированный по G_s, успешно распознаёт любую программу $\omega \in L(G)$. Для алгоритма LR(1) приводятся необходимые условия успешности такого разбора. Проводится эксперимент по итеративной разработке упрощённой LL(1) грамматики языка C# и проверка сгенерированного парсера. В **третьей** главе вводятся расширения языка описания упрощённых грамматик, для случаев LL(1) и LR(1) описываются механизмы специфического восстановления от ошибок, основанного на обработке Any. Проводится эксперимент, в котором легковесные LL(1) и LR(1) парсеры языков C# и Java применяются к девяти крупным проектам с открытым исходным кодом. В **четвёртой** главе решается задача устойчивой привязки к синтаксическим элементам программы. Описываются структуры специального вида, называемые контекстами, в которых сохраняется информация об элементе, к которому осуществляется привязка. Формулиру-

ется алгоритм перепривязки, анализируется его сложность. В эксперименте для трёх проектов с открытым исходным кодом на языке C# происходит привязка ко всем крупным синтаксическим элементам. Затем для изменённых версий проектов производится перепривязка. Аналогичный эксперимент проводится для грамматки в формате генератора парсеров Yacc. Показывается, что корректно проводимые перепривязки составляют 97,76% от общего числа отредактированных синтаксических элементов для Yacc и 99,8% для C#. В **пятой** главе модели и алгоритмы из предыдущей главы дополняются для обеспечения устойчивой привязки к произвольному однострочному или многострочному фрагменту кода. Для привязки к многострочному фрагменту вводятся формальные определения и алгоритм, позволяющие встроить узел, соответствующий многострочному фрагменту, в абстрактное синтаксическое дерево программы. Доказывается корректность дерева после встраивания. В эксперименте исследуется успешность привязки к однострочным фрагментам. Делается вывод о том, что для непустых и непробельных строк успешность перепривязки составляет от 98,1 до 99,5%. В **заключении** кратко излагаются основные результаты настоящего исследования, описываются возможные направления дальнейшей научной работы.

Глава 1

Обзор предметной области

1.1. Понятие функциональностей программы

Одним из основополагающих принципов программирования является принцип разделения ответственностей (separation of concerns) [5, 6]: за решение разных проблем должны отвечать разные части программы. Словом «concern» обозначаются как проблема, решаемая одним или несколькими фрагментами программного кода, так и сами фрагменты, решающие одну проблему. В настоящей работе в качестве русскоязычного эквивалента слова «concern» используется слово *функциональность*.

Классические парадигмы программирования (процедурно-ориентированное программирование, объектно-ориентированное программирование) предоставляют ограниченные средства для разделения ответственностей: на этапе проектирования программы фиксируется её *приоритетная декомпозиция* — разбиение на относительно самостоятельные и осмысленные единицы модульности, как составные, так и атомарные (например, разбиение на файлы, классы, методы классов в случае ООП), каждая из которых решает свою задачу. В [6] подчёркивается, что приоритетная декомпозиция единственна, в то время как пространство функциональностей программы многомерно. Функциональности могут накладываться друг на друга, один и тот же участок кода может решать несколько проблем (как проблем предметной области, так и проблем технического характера), и декомпозицию можно провести по-разному в зависимости от того, какой набор функциональностей считается более важным. К примеру, компилятор языка программирования можно рассмотреть и как набор этапов компиляции, и как набор поддерживаемых им языковых конструкций [71].

При выполнении некоторой задачи программисту нужно выявить участки

кода, имеющие к ней отношение. Идеальной является ситуация, когда все эти участки сосредоточены в одном элементе приоритетной декомпозиции. В этом случае интенсивно исследовать кодовую базу не требуется. В случае, если функциональность реализуется фрагментами кода, рассредоточенными по программе, она называется *сквозной* или *прорезающей* (crosscutting) [7]. Для работы с такой функциональностью программисту приходится исследовать код и искать все её составляющие. Главное затруднение состоит в том, что назначение фрагментов кода и сам факт того, что они решают одну проблему, – информация, относящаяся к высокоуровневой семантике кода, существующая вне кодовой базы. Она ограниченное время хранится только в памяти человека, который создавал или модифицировал соответствующую функциональность, после чего даже этому человеку приходится заново исследовать код.

В настоящем диссертационном исследовании предлагаются модели и алгоритмы, на базе которых реализуется инструмент разметки, позволяющий сохранить совместно с кодовой базой информацию обо всех элементах программы, реализующих одну функциональность. Это избавляет программиста от длительного исследования кода, упрощает навигацию по проекту и тем самым качественно улучшает процесс разработки и сопровождения программы.

1.2. Прорезающие функциональности

1.2.1. Формализация понятия прорезающей функциональности

Попытка формализовать понятие прорезающей функциональности предпринимается в исследованиях [84, 85]. В рассмотрение вводятся два множества: S — множество различных функциональностей, которые могут быть выделены в программе, и T — множество, содержащее элементы программы для некоторой степени детализации (например, в качестве элементов объектно-ориентированной программы можно рассмотреть классы и члены класса). Определяются отображение $f : S \to 2^T$, связывающее каждую функциональность со мно-

жеством реализующих её элементов программы, и отображение $g : T \to 2^S$, связывающее элемент со всеми функциональностями, в реализации которых он участвует.

Прорезающая функциональность определяется как одновременно *распределённая* (scattered) и содержащая *спутанные* элементы (tangled). Функциональность $s \in S$ называется распределённой, если реализуется более чем одним элементом. Элемент $t \in T$ называется спутанным, если участвует в реализации нескольких функциональностей. Прорезание как сочетание распределённости и спутанности для пары функциональностей $s_1, s_2 \in S$ таких, что $s_1 \neq s_2$, имеет место тогда и только тогда, когда выполняется условие:

$$|f(s_1)| > 1 \wedge \exists t \in f(s_1) : s_2 \in g(t)\,.$$

При этом говорится не о прорезании функциональностью всей программы, а о прорезании функциональностью s_1 функциональности s_2. Данная трактовка не противоречит ранее данному неформальному определению сквозной функциональности как прорезающей приоритетную декомпозицию, поскольку прорезание приоритетной декомпозиции означает прорезание заключённых в ней функциональностей.

1.2.2. Модуляризация прорезающих функциональностей

Основополагающими работами отечественных учёных, посвящёнными вопросу прорезающих функциональностей, являются труды А. Л. Фуксмана [11] и М. М. Горбунова-Посадова [12–14]. В работе А. Л. Фуксмана описывается процесс послойной разработки программы: предполагается, что программа разделяется на «основу» — некоторую базовую версию — и «расширяющие функции», реализация которых рассредоточена по основе. Работы М. М. Горбунова-Посадова описывают программу как сочетание «горизонтальных слоёв» — функциональностей, инкапсулированных в элементах приоритетной декомпозиции, — и «вертикальных слоёв» — прорезающих функциональностей. Оба автора счи-

тают, что прорезающие функциональности должны описываться в модуляризованном (сосредоточенном) виде, а после встраиваться в нужные места основы при генерации интегрированной программы. Привязываться к этим местам предполагается по номеру строки и столбца, а также с помощью поиска подстрок, что абсолютно неустойчиво к редактированию основы.

Идея отделения сквозных функциональностей от «основной» программы присутствует и в зарубежной литературе, где понятие сквозной функциональности часто упоминается в паре с понятием «аспект» — основным термином парадигмы аспектно-ориентированного программирования (Aspect-Oriented Programming, AOP) [7, 16]. Аспект — это модуль или класс, инкапсулирующий в себе сосредоточенное описание сквозной функциональности. Предполагается, что данное описание распределяется по программе на этапе «переплетения», которое может происходить как во время компиляции, так и во время выполнения. Места, в которые необходимо «вплести» части аспекта, идентифицируются преимущественно через сигнатуры классов и членов класса, что также неустойчиво к редактированию.

В работе [86] отмечается, что AOP можно считать средством для работы с прорезающими функциональностями, однако не каждая прорезающая функциональность — это аспект. Ключевой идеей AOP является идея разделения кода аспекта, инкапсулирующего рассредоточенные действия, и кода основной программы. Понятие прорезающей функциональности не содержит предположений о возможности её вынесения в отдельную единицу модульности. Таким образом, оно охватывает как типовые случаи применения аспектно-ориентированного программирования (логирование, проверка прав доступа и т.п.), так и более сложные ситуации, в которых фрагменты, составляющие сквозную функциональность, сильно связаны с остальным кодом и не могут быть исключены из него. В исследовании [87] проводится эксперимент по извлечению сквозных функциональностей в аспекты и отмечается, что более 80% участников эксперимента не смогли отделить относящийся к функциональности код от остальной

программы. Одной из причин является то, что многие участки кода связаны сразу с несколькими функциональностями.

Существуют другие парадигмы, схожие с AOP. В рамках функциональностно-ориентированного программирования [17, 18] (Feature-Oriented Programming, FOP) программа представляет собой композицию базовой функциональности и модулей, отвечающих за дополнительные функциональности. Целью FOP является наращивание базовой функциональности программы и получение линейки программных продуктов. Модуль-функциональность может добавлять или менять программные артефакты: документацию, классы, поля, методы из базовой реализации. В дельта-ориентированном программировании [19, 20] (Delta-Oriented Programming, DOP) оперируют понятиями «основной модуль» для обозначения завершённого программного продукта и «дельта-модуль» для обозначения набора изменений, которые необходимо внести в основной модуль для получения другой версии программы.

Помимо невозможности всегда обособить прорезающую функциональность, важным недостатком AOP, FOP и DOP является то, что организация взаимодействия прорезающих функциональностей между собой — проблема, требующая дополнительных усилий для своего решения. В классическом AOP существует проблема неоднозначности результата при применении разных аспектов в одной точке [88]. В случае FOP взаимодействующие между собой функциональности, реализованные в разных модулях, требуют заведения третьего модуля, комбинирующего первые два и содержащего дополнительный код для взаимодействия. Для больших проектов со значительным количеством связанных друг с другом функциональностей описание всех попарных взаимодействий является трудоёмким. В случае с DOP при описании дельты указывается, какие дельты обязательно должны ей предшествовать; явное указание зависимостей, как и в FOP, повышает сложность разработки больших проектов.

Инструмент разметки, предлагаемый в настоящем диссертационном исследовании, позволяет эффективно работать с прорезающими функциональностя-

ми программы без вынесения их кода в отдельные сущности.

Дополнительно отметим, что техники инкапсуляции прорезающих функциональностей разрабатываются не только для программ, написанных на языках общего назначения, но и для программ, написанных на языках предметной области (Domain-Specific Language, DSL). В рамках исследования [89] разработаны аспектные расширения языков LISA и ANTLR, предназначенных для описания грамматик. В LISA аспектная составляющая введена для облегчения добавления новых правил в грамматику. В случае с ANTLR основной целью аспектного расширения является упрощение создания вспомогательных инструментов — отладчика, редактора и т. п. — реализация которых требует большого количества однообразных действий для каждого правила грамматики.

Модели и алгоритмы устойчивой привязки к коду, предлагаемые в настоящей работе, являются языконезависимыми, с их помощью можно размечать код, написанный как на языках программирования общего назначения, так и на DSL.

1.2.3. Обнаружение прорезающих функциональностей

На практике полностью избавиться от прорезающих функциональностей, вплетённых в код проекта, невозможно: многие из них слишком сложно (как идейно, так и физически) отделить от «основной» логики программы. В литературе предлагаются различные вспомогательные техники для поиска прорезающих функциональностей в интегрированной программе.

В [90] описывается стратегия «информационной прозрачности» программы: программист должен позаботиться о том, чтобы все участки кода, решающие одну задачу, имели общую «сигнатуру» — набор отличительных признаков, по которым их можно легко обнаружить. В качестве признака, поиск по которому проще всего автоматизировать, предлагаются общие правила именования. В [91] утверждается, что программисты обеспечивают информационную прозрачность, используя единообразные комментарии. Авторы работы [92] пред-

лагают помечать код, относящийся к одной функциональности, при помощи аннотаций — стандартного средства языка Java. Приведённые в работе результаты экспериментов демонстрируют, что для аннотированного кода программисты выполняют задачи на 34% быстрее. В работах [93, 94] исследователи проводят подробный анализ текста программы, предполагая, что в комментариях, идентификаторах и т. п. проявляется смысловое содержание кода, а значит, код похож на текст на естественном языке. По коду строится множество всех используемых слов (за исключением ключевых слов языка программирования и слов общего значения из естественного языка), а затем проводится кластеризация по частоте встречаемости той или иной лексики. Кластеры примерно соответствуют представленным в коде функциональностям и могут быть вручную исследованы разработчиком.

В [95, 96] прорезающие функциональности ищутся динамически: рассматриваются результаты трассировки программы для различных вариантов использования. Элементы программы, задействуемые в рамках одного и того же варианта, помечаются как с высокой вероятностью составляющие одну функциональность программы и затем анализируются вручную. Авторы исследования [97] комбинируют текстовый, динамический и исторический поиск. Текстовый поиск позволяет отфильтровать методы, соответствующие некоторому запросу, динамический поиск ограничивает рассматриваемое множество методов методами, присутствующими в дереве выполнения программы, история коммитов используется для выявления методов, часто изменяемых совместно друг с другом. В работах [98, 99] программу предлагается проанализировать статически и построить её модель. Модель состоит из множества программных элементов (классов и членов класса) и отношений, заданных на этом множестве («объявляет», «имеет возвращаемый тип», «вызывает» и т. д.). Предполагается, что для некоторых программных элементов разработчиком уже принято решение об отнесении их к искомой функциональности. В этом случае для остальных элементов можно проанализировать то, насколько они связаны различными от-

ношениями с уже найденными элементами функциональности. Вычисляются две вспомогательные характеристики — «специфичность» и «подкрепление». На их основе вычисляется метрика «степень интереса» (degree of interest). Её близкое к единице значение сигнализируют программисту, что соответствующий элемент, скорее всего, принадлежит функциональности.

В исследованиях [100, 101] для элемента программы также вычисляется степень интереса, однако делается это на основе истории действий программиста. Степень интереса возрастает, если элемент редактируется или к нему осуществляется переход, и уменьшается с течением времени. Как результат, она отражает важность элемента для текущей решаемой задачи. В более поздней работе [102] подтверждается справедливость такого подхода: отмечается, что при работе над знакомым проектом 80% навигаций программист осуществляет в ранее посещённые места в коде. По имеющейся истории перемещений можно предсказать следующий программный элемент, который заинтересует разработчика: с наибольшей вероятностью это либо один из недавно посещённых элементов, либо один из наиболее часто посещаемых.

Можно заметить, что все упомянутые подходы неспособны полностью исключить участие человека. Окончательное решение о принадлежности элемента программы к определённой задаче принимается программистом не только на основе информации, непосредственно присутствующей в программе, но и на основе только ему известной высокоуровневой семантики, связанной с предметной областью и конкретной решаемой проблемой. В настоящей работе попытка автоматизировать выявление прорезающих функциональностей не предпринимается, однако устойчивая привязка к коду позволяет сохранить результаты однократного ручного поиска таких функциональностей в виде разметки. Разметка фиксирует высокоуровневую семантику в явном виде и делает её доступной для всех программистов, работающих над проектом.

1.3. Процесс разработки программы

В большом количестве научных исследований анализируется деятельность программистов — то, как они взаимодействует с кодом, полным прорезающих функциональностей, и сохраняют и передают знание о предназначении его участков. Полученные в этих исследованиях результаты обозначают ряд проблем, подтверждающих актуальность настоящей диссертационной работы.

1.3.1. Исследование кода и сохранение собранной информации

Авторы работ [1, 2, 22, 103] отмечают, что внесение правок в код требует предварительного исследования программы, и это исследование может быть достаточно масштабным как по времени, так и по объёмам изучаемого кода. В [1, 2] утверждается, что на изучение и понимание программы уходит около 70% рабочего времени программиста. Ещё 14% времени тратится на настройку IDE и адаптацию её «под себя», а 4% занимает навигация по коду. Под временем навигации понимается время, необходимое на совершение чисто технических действий (открытие файла, клик и переход и т.п.). Непосредственно редактирование кода занимает около 5% времени. Отмечается, что ключевой и самый времязатратный процесс в ходе разработки — выстраивание «мысленной модели» кода.

В [3] анализируется, как и какими стандартными инструментами пользуются программисты при исследовании кода. Разработчики вынуждены комбинировать различные средства поиска и навигации: сначала ищется некоторая начальная точка в коде, затем исследуются структурные связи (иерархия вызовов, иерархия типов), в которых она участвует. Правильность нахождения участков кода, имеющих отношение к задаче, проверяется посредством отладки. Авторы приходят к выводу, что усилия научного сообщества, направленные на создание и совершенствование специализированных инструментов для изучения кода, абсолютно обоснованы. Авторами работы [4] разработан плагин для

отслеживания поведения разработчика и подтверждены наблюдения предыдущего исследования: изучение программы начинается с некоторой «точки фокусировки», от которой программист переходит по структурным связям к другим местам в проекте.

В [21,26,27,103] поднимается вопрос о том, что важная для текущей задачи информация, выявленная, в том числе, в результате длительного исследования кода, может потеряться, если разработчик вынужден переключиться на другую проблему. Разработчику нужно время, чтобы сохранить информацию, важную для приостанавливаемой задачи, а при возобновлении работы он должен восстановить эту информацию в своей памяти. В [26] отмечается, что переключение с сохранением информации происходит даже в рамках работы над одной и той же крупной задачей. Согласно наблюдениям, программисты разбивают такую задачу на подзадачи и переключатся между подзадачами в соответствии с определёнными паттернами — решают подзадачи последовательно, параллельно, рекурсивно, одновременно рассматривают несколько вариантов решения одной подзадачи, периодически «заземляются», возвращаясь к основной задаче.

В [21] подробно анализируются стратегии приостановки и возобновления работы. Чтобы вернуться к ранее решаемой проблеме, разработчику требуется восстановить *рабочее состояние*: постановку задачи, набор связанных с ней артефактов, план действий. В краткосрочной перспективе и для высокоуровневой информации программист может положиться на собственную память, однако более надёжными и долговечными методами являются сериализация «рабочего состояния» вне кода (заметка в блокноте, записка на мониторе и т. п.) и оставление подсказок в самом окне среды разработки (выделить участок текста, поставить точки останова, оставить комментарий, намеренно внести в код ошибку). Ни один из предложенных способов не является абсолютно удобным и эффективным: согласно статистике, только в 10% случаев на обратное переключение между задачами программист тратит меньше минуты. В 30% случаев на то, чтобы собраться с мыслями перед возобновлением работы, уходит более

30 минут. Опрошенные разработчики отмечают, что для более эффективного сохранения рабочего состояния мог бы быть полезен инструмент, позволяющий пометить сущности программы, с которыми недавно велась работа.

В [27] подтверждаются данные предыдущего исследования. Авторы анализируют активность разработчиков в IDE и за её пределами и приводят визуализацию рабочего процесса участников эксперимента, демонстрирующую, что даже пятнадцатиминутный перерыв в работе над задачей на некоторое время снижает интенсивность редактирований кода после возврата к ней.

Предлагаемый нами инструмент разметки кода позволяет исследовать код однократно, помечая при этом нужные фрагменты. В дальнейшем при необходимости модифицировать уже размеченную функциональность инструмент найдёт помеченные участки автоматически с помощью алгоритма перепривязки, и к ним можно будет осуществить быстрый переход. Сохранение рабочего состояния в виде разметки даёт возможность быстро восстановить его после переключения между задачами. Пометки позволяют вернуться к нужным местам в программе, а благодаря возможности связать комментарий с каждой из пометок в разметке можно сохранить дополнительную описательную информацию о задаче и плане её решения.

1.3.2. Источники информации о функциональностях программы

В работах [23–25, 104, 105] учёные отмечают, что успешность разработки во многом связана с наличием в команде человека, понимающего, для чего был написан тот или иной код и почему он был написан именно так.

В [104] 76% опрошенных разработчиков отмечают важность социального фактора при исправлении багов: им часто приходится прибегать к помощи автора кода. В [105] опрошенные программисты назвали живое общение более предпочтительным источником информации в процессе понимания программы, чем документацию, которая может быть устаревшей или отсутствовать вовсе. 85% опрошенных часто испытывали трудности с пониманием того, что имен-

но делает программа или её часть, а 74% — с пониманием того, каковы были намерения автора кода и почему код написан именно так. Более половины опрошенных часто сталкивались с нехваткой времени на написание документации. В [23] отмечается, что в условиях растущей конкуренции компании вынуждены ускорять разработку и внедрение новых функциональностей программного продукта. Документирование кода при этом становится низкоприоритетной задачей, поэтому даже документация, имеющаяся в наличии, может быть неактуальной. Отсутствие актуальной документации вынуждает программиста исследовать код, однако в самом коде отсутствует информация о том, почему были приняты те или иные решения и какой инвариант должен соблюдаться в том или ином месте. Как следствие, приходится искать автора кода, что также помогает не всегда: автор может уже не работать в компании или не помнить всех подробностей по прошествии времени.

Риски, возникающие при увольнении разработчиков из проекта, анализируются в [24, 25]. Отмечается, что полезность исходного кода определяется наличием людей, знающих, как его использовать и улучшать. Неявное знание об устройстве большого проекта оказывается под угрозой при уходе разработчиков: вместе с каждым уволившимся программистом теряются знания, связанные с принятыми им решениями, и работа с написанным им кодом становится затруднительной для коллег. Авторы исследований анализируют несколько крупных проектов, оценивая объём «заброшенной» части кодовой базы — количество давно не редактируемых файлов, последние правки в которые внесли ушедшие из проекта программисты. Согласно собранной статистике, для некоторых проектов заброшенной становилась почти четверть кодовой базы, и не менее 25% заброшенных файлов оставались таковыми на протяжении двух и более лет.

В отсутствие авторов кода и актуальной документации приходится искать дополнительные источники информации о проекте и косвенные признаки, по которым можно судить о принадлежности участков кода к одной функцио-

нальности. В [104] анализируются способы коммуникации при выборе способа решения задачи: чаще всего обсуждение ведётся через электронную почту (44%) или на митапах (45%). В [106] сообщения корпоративной электронной почты рассматриваются в качестве важного источника знаний о высокоуровневых проектировочных решениях и низкоуровневых особенностях реализации. Авторами работы [107] обнаружено, что показателем принадлежности методов к одной функциональности может быть их близкое расположение в одном файле. В [108] анализируется возможность выявления участков, реализующих одну функциональность, на основе истории коммитов. Исследование [109] посвящено поиску упоминаний конкретных сущностей кодовой базы в сообщениях, оставляемых разработчиками на форумах. В [110] отмечается, что высокоуровневая семантика программы может быть частично выражена явным образом, если программа написана на специально разработанном DSL. Разработчики гораздо легче понимают смысл программы, написанной на таком языке, чем на языке общего назначения. В [111] предлагается концепция фреймворка для написания юнит-тестов, позволяющего сгенерировать по тесту информацию о проверяемом им коде.

Поскольку функциональности программы формулируются ещё на этапе её проектирования, в литературе отдельно исследуется вопрос об использовании специальных средств моделирования программ. На этапе разработки модели могут содержать информацию, недостающую для определения того, имеет ли некоторый код отношение к решаемой задаче. Однако в работе [112] отмечается, что ручка и бумага по-прежнему являются вторым по популярности средством моделирования после специализированных компьютерных программ, поскольку удобны для быстрой коммуникации и обсуждения проблемы. Созданные с их помощью артефакты имеют короткое время жизни, несопоставимое со временем жизни проекта. Авторы исследования [113] анализируют тенденции в области проектирования ПО с 2007 по 2017 год и приводят статистику, показывающую, что специализированные средства моделирования стали использоваться более

интенсивно, однако снизилась удовлетворённость программистов этими средствами. Среди названных причин — затруднения с использованием модели в процессе коммуникации с другими разработчиками и проблемы с синхронизацией модели и кода. 60% опрошенных программистов по-прежнему считают текстовое описание вариантов использования или картинку, нарисованную от руки, приемлемыми способами моделирования.

Разметку кода, определяемую в настоящей диссертации, можно трактовать как документацию, всегда имеющуюся в наличии. Процесс создания разметки, в отличие от процесса создания обычной документации, не отделён от написания кода: помечать участки функциональностей и связывать с ними комментарии об их предназначении и особенностях реализации можно по мере написания этих участков. Кроме того, элементы такой разметки-документации всегда остаются связанными с нужными участками кода за счёт алгоритма перепривязки.

1.3.3. Инструментальная поддержка исследования кода

Чтобы облегчить исследование кода и навигацию по нему, научным сообществом предлагаются различные расширения для IDE и анализируется практика использования таких расширений.

Так, плагин ConcernMapper [114] для среды разработки Eclipse реализует модель, предложенную в работе [98], для языка Java и позволяет программисту объединять элементы программы в функциональности. В работе [115] предлагается плагин Pollicino, реализующий «коллективные закладки»: с местом в коде разработчик может связать дополнительную информацию, этой информацией можно поделиться с коллегами и использовать как одну из форм документации. Разработчики Pollicino предлагают языконезависимый и IDE-независимый подход, основанный на запоминании имени проекта, пути к файлу и номера строки, однако при этом не рассматриваются вопросы синхронизации закладок между разными разработчиками и, в отличие от настоящего диссертационного иссле-

дования, не поднимается вопрос поведения закладок при редактировании кода. В исследованиях [116–118] описывается инструмент CodeBubbles — полное переосмысление интерфейса IDE, задача которого — абстрагироваться от концепции файлов и визуализировать «рабочее множество», относящееся к конкретной задаче, в виде набора «пузырей». Каждый «пузырь» может содержать редактируемый фрагмент исходного кода, документации, заметку программиста о работе над задачей, ошибки и предупреждения компилятора и т.д. В [103] отмечается, что не все типы задач требуют длительного исследования кода, и интерфейс IDE можно адаптировать к конкретному типу задачи, предлагая или скрывая дополнительные инструменты для поиска кода и навигации по нему.

Исследование [119] систематизирует описанные в литературе инструменты, позволяющие связать с кодом некоторую метаинформацию. Отмечается, что такая информация может храниться как в самом коде, что приводит к его загромождению, так и быть внешней по отношению к коду и ссылаться на него. При этом важно поддерживать актуальность ссылок, поскольку код может изменяться. Подчёркивается, что последнему вопросу в литературе почти не уделяется внимание. В [120] проводится исследование расширений и улучшений IDE, предложенных научным сообществом до 2017 года. Приводятся данные, свидетельствующие о том, что с 2013 по 2017 год наблюдается рост ежегодного количества публикаций по данной тематике.

Авторы работы [121] пытаются ответить на принципиальный вопрос о том, как учёный может создать полезный для программиста инструмент. Полезный инструмент должен решать важную проблему, при этом важность характеризуется совокупностью таких показателей как частота возникновения проблемы, продолжительность и влияние на качество программы. Важность проблемы должна оправдывать издержки, связанные с внедрением инструмента, а сам инструмент должен устранять причину проблемы, а не её симптомы. В [122] отмечается, что предоставляемые средой средства не должны «прерывать поток» — нарушать концентрацию разработчика и выстроенный у него в голове

контекст задачи, изолировать разработчика от кода, открываясь в модальном режиме. Для средств рефакторинга кода, интегрированных в IDE, подсчитывается, что 90% разработчиков не используют их из-за несоблюдения указанного условия.

Проблема сохранения информации о прорезающих функциональностях, которую помогают решить разрабатываемые в диссертации модели и алгоритмы, является важной: согласно работам, рассмотренным в разделе 1.3.1, программисту приходится часто исследовать код и искать связанные с функциональностью участки кода, что требует значительных временных затрат. В контексте работы [122] особую значимость приобретает то, что от 89 до 92% перепривязок по результатам проведённых нами экспериментов происходят автоматически (не требуют ручного подтверждения результата программистом).

1.4. Поиск кода в изменённой программе

Задача устойчивой алгоритмической привязки к коду родственна таким хорошо изученным проблемам, как поиск клонов кода [28–40], поиск плагиата [41–47], восстановление сценариев редактирования кода [48–53], рекомендация кода [123–125]. Как и устойчивая привязка, они относятся к области анализа эволюции программного обеспечения. Общей подпроблемой всех перечисленных задач является оценка похожести фрагментов программ [126], однако каждая из задач имеет свою специфику и дополнительные предположения, накладываемые на сравниваемые кодовые базы.

Клоны кода возникают, когда один и тот же фрагмент кода оказывается нужным в разных местах большого проекта и потому много раз копируется, возможно, подвергаясь некоторым редактированиям. В работах [32, 34, 35, 40] проводятся обзор и сравнение различных подходов к выявлению клонов. Подходы разделяются на текстовые [28], лексические [31, 36], синтаксические [29, 33] и семантические [30]. Текстовые подходы предполагают сравнение строк, коротких

«отпечатков», конструируемых для подстрок программы, либо наборов слов, используемых в комментариях и идентификаторах. Лексический поиск осуществляется на основе сравнения последовательностей токенов, возможно, подвергнутых некоторым преобразованиям. При синтаксическом поиске происходит сравнение поддеревьев АСД программы либо сравнение метрик, посчитанных для участков кода по АСД. Семантический поиск осуществляется с использованием графа потока данных и графа потока управления. В работах [37–39] в сочетании с одним или несколькими подходами используется машинное обучение.

Предложенные в научной литературе методы обнаружения плагиата сравниваются в работах [46, 47]. Данные методы можно разделить на те же группы, что и в случае клонов кода. Наиболее известные алгоритмы и инструменты поиска плагиата описаны в исследованиях [42–44].

В [45–47] подчёркиваются существенные различия между поиском клонов кода и поиском плагиата. При поиске клонов необходимо искать короткие почти одинаковые фрагменты кода, находящиеся в различающихся объемлющих фрагментах программы. Общее количество найденных фрагментов может быть сколь угодно большим. При поиске плагиата оценивается сходство двух крупных фрагментов или двух разных программ целиком, высокие значения похожести могут свидетельствовать о наличии кода, неправомерно скопированного из одной программы в другую. Неправомерно скопированный код, как правило, существенно редактируется без изменения смысла, поскольку злоумышленнику нужно сделать его как можно более непохожим на первоисточник и при этом сохранить результаты работы программы. Поиск плагиата должен быть приспособлен к таким редактированиям: они не должны существенно понижать оценку сходства, вычисляемую алгоритмом поиска.

Проблема устойчивой привязки к коду, решаемая в настоящей работе, имеет следующие особенности:

- сравниваются фрагменты кода из разных версий одной программы;

- изменение кода происходит естественным образом в процессе разработки и развития программы;
- для каждого фрагмента, присутствовавшего в исходной версии кода, существует единственное верное соответствие в актуальной версии кода, либо нет ни одного соответствия;
- исходная версия программы может быть недоступна.

Отдельная категория работ посвящена определению разницы между двумя версиями программы и восстановлению сценария редактирования кода [48–53], а также рекомендации дальнейших редактирований на основе ранее определённых сценариев [123, 124]. Как правило, сценарий редактирования — последовательность редактирований, необходимая для превращения одной версии программы в другую, — определяется на основе сравнения АСД двух версий программы. При этом, как отмечается в [51], одной из основных решаемых в данной области проблем является генерация подробных сценариев, соответствующих реально произведённым действиям, что возможно сделать только на основе хорошо детализированных деревьев.

В настоящей работе предполагается, что модели и алгоритмы, разрабатываемые для устойчивой привязки к коду программы, будут использоваться инструментом, интегрируемым в различные IDE и позволяющим размечать код на разных языках. Кроме того, фрагменты кода на разных языках могут относиться к одной и той же прорезающей функциональности, описываемой одной разметкой. Это означает, что разрабатываемые модели и алгоритмы должны быть языконезависимыми. Из требования языконезависимости следует, что при привязке к коду нельзя полагаться на хорошо детализированную структуру программы, фиксирующую особенности конкретного языка. Для устойчивой привязки нами используются легковесные абстрактные синтаксические деревья (деревья, полученные в ходе разбора программ до уровня крупных синтаксических сущностей).

В исследовании [125] описывается инструмент Aroma, предлагающий дополнения для кода, который пишет программист. Инструмент использует имеющийся фрагмент кода как запрос и ищет похожие фрагменты среди проиндексированного открытого кода крупных проектов. На основе окружения найденных фрагментов формируются варианты для дополнения.

В исследованиях [127–130] рассматривается задача, родственная задаче поддержания связи между меняющимся кодом и элементами разметки, — поддержание соответствия между кодом и моделью программы (Round-Trip Engineering). В задаче разметки, помимо текстового представления программы, существуют две модели: абстрактное синтаксическое дерево и собственно разметка кода на функциональности. При изменении текста программы АСД может быть перестроено с нуля. Разметку, в соответствии с формализацией, предлагаемой в [131], можно трактовать как модель, каждый элемент которой содержит публичную и приватную информацию. Публичная часть может быть перестроена заново: она состоит из контекстов, описывающих помеченную сущность, и содержащаяся в них информация присутствует в АСД и в тексте программы. Приватная часть относится к семантике кода и к предметной области — это сами пометки, их группировка в функциональности и связанный с каждой пометкой комментарий. Наличие приватной части не позволяет перегенерировать разметку при изменении текста. Для каждого её элемента необходимо выполнить перепривязку, сравнив связанные с ним контексты, вычисленные по старому дереву, с новыми, вычисленными для представленных в дереве элементов того же типа, и выбрав наилучшее соответствие.

1.5. Легковесный парсинг

Легковесный парсинг или *полупарсинг* [10] — вариант синтаксического анализа, при котором парсер анализирует только часть структуры программы, важную для решения определённой задачи. Легковесные парсеры могут

быть сгенерированы по легковесным грамматикам, в которых подробно описывается структура важных областей программы и почти отсутствуют описания остальных областей. В научной литературе, говоря о легковесном парсинге, также используют слова *устойчивый* или *толерантный* [8, 9], поскольку данный вид парсинга является нечувствительным к любым возможным вариациям кода в неинтересных областях, в том числе к содержащим синтаксические ошибки. Существует и специальный метод восстановления от ошибок, основанный на применении легковесного парсинга, — техника «мостового разбора» (bridge parsing), позволяющая на основе отступов, разделителей, ключевых слов и сопоставления открывающих и закрывающих скобочных конструкций локализовать область программы, в которой произошла ошибка, и внести некоторые исправления [132, 133].

1.5.1. Применение легковесного парсинга

Одной из основных причин применения легковесного парсинга является недоступность полной грамматики языка или её чрезмерная сложность. Легковесная грамматика отражает знание разработчика о том, какова структура участков программы, значимых в контексте решаемой задачи. Соответствие правил грамматики конструкциям языка с самого начала очевидно для разработчика, впоследствии правила могут быть уточнены по результатам тестирования на реальных программах, более того, легковесная грамматика может быть выведена полуавтоматически на основе имеющегося кода [134]. В противоположность этому полная грамматика или иные спецификации, существующие для языка, требуют внимательного изучения и понимания перед началом использования. Как подчёркивается в [135], данный процесс является времязатратным. Также он может оказаться невозможным в связи с закрытостью исходных кодов компилятора или отсутствием описания нужной версии языка [8, 136, 137].

Кроме того, как уже было отмечено, легковесный парсинг является нечувствительным к ошибкам в неважных частях программы, и, в отличие от полного

парсера, может быть использован для разбора неправильного или незавершённого кода. С этой же особенностью связано то, что легковесный парсинг применяется для разбора программ, в тексте которых одновременно используются несколько языков. Парсер, разработанный для распознавания одного из этих языков, должен быть толерантным по отношению к коду на остальных языках. В [138] описывается применение легковесного парсинга для разбора вставок на языке Tom в программах, написанных на языках общего назначения. В [139] рассматривается вопрос одновременного парсинга языков Visual Basic, JavaScript и HTML, используемых при описании веб-документа. В [109,140] легковесный парсинг используется для поиска кода в артефактах, написанных на естественном языке (сообщениях электронной почты и обсуждениях проблем на форумах).

Легковесные парсеры могут создаваться не только для конкретных языков, но и для конкретных проектов, в коде которых есть свои устойчивые особенности (например, повсеместное использование специальных переменных или часто встречающиеся сочетания операторов) [8]. Парсер может быть разработан специально для обнаружения этих особенностей без более глубокого анализа синтаксической структуры программы.

1.5.2. Виды легковесного парсинга

Среди большого количества разновидностей легковесного парсинга, выделяемых в литературе [10], три метода были рассмотрены нами при выборе способа реализации легковесных парсеров для инструмента разметки: нечёткий разбор, разбор на основе островной грамматики и разбор на основе скелетной грамматики.

Нечёткий разбор (fuzzy parsing) [141–143] базируется на понятии «якорей» — токенов, обозначающих начало подлежащих разбору областей. Используемая парсером грамматика фактически состоит из набора более мелких подграмматик. Для каждой из них есть свой стартовый символ, из которого порождаются строки, начинающиеся с некоторого якоря. Основным недостатком

нечёткого парсинга является то, что процесс разбора тесно связан с якорными токенами и подвержен ошибкам в случае, если эти токены могут появиться вне областей программы, которые действительно необходимо разобрать.

Скелетная грамматика (skeleton grammar) [9] частично повторяет полную грамматику языка: правила вывода, описывающие те участки программы, структура которых должна быть разобрана, берутся из полной грамматики и дополняются из неё же правилами, при помощи которых эти участки могут быть выведены из стартового символа (данный процесс носит название «дополнение корнем»). Для нетерминалов, оставшихся без правил вывода, формулируются правила «по умолчанию». Недостатком данного подхода является то, что построение скелетной грамматики невозможно без полной, однако, как уже было отмечено, текст полной грамматики доступен не всегда, а его изучение и понимание требуют значительных усилий и временных затрат. Кроме того, не анализируется корректность проводимого построения.

Островная грамматика (island grammar) [8, 61, 62], согласно неформальному определению, состоит из подробных правил вывода, описывающих представляющие интерес конструкции («острова»), и менее детальных продукций, захватывающих всё остальное («воду»). Формально островные грамматки определяются следующим образом.

Определение 1.5.1. Пусть даны контекстно-свободная грамматика $G = (N, T, P, S)$, где N — множество нетерминальных символов, T — множество терминальных символов, P — множество правил вывода, $S \in N$ — стартовый символ грамматики, и множество «островов» $I \subset T^*$ такое, что $\forall i \in I\ \exists \omega_1, \omega_2 \in T^* \colon \omega_1 i \omega_2 \in L(G)$, где $L(G)$ обозначает язык, генерируемый G. Островная грамматика $G_I = (N', T', P', S')$, сформулированная для языка $L(G)$, обладает следующими свойствами:

1. $L(G) \subset L(G_I)$
2. $\forall i \in I,\ \exists n \in N' \colon n \overset{*}{\Rightarrow} i$ и $\exists \omega_3, \omega_4 \in T^* \colon \omega_3 i \omega_4 \notin L(G) \wedge \omega_3 i \omega_4 \in L(G_I)$

3. $K(G) > K(G_I)$.

Первое свойство означает, что G_I генерирует расширение языка $L(G)$. Второе означает, что парсер языка $L(G_I)$ распознаёт «остров» по меньшей мере в одной программе, которая не принадлежит языку $L(G)$. Третье свойство использует функцию K, вычисляющую сложность грамматики, и означает, что островная грамматика является более простой по сравнению с G.

Островные грамматики широко применяются при решении типичных для легковесного парсинга задач [109,138–140]. На практике приходится учитывать, что их применение обладает двумя значимыми побочными эффектами: возможны ложноположительные и ложноотрицательные результаты [143]. В случае, если «острова» и «вода» похожи друг на друга, важные области программы могут быть распознаны как неважные и наоборот. Более того, разбор «воды» как «острова» может завершиться ошибкой. Для устранения этих эффектов грамматика разрабатывается итеративно, правила вывода уточняются по мере тестирования сгенерированных по грамматике парсеров на реальных программах.

Чтобы минимизировать описания неважных областей и количество итераций уточнения грамматики, для работы с островными грамматиками, как правило, используются обобщённые алгоритмы парсинга GLL [144–146] и GLR [147, 148]. В случае, когда обобщённый алгоритм сталкивается с неоднозначностью, он продолжает разбор программы в соответствии со всеми возможными интерпретациями очередного токена, и оставляет результаты для того пути разбора, который завершается успешно. Такой подход имеет ряд недостатков: обобщённые алгоритмы сложны для трассировки и отладки, могут возвращать несколько деревьев разбора, требующих дополнительной постобработки, а в случае, когда «острова» очень похожи на «воду», перестают быть линейными.

В настоящей работе мы вводим новый подвид островной грамматики — упрощённую грамматику — и модифицируем для работы с упрощёнными грам-

матиками детерминированные алгоритмы нисходящего синтаксического анализа LL(1) [69] и восходящего синтаксического анализа LR(1) [70]. В отличие от всех упомянутых методов легковесного парсинга, для парсинга на основе упрощённых LL(1) грамматик нами доказано, что легковесный парсер, сгенерированный по упрощённой грамматике, успешно разбирает все правильные программы на целевом языке. Для упрощённых LR(1) грамматик сформулированы необходимые условия успешного разбора.

Глава 2

Легковесный парсинг с символом «Any»

В настоящей главе вводятся формальные определения легковесной LL(1) грамматики с символом *Any*, легковесной LR(1) грамматики с символом *Any*, упрощённой LL(1) грамматики и упрощённой LR(1) грамматики, расширяющие теорию островных грамматик. *Any* является специальным терминальным символом, служащим для обозначения «водных» областей и используемым вместо описания их структуры. Для осуществления легковесного парсинга на основе упрощённых грамматик модифицируются методы LL(1) и LR(1) синтаксического анализа. Для случая LL(1) доказывается, что легковесный парсер успешно разбирает программы на целевом языке. Для случая LR(1) формулируются существенные условия успешного разбора. Проводится эксперимент по итеративной разработке упрощённой LL(1) грамматики языка C#. Сгенерированный по этой грамматике легковесный парсер проверяется на кодовых базах крупных промышленных проектов.

2.1. Существующие реализации легковесного парсинга с символом «Any»

2.1.1. Coco/R

Первым известным нам инструментом, реализующим специальный символ `ANY`, является генератор парсеров Coco/R [63]. Генерируемые им анализаторы работают методом рекурсивного спуска, заявлена поддержка грамматик LL(k).

Согласно описанию [64, с. 14], в грамматике для Coco/R символ `ANY` обозначает любой токен, не являющийся конкурентом этого `ANY` в текущем правиле вывода (термин «конкурент» будем использовать вместо используемого в [64] термина «альтернатива», чтобы избежать путаницы с альтернативами правил

```
A = a b c | {ANY} d.
```
(a)

```
A = B d | b B.  B = a {ANY}.
```
(b)

Рис. 2.1. Грамматики, при обработке которых проявляется ограниченность реализации ANY в Coco/R.

в грамматике). Для `ANY`, входящего во множество FIRST некоторого нетерминала, конкурентами являются другие токены из этого FIRST. Для `ANY`, заключённого в скобки `{ }`, обозначающие ноль и более повторений, или в скобки `[ ]`, обозначающие ноль либо одно вхождение, конкурентами являются элементы множества FIRST, построенного для последующей части правила. При генерации парсера с каждым `ANY` связывается множество токенов, допустимых в его позиции.

Главным недостатком реализации `ANY` в Coco/R является чрезмерно поверхностный анализ при определении допустимых токенов. На рисунке 2.1 приведены примеры грамматик, для которых при более сложном анализе можно расширить множество распознаваемых парсером программ. В нижнем регистре записаны терминальные символы, в верхнем регистре записаны нетерминалы. Нетерминал `A` является стартовым символом.

Грамматика на рисунке 2.1a разрешает повторение `ANY` ноль и более раз. Множество допустимых токенов для `ANY` содержит только токены `b` и `c`. Токен `d` исключён из данного множества, чтобы парсер смог найти конец последовательности `{ANY}` и сопоставить `d` явно. Токен `a` также исключён, что позволяет однозначно трактовать все строки, начинающиеся с `a`, как соответствующие первой ветви правила. Однако исключение `a` ведёт к тому, что строка `bad$` парсером не распознаётся, поскольку для `{ANY}` одно и то же множество определяет токены, допустимые как в качестве первого элемента последовательности, так и в качестве остальных элементов. В то же время первый токен входной строки — токен `b` — позволяет выбрать правильную ветвь правила для нетерминала `A`, и следующий за ним токен `a` уже не вызовет неоднозначности. Таким образом, токены, являющиеся конкурентами `ANY` из-за нахождения в FIRST одного с ним

нетерминала, можно считать допустимыми для второй и последующих позиций в последовательности, соответствующей нескольким повторениям этого `ANY`.

На рисунке 2.1b у `ANY` нет конкурентов, поэтому множество допустимых токенов состоит из всех токенов грамматики. В результате сгенерированный парсер не способен распознать строку `abcd$`. Если добавить к стандартному механизму построения данного множества анализ внешнего контекста нетерминалов, можно заметить, что токен `d` содержится в FOLLOW(B), следовательно, он может появиться после `{ANY}` и должен быть исключён из множества допустимых токенов. Тогда строка `abcd$` будет разобрана успешно.

Заметим, что нетерминальный символ `B` появляется в грамматике на рисунке 2.1b в двух различных контекстах: `B d` и `b B`. Ограничение, выведенное из первого контекста, излишне для второго. При статическом построении множества допустимых токенов оба контекста смешиваются, в результате не распознаётся входная строка `bad$`. Однако принятие решения непосредственно во время парсинга позволяет распознать эту строку успешно. После выбора конкретной ветви правила для нетерминала `A` парсеру доступна более точная информация о том, что может следовать за `{ANY}`: ясно, что при выборе ветви `b B` токен `d` не является конкурентом `{ANY}`, и строка `bad$` может быть разобрана.

В настоящей работе предлагается реализация символа *Any*, исправляющая все указанные выше недостатки за счёт того, что множество допустимых токенов не конструируется вовсе, а все решения, связанные с отнесением того или иного токена к *Any*, принимаются в процессе парсинга.

2.1.2. LightParse

Инструмент для разработки легковесных LALR(1) парсеров LightParse [65] также поддерживает написание грамматик с использованием символа `ANY`. Реализация данного символа обладает теми же недостатками, что и реализация `ANY` в Coco/R. Кроме того, грамматика в формате LightParse не используется непосредственно для генерации парсера. Она транслируется в стандартную

грамматику в формате генератора компиляторов Yacc [149]. В ходе трансляции для каждого вхождения символа `ANY` создаётся нетерминал, одноэлементные альтернативы которого содержат все допустимые в соответствующей позиции токены. Необходимость генерации корректного Yacc-файла в совокупности с отсутствием анализа внешнего контекста накладывает на реализацию `ANY` дополнительные ограничения: символ не может стоять в конце альтернативы за исключением альтернатив в правиле для стартового символа. Однако даже с учётом этих ограничений для LightParse не доказано, что по грамматике на данном языке всегда генерируются корректные грамматики в формате Yacc.

Трансляция грамматики из одного формата в другой ведёт к различию между множеством символов, определённых разработчиком в исходной грамматике, и множеством символов, фигурирующих в сообщениях генератора Yacc в случае возникновения ошибки при генерации парсера, что затрудняет отладку.

Наша реализация символа *Any* не предполагает трансляцию в какой-либо существующий язык и адаптацию грамматики к какому-либо стандартному алгоритму разбора. Вместо этого адаптируются сами алгоритмы: обработка *Any* встраивается в стандартные LL(1) и LR(1) алгоритмы синтаксического анализа. В результате устраняются оговорённые ограничения и недостатки.

2.1.3. Ограниченные моря

В работе [66] предлагается схожая с символом *Any* концепция «ограниченных морей» (bounded seas), разработанная для грамматик, разбирающих выражение (Parsing Expression Grammars, PEG) [150]. Ограниченные моря призваны полностью устранить необходимость описания «воды» в островных грамматиках. Для обозначения «моря» используется нотация вида $\sim$`island`$\sim$, где знаком $\sim$ обозначается «вода» без дополнительных правил, уточняющих её структуру, а `island` — некоторый символ грамматики, соответствующий островной сущности. Ключевым свойством «воды» в «море» является то, что она захватывает всё, кроме того, что может находиться после неё в соответствии с грамматикой.

В «водной» области токены пропускаются до тех пор, пока начиная с некоторого токена не удаётся успешно распознать один из префиксов того, что может идти после этой области.

Проверка окончания «воды» сводится к проверке выполнимости всех возможных выражений, следующих за «водной» областью, и подразумевает перебор с возвратом (бэктрекинг), означающий потенциально экспоненциальную сложность анализа. Применение оптимизаций, схожих с техникой packrat парсинга [151], сводит время к линейному, но увеличивает используемый объём памяти. Авторы отмечают, что, несмотря на сложность применяемого анализа, метод ограниченных морей не гарантирует, что пропуск «воды» завершится в нужном месте и программа будет успешно разобрана. Это также подтверждается приводимыми результатами экспериментов.

Кроме того, инструмент PetitParser, реализующий ограниченные моря, базируется на языке Smalltalk[1] и предназначен для использования внутри замкнутой экосистемы, состоящей из виртуальной машины Pharo и фреймворка Moose. Как следствие, генерируемые парсеры крайне сложно интегрировать в произвольный проект.

В настоящей работе для случая LL(1) формально доказано, что легковесный парсер, сгенерированный по грамматике с символом Any, удовлетворяющей введённому нами определению упрощённой грамматики, успешно разбирает любую корректную программу на целевом языке. Для случая LR(1) сформулированы необходимые условия успешного разбора. Для случаев LL(1) и LR(1) проведён масштабный эксперимент, демонстрирующий успешность разбора реальных программ при помощи легковесных парсеров, сгенерированных по упрощённым грамматикам. Предложенный нами генератор легковесных парсеров реализован на языке C# и может быть использован совместно с проектом на любом языке, для которого реализована поддержка .NET Framework.

[1] PetitParser также портирован на несколько других языков, но эти порты являются экспериментальными, добавление в них нового функционала PetitParser, такого как ограниченные моря, не предполагается.

2.2. Упрощённая LL(1) грамматика

2.2.1. Определение упрощённой грамматики

Введём определения легковесной LL(1) грамматики с символом Any и упрощённой LL(1) грамматики. Здесь и далее будем применять следующую систему обозначений:

- для грамматики $G = (N, T, P, S)$ используются стандартные обозначения: N — множество нетерминальных символов, T — множество терминальных символов, P — множество продукций (правил вывода), $S \in N$ — стартовый символ;
- маленькими греческими буквами обозначаются, если не указано другое, последовательности символов из $(N \cup T)^*$ для той грамматики, в которой происходит порождение;
- через $\mathrm{lhs}(p)$ и $\mathrm{rhs}(p)$ обозначаются левая и правая части продукции p соответственно, запись $x \in \mathrm{rhs}(p)$, где $x \in N \cup T$, означает, что $\mathrm{rhs}(p) = \alpha x \beta$;
- через $\mathrm{SYMBOLS}(\gamma)$ обозначается множество терминальных символов, необходимых для записи всех ω: $\gamma \overset{*}{\Rightarrow} \omega, \omega \in T^*$;
- через Any^+ обозначается последовательность из одного и более символа Any.

Также будем считать, что рассматриваются только грамматики, в которых отсутствуют бесполезные нетерминальные символы — такие нетерминальные символы, из которых не выводится ни одна $\omega \in T^*$.

Определение 2.2.1. Легковесной LL(1) грамматикой с символом Any будем называть LL(1) грамматику $G = (N, T, P, S)$, удовлетворяющую следующим условиям:

1. $Any \in T$.

2. $\exists p \in P : Any \in \text{rhs}(p)$.

3. Если является допустимым вывод $S \overset{*}{\Rightarrow} \alpha Any A \beta$, где $A \in N$, то в P не существуют одновременно такие различные продукции p_1 и p_2, $\text{lhs}(p_1) = \text{lhs}(p_2) = A$, что для некоторого $a \in T \backslash \{Any\}$ допустимы выводы $\text{rhs}(p_1)\beta \overset{*}{\Rightarrow} Any^{+} a \delta_2$ и $\text{rhs}(p_2)\beta \overset{*}{\Rightarrow} a\delta_1$.

   Исключением является случай, когда p_1 имеет вид $A \to AnyA$ и p_2 имеет вид $A \to a\delta$.

Определение 2.2.2. Пусть $G = (N, T, P, S)$ — некоторая LL(1) грамматика, $Any \notin T$. Упрощённой относительно G будем называть легковесную LL(1) грамматику с символом Any $G_s = (N_s, T_s, P_s, S_s)$, удовлетворяющую следующим условиям:

1. $S_s = S$;

2. $T_s = T \cup \{Any\}$;

3. $P_s = \{p \in f(P) \mid \text{lhs}(p) = S_s \vee \exists p' \in P_s \colon \text{lhs}(p) \in \text{rhs}(p')\}$, где отображение $f : P \to \{p = A \to \alpha \mid A \in N, \alpha \in (N \cup T \cup \{Any\})^*\}$ удовлетворяет условиям:

   а. $\exists P' \subseteq P \colon P' = \{p \in P \mid f(p) \neq p\}$.

   б. $\forall p \in P \setminus P',\ f(p) = p$.

   в. $\forall p \in P',\ \exists n \in \mathbb{N} \colon$ p представима в виде $A \to \alpha_1\gamma_1\alpha_2\gamma_2 ... \alpha_{n+1}$ и $f(p)$ представима в виде $A \to \alpha_1 Any \alpha_2 Any ... \alpha_{n+1}$.

   г. Обозначим через Any_i вхождение Any, возникшее как результат замены некоторого γ_i. Порождение $S_s \overset{*}{\Rightarrow} \alpha Any_{i_1} Any_{i_2} ... Any_{i_n} b\beta$, где $i_1, i_2, ..., i_n, n \in \mathbb{N}$, $b \in T_s \setminus \{Any\}$, не является допустимым в G_s, если $b \in \text{SYMBOLS}(\gamma_{i_1}\gamma_{i_2} ... \gamma_{i_n})$.

4. $N_s = \{A \in N \mid \exists p \in P_s \colon \text{lhs}(p) = A\}$.

Говоря о грамматике G_s, упрощённой относительно LL(1) грамматики G, будем сокращённо называть её упрощённой LL(1) грамматикой.

Согласно определению, стартовый символ упрощённой грамматики G_s остаётся тем же, что и в грамматике G. Множество токенов расширяется специальным токеном Any. Во множестве нетерминальных символов остаются только нетерминалы, присутствующие хотя бы в одном выводе из стартового символа грамматики G_s. Часть правил вывода переходит в G_s без изменений, а в некоторых правилах подпоследовательности правых частей заменяются на Any. Предполагается, что заменяемые фрагменты описывают области программы, структура которых неважна в рамках задачи, для решения которой пишется упрощённая грамматика. В терминологии островных грамматик такие области называются «водными».

Отображение f ставит в соответствие каждой продукции исходной грамматики либо саму эту продукцию, либо продукцию, получаемую путём замены некоторого фрагмента правой части на Any. Множество продукций грамматики G_s вложено в образ этого отображения. Отметим, что от отображения f не требуется инъективность, так как несколько правил вывода после замены их частей на Any могут стать одинаковыми.

Определение упрощённой грамматики означает, что в некоторых строках, генерируемых G, существуют подстроки, заменой которых на Any может быть получена строка, генерируемая G_s. Условие 3г означает, что Any не может заменять γ_i из правой части правила вывода, если из этой γ_i выводится последовательность, содержащая токен, который в некотором выводе будет следовать за этим Any. Как будет показано далее, соблюдение данного условия позволяет разбирать программу, порождаемую грамматикой G, при помощи парсера, сгенерированного по G_s.

2.3. Парсинг на основе упрощённой LL(1) грамматики

2.3.1. Алгоритм

Упрощённая LL(1) грамматика G_s является обычной LL(1) грамматикой, в которой Any — обычный терминальный символ. Однако она создаётся для того, чтобы сгенерированный по ней парсер успешно разбирал программы, порождаемые грамматикой G. Относительно грамматики G_s эти программы могут быть неправильными, поскольку в тех местах, где согласно грамматике G_s должен находиться токен Any, в этих программах находятся другие токены, выводимые из заменённых на Any последовательностей. Стандартный метод LL(1) синтаксического анализа для такого разбора не подходит. Его необходимо модифицировать, чтобы по мере разбора парсер трактовал некоторые последовательности токенов как Any, тем самым на лету «переводя» программу из $L(G)$ на язык $L(G_s)$, и проверял синтаксическую корректность относительно G_s.

Алгоритмы 1a–1d — модифицированные алгоритмы табличного LL(1) разбора [152, С. 220–228]. Серым выделены строки, добавленные для работы с упрощёнными грамматиками. В приведённом псевдокоде доступ к стеку символов осуществляется посредством переменной *Stack*, входной буфер доступен через объект лексического анализатора *Lexer*. Метод лексического анализатора NewToken возвращает следующий токен из входного потока. CurrentToken возвращает последний прочитанный токен. Вспомогательная переменная t хранит текущий обрабатываемый токен, вспомогательная переменная X хранит текущий символ на вершине стека. M соответствует таблице разбора, символ `$` обозначает конец входного потока. Построение таблицы разбора осуществляется стандартным для LL(1) способом и не требует дополнительных модификаций. Выполнение условий, нужных для исполнения строки 26 алгоритма 1a или строк 16–19 алгоритма 1b, означает, что переданная на вход легковесному парсеру программа является некорректной относительно грамматики G, и эта некорректность расположена не в «водной» области. В рамках рассуждений

настоящей главы будем считать, что при таком сценарии указанные строки не выполняются и разбор завершается с ошибкой. Назначение данных строк подробно объясняется в главе 3.

Рассмотрим алгоритм синтаксического анализа 1a. В ситуации, когда терминальный символ на вершине стека не совпадает с очередным входным токеном t, а также в случае, когда на вершине стека находится нетерминальный символ X и в ячейке $M[X, t]$ отсутствует запись, стандартный алгоритм сообщает об ошибке, поскольку для текущего токена отсутствуют явно указанные допустимые действия. Для модифицированного алгоритма эта ситуация является штатной, поскольку, как было сказано ранее, анализируемая программа не принадлежит языку, по грамматике которого построен парсер. Если на вершине стека находится символ Any или есть запись в ячейке $M[X, Any]$, модифицированный алгоритм пытается начиная с текущего токена пропустить некоторую последовательность токенов из входного потока, проинтерпретировав её как соответствующую символу Any (выводимую из той последовательности γ в правиле грамматики G, которая была заменена на данное Any). Тем самым осуществляется переход от текста из $L(G)$ к тексту из $L(G_s)$. Пропуск возможен благодаря пункту 3г определения 2.2.2: множество токенов в пропускаемой последовательности не содержит токенов, которые могут идти после Any в соответствии с текущей ситуацией на стеке. По завершении пропуска разбор продолжается стандартным образом.

Чтобы определить токены, обозначающие конец области, соответствующей Any (*стоп-токены*), в первом приближении для текущего состояния стека нужно построить множество FIRST (стек трактуется как последовательность символов, начинающаяся с его вершины). Однако в некоторых случаях стандартного алгоритма FIRST недостаточно. В ходе разбора возможна ситуация, когда в терминальной строке, выводимой из последовательности на стеке, два или более токена Any могут следовать друг за другом. Например, для грамматики

Алгоритм 1 Модифицированные алгоритмы для случая LL(1): (a) Алгоритм парсинга, (b) Алгоритм обработки «Any»; (c) Построение множества FIRST′; (d) Вспомогательные алгоритмы: применение альтернативы и мемоизация множеств FIRST′.

```
1: Parse():
2:   Stack := [];
3:   Stack.Push(new Node($));
4:   Stack.Push(new Node(S));
5:
6:   X := Stack.Peek().Symbol;
7:   t := Lexer.NextToken();
8:   while (X ≠ $) do
9:     if (t = ERROR_TOKEN) then
10:      return false;
11:    end if;
12:    if (X = t) then
13:      if (t = Any) then
14:        t := SkipAny(true);
15:      else
16:        Stack.Pop();
17:        t := Lexer.NextToken();
18:      end if;
19:    elif (M[X,t] ≠ null) then
20:      if (t = Any) then
21:        t := SkipAny(true);
22:      else
23:        Apply(M[X,t]);
24:      end if;
25:    elif (t = Any) then
26:      t := Error(null);
27:    else
28:      t := Any;
29:    end if;
30:    X := Stack.Peek().Symbol;
31:  end while;
32:
33:  if (t = $) then
34:    Accept();
35:    return true;
36:  else
37:    return false;
38:  end if;
```

(a)

```
1: SkipAny(recoveryIsEnabled):
2:   t := Lexer.CurrentToken();
3:   idx := Lexer.CurrentTokenIndex();
4:   while (Stack.Peek().Symbol ∈ N_S) do
5:     Apply(M[Stack.Peek().Symbol, Any]);
6:   end while;
7:
8:   Stack.Pop();
9:
10:  stopTokens := FIRST'(Stack);
11:  while (t ∉ stopTokens and t ≠ $) do
12:    t := Lexer.NextToken();
13:  end while;
14:
15:  if (t ∉ stopTokens) then
16:    if (recoveryIsEnabled) then
17:      Lexer.MoveTo(idx);
18:      return Error(stopTokens);
19:    else
20:      return ERROR_TOKEN;
21:    end if;
22:  end if;
23:
24:  return Lexer.CurrentToken();
```

(b)

```
FIRST'(α = Y_1 Y_2 ... Y_k):
  first := ∅;
  for (i from 1 to k) do
    if (Y_i ∈ T_S) then
      first ∪= {Y_i};
      if (Y_i ≠ Any) then break; end if;
    else
      first ∪= MemoizedFirst'[Y_i]\{ε};
      if (ε ∉ MemoizedFirst'[Y_i]) then break; end if;
    end if;
  end for;
  if (∀i ∈ [1..k]: ε ∈ MemoizedFirst'[Y_i] or Y_i = Any) then
    first ∪= {ε};
  end if;
  return first;
```

(c)

```
Apply( X → Y_1 Y_2 ... Y_k):
  parent := Stack.Pop();
  for (i from k to 1) do
    child := new Node(Y_i);
    Stack.Push(child);
    parent.Children.AddFirst(child);
  end for;
```

```
BuildFirst'():
  foreach (A ∈ N) do
    MemoizedFirst'[A] := ∅
  end foreach;
  changed := true;
  while (changed) do
    changed := false;
    foreach (A → α ∈ P) do
      MemoizedFirst'[A] ∪= FIRST'(α);
      if (MemoizedFirst'[A] изменилось) then
        changed := true;
      end if;
    end foreach;
  end while;
```

(d)

$$A \to Any\, C\, e$$

$$C \to Any \mid d,$$

упрощённой относительно грамматики

$$A \to B\, C\, e$$

$$B \to a \mid b$$

$$C \to b \mid c \mid d,$$

при обработке первого Any на стеке находится последовательность $C\, e$, и множество FIRST(Stack) $= \{d, Any\}$. В результате парсер не распознаёт все принадлежащие $L(G)$ строки, не содержащие токен d.

Алгоритм 1c осуществляет построение множества FIRST$'$, являющегося расширенной версией стандартного FIRST. Разница со стандартным алгоритмом заключается в том, что токен Any обрабатывается и как токен (помещается в результирующее множество), и как потенциально пустая строка (во множество FIRST$'$ попадают токены, которые могут встретиться после Any). Для приведённого примера FIRST$'$(Stack) $= \{d, e, Any\}$, в результате распознаются все строки из $L(G)$. Отметим, что обращение к стеку для получения информации об ожидаемых токенах приближает модифицированный LL(1) алгоритм к алгоритму full-LL(1) разбора [153, С. 248–251].

Алгоритм 1e является модифицированной версией алгоритма, приведённого в [153, С. 239–240]. Он осуществляет нерекурсивное построение множеств FIRST$'$ для всех нетерминалов грамматики. Данные множества запоминаются в словаре *MemoizedFirst′* во избежание повторных вычислений. Доказано, что исходный алгоритм является конечным, то же доказательство верно и для его модификации.

В случае, когда некоторой подпоследовательности токенов α в программе из $L(G)$ соответствует последовательность $Any_1 Any_2 ... Any_n$ в программе из $L(G_s)$ ($n \in \mathbb{N}$, $n > 1$), α целиком считается соответствующей Any_1, а последу-

ющие Any_i, $i \in [2; n]$ считаются соответствующими пустым последовательностям токенов. В упрощённой грамматике невозможно точно установить попарное соответствие между частями последовательности α и последовательными символами Any_i, для этого нужно больше информации об исходной грамматике G. Похожая проблема, называемая проблемой «перекрывающихся морей» (overlapping seas), описана в [66]: в случае, если одно море следует за другим, невозможно провести границу между окончанием первого моря и началом второго, поэтому «вода» второго моря, идущая до «острова», считается пустой.

Приведённая модификация LL(1) разбора отчасти напоминает алгоритм восстановления от ошибок [152, С. 228–231, С. 295–297]: символ Any похож на символ $error$, обозначающий место, в котором можно произвести восстановление; множество FIRST' похоже на множество синхронизационных токенов. Данное сходство закономерно: как уже было сказано, программа, разбираемая парсером языка $L(G_s)$, не принадлежит этому языку и в этом смысле действительно является ошибочной. Заменяя последовательность токенов на Any, парсер находит во входном потоке место, с которого программа снова соответствует G_s. Однако стандартное восстановление от ошибок не всегда оканчивается успешно, в ходе него может быть отброшена часть значимых результатов уже проведённого анализа, а также пропущена значительная часть входного потока, возможно, содержащая «острова». В случае с обработкой Any чётко сформулированные формальные условия, которым удовлетворяет упрощённая грамматика, гарантируют, что разбор завершится успешно и части входного потока, заменённые на Any, не включат в себя то, что описано в G_s в явном виде и является важным для текущей решаемой задачи.

Докажем, что парсер, сгенерированный по грамматике G_s, упрощённой относительно LL(1) грамматики G, использующий алгоритмы 1a–1d, успешно разбирает все правильные программы из $L(G)$. Для этого докажем вспомогательное утверждение.

Утверждение 2.3.1. Если для некоторого $X \in N$ и $t \in T$ в ячейке $M_G[X,t]$ находится продукция p, и $X \in N_s$, то $f(p)$ находится хотя бы в одной из ячеек $M_{G_s}[X,t]$ или $M_{G_s}[X,Any]$.

Доказательство. Продукция p находится в $M_G[X,t]$ тогда и только тогда, когда $t \in \mathrm{FIRST}(\mathrm{rhs}(p))$ или $\varepsilon \in \mathrm{FIRST}(\mathrm{rhs}(p)) \wedge t \in \mathrm{FOLLOW}(X)$. Поскольку в ходе построения упрощённой грамматики подпоследовательности в правой части некоторых продукций заменяются на Any, возможны два варианта:

1. В грамматике G посредством p реализуется вывод $S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} \alpha t \delta \beta$, и в грамматике G_s посредством $f(p)$ реализуется хотя бы один из выводов $S_s \overset{*}{\Rightarrow} \alpha' X \beta' \overset{*}{\Rightarrow} \alpha' t \delta' \beta'$ или $S_s \overset{*}{\Rightarrow} \alpha' X \beta' \overset{*}{\Rightarrow} \alpha' Any \zeta \beta'$. В первом случае $f(p)$ попадает в $M_{G_s}[X,t]$, во втором — в $M_{G_s}[X,Any]$.

2. В грамматике G посредством p реализуется вывод $S \overset{*}{\Rightarrow} \alpha X t \beta \overset{*}{\Rightarrow} \alpha t \beta$, и в грамматике G_s посредством $f(p)$ реализуется хотя бы один из выводов $S_s \overset{*}{\Rightarrow} \alpha' X t \beta' \overset{*}{\Rightarrow} \alpha' t \beta'$ или $S_s \overset{*}{\Rightarrow} \alpha' X Any \zeta \overset{*}{\Rightarrow} \alpha' Any \zeta$ или $S_s \overset{*}{\Rightarrow} \alpha' X \beta' \overset{*}{\Rightarrow} \alpha' Any \zeta \beta'$. В первом случае $f(p)$ попадает в $M_{G_s}[X,t]$, в остальных — в $M_{G_s}[X,Any]$.

□

Утверждение 2.3.2. Синтаксический анализатор, сгенерированный по грамматике G_s, упрощённой относительно LL(1) грамматики G, успешно распознаёт любую программу $\omega \in L(G)$.

Замечание: необходимо помнить, что упрощённая грамматика обязательно содержит символ Any и имеется в виду модифицированный алгоритм распознавания 1а.

Доказательство. По построению G_s очевидно, что программа ω анализируется синтаксическим анализатором, сгенерированным для $L(G_s)$, точно так же, как и анализатором, сгенерированным для $L(G)$, пока реализуется одна из следующих ситуаций:

- для входного токена t на вершине стека *Stack* находится нетерминальный символ X и $M_{G_s}[X, t]$ непуста;

- для входного токена t на вершине стека *Stack* находится токен t.

По определению упрощённой грамматики в P_s существует хотя бы одна продукция p', полученная путём замены одной или нескольких подпоследовательностей γ_i в правой части некоторой продукции $p \in P$ на *Any*. Следовательно, возможны два случая, означающие расхождение со стандартным разбором:

1. Для очередного входного токена t на вершине стека *Stack* находится нетерминальный символ X, и $M_{G_s}[X, t]$ пуста. Тогда, в соответствии с утверждением 2.3.1, в $M_{G_s}[X, Any]$ находится продукция $f(p)$, где p — продукция из $M_G[X, t]$, которую при встрече токена t задействует парсер, построенный по грамматике G. В соответствии с алгоритмом 1, X заменяется на правую часть продукции из $M_{G_s}[X, Any]$, после чего снова реализуется пункт 1 или происходит переход к пункту 2.

2. Для очередного входного токена t на вершине стека *Stack* находится токен *Any*. Очевидно, что данному *Any* в анализируемой программе соответствует некоторая подпоследовательность токенов, выводимая из последовательности $\gamma \in (N \cup T)^*$, заменённой на данное *Any*. В этом случае, как следует из алгоритма 1, *Any* снимается со стека. Обозначим последовательность оставшихся на стеке символов через $\delta \in (N_s \cup T_s)^+$. Далее возможны три варианта:

   а. Если $t \notin \mathrm{FIRST}'(\delta)$, пропускаем токен t, переходим к следующему токену во входном потоке и повторяем рассуждения пункта 2.

   б. Иначе, если $t \in \mathrm{FIRST}(\delta)$, это означает, что все токены программы, соответствующие последовательности, выводимой из γ, пройдены, либо эта последовательность пуста. В соответствии с пунктом 3г

определения упрощённой грамматики, t является первым символом после этой последовательности. Начиная с t, разбор возобновляется в соответствии с базовым LL(1) алгоритмом.

При этом, если после пропуска Any, соответствующего γ в полной грамматике, первым символом на стеке является нетерминал X, то в соответствии с пунктом 3 определения 2.2.1 могут существовать продукции $q_1, q_2 \in P_s$ такие, что $q_1 = X \to Any_k X$ и $q_2 = X \to t\zeta$, где Any_k соответствует некоторому γ_k в полной грамматике. То есть в грамматике G_s может быть допустим вывод $X \overset{*}{\Rightarrow} {Any_k}^+ t\zeta$, в котором на всех шагах, кроме последнего, используется продукция q_1, и на последнем шаге применяется продукция q_2. Допустим, что это так. Тогда продукция q_1 находится в ячейке $M_{G_s}[X, Any]$, продукция q_2 находится в ячейке $M_{G_s}[X, t]$. Нетерминал X будет заменён на стеке на правую часть продукции q_2, в то время как последовательность токенов, предшествовавшая t и пропущенная как Any, возможно, содержит некоторый токен u, на котором при разборе полным парсером заканчивается последовательность, выводимая из γ, и начинается последовательность, выводимая из γ_k. При разборе полным парсером замена терминала X осуществляется на токене u на правую часть продукции $p_1 \in P$ такой, что $p_1 = X \to \gamma_k X$, $t \notin \text{SYMBOLS}(\gamma_k)$, $f(p_1) = q_1$. Однако данное расхождение между полным и легковесным парсингами в выборе продукции и токена, на котором разворачивается нетерминал X, не влияет на обработку оставшейся при легковесном парсинге части входного потока: независимо от того, сколько раз полный парсер разворачивает X в соответствии с p_1, при легковесном парсинге последовательности токенов, соответствующие цепочке Any_k^+, заменяющей собой γ_k^+, уже пропущены во время прошедшей обработки Any. Префикс оставшейся части входного потока начинается с токена t, на котором полный парсер проводит замену

X на $\mathrm{rhs}(p_2)$, а легковесный — на $\mathrm{rhs}(q_2)$, где $q_2 = f(p_2)$.[2]

в. Иначе, если $t \in \mathrm{FIRST}'(\delta)$, это означает, что $Any \in \mathrm{FIRST}(\delta)$ и существует вывод $\delta \overset{*}{\Rightarrow} Any_{i_1} Any_{i_2} ... Any_{i_n} \delta'$, где $i_1, i_2, ... i_n, n \in \mathbb{N}$, $t \in \mathrm{FIRST}(\delta')$. В соответствии с пунктом 3г определения упрощённой грамматики, можно утверждать, что была пройдена последовательность токенов, выводимая из $\gamma_{i_1}\gamma_{i_2}...\gamma_{i_n}$. При этом во входном потоке уже произошло перемещение к началу последовательности, выводимой из δ', но из содержимого стека необходимо вывести $Any_{i_1} Any_{i_2} ... Any_{i_n}$. Отметим, что на стеке могут лежать как символы Any, так и нетерминалы, из которых эти Any выводятся. Здесь возможны следующие ситуации:

   i. Если очередной символ на стеке — токен Any, то повторяются рассуждения пункта 2 настоящего утверждения.

   ii. Если очередной символ на стеке — нетерминал X и ячейка $M_{G_s}[X, t]$ непуста, то X заменяется на правую часть продукции из $M_{G_s}[X, t]$, после чего разбор возобновляется в соответствии с базовым LL(1) алгоритмом. В противном случае X заменяется на правую часть продукции из $M_{G_s}[X, Any]$. Из пункта 3 определения 2.2.1 следует, что, пока не будет порождена и полностью снята со стека последовательность $Any_{i_1} Any_{i_2} ... Any_{i_n}$, ячейки $M_{G_s}[X, t]$ будут пустыми и выбор будет делаться в пользу продукции из $M_{G_s}[X, Any]$ за исключением случая, рассмотренного на шаге 2.б и не влияющего на правильность разбора.

   Выясним, какая продукция находится в $M_{G_s}[X, Any]$. Пусть в G при аналогичном выводе $\gamma_{i_1}\gamma_{i_2}...\gamma_{i_n}$ нетерминал X заменяется на правую часть некоторой продукции p. Это означает, что в по-

[2] За счёт пункта 3 определения 2.2.1 в упрощённой грамматике допускается описание списка, элементом которого среди прочего может быть символ Any. В ходе разбора один символ Any может захватывать область программы, соответствующую нескольким «водным» элементам этого списка в «полной» грамматике, при этом «островные» элементы списка распознаются в своих действительных границах.

следовательности токенов, выводимой из $\gamma_{i_1}\gamma_{i_2}...\gamma_{i_n}$, существует такой токен $u \in T$, что $p \in M_G[X, u]$. В соответствии с утверждением 2.3.1, продукция $f(p)$ попадает хотя бы в одну из ячеек $M_{G_s}[X, u]$, $M_{G_s}[X, Any]$. Предположим, что $f(p) \in M_{G_s}[X, u]$. Тогда $u \in \text{FIRST}'(\delta)$, и с этого токена разбор должен возобновиться в соответствии с базовым LL(1) алгоритмом. Получаем противоречие с тем, что вся последовательность токенов, выводимая из $\gamma_{i_1}\gamma_{i_2}...\gamma_{i_n}$, уже была пропущена, поскольку ни один из токенов этой последовательности не принадлежал $\text{FIRST}'(\delta)$. Следовательно, $f(p) \in M_{G_s}[X, Any]$.

Таким образом, если порождение программы ω в грамматике G осуществляется посредством последовательного применения продукций $p_{i_1}, p_{i_2}, ...p_{i_n}$, то в ходе разбора программы ω синтаксический анализатор, сгенерированный по грамматике G_s, последовательно применяет продукции $f(p_{i_1}), f(p_{i_2}), ...f(p_{i_n})$ за исключением двух случаев:

- в соответствии с пунктами 2.а и 2.в.i доказательства пропускаются f от тех p_{i_j}, которые используются при порождении из последовательностей γ, заменённых в упрощённой грамматике на Any, при этом порождаемые ими последовательности токенов пропускаются при обработке Any;

- в соответствии с пунктами 2.б и 2.в.ii доказательства вместо последовательностей вида $f(p_1), f(p_1), ...f(p_1), f(p_2)$, удовлетворяющих исключению, предусмотренному пунктом 3 определения 2.2.1, используются $f(p_2)$, что не влияет на успешность сопоставления токенов из входного потока.

Следовательно, парсер, сгенерированный по упрощённой грамматике G_s, успешно распознаёт любую программу из $L(G)$. □

В смысле успешного разбора парсером $L(G_s)$ программы из $L(G)$ можно утверждать, что $L(G) \subseteq L(G_s)$. Вложение является нестрогим, поскольку в

позиции Any могут быть успешно пропущены не только корректные с точки зрения грамматики G последовательности, но и любые последовательности, не содержащие токенов, допустимых после Any. Это делает парсер менее чувствительным к возможным ошибкам в «водных» областях программы.

Очевидно, что алгоритм 1a завершим, а его сложность совпадает со сложностью стандартного алгоритма LL(1) разбора и составляет $O(n)$.

2.3.2. Связь между легковесной LL(1) грамматикой с символом «Any» и упрощённой LL(1) грамматикой

Утверждение 2.3.3. Пусть $G' = (N', T', P', S')$ — легковесная LL(1) грамматика с символом Any. G' является грамматикой, упрощённой относительно любой LL(1) грамматики $G = (N, T, P, S)$, удовлетворяющей следующим условиям:

1. $S = S'$.
2. $T = T' \setminus \{Any\}$.
3. $N' \subseteq N$.
4. Пусть $\mathfrak{P} = \{p \in P | \operatorname{lhs}(p) \in N'\}$. $\forall p \in \mathfrak{P}, \exists! p' \in P'$ такая, что $p = p'$ или $p' = A \to \alpha_1 Any \alpha_2 Any ... \alpha_{n+1}$ и $p = A \to \alpha_1 \gamma_1 \alpha_2 \gamma_2 ... \alpha_{n+1}$, где $A \in N'$, $\alpha_i \in (N' \cup T)^*$, $\gamma_i \in (N \cup T)^*$, $n \in \mathbb{N}$.
5. $\forall p' \in P', \exists p \in P$ такая, что $p = p'$ или $p = A \to \alpha_1 \gamma_1 \alpha_2 \gamma_2 ... \alpha_{n+1}$ и $p' = A \to \alpha_1 Any \alpha_2 Any ... \alpha_{n+1}$, где $A \in N'$, $\alpha_i \in (N' \cup T)^*$, $\gamma_i \in (N \cup T)^*$, $n \in \mathbb{N}$
6. Пронумеруем все вхождения Any в правила грамматики G' и все γ, соответствующие им в G. Если порождение $S' \overset{*}{\Rightarrow} \alpha Any_{i_1} Any_{i_2} ... Any_{i_n} b \beta$, где $i_1, i_2, ..., i_n, n \in \mathbb{N}$, $b \in T' \setminus \{Any\}$, является допустимым в G', то $b \notin \text{SYMBOLS}(\gamma_{i_1} \gamma_{i_2} ... \gamma_{i_n})$

Доказательство. Выполнение для G' условий 1 и 2 определения упрощённой LL(1) грамматики следует из пунктов 1 и 2 настоящего утверждения.

Введём отображение $g : \mathfrak{P} \to P'$, действующее в соответствии с пунктом 4 настоящего утверждения. В соответствии с пунктом 5 настоящего утверждения, $g(\mathfrak{P}) = P'$. Отображение $f(P)$ построим таким образом, что $\forall p \in \mathfrak{P}, f(p) = g(p)$, и $\forall p \in P \setminus \mathfrak{P}, f(p) = p$. Условия 3а, 3б и 3в, задаваемые определением упрощённой LL(1) грамматики, выполняются для отображения f по построению и в соответствии с определением легковесной грамматики с символом Any. Условие 3г эквивалентно пункту 6 настоящего утверждения. Таким образом, выполняется условие 3 определения упрощённой LL(1) грамматики.

Выполнение условия 4 определения упрощённой LL(1) грамматики следует из пункта 3 настоящего утверждения и того факта, что в рассматриваемых грамматиках отсутствуют бесполезные нетерминальные символы. Нетерминалы грамматики G' одновременно являются нетерминалами грамматики G, и для каждого из них в P' существует хотя бы одно правило вывода. □

Существование грамматики G очевидно в случае, когда во множестве T' есть токены, не участвующие в правилах грамматики G' в явном виде, что является одним из определяющих признаков легковесности.

Утверждения 2.3.2 и 2.3.3 позволяют нам сформулировать следующий способ разработки легковесных парсеров. Мы исходим из предположения, что разработчик легковесного парсера знает структуру языка, программы на котором требуется разбирать, достаточно хорошо, чтобы написать некоторую легковесную грамматику G' с символом Any. Эта грамматика будет упрощённой относительно некоторой «полной» грамматики G, однако в первом приближении грамматика G может порождать не всё множество корректных программ на целевом языке. Создаваемая грамматика G' итеративно уточняется по результатам тестирования легковесного парсера, сгенерированного по ней. Ожидается, что в некоторый момент она станет упрощённой относительно G, порождающей, как минимум, все корректные программы на целевом языке.

```
DIRECTIVE  : '#' ~[\n\r]*
COMMENT    : COMMENT_L|COMMENT_ML
COMMENT_L  : '//' ~[\n\r]*
COMMENT_ML : '/*' .*? '*/'
STRING     : STRING_STD|STRING_VERB|STRING_INT
STRING_STD : '"' ('\\"'|'\\\\'|.)*? '"'
STRING_VERB: '@'('"' ~["]* '"')+
STRING_INT : '$"' ('\\"'|'\\\\'|'{{'|STRING_INT_CODE|.)*? ["\n\r]
STRING_INT_CODE : '{' (STRING|CHAR|STRING_INT_CODE|.)*? '}'
CHAR : '\'' ('\\\''|'\\\\'|.)*? '\‘’
ID   : '@'?[_a-zA-Z][_0-9a-zA-Z]*

namespace_content       = opening_directive*! (attribute|namespace|namespace_member)*
opening_directive       = ('using'|'extern') Any ';'
namespace               = 'namespace' name '{' namespace_content '}'
namespace_member        = name? (enum|delegate|class_struct_interface)
enum                    = 'enum' name Any '{' Any '}' ';’?
delegate                = 'delegate' name before_body? ';'
class_struct_interface  = ('class'|'interface'|'struct') name Any '{' class_content_element* '}' ';'?
class_content_element   = attribute | keyword_marked_entities | name (keyword_marked_entities | class_member_tail)
keyword_marked_entities = enum | delegate | class_struct_interface | operator | event
operator                = 'operator' Any arguments class_member_tail
event                   = 'event' name class_member_tail
class_member_tail       = before_body? (block init_value? | initializer | ';')
before_body             = Any ':' (arguments|Any)*
initializer             = init_expression | init_value
init_expression         = '=>' (Any|block)* ';'
init_value              = '=' (Any|block)* ';'
name                    = (ID|arguments|'extern') name_tail_element*
name_tail_element       = ID|arguments|'extern'|'.'|'?'|'<' name_tail_element* '>'|'[' Any ']'|','|'::'
attribute               = '[' (Any|attribute)* ']'
block                   = '{' (Any|block)* '}'
arguments               = '(' (Any|arguments)* ')'

%%

%parsing {
    ignoreundefined
    fragment STRING_INT_CODE
    skip COMMENT STRING DIRECTIVE
    start namespace_content
}

%nodes {
    ghost  name_tail_element keyword_marked_entities initializer
    leaf name
}
```

Рис. 2.2. Упрощённая LL(1) грамматика языка C#, полученная в ходе итеративной разработки.

2.4. Язык описания упрощённых грамматик

Для создания легковесных парсеров по упрощённым грамматикам автором настоящего диссертационного исследования разработан генератор легковесных парсеров LanD (сокращение от «Language Description», созвучное основному термину островных грамматик «island») и одноимённый язык описания грамматик.

На рисунке 2.2 представлена упрощённая LL(1) грамматика языка C#, описанная на LanD. Процесс написания данной грамматики и проверка сгенерированного легковесного парсера подробно рассмотрены в разделе 2.7. Отметим, что описание языка C# для промышленного генератора компиляторов

ANTLR, использующего расширенную версию алгоритма LL(*) [154], содержит 1159 строк в спецификации парсера и 1101 строку в спецификации лексического анализатора[3], в то время как текст LL(1) грамматики для генератора LanD насчитывает всего 47 строк (включая пустые).

В первых 11 строках определяются токены грамматики. Лексический анализатор создаётся при помощи ANTLR, выражения для распознавания текста, соответствующего каждому из токенов, записаны в формате, поддерживаемом указанным генератором. В последующих строках записаны правила, определяющие нетерминальные символы. В правилах символ `*` обозначает ноль и более повторений элемента, `+` обозначает одно и более повторений, `?` означает опциональное наличие элемента, скобки `( )` применяются для группировки. Конструкции `*!` и `?!` означают, что возможный при построении таблицы разбора конфликт между пустой и непустой альтернативой нужно разрешить в сторону непустой.

После символов `%%` располагается секция опций. Опции из группы `parsing` позволяют управлять процессом разбора программы. В частности, опция `skip` задаёт токены, которые нужно отбрасывать, если они не являются ожидаемыми в текущем состоянии парсера, опция `start` задаёт стартовый символ грамматики. Опции из группы `nodes` позволяют управлять построением синтаксического дерева. Опция `leaf` задаёт символы грамматики, для которых узлы дерева должны быть листовыми; опция `ghost` задаёт нетерминалы, для которых узлы в дереве должны отсутствовать (при этом потомки этих узлов в дереве сохраняются). Генератор LanD также поддерживает указание пользовательских опций для символов грамматики: такие опции никак не интерпретируются самим LanD, но становятся атрибутами соответствующих символам узлов синтаксического дерева. После этого опции могут быть обработаны внешним инструментом, для которого они имеют некоторое специальное значение.

Как можно заметить, терминальный символ *Any* широко применяется в

[3] https://github.com/antlr/grammars-v4/tree/master/csharp

грамматике для обозначения мест, структура которых не является существенной. Большинство из этих мест являются «водой»: наиболее крупные «водные» области, заменяемые на *Any*, — содержимое блоков кода, соответствующих телам методов (нетерминал `block`). В правило для `block` помещается минимальная структурирующая информация: указываются парные токены `{ }`, ограничивающие соответствующую ему область, а также для корректного сопоставления границ допускается самовложение. Идентичный грамматический паттерн используется для описания сущностей `attribute` и `arguments`. Символ *Any* также участвует в описании «островов» `enum`, `operator`, `class_struct_interface`, обозначая внутри них области, структура которых неважна с точки зрения разработчика грамматики.

2.5. Упрощённая LR(1) грамматика

Хотя модифицированного LL(1) алгоритма достаточно для создания легковесных парсеров, описание реального языка программирования при помощи LL(1) грамматики — непростая задача, даже если эта грамматика должна быть легковесной. Конструкции языка, представляющие интерес (например, члены класса) обычно имеют одинаковое до некоторого места начало, следовательно, каждая из них не может быть представлена в виде одной непрерывной альтернативы одного нетерминального символа. Вместо этого разработчик вынужден дробить описания: создавать правило для общего начала и правила для возможных «хвостов».

На рисунке 2.2 отсутствуют отдельные правила для каждой из крупных синтаксических сущностей, интересных в рамках задачи привязки. Вместо этого общее начало описывается правилом `class_content_element`, а «хвосты» описываются правилами `keyword_marked_entities` и `class_member_tail`. Связь между конкретными языковыми конструкциями и данными правилами недостаточно прозрачна как для новичка, приступившего к изучению грамматики, так

и для автора грамматики спустя некоторое время после её написания.

Для того, чтобы дополнительно упростить разработку легковесных парсеров, нами вводится понятие легковесной грамматики с символом Any и упрощённой грамматики для случая LR(1), а также модифицируется алгоритм LR(1) синтаксического анализа.

Определение 2.5.1. Легковесной LR(1) грамматикой с символом Any будем называть LR(1) грамматику $G = (N, T, P, S)$, удовлетворяющую следующим условиям:

1. $Any \in T$.
2. $\exists p \in P : Any \in \text{rhs}(p)$.

Определение 2.5.2. Пусть $G = (N, T, P, S)$ — некоторая LR(1) грамматика, $Any \notin T$. Упрощённой относительно G будем называть легковесную LR(1) грамматику $G_s = (N_s, T_s, P_s, S_s)$, определяемую следующим образом:

1. Для G_s выполняются пункты 1–4 определения 2.2.2.
2. Если токен $b \in \text{FIRST}(\gamma_i)$, то не производится замена γ_i на Any_i, приводящая к существованию в грамматике G_s для некоторого $A \in N_s$ таких различных продукций p_1 и p_2, что p_1 применяется первой в выводе $A \overset{*}{\Rightarrow} \alpha Any_i \beta_1$ и p_2 применяется первой в выводе $A \overset{*}{\Rightarrow} \alpha b \beta_2$, где $\alpha, \beta_1, \beta_2 \in T_s^*$.

Говоря о грамматике G_s, упрощённой относительно LR(1) грамматики G, будем сокращённо называть её упрощённой LR(1) грамматикой.

Утверждение 2.5.1. Пусть $G' = (N', T', P', S')$ — легковесная LR(1) грамматика. G' является грамматикой, упрощённой относительно любой LR(1) грамматики $G = (N, T, P, S)$, удовлетворяющей следующим условиям:

1. Для грамматики G выполняются пункты 1–6 утверждения 2.3.3.

2. Если в грамматике G' для некоторого $A \in N'$ существуют такие различные продукции p'_1 и p'_2, что p'_1 применяется первой в выводе $A \overset{*}{\Rightarrow} \alpha Any\beta_1$ и p'_2 применяется первой в выводе $A \overset{*}{\Rightarrow} \alpha b\beta_2$, , где $\alpha, \beta_1, \beta_2 \in T'^*$, $b \in T' \setminus \{Any\}$, то $b \notin \mathrm{FIRST}(\gamma)$ для γ, соответствующей этому Any в грамматике G.

Доказательство. Выполнение пункта 1 определения упрощённой LR(1) грамматики для грамматики G' является прямым следствием утверждения 2.3.3. Поскольку в утверждении 2.3.3 факт принадлежности грамматики к классу LL(1) не влияет на ход рассуждений, те же рассуждения применимы и к случаю LR(1).

Пункт 2 определения упрощённой LR(1) грамматики эквивалентен пункту 2 настоящего утверждения. □

2.6. Парсинг на основе упрощённой LR(1) грамматики

Алгоритмы 2a–2e — модифицированные версии алгоритмов, необходимых для осуществления LR(1) синтаксического анализа [152, С. 259–266]. Серым выделены строки, добавленные для работы с упрощёнными грамматиками.

Алгоритм 2a является модифицированным алгоритмом LR(1) синтаксического анализа. Он позволяет осуществлять разбор программы, порождённой LR(1) грамматикой G, при помощи парсера, сгенерированного по упрощённой грамматике G_s. Как и в стандартном алгоритме, стек символов *SymbolsStack* хранит последовательность нетерминальных и терминальных символов, образующую текущий активный префикс [152, с. 256], стек состояний *StatesStack* хранит индексы состояний, через которые потребовалось пройти для получения данного префикса. $ACTION$ — таблица действий синтаксического анализа. Элемент $ACTION[s, t]$ хранит информацию о действии, которое должно быть предпринято парсером, находящимся в состоянии s, при встрече токена t. Существуют два основных типа действия: сдвиг и свёртка. Им соответствуют алго-

Алгоритм 2 Модификация LR(1) алгоритма: (a) алгоритм синтаксического анализа; (b) алгоритм обработки символа «Any»; (c) алгоритмы сдвига и свёртки; (d) алгоритм построения замыкания для множества пунктов; (e) вспомогательный алгоритм для поиска токенов, идущих после последовательных «Any».

```
1: Parse():
2:   SymbolsStack := [];
3:   StatesStack := [];
4:   StatesStack.Push(0);
5:
6:   t := Lexer.NextToken();
7:   while (true) do
8:     if(t = ERROR_TOKEN) then
9:       return false;
10:    end if;
11:    s := StatesStack.Peek();
12:    if (ACTION[s, t] ≠ null) then
13:      if (t = Any) then
14:        t := SkipAny(true);
15:      elif (ACTION[s, t] is ShiftAction a) then
16:        Shift(t, a.NextStateIdx);
17:        t := Lexer.NextToken();
18:      elif (ACTION[s, t] is ReduceAction a) then
19:        Reduce(a.ReductionAlternative);
20:      elif (ACTION[s, t] is AcceptAction) then
21:        Accept();
22:        return true;
23:      end if;
24:    elif (t ≠ Any) then
25:      t := Any;
26:    else
27:      t := Error(null);
28:    end if;
29:  end while;
```

(a)

```
Reduce(alt = X → Y1Y2...Yk):
  parent := new Node(X);
  for (idx from k to 1) do
    StatesStack.Pop();
    child := SymbolsStack.Pop();
    parent.Children.AddFirst(child);
  end for;

  s := StatesStack.Peek();
  StatesStack.Push(GOTO[s, X]);
  SymbolsStack.Push(parent);
  return StatesStack.Peek();

Shift(token, stateIdx):
  StatesStack.Push(stateIdx);
  SymbolsStack.Push(new Node(token));
  return StatesStack.Peek();
```

(c)

```
HasAnyProvokedAction(s, t):
  return ∃[A → α•, t] ∈ Is.AnyProvokedItems
    ∨ ∃[A → α•β, t'] ∈ Is.AnyProvokedItems:
      t ∈ FIRST'(βt')
    ∨ ∃[A → α•Anyβ, t'] ∈ Is.Items:
      t ∈ FIRST'(βt');
```

(e)

```
1:  SkipAny(recoveryIsEnabled):
2:    s := StatesStack.Peek();
3:    t := Lexer.CurrentToken();
4:    idx := Lexer.CurrentTokenIndex();
5:    while (ACTION[s, Any] is ReduceAction a) do
6:      s := Reduce(a.ReductionAlternative);
7:    end while;
8:
9:    s := Shift(Any, ACTION[s, Any].NextStateIdx);
10:
11:   stopTokens := { t' ∈ T | ACTION[s, t'] ≠ null }
12:     ∪ { t' ∈ T | HasAnyProvokedAction(s, t') };
13:   while (t ∉ stopTokens and t ≠ $) do
14:     t := Lexer.NextToken();
15:   end while;
16:
17:   if (t ∉ stopTokens) then
18:     if (recoveryIsEnabled) then
19:       Lexer.MoveTo(idx);
20:       return Error(stopTokens);
21:     else
22:       return ERROR_TOKEN;
23:     end if;
24:   end if;
25:
26:   return Lexer.CurrentToken();
```

(b)

```
Closure(I):
  repeat
    foreach ([A → α•Bβ, a] ∈ I.AnyProvokedItems) do
      foreach (B → γ ∈ Ps) do
        foreach (b ∈ FIRST'(βa)) do
          I.AnyProvokedItems ∪= {[B → •γ, b]}
        end foreach;
      end foreach;
    end foreach;
    foreach ([A → α•Bβ, a] ∈ I.Items) do
      foreach (B → γ ∈ Ps) do
        foreach (b ∈ FIRST(βa)) do
          I.Items ∪= {[B → •γ, b]}
        end foreach;
        foreach (b ∈ FIRST'(βa) \ FIRST(βa)) do
          I.AnyProvokedItems ∪= {[B → •γ, b]}
        end foreach;
      end foreach;
    end foreach;
  until (I.Items не изменилось
    ∧ I.AnyProvokedItems не изменилось);

I.AnyProvokedItems \= I.Items;

return I;
```

(d)

ритмы 2c — Shift и Reduce. $GOTO$ — таблица переходов. В ячейке $GOTO[s, X]$ содержится номер состояния, в которое парсер должен перейти из состояния s после свёртки некоторой части активного префикса в нетерминал X. Выполнение условий, нужных для исполнения строки 27 алгоритма 2a или строк 18–23 алгоритма 2b, означает, что переданная на вход легковесному парсеру программа является некорректной относительно грамматики G, и эта некорректность расположена не в «водной» области. В рамках рассуждений настоящей главы будем считать, что при этом указанные строки не выполняются и разбор завершается с ошибкой. Назначение данных строк подробно объясняется в главе 3.

Таблицы $ACTION$ и $GOTO$ строятся при помощи стандартных алгоритмов, однако замыкание множества пунктов, соответствующее некоторому состоянию синтаксического анализатора, строится как два множества: *Items* и *AnyProvokedItems* (алгоритм 2d). *Items* совпадает со стандартным замыканием и используется для построения таблицы действий и таблицы переходов. Для каждого пункта p вида $[A \to \alpha \bullet \beta, Any]$, содержащегося в *Items*, в *AnyProvokedItems* присутствуют один или несколько пунктов, также содержащих продукцию $A \to \alpha \bullet \beta$. В каждом из них символом предпросмотра является токен, который может следовать за Any, являющимся символом предпросмотра в p. Например, для грамматики

$$S \to A\,B$$
$$A \to a \mid \varepsilon$$
$$B \to Any\,b$$

замыкание пункта $[S \to \bullet\, A\,B\,,\ \$]$ будет содержать в *Items* пункты

$$[S \to \bullet\, A\,B\,,\ \$]$$
$$[A \to \bullet\, a\,,\ Any]$$
$$[A \to \bullet\, \varepsilon\,,\ Any],$$

а в *AnyProvokedItems* попадут пункты

$$[A \to \bullet\, a\,,\ b]$$
$$[A \to \bullet\, \varepsilon\,,\ b].$$

Информация из *AnyProvokedItems* используется при формировании множества стоп-токенов для ситуации, когда некоторой последовательности токенов разбираемой программы соответствует последовательность из нескольких *Any*, порождаемая упрощённой грамматикой.

Смысл модификации алгоритма 2a аналогичен случаю LL(1): по ходу разбора легковесный парсер осуществляет перевод программы с полного языка, описываемого грамматикой G, на язык, описываемый соответствующей упрощённой грамматикой G_s. В случае, если допустимое действие для текущей комбинации состояния парсера и входного токена отсутствует, парсер пытается интерпретировать текущий токен программы из $L(G)$ как начало подпоследовательности токенов, соответствующей *Any* в программе из $L(G_s)$. В случае, если для токена *Any* действие есть, парсер запускает метод SkipAny (алгоритм 2b), в котором последовательно осуществляются все доступные для *Any* свёртки (строки 5–7), после чего происходит сдвиг *Any* (строка 9). Важно отметить, что таблица *ACTION*, используемая алгоритмами, предварительно очищена от конфликтов сдвига/свёртки. По аналогии с генератором Yacc, в LanD данный конфликт разрешается в пользу сдвига. После переноса *Any* в активный префикс парсер ищет первый стоп-токен — токен, ожидаемый в текущем состоянии, либо в состоянии, в которое можно попасть, совершив ещё несколько сдвигов и свёрток для *Any* (строки 11–15). Последнее будет означать, что в программе найдена последовательность токенов, соответствующая нескольким подряд идущим *Any*. При встрече стоп-токена разбор продолжается стандартным образом.

Очевидно, что алгоритм 2a завершим, а его сложность совпадает со сложностью стандартного алгоритма LR(1) разбора и составляет $O(n)$.

Утверждение 2.6.1. Условие 3г определения 2.2.2 и условие 2 определения 2.5.2,

определяющие упрощённую LR(1) грамматику, существенны в том смысле, что, если убрать любое из них, парсер, сгенерированный по грамматике G_s, упрощённой относительно LR(1) грамматики G, будет разбирать не все программы из $L(G)$.

Для каждого из условий можно предложить тривиальную грамматику G и грамматику G_s, построенную по G с нарушением этого условия, а также привести пример такой программы из $L(G)$, которая не будет распознаваться парсером, сгенерированным по G_s и использующим алгоритмы 2a–2e. Так, условие 2 определения 2.5.2 не выполняется для грамматики

$$S \to a\ Any\ d \mid T\ e$$
$$T \to a\ b\ Any,$$

построенной по грамматике

$$S \to a\ b\ c\ d \mid T\ e$$
$$T \to a\ b\ c\ d.$$

Нетрудно проверить, что входная строка *abcd* принадлежит $L(G)$, однако не будет успешно распознана модифицированным алгоритмом LR(1) синтаксического анализа.

2.7. Эксперимент

2.7.1. Итеративная разработка грамматики

Как было отмечено в разделе 2.3.2, разработка легковесного парсера ведётся итеративно. На первой итерации пишется некоторая легковесная грамматика с символом *Any*, по ней генерируется легковесный парсер, затем парсер проверяется на программах, которые он, согласно замыслу разработчика, должен разбирать. На следующей итерации по результатам работы парсера грамматика уточняется, генерируется новый парсер и т. д. Процесс останавливается, когда

Таблица 2.1. Количества неразобранных файлов для каждой итерации уточнения грамматики C#.

	LanD	PascalABC.NET	Roslyn
1	8	-	-
2	0	39	-
3	0	0	209
4	0	0	31
5	0	0	3

по грамматике становится возможным сгенерировать парсер, успешно разбирающий все программы, на которых он проверяется. В этом случае делается предположение, что разработанная грамматика является упрощённой относительно грамматики, порождающей, как минимум, все правильные программы на целевом языке. Данный итеративный подход был опробован в эксперименте по разработке упрощённой LL(1) грамматики языка C#. Результатом стала грамматика на рисунке 2.2.

Для тестирования генерируемых парсеров использовались репозитории трёх проектов с открытым исходным кодом: самого генератора парсеров LanD (93 файла)[4], системы PascalABC.NET[5] (2725 файлов), компилятора Roslyn[6] (8027 файлов).

В таблице 2.1 приведена количественная информация, описывающая процесс разработки грамматики. Строки соответствуют итерациям уточнения грамматики, столбцы соответствуют разбираемым проектам. В каждой ячейке указано количество файлов проекта, соответствующего столбцу, не разобранных легковесным парсером, сгенерированным на итерации, соответствующей строке. Прочерк означает, что на указанной итерации легковесный парсинг проекта не проводился. Начав с наименьшего проекта — самого генератора LanD, — мы включали в рассмотрение более крупные проекты по мере того, как генерируемый по грамматике парсер оказывался способен разбирать все файлы проектов,

[4] https://github.com/alexeyvale/SYRCoSE-2019
[5] https://github.com/pascalabcnet/pascalabcnet
[6] https://github.com/dotnet/roslyn

уже включённых в рассмотрение.

За три итерации количество файлов, разбор которых завершался ошибкой, уменьшилось до нуля для LanD и PascalABC.NET. Однако количество неразобранных файлов для проекта Roslyn, добавленного к рассмотрению, оказалось значительным (209 файлов из 8027). В ходе проверки данных файлов выяснилось, что ошибку разбора вызывало использование возможностей языка C#, не задействованных в других проектах. Например, источником ошибок стали методы чтения и записи свойств, оформленные в виде выражений (expression bodied properties). Их учёт потребовал добавления в грамматику «водного» правила `init_expression` и уточнения правила `init_value`. Последующие две итерации позволили существенно сократить количество ошибок для Roslyn, однако для трёх файлов разбор не удалось осуществить даже в конце эксперимента.

Первый файл является тестовым для компилятора Roslyn и содержит класс с именем на японском языке, что не позволяет распознать это имя как идентификатор. Данная проблема в случае, если она будет возникать в коде, содержащем реальные функциональности программы, может быть решена расширением определения токена `ID` в грамматике на рисунке 2.2 символами нужного национального алфавита. Второй файл также относится к тестам компилятора и содержит оператор верхнего уровня (оператор, не вложенный в метод, класс, пространство имён). На момент проведения эксперимента подобное не было разрешено в актуальной версии языка C#, то есть программа являлась синтаксически некорректной. В третьем файле код имеет следующую структуру.

```
using ... ; namespace ... { ... }
#if false
using ... ; class ... { ... }
#endif
```

Поскольку LanD является языконезависимым инструментом, генерируемые пар-

Таблица 2.2. Количество сущностей, найденных Roslyn/LanD

	Перечисл.	Классы	Поля	Свойства	Методы
LanD	14 (13)	95 (94)	390 (390)	253 (248)	436 (431)
PABC	363 (356)	4622 (4611)	16753 (16720)	12350 (12326)	42386 (42248)
Roslyn	441 (437)	21621 (21583)	19737 (19606)	21919 (21886)	108400 (108040)

серы не имеют встроенной поддержки директив условной компиляции. Внешний препроцессор также не используется в описываемом эксперименте. Поэтому код, который должен исключаться из текста программы перед разбором, остаётся на месте, и парсер закономерно признаёт программу ошибочной. Отметим, что создание легковесного препроцессора для обработки директив условной компиляции и подключение его к парсеру являются тривиальными задачами, впоследствии нами решёнными.

С учётом вышеизложенных примечаний можно заключить, что в ходе эксперимента упрощённая грамматика языка C# разработана успешно. Парсер, сгенерированный по ней, осуществляет разбор всех возможных вариаций кода, встреченных в трёх исследовательских и промышленных проектах разного масштаба. При этом грамматика действительно является легковесной: как было сказано ранее, в полной спецификации парсера и лексического анализатора C# содержится 2260 строк, а разработанная нами грамматика почти в 50 раз короче. Заметим, что при создании парсера для одного конкретного проекта может оказаться достаточной и более короткая грамматика, если известны некоторые специфичные для этого проекта ограничения. К примеру, наличие в проекте определённых правил написания кода сокращает число возможных вариаций «островов» и «воды», а при разборе унаследованного кода можно быть уверенными в том, что в нём не используются новейшие средства языка.

2.7.2. Поиск крупных синтаксических сущностей

На следующем этапе эксперимента по синтаксическим деревьям разобранных файлов были посчитаны количества успешно выделенных LanD крупных синтаксических сущностей, являющихся интересными с точки зрения задачи привязки к коду. Также было подсчитано количество таких сущностей, представленных в синтаксических деревьях, построенных промышленным компилятором языка C# Roslyn (данный компилятор применяется в среде разработки Visual Studio).

Сущности сгруппированы по пяти категориям: перечисления, классы, поля, свойства, методы. Группировка проведена в соответствии с иерархией классов, представляющих узлы синтаксического дерева в Roslyn. В категорию «Поля» попадают собственно поля, а также события, описанные без методов доступа. В столбце «Свойства» учитываются сами свойства, индексаторы и события с явным образом описанными `add` и `remove`. «Методами» считаются сами методы, конструкторы, деструкторы и операторы.

В таблице 2.2 приведены результаты подсчётов. Как можно заметить, для всех проектов во всех категориях LanD обнаруживает больше сущностей, чем Roslyn. Анализ расхождений показал, что причиной является активное использование в проектах директивы условной компиляции `#if`. К примеру, в PascalABC.NET конструкция `#if DEBUG` часто используется для включения отладочного кода. Для легковесного парсера C# директивы являются обычными пропускаемыми лексемами, в связи с чем в статистику LanD сущности попадают независимо от того, определён ли символ для секции `#if`, в которой они находятся.

Для последующих экспериментов и использования сгенерированного парсера в задаче привязки к коду нами был написан легковесный препроцессор, обрабатывающий директивы условной компиляции.

Глава 3

Дополнительные механизмы обработки символа «Any»

В настоящей главе описываются расширения языка LanD и вспомогательные механизмы обработки символа *Any*, позволяющие ещё больше сократить размер упрощённой грамматики при создании легковесных парсеров, а также повысить устойчивость парсера к ошибкам в программе. Проводится экспериментальная проверка легковесных LL(1) и LR(1) парсеров для языков C# и Java, сгенерированных по разработанным с применением всех описанных нововведений упрощённым грамматикам.

3.1. Параметризация «Any»

Рассмотрим следующую LL(1) грамматику:

$$A \to (a \,|\, b \,|\, c\,)+\ B$$
$$B \to C\ D\ a$$
$$C \to (f \,|\, g \,|\, h\,)+\ |\,\varepsilon$$
$$D \to (d \,|\, e\,)+\,.$$

Допустим, мы хотим записать упрощённую относительно неё грамматику как

$$A \to \mathit{Any}\ B$$
$$B \to \mathit{Any}\ a.$$

Такая грамматика не удовлетворяет условию 3г определения 2.2.2: хотя каждое из *Any* заменяет область, в которой не могут содержаться токены, идущие сразу после этого *Any*, в грамматике возможен вывод $A \overset{+}{\Longrightarrow} \mathit{Any}\ \mathit{Any}\ a$, при этом первое *Any* соответствует области, в которой может находиться токен a.

В язык LanD добавлены средства для передачи дополнительной информации на этап обработки *Any*. С вхождением *Any* в грамматику могут быть связаны опции *Except*, *Include* и *Avoid*. Параметрами для каждой из них являются литералы или определённые в грамматике токены. Если параметром является литерал, для него на этапе построения парсера автоматически определяется токен, причём, если данный литерал фигурирует только в параметрах опций *Any*, для соответствующего токена на глобальном уровне устанавливается опция `skip` из группы `parsing`, означающая, что его нужно пропускать везде, где он не ожидается явным образом.

Отметим, что в генераторе парсеров LightParse символ `ANY` также расширяется дополнительной опцией *Except* [59]. Она используется, чтобы скомпенсировать отсутствие анализа внешнего контекста на этапе конструирования множества допустимых токенов. В нашей интерпретации символы, указанные как параметры для *Except*, *Include* или *Avoid*, компенсируют отсутствие информации о составе области, вместо описания которой в грамматике использован символ *Any*.

Аргументы опции *Except* позволяют явным образом задать символы, которые не могут встретиться в области, соответствующей *Any*, и являются признаком того, что эта область точно закончилась. Они используются вместо обычного множества стоп-токенов, конструируемого при пропуске этого *Any*. Для приведённого ранее примера грамматика, удовлетворяющая определению упрощённой, может быть записана как

$$A \to \mathit{AnyExcept}(d, e)\, B$$
$$B \to \mathit{Any}\, a.$$

При записи базовыми средствами языка LanD наиболее простая упрощённая грамматика выглядела бы так:

$$A \to \mathit{Any}\, B$$
$$B \to \mathit{Any}(d \mid e)\, \mathit{Any}\, a.$$

Аргументы опции *Include* позволяют явным образом задать некоторые токены, которые могут встретиться в области, соответствующей *Any*. В грамматиках, приводимых в разделе 3.4, опция *Include* используется для удобного определения таких токенов «на лету».

Опция *Avoid* добавлена для контролируемого обнаружения ошибок в разбираемой программе. Её аргументы — это токены, присутствие которых в области, сопоставляемой с *Any*, сигнализирует о некорректности программы. На рисунке 3.1 изображена разработанная нами упрощённая LL(1) грамматика языка описания грамматик, используемого генератором парсеров Yacc[1]. В определении нетерминала `block` посредством *AnyAvoid* сказано, что внутри блока кода не может встретиться начало нового правила грамматики или разделитель секций грамматики. Обнаружение любого из соответствующих токенов означает, что в текущем разбираемом блоке отсутствует закрывающая фигурная скобка. Своевременное обнаружение ошибки позволяет не пропустить посредством *Any* правила, идущие после незакрытого блока. При встрече токена, являющегося параметром опции *Avoid*, запускаются алгоритмы восстановления от ошибки, описанные в разделе 3.3.

С учётом *Avoid* условие цикла **while** в алгоритмах 1b и 2b преобразуется в

$$t \notin stopTokens \wedge t \notin anyAvoidSet \wedge t \neq \$,$$

где *anyAvoidSet* — множество токенов, указанных в качестве параметров опции *Avoid* для пропускаемого *Any*.

При LR(1) разборе не всегда известно, какое именно вхождение *Any* в грамматику обрабатывается в настоящий момент. В то же время эта информация нужна, чтобы получить доступ к аргументам связанных с *Any* опций. Чтобы сделать работу с опциями *Any* возможной в случае LR(1), к стандартным конфликтам переноса/свёртки и свёртки/свёртки был добавлен *Any*/*Any*

[1] Строго говоря, парсер разработан для разбора грамматик, написанных для генератора парсеров GPPG, использующего Yacc-оподобный синтаксис.

```
COMMENT     : COMMENT_L|COMMENT_ML
COMMENT_L   : '//' ~[\n\r]*
COMMENT_ML  : '/*' .*? '*/'
STRING      : STRING_STD|STRING_ESC
STRING_SKIP : '\\"' | '\\\\'
STRING_STD  : '"' (STRING_SKIP|.)*? '"'
STRING_ESC  : '@'('"' ~["]* '"')+
LITERAL     : '\'' ('\\\''|'\\\\'|.)*? '\''
DECLARATION_CODE : '%{' (STRING|COMMENT|.)*? '%}'
RULE_HEADER : %linestart ID
ID  : [_a-zA-Z][_0-9a-zA-Z]*
DECLARATION_NAME    : '%'ID
SECTIONS_SEPARATOR  : %linestart '%%'

grammar         =   (declaration | DECLARATION_NAME | Any)* SECTIONS_SEPARATOR (rule | Any)* (SECTIONS_SEPARATOR Any)?
declaration     =    symbol_declaration_header symbol_declaration_element+
symbol_declaration_header  = ('%token' | '%left' | '%nonassoc' | '%right' | '%type' | '%start') ('<' ID '>')?
symbol_declaration_element = ID | LITERAL
rule            =   RULE_HEADER ':' (alternative ('|' alternative)* ';')
alternative     =   (alternative_element | block)* | AnyError // Recovery when ; is missing
alternative_element    =    ID | LITERAL | '%prec'
block           =   '{' (AnyAvoid(RULE_HEADER, SECTIONS_SEPARATOR) | block)+ '}' // Recovery when closing } is missing

%%

%parsing {
    ignoreundefined
    start grammar
    skip COMMENT STRING DECLARATION_CODE
    recovery
}

%nodes {
    ghost declaration
    leaf alternative_element symbol_declaration_element symbol_declaration_header
}
```

Рис. 3.1. Упрощённая LL(1) грамматика спецификаций для генератора парсеров Yacc.

конфликт. Он выявляется в случае, если существует состояние парсера, в котором несколько пунктов содержат точку перед Any, и с этими Any связаны разные наборы опций, или одноимённые опции имеют различающиеся наборы аргументов. Для успешной генерации парсера данный конфликт должен быть устранён разработчиком грамматики.

3.2. Контроль уровня вложенности

«Водные» области, вместо описания которых используется токен Any, могут включать в себя подобласти, заключённые в скобки. Интуитивно такие подобласти воспринимаются как единое целое, и разработчиком грамматики может быть упущен тот факт, что внутри них содержатся токены, являющиеся стоп-токенами для Any. В результате написанная грамматика будет нарушать условие 3г определения 2.2.2, необходимое и для случая LL(1), и для случая LR(1), и разбор программы произвести не удастся. Например, если в коде

```
int[,] a1 = new int[2, 2],
    a2 = new int[3, 3];
```

```
Person p1 = new Person { Name = "Alexey", Age = 30 },
    p2 = new Person { Name = "Stanislav", Age = 57 };
DateTime d1 = new DateTime(2019, 5, 29),
    d2 = new DateTime(2019, 5, 31);
```

необходимо распознать только перечень определяемых полей, а структура инициализаторов значения не имеет, разработчик грамматики интуитивно напишет следующие правила:

```
fields = type name init? (',' name init?)* ';'
init   = '=' Any
```

Символ , отделяет описания друг от друга и потому является одним из стоп-токенов для *Any*. В то же время символ , может присутствовать внутри инициализатора в списке аргументов, индексов массива или при инициализации объекта. Чтобы грамматика удовлетворяла определению упрощённой, разработчик вынужден сформулировать правила, продемонстрированные на рисунке 3.2a.

Чтобы облегчить написание грамматики в подобных случаях, на уровень лексического анализа нами был добавлен учёт парных конструкций. На рисунке 3.2b при помощи специального синтаксиса определены все возможные типы пар, которые могут встретиться в программе на разбираемом языке. При наличии пар, определённых в грамматике, считается, что внутри областей, соответствующих *Any*, такие пары могут встречаться в любом количестве и любым способом вкладываться друг в друга. Остановить процесс пропуска *Any* разрешается только на стоп-токене, находящемся на том же уровне вложенности, что и токен, на котором пропуск начался. Таким образом в области, соответствующей *Any*, становятся допустимыми вложенные ограниченные скобками подобласти, содержащие стоп-токены: они становятся невидимыми для парсера.

На рисунке 3.3 изображены два случая, в которых пропуск *Any* начинается и заканчивается на одной и той же глубине. Токен *a* является началом области,

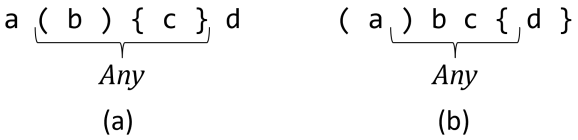
```
fields = type name init? (',' name init?)* ';'
init   = '=' water
s_water = '[' (Any | s_water)+ ']'
r_water = '(' (Any | r_water)+ ')'
c_water = '{' (Any | c_water)+ '}'
water   = (Any | c_water | r_water | s_water)+
```

(a)

```
CURVE_BRACKETED : %left '{' %right '}'
ROUND_BRACKETED : %left '(' %right ')'
SQUARE_BRACKETED : %left '[' %right ']'
…
fields = type name init? (',' name init?)* ';'
init   = '=' Any
```

(b)

Рис. 3.2. Использование механизма учёта пар: (a) фрагмент грамматики, в которой не определены пары; (b) фрагмент грамматики, в которой определены пары для скобочных конструкций.



(a) (b)

Рис. 3.3. Последовательности токенов, соответствующие «Any», начинающиеся и заканчивающиеся на одном уровне вложенности: (a) корректная ситуация; (b) некорректная ситуация.

соответствующей *Any*, *c* — стоп-токен. Очевидно, что тот способ, которым символ *Any* сопоставляется на рисунке 3.3b, нарушает семантическую целостность ограниченных скобками областей. Мы расцениваем такое использование *Any* как плохую практику. Если лексический анализатор возвращает токен, не являющийся стоп-токеном и означающий, что некоторая скобочная конструкция завершилась и нужно подняться на уровень вложенности меньший, чем тот, на котором был запущен пропуск *Any*, парсер сообщает об ошибке. В этом случае грамматика, по которой генерируется парсер, должна быть доработана.

3.3. Восстановление от ошибок

3.3.1. Алгоритмы

Как отмечалось в разделе 2.3, использование упрощённой грамматики для генерации легковесных парсеров, как и использование островных грамматик в целом, делает парсер нечувствительным к возможным ошибкам, допущенным в «водных» областях программы. Однако ошибки могут встретиться и в местах, описанных в упрощённой грамматике подробно. Более того, поскольку легковесные парсеры планируется использовать при привязке и переприязке

к элементам прорезающих функциональностей в процессе решения программистом некоторой задачи, разбираемая программа может оказаться некорректной в силу того, что её редактирование в рамках работы над задачей уже началось, но ещё не завершилось. Для алгоритмов 1 и 2 необходимо предусмотреть механизмы восстановления от ошибки, чтобы обеспечить работоспособность инструмента разметки в таких условиях.

Кроме того, как уже было сказано, на практике упрощённая грамматика G_s пишется интуитивно без обращения к «полной» грамматике. Разработчик грамматики G_s пытается описать её так, чтобы она являлась упрощённой для G, порождающей, как минимум, все корректные программы на целевом языке. Однако на практике может оказаться, что полученная G_s является упрощённой для некоторой G', близкой к G, но не описывающей часть программ, подлежащих разбору. Например, разработчик может не учесть, что в «водных» областях программы присутствуют элементы, имеющие одинаковое с «островами» начало. В результате легковесный парсер начнёт разбор «водной» области как «острова» и завершит разбор с ошибкой, поскольку с некоторого места обнаружится несоответствие структуре «острова», описанной в грамматике. Восстановление от ошибки может позволить разобрать программу до конца, чем устранит необходимость дальнейшего итеративного улучшения грамматики.

В модифицированные LL(1) и LR(1) алгоритмы парсинга добавляются специальные методы восстановления, основанные на обработке символа Any — алгоритмы 3 и 4. Обрабатываемые ими ошибки делятся на два типа. Первый тип — неожиданный токен во входном потоке. Для LL(1) разбора первый тип ошибок случается, когда парсер не может сопоставить текущий токен или выбрать альтернативу для нетерминала на вершине стека и токен Any при этом не является допустимым (строка 26 алгоритма 1а). В случае LR(1) разбора первый тип ошибки возникает, если для текущего токена и для токена Any отсутствуют допустимые действия в текущем состоянии парсера (строка 25 алгоритма 2а). Ошибки второго типа возникают, когда в ходе обработки Any не удаётся найти

ни одного стоп-токена и весь входной поток пропускается до конца, или когда в ходе пропуска обнаруживается токен, указанный как аргумент опции *Avoid* (строки 16–21 алгоритма 1b, строки 18–23 алгоритма 2b).

По аналогии со стандартными алгоритмами восстановления [152, с. 283–285] для ошибок первого типа определяется множество нетерминалов, на которых можно возобновить разбор. Эти нетерминалы называются *символами восстановления*. Для грамматики $G_s = (N_s, T_s, P_s, S_s)$ множество символов восстановления определяется следующим образом:

$$RecoverySymbols = \Big\{ n \in N_s \mid n \overset{*}{\Rightarrow} Any\,\alpha \\ \wedge \nexists n' \in N_s \colon (n \overset{*}{\Rightarrow} n'\,Any\,\alpha \wedge n' \overset{*}{\Rightarrow} \varepsilon) \Big\}, \alpha \in (N_s \cup T_s)^*.$$

Данное множество вычисляется один раз после построения необходимых для разбора таблиц. Разработчик грамматики может отключить восстановление от ошибок, разрешить восстановление на всех символах из *RecoverySymbols* или на символах из некоторого его подмножества.

В контексте задачи легковесного парсинга символы восстановления имеют специфическую семантику. Они обозначают точки принятия решения, в которых можно сделать выбор в пользу «островной» или «водной» (начинающейся с *Any*) альтернативы. При восстановлении происходит возврат к состоянию стека, в котором было ошибочно решено интерпретировать часть входного потока, начинающуюся с некоторого токена, как «остров». Последовательность, начинающаяся с этого токена и заканчивающаяся токеном, на котором обнаружили ошибку, интерпретируется как соответствующая символу *Any* в «водной» альтернативе символа восстановления. При этом возврат к токену, на котором было принято неверное решение, не требуется. На рисунке 3.4 нетерминал `entity` является одним из символов восстановления. Он позволяет парсеру распознавать классы, перечисления, методы и поля как «острова», в то время как конструкторы, делегаты, операторы и т. д. пропускаются как «вода» с привлечением

механизма восстановления.

Восстановление, запущенное для ошибок первого типа (неожиданный токен), не нарушает линейность алгоритма, поскольку разбор возобновляется на том же токене, на котором возникла ошибка. Однако если действовать таким способом для ошибок второго типа (ошибка при пропуске Any), можно потерять большое количество информации о важных элементах программы, поскольку значительная часть входного потока, возможно, не должна была быть пропущена. Для ошибок второго типа осуществляется ограниченный возврат во входном потоке. В алгоритмах 1b и 2b вызов $Lexer.\mathrm{MoveTo}(idx)$ задаёт в качестве индекса текущего рассматриваемого токена индекс того токена, с которого началась обработка Any. В позиции этого токена предпринимается попытка восстановления. Множество $RecoveredIn$ хранит индексы токенов, в позиции которых восстановление уже было проведено. Таким образом гарантируется, что между двумя восстановлениями разбор продвинется не менее чем на один токен вперёд. В разделе 3.3.2 проанализировано влияние возвратов на временную сложность синтаксического анализа.

Алгоритм 3 Восстановление от ошибок в ходе легковесного LL(1) парсинга.

```
1:  function Error(stopTokens):
2:      if (Lexer.CurrentTokenIndex() ∈ RecoveredIn) then
3:          return ERROR_TOKEN;
4:      end if;
5:      RecoveredIn ∪= { Lexer.CurrentTokenIndex() };
6:      currentNode := Stack.Pop();
7:      do
8:          if (currentNode.Parent ≠ null) then
9:              maxChildIndex := currentNode.Parent.Children.Count;
10:             indexOfCurrent := currentNode.Parent.Children.IndexOf(currentNode);
11:             for (i from indexOfCurrent + 1 to maxChildIndex) do
12:                 Stack.Pop();
13:             end for;
14:         end if;
15:         currentNode := currentNode.Parent;
16:     while(currentNode ≠ null ∧ (currentNode.Symbol ∉ RecoverySymbols
17:         ∨ Any = GetDerivation(currentNode)[1]
18:         ∨ IsUnsafeAny(stopTokens)));
19:     if (currentNode ≠ null) then
20:         return SkipAny(false);
21:     else
22:         return ERROR_TOKEN;
23:     end if;
```

Алгоритм 3 — основанный на обработке *Any* алгоритм восстановления от ошибок для случая LL(1). В нём активно используется то, что элементами стека символов в нашей реализации являются узлы выстраиваемого АСД, с каждым из которых связан некоторый символ грамматики. В произвольный момент разбора доступно частично построенное синтаксическое дерево. Часть кроны этого дерева состоит из узлов-заготовок для тех терминальных символов, которые ещё предстоит сопоставить, и нетерминалов, которые ещё предстоит развернуть. Эти узлы лежат на стеке символов. При возникновении ошибки символ восстановления ищется среди предков узла, находящегося на вершине стека и соответствующего символу, для которого не удалось совершить действие с текущим входным токеном. В цикле **do-while** алгоритма (строки 7–18) происходит подъём от этого узла по уже построенной части синтаксического дерева. На каждом шаге для текущего узла *currentNode*, который соответствует некоторому символу грамматики x и появился на стеке в результате замены некоторого нетерминала A на последовательность $\alpha x \beta$, со стека снимаются все узлы-заготовки, соответствующие символам из β (строки 9–13). После этого текущим узлом считается непосредственный предок *currentNode*.

Цикл завершается, если в некоторый момент *currentNode* соответствует символу восстановления и удовлетворяет двум дополнительным условиям. Во-первых, правило вывода, в соответствии с которым *currentNode* развернули изначально, не должно соответствовать «воде», то есть, его правая часть не должна начинаться с символа *Any*. В противном случае разбор уже пошёл по тому пути, который мы могли бы использовать для восстановления, и именно на этом пути возникла ошибка. Для проверки данного условия вызывается метод GetDerivation (строка 17). Он принимает на вход поддерево с корнем в текущем символе восстановления и возвращает последовательность листьев — частично порождённую из этого символа часть программы из $L(G_s)$. Заметим, что здесь и далее в диссертации индексация массивов начинается с единицы. Во-вторых, в случае, если ошибка произошла при пропуске *Any*, нет смысла восстанавли-

ваться на некотором символе, пропуск «воды» для которого приведёт к той же самой ошибочной ситуации. Метод IsUnsafeAny (строка 18) сравнивает множества стоп-токенов и аргументы опции *Avoid* для *Any*, на котором произошла ошибка, и *Any*, с пропуска которого планируется продолжить разбор.

Чем выше мы поднимаемся по дереву, тем большая часть входного потока будет интерпретирована как «вода» по завершении восстановления.

Алгоритм 4 Восстановление от ошибок в ходе легковесного LR(1) парсинга.

```
1:  function Error(stopTokens):
2:      if (Lexer.CurrentTokenIndex() ∈ RecoveredIn) then
3:          return ERROR_TOKEN;
4:      end;
5:      RecoveredIn ∪= { Lexer.CurrentTokenIndex() };
6:      lastMatched := ε;
7:      PDI := ∅;    // possible derivation items
8:      basePDI := ∅;
9:      do
10:         if (SymbolsStack.Count > 0) then
11:             lastMatched := SymbolsStack.Pop();
12:         end if;
13:         StatesStack.Pop();
14:         if (StatesStack.Count > 0) then
15:             s := StatesStack.Peek();
16:             basePDI := { i = X → α•Yβ | i ∈ STATE[s], Y = lastMatched.Symbol,
17:                 (PDI = ∅ ∨ ∃i' ∈ PDI : i' = X → αY•β) };
18:             PDI := basePDI;
19:             do
20:                 PDI ∪= { i = X → α•Yβ | i ∈ STATE[s], ∃i' ∈ PDI : i' = Y → •β' };
21:             while (PDI изменяется);
22:         end if;
23:     while (StatesStack.Count > 0 ∧ (| basePDI | = | PDI |
24:         ∨ ∄i ∈ PDI \ basePDI : i = X → α•Yβ, Y ∈ RecoverySymbols
25:         ∨ Any = GetDerivation(lastMatched)[1]
26:         ∨ IsUnsafeAny(stopTokens)));
27:     if (StatesStack.Count > 0) then
28:         return SkipAny(false);
29:     else
30:         return ERROR_TOKEN;
31:     end if;
```

Алгоритм 4, обеспечивающий восстановление для случая LR(1), является более сложным, как и сам восходящий синтаксический анализ. В отличие от случая LL(1) неизвестно, какая именно синтаксическая сущность анализируется в момент ошибки. На основе информации, присутствующей на стеках, строится множество возможных кандидатов.

На первой итерации цикла **do-while** (строки 9–26) последний распознанный символ снимается со стека одновременно с состоянием, в которое парсер

перешёл после его успешного распознавания (строки 10–13). Затем конструируется множество *basePDI* (basic possible derivation items). В него попадают избранные продукции, присутствующие в состоянии, индекс которого теперь находится на вершине *StatesStack*, — те, в которых точка расположена перед последним снятым со стека *SymbolsStack* символом. Определяемые этими продукциями нетерминалы — то, что парсер, возможно, пытался распознать, когда обнаружил ошибку. На основе множества *basePDI* конструируется множество *PDI*, принцип его построения можно назвать обратным к алгоритму построения замыкания 2d: изначально оно совпадает с *basePDI*, затем в него добавляются продукции с точкой перед символами, продукции для которых уже есть в *PDI* и содержат точку в самом начале. Таким образом мы постепенно продвигаемся ко всё более и более объемлющим синтаксическим конструкциям, которые могли анализироваться в момент ошибки. Цикл завершается, если в *PDI* добавляется продукция $A \to \bullet\alpha$, где A — символ восстановления, и для Any, за счёт которого можно пропустить область программы от начала A до вызвавшего ошибку токена, выполняются условия, оговорённые ранее для случая LL(1). В противном случае начинается следующая итерация, на которой продукции для *basePDI* отбираются с учётом *PDI*, сконструированного на предыдущей итерации.

В разделе 3.4 экспериментально подтверждается, что предложенные алгоритмы восстановления от ошибок позволяют осуществлять разбор программ из полного языка при помощи парсера, сгенерированного по грамматике, упрощённой относительно грамматики, порождающей собственное подмножество целевого языка.

3.3.2. Анализ сложности алгоритмов

Обработка ошибок, возникающих в ходе пропуска Any, требует ограниченного возврата во входном потоке. То, насколько при этом увеличится время работы алгоритма парсинга, зависит от общего количества и длины последова-

тельностей токенов, для которых осуществляется возврат. Запрет многократного восстановления на одном и том же токене позволяет утверждать, что к каждой позиции во входном потоке возврат может быть произведён только один раз. Следовательно, в худшем случае для всех токенов входной программы, кроме первого, выполняется следующий сценарий: на токене запускается обработка Any, в ходе обработки входной поток вычитывается до конца, происходят ошибка и возврат к этому токену. После этого запускается восстановление, токен успешно сопоставляется при помощи «водной» альтернативы символа восстановления, и происходит переход к следующему токену.

В описанной ситуации каждый токен анализируется столько раз, каков его порядковый номер (при условии, что нумерация начинается с единицы). Для i-того токена происходят $i-1$ его рассмотрений в ходе пропусков Any, начатых на предыдущих токенах и на нём самом, и ещё один анализ осуществляется при окончательном сопоставлении. Поскольку сам по себе возврат сводится к переприсваиванию индекса текущего токена, за счёт него количество проходов через остальные токены не увеличивается.

Данный наихудший сценарий реализуется для входных строк `ac$`, `aac$`, `aaac$` и им подобных и парсера, сгенерированного по следующей LL(1)-грамматике:

$$S \to a\ Any\ b \mid Any\ S \mid \varepsilon$$

Работа данного парсера для строки `aac$` пошагово представлена в таблице 3.1. В первом столбце показаны счётчики для каждого из токенов входной строки, за исключением признака конца строки. Значение счётчика показывает, сколько раз на текущий момент был проанализирован токен, находящийся в соответствующей позиции. Во втором столбце для каждого из шагов точкой обозначена позиция перед текущим токеном. В третьем столбце описывается действие, совершаемое для текущего токена.

Суммарное количество обработок токенов равно $\frac{1}{2}n^2+\frac{1}{2}n$, это означает, что

Таблица 3.1. Пример обработки входной строки «aac$»

Счётчики	Позиция	Действие
000	•aac$	Сопоставить
100	a•ac$	Начать пропуск *Any*
111	aac•$	Вернуться к токену 2
111	a•ac$	Сопоставить
121	aa•c$	Начать пропуск *Any*
122	aac•$	Вернуться к токену 3
122	aa•c$	Сопоставить
123	aac$•	Признать строку корректной

предложенные алгоритмы восстановления от ошибки в худшем случае имеют сложность $O(n^2)$. Однако эксперименты показывают, что в ходе легковесного разбора процент восстановлений, требующих возврата, пренебрежимо мал как в сравнении с общим количеством токенов программы, так и в сравнении с общим количеством восстановлений. Так, во всех проектах на языке Java, рассмотренных в разделе 3.4.2, вместе взятых 27393 файла состоят из 26255589 токенов, при этом общее число восстановлений равно 32683 для алгоритма LL(1) и 31861 для LR(1). Из этих восстановлений только 20 для каждого из типов парсинга были предприняты после ошибки, возникшей при пропуске *Any*. Следовательно, модифицированные алгоритмы LL(1) и LR(1) синтаксического анализа, использующие восстановление от ошибок, по-прежнему можно считать линейными.

3.4. Эксперимент

Для проверки алгоритмов, описанных в настоящей главе и в главе 2, разработаны легковесные LL(1) и LR(1) грамматики с символом *Any*, являющиеся упрощёнными для подмножеств программ на языках C# и Java. LR(1) грамматика на рисунке 3.4 и парная ей LL(1) грамматика являются упрощёнными для программ на C#, в которых членами класса являются только методы, поля и свойства. LR(1) грамматика на рисунке 3.5 и парная ей LL(1) грамматика являются упрощёнными для программ на Java, в которых членами класса являются только методы и поля. Остальные сущности, реально присутствующие внутри

классов в полных языках C# и Java, должны распознаваться как «вода» за счёт алгоритмов восстановления от ошибок и наличия символа восстановления `entity` в каждой из грамматик. «Водной» альтернативой в правиле для `entity` является альтернатива `water_entity`.

Сгенерированные по грамматикам парсеры применяются для анализа крупных проектов с открытым исходным кодом. Для каждого языка на первом этапе эксперимента легковесные LL(1) и LR(1) парсеры анализируют все файлы во всех выбранных для тестирования проектах. Затем для каждого файла осуществляется обход построенного парсером синтаксического дерева. В файл-отчёт извлекаются имена и типы представленных в дереве «островных» синтаксических сущностей.

На втором этапе эксперимента те же файлы анализируются полным парсером языка. Инструмент Roslyn используется как полный парсер языка C#, парсер Java генерируется по полной грамматике[2] генератором ANTLR. Отметим, что полная грамматика Java содержит 211 строк спецификации лексического анализатора и 615 строк описания синтаксиса, используемая нами грамматика содержит 42 строки (включая пустые). По деревьям, построенным полным парсером, также строится отчёт с информацией о выявленных синтаксических элементах.

На третьем этапе эксперимента отчёты первого и второго этапов автоматически сравниваются между собой. Совпадения исключаются из рассмотрения, в ручном режиме проверяются сущности, выявленные одним парсером и не выявленные другим.

Для каждого из языков приведена таблица (3.2 для C#, 3.4 для Java), в которой строки соответствуют разбираемым проектам, столбцы соответствуют типам «островов». В столбце «Классы» для языка C# указано суммарное количество найденных классов, интерфейсов и структур; для языка Java — количество классов и интерфейсов. Данные сущности объединены по причине их

[2] `https://github.com/antlr/grammars-v4/tree/master/java`

синтаксической схожести, приведение результатов для каждой из них в отдельности было бы избыточным. В ячейке таблицы указано количество «островов» соответствующего столбцу типа, найденных легковесным парсером для соответствующего строке проекта. Установлено, что это количество одинаково для LL(1) и LR(1) разборов. В случае, если легковесный парсер находит меньше «островных» сущностей, чем полный, количество пропущенных сущностей указано в скобках. Также присутствует столбец «Всего файлов», информация в нём позволяет оценить масштаб каждого проекта.

3.4.1. Легковесный разбор программ на языке C#

Для языка C# рассматриваются 5 проектов с открытым исходным кодом, принадлежащих к различным предметным областям:

- **Roslyn** включает в себя компиляторы языков C# и Visual Basic, важным для эксперимента моментом является наличие в проекте большого количества тестовых файлов для компилятора, содержащих структурно сложные, редко встречающиеся и заведомо некорректные варианты программ на C#;
- **PascalABC.NET** состоит из интегрированной среды разработки и компилятора языка программирования PascalABC.NET, данный проект имеет долгую историю, находящую отражение в значительных объёмах унаследованного кода, написанного разработчиками разного уровня опытности;
- **Mono** является сторонней реализацией .NET Framework и включает в себя компилятор языка C#, виртуальную машину общеязыковой среды выполнения, большое количество библиотек, а также, как и в случае Roslyn, большое количество тестовых файлов;

```
DIRECTIVE    : '#' ~[\n\r]*

COMMENT      : COMMENT_L|COMMENT_ML
COMMENT_L    : '//' ~[\n\r]*
COMMENT_ML   : '/*' .*? '*/'

STRING       : STRING_STD|STRING_VERB|STRING_INT
STRING_STD   : '"' ('\\"'|'\\\\'|.)*? '"'
STRING_VERB  : '@'('"' (~["]|'""')* '"')+
STRING_INT   : '$"' ('\\"'|'\\\\'|'{{'|STRING_INT_CODE|.)*? '"' | '$@' ('"' ('""'|'{{'|STRING_INT_CODE|.)*? '"')+
STRING_INT_CODE : '{' (STRING|CHAR|.)*? '}'

CHAR         : '\'' ('\\\''|'\\\\'|.)*? '\''
MODIFIER     : 'ref'|'fixed'|'public'|'private'|'protected'|'internal'|'static'|'virtual'|'const'
    |'override'|'new'|'sealed'|'unsafe'|'readonly'|'abstract'|'volatile'|'async'|'partial'
CLASS_STRUCT_INTERFACE : 'class'|'struct'|'record'|'interface'
ID           : '@'?[_a-zA-Z\u0430-\u044F\u0410-\u042F][_0-9a-zA-Z\u0430-\u044F\u0410-\u042F]*
GENERAL_ATTRIBUTE_START    :    '[' [ \t\r\f\n]*? ('assembly'|'module')

CURVE_BRACKETED : %left '{' %right '}'
ROUND_BRACKETED : %left '(' %right ')'
SQUARE_BRACKETED : %left ('['|GENERAL_ATTRIBUTE_START) %right ']'


namespace_content =    opening_directive*! (namespace|entity|general_attribute)*
opening_directive =    ('using'|'extern') Any ';'
namespace         =    'namespace' name '{' namespace_content '}'
entity            =    enum | class_struct_interface | method | field | property | water_entity
enum              =    common_beginning 'enum' name Any '{' Any '}' ';'?
class_struct_interface   =    common_beginning CLASS_STRUCT_INTERFACE name Any '{' entity* '}' ';'?
method            =    common_beginning type name arguments Any method_body
method_body       =    init_expression? ';' | block
field             =    common_beginning type name ('[' Any ']')? init_value? (',' name ('[' Any ']')? init_value?)* ';'
property          =    common_beginning type name property_body
property_body     =    (block (init_value ';')? | init_expression ';')
water_entity      =    AnyInclude('delegate', 'operator', 'this') (block | ';')+
common_beginning  =    entity_attribute* modifier*
modifier          =    MODIFIER | 'extern'
init_expression   =    '=>' Any
init_value        =    '=' init_part+
init_part         =    Any | type
name_atom         =    ID type_parameters?
name              =    name_atom (('.'|'::') name_atom)*
names_list        =    name (',' name)*
tuple             =    '(' type name? (',' type name?)* ')'
type_atom         =    ('unsigned'? ID | tuple) type_parameters? '?'? '*'*
type              =    type_atom ((('.'|'::') type_atom) | ('[' Any ']'))*!
type_parameters   =    '<' (AnyAvoid(';') | type_parameters)* '>'
entity_attribute  =    '[' Any ']'
general_attribute =    GENERAL_ATTRIBUTE_START Any ']'
arguments         =    '(' Any ')'
block             =    '{' Any '}'

%%

%parsing {
    recovery entity init_part
    fragment STRING_INT_CODE
    start namespace_content
    skip COMMENT STRING DIRECTIVE
}

%nodes {
    ghost names_list common_beginning entity
    leaf name type modifier arguments entity_attribute
}
```

Рис. 3.4. Упрощённая LR(1) грамматика подмножества языка C#.

Таблица 3.2. Результаты анализа проектов на C#.

Проект	Всего файлов	Перечисл.	Классы	Поля	Свойства	Методы
Roslyn	8759	482	23705	20265	23127 (2)	116312 (5)
PABC.NET	2802	359	5522	16739	12023	37027
ASP.NET Core	7356	333	12604	10214	16301	44163
EF Core	2997	101	7783	4687 (1)	16941 (2)	26421 (135)
Mono	37224	4928 (1)	60187 (122)	166958 (67)	99167 (36)	309580 (670)

- **ASP.NET Core** представляет другую предметную область: это кроссплатформенный фреймворк для веб-разработки;
- **Entity Framework Core** является фреймворком для организации взаимодействия с базами данных.

Как видно из таблицы 3.2, в проекте Roslyn присутствуют 5 методов, найденных полным парсером и пропущенных легковесным. 4 из них — это локальные методы (методы, описанные внутри других методов). Данная возможность языка являлась сравнительно новой на момент проведения эксперимента и ранее не встречалась при легковесном парсинге программ. Непосредственно в коде Roslyn она стала использоваться между текущим экспериментом и экспериментом из раздела 2.7. В случае необходимости добавить учёт данной категории методов можно путём ещё одной итерации уточнения грамматики, расширив определение нетерминала `block`. Пятый потерянный метод находится в тестовом файле, текст которого сохранён в кодировке Shift-JIS и содержит класс с именем на японском языке. Имя класса провоцирует ошибку, которая не влияет на обнаружение самого класса, но препятствует подробному анализу его содержимого. Данный файл вызывал аналогичную проблему и во время эксперимента, описанного в разделе 2.7. Мы полагаем, что национальные алфавиты нечасто используются для именования сущностей программы, но в случае необходимости символы нужных алфавитов могут быть добавлены в определение токена `ID`.

Легковесные парсеры пропускают в проекте Roslyn 2 свойства. В обоих случаях причиной является отсутствие выражения, задающего значение свойства, предшествующего пропущенному. В следующем коде нераспознанным оказывается свойство `IsWindows`:

```
public static ExecutionConfiguration Configuration =>
#if DEBUG
    ExecutionConfiguration.Debug;
#elif RELEASE
    ExecutionConfiguration.Release;
#else
    #error Unsupported Configuration
#endif
public static bool IsWindows =>
    Path.DirectorySeparatorChar == '\\';
```

Поскольку выражение для свойства `Configuration` зависит от символа условной компиляции, определённого во внешнем контексте, а в эксперименте ни один из ожидаемых символов не определён, используемый нами легковесный препроцессор исключает из текста программы оба варианта выражения для данного свойства. Из-за этого текст программы оказывается некорректным, и всё определение `IsWindows` распознаётся как недостающее выражение для `Configuration`. Данная проблема решается передачей препроцессору актуального символа условной компиляции.

В проектах PascalABC.NET и ASP.NET Core все сущности, найденные Roslyn, были также найдены LanD. Для проекта Entity Framework Core различие в количестве найденных полей и методов вызвано ситуацией с директивами условной компиляции, похожей на продемонстрированную в коде выше. Отличие в количестве свойств спровоцировано свойством, содержащим греческую букву в имени типа, что также отсылает нас к похожей проблеме, имевшей

Таблица 3.3. Результаты анализа проектов на C# (2).

Проект	Всего файлов	Перечисл.	Классы	Поля	Свойства	Методы
Roslyn	8759	482	23705	20265	23127	116312
PABC.NET	2802	359	5522	16739	12023	37027
ASP.NET Core	7356	333	12604	10214	16301	44163
EF Core	2997	101	7783	4687	16941	26421
Mono	37224	4928 (1)	60187 (21)	166958 (67)	99167 (5)	309580 (20)

место при разборе Roslyn.

Для Mono подавляющее большинство потерянных сущностей сконцентрировано в файлах, являющихся некорректными в терминах полной грамматики C#. Например, в 26 файлах отсутствует закрывающая парная директива для директивы условной компиляции или имеет место несоответствие количества и типов открывающих и закрывающих скобок. Половина из 122 пропущенных классов определяется в файлах с расширением .cs, содержащих код LINQ-to-SQL, записанный в соответствии с синтаксисом языка Visual Basic. Присутствуют файлы с расширением .cs, содержимое которых записано в специфическом формате, в частности, файл-шаблон для генератора парсеров jay, в котором каждая строка начинается с точки.

Оставим в таблице 3.2 только те несовпадения с полным парсингом, которые свидетельствуют о действительно проблемных ситуациях при разборе правильных программ. Получим таблицу 3.3. Mono остаётся единственным проектом, в котором есть группа пропущенных легковесным парсером сущностей. Эти сущности содержатся в тестовых файлах, именуемых как `test-async`[3] и `test- partial`[4]. В языке C# существуют ключевые слова `async` и `partial`, добавленные относительно недавно и реализованные как *контекстные*, чтобы сохранить работоспособность существующего кода. Контекстные ключевые слова по-прежнему могут быть использованы в качестве имён классов, методов и

[3] `mono/mcs/tests/test-async-*.cs`

[4] `mono/mcs/tests/test-partual-*.cs`

т. д. К примеру, следующий код является корректной программой на языке C# (тела методов опущены):

```
namespace async
{
    partial class async { partial void partial(); }
    partial class partial
    {
        // async method named 'async'
        async Task<async> async() { ... }
        // method named 'async' returning an object of type 'async'
        async async(async async) { ... }
    }
}
```

Правильная интерпретация контекстных ключевых слов невозможна без анализа контекста, в котором они находятся. Такой анализ выходит за рамки LL(1) и LR(1) разбора. В самом инструменте Roslyn, используемом для компиляции C# в IDE Visual Studio, проблема контекстных ключевых слов решается при помощи специального метода `ShouldAsyncBeTreatedAsModifier`, проверяющего большое количество специфических условий, каждое из которых покрывает конкретный случай расположения `async` относительно зарезервированных ключевых слов, предопределённых типов и ключевого слова `partial`. Кроме того, для принятия правильного решения компилятору требуется до двух дополнительных токенов из входного потока.

Отметим, что вероятность встретить контекстные ключевые слова, используемые в качестве идентификаторов невелика. В наших экспериментах такие случаи были обнаружены только в указанных синтетических тестовых файлах. Кроме того, использование контекстных ключевых слов `async` и `partial` в качестве идентификаторов публичных сущностей нарушает общие соглашения об

```
COMMENT   : '//' ~[\n\r]* | '/*' .*? '*/'
STRING    : '"' ('\\"'|'\\\\'|.)*? '"'
CHAR      : '\'' ('\\\''|'\\\\'|.)*? '\''
MODIFIER  : 'transient'|'strictfp'|'native'|'public'|'private'|'protected'|'static'|'final'
    |'synchronized'|'abstract'|'volatile'|'default'
ID        : [_$a-zA-Z][_$0-9a-zA-Z]*
CURVE_BRACKETED  : %left '{' %right '}'
ROUND_BRACKETED  : %left '(' %right ')'
SQUARE_BRACKETED : %left '[' %right ']'

file_content      =    entity*
entity            =    enum | class_interface | method | field_declaration | water_entity
enum              =    common_beginning 'enum' name Any block ';'?
class_interface   =    common_beginning ('class'|'interface') name Any '{' entity* '}' ';'?
method            =    common_beginning type name arguments Any (';' | block)
field_declaration =    common_beginning type field (',' field)* ';'
field             =    name init_value?
water_entity      =    AnyInclude('@interface', 'import', 'package') (block | ';')+
common_beginning  =    (annotation|MODIFIER)*
init_value        =    '=' init_part+
init_part         =    Any | type_parameter
name              =    name_type
type              =    name_type
name_type_atom    =    type_parameter? ID type_parameter?
name_type         =    name_type_atom ((('.'|'::') name_type_atom) | ['']')*
type_parameter    =    '<' (AnyAvoid(';') | type_parameter)* '>'
arguments         =    '(' Any ')'
annotation        =    '@' name arguments?
block             =    '{' Any '}'

%%

%parsing {
    recovery entity init_part
    start file_content
    skip COMMENT STRING
}

%nodes {
    ghost common_beginning entity
    leaf name type type_parameter annotation
}
```

Рис. 3.5. Упрощённая LR(1) грамматика подмножества языка Java.

именовании сущностей в языке C#[5], которые обычно используются как основа для более конкретных соглашений об оформления кода, применяемых в команде разработчиков.

3.4.2. Легковесный разбор программ на языке Java

Тестирование парсеров языка Java проводится на исходных кодах четырёх проектов:

- **Java Development Kit** представляет собой набор инструментов Java-разработчика и включает в себя компилятор языка Java, основные библиоте-

[5] https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/capitalization-conventions

Таблица 3.4. Результаты анализа проектов на Java.

Проект	Всего файлов	Перечисл.	Классы	Поля	Методы
JDK	7704	151	10590	46176 (3)	88709
Elastic	10972	387	14914	36830	94722
Spring	7063	100	12060	18402	61515
RxJava	1654	36	2728	6258	19931

ки и среду выполнения;

- **Elasticsearch** — библиотека для осуществления полнотекстового поиска;
- **Spring Framework** является универсальным Java-фреймворком, предназначенным для разработки приложений из различных предметных областей;
- **RxJava** — библиотека для построения асинхронных приложений, работа которых основана на обработке последовательностей событий.

Как следует из данных, представленных в таблице 3.4, существует единственное различие между результатами, полученными с использованием полного и легковесного парсеров: три поля пропущены легковесным парсером в проекте JDK. Это поля `FIND_MASK`, `NEW_MASK` и `RELEASE_MASK`, определяемые в следующем коде.

```
private final static int
    CREATE_MASK = 1<<CREATE,
    FIND_MASK = 1<<FIND,
    NEW_MASK = 1<<NEW,
    RELEASE_MASK = 1<<RELEASE,
    ALL_MASK = CREATE_MASK|FIND_MASK|NEW_MASK|RELEASE_MASK;
```

Их пропуск легковесным парсером вызван тем, что в инициализаторах используется символ `<`, который можно трактовать и как непарную конструкцию (операции сравнения, побитового сдвига), и как открывающую скобку для пара-

метров типа. В случае, если угловые скобки ограничивают параметры шаблонного типа, эти параметры важно распознать как единое целое, чтобы предотвратить трактовку внутренних запятых как разделителей полей. Правильно проинтерпретировать открывающую угловую скобку в рамках легковесного синтаксического анализа трудно, как и в рамках контекстно-свободного парсинга в целом [155]. Для устранения проблемы, найденной в JDK, в грамматике можно определить оператор `<<` как отдельный токен, однако похожая проблема может возникнуть, если в инициализаторах символ `<` будет использоваться как оператор сравнения.

3.4.3. Выводы

Результаты эксперимента свидетельствуют о том, что модифицированные нами алгоритмы LL(1) и LR(1) синтаксического анализа в совокупности с алгоритмами восстановления от ошибок позволяют успешно разбирать крупные промышленные проекты и находить все синтаксические сущности, описанные как «острова». Алгоритмы восстановления от ошибок позволяют провести успешный разбор программы не только на основе грамматики, упрощённой относительно «полной», но и на основе более простой легковесной грамматики с символом Any, упрощённой относительно грамматики, порождающей подмножество программ целевого языка.

Обнаруженные в ходе эксперимента расхождения между количествами «островов», найденных легковесным и полным парсерами, являются устранимыми либо вызваны не недостатками легковесного парсинга, а особенностями контекстно-свободного разбора как такового.

Отметим, что нетерминалы-точки восстановления одновременно являются своеобразными точками расширения грамматики: в грамматиках 3.4 и 3.5 альтернативы нетерминала `entity`, соответствующие ненужным «островам», могут быть удалены, при этом области программы, ранее распознаваемые как эти «острова», начнут распознаваться как «вода» с применением алгоритмов вос-

становления. Наоборот, чтобы добавить, например, выявление конструкторов в те же грамматики, достаточно определить новый нетерминал `constructor` и добавить его к альтернативам, образующим `entry`. В результате конструктор будет успешно распознаваться от начала до конца, а значит, на нём перестанет запускаться алгоритм восстановления.

Глава 4

Привязка к синтаксическим элементам программы

Существуют разные подходы к выбору и «запоминанию» признаков, по которым можно было бы найти некоторое место в коде. В ранее упомянутой работе А. Л. Фуксмана [11] требуется однозначно идентифицировать те точки основы, куда необходимо встроить фрагменты расширяющей функции. Эту идентификацию предполагается производить по текстовым координатам (номеру строки и столбца), а также при помощи поиска строк и подстрок. В более поздних парадигмах, также основанных на идее разделения базовой версии программы и добавляемых к ней расширений [7, 17, 19], места встраивания расширений предлагается идентифицировать по имени класса, имени метода, сигнатуре, возможно, с использованием шаблонов. Все упомянутые способы в значительной степени неустойчивы к редактированию кода и не могут быть применены для привязки в задаче разметки.

Ещё одним подходом к поиску нужного места в кодовой базе является обозначение этого места непосредственно в коде при помощи *псевдокомментариев* — комментариев специального вида, обрабатываемых компилятором языка так же, как обрабатываются обычные комментарии, но имеющих специальное значение для некоторого внешнего инструмента. Псевдокомментарии используются в различных системах разработки и сопровождения программного обеспечения [54–58], данный подход устойчив к любому редактированию кода, однако может перестать работать при случайном изменении самих псевдокомментариев. Важно и то, что псевдокомментарии «замусоривают» код информацией, имеющей, как правило, только технический смысл.

Описываемый в диссертации подход основан на идее представления запоминаемого участка в виде набора структур специального вида — *контекстов* —

и хранения данного набора-описания отдельно от кода. Содержащаяся в контекстах информация, описывающая элемент и его окружение, собирается по легковесному абстрактному синтаксическому дереву программы. Идея о привязке к коду на основе контекстов была впервые реализована в работах [59, 60], предложенные в этих работах модели и алгоритмы будем называть *базовыми*.

В настоящей главе предлагаются новые модели для описания синтаксического элемента программы, к которому необходимо произвести привязку. Описываются ориентированные на эти модели алгоритмы, необходимые для перепривязки этого элемента в изменившемся коде, анализируется сложность алгоритма перепривязки. Приводятся результаты эксперимента, подтверждающие, что привязка, выполняемая с помощью предложенных моделей и алгоритмов, является устойчивой: ранее помеченный элемент успешно находится в изменившемся коде почти в 100% случаев. Также демонстрируется повышение устойчивости привязки и количества автоматических перепривязок по сравнению с базовыми моделями и алгоритмами.

4.1. Базовые модели и алгоритмы

В работах [59, 60] рассматривается привязка к синтаксическим элементам программы. Элемент a, к которому осуществляется привязка, называется *точкой привязки*. Далее под a будем также понимать соответствующий этому элементу узел АСД.

Предполагается, что для привязки к a необходимо построить следующий упорядоченный набор (кортеж):

$$(Type_a, H_a, I_a, S_a, N_a, NearL_a, NearG_a)\,.$$

Здесь $Type_a$ — тип элемента (класс, метод, поле, свойство и т. п.). H_a — *контекст заголовка*, хранимый как список строк (например, для метода он хранит как отдельные строки все его модификаторы, возвращаемый тип, имя и

аргументы). Для контекста заголовка определяется дополнительный оператор $\mathrm{Name}(H_a)$, позволяющий отдельно получить имя элемента a.

I_a — *внутренний контекст*, хранимый как множество из не более чем $n_I \in \mathbb{Z}^{\geq 0}$ пар вида $(Type, H)$, соответствующих вложенным в a элементам (например, внутренний контекст для класса образуют члены класса).

S_a — *контекст областей* (в упомянутых работах он назван *внешним контекстом*), хранимый как список пар $(Type, H)$. Каждая пара описывает сущность, объемлющую помечаемую (для метода контекст областей обычно содержит два элемента, один из которых соответствует объемлющему классу, а второй — пространству имён).

N_a — *контекст соседей* или *горизонтальный контекст*, состоящий из двух множеств пар вида $(Type, H)$. Каждое множество хранит информацию о не более чем $n_N \in \mathbb{Z}^{\geq 0}$ сущностях. Первое хранит информацию о предшествующих a элементах, вложенных в наименьший общий с a объемлющий элемент. Второе хранит информацию об аналогичных следующих за a элементах (например, для метода контекст соседей описывает ограниченное количество соседних членов того же класса).

Числа $NearL_a$ и $NearG_a$ находятся в диапазоне $[0; 1]$. $NearL_a$ — минимальное редакционное расстояние [156], которое можно получить, сравнивая H_a с заголовками соседей a, имеющих тип $Type_a$. Для $NearG_a$ сравнение H_a происходит со всеми сущностями типа $Type_a$, кроме соседей.

Перепривязка производится по отдельности для каждой точки. В актуальной версии программы выбираются все синтаксические элементы типа $Type_a$ (будем называть их *кандидатами*). Если $NearL_a > 0$ и $NearG_a > 0$, и в текущей версии программы есть единственный элемент-кандидат c, для которого $H_a = H_c$, этот кандидат c считается текущей версией a и происходит перепривязка a к c. В противном случае для a и каждого кандидата c вычисляется

вектор поконтекстных расстояний вида

$$\mathbf{d_{ac}} = (\,\mathrm{Dist}\,(\mathrm{Name}(H_a), \mathrm{Name}(H_c))\,, \mathrm{Dist}\,(H_a, H_c)\,, \mathrm{Dist}\,(I_a, I_c)\,,$$
$$\mathrm{Dist}\,(S_a, S_c)\,, \mathrm{Dist}\,(N_a, N_c)).$$

Для $\mathrm{Name}(H_a)$ и $\mathrm{Name}(H_c)$, H_a и H_c, S_a и S_c функция вычисления расстояния *Dist* считает нормированное к диапазону $[0; 1]$ расстояние Левенштейна [156]. Для I_a и I_c, N_a и N_c расстоянием является нормированная к диапазону $[0; 1]$ мощность разности соответствующих множеств. Итоговое расстояние $\mathrm{Dist}\,(a, c)$ вычисляется как нормированное скалярное произведение вектора $\mathbf{d_{ac}}$ и вектора весов $\mathbf{w}$, имеющего вид

$$\left(w^{\mathrm{Name}(H)}, w^H, w^I, w^S, w^N\right).$$

Веса — это эмпирически подобранные числа, отражающие важность совпадения каждого из контекстов при поиске актуальной версии точки привязки.

Затем кандидаты сортируются по возрастанию итогового расстояния. Пусть c_1 — кандидат, находящийся на наименьшем расстоянии от a, c_2 — следующий за ним кандидат. Кандидат c_1 признаётся актуальной версией a, если $\mathrm{Dist}\,(a, c_1) \cdot 2 \leq \mathrm{Dist}\,(a, c_2)$. В этом случае кортеж, описывающий c_1, заменяет собой ранее сохранённое описание a, и говорится, что элемент *a переприв язан* к c_1 или *a сопоставлен* с c_1. В противном случае требуется человеческое вмешательство: правильного кандидата нужно вручную выбрать из отсортированного списка.

Хотя зачастую базового подхода достаточно для того, чтобы поставить на первое место в списке именно того кандидата, к которому нужно провести переприв язку, существует значительное число сложных случаев редактирования, когда базового подхода недостаточно для автоматической переприв язки, поскольку он не может в достаточной степени различить верного кандидата и всех остальных. Кроме того, в некоторых случаях происходит ошибка: на пер-

вое место в списке попадает не тот кандидат, который на самом деле является текущей версией искомой точки привязки. В настоящей работе предлагаются новые модели контекстов, позволяющие более подробно описать точку привязки и её окружение, и новый алгоритм перепривязки, осуществляющий более точное её сравнение с кандидатами. Применение предлагаемых в работе моделей и алгоримов позволяет понизить количество ошибок и существенно повысить количество автоматических перепривязок.

4.2. Новые модели контекстов

Построение предлагаемых нами моделей во многом опирается на дополнительную информацию, которой атрибутированы узлы АСД. Эта информация попадает в узлы дерева из грамматики, написанной для генератора LanD. На рисунке 4.1а приведён упрощённый фрагмент упрощённой грамматики языка C#. В данном фрагменте определён нетерминальный символ `method`, соответствующий методу класса. Ниже показан участок секции опций. Встроенные опции генератора LanD из группы `nodes` позволяют управлять построением абстрактного синтаксического дерева. Опция `leaf`, указанная для нетерминалов `type`, `name` и `arguments`, означает, что соответствующие им узлы дерева должны быть листовыми. Группа опций `markup` — «пользовательская» группа. Она не интерпретируется самим LanD, но входящие в неё опции связываются с узлами АСД, соответствующими перечисленным для этих опций символам, и затем обрабатываются алгоритмами привязки и перепривязки, использующими это АСД. Например, нетерминальные символы, соответствующие синтаксическим элементам, к которым можно произвести привязку, помечаются опцией `land`. Прочие опции из данной группы будут разъяснены далее.

Элемент a, к которому осуществляется привязка, (точку привязки) будем описывать в виде кортежа, который обозначим как $\mathbf{BindingPoint_a}$:

$$\mathbf{BindingPoint_a} = (\mathit{Type}_a, H_a, I_a, S_a, N_a, C_a) \ .$$

```
method : MODIFIER* type name arguments block
block  : '{' Any '}'
...
%nodes {
  leaf type name arguments
}
%markup {
  land namespace class method field property
  exactMatch MODIFIER
  priority(0.1) MODIFIER
  priority(0.3) type
  priority(0.5) arguments
  headerCore(name) namespace class method field property
}
```

(a)

```
public static int? IncreaseCounter(int? counter) {...}

((MODIFIER,  0.1, ExactMatch, ((1,"public"))),
 (MODIFIER,  0.1, ExactMatch, ((1, "static"))),
 (type,      0.3, Distance, ((1,"int"), (0.1, "?"))),
 (name,      1.0, Distance, ((1,"Increase"), (1, "Counter"))),
 (arguments, 0.5, Distance, ((0.1,"("), (1,"int"), (0.1, "?"),
                            (1,"counter"), (0.1,")"))))
```

(b)

Рис. 4.1. (a) Фрагмент упрощённой грамматики языка C#; (b) пример контекста заголовка.

Далее для компоненты кортежа будем опускать нижний индекс, обозначающий конкретный элемент, к которому произведена привязка, если знание конкретного элемента несущественно для проводимого рассуждения.

$Type_a$ — это символ грамматики, соответствующий a (например, `namespace`, `class`, `method`, `field` или `property` в случае привязки к синтаксическим элементам программы на языке C#).

H_a — *контекст заголовка*, хранимый как список элементов-четвёрок вида

$$(Type, Priority, ComparisonMode, Words)\ ,$$

по одной для каждого листового непосредственного потомка узла a. На рисунке 4.1b приведён пример метода и контекста заголовка, построенного для этого метода. В каждой четвёрке компонента $Type$ — терминальный или нетерминальный символ, соответствующий элементу заголовка. $Priority \in \mathbb{R}^{\geq 0}$ отражает важность совпадения элемента при сравнении заголовков, $ComparisonMode \in \{Distance,\ ExactMatch\}$ определяет, как проводить сравнение двух таких элементов — путём вычисления редакционного расстояния или проверки строгого совпадения. По умолчанию $Priority$ принимает значение 1, $ComparisonMode$ принимает значение $Distance$, обе компоненты могут быть заданы для некоторого типа $Type$ с помощью опций из группы `markup`, как показано на рисунке 4.1a.

В приведённом примере приоритеты аргументов и возвращаемого типа устанавливаются более высокими, чем приоритет модификаторов. Режим сравнения $ExactMatch$ устанавливается для модификаторов, поскольку они могут принимать конечное число значений и любое несовпадение означает, что перед

нами полностью другой модификатор с совершенно другим смыслом. Последний элемент четвёрки, список *Words*, конструируется путём разделения текста, соответствующего элементу заголовка, на отдельные слова и нечислобуквенные последовательности. Элементом списка является пара (*Priority*, *Text*), где *Priority* равен 0.1 для нечислобуквенного *Text* и 1 для числобуквенного. *Text* является строкой.

Дополнительно контекст заголовка разделяется на «ядро» и «не-ядро». Ядро заголовка — один или более элементов заголовка, которые считаются наиболее важными его частями. Опция `headerCore` из группы `markup` принимает в качестве аргументов (аргументы опции указываются в скобках после её имени) типы элементов, формирующих ядро заголовка. После аргументов перечислены типы элементов, для которых задаётся данное ядро. Оператор Core(H_a) возвращает список элементов из H_a, являющихся ядром заголовка. NotCore(H_a) возвращает H_a за вычетом ядра. На рисунке 4.1а `name` указан в качестве ядра для всех синтаксических сущностей, к которым допускается привязка. Оператор Name(H_a), используемый в [60], можно считать частным случаем Core(H_a).

Как было показано в разделе 4.1, предлагаемая в [59,60] модель контекста заголовка является чрезвычайно простой и не позволяет провести какой-либо глубокий анализ при сравнении двух заголовков. В предлагаемой нами модели элементы H_a типизированы, благодаря чему в процесс перепривязки может быть привнесён реальный опыт разработки промышленных программ. Приоритеты, режимы сравнения и выделение ядра фиксируют некоторое представление о том, на что сами программисты обращают внимание в первую очередь, когда производят поиск элемента в коде, и о том, как они производят сравнение элементов.

I_a — *внутренний контекст*, хранимый как тройка (*Text*, *Hash*, *Length*). Далее тройки данного вида будем называть **TextOrHash**. Компонента *Text* хранит конкатенированный текст всех нелистовых непосредственных потомков узла a. Если этот текст оказывается слишком длинным, вместо него в компо-

ненте *Hash* сохраняется его нечёткий хеш, а компонента *Text* остаётся пустой. Длина текста всегда сохраняется в *Length*. В качестве алгоритма нечёткого хеширования в настоящей работе используется алгоритм TLSH [157], поскольку, в отличие от других известных алгоритмов [158, 159], он применим как для длинных, так и для достаточно коротких текстов и не накладывает дополнительных ограничений на сравнение хешей, построенных для текстов разной длины.

Нечёткое хеширование позволяет преодолеть главный недостаток внутреннего контекста из [60]: больше не требуется искать компромисс между покрытием всего содержимого синтаксической сущности и скоростью сравнения контекстов. В предлагаемом нами I_a всегда учитывается всё, что расположено внутри a. Также можно отметить, что в [60] складывается очевидно некорректная ситуация: поскольку тело метода не имеет заголовка, оно не рассматривается как часть внутреннего контекста, и внутренний контекст метода оказывается пустым.

S_a — *контекст областей*, хранимый как список пар вида $(Type, H)$, построенных для всех предков узла a, с типами которых связана опция `land`. Каждая пара хранит тип и контекст заголовка некоторого предка.

N_a — *контекст соседей*, хранимый как пара $(Before, After)$, где каждая компонента также является парой вида $(All, Nearest)$. *Before* описывает синтаксические элементы, предшествующие a; *After* описывает элементы, следующие за a. *All* — это тройка **TextOrHash**, построенная для конкатенированного текста всех предшествующих (в *Before*) или последующих (в *After*) элементов, имеющих общего с узлом a родителя (непосредственного предка). *Nearest* — это тройка **BindingPoint**, построенная для ближайшего к a синтаксического элемента типа $Type_a$ в пределах того же файла. Этот элемент может не иметь общего с a родительского узла, от него требуется только близость расположения в тексте программы. Для самого *Nearest* контекст соседей не строится, чтобы избежать привязки к абсолютно всему содержимому файла.

Контекст соседей имеет решающее значение в ситуации, когда остальных

контекстов недостаточно для успешной перепривязки. В языках C# или Java предпочтительно принимать в расчёт только компоненту *Nearest*, поскольку содержимое компоненты *All* может различаться для смежных элементов в большом файле несущественно, а ближайшие соседи всегда уникальны. В то же время, для языков, в которых в одной области могут находиться полностью совпадающие элементы типа $Type_a$ (например, в HTML, где могут быть полностью совпадающие теги, или в языках описания грамматик, где один и тот же символ может несколько раз встретиться в одной альтернативе), хеширование всего предшествующего и последующего при привязке может быть единственной возможностью провести различие между кандидатами. Компонента *All* в N_a непуста, если для типа $Type_a$ в грамматике указана опция `notUnique`. Компонента *Nearest* конструируется всегда.

Далее будем использовать операторы Before(N_a), After(N_a) и т. п., возвращающие компоненту аргумента, имя которой совпадает с именем оператора. Также будем использовать оператор All(N_a), возвращающий пару (All(Before(N_a)), All(After(N_a))), и аналогично действующий оператор Nearest(N_a).

C_a — *контекст наиболее похожих.* Он хранится как список точек привязки **BindingPoint**, построенных для элементов типа $Type_a$, наиболее похожих на a в момент привязки к нему. Например, когда a является одной из реализаций перегруженного метода, в C_a сохраняются описания других реализаций этого метода. Наиболее похожие элементы определяются с использованием алгоритмов, приводимых далее в настоящей главе. C_a не строится для элементов из контекста соседей и из самого контекста наиболее похожих.

4.3. Новый алгоритм перепривязки

4.3.1. Сравнение контекстов

Чтобы перепривязать a в случае, если программа была отредактирована, необходимо сравнить описывающую a структуру **BindingPoint** с такими же

описаниями, построенными для элементов типа $Type_a$, имеющихся в текущей версии программы (кандидатов). Пусть b — один из кандидатов. В процессе перепривязки вычисляется вектор поконтекстных расстояний между a и b:

$$\begin{aligned}\mathbf{d_{ab}} =&(\mathrm{Dist}(\mathrm{Core}(H_a), \mathrm{Core}(H_b)), \mathrm{Dist}(\mathrm{NotCore}(H_a), \mathrm{NotCore}(H_b)),\\ &\mathrm{Dist}(I_a, I_b), \mathrm{Dist}(S_a, S_b), \mathrm{Dist}(\mathrm{All}(N_a), \mathrm{All}(N_b)),\\ &\mathrm{Dist}(\mathrm{Nearest}(N_a), \mathrm{Nearest}(N_b))).\end{aligned}$$

Далее в настоящем разделе наряду с функцией $\mathrm{Dist}(x, y)$, оценивающей расстояние между аргументами, будем использовать функцию оценки похожести $\mathrm{Sim}(x, y)$. Поскольку расстояния всегда нормируются нами к диапазону $[0; 1]$, функции связаны друг с другом тривиальным образом: $\mathrm{Sim}(x, y) = 1 - \mathrm{Dist}(x, y)$.

Контекст заголовка

Обозначим элементы, содержащиеся в контекстах заголовков H_a и H_b, как e^{H_a} и e^{H_b} соответственно. В качестве нижнего индекса элемента в случае необходимости будем указывать его индекс в списке элементов контекста. Индексирование производим с единицы. Через $\mathrm{Priority}(x)$, $\mathrm{Type}(x)$ и т. д. обозначим операторы, возвращающие тот элемент кортежа-аргумента, имя которого совпадает с оператором.

Для списков $\mathrm{Core}(H)$ и $\mathrm{NotCore}(H)$ расстояния вычисляются как расстояния Левенштейна при помощи модифицированного нами алгоритма Вагнера-Фишера [160]. В модифицированном алгоритме элемент матрицы расстояний

вычисляется по формуле

$$D_{i,j} = \begin{cases} 0, \quad i = 0, j = 0 \\ D_{i-1,j} + \text{Priority}\left(e_i^{H_a}\right),\ j = 0, i > 0 \\ D_{i,j-1} + \text{Priority}\left(e_j^{H_b}\right),\ i = 0, j > 0 \\ \text{Min}(D_{i,j-1} + \text{Priority}\left(e_j^{H_b}\right), D_{i-1,j} + \text{Priority}\left(e_i^{H_a}\right), \\ \qquad D_{i-1,j-1} + \text{Priority}\left(e_i^{H_a}\right) \cdot \text{Dist}\left(e_i^{H_a}, e_j^{H_b}\right)), \quad \text{иначе.} \end{cases} \tag{4.1}$$

Нетрудно заметить, что вес операций удаления, добавления и замены зависит от приоритета соответствующего элемента заголовка. Расстояние между e^{H_a} и e^{H_b} вычисляется как

$$\text{Dist}(e^{H_a}, e^{H_b}) = \begin{cases} +\infty, \quad \text{Type}\left(e^{H_a}\right) \neq \text{Type}\left(e^{H_b}\right) \\ 0, \quad \text{ComparisonMode}\left(e^{H_a}\right) = \textit{ExactMatch}, \\ \qquad \text{Words}\left(e^{H_a}\right) = \text{Words}\left(e^{H_b}\right) \\ 1, \quad \text{ComparisonMode}\left(e^{H_a}\right) = \textit{ExactMatch}, \\ \qquad \text{Words}\left(e^{H_a}\right) \neq \text{Words}\left(e^{H_b}\right) \\ \text{Dist}\left(\text{Words}\left(e^{H_a}\right), \text{Words}\left(e^{H_b}\right)\right), \quad \text{иначе}. \end{cases}$$

Расстояние между списками *Words* также вычисляется как расстояние Левенштейна. Элементы матрицы расстояний определяются по формуле, аналогичной формуле (4.1). Для двух элементов w_1 и w_2 из списков *Words* расстояние вычисляется как

$$\text{Dist}(w_1, w_2) = \begin{cases} +\infty, \quad \text{Priority}\left(w_1\right) \neq \text{Priority}\left(w_2\right) \\ 1 - \theta\left(\text{Sim}\left(\text{Text}\left(w_1\right), \text{Text}\left(w_2\right)\right), 0.5\right), \quad \text{иначе}. \end{cases}$$

Здесь θ — пороговая функция, возвращающая первый аргумент, если его значение больше или равно значению второго аргумента, в противном случае возвра-

щается ноль. Смысл записанного состоит в том, что расстояние Левенштейна, вычисляемое при сравнении компонент *Text*, округляется до 1, если превышает 0.5: мы считаем, что слова, различающиеся более чем наполовину, являются совершенно разными словами.

Внутренний контекст

Чтобы сравнить внутренние контексты I_a и I_b, для двух кортежей u и v типа **TextOrHash** определяется функция похожести:

$$\mathrm{Sim}\left(u,v\right)=\begin{cases}\theta\left(1-\mathrm{Dist}\left(\mathrm{Text}\left(u\right),\mathrm{Text}\left(v\right)\right),\dfrac{L_1}{L_2}\right),\\ \quad\forall k\in\{u,v\},\mathrm{Length}\left(k\right)\leq L_2\\ \theta\left(1-\mathrm{TlshDist}\left(\mathrm{Hash}\left(u\right),\mathrm{Hash}\left(v\right)\right),\dfrac{L_1}{L_2}\right),\\ \quad\forall k\in\{u,v\},\mathrm{Length}\left(k\right)\geq L_1\\ 0,\quad\text{иначе,}\end{cases}\tag{4.2}$$

Здесь функция TlshDist возвращает расстояние между нечёткими хешами, вычисленное библиотекой TLSH и нормированное к диапазону $[0;1]$ способом, предложенным в [161]. L_1 — минимальная длина текста, для которого может быть сконструирован нечёткий хеш. L_2 — максимальная длина текста, непосредственно сохраняемого в **TextOrHash**. Для текстов длиной от L_1 включительно до L_2 включительно **TextOrHash** хранит и текст, и хеш. Заметим, что u и v, для которых $\mathrm{Length}(u) > L_2$ и $\mathrm{Length}(v) < L_1$, несравнимы между собой, поскольку u содержит только хеш и v содержит только текст. Очевидно, что максимально возможная похожесть таких внутренних контекстов не может превышать максимально возможное соотношение длин соответствующих им текстов. Это соотношение равно L_1/L_2. Зафиксировав минимально возможное значение L_1 и выбрав достаточно большое значение L_2, мы можем игнорировать все похожести, меньшие чем L_1/L_2, считая их равными нулю. В экспериментах, опи-

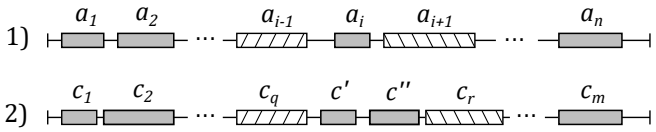


Рис. 4.2. Помеченные фрагменты в исходном тексте программы и кандидаты в текущей версии программы.

санных в разделе 4.6, $L_1 = 25$ в соответствии с техническими ограничениями TLSH, и $L_2 = 100$.

Контекст областей

Расстояние между контекстами областей S_a и S_b также вычисляется как редакционное посредством алгоритма Вагнера-Фишера. Если элементы e^{S_a} и e^{S_b} имеют разный тип, $\mathrm{Dist}(e^{S_a}, e^{S_b})$ равно $+\infty$. В противном случае оно равно расстоянию между компонентами H этих элементов.

Контекст соседей

В случае, если в контекстах соседей непусты компоненты *All*, тройки **TextOrHash** сравниваются по формуле (4.2) и $\mathrm{Dist}(\mathrm{All}(N_a), \mathrm{All}(N_b))$ вычисляется по формуле

$$\mathrm{Dist}(\mathrm{All}(N_a), \mathrm{All}(N_b)) = \frac{\mathrm{Dist}(\mathrm{All}\left(\mathrm{Before}(N_a)\right), \mathrm{All}\left(\mathrm{Before}(N_b)\right)) + \mathrm{Dist}(\mathrm{All}\left(\mathrm{After}(N_a)\right), \mathrm{All}\left(\mathrm{After}(N_b)\right))}{2}.$$

Сравнение компонент *Nearest* является более сложным. На рисунке 4.2 продемонстрирована типичная ситуация при перепривязке. Отрезок 1 обозначает исходный текст программы, прямоугольники обозначают фрагменты кода, соответствующие помеченным элементам одного типа, a_i — элемент, перепривязку которого рассматриваем в данный момент. Очевидно, что $\mathrm{Nearest}(\mathrm{Before}(N_a)) = a_{i-1}$ и $\mathrm{Nearest}(\mathrm{After}(N_a)) = a_{i+1}$. Отрезок 2 обозначает отредактированную программу, прямоугольники соответствуют кандидатам для перепривязки. Одинаковой штриховкой обозначены успешно перепривязанные элементы: их исходные версии и соответствующие им кандидаты.

Исходя из собственного опыта программирования и анализа репозиториев проектов с открытым исходным кодом, мы считаем, что взаимное расположение элементов программы меняется редко. Это означает, что после успешной перепривязки a_{i-1} к некоторому кандидату c_q мы с большей вероятностью найдем текущую версию a_i в области после c_q. После того, как a_{i+1} перепривязано к некоторому c_r, область сужается с противоположного конца, и c' и c'' становятся наиболее вероятными претендентами на роль текущей версии a_i. В соответствии с этой логикой, расстояния $\mathrm{Dist}(\mathrm{Nearest}(N_{a_i}), \mathrm{Nearest}(N_{c'}))$ и $\mathrm{Dist}(\mathrm{Nearest}(N_{a_i}), \mathrm{Nearest}(N_{c''}))$ должны быть наименьшими расстояниями среди кандидатов.

Ясно, что непосредственные соседи — это минимально необходимая информация для вычисления содержательной оценки. В реальной ситуации естественным образом накапливается избыточная информация, которая тоже может быть использована: чем больше точек привязки имеется в файле, тем больше сущностей описано как **BindingPoint**. Перепривязка каждой из них сужает наиболее вероятную область поиска для остальных.

Пусть n — общее количество точек привязки в файле, a_k и a_l — ближайшие успешно перепривязанные сущности (сущность, предшествующая a_i, и сущность, следующая за ним, соответственно). Кандидаты $c_{k'}$ и $c_{l'}$ признаны текущими версиями a_k и a_l. В ситуации, когда ни одной успешной перепривязки в пределах файла ещё не проведено, $k = 0$ и $l = n + 1$. Расстояние $\mathrm{Dist}(\mathrm{Nearest}(N_{a_i}), \mathrm{Nearest}(N_{c_j}))$ вычисляется следующим образом:

$$\mathrm{Dist}\left(\mathrm{Nearest}(N_{a_i}), \mathrm{Nearest}(N_{c_j})\right) = \begin{cases} 0, j > l' \vee j < k' \\ \frac{k+n-l+1}{n-1}, \text{ иначе}\,. \end{cases}$$

Если a_i является единственной точкой привязки в файле, расстояние между компонентами Nearest для a_i и некоторого кандидата не определено. Ещё раз отметим, что все успешно перепривязанные точки одного и того же типа в одном

и том же файле влияют на величину данного расстояния.

4.3.2. Основной алгоритм

Метод перепривязки реализован в виде алгоритма 5. Чтобы перепривязать все точки одного типа, присутствующие в одном файле, вызывается функция Rebind (строка 20). Список точек передаётся в параметре *points*, в параметре *candidates* передаётся список структур **BindingPoint**, сконструированных для кандидатов, присутствующих в текущей версии файла. Перепривязка выполняется одновременно как для точек, созданных пользователем, так и для вспомогательных точек, хранящихся в N и C.

На первом этапе происходит отсечение простых случаев, в которых для успешной перепривязки необязательно выполнять все действия, предусмотренные «большим» алгоритмом Rebind. Для этого для каждой перепривязываемой точки вызывается функция SimpleRebind (строки 23–28, строка 3). Как показано в разделах 4.4 и 4.6, SimpleRebind позволяет существенно сократить размерность задачи, отсекая большинство точек и кандидатов, даже если привязка была осуществлена очень давно и с тех пор над программой велась интенсивная работа.

SimpleRebind успешно находит точку a, если однозначно идентифицирующие a части не были отредактированы: если в момент привязки a обладала уникальным сочетанием некоторых описывающих её контекстов и в текущей версии программы есть кандидат с точно таким же содержимым данных контекстов. Предикаты $\text{AreEqual}^{\text{Core}(H)}$, $\text{AreEqual}^{\text{NotCore}(H)}$, AreEqual^{I}, AreEqual^{S} в функции SimpleRebind проверяют строгое совпадение для соответственных контекстов двух структур **BindingPoint**. Для каждого предиката наименование проверяемого контекста указано в качестве верхнего индекса. Из этих предикатов строится наименее строгая проверка, позволяющая отличить a от других существовавших в момент привязки сущностей, похожих на a (строки 5–11). Гарантируется, что при наличии сущностей, контексты которых в момент при-

Алгоритм 5 Алгоритм перепривязки.

```
1:  function CheckGap(v_1, v_2):
2:      return v_2 ≠ 0 ∧ v_1 · 2 ≤ v_2;
3:  function SimpleRebind(a, candidates = {c_1,c_2,...c_m}):
4:      predicates := [ AreEqual^{Core(H)}, AreEqual^{NotCore(H)}, AreEqual^{I} ];
5:      startIdx := Length(Core(H_a)) > 0 ? 1 : Length(H_a) > 0 ? 2 : Length(I_a) > 0 ? 3 : 0; idx := 0;
6:      if startIdx = 0 then return null;
7:      for (i from startIdx to Length(predicates)) do
8:          if ¬ (∃ k ∈ [1..Length(C_a)] : ∀ l ∈ [1..i], predicates[l](a, C_a[k]) ∧ AreEqual^{S}(a, C_a[k])) then
9:              idx := i;
10:             break;
11:     if idx = 0 then return null;
12:     for (i from idx to Length(predicates)) do
13:         if ∃! k ∈ [1..m] : ∀ l ∈ [1..i], predicates[l](a, c_k) ∧ AreEqual^{S}(a, c_k) then
14:             return c_k;
15:     return null;
16: function GetTotalDistances(a, candidates, distances, weights):
17:     for all c ∈ candidates do
18:         result[c] := distances[a][c] · weights[a] / ∑_{w ∈ weights[a]} w ;
19:     return result;
20: function Rebind(points = {a_1,a_2,...a_n}, candidates = {c_1,c_2,...c_m}):
21:     allPoints := ∪_{i∈[1..n]}(C_{a_i} ∪ {Nearest(Before(S_{a_i})), Nearest(After(S_{a_i}))}) ∪ points;
22:     unmatched := allPoints;
23:     for all p ∈ unmatched do
24:         match := SimpleRebind(p, candidates);
25:         if match ≠ null then
26:             unmatched := unmatched \ {p};
27:             candidates := candidates \ {match};
28:             if p ∈ points then autoResults[p] := {match};
29:     for all p ∈ unmatched do
30:         for all c ∈ candidates do
31:             distances[p][c] := GetDistances(p, c);
32:     oldResultLength := 0;
33:     do
34:         oldResultLength := Length(autoResults);
35:         for all p ∈ unmatched do
36:             weights[p] := GetWeights(p, candidates, distances);
37:             totalDistances[p] := GetTotalDistances(p, candidates, distances, weights);
38:             ordered[p] := OrderAsc(candidates, totalDistances[p]);
39:         for all p ∈ unmatched do
40:             c := ordered[p][1];
41:             bestDist := totalDistances[p][c];
42:             otherBestDist := Min_{p' ∈ unmatched \ {p}}(totalDistances[p][c])
43:             if bestDist ≤ MaxDist ∧ (Length(candidates) = 1
44:                 ∨ CheckGap(bestDist, totalDistances[p][ordered[p][2]]))
45:                 ∧ (otherBestDist = null ∨ CheckGap(bestDist, otherBestDist)) then
46:                 unmatched := unmatched \ {p};
47:                 candidates := candidates \ {c};
48:                 if p ∈ points then
49:                     autoResults[p] := ordered[p];
50:         for all p ∈ unmatched do
51:             UpdateNearestNeighboursDistances(p, candidates, distances[p]);
52:     while oldResultLength ≠ Length(autoResults);
53:     for all p ∈ unmatched do
54:         interactiveResults[p] := OrderAsc(candidates, totalDistances[p]);
55:     return (autoResults, interactiveResults);
```

вязки совпадают с контекстами a, наиболее похожие на a из этих сущностей представлены в C_a. Далее сконструированная проверка используется для фильтрации кандидатов. Если есть единственный кандидат c_k, который проходит фильтрацию, a переприязывается к c_k (строки 13–14). Если таких кандидатов несколько, к проверке привлекаются дополнительные предикаты.

После выполнения SimpleRebind неперепривязанные точки и оставшиеся кандидаты анализируются более тщательно. Для каждой неперепривязанной точки p и каждого кандидата c по формулам из раздела 4.3.1 строятся векторы поконтекстных расстояний $\mathbf{d_{pc}}$. Эти векторы запоминаются в массиве *distances* (строки 29–31). Затем запускается основной цикл алгоритма (строки 33–52). На каждой его итерации происходит несколько проходов по неперепривязанным точкам. Сначала для каждой p пересчитываются вектор весов вида

$$\mathbf{w_p} = (w_p^{\mathrm{Core}(H)}, w_p^{\mathrm{NotCore}(H)}, w_p^I, w_p^S, w_p^{\mathrm{All}(N)}, w_p^{\mathrm{Nearest}(N)})$$

и итоговое расстояние до каждого кандидата c. Веса сохраняются в массиве *weights*, итоговые расстояния — в массиве *totalDistances*, списки кандидатов сортируются по возрастанию расстояния и сохраняются в массиве *ordered* (строки 35–38). Вектор весов $\mathbf{w_p}$ — это вектор неотрицательных вещественных чисел, каждое из которых отражает важность совпадения некоторого контекста при перепривязке. Подробнее о его вычислении рассказано в разделе 4.3.3. Итоговое расстояние от точки p до кандидата c определяется по формуле

$$\mathrm{Dist}(p, c) = \frac{\mathbf{d_{pc}} \cdot \mathbf{w_p}}{\sum\limits_{w \in \mathbf{w_p}} w}.$$

Во время второго прохода по точкам (строки 39–49) для каждой точки p выбирается c_1 — первый кандидат в отсортированном списке — и проверяется, что c_1 достаточно похож на p и непохож на остальные точки, чтобы к нему можно произвести перепривязку. Требуется, чтобы расстояние от p до c_1 бы-

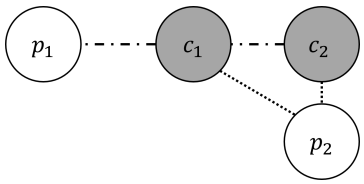


Рис. 4.3. Возможная ситуация при перепривязке.

ло меньше либо равно значению параметра *MaxDist* (строка 43), в противном случае считается, что c_1 слишком непохож на p и не может быть текущей его версией. Также требуется, чтобы расстояние от p до c_1 было намного меньше, чем расстояние между p и другими кандидатами (строка 44), и не существовала другая точка привязки p', для которой расстояние до c_1 было бы меньше, равно или незначительно больше, чем расстояние от p до c_1 (строка 45). Для проверки разницы расстояний используется функция CheckGap (строка 1).

Если все проверки пройдены, p перепривязывается к c_1 (строки 46–49). Это приводит к уменьшению количества рассматриваемых точек и кандидатов, в результате чего для некоторых неперепривязанных точек проверки в строках 43–45 могут начать проходить успешно. На рисунке 4.3 упрощённо продемонстрирована такая ситуация. p_1 — некоторая точка привязки, которую не удалось перепривязать из-за того, что наиболее близкий к ней кандидат c_1 недостаточно отстоит от следующего по похожести кандидата c_2. Однако p_2 однозначно перепривязывается к c_2. После этого кандидат c_2 вычёркивается из списка доступных, и новая попытка перепривязки p_1 на следующей итерации цикла **do − while** будет успешной.

В конце итерации цикла **do − while** для каждой неперепривязанной точки p и кандидата c пересчитывается $\mathrm{Dist}(\mathrm{Nearest}(N_p), \mathrm{Nearest}(N_c))$ (строка 51), поскольку успешные перепривязки могли сузить для p участок файла, в котором её актуальная версия присутствует с наибольшей вероятностью.

Функция Rebind возвращает пару словарей. В каждом из них ключом является исходная точка привязки, значением — список кандидатов, ранжированных по возрастанию расстояния до этой точки. В *autoResults* передаются

успешно перепривязанные элементы, первый элемент списка кандидатов — то, к чему произошла перепривязка. В *interactiveResults* представлены точки привязки, которые не удалось перепривязать автоматически. Для каждой из них правильное соответствие нужно выбрать вручную.

Отметим, что функция CheckGap используется и в момент изначальной привязки к точке при построении контекста наиболее похожих. Все имеющиеся в файле сущности того же типа, что и элемент, к которому осуществляется привязка, ранжируются по убыванию похожести на этот элемент и добавляются в контекст до тех пор, пока для очередного и предыдущего добавленного элемента не выполнятся CheckGap. Также заметим, что сценарий, при котором некоторая точка привязки переносится из одного файла проекта в другой, не рассматривается в настоящей работе.

4.3.3. Эвристики

В работах [59, 60] предполагается, что для контекстов можно зафиксировать универсальные веса, одинаково хорошо подходящие для перепривязки любой точки. Однако существуют примеры, показывающие, что важность контекстов варьируется в зависимости от ситуации. На рисунках 4.4 и 4.5 в левой части расположена исходная версия кода, рамкой очерчены сущности, к которым произведена привязка. В правой части помещён отредактированный код, в рамку взяты кандидаты для перепривязки. Отредактированные области выделены серым.

На рис. 4.4 представлен классический пример класса, содержащего набор методов с почти одинаковой сигнатурой — класс визитора по синтаксическому дереву. Отметим, что в реализации промышленного компилятора языка программирования количество методов `Visit` в одном классе может превышать несколько сотен. Поскольку сигнатуры методов a_1 и a_2, к которым была произведена привязка, отличаются только типами параметров, а при редактировании эти типы были изменены, контекст заголовка не является важным для

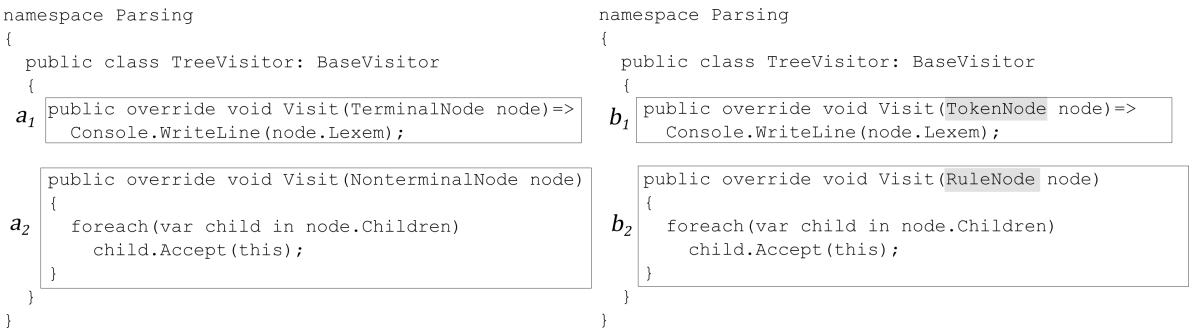


Рис. 4.4. Изменение программы, при котором для перепривязки важнее всего учёт внутреннего контекста.

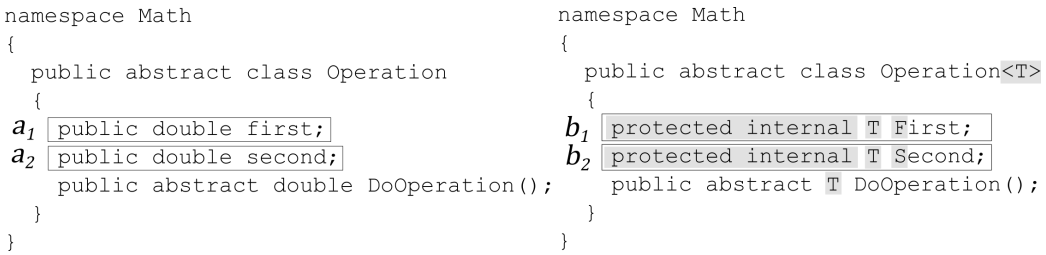


Рис. 4.5. Изменение программы, при котором для перепривязки важнее всего ядро заголовка.

правильной перепривязки. Веса $w^{\mathrm{Core}(H)}$ и $w^{\mathrm{NotCore}(H)}$ должны быть уменьшены. Внутренний контекст, напротив, был и остаётся уникальным для каждого из методов и позволяет однозначно их идентифицировать. Его вес w^I должен быть увеличен.

На рис. 4.5 в заголовках помеченных полей a_1 и a_2 полностью изменилось всё, кроме имени. Внутренний контекст в данном случае пуст, его вес следует обнулить, а наибольший вес должно иметь ядро заголовка — имена полей.

В нашем алгоритме перепривязки веса пустых контекстов устанавливаются равными нулю, а веса остальных контекстов эвристически корректируются в зависимости от того, насколько соответствующий контекст варьируется среди кандидатов и какова максимальная похожесть этого контекста на контекст, описывающий искомую точку привязки. На данный момент реализовано 5 эвристик, 3 из них представлены в алгоритме 6, остальные реализованы схожим образом.

Каждая из эвристик реализуется как функция, принимающая точку привязки a, список кандидатов и вектор весов, который нужно скорректировать. Для каждой эвристики эмпирически подобрано несколько параметров. $MaxW$

Алгоритм 6 Эвристики для настройки весов контекста.

function GetWeight(*maxW*, *minW*, *maxSim*, *minSim*, *bestSim*):
  return *minW* + (*maxW* - *minW*)
    · Max(0, Min(1, (*bestSim* - *minSim*)/(*maxSim* - *minSim*)));

function TuneInnerWeight(a, *candidates* = $\{c_1,c_2,...c_m\}$, *weights*):
  MaxW := 2; *MinW* := 0.5; *MaxSim*:= 0.9; *MinSim* := 0.6; *MinLength* := 50;
  bestSim := $\text{Max}_{c \in candidates}(\text{Sim}(I_a, I_c))$;
  w^I := GetWeight(*MaxW*, *MinW*, *MaxSim*, *MinSim*, *bestSim*);
  $w^I := w^I \cdot \text{Length}(I_a) / \text{Max}(\text{Length}(I_a), MinLength)$;

function TuneScopeWeight(a, *candidates* = $\{c_1,c_2,...c_m\}$, *weights*):
  MaxW := 1; *MinW* := 0.5; *MaxSim*:= 0.8; *MinSim* := 0;
  bestSim := $\text{Max}_{c \in candidates}(\text{Sim}(S_a, S_c))$;
  $w^S := \forall\, c \in candidates, \text{Sim}(S_a, S_c) = bestSim$
    ? MinW : GetWeight(*MaxW*, *MinW*, *MaxSim*, *MinSim*, *bestSim*);

function TuneHeaderWeight(a, *candidates* = $\{c_1,c_2,...c_m\}$, *weights*):
  $MaxW^{\text{Core}(H)}$:= 4; $MinW^{\text{Core}(H)}$:= 2; $MaxW^{\text{NotCore}(H)}$:= 4; $MinW^{\text{NotCore}(H)}$:= 1; *MaxSim* := 1; *MinSim* := 0.8;
  $w^{\text{Core}(H)} := MinW^{\text{Core}(H)}$; $w^{\text{NotCore}(H)} := MinW^{\text{NotCore}(H)}$;
  $good^{\text{Core}(H)} := \{c \in candidates \mid \text{Sim}(\text{Core}(H_a), \text{Core}(H_c)) \geq MinSim\}$;
  $good^{\text{NotCore}(H)}$:= Length($good^{\text{Core}(H)}$) > 0
    ? $\{c \in good^{\text{Core}(H)} \mid \text{Sim}(\text{NotCore}(H_a), \text{NotCore}(H_c)) \geq MinSim\}$
    : $\{c \in candidates \mid \text{Sim}(\text{NotCore}(H_a), \text{NotCore}(H_c)) \geq MinSim\}$;
  $bestSim^{\text{Core}(H)} := \text{Max}_{c \in good^{\text{Core}(H)}}(\text{Sim}(\text{Core}(H_a), \text{Core}(H_c)))$;
  $bestSim^{\text{NotCore}(H)} := \text{Max}_{c \in good^{\text{NotCore}(H)}}(\text{Sim}(\text{NotCore}(H_a), \text{NotCore}(H_c)))$;
  $bestCandidate^{\text{NotCore}(H)} := \{c \in good^{\text{NotCore}(H)} \mid \text{Sim}(\text{NotCore}(H_a), \text{NotCore}(H_c)) = bestSim^{\text{NotCore}(H)}\}[1]$;
  $secondSim^{\text{NotCore}(H)} := \text{Max}_{c \in good^{\text{NotCore}(H)} \setminus \{bestCandidate^{\text{NotCore}(H)}\}}(\text{Sim}(\text{NotCore}(H_a), \text{NotCore}(H_c)))$;
  if Length($good^{\text{Core}(H)}$) > 0 **then**
    $w^{\text{Core}(H)}$:= GetWeight($MaxW^{\text{Core}(H)}$, $MinW^{\text{Core}(H)}$, *MaxSim*, *MinSim*, $bestSim^{\text{Core}(H)}$);
  if Length($good^{\text{NotCore}(H)}$) = 1 ∨ CheckGap(1 - $bestSim^{\text{NotCore}(H)}$, 1 - $secondSim^{\text{NotCore}(H)}$) **then**
    $w^{\text{NotCore}(H)}$:= GetWeight($MaxW^{\text{NotCore}(H)}$, $MinW^{\text{NotCore}(H)}$, *MaxSim*, *MinSim*, $bestSim^{\text{NotCore}(H)}$);

и $MinW$ — это границы, в пределах которых может изменяться вес контекста. $MaxSim$ и $MinSim$ — пороговые похожести контекста. Предполагается, что вес принимает минимальное значение, если похожесть контекста для любого кандидата не превышает $MinSim$, и является максимальным, если существует кандидат, для которого похожесть контекста превышает $MaxSim$. Данная вспомогательная логика реализована в функции GetWeight (строки 1–3 алгоритма 6).

Эвристика TuneInnerWeight корректирует вес внутреннего контекста w^I. Внутренний контекст является наиболее редактируемым, поэтому его высокая похожесть является признаком того, что подходящий кандидат, скорее всего, найден. По той же причине низкая похожесть внутреннего контекста не должна оказывать существенного влияния на общую оценку. Короткий внутренний контекст менее информативен и легко изменяем, поэтому выполняется дополнительная коррекция w^I на основе длины (строка 8).

Эвристика TuneScopeWeight устанавливает w^S. Если иерархия вложенности в объемлющие элементы одинакова для всех кандидатов, что является типичным случаем при поиске в одном файле, похожесть контекста областей для всех кандидатов будет одинаковой. В этом случае w^S уменьшается до $MinW$ (строка 12), чтобы увеличить относительный вклад более информативных похожестей в итоговую оценку. При наличии нескольких иерархий вложенности для кандидатов вес w^S изменяется в зависимости от максимальной похожести контекста областей, поскольку в разных областях могут присутствовать абсолютно идентичные сущности и их нужно различать между собой.

Эвристика TuneHeaderWeight используется для настройки весов $w^{\mathrm{Core}(H)}$ и $w^{\mathrm{NotCore}(H)}$. Сначала выбираются кандидаты с «хорошими» похожестями ядра и не-ядра (строки 17—20), затем находятся похожести самого лучшего из «хороших» кандидатов и следующего за ним (строки 21–24). $w^{\mathrm{Core}(H)}$ увеличивается, если есть кандидаты с «хорошим» ядром (строки 25–26). При наличии нескольких «хороших» ядер не-ядро заголовка играет вспомогательную роль,

помогая различить кандидатов с одинаковой или близкой похожестью ядра (например, когда нужно выбрать одну из реализаций перегруженного метода). Вес $w^{\mathrm{NotCore}(H)}$ увеличивается, если есть единственный кандидат с «хорошим» не-ядром или таких кандидатов несколько, но их не-ядра непохожи друг на друга (строки 27–28).

Вес $w^{\mathrm{Nearest}(N)}$ настраивается аналогично строкам 5–7. Для компоненты *All* вес отличен от нуля только тогда, когда есть идентичные кандидаты в наиболее похожей области. В этом случае $w^{\mathrm{All}(N)}$ настраивается аналогично $w^{\mathrm{Nearest}(N)}$.

4.4. Оценка сложности алгоритма

В худшем случае на каждой итерации цикла **do − while** функции Rebind из алгоритма 5 происходит перепривязка одной точки и вычёркивание одного кандидата, тогда общее количество итераций данного цикла составляет $\min(n, m)$, где n — количество искомых точек привязки, m — количество кандидатов. Сложность функции Rebind в таком случае составляет $O(\min(n, m) \cdot n \cdot m \cdot \log(m))$.

Однако в проведённых экспериментах в 93% случаев, когда необходимо перепривязать две и более точки, от 90 до 100% точек перепривязываются на первой итерации цикла **do − while**. Менее чем 50% точек перепривязываются на первой итерации всего в 1% случаев. Если считать, что на каждой итерации цикла количество неперепривязанных точек сокращается в два и более раза, сложность перепривязки составляет $O(n \cdot m \cdot \log(m))$.

На рисунке 4.6 приведены графики зависимости времени работы алгоритма от количества точек привязки, полученные в ходе проведения экспериментов, описанных в разделе 4.6. В экспериментах привязка производилась ко всем классам, методам, полям и свойствам в коде крупных промышленных проектов с открытым исходным кодом на языке C#. Затем в отредактированной версии каждого файла из этих проектов предпринималась попытка перепривязки ко

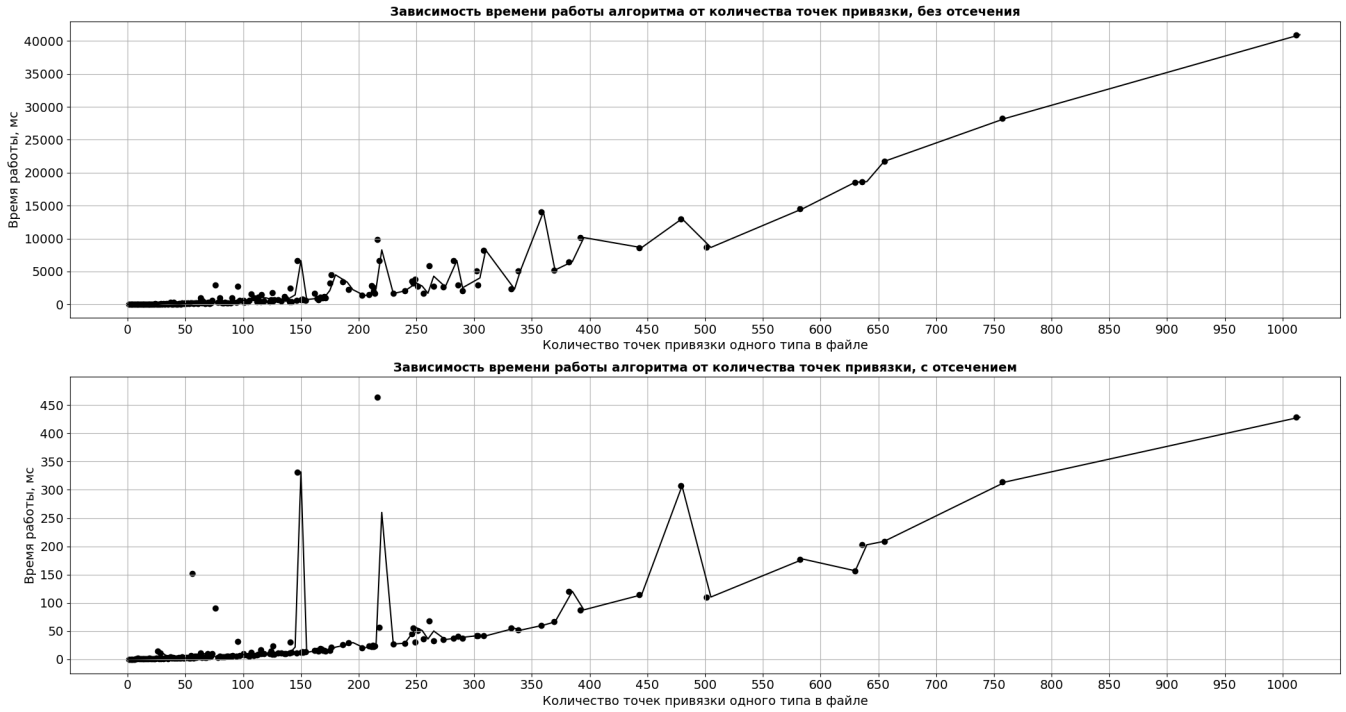


Рис. 4.6. Графики зависимости времени работы алгоритма переприв язки от количества переприв язываемых точек.

всем ранее присутствовавшим в нём элементам. Значения n и m в силу изначальной привязки ко всему в файле различались незначительно, для простоты рассуждений их можно считать одинаковыми. Поскольку одно и то же количество точек привязки возникало для разных файлов и разных типов синтаксических элементов, графики построены по медианным значениям времени работы алгоритма для каждого из количеств.

Верхний график построен при отключённом отсечении простых случаев, обеспечиваемом функцией SimpleRebind. Нижний график построен для алгоритма с включённым отсечением, в этом случае, как показано в разделе 4.6, большинство точек и кандидатов отсекаются до запуска цикла **do – while**, вносящего определяющий вклад в сложность функции Rebind. Как можно заметить, отсечение почти в 100 раз ускоряет процесс переприв язки.

Замеры проводились на вычислительной станции со следующими характеристиками:

- операционная система: Microsoft Windows 10 Домашняя;
- тип системы: x64-based PC;

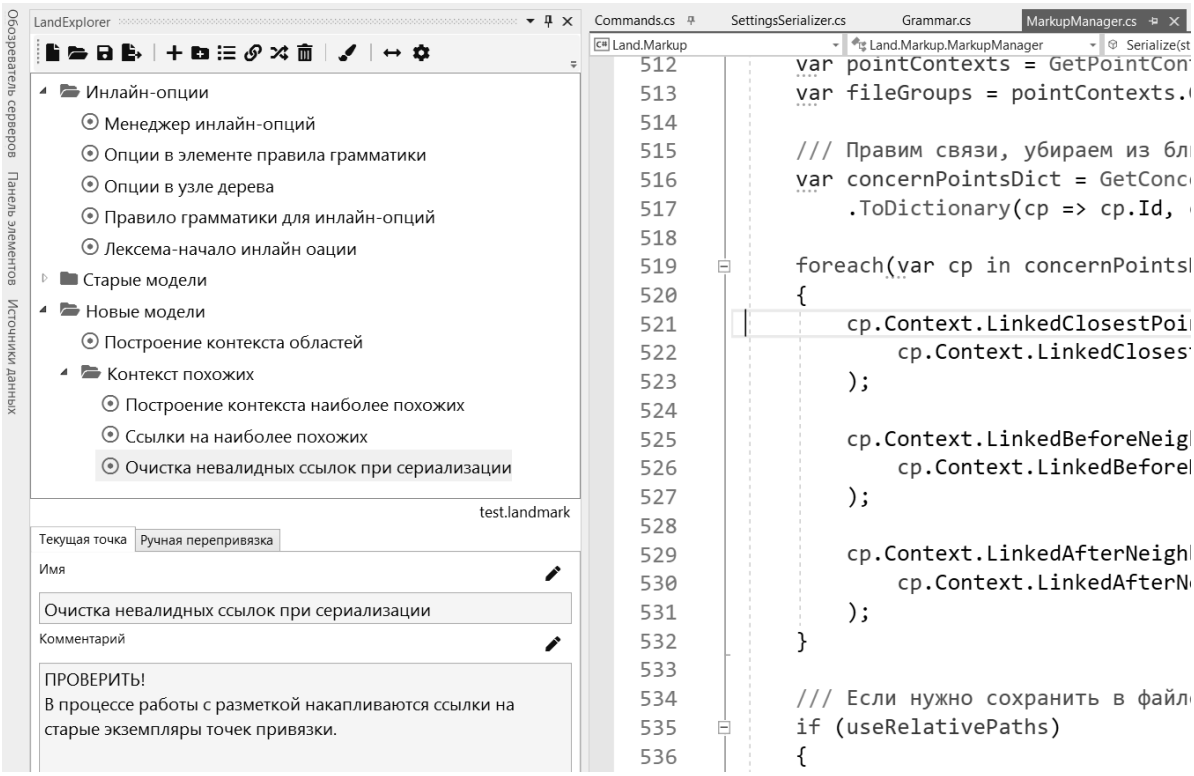


Рис. 4.7. Фрагмент окна IDE Visual Studio 2019 с интегрированной панелью разметки кода.

- процессор Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz, 4201 МГц, ядер: 4, логических процессоров: 8;
- объём оперативной памяти: 32.0 ГБ.

4.5. Инструмент разметки кода

На рисунке 4.7 представлен фрагмент окна среды разработки Visual Studio с интегрированной в неё панелью разметки кода, разработанной автором настоящей диссертации. В видимой части разметки, открытой в данной панели, присутствуют девять пометок, сгруппированных в соответствии с функциональностями, к которым они относятся: пять меток отнесены к функциональности «Инлайн-опции», четыре метки отнесены к функциональности «Новые модели», из них три — к подфункциональности «Контекст похожих».

Панель разметки использует описанные в настоящей и следующей главе модели и алгоритмы для привязки к участкам кода, которые требуется пометить как часть некоторой прорезающей функциональности. Для добавления

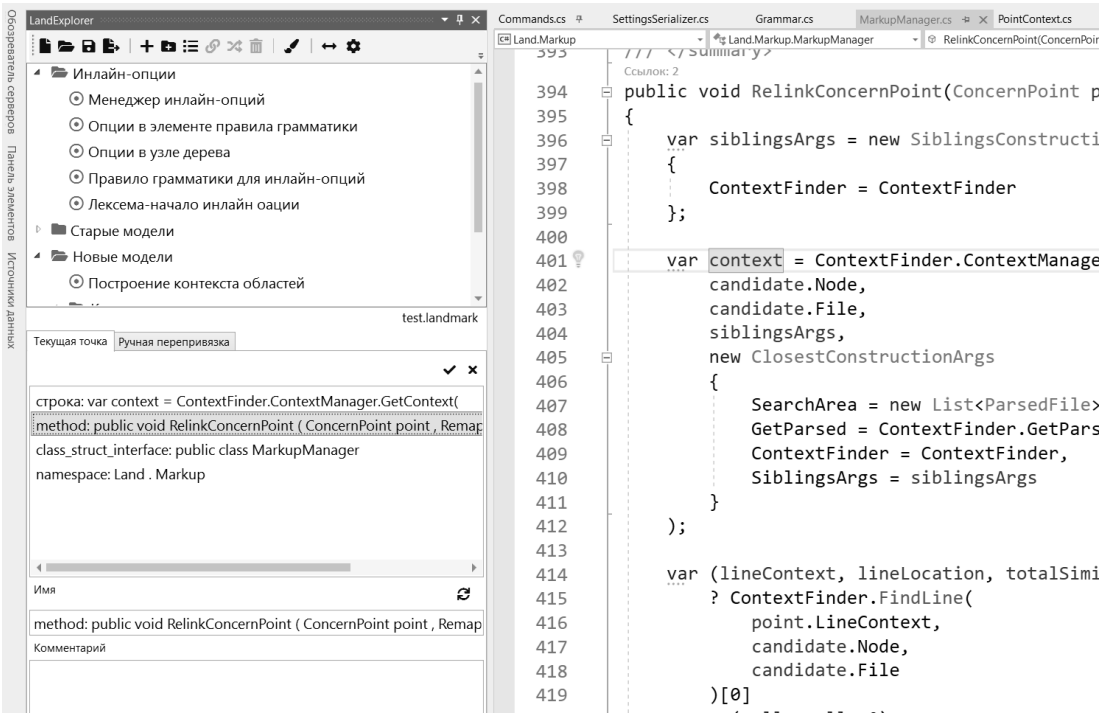


Рис. 4.8. Добавление пометки в панели разметки кода.

пометки необходимо установить курсор в нужное место в файле и нажать кнопку добавления в панели разметки. Режим создания новой пометки показан на рисунке 4.8. В нижней части панели предложен список участков кода, к которым можно осуществить привязку: это непосредственно строка, в которой установлен курсор, и все объемлющие эту строку синтаксические элементы. По умолчанию имя метки совпадает с заголовком помечаемого синтаксического элемента или с содержимым строки. Имя может быть отредактировано, как и дополнительный связываемый с пометкой комментарий. Также существует быстрый режим добавления пометки, при котором привязка происходит к наименьшей синтаксической сущности, объемлющей позицию курсора.

В случае если программист осуществляет двойное нажатие левой кнопкой мыши на какую-либо из меток в панели, происходит перепривязка точки, соответствующей этой метке, открывается нужный файл и курсор устанавливается в позицию в файле, соответствующую началу актуальной версии помеченного элемента.

4.6. Эксперименты

4.6.1. Привязка к элементам программы на языке C#

В эксперименте мы применяем модели и алгоритмы, описанные в настоящей главе, для привязки к крупным синтаксическим сущностям в промышленных программных проектах и сравниваем результаты перепривязки с результатами, полученными при помощи базовых моделей и алгоритмов. Используются репозитории проектов с открытым исходным кодом ASP.NET Core (*asp*)[1], PascalABC.NET (*pabc*)[2] и Roslyn (*ros*)[3]. Основным языком для всех проектов является C#. Для каждого проекта состояние кодовой базы на некоторую дату выбирается в качестве исходного, затем производится переход на некоторое количество коммитов вперёд и соответствующее состояние кода рассматривается как актуальное. В эксперименте участвуют только те файлы, которые представлены и в исходной, и в актуальной версии кода и были отредактированы между версиями. В таблице 4.1 для каждого проекта приведены более подробные сведения.

Таблица 4.1. Анализируемые проекты.

	Дата исходной версии	Дата актуальной версии	Коммитов	Отредактировано файлов
asp	03.02.2020	04.07.2020	3551	1398
pabc	25.05.2018	26.05.2020	1462	335
ros	03.02.2021	30.04.2021	3459	1749

В исходной версии каждого проекта при помощи нашего и базового подходов выполняется привязка ко всем классам, методам, полям и свойствам. Затем для обоих подходов делается попытка перепривязать все ранее запомненные точки к актуальной версии кодовой базы. Результаты перепривязки анализируются только для элементов, изменившихся между версиями кода, так как

[1] https://github.com/dotnet/aspnetcore
[2] https://github.com/pascalabcnet/pascalabcnet
[3] https://github.com/dotnet/roslyn

перепривязка неизмененных сущностей тривиальна и всегда корректна. Переименование объемлющей сущности также считается редактированием элемента.

В соответствии с указаниями в [60] для базового подхода используются следующие веса и параметры: $w^{\mathrm{Name}(H)} = 3$, $w^H = 1$, $w^I = 1$, $w^S = 2$, $w^N = 0$, $n_I = 10$. Вес контекста соседей равен нулю, так как в [60] утверждается, что данный контекст бесполезен для программ на C#. Базовый алгоритм дополнен некоторыми особенностями модифицированного. Во-первых, все точки привязки в файле перепривязываются одновременно, так как без этого улучшения базовый подход может ошибочно перепривязать разные точки к одному и тому же кандидату. Во-вторых, окончательные результаты получаются с помощью итеративного процесса, схожего с представленным в строках 33–52 алгоритма 5. Разница заключается в том, что пересчёт весов и оценок на каждой итерации не производится, поскольку базовый подход предполагает работу со статически заданными весами. В-третьих, функция SimpleRebind используется вместо проверки, основанной на *NearL* и *NearG*: её назначение схоже с назначением данных проверок, однако SimpleRebind отсекает большее количество простых случаев.

Для нашего подхода используются следующие значения параметров: $L_1 = 25$, $L_2 = 100$, $MaxDist = 0.4$. Они, как и параметры эвристик, подобраны опытным путём, чтобы получить хорошие результаты для проекта ASP.NET Core. Поэтому ASP.NET Core можно считать своего рода тренировочным набором для описываемых в диссертации моделей и алгоритмов, а остальные проекты — тестовыми данными.

В таблицах 4.2–4.4 представлены результаты перепривязки. Столбцы соответствуют типам сущностей, к которым производилась привязка. Числа в строке «Всего» — это общее количество сущностей каждого типа, представленных в отредактированных файлах соответствующего проекта. В строке «Отредактировано» указаны количества отредактированных сущностей в отредактированных файлах. В строке «Сложные случаи» указаны количества отредактиро-

Таблица 4.2. Результаты эксперимента для ASP.NET Core.

	Классы	Методы	Поля	Свойства
Всего	4508	18504	4691	7815
Отредактировано	2489	3771	498	3294
Сложные случаи	43	890	229	122
Автоматически	25 (21)	644 (471)	85 (83)	70 (62)
Вручную	0 (4)	36 (196)	6 (8)	8 (16)
Удалено	18 (18)	206 (207)	135 (136)	43 (43)
Ошибка	0 (0)	4 (16)	3 (2)	1 (1)

Таблица 4.3. Результаты эксперимента для PascalABC.NET.

	Классы	Методы	Поля	Свойства
Всего	1566	12601	5312	4283
Отредактировано	306	855	102	52
Сложные случаи	4	48	7	7
Автоматически	3 (1)	45 (10)	6 (6)	6 (4)
Вручную	0 (1)	1 (22)	1 (0)	1 (0)
Удалено	1 (1)	1 (1)	0 (0)	0 (0)
Ошибка	0 (1)	1 (15)	0 (1)	0 (3)

ванных сущностей, которые не были автоматически перепривязаны функцией SimpleRebind и, следовательно, обрабатывались полным алгоритмом Rebind. Заметим, что даже для длительных интервалов и большого количества коммитов между исходной и актуальной версией проекта большинство отредактированных сущностей можно перепривязать с помощью простого алгоритма SimpleRebind.

Сложные случаи дополнительно разбиты на 4 группы:

- случаи, когда алгоритм делает правильную автоматическую перепривязку (строка «Автоматически»);
- случаи, когда алгоритм запрашивает у пользователя ручную перепривязку и ставит правильного кандидата на первое место в списке кандидатов, отсортированном по увеличению расстояния до искомой точки («Вручную»);
- случаи, когда помеченный элемент удаляется между исходной версией и

Таблица 4.4. Результаты эксперимента для Roslyn.

	Классы	Методы	Поля	Свойства
Всего	3216	43678	7002	7131
Отредактировано	2104	6749	468	339
Сложные случаи	32	1084	269	132
Автоматически	16 (17)	662 (547)	99 (79)	37 (36)
Вручную	1 (0)	95 (184)	8 (28)	0 (1)
Удалено	14 (15)	303 (308)	157 (157)	92 (92)
Ошибка	1 (0)	24 (45)	5 (5)	3 (3)

текущей и алгоритм ведёт себя корректно, не перепривязываясь автоматически к неверному кандидату («Удалено»);

- случаи, когда алгоритм автоматически перепривязывается к неправильному кандидату или не ставит существующего правильного кандидата на первое место в списке, предлагаемом для ручной перепривязки («Ошибка»).

Под «правильным» кандидатом подразумевается кандидат, который действительно является актуальной версией искомой точки привязки.

В каждой из ячеек показано количество элементов соответствующего столбцу типа, для которых реализовался соответствующий строке сценарий: первое число указано для нашего алгоритма, число в скобках соответствует базовому алгоритму.

Классификация сложных случаев выполнялась автором диссертации вручную после завершения работы алгоритма перепривязки. Для этого просматривались исходная и актуальная версии кодовой базы, эмпирически оценивалось сходство искомой точки и кандидата, расстояние до которого было посчитано как минимальное, а также предпринималась попытка выявить сценарий редактирования, имевший место. Если, по мнению автора, кандидат действительно являлся актуальной версией исходного элемента, решение, принятое алгоритмом, признавалось правильным.

По результатам эксперимента можно отметить, что сложные случаи со-

ставляют 13,63% от общего количества отредактированных элементов. Для подавляющего большинства проектов и типов элементов количество успешных автоматических перепривязок в сложных случаях увеличилось по сравнению с базовым алгоритмом, а количество ошибок уменьшилось. Для базовых моделей и алгоритмов доля ошибочно принятых решений, посчитанная по всем сущностям и проектам, в сложных случаях составляет 4,87%. Аналогичный показатель для моделей и алгоритмов, предложенных в диссертации, уменьшился в два раза и составляет 2,2%. Для всех отредактированных элементов доля ошибок в базовом случае составляет 0,46%, а в результатах, полученных в диссертации, равна 0,2%. Ручная перепривязка при использовании базового метода требуется в 25,6% сложных случаев, когда найден верный кандидат для перепривязки. При использовании нашего метода доля ручных перепривязок уменьшается в три раза и составляет 8,46%.

Относительно случаев в «Удалено» было сделано наблюдение, что иногда исчезновение сущности в синтаксическом смысле не означает исчезновение в смысле семантическом. В ходе рефакторинга поле, к которому была осуществлена привязка, может быть превращено в свойство, метод может стать полем, инициализируемым лямбда-функцией, и т. п. Описанные в настоящей работе алгоритмы, как и базовые, не рассчитаны на межтиповую перепривязку, поэтому такие случаи трактуются как удаление точки привязки. Вероятно, их можно трактовать более корректно, если на заключительном этапе перепривязки попытаться найти соответствие для всех сущностей, не перепривязанных внутри своего типа, среди оставшихся кандидатов из объединённого по всем типам множества.

Вывод по группе «Ошибка», важный для задачи разметки кода, заключается в том, что для любой точки привязки и её окружения можно выполнить настолько интенсивное редактирование, что инструмент разметки не сможет произвести перепривязку. Если взаимодействие с разметкой осуществляется редко и, следовательно, редко происходит перепривязка, а программа при этом ин-

тенсивно развивается, риск накопления большого количества правок в одной помеченной сущности возрастает. Наличие элементов, похожих на помеченный, усугубляет ситуацию, так как эти элементы также могут быть отредактированы и тем самым усилят путаницу. На практике ошибки перепривязки происходят крайне редко, однако, чтобы предотвратить возникновение ошибочных ситуаций и ситуаций, когда требуется ручная перепривязка, взаимодействие с разметкой должно быть частью повседневного рабочего процесса программиста.

4.6.2. Привязка к элементам грамматики

По аналогии с экспериментом, проведённым для программ на языке C#, проводится привязка к элементам грамматики языка PascalABC.NET, описанной в формате генератора парсеров Yacc. В качестве элементов, подлежащих привязке, рассматриваются объявления токенов и нетерминалов, правила и альтернативы правил. В качестве исходной версии грамматики выбрана версия от 22.03.2019, в качестве актуальной — версия от 05.04.2021. За указанный период в проекте сделано 1323 коммита, из них грамматику затронули 76 коммитов. В таблице 4.5 представлены результаты перепривязки.

Таблица 4.5. Результаты эксперимента для грамматики PascalABC.NET.

	Объявления	Правила	Альтернативы
Всего	145	331	889
Отредактировано	14	52	68
Сложные случаи	12	4	38
Автоматически	12 (8)	1 (1)	27 (15)
Вручную	0 (2)	0 (0)	5 (15)
Удалено	0 (0)	2 (2)	4 (4)
Ошибка	0 (2)	1 (1)	2 (4)

Как следует из приведённых данных, сложные случаи составляют 40,3% от общего количества отредактированных элементов. Для базовых моделей и алгоритмов доля ошибочно принятых решений, посчитанная по всем сущностям,

в сложных случаях составляет 12,96%. Аналогичный показатель для моделей и алгоритмов, предложенных в диссертации, уменьшился более чем в два раза и составляет 5,56%. Для всех отредактированных элементов доля ошибок в базовом случае составляет 5,22%, а в результатах, полученных в диссертации, равна 2,24%. Ручная переприв язка при использовании базового метода требуется в 41,46% сложных случаев, когда найден верный кандидат для переприв язки. При использовании нашего метода доля ручных переприв язок уменьшается почти в четыре раза и составляет 11,11%.

Ошибочная переприв язка правила, допускаемая нашим алгоритмом, может быть исправлена за счёт добавления более комплексной эвристики, корректирующей вес контекста соседей с учётом похожестей остальных контекстов. В обнаруженной ошибочной ситуации исходное правило существенно отредактировано и перемещено более чем на 100 правил вниз. Все кандидаты получают невысокие оценки похожести, но наилучшую оценку получает кандидат, расположенный в точности между теми же соседями, между которыми ранее располагался искомый элемент. При этом он обладает низкой похожестью заголовка и внутреннего контекста, однако совпавший контекст соседей вносит решающий вклад в итоговую оценку за счёт большой величины $w^{\mathrm{Nearest}(N)}$.

Ошибки для типа «Альтернативы» вызваны удалением двух одноэлементных альтернатив и добавлением точно таких же альтернатив к другим правилам грамматики. Модифицированный алгоритм осуществляет автоматическую переприв язку к новым добавленным альтернативам. Решить данную проблему можно при помощи дополнительной эвристики, повышающей вес контекста областей, если известно, что наилучший кандидат встречен в ранее существовавшей области. Также ситуация может быть корректно обработана, если при переприв язке вложенных элементов учитывать результаты переприв язки объемлющих элементов.

4.6.3. Угрозы валидности экспериментов

Фактором, угрожающим валидности экспериментов, является то, что базовые модели и алгоритмы реализованы автором диссертации с нуля в соответствии с описаниями, данными в [59, 60]. При этом для упрощения обработки экспериментально полученных данных к базовому алгоритму перепривязки добавлены некоторые особенности алгоритма, предложенного в диссертации, улучшающие устойчивость привязки. Использование готовой реализации, упомянутой в [60], оказалось невозможным из-за проблем с совместимостью.

Ещё одним фактором является то, что корректность перепривязки проверялась одним человеком, не имеющим отношения к разработке соответствующих проектов и не имеющим опыта работы с выбранными для эксперимента кодовыми базами. В рамках будущих исследований планируется привлечение большего числа экспертов, возможно, разработчиков включаемых в эксперименты проектов.

Глава 5

Привязка к произвольному участку кода

Описанные в предыдущей главе модели и алгоритмы рассчитаны на привязку к крупным и средним синтаксическим сущностям программы, например, к классам и членам класса для программы на языке C#. В ходе решения задачи программисту может потребоваться пометка не метода или класса целиком, а конкретной строки кода или нескольких смежных строчек. Данная задача требует разработки дополнительных моделей и алгоритмов.

В настоящей главе описываются расширение модели, используемой для привязки к синтаксическим сущностям программы, и модификация алгоритма перепривязки, позволяющие осуществлять устойчивую привязку к одиночной строке кода. Вводится понятие пользовательского блока — вспомогательной сущности, служащей для привязки к многострочным фрагментам кода. Описывается метод встраивания в АСД узлов, соответствующих пользовательским блокам, сохраняющий корректность АСД.

5.1. Привязка к строке

Как и привязка к синтаксическим сущностям, привязка к строке осуществляется через привязку к некоторому узлу АСД, однако предполагает запоминание дополнительной информации. Ранее введённая модель **BindingPoint** расширяется дополнительным контекстом *строки* $L = (HadSame, Inner, Outer)$. Он состоит из флага дублирования строки, внутреннего контекста и внешнего контекста. $Inner$ — это тройка **TextOrHash**, построенная для текста строки, $Outer = (Before, After)$ — пара троек **TextOrHash**, построенных для конкатенированного текста всех предшествующих и всех последующих строк соответственно, расположенных в пределах той же, что и помечаемая строка, наименьшей объемлющей синтаксической сущности, которая может быть представлена

как **BindingPoint**. Например, при привязке к строке внутри метода в *Before* попадёт информация о всех предшествующих строках, принадлежащих этому же методу, а в *After* — о всех последующих. *HadSame* принимает значение 1, если в момент привязки в пределах наименьшей объемлющей синтаксической сущности существуют другие строки, содержимое которых полностью совпадает с помечаемой, и 0 в противном случае.

Привязка осуществляется в два этапа. На первом этапе производится поиск наименьшей сущности a, объемлющей интересующую программиста строку, и эта сущность запоминается как $\mathbf{BindingPoint}_a$ с пустым контекстом L_a. На втором этапе конструируется L_a, описывающий нужную строку. Отметим, что наша реализация оптимизирована таким образом, чтобы избежать повторного построения и сохранения контекстов, описывающих сущность a, при привязке к другим строкам внутри неё.

Процесс перепривязки также состоит из двух шагов: сначала осуществляется поиск сущности a в актуальной версии кода, затем, если произошла успешная перепривязка a к некоторому c, запускается алгоритм 7. Текст, соответствующий c, разбивается на строки-кандидаты, для каждой строки-кандидата вычисляются описывающий её контекст L_c и вектор покомпонентных расстояний между L_a и L_c (строки 2–6). Затем, в зависимости от того, существовали ли строки, идентичные помеченной, в момент привязки, и есть ли одинаковые или почти одинаковые по своему содержимому строки-кандидаты, похожие на искомую строку, эвристически корректируются веса компонент *Inner* и *Outer* (строки 7–10). $MaxW$ и $MinW$ — параметры, задающие максимальный и минимальный вес контекста. Приведённые в разделе 5.3 результаты получены при $MaxW = 1$ и $MinW = 0.25$. Для каждой строки-кандидата итоговое расстояние вычисляется как нормированное к диапазону $[0; 1]$ скалярное произведение вектора покомпонентных расстояний и вектора весов, после чего наилучший кандидат считается актуальной версией искомой строки.

Расстояние между соответственными тройками **TextOrHash**, образующи-

ми внутренний контекст строки и компоненты внешнего контекста, оценивается через приведённую в разделе 4.3.1 функцию похожести для **TextOrHash**.

Для компоненты *Outer* итоговое расстояние рассчитывается по формуле

$$\begin{aligned}\mathrm{Dist}(\mathrm{Outer}(L_a), \mathrm{Outer}(L_c)) = \\ (\mathrm{Dist}(\mathrm{Before}\,(\mathrm{Outer}(L_a))\,, \mathrm{Before}\,(\mathrm{Outer}(L_c))) \\ + \mathrm{Dist}(\mathrm{After}\,(\mathrm{Outer}(L_a))\,, \mathrm{After}\,(\mathrm{Outer}(L_c))))/2\,.\end{aligned}$$

Алгоритм 7 Перепривязка строки.

```
1: function REBIND(a, c)
2:     lines ← GetLines(c)
3:     lineContexts ← {line ∈ lines | GetLineContext(line)}
4:     for all L_c ∈ lineContexts do
5:         distances[L_c] ← (Dist(Inner(L_a), Inner(L_c)),
   Dist(Outer(L_a), Outer(L_c)))
6:     end for
7:     minInDist ← Min_{L_c ∈ lineContexts}(distances[L_c][1])
8:     confusionFlag ← HadSame(L_a) ∨
   |{L_c ∈ lineContexts |! CheckGap(minInDist, distances[L_c][1])}| > 1
9:     w^{Inner(L)} ← confusionFlag ? MinW : MaxW
10:    w^{Outer(L)} ← confusionFlag ? MaxW : MinW
11:    for all L_c ∈ lineContexts do
12:        totalDistances[L_c] ← distances[L_c]·(w^{Inner(L)}, w^{Outer(L)}) / (w^{Inner(L)} + w^{Outer(L)})
13:    end for
14:    orderedCandidates ← OrderByAsc (lineContexts, totalDistances)
15:    return orderedCandidates[1]
16: end function

17: function CHECKGAP(v_1, v_2)
18:    return v_2 ≠ 0 ∧ v_1 · 2 ≤ v_2
19: end function
```

Отметим, что алгоритм, предлагаемый нами для поиска однострочных фрагментов, идейно перекликается с алгоритмом LHDiff [162], разработанным для отслеживания строки кода на основе её содержимого и окружения. Однако, в отличие от [162], мы не считаем, что версия кода, существовавшая в момент

«запоминания» строки, всегда доступна. Такую доступность не может гарантировать даже применение системы контроля версий, поскольку привязка может осуществляться в моменты, когда код находится в некотором промежуточном состоянии между коммитами. Мы сохраняем ограниченный объём информации, описывающей помечаемую строку и её окружение, и опираемся только на эту сохранённую информацию при последующем поиске, предварительно сужая область поиска до объемлющей синтаксической сущности.

5.2. Многострочная привязка

В случае, если необходимо привязаться к многострочной области программы как к единому целому (такая область может состоять всего из нескольких строк либо быть довольно обширной, включающей несколько синтаксических сущностей), границы нужного фрагмента приходится указывать явно, чтобы он мог быть обнаружен на этапе синтаксического анализа программы и соответствующий ему узел мог быть помещён в АСД. После этого можно использовать описанные в главе 4 модели и алгоритмы, применяемые для привязки к крупным синтаксическим сущностям. Отметим, что привязка к многострочным областям не является в полном смысле произвольной: в силу того, что узел, соответствующий области, нужно добавить в АСД, данная область должна удовлетворять определённым требованиям, которые будут рассмотрены далее в этом разделе.

5.2.1. Обнаружение ограниченной многострочной области при парсинге программы

Чтобы распознать некоторым образом ограниченные многострочные фрагменты на этапе синтаксического анализа, можно доработать правила легковесной грамматики целевого языка программирования. Однако, как отмечено нами в [75], этот подход имеет ряд серьёзных недостатков, главный из которых —

усложнение грамматики. Многострочные фрагменты могут выделяться в разных местах программы, например, внутри метода и на уровне членов класса. Содержимое фрагмента зависит от конкретного места, в котором он находится. Чтобы описать доступный для пометки участок, для каждого из возможных расположений необходимы отдельный нетерминальный символ и отдельная альтернатива существующего правила грамматики, делающая этот символ достижимым.

Мы придерживаемся принципиально иного подхода: единственное, что нужно сообщить легковесному синтаксическому анализатору, — какие конструкции языка выступают в роли границ для помечаемых программистом многострочных фрагментов. Для этого в грамматику на языке LanD добавляется специальная конфигурационная секция `CustomBlock`. На рисунке 5.1 показаны варианты описания данной секции для языка C#. Опция `basetoken` задаёт токен, который может трактоваться как открывающая и закрывающая границы помечаемой многострочной области, опция `start` задаёт префикс, который должно иметь значение токена, указанного в `basetoken`, чтобы данный токен считался открывающей границей. Опция `end` имеет аналогичный смысл для закрывающей границы. Программист, размечающий код, самостоятельно задаёт содержательную часть границы многострочного фрагмента — всё, что идёт после `start` и `end` в тексте соответствующего токена. За счёт этого границы могут иметь ту же смысловую нагрузку, что и обычные лексемы типа `basetoken`. Отметим, что, во избежание коллизии между синтаксической сущностью программы и ограниченным многострочным фрагментом, токен, используемый в качестве `basetoken`, должен соответствовать чему-то, что пропускается в процессе синтаксического анализа, например, комментарию или директиве препроцессора.

На рисунке 5.1а продемонстрировано конфигурирование для случая, когда помечаемые многострочные фрагменты ограничиваются псевдокомментариями. На рисунке 5.2а показан текст программы, в котором таким способом выделено несколько многострочных фрагментов. На рисунке 5.1б предложен

```
%CustomBlock {
    start("//+")
    end("//-")
    basetoken COMMENT
}
```

(а)

```
%CustomBlock {
    start("#region")
    end("#endregion")
    basetoken DIRECTIVE
}
```

(б)

Рис. 5.1. Конфигурирование привязки к многострочным фрагментам для программы на языке C#: (а) фрагмент обрамляется псевдокомментариями; (б) фрагмент оформляется в виде региона.

другой вариант конфигурирования — помечаемые фрагменты можно заключать в регионы, таким образом разработчик может не только осуществлять привязку, но и лучше структурировать код в окне редактора.

Определение 5.2.1. Непрерывный участок кода, состоящий из нуля или более строк, ограниченный в соответствии с описанием секции `CustomBlock`, будем называть *обрамлённым*.

В процессе синтаксического разбора одновременно с построением АСД отслеживаются открывающие и закрывающие границы обрамлённых фрагментов. С учётом вложенностей данных фрагментов строится лес, в котором каждый узел соответствует некоторому обрамлённому участку. Затем происходит встраивание этого леса в АСД, данный процесс мы подробно описываем далее.

5.2.2. Учёт обрамлённых фрагментов в АСД

На рисунке 5.2 показан пример программы и структур, строящихся в процессе её разбора. В демонстрируемом случае лес обрамлённых фрагментов состоит из одного дерева, поскольку есть фрагмент, объемлющий всю программу. По окончании разбора происходит встраивание данного леса в АСД, результатом является АСД, в котором присутствуют узлы, соответствующие обрамлённым фрагментам (рисунок 5.2г). К обрамлённым фрагментам, узлы которых встроены в АСД, можно осуществить привязку.

Внешний вид границ обрамлённого фрагмента дополнительно регламентируется нами: мы пробуем произвести встраивание фрагмента и разрешаем

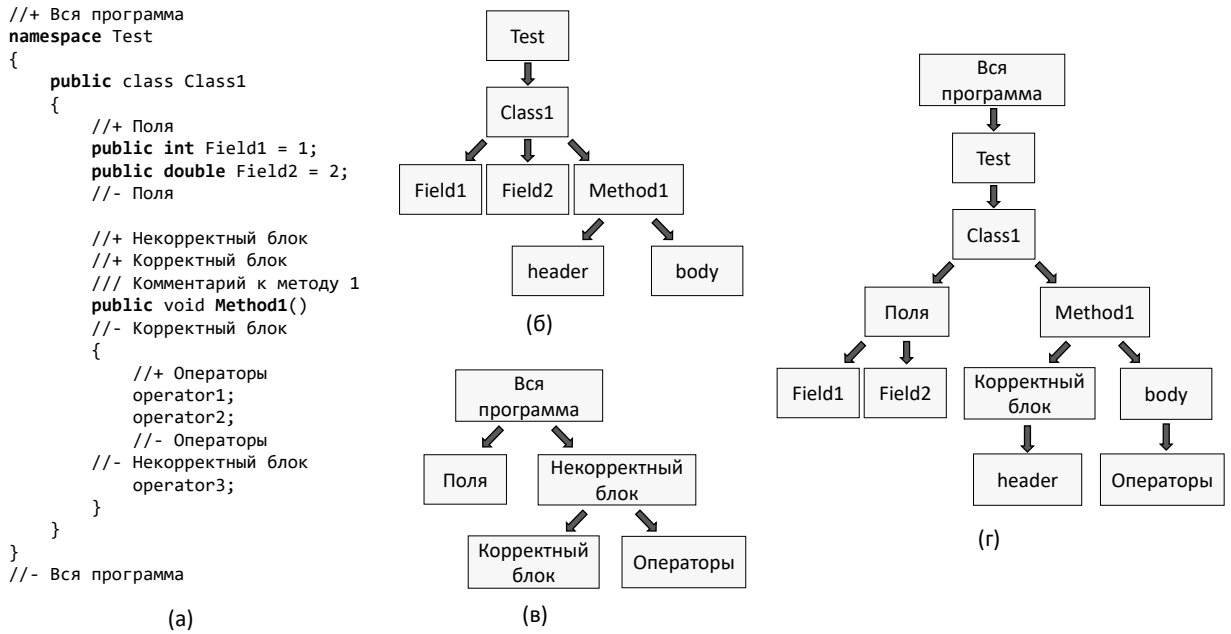


Рис. 5.2. Встраивание дерева обрамлённых фрагментов в АСД: (а) пример программы с выделенными обрамлёнными фрагментами; (б) легковесное АСД программы; (в) дерево обрамлённых фрагментов; (г) АСД, полученное в результате встраивания.

последующую привязку к нему, только если открывающая и закрывающая границы совпадают с точностью до префиксов, задаваемых опциями `start` и `end`, и совпадающая часть непуста. Благодаря этому инструмент разметки распознаёт потенциально ошибочные редактирования текста программы, нарушающие целостность сразу нескольких помеченных областей, и может проинформировать об этом программиста. Пример такого редактирования продемонстрирован на рисунке 5.3.

Помимо ограничения, указанного выше, мы накладываем на обрамлённый фрагмент ряд условий, обязательных для успешного встраивания в АСД. Эти условия гарантируют, что обрамлённый фрагмент согласуется с синтаксической структурой программы. Все они отражены в определении 5.2.5.

Пусть a и b — узлы одного или различных деревьев, построенных по тексту одной и той же программы. Определим несколько отношений на множестве узлов этих деревьев.

Определение 5.2.2. Будем говорить, что узел a *вкладывается* в узел b, и обозначать этот факт как $a \subseteq b$ тогда и только тогда, когда область текста

```
namespace Test
{
    public class Class1
    {
        public void Method1()
        {
            //+ Первый фрагмент
            operator1;
            operator2;
            //- Первый фрагмент
            operator3;
            //+ Второй фрагмент
            operator4;
            operator5;
            operator6;
            //- Второй фрагмент
        }
    }
}
```

(а)

```
namespace Test
{
    public class Class1
    {
        public void Method1()
        {
            //+ Первый фрагмент
            operator1;
            operator5;
            operator6;
            //- Второй фрагмент
        }
    }
}
```

(б)

Рис. 5.3. Потенциально опасное редактирование программы: (а) исходная версия с двумя обрамлёнными фрагментами; (б) актуальная версия.

программы, соответствующая узлу a, строго вкладывается в область текста программы, соответствующую узлу b, или совпадает с ней.

По аналогии вводится отношение строгого вложения узлов, обозначаемого как $a \subset b$, и текстового совпадения узлов, обозначаемого как $\overset{\mathrm{T}}{=}$. $a \subseteq b \equiv a \subset b \vee a \overset{\mathrm{T}}{=} b$. Отметим, что текстово совпадающие узлы могут не совпадать в обычном смысле, то есть могут являться двумя различными узлами одного или разных деревьев.

Определение 5.2.3. Будем говорить, что узел a *пересекается* с узлом b, и обозначать этот факт как $a \sqcap b$ тогда и только тогда, когда существует непустая область текста программы, вложенная и в область текста, соответствующую a, и в область текста, соответствующую b.

Определение 5.2.4. Будем говорить, что узел a *строго пересекается* с узлом b, и обозначать этот факт как $a \,!\sqcap\, b$ тогда и только тогда, когда $a \sqcap b$ и ни один узел из данной пары не вложен в другой.

Определение 5.2.5. Обрамлённый фрагмент кода будем называть *пользовательским блоком* тогда и только тогда, когда его открывающая и закрывающая границы совпадают с точностью до префиксов, задаваемых опциями `start` и

`end`, совпадающая часть этих границ непуста, и соответствующий ему узел f в дереве, принадлежащем лесу обрамлённых фрагментов, удовлетворяет одному из *условий встраивания*:

1. $r \subseteq f$;

2. существует узел n, принадлежащий АСД, такой, что для любого узла n', принадлежащего АСД и являющегося предком n либо самим n, $f \,!\sqcap\, n'$ или $f \subset n'$, а любой узел n'' такой, что $n'' \subseteq f$, является потомком n.

Первое условие встраивания позволяет добавить f в АСД, сделав его новым корнем дерева. На рисунке 5.4а показаны два пользовательских блока, узлы которых можно встроить в АСД программы в качестве корня и его непосредственного потомка. Обратите внимание на то, что область текста, соответствующая пользовательскому блоку, может быть существенно больше области, соответствующей корню АСД, поскольку пустые строки и комментарии игнорируются синтаксическим анализатором, но могут войти в состав пользовательского блока. В приведённом примере корню АСД до встраивания пользовательских блоков соответствует текст с 8 по 15 строку. Второе условие описывает тот узел n, принадлежащий АСД, к непосредственным потомкам которого можно добавить узел f, не соответствующий первому условию. Как и в случае с первым условием встраивания, области, пропускаемые синтаксическим анализатором, могут быть включены в помеченный фрагмент, поскольку определение пользовательского блока допускает не только строгую вложенность f в потенциальных предков, но и строгое пересечение с ними. На рисунке 5.4б показаны два пользовательских блока, выходящие за пределы метода `Method1`. На рисунке 5.2 продемонстрировано, как аналогичный пользовательский блок (`Корректный блок`) будет встроен в АСД. Очевидно, что в этом случае расширяется область текста, относимая к его родительскому узлу `Method1`.

Проверку второго условия и встраивание соответствующих этому условию узлов осуществляет в ходе префиксного обхода АСД алгоритм 8. В алгоритме

```
1|
2|  //+ Область 2
3|
4|  // Copyright (c) Alexey Goloveshkin
5|  // This code is distributed under
6|
7|  //+ Область 1
8|  namespace Test
9|  {
10|   public class Class1
11|   {
12|      public int Field1 = 1;
13|      public double Field2 = 2;
14|   }
15|  }
16|  //- Область 1
17|
18|  //- Область 2
```

(а)

```
1|  namespace Test
2|  {
3|    public class Class1
4|    {
5|      public int Field1 = 1;
6|
7|      //+ Область 2
8|      /// <summary>
9|      //// Метод 1
10|     /// </summary>
11|     /// <param name="n">Целое число</param>
12|     //+ Область 1
13|     public void Method1(int n)
14|     //- Область 1
15|     //- Область 2
16|     {
17|       if (n > 0)
18|       {
19|         System.Console.WriteLine("N > 0");
20|       }
21|     }
22|   }
23|  }
```

(б)

Рис. 5.4. Примеры пользовательских блоков: (а) пользовательские блоки охватывают всю программу; (б) пользовательские блоки охватывают заголовок метода.

входной параметр n — текущий узел АСД, в параметре *fragments* при посещении корня передаётся список корней всех деревьев из леса обрамлённых фрагментов. Выполнение условия 5.2.5.2 проверяется для обрамлённых фрагментов постепенно в процессе спуска по АСД. Одновременно со спуском по АСД происходит спуск по деревьям из леса обрамлённых фрагментов: при посещении очередного узла n в *fragments* передаются корни тех поддеревьев леса, узлы которых потенциально могут быть встроены в АСД в поддерево, корнем которого является n. Спуск по некоторому дереву из леса означает, что ранее рассматриваемый узел этого дерева был либо признан полностью удовлетворяющим условию 5.2.5.2 и встроен в АСД, либо отброшен из-за нарушения условия. В алгоритме 8 используется функция Children(a), обозначающая последовательность всех непосредственных потомков некоторого узла a, принадлежащего АСД либо лесу обрамлённых фрагментов.

На первом этапе (строки 4–26) для каждого узла f из *fragments* проверяется, можно ли встроить его в АСД в качестве непосредственного потомка n. В случае, если f строго пересекается с каким-то из непосредственных потомков n или вложен в него, такое встраивание произвести нельзя. В противном случае фрагмент, соответствующий f, признаётся пользовательским блоком и отбираются все непосредственные потомки n, вложенные в f или совпадающие с ним.

В результате получается некоторая подпоследовательность последовательности Children(n), возможно, пустая (строка 6). Её элементы становятся новыми непосредственными потомками f, сам f добавляется вместо этой подпоследовательности к непосредственным потомкам n. Если непосредственных потомков n, вложенных в f, не существует, f вставляется между такими непосредственными потомками n, что текст предшествующего потомка предшествует тексту f, а текст следующего следует за текстом f. Используемая в алгоритме процедура InsertChild принимает в качестве первого аргумента узел, к потомкам которого в позицию, задаваемую третьим аргументом, нужно добавить второй аргумент. По окончании цикла старые непосредственные потомки всех вставленных в АСД узлов f — элементы дерева обрамлённых фрагментов — добавляются к набору *fragments* (строки 27–31) вместо соответствующих f.

На втором этапе (строки 32–36) отбираются обрамлённые фрагменты, возможно, являющиеся пользовательскими блоками — те, что строго пересекаются ровно с одним непосредственным потомком n или строго вложены в такого потомка (строка 33). Соответствующий список передаётся в качестве второго параметра в рекурсивный вызов метода Visit для этого потомка. В итоге в массиве *fragments* остаются узлы, которые точно не удастся встроить в АСД. Однако это не означает, что не удастся встроить узлы, принадлежащие их поддеревьям. Для них проверку нужно произвести отдельно. Все узлы, оставшиеся в *fragments*, заменяются на последовательности своих непосредственных потомков (строка 37), после чего процесс повторяется.

Для абстрактного синтаксического дерева, содержащего m узлов, и леса обрамлённых фрагментов, содержащего l узлов, сложность алгоритма в худшем случае составляет $O(m \cdot l)$.

Очевидно, что алгоритм 8 позволяет добавить в АСД в качестве потомка некоторого узла n любой узел f, соответствующий второму условию встраивания: при прохождении метода Visit через предков n такой узел f попадает (непосредственно либо в составе поддерева, корень которого находится в *fragments*)

Алгоритм 8 Встраивание узлов, соответствующих пользовательским блокам, в АСД.

```
1: procedure VISIT(n, fragments)
2:     while |fragments| > 0 do
3:         inserted ← ∅
4:         for all f ∈ fragments do
5:             if {c ∈ Children(n) | f !⊓ c ∨ f ⊂ c} = ∅ then
6:                 innerChildren ← [c ∈ Children(n) | c ⊆ f]
7:                 if |innerChildren| > 0 then
8:                     index ← IndexOf(innerChildren[1], Children(n))
9:                 else
10:                    index ← 0
11:                    for i ← 1, |Children(n)| do
12:                        if Children(n)[i].StartOffset > f.EndOffset then
13:                            index ← i
14:                            break
15:                        end if
16:                    end for
17:                    if index = 0 then
18:                        index ← |Children(n)| + 1
19:                    end if
20:                end if
21:                inserted ← inserted ∪ {f};   fCopy ← f
22:                SetChildren(fCopy, innerChildren)
23:                SetChildren(n, children \ innerChildren)
24:                InsertChild(n, fCopy, index)
25:            end if
26:        end for
27:        for all f ∈ inserted do
28:            index ← IndexOf(f, fragments)
29:            RemoveAt(fragments, index)
30:            InsertAt(fragments, Children(f), index)
31:        end for
32:        for all c ∈ Children(n) do
33:            innerFragments ←
   {f ∈ fragments | f !⊓ c ∧ ∀c′ ∈ Children(n) \ {c}, ¬(c′ ⊓ f) ∨ f ⊂ c}
34:            fragments ← fragments \ innerFragments
35:            Visit(c, innerFragments)
36:        end for
37:        fragments ← ⋃_{f∈fragments} Children(f)
38:    end while
39: end procedure
```

в рекурсивные вызовы Visit, а при посещении n сам узел f или некоторый его предок из дерева обрамлённых фрагментов встраивается в АСД как охватывающий одного или нескольких непосредственных потомков n либо как не пересекающийся ни с одним из непосредственных потомков n. В случае, если на текущем шаге встраивается не сам f, а его предок из дерева обрамлённых фрагментов, обход АСД будет продолжен с учётом произошедших в нём изменений и встраивание f произойдёт при рекурсивном посещении узла, соответствующего этому предку.

На рисунке 5.2а обрамлённый фрагмент `Некорректный блок` не является пользовательским блоком, поскольку соответствующий ему узел вложен во все узлы АСД в цепочке от корня до `Method1`, однако на уровне потомков узла `Method1` он включает в себя заголовок — узел `header` — и одновременно строго пересекается с телом метода — узлом `body`. В результате `Некорректный блок` не представлен в итоговом АСД, показанном на рисунке 5.2г.

Инструмент разметки автоматически проверяет условия встраивания, если программист хочет пометить некоторый многострочный фрагмент. Создать обрамлённый фрагмент `Некорректный блок` на рисунке 5.2а при помощи инструмента разметки невозможно: при попытке привязаться к соответствующему участку кода будет предложена только привязка к объемлющему классу `Class1`.

5.2.3. Корректность определения пользовательского блока

Понятие пользовательского блока введено нами таким образом, чтобы при встраивании пользовательских блоков в АСД сохранялась корректность АСД. Под корректностью АСД мы понимаем выполнение его фундаментальных свойств: для любых узлов n_1 и n_2, принадлежащих АСД и не совпадающих друг с другом,

1. n_1 — предок $n_2 \implies n_2 \subseteq n_1$;

2. $n_1 \sqcap n_2 \iff n_1$ — предок n_2 или n_2 — предок n_1.

Утверждение 5.2.1. Пусть узел f, соответствующий пользовательскому блоку, встроен в АСД. Тогда для любых узлов n_1 и n_2, принадлежащих этому АСД и не совпадающих друг с другом и с f,

1. n_1 является предком n_2, если и только если n_1 являлся предком n_2 до встраивания f;

2. узлы n_1 и n_2 связаны отношением $\subseteq$, если и только если они были связаны этим отношением до встраивания f.

3. узлы n_1 и n_2 связаны отношением $\sqcap$, если и только если они были связаны этим отношением до встраивания f.

Доказательство. Встраивание в АСД узлов, соответствующих условию 5.2.5.1, не влияет на связи внутри изначального АСД и не изменяет области текста, соответствующие узлам, присутствовавшим в изначальном АСД. Для узлов, встраиваемых в АСД в соответствии с условием 5.2.5.2, справедливость всех частей утверждения следует из алгоритма 8. □

Отметим, что узлы n_1 и n_2, связанные отношением $\subset$ до встраивания f в АСД, могут быть связаны отношением $\overset{\mathrm{T}}{=}$ после встраивания, если f стал их потомком. Как было отмечено ранее, области текста, относимые к узлам АСД, являющимся предкам f, могут расширяться в результате встраивания f, так как должны охватывать всех потомков.

Утверждение 5.2.2. Пусть узел f, соответствующий пользовательскому блоку, встроен в АСД. Тогда для любого узла n', принадлежащего этому АСД и не совпадающего с f,

1. n' — предок $f \implies f \subseteq n'$;

2. f — предок $n' \implies n' \subseteq f$;

3. $n' \sqcap f \iff n'$ — предок f или f — предок n'.

Доказательство. Для узла f, встроенного по причине выполнения условия 5.2.5.1, предки отсутствуют, а старый корень АСД вложен в f и является его потомком, как и все остальные узлы старого АСД. Следовательно, все части утверждения выполняются. Для узлов, встраиваемых в АСД в соответствии с условием 5.2.5.2, справедливость всех частей утверждения следует из алгоритма 8. □

Утверждение 5.2.1 означает, что встраивание в АСД узла, соответствующего пользовательскому блоку, не нарушает отношения между узлами этого АСД, существовавшие до встраивания. Утверждение 5.2.2 означает, что сам узел f корректным образом включается в отношения с исходными узлами АСД.

5.3. Эксперимент

Поскольку привязка к многострочным фрагментам кода осуществляется так же, как и к синтаксическим сущностям, результаты экспериментов, приведённые в разделе 4.6, можно распространить на успешность перепривязки пользовательских блоков. Для однострочных фрагментов нами проводится отдельный эксперимент на базе тех же проектов с открытым исходным кодом, что и в разделе 4.6.

В исходной версии каждого проекта случайным образом отбираются методы (300 для *asp*, 100 для *pabc* и 300 для *ros*), которые были отредактированы в актуальной версии и успешно перепривязаны алгоритмом 5, предназначенным для крупных синтаксических элементов. Дополнительно проверяется, что в теле метода есть две и более строки. К двум случайно выбранным строкам в каждом методе производится привязка. Ожидаемым результатом является успешная перепривязка всех строк, по-прежнему существующих в актуальной версии метода. Требование успешной перепривязки самих методов позволяет исключить влияние алгоритма перепривязки крупных синтаксических элементов

на результаты настоящего эксперимента.

В таблице 5.1 приведены результаты эксперимента. Для каждого из проектов строки, к которым осуществлена привязка, разделены на две группы: «хорошие» строки — все, кроме пустых строк и строк с одиночными операторными скобками (`{` или `}`), — и «плохие» — собственно строки с одиночными операторными скобками либо пустые. В первом подстолбце для каждого проекта приведена статистика для «хороших» строк, во втором — для «плохих». Пустые строки и строки с операторными скобками выделены в отдельную группу, поскольку они почти всегда не являются уникальными в пределах метода, из-за чего для них успешность переприязки может быть значительно меньше, чем для остальных строк (в то же время вероятность того, что в реальной ситуации программист решит пометить такую строку как относящуюся к некоторой задаче, также ниже).

В строке таблицы «Всего» приведено общее количество помеченных строк, в «Переприязано, верно» показано количество случаев, когда ранее помеченная строка по-прежнему присутствует в коде и корректно переприязывается, в строке «Переприязано, неверно» указано количество строк, для которых актуальные версии по-прежнему присутствуют в коде, однако переприязка происходит не к ним. В строке «Найдено место, верно» указано количество случаев, когда искомая строка была удалена и произошла переприязка к месту, расположенному в пределах 1–2 строк от прежнего места расположения удалённой строки. В «Найдено место, неверно» считаются случаи, когда искомая строка была удалена и произошла переприязка к строке, расположенной дальше 1–2 строк от прежнего места. В строке «Успешность» посчитан процент успешно переприязанных строк от общего количества неудалённых помеченных строк.

Существенная разница между результатами переприязки «хороших» и «плохих» строк действительно наблюдается, в особенности для проекта *pabc*, где успешность переприязки неудалённых «плохих» строк составляет 74,42%, в то время как для остальных строк успешность составляет 98,66%. Подробный

Таблица 5.1. Результаты эксперимента.

	asp		*pabc*		*ros*	
Всего	433	167	155	45	435	165
Перепривязано, верно	410	151	147	32	414	144
Перепривязано, неверно	2	4	2	11	8	12
Найдено место, верно	18	9	4	1	10	6
Найдено место, неверно	3	3	2	1	3	3
Успешность, %	99,51	97,42	98,66	74,42	98,1	92,31

анализ показал, что, поскольку для «плохих» строк ключевую роль в успешной перепривязке играет внешний контекст, рядом расположенные одинаковые «плохие» строки могут стать причиной путаницы даже в случае, когда окружение редактируется незначительно. Также для них оказывается существенным то, что оценка похожести внешнего контекста ухудшается из-за изменения сигнатуры метода: в силу легковесности разбора внешний контекст строки, расположенной в теле метода, захватывает не содержимое тела, а весь текст метода. Более тонкая обработка подобных ситуаций позволила бы повысить процент успешных перепривязок.

В случае «хороших» (содержательных) строк неуспешность перепривязки в первую очередь обусловлена сильным редактированием их самих. На рисунках 5.5 и 5.6 продемонстрированы два из восьми ошибочных случаев, имевших место для «хороших» строк в *ros*. На каждом рисунке серым цветом в а) и б) выделены соответственно исходная помеченная строка и строка, к которой была осуществлена перепривязка. В рамку с прозрачным фоном взята строка, перепривязка к которой была бы корректной.

Стоит отметить, что иногда удаление строки из метода не означает, что она исчезла совсем: в каждом из проектов обнаружены случаи, когда часть метода выделяется в отдельный метод или создаётся перегруженная версия метода, куда выносится его содержимое, а старый метод вызывает свою перегруженную версию. На уровне синтаксических сущностей в этом случае происходит перепривязка к методу со старой сигнатурой, однако он не содержит ранее по-

```
var symbol = semanticModel.GetDeclaredSymbol(ancestor, cancellationToken);
if (symbol != null)
{
    if (symbol.Locations.Contains(location))
    {
        return symbol;
    }
    // We found some symbol, but it defined something else. We're not going to have a higher node defining _another_ …
    return null;
}
```

(а)

```
var symbol = semanticModel.GetDeclaredSymbol(ancestor, cancellationToken);
if (symbol != null)
{
    // The token may be part of a larger name (for example, `int` in `public static operator int[](Goo g);`.
    // So check if the symbol's location encompasses the span of the token we're asking about.
    if (symbol.Locations.Any(loc => loc.SourceTree == location.SourceTree && loc.SourceSpan.Contains(location.SourceSpan)))
        return symbol;
    // We found some symbol, but it defined something else. We're not going to have a higher node defining _another_ …
    return null;
}
```

(б)

Рис. 5.5. Пример неправильной перепривязки строки в *ros*: (а) исходная версия кода; (б) актуальная версия кода.

```
using (var token = _asyncListener.BeginAsyncOperation(nameof(FindImplementingMembersAsync)))
{
    // Let the presented know we're starting a search.
    var context = presenter.StartSearch(
        EditorFeaturesResources.Navigating, supportsReferences: true);
    using (Logger.LogBlock(
        FunctionId.CommandHandler_FindAllReference,
        KeyValueLogMessage.Create(LogType.UserAction, m => m["type"] = "streaming"),
        context.CancellationToken))
    {
```

(а)

```
using var token = _asyncListener.BeginAsyncOperation(nameof(FindImplementingMembersAsync));

// Let the presented know we're starting a search.  We pass in no cancellation token here as this
// operation itself is fire-and-forget and the user won't cancel the operation through us (though
// the window itself can cancel the operation if it is taken over for another find operation.
var (context, cancellationToken) = presenter.StartSearch(EditorFeaturesResources.Navigating, supportsReferences: true);
using (Logger.LogBlock(
    FunctionId.CommandHandler_FindAllReference,
    KeyValueLogMessage.Create(LogType.UserAction, m => m["type"] = "streaming"),
    cancellationToken))
{
```

(б)

Рис. 5.6. Пример неправильной перепривязки строки в *ros*: (а) исходная версия кода; (б) актуальная версия кода.

меченных и впоследствии вынесенных из него строк. Эти случаи учитываются в строках таблицы «Найдено место», прежним местом считается вызов метода, в котором сейчас находится искомая строка. Очевидно, что в подобных ситуациях перепривязку можно выполнить точнее, если отсутствие хороших строк-кандидатов будет влиять на перепривязку объемлющей сущности.

Заключение

Перечислим основные научные и практические результаты, полученные в ходе выполнения диссертационного исследования.

1. Предложены и формально описаны концепции легковесных LL(1) и LR(1) грамматик с символом *Any*, упрощённой LL(1) грамматики, упрощённой LR(1) грамматики, развивающие метод легковесного синтаксического анализа на основе островных грамматик.

2. Описаны и алгоритмически реализованы методы LL(1) и LR(1) синтаксического анализа, позволяющие осуществлять разбор программы, порождённой «полной» грамматикой G, на основе грамматики G_s, упрощённой относительно G; для случая LL(1) доказано, что парсер, сгенерированный по упрощённой грамматике G_s, разбирает все правильные программы, порождённые G; для случая LR(1) сформулированы условия, существенные для правильного разбора парсером, сгенерированным по упрощённой грамматике, программ, порождаемых «полной» грамматикой.

3. Реализован генератор легковесных парсеров LanD со встроенным языком описания упрощённых грамматик, использующий указанные алгоритмы; экспериментально подтверждена применимость генерируемых легковесных парсеров для анализа крупных программных проектов и выявления сущностей, интересных с точки зрения задачи привязки к коду.

4. Предложены и реализованы модели контекстов, описывающих участок программы, к которому необходимо осуществить привязку, и метод переприязки, необходимый для поиска участка в изменённом коде; экспериментально подтверждена устойчивость выполняемой с их помощью привязки: для языка C# переприязка осуществляется корректно в 99,8% случаев, причём в 91,54% успешных сложных случаев она осуществляет-

ся автоматически.

5. Предложен и алгоритмически реализован метод учёта произвольных участков кода программы в абстрактном синтаксическом дереве этой программы; сформулирован критерий корректности выделения таких участков, доказано, что встраивание узлов, соответствующих этим участкам, в абстрактное синтаксическое дерево не нарушает корректность этого дерева.

6. Реализована панель разметки кода, предназначенная для интеграции в различные интерактивные среды разработки, использующая разработанные модели и алгоритмы привязки и перепривязки.

Полученные результаты позволяют рассматривать следующие направления для продолжения исследования:

- построение модели отношений между элементами размеченных прорезающих функциональностей и использование этой модели для проверки корректности вносимых программистом изменений, затрагивающих прорезающие функциональности;

- разработка алгоритмов перепривязки, учитывающих конкретные сценарии редактирования кода (например, при рефакторинге);

- разработка метода слияния разметок, относящихся к разным версиям кода, и создание специализированной утилиты.

Список литературы

1. Minelli, R., Mocci, A., Lanza, M., Kobayashi, T. Quantifying program comprehension with interaction data // 2014 14th International Conference on Quality Software / IEEE. 2014. P. 276–285.
2. Minelli, R., Mocci, A., Lanza, M. I Know What You Did Last Summer: An Investigation of How Developers Spend Their Time // Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension. ICPC '15. IEEE Press, 2015. P. 25–35.
3. Damevski, K., Shepherd, D., Pollock, L. A field study of how developers locate features in source code // Empirical Software Engineering. 2016. Vol. 21, no. 2. P. 724–747.
4. Schröer, M., Koschke, R. Recording, Visualising and Understanding Developer Programming Behaviour // 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) / IEEE. 2021. P. 561–566.
5. Parnas, D. L. On the Criteria to Be Used in Decomposing Systems into Modules // Commun. ACM. 1972. Vol. 15, no. 12. P. 1053–1058.
6. Tarr, P., Ossher, H., Harrison, W., Sutton, S. M. N Degrees of Separation: Multi-Dimensional Separation of Concerns // Proceedings of the 21st International Conference on Software Engineering. ICSE '99. New York, NY, USA: Association for Computing Machinery, 1999. P. 107–119.
7. Kiczales, G., Lamping, J., Mendhekar, A. et al. Aspect-oriented programming // ECOOP'97 — Object-Oriented Programming. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997. P. 220–242.
8. Moonen, L. Generating Robust Parsers Using Island Grammars // Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01). WCRE '01. Washington, DC, USA: IEEE Computer Society, 2001. P. 13–22.
9. Klusener, S., Lämmel, R. Deriving Tolerant Grammars from a Base-line Grammar // Proceedings of the International Conference on Software Maintenance.

ICSM '03. Washington, DC, USA: IEEE Computer Society, 2003. P. 179–188.

10. Zaytsev, V. Formal foundations for semi-parsing // 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE) / IEEE. 2014. P. 313–317.
11. Фуксман, А. Л. Технологические аспекты создания программных систем. М.: Статистика, 1979. 183 с.
12. Горбунов-Посадов, М. М. Безболезненное развитие программы // Открытые системы. 1996. № 4. С. 65–70.
13. Горбунов-Посадов, М. М. Расширяемые программы. М.: Полиптих, 1999. 336 с.
14. Горбунов-Посадов, М. М. Как растёт программа // Открытые системы. 2000. № 10. С. 43–47.
15. Kiczales, G., Hilsdale, E., Hugunin, J. et al. An Overview of AspectJ // ECOOP 2001 — Object-Oriented Programming / Ed. by Jørgen Lindskov Knudsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. P. 327–354.
16. Masuhara, H., Kiczales, G. Modeling Crosscutting in Aspect-Oriented Mechanisms // ECOOP 2003 – Object-Oriented Programming / Ed. by Luca Cardelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. P. 2–28.
17. Prehofer, C. Feature-oriented programming: A fresh look at objects // European Conference on Object-Oriented Programming / Springer. 1997. P. 419–443.
18. Apel, S., Kästner, C., Lengauer, C. Language-Independent and Automated Software Composition: The FeatureHouse Experience // IEEE Transactions on Software Engineering. 2013. Vol. 39, no. 1. P. 63–79.
19. Schaefer, I., Bettini, L., Bono, V. et al. Delta-Oriented Programming of Software Product Lines // Proceedings of the International Software Product Line Conference (SPLC). Vol. 6287 of Lecture Notes in Computer Science. Springer-Verlag, 2010. P. 77–91.

20. Camargo, L., Fantin, L., Lobão, G. et al. Evolving Delta-Oriented Product Lines: A Case Study on Feature Interaction, Safe and Partially Safe Evolution // Brazilian Symposium on Software Engineering. SBES '21. New York, NY, USA: Association for Computing Machinery, 2021. P. 95–104.
21. Parnin, C., Rugaber, S. Resumption strategies for interrupted programming tasks // Software Quality Journal. 2011. Vol. 19, no. 1. P. 5–34.
22. Robillard, M. P., Coelho, W., Murphy, G. C. How effective developers investigate source code: An exploratory study // IEEE Transactions on software engineering. 2004. Vol. 30, no. 12. P. 889–903.
23. Rubin, J., Rinard, M. The Challenges of Staying Together While Moving Fast: An Exploratory Study // Proceedings of the 38th International Conference on Software Engineering. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016. P. 982–993.
24. Rigby, P. C., Zhu, Y. C., Donadelli, S. M., Mockus, A. Quantifying and Mitigating Turnover-Induced Knowledge Loss: Case Studies of Chrome and a Project at Avaya // Proceedings of the 38th International Conference on Software Engineering. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016. P. 1006–1016.
25. Nassif, M., Robillard, M. P. Revisiting turnover-induced knowledge loss in software projects // 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME) / IEEE. 2017. P. 261–272.
26. Chattopadhyay, S., Nelson, N., Gonzalez, Y. R. et al. Latent Patterns in Activities: A Field Study of How Developers Manage Context // Proceedings of the 41st International Conference on Software Engineering. ICSE '19. IEEE Press, 2019. P. 373–383.
27. Di Rosa, G., Mocci, A., D'Ambros, M. Visualizing Interaction Data Inside & Outside the IDE to Characterize Developer Productivity // 2020 Working Conference on Software Visualization (VISSOFT) / IEEE. 2020. P. 38–48.
28. Baker, B. S. A program for identifying duplicated code // Computing Science

and Statistics. 1992.

29. Baxter, I. D., Yahin, A., Moura, L. et al. Clone detection using abstract syntax trees // Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). 1998. P. 368–377.
30. Krinke, J. Identifying similar code with program dependence graphs // Proceedings Eighth Working Conference on Reverse Engineering. 2001. P. 301–309.
31. Kamiya, T., Kusumoto, S., Inoue, K. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code // IEEE Transactions on Software Engineering. 2002. Vol. 28, no. 7. P. 654–670.
32. Bellon, S., Koschke, R., Antoniol, G. et al. Comparison and Evaluation of Clone Detection Tools // IEEE Trans. Softw. Eng. 2007. Vol. 33, no. 9. P. 577–591.
33. Jiang, L., Misherghi, G., Su, Z., Glondu, S. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones // Proceedings of the 29th International Conference on Software Engineering. ICSE '07. USA: IEEE Computer Society, 2007. P. 96–105.
34. Roy, C. K., Cordy, J. R., Koschke, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach // Science of Computer Programming. 2009. Vol. 74, no. 7. P. 470–495.
35. Rattan, D., Bhatia, R., Singh, M. Software clone detection: A systematic review // Information and Software Technology. 2013. Vol. 55, no. 7. P. 1165–1199.
36. Sajnani, H., Saini, V., Svajlenko, J. et al. SourcererCC: Scaling Code Clone Detection to Big-Code // Proceedings of the 38th International Conference on Software Engineering. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016. P. 1157–1168.
37. Perez, D., Chiba, S. Cross-Language Clone Detection by Learning over Abstract Syntax Trees // Proceedings of the 16th International Conference on Mining Software Repositories. MSR '19. IEEE Press, 2019. P. 518–528.
38. Mostaeen, G., Svajlenko, J., Roy, B. et al. CloneCognition: Machine Learning

Based Code Clone Validation Tool // Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019. P. 1105–1109.

39. Wang, W., Li, G., Ma, B. et al. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree // 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2020. P. 261–271.
40. Walker, A., Cerny, T., Song, E. Open-Source Tools and Benchmarks for Code-Clone Detection: Past, Present, and Future Trends // SIGAPP Appl. Comput. Rev. 2020. Vol. 19, no. 4. P. 28–39.
41. Parker, A., Hamblen, J. O. Computer Algorithms for Plagiarism Detection // IEEE Trans. on Educ. 1989. Vol. 32, no. 2. P. 94–99.
42. Bowyer, K.W., Hall, L.O. Experience using "MOSS" to detect cheating on programming assignments // FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No.99CH37011. Vol. 3. 1999. P. 13B3/18–13B3/22.
43. Joy, M., Luck, M. Plagiarism in programming assignments // IEEE Transactions on education. 1999. Vol. 42, no. 2. P. 129–133.
44. Prechelt, L., Malpohl, G., Philippsen, M. et al. Finding plagiarisms among a set of programs with JPlag // J. Univers. Comput. Sci. 2002. Vol. 8, no. 11. P. 1016–1038.
45. Brixtel, R., Fontaine, M., Lesner, B. et al. Language-Independent Clone Detection Applied to Plagiarism Detection // 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation. 2010. P. 77–86.
46. Novak, M., Joy, M., Kermek, D. Source-Code Similarity Detection and Detection Tools Used in Academia: A Systematic Review // ACM Trans. Comput. Educ. 2019. Vol. 19, no. 3.

47. Cheers, H., Lin, Y., Smith, S. P. Evaluating the robustness of source code plagiarism detection tools to pervasive plagiarism-hiding modifications // Empirical Software Engineering. 2021. Vol. 26, no. 5. P. 1–62.
48. Yang, W. Identifying syntactic differences between two programs // Software: Practice and Experience. 1991. Vol. 21, no. 7. P. 739–755.
49. Fluri, B., Wuersch, M., Pinzger, M., Gall, H. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction // IEEE Trans. Softw. Eng. 2007. Vol. 33, no. 11. P. 725–743.
50. Kim, M., Notkin, D., Grossman, D., Wilson, G. Identifying and summarizing systematic code changes via rule inference // IEEE Transactions on Software Engineering. 2012. Vol. 39, no. 1. P. 45–62.
51. Falleri, J.-R., Morandat, F., Blanc, X. et al. Fine-Grained and Accurate Source Code Differencing // Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014. P. 313–324.
52. Dotzler, G., Philippsen, M. Move-Optimized Source Code Tree Differencing // Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016. P. 660–671.
53. Frick, V., Grassauer, T., Beck, F., Pinzger, M. Generating accurate and compact edit scripts using tree differencing // 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME) / IEEE. 2018. P. 264–274.
54. Калинин, С. Ю., Колоколов, И. А., Литвиненко, А. Н. Применение концепций АОП в разработке расширяемых приложений // Известия Южного федерального университета. Технические науки. 2010. Т. 103, № 2. С. 58–68.
55. Колоколов, И. А., Литвиненко, А. Н. Применение концепций АОП в разработке расширяемых приложений // Известия Южного федерального университета. Технические науки. 2010. Т. 103, № 2. С. 98–107.
56. Quatrani, T. Visual Modeling with Rational Rose 2002 and UML. 3rd edition.

Addison-Wesley, 2002.

57. Kanai, S. Customize code generation using IBM Rational Rhapsody for C++ [Электронный ресурс]. URL: https://www.ibm.com/support/pages/customize-code-generation-using-ibm-rational-rhapsody-c (дата обращения: 05.01.2022).
58. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M. EMF: Eclipse Modeling Framework. Eclipse Series. 2nd edition. Addison-Wesley, 2008.
59. Malevannyy, M., Mikhalkovich, S. Context-based model for concern markup of a source code // Trudy ISP RAN [Proc. ISP RAS]. 2016. Vol. 28, no. 2. P. 63–78.
60. Malevannyy, M., Mikhalkovich, S. Robust binding to syntactic elements in a changing code // Proceedings of the 12th Central and Eastern European Software Engineering Conference in Russia. CEE-SECR '16. New York, NY, USA: ACM, 2016. P. 13:1–13:8.
61. Van Deursen, A., Kuipers, T. Building documentation generators // Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99) / IEEE. 1999. P. 40–49.
62. Moonen, L. Lightweight Impact Analysis using Island Grammars // Proceedings of the 10th International Workshop on Program Comprehension (IWPC). IEEE Computer Society, 2002. P. 219–228.
63. Mössenböck, H. A generator for fast compiler front-ends // Report 127, Dept. Informatik, ETH Zürich. 1990. 28 p.
64. Mössenböck, H. The Compiler Generator Coco/R [Электронный ресурс]. URL: http://ssw.jku.at/Coco/Doc/UserManual.pdf (дата обращения: 05.01.2022).
65. Малёванный, М. С. Легковесный парсинг и его использование для функций среды разработки // Информатизация и связь. 2015. Т. 3. С. 89–94.
66. Kurš, J., Lungu, M., Iyadurai, R., Nierstrasz, O. Bounded Seas // Comput. Lang. Syst. Struct. 2015. Vol. 44, no. PA. P. 114–140.

67. Chomsky, N. Three models for the description of language // IRE Transactions on Information Theory. 1956. Vol. 2, no. 3. P. 113–124.
68. Chomsky, N. On certain formal properties of grammars // Information and Control. 1959. Vol. 2, no. 2. P. 137–167.
69. Lewis, P. M., Stearns, R. E. Syntax-Directed Transduction // J. ACM. 1968. Vol. 15, no. 3. P. 465–488.
70. Knuth, D. E. On the translation of languages from left to right // Information and Control. 1965. Vol. 8, no. 6. P. 607–639.
71. Головешкин, А. В. Поиск и анализ сквозных функциональностей в размеченной грамматике языка программирования // Известия вузов. Северо-Кавказский регион. Технические науки. 2017. № 3. С. 29–34.
72. Головешкин, А. В. Сквозная функциональность и её анализ в грамматике языка программирования // Языки программирования и компиляторы - 2017. Труды конференции. 2017. С. 82–85.
73. Головешкин, А. В., Михалкович, С. С. LanD: инструментальный комплекс поддержки послойной разработки программ // Труды XXV всероссийской научной конференции «Современные информационные технологии: тенденции и перспективы развития». 2018. С. 53–56.
74. Goloveshkin, A. V., Mikhalkovich, S. S. Tolerant parsing with a special kind of “Any” symbol: the algorithm and practical application // Trudy ISP RAN [Proc. ISP RAS]. 2018. Vol. 30, no. 4. P. 7–28.
75. Головешкин, А. В., Михалкович, С. С. Привязка к произвольному участку программы в задаче разметки программного кода // Труды XXVI всероссийской научной конференции «Современные информационные технологии: тенденции и перспективы развития». 2019. С. 86–89.
76. Goloveshkin, A. V., Mikhalkovich, S. S. Tolerant parsing using modified LR(1) and LL(1) algorithms with embedded “Any” symbol // Trudy ISP RAN [Proc. ISP RAS]. 2019. Vol. 31, no. 3. P. 7–28.
77. Головешкин, А. В., Михалкович, С. С. Разметка сквозных функционально-

стей в коде программы // Научный сервис в сети Интернет: труды XXI Всероссийской научной конференции. 2019. С. 245–256.

78. Головешкин, А. В. Устойчивая разметка прорезающих функциональностей в грамматике языка программирования // Материалы XXVIII всероссийской научной конференции «Современные информационные технологии: тенденции и перспективы развития». 2021. С. 143–146.
79. Goloveshkin, A. V., Mikhalkovich, S. S. Using improved context-based code description for robust algorithmic binding to changing code // Procedia Computer Science. 2021. Vol. 193. P. 239—249.
80. Головешкин, А. В., Михалкович, С. С. Устойчивая алгоритмическая привязка к произвольному участку кода программы // Программные системы: теория и приложения. 2022. Т. 13, № 1. С. 3–33.
81. Свидетельство о государственной регистрации программы для ЭВМ №2022610266 Российская Федерация. Генератор легковесных LL(1) и LR(1) синтаксических анализаторов / А. В. Головешкин; правообладатель А. В. Головешкин (RU). – №2021681187; заявл. 20.12.2021; опубл. 11.01.2022, Бюл. № 1. – 1 с.
82. Свидетельство о государственной регистрации программы для ЭВМ №2022616186 Российская Федерация. Свидетельство об официальной регистрации программы для ЭВМ. Панель разметки кода / А. В. Головешкин, С. С. Михалкович; правообладатели А. В. Головешкин (RU), С. С. Михалкович (RU). – №2022613947; заявл. 17.03.2022; опубл. 05.04.2022, Бюл. № 4. – 1 с.
83. Свидетельство о государственной регистрации программы для ЭВМ №2022616984 Российская Федерация. Библиотека устойчивой алгоритмической привязки к коду программы / А. В. Головешкин, С. С. Михалкович; правообладатели А. В. Головешкин (RU), С. С. Михалкович (RU). – №2022613933; заявл. 15.03.2022; опубл. 18.04.2022, Бюл. № 4. – 1 с.
84. Conejero, J. M., Hernández, J. Analysis of Crosscutting Features in Software

Product Lines // Proceedings of the 13th International Workshop on Early Aspects. EA '08. New York, NY, USA: Association for Computing Machinery, 2008. P. 3–10.

85. Conejero, J. M., Hernández, J., Jurado, E., van den Berg, K. Mining Early Aspects Based on Syntactical and Dependency Analyses // Sci. Comput. Program. 2010. Vol. 75, no. 11. P. 1113–1141.
86. Kaindl, H. What is an Aspect in Aspect-oriented Requirements Engineering // Proceedings of 13th International Workshop on Exploring Modeling Methods for Systems Analysis and Design. Vol. 337. 2008. P. 164–170.
87. Santos, A., Alves, P., Figueiredo, E., Ferrari, F. Avoiding code pitfalls in Aspect-Oriented Programming // Science of Computer Programming. 2016. Vol. 119. P. 31–50.
88. Aksit, M., Rensink, A., Staijen, T. A Graph-Transformation-Based Simulation Approach for Analysing Aspect Interference on Shared Join Points // Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development. AOSD '09. New York, NY, USA: Association for Computing Machinery, 2009. P. 39–50.
89. Rebernak, D., Mernik, M., Wu, H., Gray, J. Domain-specific aspect languages for modularising crosscutting concerns in grammars // IET software. 2009. Vol. 3, no. 3. P. 184–200.
90. Griswold, W. G. Coping with Crosscutting Software Changes Using Information Transparency // Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns. REFLECTION '01. Berlin, Heidelberg: Springer-Verlag, 2001. P. 250–265.
91. Ying, A. T. T., Wright, J. L., Abrams, S. Source Code That Talks: An Exploration of Eclipse Task Comments and Their Implication to Repository Mining // SIGSOFT Softw. Eng. Notes. 2005. Vol. 30, no. 4. P. 1–5.
92. Sulír, M., Nosál', M. Sharing developers' mental models through source code annotations // 2015 Federated Conference on Computer Science and Informa-

tion Systems (FedCSIS) / IEEE. 2015. P. 997–1006.

93. Kuhn, A., Ducasse, S., Girba, T. Enriching Reverse Engineering with Semantic Clustering // Proceedings of the 12th Working Conference on Reverse Engineering. IEEE, 2005. P. 133–142.
94. Corazza, A., Martino, S., Maggio, V., Scanniello, G. Weighing Lexical Information for Software Clustering in the Context of Architecture Recovery // Empirical Softw. Engg. 2016. Vol. 21, no. 1. P. 72–103.
95. Tonella, P., Ceccato, M. Aspect mining through the formal concept analysis of execution traces // 11th Working Conference on Reverse Engineering. 2004. P. 112–121.
96. Ceccato, M., Marin, M., Mens, K. et al. Applying and combining three different aspect Mining Techniques // Software Quality Journal. 2006. Vol. 14, no. 3. P. 209–231.
97. Beck, F., Dit, B., Velasco-Madden, J. et al. Rethinking User Interfaces for Feature Location // Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension. ICPC '15. IEEE Press, 2015. P. 151–162.
98. Robillard, M. P. Representing Concerns in Source Code: Ph. D. thesis / University of British Columbia, Vancouver. 2003. 139 p.
99. Robillard, M. P. Topology analysis of software dependencies // ACM Transactions on Software Engineering and Methodology. 2008. Vol. 17, no. 4. P. 18:1–18:36.
100. Kersten, M., Murphy, G. C. Using Task Context to Improve Programmer Productivity // Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering. SIGSOFT '06/FSE-14. New York, NY, USA: Association for Computing Machinery, 2006. P. 1–11.
101. Kersten, M., Murphy, G. C. Mylar: A Degree-of-Interest Model for IDEs // Proceedings of the 4th International Conference on Aspect-Oriented Software Development. AOSD '05. New York, NY, USA: Association for Computing Machinery, 2005. P. 159–168.

102. Singh, A., Henley, A. Z., Fleming, S. D., Luong, M. V. An Empirical Evaluation of Models of Programmer Navigation // 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). 2016. P. 9–19.

103. Ying, A. T. T., Robillard, M. P. The influence of the task on programmer behaviour // 2011 IEEE 19th International Conference on Program Comprehension / IEEE. 2011. P. 31–40.

104. Murphy-Hill, E., Zimmermann, T., Bird, C., Nagappan, N. The Design of Bug Fixes // Proceedings of the 2013 International Conference on Software Engineering. ICSE '13. IEEE Press, 2013. P. 332–341.

105. Maalej, W., Tiarks, R., Roehm, T., Koschke, R. On the Comprehension of Program Comprehension // ACM Trans. Softw. Eng. Methodol. 2014. Vol. 23, no. 4. 37 p.

106. Bacchelli, A., Lanza, M., Robbes, R. Linking E-Mails and Source Code Artifacts // Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010. P. 375–384.

107. Biegel, B., Beck, F., Hornig, W., Diehl, S. The order of things: How developers sort fields and methods // 2012 28th IEEE International Conference on Software Maintenance (ICSM) / IEEE. 2012. P. 88–97.

108. Herzig, K., Zeller, A. The Impact of Tangled Code Changes // Proceedings of the 10th Working Conference on Mining Software Repositories. MSR '13. IEEE Press, 2013. P. 121–130.

109. Rigby, P. C., Robillard, M. P. Discovering Essential Code Elements in Informal Documentation // Proceedings of the 2013 International Conference on Software Engineering. ICSE '13. IEEE Press, 2013. P. 832–841.

110. Kosar, T., Gaberc, S., Carver, J. C., Mernik, M. Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments // Empirical Software Engineering. 2018. Vol. 23, no. 5. P. 2734–2763.

111. Nassif, M., Robillard, M. P. Constructural Software Documentation // Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings. ICSE '19. IEEE Press, 2019. P. 308–309.

112. Akdur, D., Garousi, V., Demirörs, O. A survey on modeling and model-driven engineering practices in the embedded software industry // Journal of Systems Architecture. 2018. Vol. 91. P. 62–82.

113. Badreddin, O., Khandoker, R., Forward, A. et al. A Decade of Software Design and Modeling: A Survey to Uncover Trends of the Practice // Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. MODELS '18. New York, NY, USA: Association for Computing Machinery, 2018. P. 245–255.

114. Robillard, M. P., Weigand-Warr, F. ConcernMapper: Simple View-Based Separation of Scattered Concerns // Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange. eclipse '05. New York, NY, USA: Association for Computing Machinery, 2005. P. 65–69.

115. Guzzi, A., Hattori, L., Lanza, M. et al. Collective code bookmarks for program comprehension // 2011 IEEE 19th International Conference on Program Comprehension / IEEE. 2011. P. 101–110.

116. Bragdon, A., Zeleznik, R., Reiss, S. P. et al. Code Bubbles: A Working Set-Based Interface for Code Understanding and Maintenance // Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. CHI '10. New York, NY, USA: Association for Computing Machinery, 2010. P. 2503–2512.

117. Bragdon, A., Reiss, S. P., Zeleznik, R. et al. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments // Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010. P. 455–464.

118. Reiss, S. P., Bott, J. N., La Viola Jr, J. J. Plugging in and into code bubbles: the

code bubbles architecture // Software: Practice and Experience. 2014. Vol. 44, no. 3. P. 261–276.

119. Sulír, M., Porubän, J. Labeling source code with metadata: A survey and taxonomy // 2017 Federated Conference on Computer Science and Information Systems (FedCSIS) / IEEE. 2017. P. 721–729.
120. Sulír, M., Bačíková, M., Chodarev, S., Porubän, J. Visual augmentation of source code editors: A systematic mapping study // Journal of Visual Languages & Computing. 2018. Vol. 49. P. 46–59.
121. LaToza, T. D., Myers, B. A. Designing Useful Tools for Developers // Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools. PLATEAU '11. New York, NY, USA: Association for Computing Machinery, 2011. P. 45–50.
122. Murphy-Hill, E., Parnin, C., Black, A. P. How we refactor, and how we know it // IEEE Transactions on Software Engineering. 2011. Vol. 38, no. 1. P. 5–18.
123. Dotzler, G., Kamp, M., Kreutzer, P., Philippsen, M. More Accurate Recommendations for Method-Level Changes // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017. P. 798–808.
124. Brody, S., Alon, U., Yahav, E. A Structural Model for Contextual Code Changes // Proc. ACM Program. Lang. 2020. Vol. 4, no. OOPSLA. P. 215:1–215:28.
125. Luan, S., Yang, D., Barnaby, C. et al. Aroma: Code Recommendation via Structural Code Search // Proc. ACM Program. Lang. 2019. Vol. 3, no. OOPSLA. P. 152:1–152:28.
126. Ragkhitwetsagul, C., Krinke, J., Clark, D. A Comparison of Code Similarity Analysers // Empirical Softw. Engg. 2018. Vol. 23, no. 4. P. 2464–2519.
127. Sendall, S., Küster, J. Taming model round-trip engineering // Proceedings of Workshop on Best Practices for Model-Driven Software Development. 2004.
128. Van Paesschen, E., De Meuter, W., D'Hondt, M. SelfSync: A Dynamic

Round-Trip Engineering Environment // Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems. MoDELS'05. Berlin, Heidelberg: Springer-Verlag, 2005. P. 633–647.

129. Eden, A. H., Gasparis, E., Nicholson, J., Kazman, R. Modeling and visualizing object-oriented programs with Codecharts // Formal Methods in System Design. 2013. Vol. 43. P. 1–28.
130. Eden, A. H., Gasparis, E., Nicholson, J., Kazman, R. Round-trip engineering with the two-tier programming toolkit // Software Quality Journal. 2018. Vol. 26, no. 2. P. 249–271.
131. Diskin, Z., Gholizadeh, H., Wider, A., Czarnecki, K. A Three-dimensional Taxonomy for Bidirectional Model Synchronization // J. Syst. Softw. 2016. Vol. 111, no. C. P. 298–322.
132. Nilsson-Nyman, E., Ekman, T., Hedin, G. Practical Scope Recovery Using Bridge Parsing // Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers / Ed. by Dragan Gašević, Ralf Lämmel, Eric Van Wyk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. P. 95–113.
133. de Jonge, M., Nilsson-Nyman, E., Kats, L. C. L., Visser, E. Natural and Flexible Error Recovery for Generated Parsers // Software Language Engineering: Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. P. 204–223.
134. Nierstrasz, O., Kobel, M., Girba, T., Lanza, M. Example-Driven Reconstruction of Software Models // 11th European Conference on Software Maintenance and Reengineering (CSMR'07). 2007. P. 275–286.
135. Van den Brand, M., Sellink, M. P. A., Verhoef, C. Obtaining a COBOL Grammar from Legacy Code for Reengineering Purposes // Proceedings of the 2Nd International Conference on Theory and Practice of Algebraic Specifications. Algebraic'97. Swindon, UK: BCS Learning & Development Ltd., 1997. P. 6–16.

136. Терехов, А. Н., Л. А., Эрлих, А. А., Терехов. История и архитектура проекта RescueWare // Автоматизированный реинжиниринг программ. Издательство Санкт-Петербургского государственного университета, 2000. С. 7–19.

137. Zaytsev, V. Parser Generation by Example for Legacy Pattern Languages // SIGPLAN Not. 2017. Vol. 52, no. 12. P. 212–218.

138. Afroozeh, A., Bach, J.-C., van den Brand, M. et al. Island Grammar-Based Parsing Using GLL and Tom // Software Language Engineering: 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. P. 224–243.

139. Synytskyy, N., Cordy, J. R., Dean, T. R. Robust Multilingual Parsing Using Island Grammars // Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research. CASCON '03. IBM Press, 2003. P. 266–278.

140. Bacchelli, A., Cleve, A., Lanza, M., Mocci, A. Extracting Structured Data from Natural Language Documents with Island Parsing // Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. ASE '11. USA: IEEE Computer Society, 2011. P. 476–479.

141. Bischofberger, W. R. Sniff: A Pragmatic Approach to a C++ Programming Environment // SIGPLAN OOPS Mess. 1992. Vol. 4, no. 2. P. 229–239.

142. Koppler, R. A Systematic Approach to Fuzzy Parsing // Software: Practice and Experience. 1997. Vol. 27, no. 6. P. 637–649.

143. Carvalho, P., Oliveira, N., Henriques, P. R. Unfuzzying Fuzzy Parsing // 3rd Symposium on Languages, Applications and Technologies / Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. Vol. 38 of OpenAccess Series in Informatics (OASIcs). Bragança, Portugal: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014. P. 101–108.

144. Scott, E., Johnstone, A. GLL Parsing // Electron. Notes Theor. Comput. Sci.

2010. Vol. 253, no. 7. P. 177–189.

145. Afroozeh, A., Izmaylova, A. Faster, practical GLL parsing // International Conference on Compiler Construction / Springer. Vol. 9031 of Lecture Notes in Computer Science. 2015. P. 89–108.
146. Scott, E., Johnstone, A. Structuring the GLL parsing algorithm for performance // Science of Computer Programming. 2016. Vol. 125. P. 1–22.
147. Tomita, M. Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Norwell, MA: Kluwer Academic Publishers, 1985.
148. Verbitskaia, E., Grigorev, S., Avdyukhin, D. Relaxed parsing of regular approximations of string-embedded languages // International Andrei Ershov Memorial Conference on Perspectives of System Informatics / Springer. 2015. P. 291–302.
149. Johnson, S. C. et al. Yacc: Yet another compiler-compiler. Technical Report no. 32. Bell Laboratories Murray Hill, NJ, 07974, 1975.
150. Ford, B. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation // SIGPLAN Not. 2004. Vol. 39, no. 1. P. 111–122.
151. Ford, B. Packrat Parsing: Simple, Powerful, Lazy, Linear Time // Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming. ICFP '02. New York, NY, USA: ACM, 2002. P. 36–47.
152. Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D. Compilers: Principles, Techniques, and Tools. 2nd edition. Addison-Wesley, 2007.
153. Grune, D., Jacobs, C. J. H. Parsing Techniques: A Practical Guide. 2nd edition. New York, NY: Springer, 2008. 662 p.
154. Parr, T., Harwell, S., Fisher, K. Adaptive LL(*) Parsing: The Power of Dynamic Analysis // SIGPLAN Not. 2014. Vol. 49, no. 10. P. 579–598.
155. Van Wyk, E. R., Schwerdfeger, A. C. Context-aware Scanning for Parsing Extensible Languages // Proceedings of the 6th International Conference on Generative Programming and Component Engineering. GPCE '07. New York, NY, USA: ACM, 2007. P. 63–72.

156. Левенштейн, В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов // Доклады Академии наук СССР. 1965. Т. 163, № 4. С. 845–848.

157. Oliver, J., Cheng, C., Chen, Y. TLSH – A Locality Sensitive Hash // Proceedings of the 2013 Fourth Cybercrime and Trustworthy Computing Workshop. CTC '13. Washington, DC, USA: IEEE Computer Society, 2013. P. 7–13.

158. Kornblum, J. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing // Digit. Investig. 2006. Vol. 3. P. 91–97.

159. Roussev, V. Data Fingerprinting with Similarity Digests // Advances in Digital Forensics VI / Ed. by Kam-Pui Chow, Sujeet Shenoi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. P. 207–226.

160. Wagner, R. A., Fischer, M. J. The String-to-String Correction Problem // J. ACM. 1974. Vol. 21, no. 1. P. 168–173.

161. Pagani, F., Dell'Amico, M., Balzarotti, D. Beyond Precision and Recall: Understanding Uses (and Misuses) of Similarity Hashes in Binary Analysis // Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy. CODASPY '18. New York, NY, USA: Association for Computing Machinery, 2018. P. 354–365.

162. Asaduzzaman, M., Roy, C. K., Schneider, K. A., Di Penta, M. Lhdiff: A language-independent hybrid approach for tracking source code lines // 2013 IEEE International Conference on Software Maintenance / IEEE. 2013. P. 230–239.