Российская Академия Наук
Институт Системного Программирования

УДК 681.3.06

На правах рукописи

Архипова Мария Викторовна

АВТОМАТИЧЕСКАЯ ГЕНЕРАЦИЯ ТЕСТОВ ДЛЯ СЕМАНТИЧЕСКИХ АНАЛИЗАТОРОВ ТРАНСЛЯТОРОВ

Специальность 05.13.11 –
математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

Диссертация
на соискание ученой степени
кандидата физико-математических наук

Научный руководитель:
доктор физико-математических наук,
Петренко Александр Константинович

Москва
2006

ВВЕДЕНИЕ ... 3

1 ГЕНЕРАЦИЯ ТЕСТОВ ДЛЯ АНАЛИЗАТОРОВ КОНТЕКСТНЫХ УСЛОВИЙ ТРАНСЛЯТОРОВ: СОВРЕМЕННОЕ СОСТОЯНИЕ ... 17

1.1 Обзор методов генерации тестов для трансляторов ... 18
1.2 Основные принципы проектирования набора тестов для анализатора контекстных условий транслятора ... 23
1.3 Организация целенаправленной генерации тестов с использованием классических атрибутных грамматик ... 25
1.4 Анализ результатов генерации тестов с использованием классических атрибутных грамматик ... 33

2 КОНСТРУКТИВНОЕ ОПИСАНИЕ СТАТИЧЕСКОЙ СЕМАНТИКИ ФОРМАЛЬНЫХ ЯЗЫКОВ ... 38

2.1 Неформальное описание идеи ... 38
2.2 Конструктивное описание семантики ... 41
2.3 Критерий покрытия ... 59
2.3.1 Критерий покрытия для позитивных тестов ... 61
2.3.2 Критерий покрытия для негативных тестов ... 64

3 СЕМАНТИЧЕСКИ УПРАВЛЯЕМАЯ ГЕНЕРАЦИЯ ТЕСТОВЫХ ВХОДНЫХ ДАННЫХ ... 67

3.1 Представление данных ... 67
3.2 Генерация входных данных для позитивных тестов на основе конструктивного описания статической семантики ... 71
3.2.1 Построение первичных поддеревьев ... 80
3.2.2 Обеспечение семантической корректности тестовых текстов ... 88
3.3 Генерация негативных тестовых входных данных на основе конструктивного описания статической семантики ... 100

4 SRL: ОПИСАНИЕ НЕТРИВИАЛЬНЫХ КОНТЕКСТНЫХ УСЛОВИЙ ... 103

4.1 Контекст семантического правила ... 103
4.2 Семантические правила с фильтрами ... 106
4.3 Зависимые семантические правила ... 108
4.4 Описание объектов семантического правила посредством задания пути 113
4.5 Семантика типов ... 120
4.6 Семантические ограничения на синтаксис ... 122

ЗАКЛЮЧЕНИЕ ... 127

ЛИТЕРАТУРА ... 131

ПРИЛОЖЕНИЕ A. ГРАММАТИКА SRL (SEMANTIC RELATION LANGUAGE) 139

ПРИЛОЖЕНИЕ B. АТРИБУТНАЯ ГРАММАТИКА ЯЗЫКА L_{OO} ... 143

ПРИЛОЖЕНИЕ C. СВОД КОНТЕКСТНЫХ УСЛОВИЙ L_{OO} НА ЯЗЫКЕ SRL ... 147

ПРИЛОЖЕНИЕ D. СВОД КОНТЕКСТНЫХ УСЛОВИЙ ДЛЯ ПОДМНОЖЕСТВА ЯЗЫКА JAVA 5.0 НА ЯЗЫКЕ SRL ... 149

Введение

В данной работе будем называть трансляторами широкий класс программ, оперирующих формальными текстами и трансформирующих их либо интерпретирующих их. В качестве примера трансляторов в первую очередь следует упомянуть компиляторы языков программирования, которые получают на вход программы и переводят (трансформируют) их в машинный код. Задача трансляции решается в различных XML-процессорах, предназначенных для разбора xml-документов. Примерами трансляторов являются браузеры, трансформирующие тексты на языке HTML в команды "рисования" в экранной области памяти, текстовые процессоры с возможностями форматирования, трансформирующие описания форматированного текста в команды "рисования" или вывода на устройство печати, сервера SQL СУБД, транслирующие запросы на языке SQL в последовательность низкоуровневых операций с базой данных.

Теория и практика разработки трансляторов развиваются уже несколько десятилетий, и за это время были разработаны сотни трансляторов, тем не менее, это направление программной инженерии не теряет своей актуальности и сегодня. Развитие компьютерной индустрии, появление новых аппаратных платформ и процессоров, обострение конкуренции на рынке программного обеспечения обуславливают необходимость разработки новых трансляторов, отвечающих современным условиям. Трансляторы играют важную роль в разработке программного обеспечения, при передаче сообщений по сети, при организации B2B соединений и т.д. Поэтому требования к надежности трансляторов чрезвычайно высоки, так как от правильности их работы зависит правильность работы систем частями, которых они являются.

Одним из методов достижения высокого качества и надежности разрабатываемых трансляторов является тестирование. Трансляторы являются сложными программными системами, поэтому, тестирование

трансляторов, как и любых других сложных систем, нужно проводить как на системном уровне, так и на уровне отдельных модулей, подсистем и отдельных возможностей.

Одной из функций транслятора, является проведение анализа входных данных. Перед тем, как произвести трансляцию входных данных в другой формат, транслятор должен проанализировать входной текст для того, чтобы убедиться, что текст написан на заявленном языке, то есть в соответствии с требованиями этого языка. Во время анализа транслятор должен, в том числе, проверять, удовлетворяет ли входной текст *семантическим правилам* (или *контекстным условиям*) входного языка[1]. Эта фаза анализа обычно называется *семантическим анализом* (или *контекстным анализом*), а отвечающие за ее выполнение части транслятора, *семантическим анализатором* (или *анализатором контекстных условий*).

Качество, проводимого транслятором семантического анализа, в первую очередь определяется тем, насколько полно проверяются контекстные условия, то есть насколько широко в тестовом наборе представлены варианты исходных текстов, содержащие различные языковые конструкции и их сочетания, отличающиеся друг от друга свойствами существенными с позиций семантического анализа. *Данная работа посвящена решению именно этой проблемы: проблемы генерации входных данных для тестирования семантических анализаторов трансляторов.*

Для решения близкой задачи, задачи генерации входных данных для тестирования лексического и синтаксического анализаторов, разработан ряд хорошо зарекомендовавших себя методов автоматической генерации тестов [10-13 и др.], тесты для семантических анализаторов трансляторов, как правило, разрабатываются вручную. Методы, полученные в результате проведенных теоретических исследований в области тестирования семантических анализаторов, не получили широкого применения на

[1] Под семантическими правилами в данной работе понимаются правила статической семантики формального языка или *контекстные условия*. Вопросы, касающиеся динамической семантики входного языка (если таковая имеется), выходят за пределы данной работы.

практике [14-20 и др.], что частично объясняется более высокой сложностью описания статической семантики по сравнению с описанием синтаксиса формальных языков[2].

Для описания синтаксических правил существует формализм (BNF), обладающий порождающей природой, то есть позволяющий организовать *целенаправленную* генерацию входных данных без массовой генерации строк с последующей фильтрацией, отбраковкой тех, которые не отвечают требованиям грамматики. Для описания правил статической семантики на настоящий момент наиболее известен формализм атрибутных грамматик [1, 2]. Однако известные на сегодняшний день способы использования атрибутных грамматик для генерации тестов для проверки семантических анализаторов, во-первых, так или иначе, опираются на массовую генерацию синтаксически корректных тестов и дальнейшую фильтрацию (селекцию) среди них семантически корректных (то есть, не обеспечивают *целенаправленную* генерацию) и, во-вторых, обладают двумя существенными недостатками:

1. высока трудоемкость разработки атрибутной грамматики (по крайней мере, в случае реальных языков программирования);
2. в описании атрибутной грамматики высока вероятность повторения тех же ошибок, что и в самом семантическом анализаторе, так как разработчики анализаторов и атрибутных грамматик часто следуют одной и той же логике проектирования[3].

Преодоление этих недостатков будет означать, что мы нашли *способ целенаправленной генерации тестов для семантических анализаторов, пригодный на практике*, что и является целью данной работы. Для достижения этой цели необходимо решить две задачи:

---

[2] Далее в первой главе более подробно будут описаны существующие методы тестирования лексического, синтаксического и семантического анализаторов трансляторов.

[3] Во второй главе рассматривается пример, на котором подробно объясняются причины, по которым атрибутные грамматики слабо применимы для задачи генерации текстов с заданной семантикой.

- разработать способ формального описания статической семантики, которое было бы относительно простым и удобным для разработчиков трансляторов и при этом давало бы возможность целенаправленной генерации тестовых текстов, т.е. без массовой генерации синтаксически корректных текстов с последующей фильтрацией семантически корректных текстов;

- разработать методики генерации *позитивных* и *негативных* тестовых входных данных.

Под позитивными и негативными тестовыми входными данными в контексте данной работы понимается следующее (см. Рис. 1).

Тексты, удовлетворяющие синтаксическим требованиям входного языка, называются *синтаксически корректными*, остальные будут *синтаксически некорректными* с точки зрения грамматики данного входного языка.

Среди множества синтаксически корректных текстов можно выделить подмножество текстов, удовлетворяющих контекстным условиям входного языка, т.е. подмножество *семантически корректных* текстов. Семантически корректные тексты могут использоваться в качестве *позитивных тестовых входных данных* для анализатора контекстных условий транслятора. Позитивные тестовые входные данные предназначены для тестирования правильности проверок, осуществляемых семантическим анализатором. Такие данные часто являются самопроверяющимися тестами, т.е. в результате обработки такого теста транслятор может сигнализировать об успешном окончании обработки теста, и это будет правильным ожидаемым поведением, либо прервать свою работу сообщением об ошибке, и это будет неправильным поведением анализатора контекстных условий.

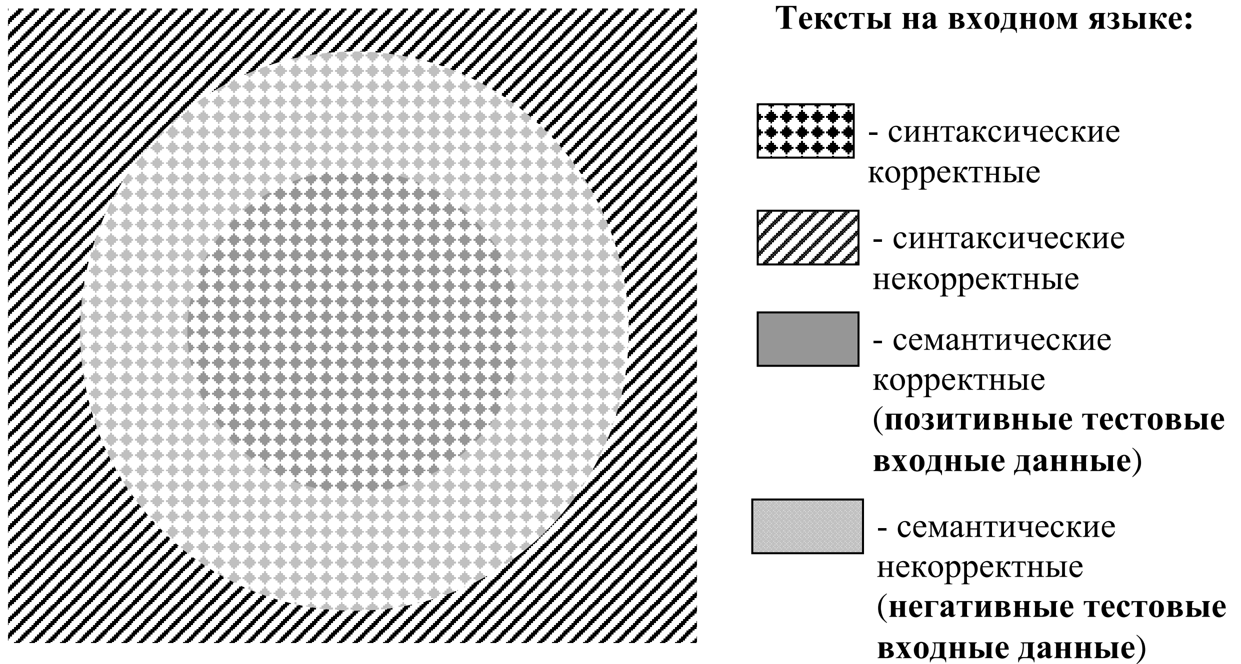


Рис. 1 Тесты для анализатора контекстных условий

Остальные синтаксически корректные тексты, которые не удовлетворяют семантическим правилам входного языка, могут использоваться в качестве входных данных для *негативных тестов* для анализатора контекстных условий, соответствующего транслятора. Негативные тесты предназначены для тестирования полноты и правильности осуществляемых проверок, и также являются самопроверяющимися. В результате обработки негативного теста ожидаемым поведением анализатора контекстных условий является демонстрация сообщения об ошибке, а не правильным поведением – сигнализация об успешном окончании обработки теста.

Перейдем теперь к краткому описанию предложенного решения поставленной задачи.

В первую очередь следует определить, что является входными данными для генератора входных данных для тестирования семантических анализаторов. По аналогии с генераторами тестов для синтаксических анализаторов, на вход генератору семантических тестов должны подаваться формальные описания синтаксиса и семантики, после чего генератор должен

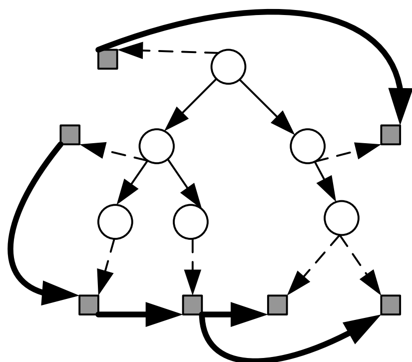
построить деревья вывода, которые затем можно было бы отобразить в тексты на формальном языке. Ясно, что наиболее удобным способом описания синтаксических правил является BNF и его производные. Нерешенным остается вопрос, *каким образом описать правила статической семантики для генератора*?

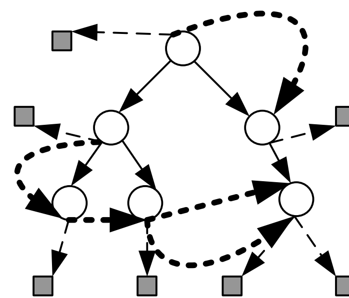
Наиболее широко используемый формальный метод описания правил статической семантики – это *атрибутные грамматики* [1, 2]. Атрибутные грамматики (АГ) удобны для решения задачи семантического анализа текстов. Так как одной из целей поставленной задачи является разработать метод *целенаправленной* генерации текстов на входном языке, то использовать атрибутные грамматики для описания семантических правил для генерации семантически корректных и некорректных текстов нельзя из-за их неконструктивной природы. Необходимо описать требования на семантическую корректность генерируемых текстов в каком-то *конструктивном* виде, который облегчит целенаправленную генерацию и позволит уменьшить непроизводительные затраты. Будем называть описание правил статической семантики в таком виде *конструктивной грамматикой*.

Рассмотрим текст на некотором формальном языке. Рассмотрим, соответствующее ему атрибутированное дерево вывода. Пусть для этого дерева задан граф атрибутной зависимости. Граф атрибутной зависимости определяет правила и порядок вычисления соответствующих атрибутов (см. Рис. 2).

Построим еще один ориентированный граф, назовем его *графом семантической зависимости*. Вершинами этого графа пусть будут вершины рассматриваемого дерева, а дугами соединяются те вершины, атрибуты которых являются смежными в соответствующем графе атрибутной зависимости, причем направление дуг в графе семантической зависимости должно совпадать с направлением соответствующих дуг графа атрибутной зависимости.

Вспомним о том, что каждая вершина дерева помечается нетерминалом входного языка. Построенный таким образом новый граф явно задает семантические зависимости между вершинами дерева подобно тому, как дерево задает синтаксические зависимости.





Граф атрибутной зависимости **Граф семантической зависимости**

Рис. 2 Графы атрибутной и семантической зависимости

Пусть каждая дуга графа семантической зависимости соответствует некоторому семантическому отношению *R*. Нетерминал, помечающий начало дуги графа семантической зависимости будем называть *источником* семантического отношения *R*, а нетерминал, помечающий конец данной дуги, – *целью* семантического отношения *R*.

Для того чтобы автоматически создать текст, удовлетворяющий семантическим требованиям входного языка, необходимо создать такое дерево, что его вершины будут составлять граф семантической зависимости, которому будет соответствовать разрешимый граф атрибутной зависимости. Описание всех возможных дуг графов семантической и атрибутной зависимости будем называть *конструктивной грамматикой*[4].

Теперь необходимо ответить на вопрос: *как на основе семантики, заданной конструктивной грамматикой, целенаправленно построить*

---

[4] Во второй и четвертой главах приводится подробное описание формального языка для конструктивного задания статической семантики.

позитивные тестовые входные данные для анализатора контекстных условий транслятора?

Схематично алгоритм целенаправленного построения теста выглядит следующим образом. Пусть требуется построить тест для проверки способности анализатора проверить требования контекстного условия R. Построим некоторое дерево вывода, в котором будут присутствовать вершины S и T, соответствующие источнику и цели контекстного правила R, причем в одном из возможных *контекстов*[5]. Отобразив построенное дерево в текст на входном языке, мы получим текст, который должен удовлетворять контекстному условию, соответствующему зафиксированному семантическому правилу R. Следовательно, будет получен входной текст, при анализе которого семантический анализатор транслятора должен будет выполнить соответствующую проверку. При условии, что данный текст будет удовлетворять и другим семантическим правилам входного языка.

Таким образом, возникает следующая проблема: на основе правила R была сгенерирована часть текста (часть дерева вывода), отвечающая семантическому правилу R, но, при этом, некоторые синтаксические и семантические требования могут быть не выполнены. Эту проблему можно решить при помощи достраивания дерева вывода таким образом, чтобы текст, соответствующий ему стал синтаксически и семантически корректным, а фрагменты, соответствующие правилу R, оставались неизменными.

В работе предложен способ достраивания дерева вывода до синтаксически и семантически корректного. Сначала строится поддерево соединяющее вершину, соответствующую стартовому правилу грамматики с источником и целью зафиксированного семантического правила R. Далее полученное дерево достраивается путем присоединения новых вершин в том

---

[5] *Контекстом* называется поддерево дерева вывода, в котором находятся вершины, соответствующие источнику и цели. В третьей главе приводится подробное описание алгоритма построения дерева, содержащего вершины, соответствующие источнику и цели зафиксированного семантического правила.

случае, если этого требует синтаксис входного языка или семантические правила.

В соответствии с синтаксическими требованиями входного языка в дерево требуется достроить вершину в том случае, если в нем существует вершина, помеченная нетерминалом, которому соответствует грамматическое правило вывода, в правой части которого есть неопциональные нетерминалы, а в дереве у данной вершины нет дочерних вершин, помеченных этими нетерминалами. Такие деревья будем называть *синтаксически неполными.*

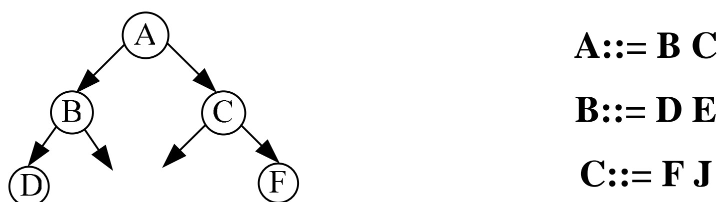


Рис. 3 Синтаксически неполное дерево вывода

Способы пополнения дерева в соответствии с требованиями семантики, определяются *типами семантических отношений*. В результате анализа статической семантики языков C и Java были выделены три типа семантических отношений, которые определяют правила построения вершин графа семантической зависимости для обеспечения семантической корректности генерируемого текста[6].

При построении очередной вершины графа семантической зависимости необходимо проверять, не противоречит ли это остальным контекстным зависимостям уже построенным в дереве. В случае обнаружения противоречия после добавления в дерево новой вершины, следует отказаться от добавления в дерево этой вершины и добавить в дерево в другую, либо, если первое невозможно, отбросить все дерево, признав его семантически неразрешимым. Подробнее процесс построения графа семантической зависимости будет рассмотрен в третьей главе.

---

[6] Более подробно типы семантических отношений («многие-ко-многим», «один-ко-многим», «один-ко-одному» и «разрешимое в пределах одного узла») рассматриваются во второй главе.

После того, как будет построено синтаксически и семантически полное дерево, отобразить его в семантически корректный текст на входном языке.

Таким образом, перебирая поочередно все семантические правила и повторив для каждого семантического отношения такой процесс построения деревьев, можно получить множество семантически корректных текстов[7], направленных на тестирование правильности проверок правил статической семантики входного языка в трансляторе.

Для конструктивного задания правил статической семантики формальных языков был разработан язык *SRL* (*Semantic Relation Language*). Конструкции *SRL* в декларативной манере позволяют описывать правила статической семантики, задавая при этом тип правила (один из трех) и предикаты на вершины дерева вывода, удовлетворяющие требованиям контекстных условий.

Предлагаемый язык SRL для формального описания статической семантики формальных языков позволяет систематизировать знания о контекстных зависимостях между нетерминалами входного языка и подготовить их для автоматического использования в задачах анализа и генерации текстов с заданной семантикой.

Представление статической семантики на языке SRL используется в методе автоматической генерации тестов для семантических анализаторов, предложенном в данной работе. Указанный метод позволяет существенно облегчить работу по созданию тестовых наборов, заменяя трудоемкий процесс написания тестовых входных данных вручную автоматическим синтезом. Кроме того, используя предложенный метод, пользователь получает возможность заранее формально описать области тестирования, и затем осуществить процесс разработки тестов, опираясь на строгие требования, продиктованные формальным описанием.

---

[7] Метод построения семантически некорректных текстов подробно описывается во второй главе.

Особенно перспективным направлением использования данного метода является генерация тестов для диалектов языков/нотаций, так как вносить небольшие изменения в их описания достаточно просто.

В работе предложены критерии покрытия, которые дают возможность автоматически определять время остановки генерации тестов[8].

На основе предложенного метода разработан инструмент *STG* (*Semantic Test Generator*) для автоматической генерации тестов для семантических анализаторов. Проводилось сравнение скоростных характеристик *STG* с гипотетической схемой генерации, базирующейся на массовой генерации и последующей фильтрации, и с генератором семантически корректных тестов, разработанным М. Посыпкиным [42]. Сопоставление показало, что скорость *STG* в первом случае выше на 3 порядка, во втором – на 1 порядок.

С помощью реализованных в рамках работы над диссертацией инструментов для автоматической генерации тестов для семантических анализаторов были получены тестовые наборы для языка C, для анализатора XML-текстов, соответствующих подмножеству описаний, заданных в стандарте MPEG-21 [43]. Также разработана часть тестового набора для анализатора контекстных условий языка Java, позволившая обнаружить ошибки в компиляторе, разрабатываемом в рамках промышленного проекта.

По теме диссертации опубликовано 6 работ, отражающих основные научные результаты диссертации:

1. Архипова (Напрасникова) М.В. Автоматическая генерация семантических тестов для трансляторов // Методы и средства обработки информации. Труды первой Всероссийской научной конференции/ Под ред. Королева Л.Н. М.: Издательский отдел факультета вычислительной математики и кибернетики МГУ им. М.В.Ломоносова, 2003. С. 448-453.
2. Штейнберг Б.Я., Архипова (Напрасникова) М.В. Минимальное множество контрольных дуг при тестировании программных модулей

[8] Критерии подробно описываются во второй главе.

// Известия ВУЗов. Северо-Кавказский регион. Естественные науки. Ростов-на-Дону: Ростовский государственный университет, 2003. № 4. С. 15-18.

3. Архипова М.В. Генерация тестов для модулей проверки статической семантики в трансляторах // Труды Института Системного Программирования/ Под ред. чл.-корр. РАН Иванникова В.П. М.: Институт Системного Программирования РАН, 2004. 8. № 1. С. 59-76.
4. Архипова М.В. Конструктивное описание правил статической семантики языков программирования // Методы и средства обработки информации. Труды второй Всероссийской научной конференции/ Под ред. Королева Л.Н. М.: Издательский отдел факультета вычислительной математики и кибернетики МГУ им. М.В.Ломоносова, 2005. С. 323-329.
5. Архипова М.В. Генерация тестов для семантических анализаторов. Препринт. М.: Институт Системного Программирования РАН, 2005. 25 с.
6. Архипова М. В. Генерация тестов для семантических анализаторов // Вычислительные методы и программирование. 2006. 7. С. 55-70. [PDF] (http://num-meth.srcc.msu.su/zhurnal/tom_2006/pdf/v7r206.pdf).

Основные положения работы докладывались на следующих конференциях и семинарах:

- на Первой Всероссийской научной конференции “Методы и средства обработки информации”, МГУ, октябрь 2003 г.
- на Второй Всероссийской научной конференции “Методы и средства обработки информации”, МГУ, октябрь 2005 г.
- на семинарах Института Системного Программирования РАН, 2003-2005 гг.
- на семинарах ВЦ РАН, май-июнь 2005 г.

Диссертация состоит из введения, четырех глав, заключения, списка литературы, включающего 59 наименования, и четырех приложений. Основной текст (без приложений) занимает 138 страниц.

В **первой главе** описывается текущее состояние исследований, связанных с тестированием трансляторов и семантических анализаторов, и формулируется основная задача диссертации.

Во **второй главе** подробно описывается способ конструктивного описания правил статической семантики и формулируются критерии остановки процесса генерации тестов.

В **третьей главе** приводится метод целенаправленной генерации позитивных и негативных тестовых входных данных для семантических анализаторов трансляторов.

В **четвертой главе** рассмотрены методы описания на языке SRL сложных семантических правил на примере семантических правил реальных языков программирования, таких как Java. В том числе будут рассмотрены способы формализации следующих групп правил:

- семантических правил, связанных с механизмом наследования в объектно-ориентированных языках программирования;
- правил, требующих проверки дополнительных условий;
- правил, описывающих условия при которых нельзя использовать некоторые синтаксические конструкции;
- семантических правил, описывающих правила приведения типов в типизированных языках программирования.

Приводятся описания конструкций *SRL* и подробные примеры их использования.

В **приложении А** приводится полная грамматика языка *SRL* в виде *EBNF*.

В **приложении В** приводится полная атрибутная грамматика модельного языка программирования L_{oo}, обладающего объектно-ориентированными свойствами, и неформальное описание правил статической семантики языка L_{oo}.

В **приложении C** приводится формальное описание семантики языка L_{oo} на языке *SRL*.

В **приложении D** содержит формальное описание семантических правил на языке *SRL* для замкнутого подмножества языка *Java 1.5*, охватывающего работу:

- с декларациями пакетов, классов, полей, методов, параметризованных классов;
- с выражениями: присваивание, арифметические операции, вызов метода и другими;
- с операторами: операторами циклов, оператором условного перехода и другими.

1 Генерация тестов для анализаторов контекстных условий трансляторов: современное состояние

В данной работе под тестированием анализатора контекстных условий в составе транслятора понимается особый вид тестирования, имеющий своей целью выявление степени соответствия анализатора контекстных условий (или *правил статической семантики*) спецификации исходного формального языка. Под спецификацией исходного формального языка подразумевается неформальное описание языка на естественном языке, например: спецификация языка Java [36].

Транслятор имеет синтаксически и семантически сложный программный вход, для которого характерна потенциальная бесконечность входных значений. Поэтому для обеспечения достижения основной цели тестирования за конечное время необходимо ограничить набор тестовых входных данных.

Определение 1. Синтаксически корректный текст, удовлетворяющий всем контекстным условиям языка, на котором этот текст написан, называется *семантически корректным.*

Для тестирования анализатора контекстных условий транслятора существенными являются семантически корректные тексты (*позитивные тестовые входные данные*), и синтаксически корректные тексты, в которых нарушены определенные контекстные ограничения (*негативные тестовые входные данные*).

Позитивные тесты, частью которых являются позитивные тестовые входные данные, позволяют проверить правильность реализации контекстных условий в трансляторе.

Негативные тесты, частью которых являются негативные тестовые входные данные, могут использоваться для проверки правильности сообщений об ошибках, выдаваемых анализатором, и для проверки способности транслятора обнаруживать ошибочные места во входных данных.

1.1 Обзор методов генерации тестов для трансляторов

Начиная с 60-х годов 20 века, были опубликованы результаты многих исследований, посвященных вопросам генерации тестов для трансляторов [10-15, 17-20 и др.]. Большинство из предложенных способов вообще не позволяют генерировать тексты с учетом контекстных ограничений входного языка, а те немногие, что позволяют, обладают существенными недостатками.

Рассмотрим основные существующие подходы.

В ранних исследованиях, посвященных генерации тестов для трансляторов, К.В. Ханфорд [10] и П. Пардом [11] представили методы, позволяющие генерировать синтаксически корректные без учета правил статической семантики программы для компиляторов процедурных языков.

Так в 1970 году была опубликована работа [10], в которой автор предлагал способ генерации тестовых данных для компилятора PL/1 на основе динамической грамматики. В результате работы алгоритма были получены синтаксически корректные программы, среди которых присутствовали семантически некорректные.

Далее следует упомянуть работу [11], ставшую фундаментальной в области генерации тестов для трансляторов. Это исследование в дальнейшем было базовым для многих других.

В качестве входных данных алгоритм П. Пардома использует контекстно-свободную грамматику, из которой выводит множество фраз так, чтобы при этом каждое продукционное правило использовалось не менее одного раза. Данный алгоритм не позволяет учитывать контекстные правила входного формального языка и может использоваться только для генерации тестов для парсеров.

В 1976 году была опубликована работа Б.А. Уичмана и Б. Джонса [12], в которой авторы обсуждали необходимость построения таких тестовых программ для компиляторов, которые позволяют проверить не только правильность синтаксического парсера, но также правильность обработки

ограничений на глубину вложенности процедур, блоков, циклов и др. При этом другие правила статической семантики, например, касающиеся использования имен переменных, при построении тестовых программ опять не учитывались.

В 1980 А. Челентано и др. в работе [13] была предложена система, частично автоматизирующая фазу тестирования при разработке компилятора. Синтаксически корректные программы генерируются по алгоритму Пардома. Грамматика входного языка, подаваемая на вход генератору, дополняется кодом, в котором производится проверка контекстных условий и преобразование синтаксически корректных программ в семантически корректные, если это необходимо. Предложенный метод применялся для подмножества языка Pascal. Было отмечено, что описание семантических проверок, например, для пользовательских типов в Pascal, уже требует значительных усилий. Стало понятно, что использование данного метода для тестирования компиляторов с объектно-ориентированных языков нецелесообразно, так как потребует еще больше усилий.

В работе [14] А. Г. Данкэн и Дж. С. Хатчисон предложили использовать атрибутные грамматики для описания входных данных генератора тестов. В процессе генерации последовательно раскрываются все нетерминальные символы, если это позволяют контекстные ограничения. Таким образом, в результате получается набор синтаксически и семантически корректных тестов, покрывающий все продукционные правила грамматики и все контекстные условия. Такой подход позволяет только вести анализ выполненных контекстных условий. Это ведет к большому количеству пустых прогонов генератора, когда из-за невыполненных контекстных условий приходится прерывать процесс генерации, и к построению большого числа семантически неинтересных тестов.

В исследовании [17], авторами которого являются Е. Г. Сирер и Б. Н. Бершэд, описывается спецификационный язык *lava*. Грамматика, заданная на *lava,* напоминает EBNF-грамматику дополненную Java-кодом

для описания контекстных ограничений. Подход применялся авторами для генерации небольшого числа тестов (~ 6 тестов), имеющих большой размер (~ 60000 инструкций). Такие тесты позволили произвести некоторые проверки JVM в рамках тестирования на устойчивость. К сожалению, в работе не приводятся оценки достигнутого покрытия тестируемой системы.

Исследования [18, 19, 20], выполненные А.С. Косачевым, М. А. Посыпкиным и др., посвящены генерации тестов для компиляторов на основе ASM-спецификации [35]. В этих работах рассматриваются также корректность тестов с точки зрения динамической семантики. Для генерации тестов используется простейший подход, подразумевающий генерацию синтаксически корректных тестов и дальнейший отбор среди них семантически корректных. Алгоритм работает следующим образом. Процесс генерации разбит на шаги. На первом шаге строятся тесты, содержащие простейшие выражения (идентификаторы и константы). Результаты работы первого шага подаются на вход генератору на втором шаге для построения более сложных выражений и т.д. Вначале оценки времени генерации тестов в соответствии с предложенным подходом были не очень хорошими: генерация тестов для mpc-компилятора на втором шаге алгоритма заняла 63 часа, приблизительная оценка времени работы алгоритма на третьем шаге – один год [19]. Позднее авторами были предложены оптимизации, позволившие генерировать тесты за приемлемое время [20].

Приведем сравнительную таблицу основных существующих подходов. В таблицу не вошли подходы, которые не дают достаточного покрытия синтаксиса входного языка.

Таб. 1. Методы генерации тестов для трансляторов

Ref.	**Входные данные генератора**	**Выполнение правил синтаксиса**	**Выполнение правил семантики**	**Возможность автоматизации**	**Применимость**
[10]	Динамическая грамматика	Да	Нет	Да	Процедурные ЯП

Ref.	Входные данные генератора	Выполнение правил синтаксиса	Выполнение правил семантики	Возможность автоматизации	Применимость
[11]	КС-грамматика	Да	Нет	Да	Функциональные ЯП
[12]	КС-грамматика	Да	Недостаточно	Нет	Функциональные ЯП
[13]	EBNF	Да	Недостаточно	Частично	Заявлена применимость к любым языковым конструкциям
[14]	Атрибутная грамматика	Да	Да	Да	Процедурные ЯП
[17]	КС-грамматика + описание контекстных условий на Java	Да	Да	Да	Применялся к java транслятору на примере подмножества java
[18] [19] [20]	ASM-based Montage	Да	Да	Да	Применялся к mpC компилятору на примере подмножества mpC.

Только три метода генерации тестов для трансляторов, из представленных в Таб. 1, позволяют построить тесты с учетом правил статической семантики входного языка, но и эти методы обладают существенными недостатками, как следует из приведенного выше подробного описания.

Одной из причин, по которым вплоть до настоящего времени не найден удобный с практической точки зрения метод генерации семантически корректных текстов, является отсутствие общепринятого, простого способа формального описания семантики формальных языков удобного для генерации тестов.

Существуют различные способы формального описания семантики формальных языков. Среди них наиболее известными являются W-

грамматики [21], аксиоматическое описание семантики [22], венский метод [23, 24, 25, 26] и атрибутные грамматики [1, 2].

Атрибутные грамматики получили наибольшее распространение и широко используются для формализации семантических контекстных ограничений языков программирования с целью теоретических исследований и для описания контекстных ограничений при разработке трансляторов и компиляторов.

Было создано несколько систем автоматизации разработки трансляторов и расширений для них, основанных на формализме атрибутных грамматик: Yacc и Lex [3], LIGA [4], Ox [5]. Опыт их использования показал, что атрибутный формализм может быть применен для частичного описания семантики языка при создании анализатора контекстных условий транслятора.

Вместе с тем выяснилось, что реализация вычислителей для атрибутных грамматик общего вида сопряжена с большими трудностями. В связи с этим рассматривались различные классы атрибутных грамматик, обладающих «хорошими» свойствами. К числу таких свойств относятся, прежде всего, простота алгоритма проверки атрибутной грамматики на зацикливание и простота алгоритма вычисления атрибутов для атрибутных грамматик данного класса.

Однако, как следует из ряда проведенных исследований [14], прямолинейное использование атрибутных грамматик для задачи генерации тестовых текстов дает возможность только вести анализ выполненных контекстных условий. Это ведет к схеме массовой генерации синтаксически правильных текстов с последующей фильтрацией (т.е. отбрасыванием семантически некорректных текстов), что требует разработки анализатора контекстных условий и существенно усложняет задачу тестирования.

До этого момента мы обсуждали существующие решения задачи генерации тестов для трансляторов. Вернемся к более узкой задаче, которой посвящена данная работа, а именно: к задаче генерации тестов для

анализатора контекстных условий в трансляторе. Сформулируем основные принципы проектирования и требования для набора тестов для анализатора контекстных условий и для алгоритма генерации тестов.

1.2 Основные принципы проектирования набора тестов для анализатора контекстных условий транслятора

Существуют два полярных подхода к тестированию: модульное и системное [37, 38, 39]. В первом случае тесты применяются непосредственно к модулям тестируемой системы (обычно через программный интерфейс — API), во втором случае — к системе в целом и, соответственно, проверяется результат работы всей системы, а не ее отдельных модулей.

На практике обычно сложно выделить в трансляторе модуль, реализующий проверку контекстных ограничений — анализатор контекстных условий, для того, чтобы протестировать его. Поэтому проводят системное тестирование трансляторов: на вход транслятора подают тестовые тексты и оценивают результат работы транслятора целиком, а не его отдельных модулей.

Однако даже при системном тестировании проектирование тестов может быть направлено на проверку каких-то определенных этапов компиляции. В тестировании на основе моделей такой прием является типовым [40, 41]. В таком случае при тестировании анализатора контекстных условий предполагается, что

- в тестируемом трансляторе есть модуль (подсистема), отвечающий за анализ контекстных условий;
- перечень функций анализа задается описанием языка;
- систематическая проверка транслятора в ситуациях, когда эти функции должны будут выполняться, даст хорошее тестовое покрытие.

Далее формальное описание языка будет рассматриваться нами в качестве основной части модели входных данных транслятора.

Использование описания языка в качестве исходных данных для генерации тестов обладает следующими преимуществами:

1. четко привязывает тесты к функциональным требованиям тестируемого транслятора;
2. позволяет легко вносить изменения в тестовый набор при работе с диалектами и “расширенными подмножествами” языков, с которыми приходится работать в реальной жизни.

Формальное описание языка должно предоставлять возможности для целенаправленной генерации, которая позволит построить тестовый набор, отвечающий следующим требованиям:

- набор тестов должен быть компактным;
- тесты должны быть нацелены на проверку определенных возможностей тестируемого транслятора (например: способность проверять выполнение контекстных условий).

Общая сложность целевой системы и спецификации являются причиной высокой трудоемкости построения и сопровождения тестового набора. Проблему трудоемкости можно решить путем автоматизации построения тестов. Для автоматизации процесса генерации тестов для анализатора контекстных условий транслятора требуется перевести спецификацию контекстных ограничений исходного языка с естественного языка на формальный, разработать и реализовать алгоритм генерации тестов.

С точки зрения пользователя идеальная автоматическая генерация тестов для транслятора могла бы быть организована следующим образом. Пользователь описывает синтаксис и статическую семантику языка, а также указывает желаемый размер набора тестов, и/или требования к полноте набора тестов в терминах некоторой метрики тестового покрытия. После этого некоторый автомат строит необходимый набор тестовых текстов на данном формальном языке. Причем известно, на проверку, какой функции тестируемого транслятора, направлен каждый тест. Так, например, каждый позитивный или негативный семантический тест может быть направлен на

правильность действий, выполняемых транслятором для проверки какого-то контекстного условия. Для этого во входных данных в позитивном тесте данное контекстное условие должно обязательно выполняться, а в негативном должно быть нарушено.

Для реализации такой генерации тестов необходимо, чтобы входные данные генератора позволяли указывать способ конструирования подходящих тестов. Такой подход мог бы сократить непроизводительные затраты, вызванные массовой генерацией и последующей фильтрацией. Описание семантики такого рода будем называть *конструктивным*.

В следующих двух параграфах приводится анализ проблем, связанных с построением целенаправленной генерации семантических тестов, и предлагаются пути решения найденных проблем.

1.3 Организация целенаправленной генерации тестов с использованием классических атрибутных грамматик

Как уже упоминалось ранее, автоматический метод генерации тестов для анализатора контекстных условий, в основе которого лежит построение всех синтаксически корректных текстов и дальнейший поиск среди них семантически корректных текстов, является низкоэффективным. Применение этого метода требует построения фильтра эквивалентного анализатору контекстных условий транслятора. Итерация всех синтаксически корректных текстов с дальнейшей фильтрацией в применении, например, к реальным языкам программирования оказывается слишком ресурсоемкой и, кроме того, изменение грамматики исходного языка влечет трудоемкие изменения в фильтре, что затрудняет сопровождение тестового набора.

Несмотря на все недостатки, такой метод генерации тестов широко используется [14, 18, 19, 20]. Существует множество способов оптимизации, указанного способа построения тестов, которые позволяют получить приемлемые результаты. Например, можно не дожидаться построения теста и проверять контекстные условия на раннем этапе прямо перед раскрытием очередного продукционного правила [14].

В работе [14] предложен алгоритм, на вход которому подается формальное описание грамматики исходного языка в виде EBNF и формальное описание контекстных ограничений в виде атрибутной грамматики. Затем продукционные правила исходного языка последовательно раскрываются в случае, если это позволяют соответствующие контекстные ограничения. Процесс продолжается до тех пор, пока не будут раскрыты все продукционные правила хотя бы по одному разу. Пользователь задает параметры, ограничивающие глубину рекурсии раскрытия продукционных правил.

Рассмотрим несколько примеров. Для описания контекстных ограничений исходного языка в стиле атрибутных грамматик применяется форма записи, использованная в работах [6, 7]. Приведем основные обозначения:

- Nonterminal.attribute – слева от точки находится нетерминал грамматики, справа – его атрибут;
- := – операция, обозначающая вычисление значения атрибута;
- f (t1.a, t2.a) – функция, которая обычно пишется на Си и реализует какие-то проверки контекстных условий и вычисления более сложные чем присваивание; параметрами функции могут быть атрибуты, относящиеся к нетерминалам;
- = – обозначает операцию проверки эквивалентности значений слева и справа от знака =;
- Cond: – предшествует выражению, описывающему проверку условия.

Пример 1.

В качестве модельного языка рассмотрим простой язык *L*, задающий описание переменных и их синонимов, при этом каждая переменная и синоним должны иметь уникальные имена. Синоним может быть объявлен только для объявленной переменной или синонима. Идентификаторы всех переменных и синонимов должны быть различными.

Ниже приводится формальное описание грамматики языка *L* и полужирным шрифтом формальное описание контекстных ограничений, которые для удобства пронумерованы.

```
program ::= "program" identifier ";" declarations
  1. program.symtab := emptysymtab()
declarations ::= declaration | declarations
declaration ::= ( "var" new_var_identifier )
  2. Cond: contains(declaration.new_var_identifier, program.symtab)=FALSE
  3. program.symtab := push(declaration.new_var_identifier,
     program.symtab)|( "syn" new_syn_identifier "=" var_identifier) ";"
  4. Cond: contains(declaration.new_syn_identifier, program.symtab)=FALSE
  5. Cond: contains(declaration.var_identifier, program.symtab)=TRUE
  6. Cond: being_ahead(declaration.var_identifier,
     new_syn_identifier)=TRUE
  7. program.symtab := push(declaration.new_syn_identifier,
     program.symtab)
new_var_identifier ::= identifier
new_syn_identifier ::= identifier
var_identifier ::= identifier
identifier ::= <String>
```

Неформальное описание контекстных условий приводится в Таб. 2. Затемненные ряды в таблице соответствуют собственно требованиям контекстных условий.

Таб. 2. Комментарии к описанию атрибутной грамматики языка L

№	Описание
1	Инициализируется атрибут symtab нетерминала program, описывающий таблицу имен.
2	Проверка вхождения значения атрибута new_var_identifier нетерминала declaration в таблицу имен. Производится проверка уникальности идентификатора новой переменной.
3	Добавление значения атрибута new_var_identifier нетерминала declaration в таблицу имен.

№	Описание
4	Проверка вхождения значения атрибута new_syn_identifier нетерминала declaration в таблицу имен. Производится проверка уникальности идентификатора нового синонима.
5	Проверка вхождения значения атрибута var_identifier нетерминала declaration в таблицу имен. Производится проверка существования декларации идентификатора.
6	Проверка того, что декларация переменной var_identifier находится перед декларацией new_syn_identifier.
7	Добавление значения атрибута new_syn_identifier нетерминала declaration в таблицу имен.

На Рис. 4 представлены первые пять деревьев вывода, которые получены в результате раскрытия продукционных правил грамматики языка L, в соответствии с приведенным ранее алгоритмом. Следующие деревья строятся аналогично.

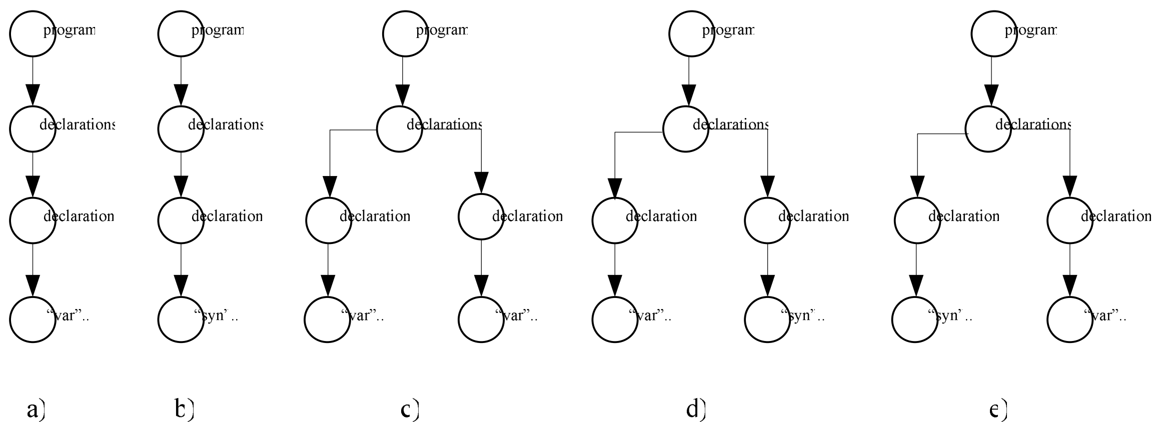


Рис. 4 Первые пять деревьев вывода, полученные в результате раскрытия продукционных правил грамматики языка L

Тестовые программы, соответствующие этим деревьям, приведены в Таб. 3.

Как мы видим, в случаях (b) и (e) на *Рис. 4* семантически корректные программы построить не удалось, так как при раскрытии нетерминала `declaration` по второй альтернативе (`"syn" new_syn_identifier"="var_identifier`) нарушается контекстное

условие 5, поскольку ни одна переменная не была объявлена, и таблица символов пуста.

Таб. 3. Результаты генерации тестов для языка L

(a)	`program p1;` `var a;`
(b)	*Программа не будет построена, так как нарушается контекстное ограничение 5*

(c)	`program p1;`
	`var a;`
	`var b;`

(d)	`program p1;`
	`var a;`
	`syn b = a;`

(e)	*Программа не будет построена, так как нарушается контекстное ограничение 5*

Выводы к примеру 1

Функции, использованные в описании атрибутной грамматики языка *L*, это пользовательские функции, которые пишутся, например, на языке Си. В примере 1 пришлось написать три таких функций.

При изменении семантики языка *L* возможно потребуется изменять эти функции или писать новые.

Построение двух из пяти деревьев вывода было прервано из-за нарушения контекстных условий. Программа (a) семантически «неинтересна», т.к. она не обязана удовлетворять никаким контекстным условиям.

Вернемся к случаям (b) и (e). Построенное синтаксическое (то есть, синтаксически корректное) дерево в обоих случаях пришлось отбросить, как не удовлетворяющее требованию 5. Однако, обнаружив такую ситуацию, генератор мог бы не отказываться от результатов уже проделанной работы, а постараться достроить дерево так, чтобы удовлетворить данное требование.

Такой прием позволил бы сократить число неуспешных попыток построить семантически корректный тест, то есть процесс генерации стал бы более целенаправленным. При этом, каждый раз решая задачу достраивания синтаксического дерева, мы не ищем подходящий способ достройки среди множества синтаксически корректных структур, а строим только те части дерева, которые необходимы для удовлетворения рассматриваемого контекстного условия.

Если довести пример до конца и сгенерировать все возможные синтаксически корректные тексты, то процент отсева будет существенно больше, то же можно сказать и про процент «неинтересных» тестов.

Пример 2.

В настоящее время широкое распространение получили объектно-ориентированные языки программирования C++, Java, C#. Несмотря на это, объектно-ориентированная парадигма слабо отражена в работах, посвященных генерации тестов для трансляторов.

При появлении понятий “объект” и “наследование” существенно усложнились правила определения областей видимости деклараций переменных. В объектно-ориентированных языках переменная, декларированная в одном классе, может быть унаследована, и затем использоваться в дочерних классах. Наследование классов влечет за собой появление зависимостей между контекстными условиями. Например, в языке *Java* если класс является дочерним по отношению к другому классу, то в теле данного класса можно обращаться к членам родительского класса при помощи ключевого слова *super*.

Понятие инкапсуляции описывает механизм, регулирующий правила доступа к членам классов. Область видимости поля класса зависит от модификатора, указанного в описании переменной (*private, public, protected*).

В объектно-ориентированных языках правила приведения типов стали более гибкими и одновременно более сложными. Введение иерархий классов

фактически дало пользователю возможность задавать свои типы и правила их приведения.

В Таб. 4 приводятся общие отличия статической семантики объектно-ориентированных и процедурных языков.

Таб. 4. Влияние ОО свойств на статическую семантику процедурных языков

ОО свойство	**Влияние на семантику процедурных языков**
Наследование	Расширение области возможного месторасположения декларации объекта.
	Изменение области видимости декларации.
	Появление зависимостей между контекстными условиями.
Инкапсуляция	Параметризация понятия области видимости.
Полиморфизм	Усложнение правил неявного приведения типов.

Рассмотрим модельный язык L_{oo}, реализующий основные свойства современных объектно-ориентированных языков программирования: наследование, полиморфизм и инкапсуляцию. Пусть язык L_{oo} позволяет описывать классы (наподобие Java-классов), которые могут наследоваться друг от друга. В графе наследования классов не должно быть циклов. Внутри класса могут быть заданы *private* и *public* методы. *Public* методы наследуются. Метод с модификатором *private* может быть вызван только внутри класса, в котором этот метод задан. Методы с модификатором *public* доступны и из других классов. Вызов метода осуществляется, как вызов статических методов в Java, через имя класса. Имена всех классов в одной программе должны быть уникальны; имена всех методов (декларированных в

этом классе и унаследованных) в одном классе должны быть также уникальны.

Приложение В содержит формальное описание грамматики языка L_{oo} (полужирным шрифтом обозначено формальное описание контекстных ограничений, которые для удобства пронумерованы) и таблицу с неформальными описаниями контекстных ограничений.

Аналогично тому, как это сделано в примере 1, последовательность раскрытия нетерминалов в процессе генерации программ на языке L_{oo} представлена на *Рис. 5*.

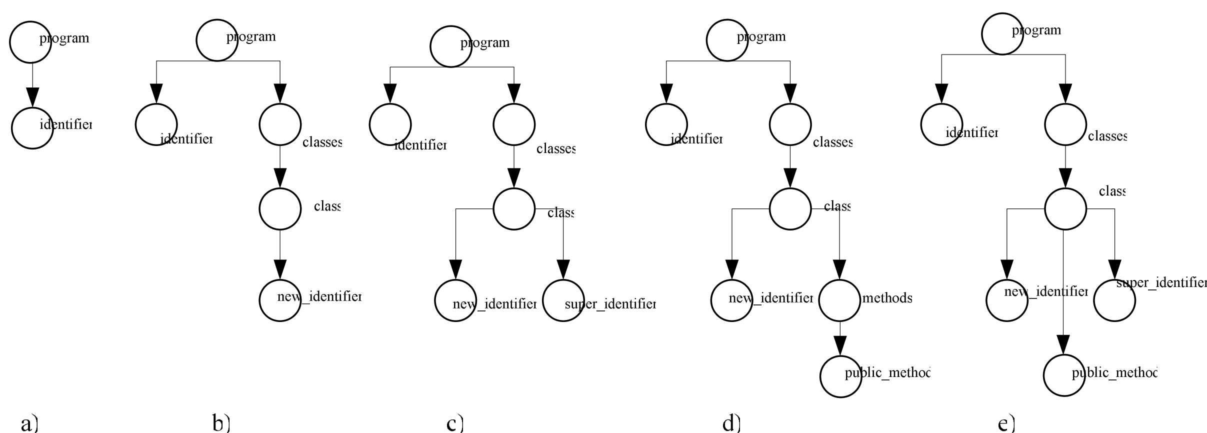


Рис. 5 Первые пять деревьев вывода, полученные в результате раскрытия продукционных правил грамматики языка L_{oo}

Программы, построенные для этих деревьев вывода, приведены в Таб. 5.

Таб. 5. Результаты генерации тестов для языка L_{oo}

№	Тексты на языке Loo
1.	`program p1;`
2.	`program p1;` `class C1{}`
3.	*Программа не будет построена, так как нарушаются контекстные условия (5),(6).*
4.	`program p1;` `class C1{ public m1{} }`

№	**Тексты на языке Loo**
5.	*Программа не будет построена, так как нарушаются контекстные условия (5),(6).*

Выводы к примеру 2

Для того чтобы описать семантику языка L_{oo} в виде атрибутных грамматик, пришлось описать несколько таблиц имен (таблицу имен классов и таблицу имен методов) и обеспечить работу с ними.

Пришлось написать восемь пользовательских функций, сложность некоторых из которых выходит за пределы написания стандартных функций для работы с таблицей имен.

Для того чтобы описать параметризованное контекстное условие, описывающее изменение области видимости метода в зависимости от модификатора, было искусственно дублировано описание нетерминала *method*, появились *private_method* и *public_method*.

Построение двух из первых пяти деревьев было прервано из-за нарушения контекстных условий. В примере 2 были построены только три семантически корректных теста. Однако все они оказались неинтересными с точки зрения проверки семантики, так как в этих программах не присутствует никаких семантических зависимостей.

1.4 Анализ результатов генерации тестов с использованием классических атрибутных грамматик

Проанализируем причины неудовлетворительных результатов генерации с использованием атрибутных грамматик, полученных в примерах 1 и 2.

Основная причина неудачи кроется в том, что при генерации мы руководствовались только синтаксической информацией. Семантические ограничения при этом использовались только для фильтрации. В процессе генерации некоторые контекстные условия не были выполнены из-за того, что какие-то необходимые элементы еще не построены или были построены

не в том виде. Описание статической семантики в виде классических атрибутных грамматик не позволяет генератору *a priori* определить контекстные зависимости между элементами и установить порядок генерации, т.е. описание семантики в виде классических атрибутных грамматик не является конструктивным.

Кроме того, в примере 2 обнаружены дополнительные проблемы, связанные с появлением объектно-ориентированных свойств.

В Таб. 6 рассматриваются изменения, которым подвергается атрибутная грамматика в результате появления объектно-ориентированных свойств. В самом правом столбце приводятся номера правил статической семантики из Таб. 22 в качестве примеров, иллюстрирующих первые три столбца таблицы.

Таб. 6. Влияние ОО свойств на атрибутную грамматику

ОО свойство	Влияние на семантику	Изменения в АГ	Номера правил
Наследование	Расширение области возможного месторасположения декларации объекта.	Усложнение работы с таблицами имен.	(2), (3), (8), (13), (14), (16), (17)
	Изменение области видимости декларации.		
Инкапсуляция	Параметризация понятия области видимости.	Появление условий, при которых возможна проверка контекстных условий.	(24)

ОО свойство	Влияние на семантику	Изменения в АГ	Номера правил
Полиморфизм	Усложнение правил неявного приведения типов.	Появление сложных пользовательских функций.	(6), (8)
	Появление зависимостей между контекстными условиями.	Появление контекстных условий, влияющих на возможность проверки других контекстных условий.	(24)

Описание атрибутной грамматики в примере 2 значительно сложнее, чем в примере 1. Увеличивается количество таблиц имен, в пользовательских функциях описываются сложные семантические действия (например: поиск циклической зависимости в графе наследования).

Возникновение зависимостей между контекстными условиями требует создания новых структур данных для хранения результатов выполнения контекстных условий. Так, например, для *private* и *public* методов предусмотрены различные таблицы имен.

В примере 2 хорошо виден недостаток классических атрибутных грамматик (отмеченный во многих исследованиях), связанный с высокой степенью дублируемости информации: правила (9), (10), (11), (12), (18), (19), (20), (21), (22) направлены на передачу сверху вниз по дереву атрибута, значением которого является идентификатор описываемого класса, и фактически дублируют друг друга.

С увеличением сложности входного языка возрастает сложность описания атрибутной грамматики для него. В результате трудоемкость по описанию

входных данных для генератора становится сравнимой с ценой затраченных усилий на написание анализатора контекстных условий.

По результатам проведенного анализа можно сделать следующие выводы:

1. организация генерации тестов для анализатора контекстных условий с использованием описания семантики с помощью атрибутных грамматик практически нецелесообразно, так как:
    a. атрибутные грамматики для объектно-ориентированных языков требуют написания большого количества пользовательских функций;
    b. в атрибутных грамматиках семантически зависимые синтаксические конструкции указываются неявно;
    c. атрибутные грамматики не позволяют локализовать формальное описание каждого неформального контекстного условия в пределах одного формального правила.
2. для построения целенаправленной генерации тестов для анализатора контекстных условий требуется описать правила статической семантики входного языка в конструктивной манере.

В настоящей работе предлагается способ конструктивного описания правил статической семантики формальных языков, который позволяет *минимизировать количество предикатов, и увеличить число инструкций, указывающих генератору какие элементы, в каком виде и когда строить*. Данный способ получил название *конструктивная грамматика.*

При использовании конструктивной грамматики для описания правил статической семантики отпадает необходимость написания пользовательских функций и дублирования информации, описание статической семантики отделяется от синтаксиса. В результате при необходимости изменить описание семантики становиться значительно проще.

Предложенный способ описания входных данных дает возможность создать семантически управляемый алгоритм генерации тестовых текстов, что позволяет систематически строить семантически корректные тесты.

В следующей главе формально вводится понятие конструктивных грамматик, приводится описание их формальных свойств и рассматриваются примеры.

2 Конструктивное описание статической семантики формальных языков

2.1 Неформальное описание идеи

В начале этой главы прежде, чем перейти к формальному описанию, предлагаемого конструктивного способа задания контекстных условий, приведем основанные на здравом смысле неформальные рассуждения, которые легли в его основу.

Напомним, что целью данной работы является получение целенаправленного метода автоматической генерации текстов, удовлетворяющих контекстным условиям соответствующего языка. Целенаправленность метода заключается в следующем, сгенерированные тексты должны быть такими, чтобы каждый из них можно было использовать в качестве входных данных для теста, направленного на проверку определенного аспекта семантического анализатора транслятора.

Единственный способ систематического построения всех возможных текстов на некотором языке основан на последовательном раскрытии продукционных правил грамматики этого языка. В связи с этим явное указание зависимости контекстных условий от грамматических символов позволило бы направлять процесс раскрытия продукционных правил с целью получения входных данных, удовлетворяющих желаемым контекстным условиям.

Рассмотрим некоторый синтаксически и семантически корректный текст p, написанный на языке L, порожденном однозначной контекстно-свободной грамматикой G. Тексту p взаимно однозначно соответствует дерево вывода t.

Напомним, что *деревом вывода* называется дерево, каждый узел которого помечен грамматическим символом или ε, символом соответствующим пустой последовательности токенов. Внутренний узел и его дочерние узлы

соответствуют продукции, причем узел соответствует левой части продукции, а потомки – правой [28].

Предположим, что для разбора текста p и проверки его семантической корректности использовалась атрибутная грамматика, и каждой вершине дерева t, соответствует множество атрибутов. Некоторые атрибуты соединены друг с другом дугами. Атрибуты и дуги их соединяющие образуют *граф атрибутной зависимости* [12]. Дуги, соединяющие вершины дерева t, отражают синтаксические зависимости, а дуги графа атрибутной зависимости – контекстные зависимости, присутствующие в тексте p.

Построим еще один ориентированный граф (см. Рис. 2). Вершинами этого графа будут вершины рассматриваемого дерева t, а дуги будут соединять те вершины, атрибуты которых являются смежными в соответствующем графе атрибутной зависимости, причем направление дуг в новом графе должно совпадать с направлением соответствующих дуг графа атрибутной зависимости.

Вспомним о том, что каждая вершина дерева t помечается грамматическим символом входного языка. Построенный таким образом новый граф явно задает встречающиеся в тексте p семантические зависимости между грамматическими символами подобно тому, как дерево t задает синтаксические зависимости.

Введем специальные названия для пары смежных вершин, нового графа. Для любой дуги d этого графа, соответствующей некоторому контекстному условию R, будем называть *источником* R вершину, из которой исходит эта дуга, и *целью* R – вершину, в которую дуга d направлена[9].

Заметим, что не для каждого контекстного условия языка L в дереве t будут присутствовать источник и цель. Рассмотрим множество деревьев $\{t_i\}$, соответствующее множеству всех возможных текстов $\{p_i\}$ на языке L. Тогда существует множество графов $\{g_i\}$, где g_i – это граф семантической

---

[9] Очевидно, что в дереве может существовать несколько пар вершин, являющихся источником и целью одного и того же контекстного условия. Заметим также, что вершина, являющаяся источником по отношению к одной вершине, может одновременно быть целью по отношению к другой вершине.

зависимости между вершинами дерева t_i. Среди всех дуг множества графов $\{g_i\}$ для каждого контекстного условия R_j языка L обязательно найдется хотя бы одна дуга, соединяющая пару вершин (источник и цель R_j) какого-то дерева $t' \in \{t_i\}$. Иначе соответствующее правило может быть исключено из множества контекстных условий языка L.

Таким образом, каждому контекстному условию языка L можно поставить в соответствие пару предикатов, заданных на множестве вершин $\forall t \in \{t_i\}$. Эти предикаты описывают требования на тип и расположение в дереве вершин, которые являются источником и целью, соответствующего контекстного условия. Заметим, что для описания каждого контекстного условия пары предикатов еще недостаточно, так как граф семантической зависимости между вершинами дерева не отражает зависимости между атрибутами.

Рассмотрим атрибуты, являющиеся вершинами графа атрибутной зависимости. Значение каждого атрибута дерева t определяется состоянием или видом некоторого поддерева t. Следовательно, каждому контекстному условию можно поставить в соответствие еще по три предиката: по одному для описания каждого атрибута и еще один для описания связи между парой смежных атрибутов. Таким образом, множество контекстных условий можно описать в виде системы правил, состоящих из предикатов, заданных на множестве вершин деревьев $\{t_i\}$. Если множество вершин $U(t)$ некоторого дерева t, соответствующего синтаксически корректному тексту, является решением этой системы правил при условии, что каждое правило принимает значение *"истина"*, тогда текст, соответствующий дереву t будет также и семантически корректен.

Если каждому контекстному условию поставить в соответствие правило построения вершин, удовлетворяющих данному контекстному условию, тогда используя эти правила построения и продукционные правила можно организовать целенаправленный процесс создания деревьев, соответствующих синтаксически и семантически корректным текстам.

Приведенный неформальный подход был применен для описания контекстных условий языков Си, Java и XML-схемы для документов IPMP [46]. Проведенные практические исследования доказали справедливость сделанных предположений и позволили выделить несколько типов контекстных условий, позволивших описать все правила статической семантики, указанных языков. Более того, контекстные условия были заданы таким образом, что удалось найти правила построения вершин, им удовлетворяющих.

В следующем параграфе приводится формальное описание контекстных условий и предикатов, входящих в состав контекстных условий. Далее в этой главе дается описание алгоритма построения деревьев, соответствующих синтаксически и семантически корректным текстам, приводятся критерии полноты построенных наборов текстов и правила создания вершин, удовлетворяющих ранее заданным формальным контекстным условиям. Кроме того, в последнем параграфе описывается как, используя предложенный способ задания контекстных условий и алгоритм генерации синтаксически и семантически корректных текстов, построить тексты, нарушающие некоторые указанные контекстные условия.

2.2 Конструктивное описание семантики

Конструктивная грамматика – это формализм, который позволяет описывать подмножество языка, заданного контекстно-свободной грамматикой, удовлетворяющее требованиям контекстных условий. Для того чтобы получить конструктивную грамматику (КГ) необходимо дополнить контекстно-свободную грамматику правилами статической семантики.

Определение 2. *Конструктивной грамматикой* называется пара *(G, C)*, где *G* – это однозначная КС-грамматика, а *C* – это множество контекстных условий.

Прежде, чем дать определение контекстным условиям в конструктивной грамматике введем некоторые понятия.

Пусть задана однозначная КС-грамматика $G = (V, N, S, K)$, где V – это алфавит терминальных символов, N - это множество нетерминалов, $S \in N$ – стартовый символ, K – это множество продукций (или синтаксических правил).

Обозначим язык, порождаемый грамматикой G, через $L(G)$.

Грамматика G выводит, или порождает, цепочки терминальных символов, начиная со стартового символа и неоднократно замещая нетерминалы правыми частями продукций этих нетерминалов. Каждой такой цепочке соответствует *дерево вывода*. И, поскольку, грамматика G однозначная, то каждому дереву вывода соответствует одна цепочка, а цепочке только одно дерево вывода.

Будем говорить, что вершина *u* дерева вывода имеет *тип A*, если она помечена грамматическим символом A.

Определение 3. *Именованным деревом вывода* называется дерево вывода, каждый узел которого, кроме корневого узла, имеет имя, удовлетворяющее *правилам именования*.

Правила именования:

1. Все узлы, имеющие общий родительский узел, должны иметь уникальные имена.
2. Любые два узла, принадлежащие одному дереву или двум разным деревьям, порожденным одной и той же грамматикой, должны иметь одинаковые имена, если они соответствуют одному и тому же вхождению символа в правую часть одного и того же продукционного правила.
3. Любые два узла, принадлежащие одному дереву или двум разным деревьям, порожденным одной и той же грамматикой, должны иметь одинаковые имена, если они соответствуют альтернативам из одного и того же списка альтернатив.

В следующем параграфе будет дано подробное описание метода именования деревьев вывода.

Таким образом, можно сказать, что грамматике G соответствует множество именованных деревьев вывода. Обозначим это множество, объединенное с множеством всех поддеревьев, через $M(G)$. Далее везде для краткости для обозначения *именованного дерева вывода* будет использоваться термин *дерево*.

Каждая вершина любого дерева $t \in M(G)$ помечена каким-то грамматическим символом грамматики G. Пусть $U(t)$ – множество вершин дерева t. В дереве t могут присутствовать несколько вершин, помеченных одним и тем же символом.

Определение 4. Будем называть *вхождением* символа $X \in V \cup N \cup \{S\}$ в дерево вершину дерева t, помеченную грамматическим символом X.

Определение 5. Будем называть *положением P* предикат $P(u)$, где u – это вершина дерева t, $u \in U(t)$.

Определение 6. Будем называть *вхождением* символа *X с положением P* вершину u дерева t, помеченную грамматическим символом $X \in V \cup N \cup \{S\}$ и имеющую положение P, т.е. $P(u)$.

Тогда контекстные условия задаются на множестве вершин деревьев, соответствующих грамматике G, и принимают один из трех видов.

Определение 7. *Контекстными условиями* в конструктивной грамматике называются *выражения* вида (1)-(3).

Таб. 7 Типы контекстных условий в конструктивной грамматике

Типы контекстных условий	**Формулы, описывающие контекстные условия**	
Правило разрешимое в пределах одного узла (one-node)	$R(u) \equiv P_{trg}(u) \Rightarrow F(u, u)$	(1)
Многие-ко-многим (many-to-many)	$R(u, v) \equiv (u \neq v \wedge P_{trg}(u) \wedge P_{src}(u, v) \Rightarrow F(u, v))$	(2)
Один-ко-многим (one-to-many)	$R(u) \equiv (P_{trg}(u) \Rightarrow \exists v\ (u \neq v \wedge P_{src}(u, v) \wedge F(u, v)))$	(3)

Здесь u, v – это вершины дерева $t \in M(G)$, $P_{trg}(u)$, $P_{src}(u, v)$ – это предикаты, описывающие положения и типы вершин u и v, к которым должно применяться данное контекстное условие, $F(u, v)$ – это предикат, описывающий требования на поддеревья, с корнями в вершинах u и v соответственно. В общем виде положение вершины v может зависеть от вершины u, поэтому положение v описывается двуместным предикатом $P_{src}(u, v)$, который для некоторых условий может вырождаться в одноместный предикат $P_{src}(v)$, когда положение v не зависит от вершины u.

Формула (1) позволяет описывать контекстные условия для одной вершины, удовлетворяющей условию $P_{trg}(u)$, накладывая при этом условие $F(u, u)$ на вид поддерева, корнем которого является вершина u. Примером такого контекстного условия является требование на отсутствие модификатора *final* в декларации абстрактных методов в языке Java. В данном контекстном условии предикат положения $P_{trg}(u)$ должен быть истинен для любой вершины u, соответствующей декларации метода, а в заголовке метода в поддереве с корнем в вершине u не должны одновременно присутствовать вершины, соответствующие модификаторам *final* и *abstract*,

это требование, должно описываться предикатом $F(u, u)$. Таким образом, контекстное условие примет вид $R(u)\equiv\forall t\in M(G)\ \forall u\in U(t)$ *“вершина u помечена нетерминалом, соответствующим декларации метода”* $\Rightarrow$ *“в поддереве с корнем в u в ближайшей дочерней вершине, соответствующей сигнатуре метода, одновременно нет дочерних вершин, соответствующих модификаторам abstract и final”*.

При помощи формулы (2) можно задавать контекстные зависимости между всеми вершинами, удовлетворяющими условиям $P_{trg}(u)$, $P_{src}(u, v)$, за исключением тождественных, и накладывать требования $F(u, v)$ на вид поддеревьев с корнями в *u* и *v* соответственно. Такие контекстные условия называются *многие-ко-многим*. Примером такого контекстного условия является требование на уникальность имен переменных. Контекстное условие примет вид $R(u, v)\equiv\ \forall\ t\in M(G)\ \forall u\in U(t)\ \forall v\in U(t)\ \ (u\neq v\ \wedge$ *“вершина u помечена нетерминалом, соответствующим декларации переменной”* $\wedge$ *“вершина v помечена нетерминалом, соответствующим декларации переменной”* $\Rightarrow$ *“поддерево с корнем в дочерней вершине вершины u, соответствующей имени переменной, декларированной в u, не равно поддереву с корнем в дочерней вершине вершины v, соответствующей имени переменной, декларированной в v”).*

Формулу (3) следует использовать для описания контекстных условий типа *один-ко-многим*. Примером такого типа контекстных условий может служить требование о существовании декларации для каждой переменной. В общем виде для каждой вершины *u*, удовлетворяющей некоторым условиям $P_{trg}(u)$, должна существовать вершина *v*, в свою очередь, удовлетворяющая некоторым другим условиям $P_{src}(u, v)$, и тогда соответствующие поддеревья с корнями в вершинах *u* и *v* должны удовлетворять $F(u, v)$. Контекстное условие примет вид $R(u, v)\equiv\ \forall t\in M(G)\ \forall u\in U(t)$ (*“вершина u помечена нетерминальным символом, соответствующим использованию переменной”* $\Rightarrow\exists v\in U(t)\ \ (u\neq v\ \ \wedge$ *“вершина v помечена нетерминальным символом,*

соответствующим декларации переменной” ∧ *“поддерево с корнем в дочерней вершине вершины u, соответствующей имени переменной, соответствующей u, равно поддереву с корнем в дочерней вершине вершины v, соответствующей имени переменной, декларированной в v”)).*

Заметим, что приведенные формулы (2), (3) можно свести к общему виду (1) контекстных условий, выделив общую вершину предка для *u* и *v* и переписав соответствующие предикаты. Поэтому далее при обсуждении примеров или свойств контекстных условий часто будет использоваться формула (1).

В результате практических исследований и проведенного анализа реальных языков программирования, таких, как Java и C, оказалось, что все контекстные условия принимают один из указанных видов.

Предикаты *F(u, v)*, задающие требования на вид поддеревьев в контекстном условии, представляются одной из формул, приведенных в Таб. 8 и Таб. 9.

Таб. 8 Виды предиката *F(u, v)*

Тип	**Формулы F(u, v)**	**Описание**
equal	$(\exists x \in U(T^u)$ $\wedge P_{trg_subtree}(x)$ $\wedge \exists y \in U(T^v)$ $\wedge P_{src_subtree}(y))$ $\Rightarrow (T^x == T^y)$	P- формулы, описывающие положение вершин, T^x- поддерево с корнем в вершине *x*, T^y- поддерево с корнем в вершине *y*, $U(T^u)$-множество вершин поддерева с корнем в вершине *u*, $U(T^v)$-множество вершин поддерева с корнем в вершине v.

Тип	Формулы F(u, v)	Описание
unequal	$(\exists x \in U(T^u)$ $\wedge P_{trg_subtree}(x)$ $\wedge\ \exists y \in\ U(T^v)$ $\wedge\ P_{src_subtree}(y)) \Rightarrow$ $(T^x \sim= T^y)$	P- формулы, описывающие положение вершин, T^x- поддерево с корнем в вершине x, T^y- поддерево с корнем в вершине y, $U(T^u)$-множество вершин поддерева с корнем в вершине u, $U(T^v)$-множество вершин поддерева с корнем в вершине v.
present	$(\exists x \in U(T^u)$ $\wedge\ P_{trg_subtree}(x))$ $\Rightarrow (\exists y \in\ U(T^v)$ $\wedge\ P_{src_subtree}(y))$	Если в поддереве с корнем в вершине u существует вершина x, тогда в поддереве с корнем в v должна существовать вершина y, $U(T^u)$-множество вершин поддерева с корнем в вершине u, $U(T^v)$-множество вершин поддерева с корнем в вершине v.
absent	$(\exists x \in U(T^u)$ $\wedge P_{trg_subtree}(x))$ $\Rightarrow (\sim\exists y \in\ U(T^v)$ $\wedge\ P_{src_subtree}(y))$	Если в поддереве с корнем в вершине u существует вершина x, тогда в поддереве с корнем в v не должна существовать вершина y, $U(T^u)$-множество вершин поддерева с корнем в вершине u, $U(T^v)$-множество вершин поддерева с корнем в вершине v.

Для языков программирования существенное количество контекстных условий связано с проверкой приводимости типов выражений. Для того чтобы описывать подобные контекстные условия будем считать, что у всех

узлов, соответствующих выражениям есть дочерняя вершина, являющаяся корнем поддерева, описывающего тип данного выражения. В следующей таблице приводится вид предиката *F(u, v)* для контекстных условий проверки приводимости типов выражений. Приводимости типов и описанию контекстных условий, связанных с этим, посвящен раздел 4.5 главы 4.

Таб. 9 Виды предиката F(u, v) для контекстных условий проверки приводимости типов

Тип	**Формула F(u, v)**	**Описание**
compatible	$(\exists x \in U(T^u) \wedge P_{trg_subtree}(x) \wedge \exists y \in U(T^v) \wedge P_{src_subtree}(y)) \Rightarrow (type(x) \rightarrow type(y))$	*type(x)*- тип выражения *x*, *type(y)*- тип выражения y, знак $\rightarrow$ означает приводимость типа, стоящего слева от этого знака к типу стоящему справа от этого знака.

Данное описание основано на эвристических рассуждениях и результатах практических исследований. В работе не приводится строгого доказательства того, что приведенных формул достаточно для описания контекстных условий формальных языков, порожденных кс-грамматиками. В связи с этим дальнейшие рассуждения будут проводиться в предположении, что вообще говоря, предикат *F(u, v)* может принимать какой-то другой более сложный вид заданный пользователем. В этом случае пользователь сам должен описать правила построения таких вершин, которые бы удовлетворяли заданным им требованиям. Тип такого предиката *F(u, v)* получил название *custom*. На практике предикат *F(u, v)* типа *custom* потребовался только для описания требования уникальности сигнатур методов одного класса и для описания требования соответствия вызова метода сигнатуре этого метода.

Заметим, что в случае контекстного условия вида (1) типа *one-node*, задаваемого для одной вершины, переменные *u* и *v* в формулах в Таб. 8 обозначают одну и ту же вершину: *u=v*.

Введем два понятия.

Рассмотрим контекстное условие *R*.

Определение 8. Будем называть *источником (source) правила R* вершину, удовлетворяющую требованиям предикатов положения *R* при подстановке

- вместо независимой переменной *u*, если *R* вида (1), т.е. $(P_{trg}(u) \wedge \exists x, y \in U(T^u)(P_{trg_subtree}(x) \wedge P_{src_subtree}(y)))$;
- вместо независимой переменной *v*, если *R* вида (2), типа *many-to-many,* т.е. $(P_{src}(v) \wedge \exists y \in U(T^v)\ P_{src_subtree}(y))$;
- вместо зависимой переменной *v*, если *R* вида (3), типа *one-to-many*, т.е. $(P_{src}(u, v) \wedge \exists y \in U(T^v)\ P_{src_subtree}(y))$.

Определение 9. Будем называть *целью (target) правила R* вершину, удовлетворяющую требованиям *R* при подстановке вместо независимой переменной *u* вне зависимости от вида *R*[10], т.е. $(P_{trg}(u) \wedge \exists x, y \in U(T^u) \wedge P_{trg_subtree}(x) \wedge P_{src_subtree}(y))$.

Положение *P* вершины *u* в дереве *t* будем задавать при помощи формул, состоящих из следующих символов:

1) *u, v,…, x, y,…* – переменные, множество значений которых совпадает со множеством вершин некоторого фиксированного дерева *t*;
2) A, B, C… - переменные, множество значений которых совпадает со множеством грамматических символов рассматриваемой грамматики;
3) a, b, c… – переменные, множество значений которых совпадает со множеством имен вершин дерева *t*;
4) R1, R2… – переменные, множество значений которых совпадает со множеством идентификаторов контекстных условий, заданных в виде контекстных условий для конструктивной грамматики (1)-(3).

Комбинации данных символов представляют собой атомарные одноместные формулы:

---

[10] Для правил вида (1) источник и цель совпадают.

Таб. 10 Атомарные одноместные формулы в КГ

№	**Атомарные одноместные предикаты, описывающие положение вершин**	**Предикат истинен, если**
1.	$A(u)$	вершина u помечена грамматическим символом A.
2.	$\varepsilon(u)$	вершина u помечена ε.
3.	$b(u)$	вершина u имеет имя b.

Введем атомарные двуместные формулы:

Таб. 11 Атомарные двуместные формулы в КГ

№	**Атомарные двуместные предикаты, описывающие положение вершин**	**Предикат истинен, если**
1.	$parent(u, v)$	v - родительская вершина вершины u.
2.	$\hat{}B(u, v)$	v - ближайшая по отношению к вершине u вершина-предок, помеченная грамматическим символом B.
3.	$<n(u, v)$	v - вершина находится на глубине меньше n от вершины u.
4.	$<=n(u, v)$	v - вершина находится на глубине не больше n от вершины u.
5.	$>n(u, v)$	v - вершина находится на глубине больше n от вершины u.
6.	$>=n(u, v)$	v - вершина находится на глубине не меньше n от вершины u.
7.	$n(u, v)$	v - вершина находится на глубине n от вершины u.
8.	$00(u, v)$	v - вершина находится на любой глубине от вершины u.

№	Атомарные двуместные предикаты, описывающие положение вершин	Предикат истинен, если
9.	*precede(u, v)*	*v* – вершина, которая при обходе дерева слева направо будет пройдена перед посещением вершины *u*

Комбинируя введенные атомарные предикаты и их отрицания можно получить другие формулы, применяя правило (4).

$$P(x) \equiv \forall x_0,..., x_n \bigwedge_{i=1,k} P_i(x) \bigwedge_{j=0,n} \sim P_j(x, x_j) \qquad (4)$$

Рассмотрим пример, построения предиката положения по правилу (4).

Пример 3

Опишем предикат положения вершины типа *A* с именем *b*, не имеющей вершин-предков типа *B*. Соответствующий предикат будет выглядеть следующим образом: $P(u) \equiv A(u) \wedge b(u) \wedge \forall x \sim\wedge B(u, x)$.

Все остальные формулы, задающие требования на положение и тип вершин, строятся из атомарных формул и формул, полученных по правилу (4), с помощью применения правил (5), (6), где k>0, P_i - атомарные предикаты из Таб. 10 и Таб. 11 или предикат, полученный по формуле (4).

$$P(x_k) \equiv \exists x_0,...,x_{k-1} P_0(x_0) \wedge \bigwedge_{i=1,k} P_i(x_{i-1}, x_i) \bigwedge_{j=0,n} P_j(x_i) \qquad (5)$$

$$P(u, x_k) \equiv \exists x_0,...,x_{k-1} P_0(u, x_0) \bigwedge_{j=0,n} P_j(x_0) \wedge \bigwedge_{i=1,k} P_i(x_{i-1}, x_i) \bigwedge_{j=0,n} P_j(x_i) \qquad (6)$$

Рассмотрим пример, в котором предикат положения имеет вид (5).

Пример 4

Ниже приведен фрагмент грамматики языка Java.

```
RootNode ::= PackageNode+

PackageNode ::= PackageDeclaration PackageNode*

PackageDeclaration ::= Identifier ";"

FullTypeName ::= PackageName SimpleTypeName

PackageName ::= Identifier | (PackageName  "." Identifier)
```

В этот фрагмент включены следующие продукционные правила:

- *RootNode* является стартовым правилом;
- *PackageNode* задает пакет Java;
- *PackageDeclaration* соответствует декларации пакета Java;
- *FullTypeName* соответствует квалифицированному имени типа;
- *PackageName* соответствует имени пакета.

Опишем предикаты положения источника и цели контекстного условия: *“имя пакета верхнего уровня должно быть задано”*.

В программе на языке Java имя одного пакета может использоваться много раз, следовательно, данное правило типа *one-to-many*. Причем целью данного правила будет первое имя пакета, встречающееся в квалифицированном имени типа, а источником – декларация пакета, не являющегося подпакетом другого пакета. Таким образом, предикат положения цели $P_{trg}(u)$ примет вид:

$$P_{trg}(u) \equiv \exists x_0\, FullTypeName(x_0) \wedge 1.PackageName(x_0, u)$$

А предикат положения источника $P_{src}(v)$ примет такой вид:

$$P_{src}(v) \equiv \exists x_0\, RootNode(x_0) \wedge \exists x_1\, 1.PackageNode(x_0, x_1) \wedge 1.PackageDeclaration(x_0, v).$$

Рассмотрим пример, в котором предикат положения имеет вид (6).

Пример 5

Ниже приведен фрагмент грамматики языка Java.

```
ClassDeclaration ::= Modifiers* Identifier Super? Interfaces*
ClassBody
ClassBody ::= ClassBodyDeclaration*
ClassBodyDeclaration ::= ConstructorDeclaration
| MemberDeclaration
ConstructorDeclaration ::= Modifiers* Identifier  Parameters*
ConstructorBody
```

В этот фрагмент включены следующие продукционные правила:

- *ClassDeclaration* задает декларацию класса;
- *ClassBody* соответствует телу класса;
- *ClassBodyDeclaration* задает либо конструктор класса, либо члены класса такие, как поля или методы;
- *ConstructorDeclaration* соответствует декларации конструктора.

Опишем предикаты положения источника и цели контекстного условия: *“имя конструктора должно совпадать с именем класса, в котором он задан”*.

В одном классе может присутствовать несколько конструкторов, следовательно, данное контекстное условие имеет тип *one-to-many*. Целью в нем будет вершина, помеченная нетерминалом, соответствующим декларации конструктора, а источником будет ближайшая к цели вершина-предок, соответствующая декларации класса. Таким образом, предикат цели $P_{trg}(u)$ примет вид:

$$P_{trg}(u) \equiv ConstructorDeclaration(u)$$

А предикат положения источника $P_{src}(v)$ примет такой вид:

$$P_{src}(v) \equiv \exists x_0\ ConstructorDeclaration(x_0) \wedge \hat{}\ ClassDeclaration(x_0, v).$$

Продемонстрируем, как предложенный метод задания контекстных условий применяется к формальным языкам. Рассмотрим пример.

Пример 6

Напомним, что EBNF (расширенная BNF, форма Бэкуса-Наура, нормальная форма Бэкуса) является общепринятым способом описания формальных грамматик[11].

Пусть у нас имеется язык *L*, грамматика которого задана при помощи EBNF.

```
program ::= “ program ” identifier “ ;” (declaration)*
declaration ::= (“ var ” new_var_identifier )
            |(“ syn ” new_syn_identifier “ = ” var_identifier)“ ;”
new_var_identifier ::=identifier
new_syn_identifier ::=identifier
var_identifier ::= identifier
identifier ::= (a|b|c|…|z)+ (0|1|2|3|4|5|6|7|8|9)*
```

Язык *L* позволяет вводить переменные, задавая их имена в виде идентификаторов, и синонимы для уже заданных переменных. При этом имена всех переменных и синонимов должны быть различны.

Опишем множество контекстных условий языка *L* в терминах конструктивных грамматик.

Рассмотрим первое неформальное контекстное условие: *«имена всех переменных и синонимов должны быть различны»*. Данному неформальному условию соответствует несколько формальных контекстных условий типа *many-to-many*:

---

[11] Выражение, стоящее справа от символа ::=, называется правой частью правила. Правая часть правила EBNF является регулярным выражением над алфавитом $N \cup T \cup A$, где N – алфавит (множество) нетерминальных символов (нетерминалов), T – алфавит (множество) терминальных символов (терминалов), A – алфавит (множество) символов действия (СД). Множество регулярных выражений над алфавитом X обозначим $REG(X)$. Регулярное выражение $r \in REG(X)$ порождает некоторое множество цепочек символов алфавита X. Множество $REG(X)$ и порождаемые его элементами цепочки определим следующим образом:

- $a \in REG(X)$ $\forall a \in X$. Выражение a порождает цепочку, состоящую из одного символа a;
- если $a \in REG(X)$, $b \in REG(X)$, то $a\ b \in REG(X)$. Выражение $a\ b$ порождает множество цепочек, получаемых конкатенацией некоторой цепочки, порождаемой выражением a и некоторой цепочки, порождаемой b;
- если $a \in REG(X)$, $b \in REG(X)$, то $(a \mid b) \in REG(X)$. Выражение $(a \mid b)$ порождает объединение множеств цепочек, описываемых a и b;
- если $a \in REG(X)$, то $(a)^* \in REG(X)$. Выражение $(a)^*$ называется итерацией a и порождает множество цепочек, являющихся конкатенацией произвольного (нулевого или положительного) количества цепочек, порождаемых выражением a;
- если $a \in REG(X)$, то $(a)+ \in REG(X)$. Выражение $(a)+$ называется положительной итерацией a. Положительная итерация a отличается от итерации a тем, что порождает конкатенацию положительного числа цепочек;

если $a \in REG(X)$, то $(a)? \in REG(X)$. Выражение $(a)?$ порождает цепочку, порождаемую выражением a, или пустую цепочку (т.е. цепочку нулевой длины).

– во-первых, это условие означает, что во всех деревьях, соответствующих грамматике *L*, поддеревья с корнями в вершинах, помеченных грамматическим символом *identifier* и являющихся потомками вершин, помеченных символами *new_var_identifier* или *new_syn_identifier*, должны быть не эквивалентны, т.е.

$R_1\ (u, v) \equiv (u \neq v \wedge new_var_identifier(u) \wedge new_syn_identifier(v)$

$\Rightarrow (\exists x \in U(T^u)\ 1.identifier(u, x) \wedge \exists y \in U(T^v)\ 1.identifier(v, y)$

$\Rightarrow (T^x \sim= T^y)))$

– во-вторых, это условие означает, что во всех деревьях, соответствующих грамматике *L*, поддеревья с корнями в вершинах, помеченных грамматическим символом *identifier* и являющихся потомками вершин, помеченных символами *new_var_identifier*, должны быть не эквивалентны, т.е.

$R_2\ (u, v) \equiv (u \neq v \wedge new_var_identifier(u) \wedge new_var_identifier(v)$

$\Rightarrow (\exists x \in U(T^u)\ 1.identifier(u, x) \wedge \exists y \in U(T^v)\ 1.identifier(v, y)$

$\Rightarrow (T^x \sim= T^y)))$

– в-третьих, это условие означает, что во всех деревьях, соответствующих грамматике *L*, поддеревья с корнями в вершинах, помеченных грамматическим символом *identifier* и являющихся потомками вершин, помеченных символом *new_syn_identifier*, должны быть не эквивалентны, т.е.

$R_3\ (u, v) \equiv (u \neq v \wedge new_syn_identifier(u) \wedge new_syn_identifier(v)$

$\Rightarrow (\exists x \in U(T^u)\ 1.identifier(u, x) \wedge \exists y \in U(T^v)\ 1.identifier(v, y)$

$\Rightarrow (T^x \sim= T^y)))$

Кроме того, существует еще одно неформальное контекстное условие для языка *L*, а именно: *«синоним можно задавать только для заданной переменной»*. Данному неформальному требованию соответствует одно формальное контекстное условие типа *one-to-many*:

$R_4(u) \equiv (var_identifier(u) \Rightarrow (\exists v \in U(t)\ u \neq v\ \wedge new_var_identifier(v) \wedge$

$\exists x \in U(T^u)\ 1.identifier(u, x) \wedge \exists y \in U(T^v)\ 1.identifier(v, y) \Rightarrow (T^x == T^y))),$

которое означает, что для каждого вхождения символа *var_identifier* в дерево *t* должно существовать хотя бы одно вхождение символа *new_var_identifier* в дерево *t*.

Для того чтобы задавать контекстные условия с учетом предложенной концепции, но в более удобном виде разработан язык SRL (Semantic Relation Language)[47][54][12]. Язык SRL представляет собой сокращенную запись, приведенного ранее формализма.

При задании контекстных условий на языке SRL каждому контекстному условию присваивается уникальный идентификатор и указывается тип контекстного условия (*one-to-many, many-to-many, one-node*). Затем после ключевого слова *target* задается предикат положения $P_{trg}(u)$. Причем указываются только имена атомарных предикатов, составляющих часть $P_{trg}(u)$, а знаки амперсандов заменяются точками. После этого аналогичным образом в фигурных скобках указывается предикат $P_{trg_subtree}(x)$, являющийся частью $F(u, v)$. Аналогично после ключевого слова *source* задаются предикаты, зависящие от *v*. Вид предиката, являющегося следствием в предикате $F(u, v)$ определяется при помощи ключевого слова *equal*, *unequal*, *absent*, *present*, *compatible* или *custom* (см. Таб. 8 и Таб. 9).

В следующей таблице приведены контекстные условия R_1-R_4 в формальном виде и на языке SRL.

[12] Грамматика языка SRL приводится в Приложении А.

Таб. 12 Описание контекстных условий на языке SRL

Контекстные условия в виде формул	**Контекстные условия на языке SRL**

$R_1(u, v) \equiv (u \neq v$
$\wedge\, new_var_identifier(u)$
$\wedge\, new_syn_identifier(v)$
$\Rightarrow (\exists x \in U(T^u)\, 1.identifier(u, x)$
$\wedge\, \exists y \in U(T^v)\, 1.identifier(v, y)$
$\Rightarrow (T^x \sim= T^y)))$
	many-to-many relation R1
	{
	unequal
	target new_var_identifier
	{ 1.identifier }
	source new_syn_identifier
	{ 1.identifier }
	}

$R_2(u, v) \equiv (u \neq v$
$\wedge\, new_var_identifier(u)$
$\wedge\, new_var_identifier(v)$
$\Rightarrow (\exists x \in U(T^u)\, 1.identifier(u, x)$
$\wedge\, \exists y \in U(T^v)\, 1.identifier(v, y)$
$\Rightarrow (T^x \sim= T^y)))$
	many-to-many relation R2
	{
	unequal
	target new_var_identifier
	{ 1.identifier }
	source new_var_identifier
	{ 1.identifier }
	}

$R_3(u, v) \equiv (u \neq v$
$\wedge\, new_syn_identifier(u)$
$\wedge\, new_syn_identifier(v)$
$\Rightarrow (\exists x \in U(T^u)\, 1.identifier(u, x) \wedge$
$\exists y \in U(T^v)\, 1.identifier(v, y)$
$\Rightarrow (T^x \sim= T^y)))$
	many-to-many relation R3
	{
	unequal
	target new_ syn_identifier
	{ 1.identifier }
	source new_syn_identifier
	{ 1.identifier }
	}

Контекстные условия в виде формул	Контекстные условия на языке SRL
$R4(u) \equiv (var_identifier(u) \Rightarrow (\exists v \in U(t)\ u \neq v$ $\wedge\ new_var_identifier(v)$ $\wedge\ \exists x \in U(Tu)\ 1.identifier(u, x)\ \wedge$ $\exists y \in U(Tv) 1.identifier(v, y) \Rightarrow (Tx == Ty)))$	`one-to-many relation R4 { equal target var_identifier { 1.identifier } source new_var_identifier { 1.identifier } }`

Рассмотренный пример показывает, каким образом при помощи SRL можно компактно записывать контекстные условия для конструктивной грамматики.

Опишем основные особенности конструктивного задания правил статической семантики для задач генерации тестов для анализаторов контекстных условий по сравнению с классическими атрибутными грамматиками:

- нет необходимости писать пользовательские функции[13];
- семантически зависимые синтаксические конструкции указаны явно;
- подход позволяет локализовать формальное описание каждого неформального контекстного условия в пределах одного формального правила.

Для сравнения при использовании атрибутных грамматик в примере 2 значение идентификатора класса *class_id* пришлось передавать вниз по дереву членам класса для того, чтобы была возможность описать контекстные условия.

Ни в примере 1, ни в примере 2 явно не заданы участники семантических зависимостей, контекстные условия описываются рядом с синтаксическими продукциями.

[13] Так, в примере 1 пользователь должен был реализовать функции emptysymtab(), contains(), push(), being_ahead().

В примере 2 правила, передающие идентификатор класса вниз по дереву, являются частью других правил, например, правило 0 является частью правила 0, и изменение одного влечет изменение другого.

В результате разделения описаний синтаксиса и семантики, использования явного задания участников и компактной записи контекстных условий формальное описание правил статической семантики на SRL выглядит прозрачным и позволяет сформулировать критерии, гарантирующие покрытие тестами каждого контекстного условия.

2.3 Критерий покрытия

В тестировании принято исходить из того, что полная (exhaustive) проверка функционирования реальных сложных систем невозможна, так как число возможных тестовых данных или число различных условий функционирования реальной системы либо слишком велико, либо, вообще, бесконечно. Тем не менее, хотя полное тестирование и невозможно, можно и нужно оценивать качество (или *полноту*) тестирования и полноту тестовых наборов. Для этой цели предлагаются различные метрики измерения полноты тестирования. Требования к полноте тестовых наборов обычно называют *критериями тестового покрытия*. Примером критериев тестового покрытия для методов «белого ящика» служит, требование покрытия тестами всех операторов тестируемой программы. Примером критериев тестового покрытия для методов «черного ящика» служит, требование покрытия тестами всех требований к тестируемой программе или всех интерфейсных операций, предоставляемых системой. Помимо оценки качества тестовых наборов критерии тестового покрытия часто используются в качестве условий завершения (stop condition) тестирования (или генерации тестов).

Для широко распространенных методов тестирования известно, какими критериями тестового покрытия следует пользоваться. Так в реальной практике считается, что достаточно высокое качество тестового набора

достигается, если покрыты все операторы программы. Вместе с тем, для систем с повышенными требованиями к надежности приемлемым считается требование покрытия всех переходов. Для ответственных систем требуется проверка всех классов ситуаций, отличающихся друг от друга хотя бы одним условием (критерий MCDC [55]).

Практически все работы по автоматической генерации тестов, так или иначе, рассматривают вопрос критериев покрытия, хотя бы по той причине, что при автоматической генерации тестов всегда нужно располагать условием завершения. Для генерации синтаксических тестов были предложены критерии, которые хорошо показали себя на практике [58]. Для семантических тестов, особенно для негативных семантических тестов, хорошо зарекомендовавших себя критериев неизвестно.

Очевидно, что выбор критерия покрытия является важной задачей обеспечения качественного тестирования любой системы. Кроме того, известно, что, с одной стороны, для ответственных программных систем необходимо пользоваться широким набором приемов тестирования, в том числе критериями тестирования, построенными на метриках по реализации и метриках по спецификациям или моделям тестируемых систем. С другой стороны, также известно, что при использовании адекватных моделей в качестве основы для построения критериев тестового покрытия, в частности, формальных спецификаций языков программирования, также удается получать тестовые наборы высокого качества.

В данной работе, поскольку тесты предлагается генерировать только на основе описания семантики языка (то есть, не на основе реализации транслятора), необходимы критерии покрытия, сформулированные в терминах этого описания. То есть, мы будем строить критерий тестового покрытия, сформулированный в терминах описания семантики языка.

2.3.1 Критерий покрытия для позитивных тестов

Первым кандидатом на роль критерия покрытия является требование покрытия всех контекстных условий.

Определение 10. Текст на входном языке *L* *покрывает контекстное условие* $R'(u) \equiv P'_{trg}(u) \Rightarrow F(u, u)$, если для дерева *t*, соответствующего этому тексту выполняется следующее $\exists u' \in U(t) / P'_{trg}(u)$. Текст на входном языке *L* *покрывает контекстное условие* $R''(u, v) \equiv (u \neq v \wedge P''_{trg}(u) \wedge P''_{src}(u,v) \Rightarrow F(u, v))$, если для дерева *t*, соответствующего этому тексту выполняется следующее $\exists u'', v'' \in U(t) / \ P''_{trg}(u'') \wedge P''_{src}(u'',v'')$.

Рассмотрим пример.

Пример 7.

В языке программирования Java есть контекстное условие о необходимости описать локальную переменную, перед тем как ее использовать. В соответствии с приведенными ранее рассуждениями фрагмент программы на языке Java (7) покрывает данное контекстное условие:

```
{
  int n = 0;
  int m = n;
}
```
(7)

Следующая программа (8) также покрывает данное контекстное условие.

```
{
  int n = 0;
  {
      int m = n;
  }
}
```
(8)

Однако опытный разработчик трансляторов заметит, что второй тест нельзя признать лишним, избыточным, так как он проверяет работу транслятора с тем же контекстным условием, но в другом контексте. В случае (7) мы имели одну сферу видимости, а в случае (8) – две вложенные друг в друга сферы видимости. В таких случаях говорят, что первый случай нацелен на проверку гипотезы о том, ошибка проявляется при обработке данного контекстного условия при принадлежности составляющих правила одной сфере видимости. Второй случай нацелен на другую гипотезу: ошибка проявляется при разнесении компонентов правила в разные (в данном случае во вложенные) сферы видимости.

В примере 7 представлен частный случай, показывающий, что для тестирования анализатора контекстных условий недостаточно построить по одному тексту, реализующему каждое контекстное условие. Необходимо построить множество текстов, в которых соответствующее правило применялось бы в различных синтаксических контекстах. Таким образом, мы можем сформулировать новый критерий тестового покрытия, например, так:

– покрыть все контекстные условия во всех возможных (синтаксически различных) контекстах.

Этот критерий имеет недостаток. «Множество всех возможных контекстов» бесконечно, поэтому мы сможем сравнивать два тестовых набора, но не сможем дать некоторую объективную оценку полноты тестового набора.

Этот недостаток можно сгладить, если «множество всех возможных контекстов» ограничить, например числом нетерминалов, в него входящих или каким-либо другим образом.

Однако и у данного критерия есть недостаток. Он не учитывает наличия гипотезы о том, что обработка рассматриваемого правила для сущностей одного вида (например, для базовых типов данных) и для сущностей другого вида (например, для пользовательских типов данных) выполняется по-

разному и, следовательно, ошибка может проявиться в одном случае и не проявиться в другом.

Эта гипотеза позволяет пополнить критерий полноты:

– покрыть все контекстные условия во всех возможных (синтаксически различных) контекстах (в некоторых ограничениях на число контекстов), перебрав все возможные вершины.

Этот критерий представляется вполне разумным, однако у него тоже есть недостаток. Не всегда можно перебрать «все контексты» или «все виды вершин», так существование в дереве некоторых вершин будет, запрещено этим же или другим контекстным условием. Поэтому вместо требования «все» лучше сказать «все возможные», тогда критерий становится вполне четким и практичным.

Таким образом, критерий полноты позитивного тестового набора для тестирования анализаторов контекстных условий можно сформулировать следующим образом:

Для каждого контекстного условия R из множества всех контекстных условий SR_{all}, набор тестов должен содержать тексты, покрывающие R и такие, что соответствующее множество деревьев содержит все возможные вершины u_i и v_j во всех возможных контекстах.

Мы показали, как строится критерий покрытия для анализаторов контекстных условий на основе описания семантики, представленной на SRL. Опыт использования данного критерия показал, что размер тестовых наборов, отвечающих сформулированному критерию, имеет вполне разумные размеры (от нескольких тысяч до нескольких десятков тысяч тестов для языка типа Java), и качество тестового набора вполне удовлетворительно, о чем говорилось во введении.

Теперь сформулируем критерий покрытия для негативных тестов для анализатора контекстных условий транслятора.

2.3.2 Критерий покрытия для негативных тестов

Негативные тесты используются для того, чтобы оценить правильность реализации в трансляторе диагностики ошибок. Для этого необходимо сгенерировать тестовые тексты, в которых нарушаются одно или несколько правил статической семантики входного языка.

Задача генерации негативных тестов, на первый взгляд, не кажется принципиально более трудной, чем задача генерации позитивных тестов. При этом рассуждения строятся примерно следующим образом. Если мы умеем генерировать тесты на основе спецификаций корректных входных данных, то можно внести изменения в эти спецификации, сделав их спецификацией некорректных данных, и при помощи уже имеющихся методов генерировать «некорректные», то есть негативные тесты. В рамках данной работы были проведены эксперименты, которые показали, что спецификации на SRL могут быть использованы для генерации и негативных тестов тоже.

Вместе с тем имеется фундаментальная трудность генерации негативных тестов, которая на самом деле является проблемой составления спецификации некорректных данных. Дело в том, что спецификации некорректных данных разрабатываются не сами по себе, хотелось бы их получать на основе имеющихся спецификаций корректных данных путем относительно простых модификаций (часто такие модификации называют *мутациями*)[56, 57]. При этом известно, что мутация спецификации корректных данных не всегда приводит к появлению спецификации некорректных данных, то есть, в результате мы можем не получить гарантированно негативный тест. В связи с наличием этой проблемы, в данной работе предлагается не полностью автоматический алгоритм получения негативных тестов, а набор рекомендаций, который позволяет получить негативные тесты в типовых случаях.

В данной работе для генерации негативных тестов предлагается специальным образом модифицировать SRL-спецификацию контекстных условий целевого языка. Более подробно вопросы, касающиеся генерации

негативных тестов будут рассмотрены в параграфе 3.3. В данном параграфе описываются только структура негативных тестов, которые предлагается генерировать, приводится обоснование целесообразности использования таких тестов и формулируется критерий покрытия для предлагаемого набора негативных тестов.

Используя *модифицированную («мутированную»)* SRL-спецификацию контекстных условий, в данной работе предлагается строить в качестве негативных тестов для анализатора контекстных условий транслятора тексты на входном языке, каждый из которых нарушает некоторое правило статической семантики входного языка и удовлетворяет всем остальным правилам статической семантики. Таким образом, все негативные тесты можно разбить на группы так, чтобы в одной группе находились тесты, нарушающие одно и то же контекстное условие. Рассмотрим группу правил, содержащую тексты, нарушающие контекстное условие R. Тогда все тесты, принадлежащие к этой группе, удовлетворяют контекстным условиям, заданным в *модифицированной* SRL-спецификации, в которой без изменений присутствуют все (кроме требования R) контекстные условия входного языка, и, плюс, еще одно правило, представляющее собой контекстное условие, выполнение которого будет означать нарушение правила R. Будем называть такое правило *мутацией контекстного условия* R и обозначать $\tilde{R}$.

В общем случае одно и то же контекстное условие R может быть нарушено несколькими способами. Это означает, что может существовать несколько мутаций контекстного условия R.

Такого рода тесты можно использовать в качестве негативных, если модифицированная спецификация, упомянутая выше, действительно будет описывать данные некорректные с точки зрения не модифицированной спецификации.

Сформулируем критерий покрытия для негативных тестов для анализаторов контекстных условий трансляторов.

Для каждой мутации $\tilde{R}$ *каждого контекстного условия* R *из множества всех контекстных условий* SR_{all}, *набор тестов должен содержать множество текстов покрывающих* $\tilde{R}$ *и таких, что соответствующее множество деревьев содержит все возможные вершины* u_i *и* v_j *во всех возможных контекстах такие.*

Заметим, что критерии покрытия, сформулированные в этой главе, предполагают выполнение гипотезы о том, что для анализатора контекстных условий все строковые константы эквивалентны, и все числовые константы одного типа также эквивалентны.

Обычно тесты генерируются таким образом, чтобы по построению удовлетворять сформулированному критерию покрытия. Следующий параграф посвящен вопросу генерации входных данных для наборов позитивных и негативных тестов, удовлетворяющих сформулированным критериям покрытия для позитивных и негативных тестов.

3 Семантически управляемая генерация тестовых входных данных

Рассмотрим технологию генерации тестов для анализатора контекстных условий.

Прежде чем перейти к описанию алгоритма генерации тестов для анализатора контекстных условий, рассмотрим способ представления данных, с которым придется работать.

3.1 Представление данных

В качестве внутреннего представления тестовых входных данных удобно использовать *(абстрактные) синтаксические деревья (AST – abstract syntax tree)*. Синтаксическое дерево [28] представляет собой дерево вывода в сжатом виде, удобном для представления языковых конструкций. Листья AST соответствуют терминалам, а остальные узлы – нетерминалам BNF-грамматики, корень AST соответствует стартовому символу грамматики. В синтаксическом дереве операции и ключевые слова не представлены в качестве листьев, а связываются с внутренним узлом, который выступает в дереве разбора родительским для этих листьев. Другое упрощение заключается в том, что цепочки одиночных продукций могут быть свернуты. Продукция $S \rightarrow if\ B\ then\ S_1\ else\ S_2$ может быть представлена в синтаксическом дереве в виде фрагмента AST, представленного на *Рис. 6*.

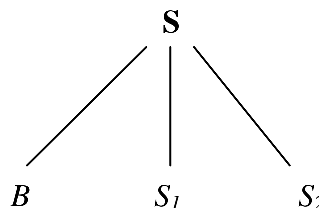


Рис. 6 Фрагмент дерева AST, соответствующего продукционному правилу
$S \rightarrow if\ B\ then\ S1\ else\ S2$

Каждый узел дерева помечается либо терминальным символом, либо нетерминалом X_{p0}, соответствующим применению продукционного

правила *p*. В последнем случае у этого узла будет n_p непосредственных потомков, n_p – количество символов, находящихся в правой части правила *p*. Корень дерева всегда помечается стартовым символом грамматики входного языка.

Таким образом, каждый узел дерева абстрактного синтаксиса грамматики входного языка имеет свой тип, который совпадает с именем какого-то нетерминального символа грамматики входного языка. Например, на *Рис. 6* представлены узлы следующих типов:

- *S* – оператор условного перехода *if-then-else;*
- *B* – условное выражение в *if-then-else;*
- S_1 – инструкция, стоящая после *then*;
- S_2 – инструкция, стоящая после *else.*

Будем описывать синтаксические деревья при помощи языка TreeDL [29] (*Tree Description Language*), разработанного группой RedVerst ИСП РАН [30] для описания сложных структур данных. TreeDL является аналогом таких нотаций как ASN.1 [31, 32] и ASDL [33] и предоставляет специальные средства для описания именованных деревьев[14]. Ниже представлена часть EBNF-грамматики языка TreeDL, которая будет использоваться далее в примерах:

```
tree      ::= ( node_type )+
node_type ::= ( “abstract” )? “node” <name:ID>
                  ( “:” <base:ID> )? “{” ( member )* “}”
member    ::= attribute | child
attribute ::= “attribute” <type_name> ( “?” | “*” | “+” )?
<name:ID> “;”
child::= “child” <type:ID> ( “?” | “*” | “+” )? <name:ID> “;”
```

Тип узла дерева внутреннего представления (node_type) описывает набор именованных дочерних узлов (child) и набор атрибутов (attribute). Атрибуты предназначены для хранения информации о терминалах. Для типа узла

---

[14] Для того, чтобы использовать формализм конструктивных грамматик требуется строить именованные деревья вывода. См. Определение 3.

node_type идентификатор <name:ID> задает его имя. Для типов узлов определен механизм одиночного наследования. Базовый тип узла может быть задан идентификатором <base:ID>. В этом случае все члены, определенные для базового типа, определены и для производного типа. Если присутствует модификатор abstract, то тип узла является абстрактным. Такие типы используются как абстрактная база для своих производных типов. Флажки ?, * и + имеют тот же смысл, что и в EBNF-нотации.

Пример 8.

Рассмотрим, как будет выглядеть на языке TreeDL описание узлов синтаксических деревьев, соответствующих текстам на языке L (см. пример 1).

Таб. 13 Синтаксис языка *L* в EBNF и TreeDL нотациях

<table>
<tr><th>Синтаксис: EBNF-описание</th><th>Синтаксис: TreeDL-описание</th></tr>
<tr><td>program ::= “program” identifier
“;”
(declaration)*</td><td>node Program
{
child Identifier id;
child Declarations decl;
}</td></tr>
<tr><td>declaration ::=
(“var” new_var_identifier)
|(“syn” new_syn_identifier
“=”
var_identifier)“;”

new_var_identifier ::=identifier

new_syn_identifier ::=identifier</td><td>abstract node Declarations{}
node CompoundDeclaration : Declarations
{
child Declaration decl;
child Declarations decls;
}
abstract node Declaration :

Declarations {}
node VarDecl : Declaration{
child Identifier id; }</td></tr>
</table>

Синтаксис: EBNF-описание	Синтаксис: TreeDL-описание

`var_identifier ::= identifier`
 	
`identifier ::= (a|b|c|…|z)+ (0|1|2|3|4|5|6|7|8|9)*`
	`node SynDecl : Declaration`
	`{`
	`  child Identifier synId;`
	`  child Identifier varId;`
	`}`
	`node Identifier`
	`{`
	`  attribute String id;`
	`}`

TreeDL-форма грамматики является абстракцией EBNF в следующем смысле: игнорируются константные терминальные символы, а также взаимное расположение детей и атрибутов внутри узла (за исключением случая элементов списка). И то и другое восстанавливается из контекста. Более строго, EBNF-запись грамматики эквивалентна паре, состоящей из TreeDL-записи и функции восстановления текста по дереву.

Далее в SRL-спецификациях вместо имен грамматических символов EBNF-грамматики входного языка будут использоваться типы узлов из TreeDL-описания, а в качестве имен узлов в SRL-описании будут использоваться имена дочерних узлов, указанные в TreeDL-описании.

Описание правил статической семантики на языке SRL с использованием имен, заданных в TreeDL описании приведено в следующей таблице.

Таб. 14 Статическая семантика языка *L* на языке SRL

№	Спецификация контекстных условий

1.	`many-to-many relation R1`
	`{`
	`   unequal`
	`   target VarDecl { id }`
	`   source SynDecl { synId }`
	`}`

<table>
<tr><th>№</th><th>Спецификация контекстных условий</th></tr>
<tr><td>2.</td><td><pre>many-to-many relation R2
{
   unequal
   target VarDecl { id }
   source VarDecl { id }
}</pre></td></tr>
<tr><td>3.</td><td><pre>many-to-many relation R3
{
   unequal
   target SynDecl { synId }
   source SynDecl { synId }
}</pre></td></tr>
<tr><td>4.</td><td><pre>one-to-many relation R4
{
   equal
   target SynDecl { varId }
   source VarDecl { id }
}</pre></td></tr>
</table>

Заметим, что в TreeDL-спецификации в Таб. 13 оба дочерних узла SynDecl имеют тип Identifier, следовательно различить их по типу нельзя. Поэтому в Таб. 14 в фигурных скобках используются имена дочерних узлов, вместо их типов как это было сделано в Таб. 12.

3.2 Генерация входных данных для позитивных тестов на основе конструктивного описания статической семантики

Использование конструктивного описания на языке SRL позволило разработать алгоритм *семантически управляемой генерации*, следуя которому можно целенаправленно систематически построить входные данные для

позитивных или негативных тестов, удовлетворяющие заданным в предыдущем параграфе критериям покрытия.

В общем случае множество всех возможных текстов бесконечно. Однако, поскольку тестовый набор должен быть ограничен, мы должны определить некоторые количественные ограничения на размер тестового набора. С точки зрения полноты тестирования это означает, что мы должны принять гипотезу о характере возможных ошибок. Наш метод должен находить все ошибки в анализаторе контекстных условий, если гипотеза верна, и может пропускать некоторые из ошибок, если гипотеза неверна.

Ограничения на размер тестового набора мы введем в два приема. Сначала ограничим глубину рекурсии при раскрытии продукционных правил грамматики, потом введем классы эквивалентности деревьев с тем, чтобы из одного класса эквивалентности генерировать только одного представителя.

Гипотеза 1. Разумное ограничение глубины рекурсии при раскрытии продукционных правил грамматики входного языка позволит построить набор тестов, достаточный для проведения тестирования анализатора контекстных условий транслятора[15].

Однако, даже после введения ограничения на глубину рекурсии, количество тестовых текстов будет слишком велико. Необходимо еще больше ограничить множество тестовых входных данных.

Напомним, что нашей целью является генерация входных данных для организации целенаправленного систематического тестирования. Для достижения этой цели требуется генерировать такие тексты, чтобы о каждом было известно, для тестирования какого аспекта анализатора контекстных условий он предназначен (таким образом будет достигнута целенаправленность), и чтобы в сгенерированном наборе существовали тексты, предназначенные для тестирования каждого аспекта анализатора контекстных условий. Основной задачей анализатора контекстных условий является проверка правил статической семантики соответствующего языка во

[15] Размер ограничения на глубину рекурсии в рамках данной гипотезы должен определяться опытным путем.

входных данных. Следовательно, для достижения поставленной цели необходимо сгенерировать такой набор текстов такой, чтобы о каждом тексте было известно, проверку какого контекстного условия будет осуществлять анализатор при обработке данного текста. Такое контекстное условие будем называть *первичным контекстным условием* (или *первичным правилом*) данного текста.

Заметим, что текст p всегда покрывает свое первичное контекстное условие, т. е. по определению 10 $\exists u_1 \in U(t) | P_{trg}(u)$, если первичное правило имеет вид (1) или (3), или $\exists u_2, v_2 \in U(t) | P_{trg}(u_2) \wedge P_{src}(u_2, v_2)$, если первичное правило вида (2).

На самом деле многие тексты на целевом языке могут быть использованы для проверки реализации нескольких контекстных условий в анализаторе. Каждому тексту может соответствовать только одно первичное контекстное условие. Однако разным первичным условиям могут соответствовать одинаковые тексты. Поэтому множества текстов предназначенных для тестирования двух различных аспектов анализатора контекстных условий могут пересекаться.

Рассмотрим некоторое дерево $t \in M(G)$. Пусть t соответствует первичному правилу R. Заметим, что в t может существовать несколько различных вершин, удовлетворяющих требованиям определения источника и цели правила R (см. определение 8). Но только одну из этих вершин будем называть *первичным источником* и только одну из вершин *первичной целью*.

Рассмотрим поддерево, соединяющее корень дерева тестового текста с первичным источником и первичной целью.

Определение 11. *Первичным деревом* называется минимальное поддерево t' дерева $t \in M(G)$, содержащее корневую вершину s дерева t, первичную цель u дерева t и первичный источник v дерева t и такое, что u и v могут быть соответственно первичной целью и первичным источником для t'.

Таким образом, первичное дерево содержит не только вершины, расположенные на путях от s к u и от s к v, но и другие вершины, существование которых требуется для истинности соответствующих предикатов положения P_{trg} и P_{src} первичного правила R. Только в этом случае выполнится следующее $(P_{trg}(u) \wedge \exists x \in U(T^{u\prime}) \wedge P_{trg_subtree}(x) \wedge P_{src}(u, v) \wedge \exists y \in U(T^{v\prime}) \wedge P_{src_subtree}(y))$, где $T^{u\prime}$, $T^{v\prime}$– поддеревья дерева t' с корнями в вершинах u и v соответственно.

Первичные деревья часто являются *синтаксически* и *семантически неполными*.

Определение 12. *Синтаксически неполными* будем называть деревья, которым не соответствует синтаксически корректная цепочка символов на соответствующем языке.

Определение 13. *Синтаксически полными* будем называть деревья, которым соответствует синтаксически корректная цепочка на соответствующем языке.

Определение 14. *Семантически неполными* будем называть синтаксически полные деревья, которым не соответствует корректная с точки зрения правил статической семантики цепочка символов на соответствующем языке.

Определение 15. *Семантически полными* будем называть синтаксически полные деревья, которым соответствует корректная с точки зрения правил статической семантики цепочка на соответствующем языке.

При условии ограничения на глубину рекурсии и в силу конечности множества продукционных правил любое синтаксически неполное дерево может быть достроено до синтаксически полного за конечное число шагов путем добавления узлов, полученных после раскрытия нетерминалов, соответствующих внутренним узлам, которые не обладают полным набором дочерних узлов. Эти дочерние узлы описываются обязательными вхождениями нетерминалов в правые части продукционных правил,

задающих способы раскрытия нетерминалов, представленных в дереве рассматриваемыми внутренними узлами.

Проблема достижения семантической полноты является более сложной и будет рассмотрена отдельно в параграфе 3.2.2.

После того, как были введены понятия первичного поддерева, синтаксически полного и синтаксически неполного поддеревьев продолжим наши рассуждения.

Сформулируем еще одну гипотезу для ограничения количества тестов.

Гипотеза 2. Сгенерированный тестовый набор для тестирования анализатора контекстных условий будет полным, если в него будет входить хотя бы по одному представителю из каждого класса эквивалентности, задаваемого критерием покрытия.

Сформулированный ранее критерий покрытия для позитивных тестовых данных требует поочередного рассмотрения в качестве первичного правила каждого контекстного условия целевого языка и построения дерева для каждого возможного расположения вершин u и v таких, что $R(u)$ или $R(u, v)$, в зависимости от вида R. Построив для каждого первичного правила множество всех возможных первичных деревьев и достроив их до синтаксически полных, мы тем самым получим множество деревьев, удовлетворяющих условиям критерия покрытия.

Далее приводится описание сформулированной идеи в более строгой форме.

Разобьем множество всех позитивных тестовых текстов на подмножества таким образом, что каждое подмножество будет содержать, по крайней мере, один тестовый текст, являющийся представителем одного из классов эквивалентности, на которые критерий покрытия для позитивных тестов разбивает все множество тестовых текстов.

Пусть T_R – конечное множество всех тестовых текстов, соответствующих одному контекстному условию R, т.е. контекстное условие R – это *первичное правило* для всех текстов из T_R.

Пусть F_R — семейство подмножеств множества T_R. При этом каждый элемент множества T_R принадлежит хотя бы одному из подмножеств семейства F_R: $T_R = \bigcup_{S \in F_R} S$.

Определение 16. Подмножество S множества T_R будем называть *элементом семейства подмножеств* F_R, если S принадлежат только те тексты, синтаксические деревья которых содержат одинаковые первичные поддеревья.

Для того чтобы добиться выполнения условий критерия покрытия, достаточно построить по одному представителю (тестовому тексту) каждого подмножества семейства F_R, то есть построить набор тестовых текстов $\mathcal{T} = \{p_i \in S_i \mid S_i \in F_R, 1 \le i \le |F_R|\}$.

Прежде чем перейти к описанию шагов алгоритма вспомним неформальное предположение, приведенное в конце параграфа 2.1.

В соответствии с ним, если множество контекстных условий языка L можно описать в виде системы правил, состоящих из предикатов, заданных на множестве вершин деревьев, соответствующих всем возможным текстам на языке L, тогда, если множество вершин $U(t)$ некоторого дерева t, соответствующего синтаксически корректному тексту, является решением этой системы правил при условии, что каждое правило принимает значение *"истина"*, тогда текст, соответствующий дереву t будет также и семантически корректен. Оказывается, что множество контекстных условий требованиям, которых должен удовлетворять некоторый текст, может быть подмножеством множества всех контекстных условий языка L. Приведем конструктивное определение такого подмножества и докажем теорему.

Определение 17. *Базовым множеством* R^t контекстных условий дерева $t \in M(G)$ называется множество таких контекстных условий, посылки в которых могут принимать значение *"истина"* на некоторых вершинах дерева t

$$R^t = \{R'(u) \in SR_{all} \mid \exists u' \in U(t) : P'_{trg}(u')\}$$

$$\cup \{R''(u,v) \in SR_{all} \mid \exists u'', v'' \in U(t) : P''_{trg}(u) \wedge P''_{src}(v)\}.$$

Все контекстные условие, принадлежащие R^t, будем называть *базовыми контекстными условиями* (или *базовыми правилами*).

Любое одно базовое контекстное условие может быть назначено первичным правилом. Следовательно, любое первичное правило принадлежит базовому множеству.

Докажем теорему.

Теорема 1 ***(Критерий семантической корректности)*** *Для того, чтобы текст p, написанный на языке, порожденном однозначной КС-грамматикой, был семантически корректен, необходимо и достаточно, чтобы выполнялись все базовые контекстные условия, соответствующие дереву текста p.*

Доказательство: Поскольку текст p написан на языке, порожденном однозначной КС-грамматикой, то ему однозначно соответствует АСТ t. Если текст p семантически корректен, то по определению *1* он удовлетворяет всем семантическим правилам соответствующего языка. Базовое множество семантических правил R^t дерева t является подмножеством множества всех семантических правил SR_{all}, заданных в спецификации входного языка, следовательно, все базовые правила из R^t выполняются на множестве вершин дерева t, и, следовательно, текст p удовлетворяет семантическим правилам из базового множества контекстных условий, соответствующих t.

Пусть текст p удовлетворяет всем базовым контекстным условиям, соответствующим дереву текста p. Разобьем множество всех контекстных условий SR_{all} на два подмножества $SR_{all}=R^t \cup SR_{all} \backslash R^t$, где R^t – это базовое множество контекстных условий дерева t, которое является деревом текста p и

$$R^t = \{R'(u) \in SR_{all} \mid \exists u' \in U(t) : P'_{trg}(u')\}$$

$$\cup \{R''(u,v) \in SR_{all} \mid \exists u'' \in U(t) : P''_{trg}(u'') \vee \exists v'' \in U(t) : P''_{src}(v'')\}.$$

Тогда

$R_{all}\backslash R^{t}=\{R'(u)\in SR_{all} \mid \sim \exists u'\in U(t): P'_{trg}(u')\}$

$\cup\{R''(u,v)\in SR_{all} \mid \sim \exists u''\in U(t): P''_{trg}(u'')\wedge \sim \exists v''\in U(t): P''_{src}(v'')\}$, следовательно,

$\forall R'(u)\in SR_{all}\setminus R^{t}\ \forall u\in U(t)\ \sim P'_{trg}(u)$ и

$\forall R''(u,v)\in SR_{all}\setminus R^{t}\ \forall u''\in U(t)\ \sim P''_{trg}(u'')\wedge\forall v''\in U(t)\ \sim P''_{src}(v'')$. Вспомним, что любое контекстное условие R – это импликация посылка, которой представлена либо одним предикатом P_{trg}, либо конъюнкцией предикатов $P_{trg} \wedge P_{src}$, а импликация, посылка в которой всегда ложна, также ложна. Из чего следует, что $\forall R\in SR_{all}\backslash R^{t}\ \ \forall u\in U(t)\ \ R(u)$. Следовательно, на множестве вершин дерева t текста p выполняются все контекстные условия соответствующего языка, заданные множеством SR_{all}. Следовательно, текст p удовлетворяет всем контекстным условиям языка и по определению *1* семантически корректен. Теорема доказана.

Перейдем к описанию алгоритма генерации позитивных тестовых входных данных для анализатора контекстных условий. Главными шагами в этом алгоритме являются следующие действия:

Алгоритм 1 (***Алгоритм семантически управляемой генерации.***)

A1S1. Из множества SR_{all} всех контекстных условий языка L выбрать непомеченное контекстное условие R и пометить его. Если такого контекстного условия нет, то КОНЕЦ алгоритма, иначе перейти на следующий шаг.

A1S2. Для первичного правила R построить множество всех *первичных деревьев* с учетом ограничения глубины рекурсии.

A1S3. Выбрать из множества первичных деревьев, построенных на предыдущем шаге, непомеченное дерево $t_{primary}$, пометить его и перейти на следующий шаг, если такого дерева нет, то A1S1.

A1S4. Первичное поддерево $t_{primary}$, выбранное на предыдущем шаге, достроить минимальным образом до синтаксически полного дерева t, т.е. продолжить раскрытие нетерминалов до тех пор, пока это позволяет грамматика языка.

Достраивание дерева минимальным образом осуществляется за счет построения всех возможных дочерних вершин, помеченных ε.

A1S5. Выделить для синтаксически полного дерева *t*, полученного на предыдущем шаге, *базовое множество* контекстных условий R^t.

A1S6. Для того чтобы убедиться в семантической корректности текста в силу критерия семантической корректности необходимо и достаточно проверить удовлетворяет ли текст контекстным условиям из соответствующего базового множества. Проверить, истинны ли базовые правила для данного дерева *t*, т.е. верно ли следующее $\forall u,\ v \in U(t)\ \forall i = \overline{1,k},\ \forall j = \overline{1,m}\ \ (R_i(u)) \wedge (R_j(u,\ v))$, где R_i, $R_j \in R^t$, *m*– это количество базовых правил вида (2), т.е. типа *many-to-many*, а $k=(|R^t|-m)$. В силу построения базовые контекстные условия всегда ложны тогда и только тогда, когда не выполняются предикаты, стоящие справа от знака импликации.

A1S7. Если все базовые правила всегда истинны на множестве *U(t)*, то перейти к шагу A1S8, иначе произвести изменения в дереве *t*, не затрагивая первичного дерева, такие, чтобы все базовые правила были всегда истинны. Если *удалось* произвести требуемые изменения в дереве, то перейти к шагу A1S6, иначе дерево *t* объявить *семантически неразрешимым* и перейти к шагу A1S3.

A1S8. Базовые контекстные условия для дерева *t* истинны на множестве *U(t)*. Отобразить дерево *t* в текст на входном языке. Построенный текст будет удовлетворять всем синтаксическим и контекстным требованиям соответствующего языка. Перейти к шагу A1S3.

Заметим, что при таком построении гарантируется, что каждый тестовый текст обязательно удовлетворяет некоторому фиксированному контекстному условию, а именно, первичному правилу, а в общем случае в этом тексте могут выполняться еще несколько контекстных условий. В результате позитивный тестовый набор может оказаться избыточным, но при этом по построению гарантируется его полнота в терминах заданного критерия покрытия.

Рассмотрим более подробно этапы процесса генерации позитивного тестового набора.

Далее будем считать, что множество $SR_{all} \neq \emptyset$. Иначе это означает, что в языке нет никаких дополнительных условий, кроме синтаксических требований. Для генерации тестовых данных в этом случае следует использовать алгоритмы генерации синтаксических тестов, приведенные в работах [11, 27].

Реализация шага A1S1 тривиальна. В силу конечности множества SR_{all} его элементы можно перенумеровать и представить множество контекстных условий в виде массива `srAll`. При первом обращении к шагу A1S1 создать некоторую целочисленную переменную `ruleNumber` проинициализировать ее: `ruleNumber:=0`. При каждом выборе очередного первичного правила изменить значение индекса на единицу `ruleNumber:=ruleNumber+1` и проинициализировать объект, соответствующий первичному правилу значением `srAll[ruleNumber]`.

Наиболее сложными являются шаги A1S2 и A1S6, поэтому описания этих шагов выделены в два отдельных параграфа 3.2.1 и 3.2.2 соответственно. Остальные шаги алгоритма A1S3, A1S4, A1S5, A1S8 реализуются достаточно просто и являются переходными, краткое описание этих шагов будет дано в следующих двух параграфах.

3.2.1 Построение первичных поддеревьев

Итак, на шаге A1S1 было выбрано первичное правило *R*. На шаге A1S2 требуется построить с учетом ограничения глубины рекурсии множество всех возможных первичных деревьев ему соответствующих. Разработаем алгоритм построения первичных поддеревьев правила *R*.

Основными шагами в этом алгоритме являются следующие действия.

Алгоритм 2 *(Построение первичных поддеревьев для контекстного условия R.)*

A2S1. Используя продукционные правила грамматики исходного языка и алгоритм поиска в глубину [59], найти все возможные пути l_q от вершины s, соответствующей стартовому правилу грамматики до всех возможных первичных целей u_i и пути h_p до всех возможных первичных источников v_j.

A2S2. Используя элементы декартова произведения $\{u_i\}_{i=1,\dots,I}\times\{l_q\}_{q=1,\dots,Q}$, где $I=|\{u_i\}|$ и $Q=|\{l_q\}|$, построить множество цепочек $\{g_{iq}\}_{i=1,\dots,I,\ q=1,\dots,Q}$.

A2S3. Используя элементы декартова произведения $\{v_j\}_{j=1,\dots,J}\times\{h_p\}_{p=1,\dots,P}$, где $J=|\{v_j\}|$ и $P=|\{h_p\}|$, построить множество цепочек $\{g_{jp}\}_{j=1,\dots,J,\ p=1,\dots,P}$.

A2S4. Используя элементы декартова произведения $\{g_{iq}\}_{i=1,\dots,I,\ q=1,\dots,Q}\times\{g_{jp}\}_{j=1,\dots,J,\ p=1,\dots,P}$ построить первичные деревья $\{t_{iqjp}\}_{i=1,\dots,I,\ q=1,\dots,Q,\ j=1,\dots,J,\ p=1,\dots,P}$.

Рассмотрим более подробно все шаги алгоритма 2.

Не теряя общности, будем считать, что первичное правило R имеет вид (1), т.е. $R(u)\equiv\forall\, t\in M(G),\ \forall\, u\in U(t),\ P_{trg}(u)\Rightarrow F(u)$. Тогда в общем виде P_{trg} будет иметь вид (5), т.е. $P_{trg}(x_k)\equiv\exists x_0,\dots,x_{k-1}P_0(x_0)\wedge\bigwedge_{i=1,k}P_i(x_{i-1},x_i)\bigwedge_{l=0,n}P_l(x_i)$, где

k>0, P_i – атомарные формулы, представленные в таблицах 10, 11 и полученные по правилу (4). Для того чтобы выполнить первый шаг A2S1 алгоритма, необходимо сначала построить множество всех возможных первичных целей, т.е. таких вершин $\{u_i\}_{i=1,\dots,I}$, что $\forall u_j\in\{u_i\}_{i=\overline{1,I}}$

$(P_{trg}(u)\wedge\exists x,y\in U(T^u)\wedge P_{trg_subtree}(x)\wedge P_{src_subtree}(y))$. Следовательно, для $\forall u_j\in\{u_i\}_{i=1,\dots,I}$ необходимо обеспечить существование вершин $x_0,\dots,x_{k-1}$ таких, что $P_0(x_0),\ \forall l,\ 0\le l\le n\ P_l(x_i),\ \forall i,\ 1\le i\le k\ P_i(x_{i-1},\ x_i)$. Атомарные формулы, представленные в таблицах 10 и 11, легко могут быть интерпретированы в качестве инструкций о создании вершин, удовлетворяющих определенным требованиям. Это означает, что атомарным предикатам $P(x)$ или $P(x,\ y)$ можно поставить в соответствие функцию для вычисления значения переменной x и y. Далее приводится таблица, в которой каждому атомарному предикату ставится в соответствие функция вычисления значения переменных, удовлетворяющих данному предикату.

Таб. 15 Формулы вычисления значений переменных, удовлетворяющих требованиям атомарных предикатов в КГ

№	**Атомарные формулы, описывающие положение вершин**	**Функции вычисления значения переменных**	**Описание**
1.	*A(u)*	*u = createNodeOfType("A")*	Функция создания вершины типа type
2.	*ε(u)*	*u = createEpsilonNode()*	Функция создания вершины с меткой ε
3.	*b(u)*	*u = createNodeWithName("b")*	Функция создания вершины с именем *b*
4.	*parent(u, v)*	*v = getParent(u)*	Функция, возвращающая родительскую вершину вершины *u*
5.	*^B(u, v)*	*v=getAncestorOfType(u, "B")*	Функция, возвращающая ближайшую вершину-предка типа *type* по отношению к вершине *u*
6.	*<n.B(u, v)*	*v=createDescendentOfType(u, "B", n, LESS)*	Функция создания вершины типа type в поддереве с корнем в вершине u на глубине меньше, больше и т. д., чем depth. Вместо параметра sign должна быть подставлена константа (в данном случае LESS), поясняющая смысл параметра depth. Константа LESS означает, что длина пути от вершины u до вершины v должна быть меньше depth. Следующие четыре формулы используют эту же функцию и другие константы в качестве значения параметра sign.
7.	*<=n.B(u, v)*	*v=createDescendentOfType(u, "B", n, LESSEorQUAL)*	Константа *LESSEorQUAL* означает, что длина пути от вершины u до вершины v должна быть не больше depth.
8.	*>n.B(u, v)*	*v=createDescendentOfType(u, "B", n, MORE)*	Константа *MORE* означает, что длина пути от вершины u до вершины v должна быть больше depth.

9.	*>=n.B(u, v)*	*v=createDescendentOfType(u, “B”, n, MOREorEQUAL)*	Константа *MOREorEQUAL* означает, что длина пути от вершины u до вершины v должна быть не меньше, чем depth.
10.	*n.B(u, v)*	*v=createDescendentOfType(u, “B”, n, EQUAL)*	Константа *EQUAL* означает, что длина пути от вершины u до вершины v должна быть равна depth.
11.	*00.B(u, v)*	*v=createDescendentOfType(u, “B”)*	Функция создания вершины типа type в поддереве с корнем в вершине u на любой глубине.

Теперь, когда для каждого атомарного предиката существует функция построения вершины, которая бы ему удовлетворяла, опишем порядок построения вершин необходимых для того, чтобы выполнялось первичное правило.

В общем случае построение $\forall u_j \in \{u_i\}_{i=1,\ldots,I}$ и соответствующего множества $\{x_m\}_{m=0,\ldots,kj-1}$ следует начать с построения вершины x_0, удовлетворяющей одноместному предикату P_0. Используя функцию построения соответствующую предикату P_0, строим вершину x_0. Если, например, предикат $P_0(x) \equiv A(x)$, это означает, что любая вершина типа A удовлетворяет требованиям предиката P_0. Следовательно, x_0 = *createNodeOfType(“A”)*. Теперь можно перейти к построению вершины x_1.

Вершины $x_1, x_2, \ldots, x_{kj-1}, u_j$ должны удовлетворять требованиям двуместных атомарных предикатов. Заметим, что с них нельзя начинать построение, так как всем двуместным атомарным предикатам соответствуют функции, которые в качестве параметров используют ранее построенные вершины. Так для построения вершины x_1 потребуется вершина x_0, для построения x_2 – вершина x_1, для построения x_{kj-1} – вершина x_{kj-2}, для построения u_j – вершина x_{kj-2}.

Рассмотрим пример.

Пример 9.

Рассмотрим первичное контекстное условие из примера 6

$R_4(u) \equiv \forall t \in M(G), \forall u \in U(t)$ $(\ var_identifier(u) \Rightarrow (\exists v \in U(t)\ u \neq v \wedge new_var_identifier(v) \wedge \exists x \in U(T^u)\ 1.identifier(u, x) \wedge \exists y \in U(T^v)\ 1.identifier(v, y) \Rightarrow T^x == T^y))$

Для $R_4(u)$ предикат положения цели $P_{trg}(u) \equiv var_identifier(u)$, а предикат положения источника $P_{src}(v) \equiv new_var_identifier(v)$. Первичную цель для правила $R_4(u)$ можно получить следующим образом *u=createNodeOfType("var_identifier")*, а первичный источник – *v=createNodeOfType("new_var_identifier")*.

Заметим, что при построении первичной цели или источника может быть получена не одиночная вершина, а целое поддерево. Пусть предикат положения цели первичного правила R имеет вид $P_{trg}(x_k) \equiv \exists x_0, ..., x_{k-1} P_0(x_0) \wedge (\bigwedge_{i=1,k} P_i(x_{i-1}, x_i))$. Если $k=0$, тогда будет построена одиночная вершина (см. пример 9), если $k>0$, тогда будет построен поддерево, содержащее первичную цель.

Именно по этой причине мы начали построение с первичных вершин. Теперь можно перейти к поиску путей к полученным вершинам. Если в результате построения множества первичных целей $\{u_i\}_{i=1,...,I}$ или первичных источников $\{v_j\}_{j=1,...,J}$ были получены поддеревья, то пути $\{l_q\}_{q=1,...,Q}$ и $\{h_p\}_{p=1,...,P}$ нужно искать до корневых вершин, полученных поддеревьев, содержащих первичные источник и цель.

Как уже говорилось ранее, для поиска путей будем использовать алгоритм поиска в глубину. Нам уже известны типы вершины, путь до которых нужно найти. Не теряя общности, будем считать, что необходимо найти путь от вершины типа S до вершины типа X.

Теперь, когда найдены первичные цели, источники и пути к ним, можно перейти к реализации шагов A2S2, A2S3. Для создания цепочки вершин необходимо последовательно, начиная со стартовой вершины, создать вершины всех типов, указанные в пути, присоединяя их друг к другу в отношении "родительская вершина – дочерняя вершина". К последней

вершине в цепочке добавить в качестве дочерней первичную вершину, либо корень поддерева, содержащего первичную вершину. В результате будут построены множество цепочек от стартовой вершины *s* до первичной цели $\{g_{iq}\}_{i=1,...,I,\ q=1,...,Q}$ и множество цепочек от стартовой вершины *s* до первичного источника $\{g_{jp}\}_{j=1,...,J,\ p=1,...,P}$.

На шаге A2S4 необходимо построить первичные деревья из полученных на предыдущем шаге цепочек. Будем делать это следующим образом. Каждая дуга, соединяющая две вершины в цепочке, соответствует отношению между родительским узлом и одним из дочерних узлов и может быть помечена именем соответствующего дочернего узла из TreeDL-описания.

Будем соединять цепочки следующим образом: начиная от стартового узла каждой цепочки, следуя сверху вниз, будем сливать воедино узлы одинаковых типов при условии, что в них направлены дуги с одинаковыми метками. При слиянии таких узлов все исходящие дуги присоединяются к узлу, получившемуся в результате слияния. В результате будет построено множество первичных деревьев, соответствующих некоторому первичному правилу.

Итак, мы рассмотрели реализацию шага A1S2. Следующий шаг A1S3 является переходным и реализуется тривиально. На этом шаге из построенных деревьев выбирается одно, на основе которого будет строиться тестовый семантически корректный текст. После того как в результате выполнения дальнейших шагов алгоритма этот текст будет построен или будет принято решение о невозможности его построения, произойдет переход к шагу A1S3, на котором будет выбрано следующее первичное дерево.

На следующем шаге A1S4 происходит достраивание первичного дерева до синтаксически полного. Рассмотрим пример, в котором будет проиллюстрирован процесс построения первичных деревьев и получения из них синтаксически полных деревьев.

Пример 10.

Рассмотрим контекстное условие 2 из Таб. 14 (с остальными контекстными условиями поступим аналогично). Типы источника и цели данного контекстного условия совпадают.

На *Рис. 7* представлено множество таких цепочек при ограничении глубины рекурсии равному трем. Узлы деревьев обозначены кругами, внутри которых написаны названия типов узлов; стрелки, соединяющие узлы деревьев, направлены от родительского узла к дочерним; надписи над стрелками соответствуют названиям детей, которые присоединены к родительскому узлу.

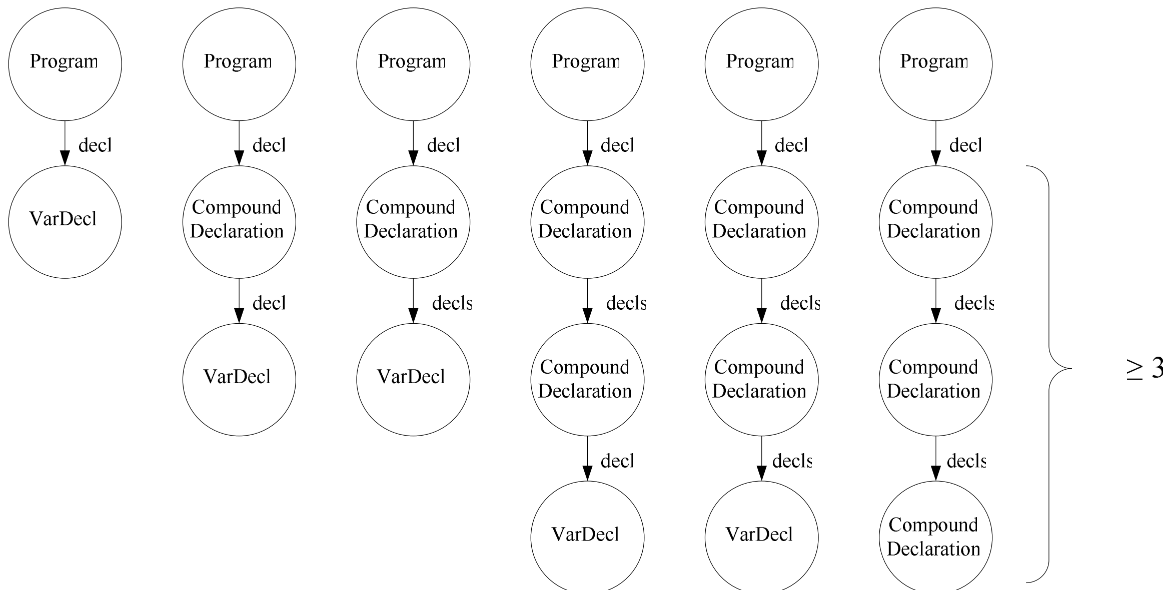


Рис. 7 Цепочки вершин от корневой вершины до вершины цели (источника) при ограничении глубины рекурсии ра <3

Всего существует пять таких цепочек. Шестая цепочка с *Рис. 7* уже не удовлетворяет ограничению на глубину рекурсии.

Теперь необходимо попарно соединить цепочки для того, чтобы построить множество первичных деревьев. Проанализируем *Рис. 7* с целью определить места возможного слияния цепочек. К первой цепочке можно присоединить только два узла типа *Identifier*, поэтому ее нельзя объединить ни с одной другой цепочкой из представленных на *Рис. 7*. На второй цепочке, кроме узлов типа *Identifier,* также можно присоединить дочерний узел типа

Declarations под именем *decls*. Следовательно, вторую цепочку можно объединить с третьей, четвертой и пятой цепочками. В процессе объединения цепочек эквивалентные узлы сливаются, а хвосты присоединяются в качестве дочерних узлов.

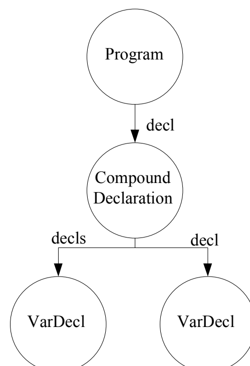


Рис. 8 Первичное дерево, полученное в результате объединения второй и третьей цепочек, представленных на Рис. 7

Например, в результате объединения второй и третьей цепочек будет получено первичное поддерево, представленное на *Рис. 8*.

Всего в результате объединения цепочек (см. *Рис. 7*) будет построено четыре различных первичных поддерева (см. *Рис. 9*).

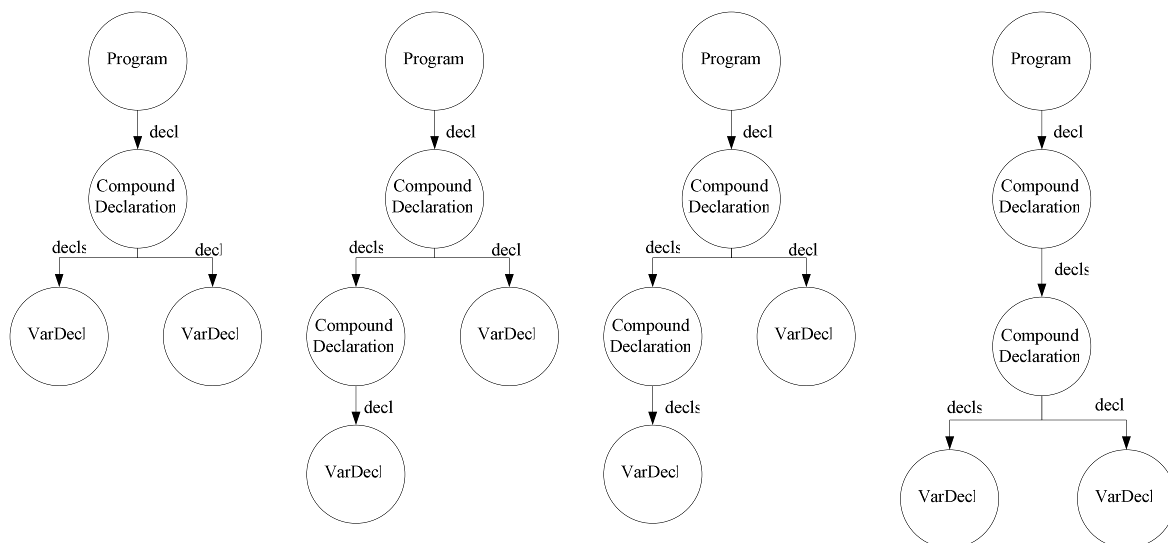


Рис. 9 Первичные поддеревья, полученные в результате объединения цепочек, представленных на Рис. 7

Каждое поддерево, представленное на *Рис. 9*, содержит два узла типа *VarDecl*, составляющих пару “источник – цель” для первичного контекстного условия, которому соответствуют эти поддеревья.

Заметим, что на *Рис. 9* представлены синтаксически неполные деревья, так как каждый узел типа *Program* или *VarDecl* должен быть достроен дочерним узлом типа *Identifier,* а во 2, 3 и 4 деревьях узлам типа *CompoundDeclaration* не хватает детей с именами *decl* или *decls*. Покажем на этом примере, что должно происходить на следующем шаге алгоритма 1.

В данном случае существует несколько способов достроить эти деревья до синтаксически полных как этого требует шаг A1S4 алгоритма 1. Например, в качестве ребенка по имени *decl* к узлу типа *CompoundDeclaration* может быть присоединен как узел типа *VarDecl*, так и типа *SynDecl.* В соответствии с гипотезой 2 в тестовом наборе должен существовать хотя бы один представитель каждого класса эквивалентности. Наличие представителя заданного класса эквивалентности определяется существованием в тестовом наборе текста, синтаксическое дерево которого содержит первичное поддерево, принадлежащее заданному классу эквивалентности. Так как в данному примере до синтаксической полноты требуется достроить первичное поддерево, следовательно, какой бы способ добавления вершин мы не выбрали, данный тест все равно останется представителем того класса эквивалентности, к которому принадлежит данное первичное поддерево.

Следовательно, все варианты получения синтаксически полных деревьев из первичных поддеревьев можно считать эквивалентными, поэтому можно выбирать любой вариант.

Перейдем к описанию шагов A1S5, A1S6 алгоритма 1.

3.2.2 Обеспечение семантической корректности тестовых текстов

Итак, в предыдущем параграфе был приведен способ построения первичных поддеревьев и дополнение их до синтаксически полных деревьев. Из каждого такого дерева теперь может быть получен синтаксически

корректный текст на целевом языке, но этот текст не обязательно будет удовлетворять требованиям статической семантики целевого языка. Поэтому в данном параграфе описываются дальнейшие действия по обеспечению семантической корректности строящихся текстов.

Сначала мы определим множество контекстных условий, которым должен удовлетворять текст *p*, соответствующий дереву *t*.

В соответствии с критерием семантической корректности для того, чтобы текст *p* был семантически корректным необходимо и достаточно, чтобы он удовлетворял всем контекстным условиям из R^t – базового множества контекстных условий дерева *t*.

Для того чтобы построить базовое множество контекстных условий, соответствующее дереву *t*, нужно, в соответствии с определением 17, обойти дерево *t* с тем, чтобы определить для каких контекстных условий $R'(u) \in SR_{all}$ $\exists u' \in U(t)\ P'_{trg}(u')$ и $R''(u, v) \in SR_{all}\ \exists u'' \in U(t)\ P''_{trg}(u'') \vee \exists v'' \in U(t)\ P''_{src}(v'')$. В этом заключается суть шага A1S5 алгоритма 1.

Перейдем к описанию шагов A1S6 и A1S7 алгоритма 1.

После того, как будет получено базовое множество контекстных условий дерева *t*, необходимо проверить удовлетворяет ли дерево *t* каждому из контекстных условий базового множества.

Проверка выполнения требований контекстных условий на множестве вершин дерева *t* для каждого типа контекстных условий имеет свои особенности.

Напомним, что в конструктивной грамматике контекстные условия бывают трех типов, которые называются *one-node*, *many-to-many* и *one-to-many*. Рассмотрим особенности проверки контекстных условий для каждого из типов отдельно.

Пусть *t* – это некоторое дерево, соответствующее тексту *p* на языке *L(G)*, и пусть дереву *t* соответствует базовое множество контекстных условий R^t.

Требуется определить справедливость следующего утверждения $\forall u,\ v \in U(t)$ $\forall R'(u),\ R''(u,\ v) \in R^t\ R'(u) \wedge R''(u,\ v)$.

Пусть в базовом множестве контекстных условий R^t дерева t присутствует контекстное условие типа *one-node*, т.е. $\exists R'(u) \in R^t$ такое, что $R'(u) \equiv \forall u \in U(t)\ P'_{trg}(u) \Rightarrow F'(u,\ u)$. Если $\exists u'' \in U(t) \sim P'(u'')$, тогда $R'(u'')$, в силу определения импликации. Заметим, что, поскольку $R'(u) \in R^t$, то по определению $\exists u' \in U(t)$, что $P'_{trg}(u')$. Следовательно, в дереве t присутствует хотя бы одна вершина, удовлетворяющая требованиям предиката положения рассматриваемого контекстного условия. В этом случае $R'(u)$ только в том случае, если выполняется следующее $F(u',\ u')$. Особенности проверки истинности предиката $F(u,\ u)$ будет рассмотрена позже, так как производится аналогично для контекстных условий всех типов. Сейчас рассмотрим особенности поиска решений для других типов контекстных условий.

Пусть в базовом множестве контекстных условий R^t дерева t присутствует контекстное условие типа *many-to-many*. Требуется определить, выполняются ли требования контекстного условия $R''''(u,v) \equiv \forall u,v \in U(t)(u \neq v \wedge P''''_{trg}(u) \wedge P''''_{src}(v) \Rightarrow F''''(u,v))$. Так как $R'''(u) \in R^t$, то по определению $\exists u''' \in U(t)\ |P_{trg}(u''') \vee \exists v''' \in U(t)\ |P_{src}(u,\ v''')$. Пусть, например, $\exists u''' \in U(t)\ |P_{trg}(u''')$ при этом, если $\forall v \in U(t)(\ u''' \neq v \wedge \sim P''''_{src}(v))$, то $R'''(u''',\ v)$. Аналогично, если $\exists v''' \in U(t)\ |\ P''''_{src}(v''')$ при этом, если $\forall u \in U(t)(\ u \neq v''' \wedge \sim P''''_{src}(v))$, то $R'''(u,\ v''')$. И только в случае, когда $(\exists u\ \forall v \in U(t) \vee \forall u, \exists v \in U(t) \vee \forall u,\ v \in U(t)) \wedge (u \neq v \wedge P''''_{trg}(u) \wedge P''''_{src}(v)))$ истинность $R'''(u,\ v)$ зависит от истинности предиката $F'''(u,\ v)$.

Рассмотрим теперь контекстные условия типа *one-to-many*. Пусть в базовом множестве контекстных условий R^t дерева t присутствует контекстное условие такого типа. Требуется определить, истинно ли контекстное условие R'', имеющее вид $R''(u) \equiv \forall u \in U(t)(\ P''_{trg}(u) \Rightarrow (\exists v \in U(t)$ $\wedge\ u \neq v \wedge P''_{src}(v) \wedge F''(u,v)))$. Поскольку $R''(u) \in R^t$, то по определению $\exists u'' \in U(t)$

такая, что $P''_{trg}(u'')$, следовательно, для того, чтобы $R''(u)$ в дереве t должна существовать вершина v, чтобы выполнялось следующее $(\exists v \in U(t) \wedge u \neq v \wedge P''_{src}(v) \wedge F''(u, v))$. Т.е. для каждой цели контекстного условия типа *one-to-many* в дереве t должен существовать источник. Если в ходе поиска решения для такого контекстного условия, выясняется, что для какой-то цели нет в дереве ни одного источника, то следует изменить дерево t так, чтобы в нем появился искомый источник.

Заметим, что для того, чтобы в результате получился тест, нацеленный на проверку определенного аспекта анализатора контекстных условий, первичное дерево, с которого началось построение, дерева t должно оставаться неизменным.

Для того чтобы в дереве t появилась требуемая вершина v необходимо последовательно обойти все вершины дерева, определяя можно ли добавить требуемую вершину v в качестве дочерней непосредственно к какому-нибудь узлу дерева или требуется достроить поддерево, содержащее вершину v. При построении вершины необходимо воспользоваться функциями вычисления значений переменных приведенных в таблице 15. На расположение переменной v в данном случае накладывается единственное требование, которое заключается в том, что v должна быть такой, чтобы $R(u)$. При этом не требуется перебирать все возможные способы расположения v в дереве, так как v не является первичным источником и в соответствии с гипотезой 2 любая вершина v, удовлетворяющая условиям $v \neq u$, $P_{src}(u, v) \wedge F(u, v)$ подойдет.

Если не удается изменить дерево так, чтобы в нем появилась требуемая вершина, то такое дерево называется *семантически неразрешимым*.[16] Это означает, что дальнейшая работа с ним не приведет к построению требуемых данных, поэтому это дерево следует отбросить и перейти к следующему.

Заметим, что при изменении дерева t может измениться базовое множество контекстных условий ему соответствующее. Пусть в результате

[16] Предполагается, что синтаксис и семантика заданы корректно.

изменений дерево t превратилось в дерево t', тогда $R^t \neq R^{t'}$, если $\exists u' \in U(t') \wedge u' \notin U(t) \wedge \exists R'(u) \in SR_{all} \backslash R^t \wedge P'_{trg}(u')$ или $\exists u'' \in U(t') \wedge u'' \notin U(t) \wedge \exists R''(u, v) \in SR_{all} \backslash R^t \wedge (P''_{trg}(u'') \vee P''_{src}(u''))$, в этом случае, $R^{t'} \backslash R^t = \{R'(u)\}$ или $R^{t'} \backslash R^t = \{R''(u, v)\}$.

После изменения дерева потребуется повторить шаг A1S6, так как в силу появления новых вершин в дереве необходимо проверить удовлетворяют ли они базовым правилам.

Итак, были рассмотрены особенности проверки справедливости контекстных условий для дерева t. В результате можно сделать следующий вывод. Если контекстное условие

- $R(u) \in R^t$ типа *one-node*, то для проверки справедливости утверждения $\forall u \in U(t)\ R(u)$ достаточно для всех вершин $u \in U(t)$ таких, что $P_{trg}(u)$ проверить справедливость $F(u, u)$;
- $R(u, v) \in R^t$ типа *many-to-many*, то для проверки справедливости утверждения $\forall u, v \in U(t)\ R(u, v)$ достаточно для всех вершин $u, v \in U(t)$ таких, что $P_{trg}(u) \wedge P_{src}(u, v)$ проверить справедливость $F(u, v)$;
- $R(u) \in R^t$ типа *one-to-many*, то для проверки справедливости утверждения $\forall u \in U(t)\ R(u)$ достаточно для всех вершин $u \in U(t)$ таких, что $P_{trg}(u)$ найти вершину $v \in U(t),\ v \neq u$ и такую, что $P_{src}(u, v) \wedge F(u, v)$.

Перейдем к проверке истинности предикатов $F(u, v)$, виды которых приведены в таблице 8. В предикатах $F(u, v)$ присутствуют кванторы существования из этого следует, что для того, чтобы $F(u, v)$ может потребоваться изменить дерево с тем, чтобы обеспечить в нем существование требуемых вершин. Действия по изменению дерева в данном случае аналогичны действиям, производимым для изменения дерева при рассмотрении проверки требований контекстных условий типа *one-to-many*.

А для того, чтобы выполнить требования $T^x==T^y$ и $T^x\sim=T^y$ можно использовать функции аналогичные тем, что приведены в таблице 15.

Таб. 16 Формулы вычисления поддерева требуемого вида

№	**Требование на вид поддерева**	**Функция вычисления поддерева требуемого вида**	**Описание**
1.	$T^x==T^y$	*setSubtree(x, T)*	функция подставляющая, если это возможно, в качестве поддерева с корнем в вершине x, дерево T.
2.	$T^x\sim=T^y$	*createNewSubtree(x, T)*	функция, строящая новое поддерево с корнем в вершине x отличное от дерева T.

В общем случае для контекстного условия *R(u)* типа *one-to-many* может существовать несколько вершин $v \in U(t)$, удовлетворяющих требованиям *R(u)*. Заметим, что все базовые контекстные условия дерева *t* должны быть одновременно истины на множестве вершин дерева *t*. Любая вершина дерева *t*, $\forall u, v \in U(t)$, должна быть решением системы

$$\begin{cases} R_1(u) \equiv TRUE \\ \dots \\ R_k(u) \equiv TRUE \\ R_{k+1}(u, v) \equiv TRUE \\ \dots \\ R_n(u, v) \equiv TRUE \end{cases} \quad (9)$$

где $R_i(u),\ R_j(u, v) \in R^t,\ i = \overline{1,k},\ j = \overline{k+1,n},\ n=|R^t|$.

Рассмотрим на примере ситуацию, когда в результате изменения дерева *t* соответствующая система вида (9) будет неразрешима.

Пример 11.

Рассмотрим язык *L* из примера 1. Добавим несколько синтаксических правил в грамматику языка *L*, а именно: добавим возможность инициализировать объявленную переменную некоторым числовым значением. Новая грамматика языка *L* представлена в Таб. 17.

Таб. 17 Новый синтаксис языка *L* в EBNF и TreeDL нотациях

Синтаксис: EBNF-описание	Синтаксис: TreeDL-описание

program ::= “program”
identifier “;” stmts;
	node Program {
	child Identifier id;
	child Stmts stmts;
	}

stmts ::= stmt
        | (stmt stmts);
	abstract node Stmts {}
	
	node CompoundStmts : Stmts {
	  child Stmt stmt;
	  child Stmts stmts;
	}
	abstract node Stmt : Stmts{}

stmt ::= declaration | expression;	abstract node Declaration : Stmt {

declaration ::= ((“var”
new_var_identifier)
|(“syn” new_syn_identifier
“=” var_identifier)) “;” ;
	child Identifier id;
	}
	node VarDecl : Declaration {}
	node SynDecl : Declaration {
	  child Identifier varId; }

new_var_identifier
::=<identifier>;
 	
new_syn_identifier
::=<identifier>;
var_identifier ::=
<identifier>;
	node Identifier
	{
	  attribute String id;
	}

expression ::= var_identifier
“:=” <digit> “;”;
	node Expression : Stmt
	{
	  child Identifier varId;
	  child Identifier digit;
	}

В связи с изменением грамматики *L*, семантика языка также подверглась изменениям. Добавились два новых контекстных условия:

1. Описание переменной должно предшествовать инициализации этой переменной.
2. Переменная должна быть проинициализирована до того, как для нее будет создан синоним.

Полное SRL-описание семантики L представлено в Таб. 18.

Таб. 18 Семантика языка L

№	Описание семантических правил языка L на SRL	Описание семантических правил языка L в виде формул

1.	`many-to-many relation R1 {`
	`“Имя переменной уникально”`
	`unequal`
	`target VarDecl { id }`
	`source VarDecl { id }`
	`}`
		$R_1(u, v) \equiv (u \neq v \wedge VarDecl(u)$
		$\wedge VarDecl(v) \Rightarrow (\exists x \in U(T^u)$
		$1.Identifier(u, x) \wedge id(x)$
		$\wedge \exists y \in U(T^v)\ 1.Identifier(v, y)$
		$\wedge id(y) \Rightarrow (T^x \sim= T^y)))$

2.	`many-to-many relation R2 {`
	`“Имя синонима уникально”`
	`unequal`
	`target SynDecl { id }`
	`source SynDecl { id }`
	`}`
		$R2(u, v) \equiv (u \neq v \wedge SynDecl(u)$
		$\wedge SynDecl(v) \Rightarrow (\exists x \in U(T^u)$
		$1.Identifier(u, x) \wedge id(x) \wedge \exists y \in U(T^v)$
		$1.Identifier(v, y) \wedge id(y) \Rightarrow (Tx \sim= Ty)))$

3.	`many-to-many relation R3 {`
	`“Имя любого синонима отличается от имени любой переменной”`
	`unequal`
	`target VarDecl { id }`
	`source SynDecl { id }`
	`}`
		$R3(u, v) \equiv (u \neq v \wedge VarDecl(u)$
		$\wedge SynDecl(v) \Rightarrow (\exists x \in U(T^u)$
		$1.Identifier(u, x) \wedge id(x) \wedge \exists y \in U(T^v)$
		$1.Identifier(v, y) \wedge id(y) \Rightarrow (Tx \sim= Ty)))$

№	**Описание семантических правил языка L на SRL**	**Описание семантических правил языка L в виде формул**

4.	one-to-many relation R4 {
	"Синоним может быть определен только для существующей переменной или синонима"
	equal
	target SynDecl { varId }
	source Declaration { id }
	}
		$R4(u) \equiv (SynDecl(u) \Rightarrow (\exists v \in U(t) \wedge u \neq v \wedge Declaration(v) \wedge \exists x \in U(T^u)\ 1.Identifier(u, x) \wedge varId(x) \wedge \exists y \in U(T^v)\ 1.Identifier(v, y) \wedge id(y) \Rightarrow (Tx==Ty)))$

5.	one-to-many relation R5 {
	"Описание переменной должно предшествовать инициализации этой переменной"
	ordered
	equal
	target Expression { varId }
	source VarDecl { id }
	}
		$R5(u) \equiv (Expression(u) \Rightarrow (\exists v \in U(t) \wedge u \neq v \wedge VarDecl(v) \wedge precede(u, v) \wedge \exists x \in U(T^u)\ 1.Identifier(u, x) \wedge varId(x) \wedge \exists y \in U(T^v)\ 1.Identifier(v, y) \wedge id(y) \Rightarrow (Tx==Ty)))$

6.	one-to-many relation R6 {
	"Переменная должна быть проинициализирована до того, как для нее будет создан синоним"
	ordered
	equal
	target SynDecl { varId }
	source Expression { varId }
	}
		$R6(u) \equiv (SynDecl(u) \Rightarrow (\exists v \in U(t) \wedge u \neq v \wedge Expression(v) \wedge precede(u, v) \wedge \exists x \in U(T^u)\ 1.Identifier(u, x) \wedge varId(x) \wedge \exists y \in U(T^v)\ 1.Identifier(v, y) \wedge varId(y) \Rightarrow (Tx==Ty)))$

Рассмотрим программу на языке *L*:

```
program P;
var <id1>;
var <id2>;                                   (10)
<id3> = 0;
syn <id4> = <id5>;
```

в которой семантически зависимые значения идентификаторов заменены условными обозначениями <id_i>.

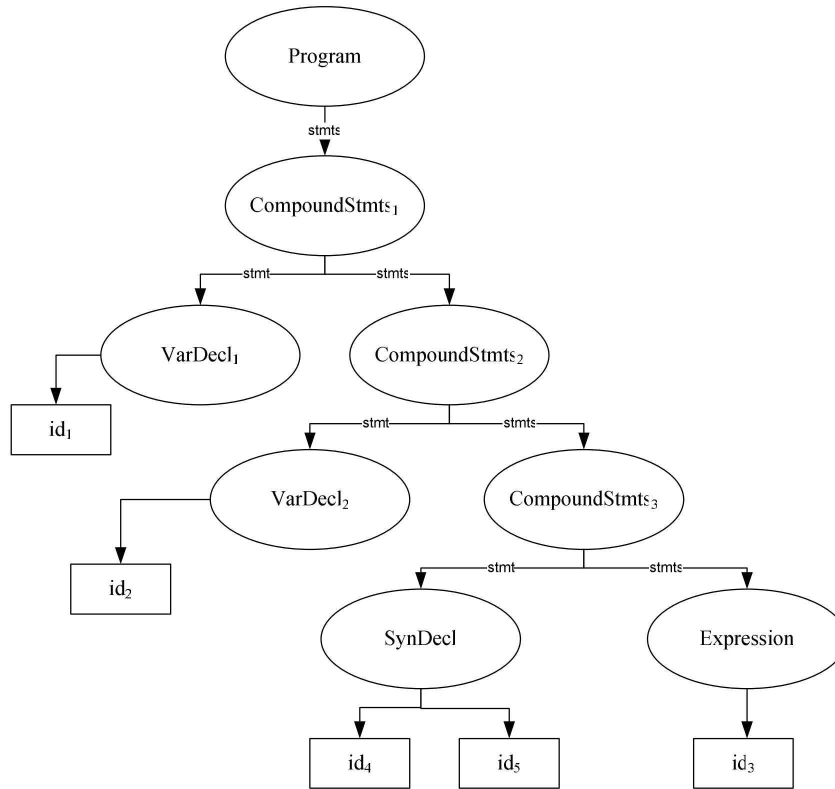


Рис. 10 Фрагмент дерева программы (10)

Фрагмент дерева *t* программы *p* представлен на *Рис. 10*. Овалами обозначены узлы дерева, внутри овалов написаны типы узлов. Дуги графа названы в соответствии с именами непосредственных потомков узлов из TreeDL описания языка. В виде прямоугольников представлены семантически зависимые вершины, названные в соответствии с условными обозначениями, использованными в (10).

В базовое множество контекстных условий программы *p* входят все правила из Таб. 18. Определим при каких значениях идентификаторов в программе (10) базовые контекстные условия Будут выполняться.

Система (9) для данного примера примет вид

$$\begin{cases} R_1(u, v) \equiv TRUE \\ R_2(u, v) \equiv TRUE \\ R_3(u, v) \equiv TRUE \\ R_4(u) \equiv TRUE \\ R_5(u) \equiv TRUE \\ R_6(u) \equiv TRUE \end{cases} \qquad (11)$$

Проанализируем правило $R_1(u, v)$. За исключением вершин $VarDecl_1$ и $VarDecl_2$, для всех остальных вершин дерева t посылка в $R_1(u, v)$ ложна, из чего следует, что само правило $R_1(u, v)$ всегда истинно. Для того чтобы $R_1(VarDecl_1, VarDecl_2)$ необходимо проверить истинно ли следствие, т.е. выполняется ли следующее требование $(\exists x \in U(T^u)\ id(u, x) \wedge \exists y \in U(T^v)\ id(v, y) \Rightarrow (T^x \sim= T^y))$ при $u=VarDecl_1$ и $v=VarDecl_2$. Посылка в этом требовании истинна, так как и у вершины $VarDecl_1$ и у вершины $VarDecl_2$ есть дочерние вершины с именем id. Осталось проверить следствие, если $T^x \sim= T^y$ истинно при $x=id_1$, $y=id_2$, тогда верно $R_1(VarDecl_1, VarDecl_2)$. В данном примере $T^x \sim= T^y$, если $id_1 \neq id_2$.

Для $R_1(VarDecl_2, VarDecl_1)$ справедливы такие же рассуждения. В результате получим, что $R_1(VarDecl_2, VarDecl_1)$, если $id_2 \neq id_1$. Таким образом, если $R_1(VarDecl_1, VarDecl_2)$, то $R_1(VarDecl_2, VarDecl_1)$, следовательно, достаточно проверить только одно утверждение, например первое, для которого необходимым условием истинности является $id_1 \neq id_2$. Заменим в системе (11) первое уравнение необходимым условием его истинности. Система примет вид

$$\begin{cases} id_1 \neq id_2 \\ R_2(u, v) \equiv TRUE \\ R_3(u, v) \equiv TRUE \\ R_4(u) \equiv TRUE \\ R_5(u) \equiv TRUE \\ R_6(u) \equiv TRUE \end{cases} \qquad (12)$$

Проведя аналогичные рассуждения для контекстных условий $R_2(u, v)$ и $R_3(u, v)$ и заменив в системе (12) соответствующе уравнения необходимыми условиями их истинности получим

$$\begin{cases} id_1 \neq id_2 \\ id_1 \neq id_4 \\ id_2 \neq id_4 \\ R_4(u) \equiv TRUE \\ R_5(u) \equiv TRUE \\ R_6(u) \equiv TRUE \end{cases} \tag{13}$$

Рассмотрим правило $R_4(u)$. В дереве t программы p существует две вершины $VarDecl_1$ и $VarDecl_2$, удовлетворяющие требованиям, заданным в следствие $R_4(u)$ по отношению к вершине v: $\exists v \in U(t) \wedge u \neq v \wedge Declaration(v)$. Рассуждая аналогично тому, как это было сделано для правил $R_1(u, v)$, $R_2(u, v)$, $R_3(u, v)$, получим, что необходимым условием истинности $R_4(u)$ будет следующее: $id_1 = id_3 \vee id_2 = id_3$.

Рассуждая аналогичным образом, получим необходимые условия истинности правил $R_5(u)$ и $R_6(u)$: $id_1 = id_5 \vee id_2 = id_5$ и $id_3 = id_5$.

Заменяя в системе (13) уравнения соответствующими необходимыми условиями их истинности получим несколько систем, решение каждой из которых позволит получить такие значения id_1, id_2, id_3, id_4, id_5, что $\forall u,v \in U(t)$ будут решениями исходной системы (11). Системы необходимых условий истинности выглядят следующим образом

$$\begin{cases} id_1 \neq id_2 \\ id_1 \neq id_4 \\ id_2 \neq id_4 \\ id_1 = id_3 \\ id_1 = id_5 \\ id_3 = id_5 \end{cases} \qquad \begin{cases} id_1 \neq id_2 \\ id_1 \neq id_4 \\ id_2 \neq id_4 \\ id_1 = id_3 \\ id_2 = id_5 \\ id_3 = id_5 \end{cases} \qquad \begin{cases} id_1 \neq id_2 \\ id_1 \neq id_4 \\ id_2 \neq id_4 \\ id_2 = id_3 \\ id_1 = id_5 \\ id_3 = id_5 \end{cases} \qquad \begin{cases} id_1 \neq id_2 \\ id_1 \neq id_4 \\ id_2 \neq id_4 \\ id_2 = id_3 \\ id_2 = id_5 \\ id_3 = id_5 \end{cases}$$

(14) (15) (16) (17)

Заметим, что системы (15), (16) не имеют решения. Следовательно, значения id_1, id_2, id_3, id_4, id_5 надо искать посредством решения одной из

систем (14) или (17). Подставив полученные значения в программу (10), получим две семантически корректные программы:

```
program P;
var a;
var b;
a = 0;
syn c = a;
```

```
program P;
var a;
var b;
b = 0;
syn c = b;
```

Подобные рассуждения применимы для базовых контекстных условий, которые не являются первичными правилами, следовательно, в соответствии с гипотезой 2 обе эти программы относятся к одному классу эквивалентности и достаточно построить только одну из них.

3.3 Генерация негативных тестовых входных данных на основе конструктивного описания статической семантики

Предлагаемая технология позволяет генерировать входные данные не только для *позитивных*, но и для *негативных* тестов. Для этого необходимо создать описания мутаций контекстных условий. Затем, используя алгоритм 1, запустить процесс генерации тестов, выбирая в качестве первичного правила одну из мутаций правил [17].

Пример 12.

Модифицируем некоторые правила из примера 11. Результат приводится в Таб. 19.

---

[17] Благодаря простоте описания правил на языке SRL, процесс создания мутаций правил можно автоматизировать.

Таб. 19 Пример мутаций контекстных условий.

№	Семантические правила	Мутации семантических правил

1.	`many-to-many relation R3 {`
	`“Имя любого синонима отличается от имени любой переменной”`
	`unordered`
	`unequal`
	`target VarDecl { id }`
	`source SynDecl { synId }`
	`}`
		`one-to-many relation R3rev {`
		`“Имя некоторого синонима совпадает с именем некоторой переменной”`
		`unordered`
		`equal`
		`target VarDecl { id }`
		`source SynDecl { synId }`
		`}`

2.	`one-to-many relation R4 {`
	`“Синоним может быть определен только для уже определенной переменной ”`
	`ordered`
	`equal`
	`target SynDecl { varId }`
	`source VarDecl { id }`
	`}`
		`many-to-many relation R4rev {`
		`“В определении синонима не должно встречаться имя никакой из переменных, ранее определенных. ”`
		`ordered`
		`unequal`
		`target SynDecl { varId }`
		`source VarDecl { id }`
		`}`

Пусть G ‒ однозначная КС-грамматика, L – некоторый формальный язык, порожденный G, $M(G)$ ‒ лес деревьев и поддеревьев соответствующих всем фразам языка L, SR_{all} – множество контекстных условий языка L, $S\tilde{R}_{all}$ - множество мутаций контекстных условий языка L.

Определение 18. Будем называть *мутацией контекстного условия* R' некоторое контекстное условие $\tilde{R}'$ такое, что если некоторый текст p на языке $L(G)$ удовлетворяет $\tilde{R}'$, то p не удовлетворяет R' и наоборот, т.е. $\forall t \in M(G)$ $\forall u, v \in U(t)$ $\forall R'(u)$, $R''(u, v) \in SR_{all}$, $\forall \tilde{R}'(u)$, $\tilde{R}'(u, v) \in S\tilde{R}_{all}$ $\tilde{R}'(u) \Leftrightarrow$ $\sim R'(u)$ и $\tilde{R}''(u, v) \Leftrightarrow \sim R''(u, v)$.

Рассмотрим язык L', имеющий синтаксис такой же, как и язык L, и множество контекстных условий $SR_{all} \backslash R_i \cup \tilde{R}_i$. Тогда, используя приведенный ранее метод генерации тестовых текстов (программ) для транслятора с языка

L', задавая при этом в качестве первичного правила - $\tilde{R}_i$, можно построить набор негативных тестов N_i, нацеленных на проверку способности транслятора языка L обнаруживать ошибки в программах, вызванные нарушениями контекстного условия R_i. В результате будет получено множество негативных тестов $N_L = \bigcup_{1 \le i \le |SR_{all} \setminus R_i \cup \tilde{R}_i|} N_i$ для транслятора языка L, направленных на проверку правильности поведения транслятора при встрече одиночного нарушения некоторого контекстного условия.

4 SRL: описание нетривиальных контекстных условий

В этой главе на примере контекстных условий реальных языков программирования, таких как Java, будут рассмотрены средства языка SRL, предназначенные для описания сложных контекстных условий. В том числе будут рассмотрены способы формализации следующих групп правил:

- контекстных условий, связанных с механизмом наследования в объектно-ориентированных языках;
- правил, требующих проверки дополнительных условий;
- правил, описывающих условия при которых нельзя использовать некоторые синтаксические конструкции;
- контекстных условий, описывающих правила приведения типов.

4.1 Контекст семантического правила

Начнем с рассмотрения примера. В спецификации языка Java [36] описано контекстное условие: *все классы верхнего уровня в одном пакете должны иметь уникальные имена*. Ниже приведен фрагмент синтаксиса Java на языке TreeDL, описывающий способ декларации классов в пакете.

```
node RootNode : <BaseNode>
{
  child PackageNode+ packages;
}
node PackageNode : <BaseNode>
{
  child PackageDeclaration packDeclaration;
  child CompilationUnit* units;
}
node CompilationUnit : <BaseNode>
{
  child ImportDeclaration* optImportDeclarationList;
  child ClassDeclaration classDecl;
}
```

```
node ImportDeclaration : <BaseNode>
{
  PackageName packName;
  Identifier typeName;
}
node ClassDeclaration : <BaseNode>
{
  child ClassModifiers modifiers;
  child Identifier ident;
  child Identifier? superClassIdent;
  child Identifier* superInterfaceList;
  child ClassBody classBody;
}
node ClassModifiers : <BaseNode>
{
  child PublicModifier? isPublic;
  child PrivateModifier? isPrivate;
  child ProtectedModifier? isProtected;
  child AbstractModifier? isAbstract;
  child StaticModifier? isStatic;
  child FinalModifier? isFinal;
  child StrictfpModifier? isStrictfp;
}
```

Данное правило можно интерпретировать следующим образом: если в рамках одной и той же конструкции типа *PackageNode* присутствуют две конструкции типа *ClassDeclaration* (*источник* и *цель* в терминах конструктивного описания), то идентификаторы, конструкции типа *Identifier* (*атрибуты* источника и цели), являющиеся их дочерними узлами, должны иметь разные значения.

При этом заметим, что синтаксис позволяет строить такие тесты, в которых декларации классов будут находиться в разных пакетах. Однако такие тесты не представляют интереса с точки зрения проверки правильности

реализации рассматриваемого правила в трансляторе с языка Java. В связи с этим требуются средства для уточнения описания контекстного условия.

Определение 19. Пусть L – это некоторый формальный язык. Пусть p – это некоторый текст на языке L, а R – это контекстное условие из базового множества правил текста p. Будем называть *контекстом источника (цели) семантического правила* R поддерево синтаксического дерева текста p, в котором должен находиться *узел-источник* (*узел-цель*), для которого должно выполняться правило R.

В синтаксисе SRL описание контекста выглядит следующим образом:

```
context ::= "context" ":" (  sngl_context
     | dbl_context );
sngl_context ::= "same" <nonterm_type>;
dbl_context ::= "differ"                                          (18)
                    "source_context" <nonterm_type>
                    "target_context" <nonterm_type>;
```

Контекст характеризуется корневой вершиной (*<nonterm_type>*). Кроме того, контексты источника и цели могут совпадать (как в правиле *об уникальности имен классов в одном пакете*), для этого используется конструкция *sngl_context*, а могут быть различными – *dbl_context*.

Например, в правиле, описывающем отношения между декларацией класса и заданием имени базового класса, можно потребовать, чтобы источник (*ClassDeclaration*) находился в одном контексте (корень которого совпадает с источником), а цель – находилась в другом контексте (т.е. в декларации другого класса).

Ниже (на языке SRL) приводятся описания контекстных условий *об уникальности имен классов, находящихся в одном пакете* и *о существовании декларации класса, являющегося базовым для другого класса*:

```
many_to_many relation R1
{
comment "Все классы верхнего уровня в одном пакете
должны иметь уникальные имена."
unequal                                                    (19)
target ClassDeclaration { ident }
source ClassDeclaration { ident }
context : same PackageNode
}
one_to_many relation R2
{
comment "Любой базовый класс должен быть описан."
equal
target ClassDeclaration { superClassIdent }                (20)
source ClassDeclaration { ident }
context : differ source_context PackageNode
differ target_context PackageNode
}
```

Введение понятия *контекст* позволяет более точно строить синтаксические деревья тестовых текстов на формальных языках.

4.2 Семантические правила с фильтрами

В спецификации языка Java [36] описано семантическое правило: *из заданных в другом пакете классов разрешается импортировать только классы с модификатором public*. В терминах SRL данное правило означает, что для каждого узла-цели типа *ImportDeclaration* должен существовать узел-источник типа *ClassDeclaration*. Причем в данном случае на роль источника нельзя отбирать узлы, используя только их тип. Существует некоторое дополнительное условие на вид узла-источника: *класс должен быть объявлен с модификатором public*. Следовательно, в дочернем узле (*ClassModifier*) узла-источника (*ClassDeclaration*) должен присутствовать

дочерний узел типа *PublicModifier*. Источник и цель должны находиться в разных пакетах.

Для того, чтобы описывать подобные семантические правила в SRL была введена конструкция, названная *фильтром*:

```
nonterm_type ::= <nonterm_type> (filter)?
  "{"
       <attr_name>("," <attr_name>)*
  "}";
filter ::= "[" <child_id>("."<child_id>)*
(
  ("==" ( value | "null"))
  | ("!=" ( value | "null"))                          (21)
  | ("is" <nodeId>)
  | ("notis" <nodeId>)
  | (( "in" | "notin" ) set)
) "]";
value ::= <str_value>;
set ::= "{" <str_value> ("," <str_value> )* " "}";
```

Фильтр может быть написан после типа источника или цели в виде некоторого предиката, аргументом которого является узел-потомок источника или цели, к которому известен путь в виде цепочки имен узлов-потомков источника или цели, разделенных точками: `<child_id>("."<child_id>)*`.

С помощью фильтров в SRL:

- можно проверить эквивалентность/неэквивалентность некоторому значению (`== value | != value`);
- наличие/отсутствие узла-потомка в дереве (`== null | != null`);
- тип узла-потомка (`is <nodeId> | notis <nodeId>`);
- принадлежность некоторому множеству (`"in" set | "notin" set`).

Ниже приводится описание семантического правила *об импорте классов* на языке SRL.

```
one_to_many relation R3
{
  comment "Из другого пакета можно импортировать
только public классы."
  unordered
  equal
  target ImportDeclaration { typeName }
  source ClassDeclaration[modifiers.isPublic!=null]
{ ident }
  context : differ source_context PackageNode
differ target_context PackageNode
}
```

(22)

4.3 Зависимые семантические правила

В описании семантического правила разрешено ссылаться на объекты (источник, цель, корень контекста) другого семантического правила, в этом случае называемого *базовым*. Рассмотрим пример.

В программах, написанных на языке Java, для задания типа объекта можно использовать полное имя типа. Полное имя (*QualifiedName*) типа верхнего уровня состоит из имени пакета, в котором этот тип описан и простого имени типа, разделенных точкой: *PackageName.Identifier*. Если речь идет о типе, объявленном как вложенный класс или интерфейс, тогда между именем пакета и собственно именем типа должны указывать соответствующие имена объемлющих типов [36]. В этом примере рассмотрим типы верхнего уровня.

Ниже приводится фрагмент TreeDL-описания языка Java, задающий имена, используемые далее в примере.

```
abstract node TypeDeclaration
{
   child Modifiers modifiers;
   child Identifier name;
}
node PackageDeclaration
{
  child Identifier name;
  child TypeDeclaration typeDecl;
}
node QualifiedName
{
  child PackageName name;
  child Identifier ident;
}
abstract node PackageName
{
  child Identifier ident;
}
node Identifier
{
    attribute <String> value;
}
```

Также в спецификации Java существует требование на существование пакета, имя которого затем упоминается. Опишем это правило на SRL (см. (23)).

```
one_to_many relation R5
{
  comment "Имя пакета должно декларироваться перед
использованием."
  ordered
  equal
  target PackageName { ident }
  source PackageDeclaration { name }
}
```

(23)

Правило (23) необходимо нам для того, чтобы описать требование на существование декларации типа, полное имя которого используется в некотором контексте.

В данном случае недостаточно указать в качестве цели *QualifiedName*, а в качестве источника – *TypeDeclaration*, так как семантически неверно использовать в качестве значения *Identifier* в полном имени типа – имя любого описанного типа. В данном случае множество возможных значений ограничивается именами типов, описанных в пакете *(PackageDeclaration)*, имя которого является префиксом полного имени типа.

Для того, чтобы получить множество допустимых имен типов необходимо сослаться на правило, описывающее связь между декларацией пакета и использованием его имени (23). Используем механизм зависимости между семантическими связями, предоставляемый SRL.

```
one_to_many relation R6 : R5
{
  comment "Имя типа должно декларироваться перед
использованием."
  ordered                                          (24)
  equal
  target super.target.parent { ident }
  source super.source.TypeDeclaration { name }
}
```

При описании правила (24) использовались несколько новых механизмов SRL, а именно: *установление зависимости между семантическими правилами* и *описание источника и цели при помощи указания пути*.

Для того, чтобы установить зависимость между семантическими правилами, необходимо при описании одного семантического правила после указания его идентификатора задать идентификатор семантического правила, на которое предполагается ссылаться далее при описании данного правила.

```
relation ::= ("many-to-many"|"one-to-many")
            "relation" <new_rule_id> :<base_rule_id>
                "{"  (comment)?
                     ("ordered" | "unordered")
                      ("unequal" | "equal")                        (25)
                      "target" nonterm_type
                      "source" nonterm_type
                "}";
```

Определение 20. Семантическое правило, идентификатор которого указан после двоеточия, будем называть *базовым* по отношению к данному семантическому правилу.

В контексте установленной зависимости между правилами данное правило будем называть *зависимым* по отношению к базовому, а базовое – *независимым*.

Любое семантическое правило может иметь только одно *базовое правило* и сколько угодно *зависимых*.[18] Кроме того, появление нескольких базовых правил значительно усложнило бы задачу трансляции SRL.

Естественным требованием является запрет на существование взаимных зависимостей между семантическими правилами. Ориентированный граф зависимостей контекстных условий не должен иметь циклов.

Наличие у данного семантического правила базового правила позволяет ссылаться на объекты (источник, цель и корень контекста) базового правила при описании объектов данного правила.

Транслятор SRL воспринимает в качестве ссылки на базовое правило ключевое слово *super*, за которым через точку должно следовать одно из ключевых слов: *source*, *target*, *source_context*, *target_context*, указывающее на источник, цель, корень контекста источника или корень контекста цели соответственно.

---

[18] Как показывает практика, достаточно поддерживать возможность задания одного базового правила для описания статической семантики языков программирования таких, как Java, C, C#.

Если в описании, например, цели данного семантического правила указан путь *super.target*, это означает, что, если данное семантическое правило входит в базовое множество контекстных условий некоторого синтаксического дерева, то, во-первых, в базовое множество правил также входит и базовое правило данного семантического правила; во-вторых, данное правило должно выполняться для всех вершин синтаксического дерева, являющихся целью базового правила. Аналогичный смысл имеют и другие ключевые слова.

Рассмотрим пример описания правила (24). Описание цели семантического правила не ограничивается ссылкой на цель базового правила. После ключевого слова *target* через точку указано ключевое слово *parent*, в данном случае указывающее на родительскую вершину узла, являющегося целью базового правила. В данном примере путь *super.target.parent* указывает на вершину типа *QualifiedName*, дочерние узлы которой могут быть двух типов *Identifier* и *PackageName*. Следовательно, логично разрешить указывать ссылку на дочерний узел, если тип узла, на который указывает данный путь однозначно определен.

Таким образом путь *super.target.parent.Identifier*, использованный в описании цели семантического правила (24), указывает на вершину типа *Identifier* дочернюю по отношению к вершине типа *QualifiedName*.

Аналогичным образом в (24) задано описание источника: *super.source.TypeDeclaration*. В качестве источника правила *R6* может использоваться вершина типа *TypeDeclaration* дочерняя по отношению к узлу типа *PackageDeclaration*, который является источником в базовом семантическом правиле. Таким образом семантическое правило (24) связывает простые имена типов, указанные после имени пакета, только с теми декларациями типов, которые объявлены в этом пакете.

4.4 Описание объектов семантического правила посредством задания пути

Путь представляет собой последовательность ключевых слов, типов и имен вершин, идентификаторов контекстных условий, соединенных точками. Далее эти объекты будем называть *элементами* пути.

Каждая часть пути от начала до любого элемента пути обозначает какой-то объект грамматики входного языка. Далее будем называть такие части пути *ссылками*.

Существуют два вида ссылок: на вершины синтаксического дерева и на семантические правила.

Ссылки на вершины бывают:

- по типу;
- по ключевым словам (например: *source, target*).

Ссылки на семантические правила различаются:

- по идентификатору;
- по ключевым словам (например: *this, super*).

Кроме ссылок на объекты в описании пути используются также и фильтры. Фильтры описываются правилами, заключенными в квадратные скобки, и следуют за соответствующей ссылкой вплотную, не отделяясь от нее точкой. Существуют фильтры для вершин и для контекстных условий.

В фильтрах можно проверить:

- эквивалентность другой вершине или *null*;
- соответствие типу;
- принадлежность какому-то множеству.

Конструкция, описывающая правила фильтрации для вершин, выглядит следующим образом:

```
path_elem_filter ::=   ("==" (path | "null"))
                      | ("!=" (path | "null"))
                      | ( "is" node_type )                    (26)
                      | ( "notis" node_type )
                      | (( "in" | "notin" ) path);
```

Существуют три вида правил фильтрации:

1. Проверка на эквивалентность/неэквивалентность (==, !=) некоторой вершине дерева, месторасположение которой задается путем (*path*) или проверка на эквивалентность пустой вершине (==, != *null*).
2. Проверка на соответствие (*is/notis*) к типу (*node_type*) вершин.
3. Проверка на вхождение (*in/notin*) во множество вершин, месторасположение которых описывается путем (*path*). Генератор создает коллекцию из вершин, найденных по указанному пути. Коллекция может состоять из нескольких вершин, только из одной вершины или быть пустой.

Далее следует подробное описание пути к вершинам–объектам семантического правила.

Путь задается следующим образом

```
path ::= (
          (
           (
              (start_rule ".")+
            | (mid_rule (super)* ".")                        (27)
           )
          ) rule_obj("[" path_elem_filter "]")? "."
         )?
         (path_elem)?("." path_elem)*;
```

Описание пути может начинаться с конструкции

```
start_rule ::= super | this ;
```

описывающей ссылку на базовое правило (*super*) или на данное описываемое правило (*this*), или с конструкции

```
mid_rule ::= rule_id "[" rule_filter "]";
```

указывающей генератору на необходимость рассмотрения контекстных условий с идентификатором *rule_id*:

```
rule_id ::= <RULE_ID>;
```

Семантическое правило с идентификатором *rule_id* должно быть базовым для рассматриваемого семантического правила.

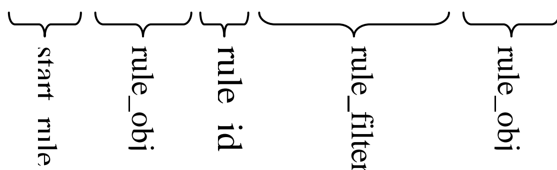
Можно сузить множество рассматриваемых базовых контекстных условий посредством введения правил фильтрации:

```
rule_filter ::= rule_obj "== prev";
rule_obj ::= "source"
           | "target"
           | "source_context"
           | "target_context";
```

Такие правила фильтрации могут вводиться только для ссылок на семантические правила и только, если конструкции *mid_rule* предшествует другая ссылка, причем на вершину, а не на семантическое правило. Тогда конструкция *rule_obj "== prev"* означает следующее: источник (цель или корень контекста) совпадает с вершиной, ссылка на который предшествует *mid_rule*.

Пример 13.

```
super.source.S4[target==prev].source
```



Этот описание предписывает генератору взять базовое правило (*super*), взять у него источник (*source*); затем взять семантическое правило с идентификатором *S4,* определенное для текущего синтаксического дерева, такое, чтобы полученная перед этим вершина выступала в этом правиле в качестве цели (*target==prev*); у полученного семантического правила взять источник.

Рассмотрим более подробно конструкцию, которая описывает элементы пути.

```
path_elem ::= (mid_rule (super_r)* "." rule_obj)
            | elem
            | desc_elem                                    (28)
            | ances_elem
            | recur_set;
```

Первая альтернатива данного продукционного правила

```
mid_rule (super_r)* "." rule_obj
```

описывает последовательность ссылок, указывающих на вершину–объект семантического правила (источник, цель корень контекста). Конструкция

```
super_r :: = "." super
```

описывает ссылку на базовое правило семантического правила, на которое ссылается *mid_rule*. У базового правила в свою очередь может существовать свое базовое правило, поэтому допустимо писать, например: *S4.super.super.super*.

Ссылаться при помощи ключевого слова *super* на базовое правило можно только, если перед этим стоит ссылка на зависимое семантическое правило.

Вторая альтернатива продукционного правила *path_elem* описывает ссылку по имени на вершины дочерние вершине, на которую указывает предшествующая ссылка.

```
elem ::= field_id("[" path_elem_filter "]")?;
```

Конструкция *elem* может использоваться только после ссылки на вершину.

Ссылка на дочернюю вершину состоит из имени *field_id*

```
field_id ::= <FIELD_ID>;
```

и правил фильтрации, которые являются не обязательными.

У вершины, на которую указывает предшествующая ссылка, в TreeDL–файле должна быть описана дочерняя вершина (*child*) с именем *field_id.*

Правила фильтрации описываются конструкцией *path_elem_filter* (26).

Третья альтернатива продукционного правила *path_elem* описывает ссылку на вершину, находящуюся в поддереве, корнем которого является вершина, на которую ссылается предыдущая ссылка в пути.

```
desc_elem ::=
(( ("<" | "<=" | ">" | ">=")? <number>) | "00")                (29)
"." <nodeId:ID> ("[" path_elem_filter "]")?;
```

Для описания такой вершины требуется указать тип дочерней вершины (<nodeId:ID>), ограничений на глубину поиска (<number>) и условие фильтрации. Причем, существует возможность указать верхнюю (<, <=), нижнюю (>, >=) границу поиска, или вообще не ограничивать глубину (“00”).

Четвертая альтернатива продукционного правила *path_elem* описывает ссылку на вершину, являющуюся вершиной–предком для вершины, на которую указывает предыдущая ссылка

```
ances_elem ::= ances ("[" path_elem_filter "]")?;
ances ::= "parent" | "^" node_type;
```

Ссылка на вершину предка бывает двух видов:

- по ключевому слову *parent,* указывает на непосредственного родителя вершины, на которую указывает предшествующий элемент пути;

- по типу *"^" node_type,* указывает на ближайшего предка указанного типа.

Ссылка на предка может завершаться фильтром *path_elem_filter*.

Пятая альтернатива продукционного правила *path_elem* позволяет задавать рекурсивное описание последовательности ссылок на вершины

```
recur_set ::= "(" recur_elem ("." recur_elem)* ")*";
```

Конструкция *recur_set* обязана предваряться описанием ссылки на вершину.

Ссылка на вершину, предшествующая *recur_set,* используется в качестве начального элемента последовательности ссылок, описываемой посредством *recur_set*.

Пример 14.

Рассмотрим в качестве примера следующее описания пути. Пусть в TreeDL-описании грамматики языка заданы следующие описания вершин

```
node QualifiedName
{
 child CompoundPackageName packName;
 child Identifier typeName;
}
abstract node PackageName
{
 child Identifier ident;
}
node SimplePackageName : PackageName
{
}
node CompoundPackageName : PackageName
{
 child PackageName packageName;
}
```

Тогда при описании некоторого семантического правила на SRL может быть указан путь

```
QualifiedName.1:CompoundPackageName.(1:CompoundPackageName)*
```

Нулевым элементом в строящейся последовательности выступит вершина, имеющая тип *CompoundPackageName* и являющаяся дочерней вершине типа *QualifiedName*. N-ым элементом будет вершина, имеющая тип *CompoundPackageName* и являющаяся дочерней по отношению к (n-1)-ому элементу последовательности вершин.

Элементами конструкции *recur_set* могут выступать уже рассмотренные ранее конструкции

```
recur_elem ::= (mid_rule (super_r)* "." rule_obj)
             | elem
             | ances_elem;
```

Пример 15.

Рассмотрим рекурсивное описание пути на SRL.

```
super.target.(S4[source == prev].target.className.1.Identifier)*
```

Используя такое рекурсивное описание, генератор построит следующую последовательность вершин:

- 0-ой элемент: цель базового правила;
- 1-ый элемент: ищется семантическое правило с идентификатором *S4* такое, чтобы его источник совпадал с нулевым элементом последовательности; ищется цель полученного семантического правила, у которого, в свою очередь, ищется дочерняя вершина с именем *className;* затем определяется потомок последней найденной вершины типа *Identifier*, расположенный на глубине равной единице.
- i-ый элемент: строится как 1-ый элемент с той лишь разницей, что ищется семантическое правило с идентификаторов *S4*, источник в котором совпадает с (i-1)-ым элементом.

Последовательность строится до тех пор пока при поиске очередного элемента последовательности не будет получен *null,* или пока не будет превышен заданный уровень глубины рекурсии.

Может так случиться, что при поиске k-ого элемента будет найдено несколько вершин, соответствующих описанию (например: если существует несколько семантических связей с одинаковыми идентификаторами и источниками). В этом случае последовательность разветвляется, и все найденные элементы образуют свою ветвь последовательности. Построение продолжается.

Рекурсивное описание последовательности ссылок используется для разных целей, например:

- для описания рекурсивных семантических зависимостей между источниками и целями;
- для описания множества вершин в фильтре.

4.5 Семантика типов

Для того чтобы автоматизировать с помощью STG описание правил семантики статически типизируемых языков программирования, в SRL были введены понятия *семантический тип, семейство типов, семантически типизированная вершина, совместимость* и *приведение типов*.

Вершины в TreeDL–описании, моделирующие типы входного языка, должны наследовать базовый тип вершин *ru.ispras.redverst.stg.semrel.TypeDesc*.

Пример 16.

Рассмотрим пример такое описания на языке TreeDL.

```
abstract node Type : <TypeDesc> {}
abstract node VarType : Type{}
node PrimitiveType : VarType
{
    attribute enum { DOUBLE, INT } enumPrimitiveType;
}
node ClassType : VarType
{
    child ID classname_id;
}
```

Кроме того для вершин, моделирующих конструкции входного языка, для которых определено понятие типа, должна быть определена дочерняя вершина с именем *type*, имеющая тип унаследованный от *TypeDesc*. Такую дочернюю вершину будем называть *семантическим типом* вершины.

Пример 17.

Семантический тип задан в следующем описании на TreeDL

```
abstract node OOLValue
{
    child Type type;
}
node Value_d_0 : OOLValue
{
    attribute <Integer> int1;
}
node VarUse : OOLValue
{
    child ID varname_id;
}
```

Вершины, для которых определен семантический тип будем называть *семантически типизированными.*

Все типы данных, существующие во входном языке, подразделяются на *семейства типов,* которые описываются с помощью конструкции языка SRL рядом с семантическими правилами.

```
type_set ::= "type_set" <type_set_id>
"{"
    type_desc
  | type_group
  (","  type_desc
     | type_group )*
"}";
```

В одном семействе типов должны находиться *совместимые* (в терминах семантики входного языка) типы данных.

Типы данных в семействе делятся на *группы эквивалентных типов*, состоящие из одного

```
type_desc ::= (<type_id>("["path_elem_filter"]")?("." path_elem)+ )
            | (<type_enum_id> "." type_enum_const);
```

или нескольких типов

```
type_group ::= "{" type_desc ("," type_desc)* "}";
```

Тип данных считается *приводимым* (в терминах семантики входного языка) ко всем типам данным, не относящимся к той же группе эквивалентных типов и стоящим в описании семейства типов правее него.

Пример 18.

Рассмотрим примеры описания семейств типов.

Семейства предопределенных типов известны заранее и могут быть описаны следующим образом.

```
typeset IntegralTypes { PrimitiveType.BYTE
                      , PrimitiveType.SHORT
                      , PrimitiveType.INT
                      , PrimitiveType.LONG }
```

Если при описании семейства типов используются ссылки на вершины дерева, описывающие пользовательские типы входного языка, то такое семейства типов не может быть описано предыдущим способом, т.к. заранее названия типов неизвестны и определяются динамически при построении дерева и вычислении семантических атрибутов. Описание семейства пользовательских типов может выглядеть так:

```
typeset ObjectTypes {  Class[notin S2.source]
                       .S4[source==prev].target
                       .(parent[is Class]
                       .S4[source==prev].target)* }
```

4.6 Семантические ограничения на синтаксис

В семантике языков программирования встречаются правила, которые можно было бы назвать *семантическими ограничениями на синтаксис*. Речь идет, например, о правиле запрещающем использовать оператор *break* вне тела цикла и оператора *switch*.

Суть их сводится к следующему. Такие ограничения указывают, вершины каких типов не могут находиться в отношении потомок-предок, а какие обязаны находится в таких отношениях.

Для формализации таких правил в языке SRL существует специальная конструкция

```
sscl ::= (node_ssconstraint)*;
```

При формальном описании семантических ограничений на синтаксис каждое такое ограничение определяет правильность синтаксического окружения для вершин определенного типа:

```
node_ssconstraint ::= "node" <node_type> "{"(ssconstraint)+ "}";
```

При описании вершины под *node_type* подразумевается идентификатор вершины, который должен быть описан в соответствующем TreeDL–файле.

Далее в фигурных скобках следует описание всех ограничений, которые должны проверяться для вершин указанного типа при их добавлении в синтаксическое дерево строящейся тестовой программы.

```
ssconstraint ::= "ssconstraint"
                      "{"
                      (<comment>)?
                      "forbidden" | "mandatory"
                      "ancestor" <node_type>
                      ("irrelevant" "("node_list")")?
                      ("relevant"   "("node_list")")?
                      "horizontal"
                      |( "vertical"
                        ("depth" <NUMBER>)?
                        )
                      "}";
node_list ::= (<node_type> (","<node_type>)*)?;
```

Для каждого ограничения следует указать его тип:

- *forbidden* (запрещенный) – правило запрещает синтаксические связи такого вида;
- *mandatory* (обязательный) – правило обязывает вершины указанного типа находится в описанном синтаксическом контексте.

Далее следует описание *шаблона синтаксического окружения*.

После ключевого слова *ancestor* следует тип вершины-предка по отношению к вершине описываемого типа.

Шаблоны подразделяются на *вертикальные* и *горизонтальные*.

Для горизонтальных шаблонов значение *depth* всегда равно единице .

Для горизонтальных шаблонов *irrelevant* (*relevant*) - список типов вершин, которые не должны быть (должны быть) дочерними вершинами указанного предка для соответствия шаблону.

Пример 19.

```
node DefaultCaseStatement {
  ssconstraint  {
    "Оператор case может содержать только одну ветку по умолчанию:
DefaultCaseStatement"
    forbidden
    ancestor SwitchStatement
    relevant DefaultCaseStatement
    horizontal
  } }
```

Для вертикальных шаблонов список *irrelevant* (*relevant*) вершин означает следующее: это список типов вершин, которые не должны находиться между *ancestor* и проверяемой вершиной (которые должны находиться между *ancestor* и проверяемой вершиной) для того, чтобы признать, что синтаксическая конструкция соответствует шаблону.

Для вертикальных шаблонов ключевое слово *depth* обозначает наибольшую разрешенную для данного шаблона длину пути от предка до рассматриваемой вершины.

Пример 20.

Рассмотрим примеры описания семантических ограничений на синтаксис при помощи вертикальных шаблонов.

```
node CaseLabeledStatement {
  ssconstraint  {
    "Оператор case может использоваться только в составном
операторе switch"
    mandatory
    ancestor SwitchStatement
    vertical
  } }
node ContinueStatement {
  ssconstraint  {
    "Оператор continue может находиться в составном операторе цикла
do…while"
    mandatory
    ancestor DoStatement
    vertical
  }
  ssconstraint  {
    "Оператор continue может находиться в составном операторе цикла
while"
    mandatory
    ancestor WhileStatement
    vertical
  }
  ssconstraint  {
    "Оператор continue может находиться в составном операторе цикла
for"
    mandatory
    ancestor ForStatement
    vertical
  }
}
```

Выполнение семантических ограничений на синтаксис проверяется на этапе формирования синтаксического дерева будущего теста. Синтаксический контекст из ограничения рассматривается как некоторый шаблон, который ищется в окружении вершины определенного типа, указанного в описании ограничения. Если найдена синтаксическая конструкция, соответствующая шаблону, то в зависимости от типа ограничения конструкция разрешается или запрещается.

Заключение

Данная диссертационная работа посвящена решению проблемы генерации входных данных для тестирования семантических анализаторов трансляторов. Для решения указанной проблемы в данной работе предлагается новый способ формального описания статической семантики, который дает возможность организовать целенаправленную генерацию тестовых текстов для семантических анализаторов. Под словами целенаправленная генерация в данном случае подразумевается автоматическое создание текстов без массовой генерации синтаксически корректных текстов с последующей фильтрацией семантически корректных текстов.

С целью практического применения полученных результатов был разработан язык SRL (Semantic Relation Language) для описания правил статической семантики формальных языков и инструмент STG (Semantic Test Generator) для автоматической генерации тестов для семантических анализаторов.

В ходе работы над диссертацией язык SRL и инструмент STG претерпели значительные изменения по сравнению со своим первоначальным состоянием. Так при поведении первых экспериментов почти полностью была описана статическая семантика языка C и были получены тестовые наборы для анализатора контекстных условий компилятора с языка C. Тогда языка SRL как такового не было и описания правил статической семантики представляло собой набор вызовов специальных методов на языке Java. При этом спецификации получались значительно большего размера, нежели сейчас. Конкретные значения приводятся в сравнительной таблице (см. Таб. 20).

Затем был получен тестовый набор для анализатора XML-текстов, соответствующих подмножеству описаний, заданных в стандарте MPEG 21 [43] (см. Таб. 20).

В настоящее время инструмент STG используется в проекте, ведущем разработку транслятора спецификационного расширения языка Java. Описание семантики языка Java значительно сложнее упомянутых выше проектов. Уже разработана часть тестового набора для анализатора контекстных условий языка Java, содержащая тесты, охватывающие декларации пакетов, классов, полей, методов, параметризованных классов, выражения присваивания и др. Сгенерированные тесты позволили обнаружить несколько ошибок в трансляторе. Количественные характеристики спецификаций и тестов, разработанных при помощи STG приведены в Таб. 20.

Таб. 20 Сравнительная таблица результатов применения метода

	C	XML	Java
Процент покрытых страниц стандарта	~ 70 %	-	~ 69 %
Объем SRL-спецификации (строк)	10196	147	3319
Кол-во тестов	~ 10000	~ 52	~ 11000
Общий объем	~ 10 Mb	~ 161 Kb	~ 11 Mb

В следующей таблице приводятся количественные оценки зависимости объемов сгенерированного тестового набора от параметров, задаваемых на вход генератору STG. Данные приведены по результатам испытаний, проведенных для описания языка Java.

Таб. 21 Сравнительная таблица результатов применения метода

<table>
<tr><th colspan="5">Параметры генератора STG</th><th rowspan="2">Объем сгенерированного тестового набора (тестов)</th></tr>
<tr><td>Кол-во продукционных правил</td><td>Кол-во семантических правил</td><td>Глубина рекурсии</td><td>Глубина итерирования</td><td>Кол-во тестов для одного правила</td></tr>
<tr><td rowspan="3">440</td><td rowspan="3">276</td><td>1</td><td>2</td><td>1</td><td>276</td></tr>
<tr><td>1</td><td>2</td><td>все</td><td>~ 20000</td></tr>
<tr><td>2</td><td>2</td><td>все</td><td>~ 1000000</td></tr>
</table>

В заключение перечислим основные результаты, полученные в диссертационной работе:

1. Предложен способ использования логики предикатов для формализации статической семантики языков программирования и других формальных нотаций для генерации текстов на входном языке.
2. Для проверки корректности реализации статической семантики входного языка в трансляторе предложен метод целенаправленной генерации входных данных, в котором в отличие от известных методов удается отказаться от фильтрации (селекции) семантически корректных тестов среди множества всех синтаксически корректных, и который позволяет строить тесты, нацеленные на проверку корректности анализа семантических правил в семантическом анализаторе транслятора.
3. Реализован генератор входных данных для тестов, основанный на разработанном методе.
4. Проведена апробация метода и генератора тестов в ряде исследовательских и промышленных проектов при тестировании компиляторов, трансляторов формальных

спецификаций и XML-парсеров протоколов. В рамках апробации метода были разработаны формальная спецификация статической семантики языка C и подмножества языка Java, включающего в себя описания деклараций полей, методов, параметризованных классов, арифметические выражения и др.

Литература

1. Knuth D.E. Semantics of Context-Free Languages // Mathematical Systems Theory. 1968. 2. N 2. P. 127-146.
2. Knuth D.E. Semantics of Context-Free Languages: Correction // Mathematical Systems Theory. 1971. 5. N 1. P. 95-96.
3. Описание Yacc и Lex [HTML](http://dinosaur.compilertools.net/).
4. Kastens U. Attribute Grammars as a Specification Method // Proceedings of the International Summer School on Attribute Grammars. Lecture Notes in Computer Science. Vol. 545. NewYork-Heidelberg-Berlin: Springer-Verlag, 1991. P. 16-47.
5. Kurt M. Bischoff. Ox: An attribute-grammar compiling system based on yacc, lex and c: User reference manual [HTML] (http://citeseer.ist.psu.edu/bischoff93ox.html).
6. F. Pagan. Formal Specification of Programming Languages: A Panoramic Primer. Prentice Hall. 1981. 256 p.
7. Kenneth Slonneger, Barry L. Kurtz. Formal, Syntax and Semantics of Programming Languages: A Laboratory Based Approach, 1st edition. London:Addison-Wesley Longman Publishing Co. Inc. 1994. 637 p.
8. A. S. Kossatchev, M. A. Posypkin. Survey of compiler testing methods // Programming and Computing Software. 2005. 31. N 1. P. 10 –19.

9. A. Boujarwah, K. Saleh. Compiler test case generation methods: a survey and assessment // Journal of Information and Software Technology. 1997. 39. N 9. P. 617-625.

10. Hanford K.V. Automatic generation of test cases // IBM System Journal. 1970. 9. N 4. P. 242-257.

11. Purdom P. A sentence generator for testing parsers // Behavior and Information Technology. 1972.12. N 3. P. 366-375.

12. Wichmann B.A., Jones B. Testing ALGOL 60 compilers // Software – Practice and experience. 1976. 6. N 2. P. 261-270.

13. Celentano A., Crespi Reghezzi S., Della Vigna P., Ghezzi C., Granata G., Savoretti F. Compiler Testing using a Sentence Generator // Software – Practice and Experience. 1980. 10. N 11. P. 897-918.

14. Duncan A.G., Hutchison J.S. Using Attributed Grammars to Test Designs and Implementation // In Proceedings of the 5th international conference on Software engineering. Piscataway, NJ, USA: IEEE Press, 1981. P. 170-178.

15. Franco Bazzichi and Ippolito Spadafora. An Automatic Generator for Compiler Testing // IEEE Transactions on Software Engineering. 1982. 8. N 4. P. 343-353.

16. J. Harm. Automatic test program generation from formal language specifications // Rostocker Informatik-Berishte. 1997. 20. P. 33-56.

17. Emin G.un Sirer, Brian N. Bershad. Using production grammars in software testing // In Proc. 2nd conference on Domain-specific languages. New York, NY, USA: ACM Press, 1999. 1–13.

18. Kalinov A., Kossatchev A., Petrenko A., Posypkin M., Shishkov V. Using ASM Specifications for automatic test suite generation for mpC parallel programming language compiler // In Proceedings of the Fourth International Workshop on Action Semantics, AS 2002. University of Aarhus, Department of Computer Science, Denmark: BRICS Notes Series, 2002. NS-02-8**.** P. 96-106.

19. Kalinov A., Kossatchev A., Petrenko A., Posypkin M., Shishkov V. Using ASM Specifications for Compiler Testing // In Abstract State Machines 2003 – Advances in Theory and Applications10th International Workshop, ASM 2003. Lecture Notes in Computer Science. Vol. 2589. New York-Heidelberg-Berlin: Springer-Verlag, 2003. P. 415.

20. Kossatchev A.S., Kutter P., Posypkin M.A. Automated Generation of Strictly Conforming Tests Based on Formal Specification of Dynamic Semantics of the Programming Language // Programming and Computing Software. 2004. 30. N 4. P. 218-229.

21. van Wijngaarden A., Mailloux B.J., Peck J.E., Koster C.H.A. Report on the algorithmic language Algol 68. MR 101, Mathematisch Centrum, Amsterdam, 1969.

22. Hoare C.A.R., Wirth N. An axiomatic definition of the programming language PASCAL // Acta Informatica. 1973. 2. N 4. P. 335-355.

23. Lee JAN Computer Semantics. Van Nostrand Co., New York, 1972.

24. Lucas P., Lauer P., Stigleitner H. Method and notation for the formal definition of programming languages. IBM Technical Report 25.087. Vienna :IBM Lab., 1968.

25. Lucas P., Walk K. On the formal description of PL/1 // Annual Review in Automatic Programming. 1969. 6. N 3. P. 105-182.

26. Wegner P. The Vienna Definition Language // Computer Surveys. 1972. 4. N 1. P. 5-63.

27. Демаков А.В., Зеленова С.А., Зеленов С.В. Тестирование парсеров текстов на формальных языках // Программные системы и инструменты: Тематический сборник факультета ВМиК МГУ. Вып. 2. М: ВМиК МГУ, 2001. С. 150--156.

28. А.Ахо, Р.Сети, Д.Ульман. Компиляторы: принципы, технологии, инструменты. Москва–Санкт-Петербург–Киев: Вильямс, 2001. 767 с.

29. А.В.Демаков. TreeDL. [HTML](http://sourceforge.net/projects/treedl).

30. RedVerst. [HTML](http://www.ispras.ru/~RedVerst/).

31. Information Processing – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1) // International Organization for Standardization and International Electrotechnical Committee, 1987. International Standard 8824.

32. Information Technology – Abstract Syntax Notation One (ASN.1): Specification of Basic Notation // International Telecommuncation Union, 1995. ITU-T Recommendation X.680.

33. Wang D. C., Appel A. W., Korn J. L., Serra C. S. The Zephyr Abstract Syntax Description Language // In Proceedings of the USENIX Conference on Domain-Specific Languages. Santa Barbara, California, USA: USENIX Association. 1997. P. 213–228.

34. Кристофидес Н. Теория графов. Алгоритмический подход. М.: Мир, 1978. 215 с.

35. Yuri Gurevich. Abstract State Machines: An Overview of the Project, "Foundations of Information and Knowledge Systems", Springer Lecture Notes in Computer Science, volume 2942 (2004), pages 6-13.

36. James Gosling, Bill Joy, Guy Steele, Gilad Bracha. The Java Language Specification, Third Edition, ISBN 0-321-24678-0.

37. Paulc Jorgensen. Software Testing: A Craftsman's Approach. CRC Press, 2nd edition (June 26, 2002).

38. Boris Beizer. Black-Box Testing : Techniques for Functional Testing of Software and Systems. John Wiley & Sons, Inc., 1995.

39. Brian Marick. Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing. Prentice Hall PTR (November 28, 1994).

40. Cem Kaner. Lessons Learned in Software Testing. Wiley; 1st edition (December 15, 2001)
41. Model-Based Testing of Reactive Systems : Advanced Lectures (Lecture Notes in Computer Science). Springer; 1 edition (August, 2005)
42. М.А. Посыпкин. Применение формальных методов для тестирования компиляторов. Диссертационная работа на сосискание степени кандидата физико-математических наук. ИСП РАН. Москва, 2004.
43. MPEG-21. http://www.iso.ch/iso/en/prods-services/popstds/mpeg.html
44. Robert M. Poston. Automating Specification-Based Software Testing : Institute of Electrical & Electronics Enginee (June, 1996)
45. Напрасникова М.В. Автоматическая генерация семантических тестов для компиляторов, Первая Всероссийская научная конференция “Методы и средства обработки информации”, МГУ, октябрь 2003 г.
46. Архипова М.В. Генерация тестов для модулей проверки статической семантики в компиляторах, сборник трудов ИСП, Том 8, часть 1, 2004г., стр. 59-76
47. Архипова М.В. Конструктивное описание правил статической семантики языков программирования, Вторая Всероссийская научная конференция “Методы и средства обработки информации”, МГУ, октябрь 2005 г. , стр. 323-329

48. Роберт Калбертсон, Крис Браун, Гэри Кобб. Быстрое тестирование, М., Издательский дом Вильямс, 2002.

49. Компилятор полного стандарта языка С++ как ядро систем разработки программного обеспечения. Под ред. А.Г. Сергеева Приложение к журналу “КомпьюЛог” № 3, 2000 г.

50. ISO/IEC 9126-1:2001. Software engineering — Software product quality — Part 1: Quality model.

51. ISO/IEC TR 9126-2:2003 Software engineering — Product quality — Part 2: External metrics.

52. ISO/IEC TR 9126-3:2003 Software engineering — Product quality — Part 3: Internal metrics.

53. ISO/IEC TR 9126-4:2004 Software engineering — Product quality — Part 4: Quality in use metrics.

54. STG Reference. http://unitesk.com/papers/sematesk/STG-reference-draft-1.0.pdf

55. J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, pp. 193-200, September 1994.

56. Bernhard K. Aichernig. Contract-based mutation testing in the refinement calculus. In REFINE'02, the British Computer Society - Formal Aspects of Computing refinement workshop, Copenhagen, Denmark, July 20-21, 2002, affiliated with FME 2002, volume 70 No. 3 of Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

57. Bernhard K. Aichernig and Percy Antonio Pari Salas. Test case generation by OCL mutation and constraint solving. In Kai-Yuan Cai and Atsushi Ohnishi, editors, QSIC 2OO5, Fifth International Conference on Quality Software, Melbourne, Australia, September 19-21, 2005. IEEE Computer Society Press, 2005.

58. Борисова М.В., Морозова Т.А., Петренко А.К. Чацкина Т.А. Тестирование компиляторов на основе формальной модели языка // Препринт ИПМ, N 45, 1992, 15 стр.

59. Свами М., Тхуласираман К. Графы, сети и алгоритмы//М.: «Мир», 1984, 335-338.

Приложение А. Грамматика SRL (Semantic Relation Language)

```
srl_file ::= ("header" "{" code "}")? (relation | type_set | sscl )+;
// code - внешние библиотеки (откл)
// relation - описание семантического правила
// type_set - описание семейства типов

code ::= <TEXT>;

relation ::= (s_type)? "relation" <name:ID> (":" base_rule ("," base_rule)*)?
"{"
    "comment"     (comment)?
      ("arbitrary"|"semantic")
      ("unequal" |"equal" |"present" |"absent" |"compatible" |"custom")
      ("recursive")?
      "target" node_desc
      ("source" node_desc)?
      ("context" s_context)*
      ("define" code
       "synchronize" code )?
"}"
// name:ID - уникальный идентификатор семантического правила
// base_rule - описание базового правила (множественное наследование откл.)

s_type ::= "one_to_one"|"one_to_many"|"many_to_many"|"one_node";

base_rule ::= <rel_id:ID> ("[" <context_id:ID> ("," <context_id:ID>)* "]")?
// resl_id:ID - идентификатор ранее описанного базового правила
// context_id:ID - идентификатор контекста базового правила

comment ::= <STRING_LITERAL>;

node_desc::= (path | ( <nodeId:ID> ("[" path_elem_filter "]")? ("."
path_elem)*))
                     "{"
                          (attr_list | attr_desc)("," (attr_list |
attr_desc))*
                     "}";
//nodeId:ID - идентификатор типа узла, описанного в tdl-файле

path ::=(
           (
                ( (start_rule ".")+
               |(mid_rule "."(super_r ".")*)
              ) rule_obj
             )
             | <nodeId:ID>
            )("[" path_elem_filter "]")?
            ("." path_elem_list)?;

path_elem_list ::= path_elem ("." path_elem)*;

path_elem_filter ::=(<child_id:ID> )?
                     (
                      ("==" (value | "null"))
                          | ("!=" value)
                          | ("is" ( (<nodeId:ID>("." <constId:ID>)?)
                                    | ("typeset" <typeset_id:ID>)
                                 )
```

```
                            )
                          | ("notis" ( (<nodeId:ID>("." <constId:ID>)?)
                                     | ("typeset" <typeset_id:ID>)
                                   )
                            )
                          | (( "in"
                              | "notin" ) set)
                       );
// nodeId:ID - идентификатор типа узла, описанного в tdl-файле
// constId:ID - идентификатор enum-константы, заданный в tdl-файле в описании
узла типа nodeId:ID
// typeset_id:ID - идентификатор семейства типов, описанный в srl-файле
// child_id:ID - идентификатор ребенка узла, описанного в tdl-файле

start_rule ::= "super" | "this" ;

mid_rule ::= <rel_id:ID> "[" (rule_filter | "*") "]" ;
// rel_id:ID - идентификатор семантического правила, описанный в srl-файле

rule_filter ::= rule_obj "== prev";
// prev - ссылка на результат поиска по пути описанному до ближайшей точки
слева от prev

super_r ::= "super";
// super - ссылка на экземпляр базового правила на основе, которого строится
данное правило, определяется во время исполнения

rule_obj ::= "source" | "target" | "context" | "source_context" |
"target_context";

path_elem ::= (mid_rule "." (super_r ".")* rule_obj) | elem | ances_elem |
recur_set | desc_elem;

value ::= str_value | path;
// str_value - явное писание объекта
// path - ссылочное описание объекта

str_value ::= <STRING_LITERAL>;
//<STRING_LITERAL> - описание создания какого-то объекта, например, new
Integer(1)

set ::= "{" str_value ("," str_value)* "}" | "typeset" <typeset_id:ID> | path
;
// typeset_id:ID - идентификатор семейства типов, описанный в srl-файле

elem ::=  <child_id:ID> ("[i]")? ("[" path_elem_filter "]")?;
// typeset_id:ID - идентификатор семейства типов, описанный в srl-файле
// [i] - временно не поддерживается

ances_elem ::= ances ("[" path_elem_filter "]")?;

ances ::= "parent" | ("^" <node_id:ID>);
//nodeId:ID - идентификатор типа узла, описанного в tdl-файле

recur_set ::= "(" recur_elem ("." recur_elem)* ")*";

recur_elem ::= (mid_rule "."(super_r ".")* rule_obj) | elem | ances_elem;

desc_elem ::= (( ("<" | "<=" | ">" | ">=") <number>) | "00") "." <nodeId:ID>
;

attr_list ::= "{" attr_desc ("," attr_desc)* "}";

attr_desc ::= path_elem ("." path_elem)* (attr_value_set )?;
```

```
attr_value_set ::= "{"(attr_value_group | (("<" <java_class_id:ID> ">")?
("not")? attr_value)) ("," (attr_value_group | (("<"type">")? ("not")?
attr_value)))*"}";
//java_class_id:ID - идентификатор java-класса, который должен быть типом
attr_value

attr_value_group ::= ("<" <java_class_id:ID> ">")? ("not")? "{"attr_value
("," attr_value)* "}";
//java_class_id:ID - идентификатор java-класса, который должен быть типом
attr_value

attr_value ::= ( str_value | "*" | recur_set);
// * - любое значение указанного типа

s_context ::= <name:ID> ":" (sngl_context | dbl_context);
// name:ID - уникальное в пределах одного relation имя контекста

sngl_context ::= "same" <node_id:ID> ("["path_elem_filter"]")? ("."
path_elem )*;
//nodeId:ID - идентификатор типа узла, описанного в tdl-файле, который будет
типом корня описываемого контекста

dbl_context ::= "differ" <node_id:ID>
               (
                "{"
                  "source_context" "[" path_elem_filter "]" ("." path_elem )*
                     "," "target_context" "[" path_elem_filter "]" ("."
path_elem )*
                "}"
               )?;
//nodeId:ID - идентификатор типа узла, описанного в tdl-файле, который будет
типом корня описываемого контекста

type_set ::= "type_set" <name:ID> "{" (type_desc | type_group) (","
(type_desc | type_group) )* "}";
//name:ID - уникальное имя семейства типов

type_desc ::= ("ref" <nodeId:ID> ("["path_elem_filter"]")? ("." path_elem)+ )
| ("const" <nodeId:ID> "." <constId:ID>);
//nodeId:ID - идентификатор типа узла, описанного в tdl-файле;
//если после const, то в узле указанного типа (nodeId:ID) должна быть
описанна константа с именем constId:ID

type_group ::= "{" type_desc ("," type_desc)* "}";
sscl ::= (node_ssconstraint)*;

node_ssconstraint ::= "node" node_type "{"(ssconstraint)+ "}";

ssconstraint ::= "ssconstraint"
                      "{"
                      (comment)?
                      "forbidden" | "mandatory"
                      "ancestor" node_type
                      ("irrelevant" "("node_list")")?
                      ("relevant"   "("node_list")")?
                      "horizontal"
                      |( "vertical"
                      ("descendent" node_type)?
                      ("depth" depth)?)
                      "}";

node_type ::= <ID>;
```

```
depth ::= <NUMBER>;

comment ::= "comment" <TEXT>;

node_list ::= (node_type("," node_type)*)?;
```

Приложение B. Атрибутная грамматика языка L_{oo}

```
program ::= “program” identifier “;” [classes]
        program.classestab := empty_classestab()
        program.public_methodstab := empty_methodstab()
        program.private_methodstab := empty_methodstab()
classes ::= class | classes
class ::= “class” new_identifier [“extends” super_identifier]“{”
[methods] “}”
        Cond: contains(class.new_identifier, program.classestab)=FALSE
        Cond: contains(class.super_identifier, program.classestab)=TRUE
        Cond: cycle_exists(class.new_identifier,
class.super_identifier)=FALSE
        program.classestab := push(class.new_identifier,
program.classestab)
        program.public_methodstab := pushAll(class.new_identifier,
getmethods(class.super_identifier, program.public_methodstab),
program.public_methodstab)
        methods.class_id := class.new_identifier
new_identifier ::= identifier
super_identifier ::= identifier
methods ::= method | methods
        method.class_id := methods.class_id
method ::= public_method | private_method
        public_method.class_id := method.class_id
        private_method.class_id := method.class_id
public_method ::= “public” identifier body
        Cond: contains(identifier, getmethods(public_method.class_id,
program.public_methodstab))=FALSE
        program.public_methodstab := push(public_method.class_id,
public_method.identifier, program.public_methodstab)
        body.class_id := public_method.class_id
private_method ::= “private” identifier body
        Cond: contains(identifier, getmethods(private_method.class_id,
program.private_methodstab))=FALSE
        program.private_methodstab := push(private_method.class_id,
private_method.identifier, program.private_methodstab)
```

```
        body.class_id := private_method.class_id
body ::= "{" [stmts] "}"
        stmts.class_id := body.class_id
stmts ::= stmt | stmts
        stmt.class_id := stmts.class_id
stmt ::= (call_expr | print_expr) ";"
        call_expr.class_id := stmt.class_id
        print_expr.class_id := stmt.class_id
print_expr ::= "print" message
call_expr ::= class_identifier "." method_identifier
        Cond: contains(call_expr.class_identifier, program.classestab)=TRUE
        Cond: (contains(method_identifier,
getmethods(call_expr.class_identifier,
        program.private_methodstab))=TRUE &&
call_expr.class=call_expr.class_identifier) || contains(method_identifier,
getmethods(call_expr.class_identifier, program.public_methodstab))=TRUE
class_identifier ::= identifier
method_identifier ::= identifier
message ::= lq identifier rq
```

Таб. 22. Неформальное описание контекстных условий языка L_{oo}

№	Описание
1.	Инициализируется атрибут classestab нетерминала program, описывающий таблицу имен классов.
2.	Инициализируется атрибут public_methodstab нетерминала program, описывающий таблицу имен public методов классов.
3.	Инициализируется атрибут private_methodstab нетерминала program, описывающий таблицу имен private методов классов.
4.	Проверка уникальности имени класса.
5.	Проверка наличия описания базового класса.
6.	Проверка отсутствия цикла в графе наследования.
7.	Добавление имени нового класса (class.new_identifier) в таблицу имен классов (program.classestab).

№	**Описание**
8.	Наследование в новом классе (class.new_identifier) всех public методов базового класса (class.super_identifier).
9.	Инициализация атрибута class_id нетерминала methods, значением которого является идентификатор родительского класса методов.
10.	Инициализация атрибута class_id нетерминала method, значением которого является идентификатор родительского класса метода.
11.	Инициализация атрибута class_id нетерминала public_method, значением которого является идентификатор родительского класса public метода.
12.	Инициализация атрибута class_id нетерминала private_method, значением которого является идентификатор родительского класса private метода.
13.	Проверка уникальности имени public метода в контексте родительского класса этого метода.
14.	Добавление имени нового public метода в таблицу имен public методов родительского класса.
15.	Инициализация атрибута class_id нетерминала body, значением которого является идентификатор родительского класса.
16.	Проверка уникальности имени private метода в контексте родительского класса этого метода.
17.	Добавление имени нового private метода в таблицу имен public методов родительского класса.
18.	Инициализация атрибута class_id нетерминала body, значением которого является идентификатор родительского класса.
19.	Инициализация атрибута class_id нетерминала stmts, значением которого является идентификатор родительского класса.
20.	Инициализация атрибута class_id нетерминала stmt, значением которого является идентификатор родительского класса.
21.	Инициализация атрибута class_id нетерминала call_expr, значением которого является идентификатор родительского класса.
22.	Инициализация атрибута class_id нетерминала print_expr, значением которого является идентификатор родительского класса.
23.	Проверка существования описания класса с именем call_expr.class_identifier.

№	**Описание**
24.	Проверка существования описания private метода с именем call_expr.method_identifier в классе с именем call_expr.class_identifier в случае, если вызов метода производится в теле класса call_expr.class_identifier, или проверка существования описания public метода с именем call_expr.method_identifier в классе с именем call_expr.class_identifier во всех остальных случаях.

Приложение C. Свод контекстных условий L_{oo} на языке SRL

```
many-to-many relation R1
 {
 "В программе имя класса должно быть уникально."
 unordered
 unequal
 target Class { name }
 source Class { name }
 context c1 : same Program
}
one-to-many relation R2 : R4
{
 "Класс не должен быть базовым для самого себя, а также не должен
 быть базовым для своего базового класса и т.д."
 ordered
 unequal
 target super.target[is Class].(R4[target=prev].source)* { superName }
 source super.target[is Class] { name }
 context c1 : same Program
}
many-to-many relation R3
 {
 "В классе имена методов должны быть уникальны."
 unordered
 unequal
 target Method { name }
 source Method { name }
 context c1 : same Class
}
```

```
one-to-many relation R4
 {
 " Имя класса должно быть описано до использования. "
 ordered
 equal
 target Class { superName } | CallStmt { className }
 source Class { name }
 context c1 : same Program
}
one-to-many relation R5_R6_R7 : R4
{
 " В конструкции, задающей вызов метода, после точки должно стоять
 имя метода описанного, в том классе имя, которого находится перед
 точкой. "
 ordered
 equal
 target CallStmt { methodName }
 source this.target.R4[target = prev].source[= this.target.^Class]
          .1:Method { name }
       | this.target.R4[target = prev].source[!= this.target.^Class]
          .1:Method[modifier=Method.PUBLIC] { name }
 context c1 : same Program
}
```

Приложение D. Свод контекстных условий для подмножества языка Java 5.0 на языке SRL

```
 // Line 5
one_to_many relation R1_1_0
{
  comment "Имя пакета верхнего уровня должно быть задано"
  arbitrary
  equal
  target FullTypeName.1.SimplePackageName  {ident}
  source this.target_context.1.PackageNode[subpackages = null].1.PackageDeclaration {name}
  context c1 : same RootNode

}

// Line 6
one_to_many relation R1_2_0
{
  comment "Имя пакета верхнего уровня должно быть задано"
  arbitrary
  equal
  target FullTypeName.1.CompoundPackageName  {ident}
  source this.target_context.1.PackageNode[subpackages != null].1.PackageDeclaration {name}
  context c1 : same RootNode

}

// Line 7
one_to_one relation R1_3_0: R1_2_0
{
  comment "Подпакет должен быть описан в соответствующем пакете"
  arbitrary
  equal
  target super.target.1.PackageName.(1.PackageName)*  {ident}
  source
super.source.parent.1.PackageNode.1.PackageDeclaration.(parent.1.PackageNode.1.PackageDec
laration)* {name}
  context c1 : same RootNode

}

// Line 8
one_to_many relation R1_4_General_0: R1_1_0
{
  comment "Тип верхнего уровня должен быть описан в соответствующем пакете (обычное
использование)."
  arbitrary
  equal
```

```
  target super.target[is
SimplePackageName].^FullTypeName[^Interfaces=null].1.SimpleTypeName.1.NonParametrize
dTypeDeclSpecifier.00.TopLevelTypeName  {ident}
  source super.source.parent.1.CompilationUnit.1.TypeDeclaration {ident}
  context c1 : same RootNode

}

// Line 8
one_to_many relation R1_4_General_1: R1_3_0
{
  comment "Тип верхнего уровня должен быть описан в соответствующем пакете (обычное
использование)."
  arbitrary
  equal
  target super.target[is
SimplePackageName].^FullTypeName[^Interfaces=null].1.SimpleTypeName.1.NonParametrize
dTypeDeclSpecifier.00.TopLevelTypeName  {ident}
  source super.source.parent.1.CompilationUnit.1.TypeDeclaration {ident}
  context c1 : same RootNode

}

// Line 9
one_to_many relation R1_4_InInterfaceList_0: R1_1_0
{
  comment "Тип верхнего уровня должен быть описан в соответствующем пакете
(использование в списке интерфейсов)."
  arbitrary
  equal
  target super.target[is
SimplePackageName].^FullTypeName[^Interfaces!=null].1.SimpleTypeName.1.NonParametrize
dTypeDeclSpecifier.00.TopLevelTypeName  {ident}
  source super.source.parent.1.CompilationUnit.1.InterfaceDeclaration {ident}
  context c1 : same RootNode

}

// Line 9
one_to_many relation R1_4_InInterfaceList_1: R1_3_0
{
  comment "Тип верхнего уровня должен быть описан в соответствующем пакете
(использование в списке интерфейсов)."
  arbitrary
  equal
  target super.target[is
SimplePackageName].^FullTypeName[^Interfaces!=null].1.SimpleTypeName.1.NonParametrize
dTypeDeclSpecifier.00.TopLevelTypeName  {ident}
  source super.source.parent.1.CompilationUnit.1.InterfaceDeclaration {ident}
  context c1 : same RootNode

}
```

```
// Line 10
one_to_many relation R1_4_InSuperClass_0: R1_1_0
{
  comment "Тип верхнего уровня должен быть описан в соответствующем пакете
(использование в extends)."
  arbitrary
  equal
  target super.target[is
SimplePackageName].^FullTypeName[^Super!=null].1.SimpleTypeName.1.NonParametrizedTy
peDeclSpecifier.00.TopLevelTypeName  {ident}
  source super.source.parent.1.CompilationUnit.1.ClassDeclaration {ident}
  context c1 : same RootNode

}

// Line 10
one_to_many relation R1_4_InSuperClass_1: R1_3_0
{
  comment "Тип верхнего уровня должен быть описан в соответствующем пакете
(использование в extends)."
  arbitrary
  equal
  target super.target[is
SimplePackageName].^FullTypeName[^Super!=null].1.SimpleTypeName.1.NonParametrizedTy
peDeclSpecifier.00.TopLevelTypeName  {ident}
  source super.source.parent.1.CompilationUnit.1.ClassDeclaration {ident}
  context c1 : same RootNode

}

// Line 11
one_to_one relation R2_1_0: R1_4_General_0
{
  comment "Внутренний тип должен быть описан в соответствующем объемлющем типе"
  arbitrary
  equal
  target super.target.^NestedTypeName.(^NestedTypeName)*  {ident}
  source super.source.2.TypeDeclaration.(2.TypeDeclaration)* {ident}
  context c1 : same RootNode

}

// Line 11
one_to_one relation R2_1_1: R1_4_General_1
{
  comment "Внутренний тип должен быть описан в соответствующем объемлющем типе"
  arbitrary
  equal
  target super.target.^NestedTypeName.(^NestedTypeName)*  {ident}
  source super.source.2.TypeDeclaration.(2.TypeDeclaration)* {ident}
  context c1 : same RootNode
```

```
}

// Line 11
one_to_one relation R2_1_2: R1_4_InInterfaceList_0
{
  comment "Внутренний тип должен быть описан в соответствующем объемлющем типе"
  arbitrary
  equal
  target super.target.^NestedTypeName.(^NestedTypeName)*  {ident}
  source super.source.2.TypeDeclaration.(2.TypeDeclaration)* {ident}
  context c1 : same RootNode

}

// Line 11
one_to_one relation R2_1_3: R1_4_InInterfaceList_1
{
  comment "Внутренний тип должен быть описан в соответствующем объемлющем типе"
  arbitrary
  equal
  target super.target.^NestedTypeName.(^NestedTypeName)*  {ident}
  source super.source.2.TypeDeclaration.(2.TypeDeclaration)* {ident}
  context c1 : same RootNode

}

// Line 11
one_to_one relation R2_1_4: R1_4_InSuperClass_0
{
  comment "Внутренний тип должен быть описан в соответствующем объемлющем типе"
  arbitrary
  equal
  target super.target.^NestedTypeName.(^NestedTypeName)*  {ident}
  source super.source.2.TypeDeclaration.(2.TypeDeclaration)* {ident}
  context c1 : same RootNode

}

// Line 11
one_to_one relation R2_1_5: R1_4_InSuperClass_1
{
  comment "Внутренний тип должен быть описан в соответствующем объемлющем типе"
  arbitrary
  equal
  target super.target.^NestedTypeName.(^NestedTypeName)*  {ident}
  source super.source.2.TypeDeclaration.(2.TypeDeclaration)* {ident}
  context c1 : same RootNode

}

// Line 50
```

```
one_to_many relation R15_1_0
{
 comment "Имя конструктора должно совпадать с именем класса, в котором он описан."
 arbitrary
 equal
 target ConstructorDeclarator {ident}
 source this.target.^ClassDeclaration {ident}
 context c1 : same ClassDeclaration

}

// Line 51

one_to_many relation R15_2_0
{
 comment "Имя конструктора должно совпадать с именем класса, в котором он описан."
 arbitrary
 equal
 target ConstructorDeclarator {ident}
 source this.target.^NewClassInstanceCreationExpression.<=3.TypeDeclSpecifier.1.TypeName
{пусто}
 context c1 : same NewClassInstanceCreationExpression

}
*/

// Line 52

one_to_many relation R15_3_0
{
 comment "Имя конструктора должно совпадать с именем класса, в котором он описан."
 arbitrary
 equal
 target ConstructorDeclarator {ident}
 source this.target.^PrimaryClassInstanceCreationExpression {пусто}
 context c1 : same PrimaryClassInstanceCreationExpression

}
*/

// Line 73
many_to_many relation R28_1_0
{
 comment "Имена подпакетов в одном пакете должны отличаться"
 arbitrary
 unequal
 target PackageNode {optPackageDeclaration.name}
 source PackageNode {optPackageDeclaration.name}
 context c1 : same PackageNode

}
```

```
// Line 74
many_to_many relation R28_2_0
{
  comment "Имена пакетов верхнего уровня должны отличаться."
  arbitrary
  unequal
  target RootNode.1.PackageNode {optPackageDeclaration.name}
  source RootNode.1.PackageNode {optPackageDeclaration.name}
  context c1 : same RootNode

}

// Line 75
many_to_many relation R28_3_0
{
  comment "Имена типов верхнего уровня в одном пакете должны отличаться"
  arbitrary
  unequal
  target PackageNode.2.TypeDeclaration {ident}
  source PackageNode.2.TypeDeclaration {ident}
  context c1 : same PackageNode

}

// Line 83
many_to_many relation R29_1_0
{
  comment "В одном классе или интерфейсе нельзя объявлять поля с одинаковыми
простыми именами."
  arbitrary
  unequal
  target FieldDeclaration.2.VariableDeclarator.00.SimpleVariableDeclaratorId {ident}
  source FieldDeclaration.2.VariableDeclarator.00.SimpleVariableDeclaratorId {ident}
  context c1 : same ClassBody

}

// Line 84
/*
many_to_many relation R30_1_0
{
  comment "эквивалентность сигнатур методов"
  arbitrary
  custom
  target MethodHeader {declarator.ident}
  source MethodHeader {declarator.ident}
  context c1 : same ClassBody
  context c2 : same InterfaceBody
  define {

if(((MethodDeclaration)semRel.getTarget().getParent()).isSignatureEqual((MethodDeclaration)s
emRel.getSource().getParent()))
```

```
 {
    trgAttr.setSetMode(callType);
    trgAttr.setNewValue();
    res = true;
 }
  }
  synchronize {

if(((MethodDeclaration)semRel.getSource().getParent()).isSignatureEqual((MethodDeclaration)s
emRel.getTarget().getParent()))
 {
    srcAttr.setSetMode(callType);
    srcAttr.setNewValue();
    res = false;
 }
  }

}
*/

// Line 141
one_to_many relation R43_1_0
{
  comment "Простое имя типа должно находиться в области действия декларации этого
типа."
  arbitrary
  equal
  target SimpleTypeName[^FullTypeName=null].2.TopLevelTypeName {ident}
  source  ClassDeclarationBlockStatement.1.ClassDeclaration {ident}
  context c1 : same Block

}

// Line 142
one_to_many relation R43_2_0
{
  comment "Простое имя типа должно находиться в области действия декларации этого
типа."
  arbitrary
  equal
  target SimpleTypeName[^FullTypeName=null].2.TopLevelTypeName {ident}
  source CompilationUnit.1.TypeDeclaration {ident}
  context c1 : same AbstractPackageNode

}

// Line 143
one_to_many relation R43_3_0
{
  comment "Простое имя типа должно находиться в области действия декларации этого
типа."
  semantic
```

```
  equal
  target SimpleTypeName[^FullTypeName=null].2.TopLevelTypeName {ident}
  source SingleTypeImportDeclaration.00.TopLevelTypeName {ident}
  context c1 : same CompilationUnit

}

// Line 144
one_to_many relation R43_4_0
{
  comment "Простое имя типа должно находиться в области действия декларации этого типа."
  arbitrary
  equal
  target SimpleTypeName[^FullTypeName=null].2.TopLevelTypeName {ident}
  source TypeDeclaration {ident}
  context c1 : same ClassBody

}

// Line 145
one_to_many relation R43_5_0
{
  comment "Простое имя типа должно находиться в области действия декларации этого типа."
  arbitrary
  equal
  target SimpleTypeName[^FullTypeName=null].2.TopLevelTypeName {ident}
  source TypeDeclaration {ident}
  context c1 : same InterfaceBody

}

// Line 146
one_to_many relation R44_ExprNameDecl_1_0
{
  comment "Идентификатор должен быть описан перед использованием"
  semantic
  equal
  target ExpressionName {ident}
  source VariableDeclarator {varDeclId.ident}
  context c1 : same Block


}

// Line 147
one_to_many relation R44_ExprNameDecl_2_0: R44_ExprNameDecl_1_0
{
  comment "Идентификатор должен быть описан перед использованием"
  semantic
  equal
```

```
  target super.target {type}
  source super.source.^VariableDeclaratorWrapper {type}
  context c1 : same Block


}

// Line 264
many_to_many relation R69_1_0
{
  comment "Имена подпакетов и top level типов в одном пакете различны"
  arbitrary
  unequal
  target PackageNode.1.PackageDeclaration {name}
  source this.target_context.2.TypeDeclaration {ident}
  context c1 : same PackageNode

}

// Line 294
many_to_many relation R82_1_0
{
  comment "Нельзя импортировать тип из unnamed пакета."
  arbitrary
  unequal
  target ImportDeclaration.2.SimpleTypeName.2.TopLevelTypeName {ident}
  source UnnamedPackageNode.2.TypeDeclaration {ident}
  context c1 : same RootNode

}

// Line 373
many_to_many relation R112_1_0
{
  comment "Нельзя импортировать public тип при помощи single-type-import в compilation
unit, в котором объявлен тип с таким же простым именем"
  arbitrary
  unequal
  target SingleTypeImportDeclaration.00.TopLevelTypeName {ident}
  source TypeDeclaration.modifiers[accessibility = "new
AccesibilityModifiers(AccesibilityModifiers.PUBLIC)"].parent {ident}
  context c1 : same CompilationUnit

}

// Line 375
/*
one_to_one relation R113_1_0
{
  comment "Можно импортировать типы из того же пакета в котором описан класс."
  arbitrary
  equal
```

```
  target
TypeImportOnDemandDeclaration.2.CompoundPackageName.00.SimplePackageName.(^Comp
oundPackageName)* {ident}
  source this.target_context.^PackageNode.(^PackageNode)* {optPackageDeclaration.name}
  context c1 : same CompilationUnit

}
*/

// Line 376
/*
one_to_many relation R113_2_0
{
  comment "Можно импортировать типы из того же пакета в котором описан класс."
  arbitrary
  equal
  target TypeImportOnDemandDeclaration.2.SimplePackageName {ident}
  source this.target_context.^PackageNode {optPackageDeclaration.name}
  context c1 : same CompilationUnit

}
*/

// Line 377
/*
one_to_many relation R113_3_0
{
  comment "Можно импортировать типы из того же пакета в котором описан класс."
  arbitrary
  equal
  target TypeImportOnDemandDeclaration.00.TopLevelTypeName {ident}
  source CompilationUnit.1.TypeDeclaration.modifiers[accessibility = "new
AccesibilityModifiers(AccesibilityModifiers.PUBLIC)"].parent {ident}
  context c1 : same CompilationUnit

}
*/

// Line 585
many_to_many relation R145_1_0
{
  comment "Среди списка AdditionalBoundList могут быть только интерфейсы."
  arbitrary
  equal
  target
ParameterizedClassDeclaration.1.TypeParameters.00.TypeParameter.1.TypeBound.optAdditiona
lBoundInterfaceList.1.SimpleTypeName.2.TypeName {ident}
  source InterfaceDeclaration {ident}
  context c1 : same RootNode

}
```

```
// Line 815
many_to_many relation R196_1_UniqueClassIdent_0
{
  comment "Имя вложенного класса не должно совпадать с именами объемлющих классов
или интерфейсов"
  arbitrary
  unequal
  target TypeDeclaration {ident}
  source this.target.2.ClassDeclaration {ident}
  context c1 : same RootNode

}

// Line 816
many_to_many relation R196_2_UniqueInterfaceIdent_0
{
  comment "Имя вложенного интерфейса не должно совпадать с именами объемлющих
классов или интерфейсов"
  arbitrary
  unequal
  target TypeDeclaration {ident}
  source this.target.2.InterfaceDeclaration {ident}
  context c1 : same RootNode

}

// Line 817
/*
one_node relation R197_1_0
{
  comment "Нельзя использовать модификатор public для local типов"
  arbitrary
  unequal
  target TypeDeclaration.1.Modifiers {accessibility {<AccesibilityModifiers> not
"AccesibilityModifiers.PUBLIC"}}
  context c1 : same ClassBody.1.MethodDeclaration.00.ClassInstanceCreationExpression

}
*/

// Line 818
one_node relation RClassMemberDecl_Protected_0
{
  comment "Модификторы доступа private и protected в декларации классов могут
использоваться только при описании классов-членов непосредственно в теле другого
класса и запрещены для описания классов верхнего уровня."
  arbitrary
  unequal
  target ClassDeclaration[^ClassDeclaration=null].1.Modifiers {accessibility
{<AccesibilityModifiers> not "AccesibilityModifiers.PROTECTED"}}
  context c1 : same RootNode
```

```
}

// Line 819
one_node relation RClassMemberDecl_Private_0
{
  comment "Модификторы доступа private и protected в декларации классов могут
использоваться только при описании классов-членов непосредственно в теле другого
класса и запрещены для описания классов верхнего уровня."
  arbitrary
  unequal
  target ClassDeclaration[^ClassDeclaration=null].1.Modifiers {accessibility
{<AccesibilityModifiers> not "AccesibilityModifiers.PRIVATE"}}
  context c1 : same RootNode

}

// Line 820
one_node relation RClassMemberDecl_Static_0
{
  comment "Модификатор static при описании классов можно использоваться только в
декларации классов-членов и запрещен для описания классов верхнего уровня."
  arbitrary
  absent
  target ClassDeclaration[^ClassDeclaration=null].1.Modifiers {isStatic}
  context c1 : same RootNode

}

// Line 823
many_to_many relation R199_AbstractClassDecl_0
{
  comment "Только абстрактные классы могут иметь абстрактные методы."
  arbitrary
  present
  target ClassDeclaration {modifiers.isAbstract}
  source this.target.2.MethodDeclaration.1.MethodHeader.1.MethodModifiers {isAbstract}
  context c1 : same RootNode

}

// Line 824
/*
many_to_many relation R200_EnumAbstract_0
{
  comment "R200_EnumAbstract"
  arbitrary
  present
  target ClassDeclaration {modifiers.isAbstract}
  context c1 : same RootNode

}
*/
```

```
// Line 836
one_node relation RClassDeclFinal_Abstract_0
{
  comment "Абстрактный класс не может быть final"
  arbitrary
  absent
  target ClassDeclaration.1.Modifiers {isFinal}
  source ClassDeclaration.1.Modifiers {isAbstract}
  context c1 : same RootNode

}

// Line 846
many_to_many relation R212_InnerClass_StaticInit_0
{
  comment "Во внутреннем классе нельзя описывать блоки статической инициализации."
  arbitrary
  present
  target ClassDeclaration[^TypeDeclaration!=null] {modifiers.isStatic}
  source this.target.2.StaticInitializer {block}
  context c1 : same RootNode

}

// Line 847
many_to_many relation R212_InnerClass_InterfDecl_0
{
  comment "Во внутреннем классе нельзя описывать интерфейсы."
  arbitrary
  present
  target ClassDeclaration[^TypeDeclaration!=null] {modifiers.isStatic}
  source this.target.2.InterfaceDeclaration {ident}
  context c1 : same RootNode

}

// Line 848
many_to_many relation R212_InnerClass_FieldDecl_0
{
  comment "Во внутреннем классе нельзя описывать любые другие статические члены,
кроме констант времени компиляции."
  arbitrary
  equal
  target ClassDeclaration[^TypeDeclaration!=null] {modifiers.isStatic}
  source this.target.2.FieldDeclaration.1.FieldModifiers {isStatic}
  context c1 : same RootNode

}

// Line 849
many_to_many relation R212_InnerClass_MethodDecl_0
```

```
{
  comment "Во внутреннем классе нельзя описывать любые другие статические члены, кроме констант времени компиляции."
  arbitrary
  equal
  target ClassDeclaration[^TypeDeclaration!=null] {modifiers.isStatic}
  source this.target.2.MethodDeclaration.1.MethodHeader.1.MethodModifiers {isStatic}
  context c1 : same RootNode

}

// Line 868
one_to_many relation R227_1_0
{
  comment "Для имени типа указанного в extends должен существовать класс с тем же именем"
  arbitrary
  equal
  target NormalClassDeclaration.superClass.00.TypeDeclSpecifier.1.TopLevelTypeName {ident}
  source ClassDeclaration {ident}
  context c1 : same RootNode

}

// Line 869
/*
one_to_many relation R227_2_0
{
  comment "Для простого имени типа указанного в extends должен существовать класс с тем же именем"
  arbitrary
  equal
  target
NormalClassDeclaration.superClass.1.SimpleTypeName.1.TypeDeclSpecifier.1.TopLevelTypeName {ident}
  source ClassDeclaration {ident}
  context c1 : same RootNode

}
*/

// Line 870
one_to_many relation R227_3_0: R227_1_0
{
  comment "Пакет полного имени типа указанного в extends должен отличаться от пакета описываемого класса"
  arbitrary
  equal
  target super.target.^FullTypeName {packName}
  source super.source.^PackageNode {optPackageDeclaration.name}
  context c1 : same RootNode
```

```
}

// Line 871
/*
one_to_many relation R227_4_0
{
  comment "Для полного имени типа указанного в extends должен существовать класс с тем
же именем"
  arbitrary
  equal
  target
NormalClassDeclaration.superClass.1.FullTypeName.1.SimpleTypeName.1.TypeDeclSpecifier.
1.TopLevelTypeName {packName}
  context c1 : same RootNode

}
*/

// Line 872
one_node relation R227_ExtendsPublic_0: R227_1_0
{
  comment "Имя класса должно быть accessible."
  arbitrary
  present
  target super.source.1.Modifiers {accessibility {<AccesibilityModifiers>
"AccesibilityModifiers.PUBLIC"}}
  context c1 : same RootNode

}

// Line 873
one_node relation R227_ExtendsFinal_0: R227_1_0
{
  comment "Имя класса в Super.ClassType не может быть именем final-класса."
  arbitrary
  absent
  target super.source.1.Modifiers {isFinal}
  context c1 : same RootNode

}

// Line 878
one_to_many relation R229_TD_0
{
  comment "Тип класса или интерфейса должен быть описан"
  arbitrary
  equal
  target TypeName {ident}
  source TypeDeclaration {ident}
  context c1 : same RootNode
```

```
}

// Line 879
one_to_many relation R229_CRL_0: R229_TD_0
{
  comment "Ориентированный граф зависимостей наследования не должен иметь циклов."
  arbitrary
  unequal
  target
this.target_context.1.Super.1.ClassOrInterfaceType.00.TypeName.(R229_TD_0[target=prev].source.1.Super.1.ClassOrInterfaceType.00.TypeName)* {ident}
  source this.target_context {ident}
  context c1 : same TypeDeclaration

}

// Line 884
one_to_many relation R233_ImplementsInterfaces_0
{
  comment "При описании класс может реализовывать только доступные интерфейсы. "
  arbitrary
  equal
  target
NormalClassDeclaration.1.Interfaces.1.ClassOrInterfaceType.00.TypeDeclSpecifier.1.TopLevelTypeName {ident}
  source InterfaceDeclaration {ident}
  context c1 : same RootNode

}

// Line 885
one_node relation R233_ImplementsPublic_0: R227_1_0
{
  comment "Имя интерфейса должно быть accessible."
  arbitrary
  present
  target super.source.1.Modifiers {accessibility {<AccesibilityModifiers>
"AccesibilityModifiers.PUBLIC"}}
  context c1 : same RootNode

}

// Line 887
many_to_many relation R233_InterfacesDifferentNames_0
{
  comment "Один и тот же интерфейс нельзя указать дважды в списке базовых
интерфейсов при описании класса. Даже, если одно имя будет простым, а второе
квалифицированным."
  arbitrary
  unequal
  target Interfaces.1.ClassOrInterfaceType.00.TypeDeclSpecifier.1.TypeName {ident}
```

```
  source this.target.^Interfaces.1.ClassOrInterfaceType.00.TypeDeclSpecifier.1.TypeName
{ident}
  context c1 : same RootNode

}

// Line 903
/*
many_to_many relation R245_1_0
{
  comment "В одном классе не должно быть полей с одинаковыми именами."
  arbitrary
  unequal
  target FieldDeclaration.2.VariableDeclarator.1.SimpleVariableDeclaratorId {ident}
  source FieldDeclaration.2.VariableDeclarator.1.SimpleVariableDeclaratorId {ident}
  context c1 : same ClassDeclaration

}
*/

// Line 944
one_node relation R261_1_0
{
  comment "Поле объявленное как final не может быть volatile"
  arbitrary
  absent
  target FieldModifiers {isFinal}
  source FieldModifiers {isVolatile}
  context c1 : same FieldDeclaration

}

// Line 946
one_to_one relation R262_FieldInitType_0
{
  comment "Тип инициализирующего выражения должен приводиться к типу переменной"
  arbitrary
  compatible
  target
VariableDeclaratorWrapper.1.CompoundVariableDeclarator.1.VariableInitializer.1.Expression
{type}
  source this.target.^VariableDeclaratorWrapper {type}
  context c1 : same FieldDeclaration

}

// Line 986
many_to_many relation R274_1_0
{
  comment "Метод не может иметь параметров с одинаковыми именами."
  arbitrary
  unequal
```

```
  target MethodDeclarator.00.SimpleVariableDeclaratorId {ident}
  source MethodDeclarator.00.SimpleVariableDeclaratorId {ident}
  context c1 : same MethodHeader

}

// Line 987
one_node relation RMethodHeader_ParamInit_0
{
  comment "Параметры методов нельзя инициализировать значениями."
  arbitrary
  present
  target MethodDeclarator.00.SimpleVariableDeclaratorWrapper {varDeclarator[is
SimpleVariableDeclarator]}
  context c1 : same MethodHeader

}

// Line 1046
one_node relation R287_1_0
{
  comment "Abstract метод не может быть private"
  arbitrary
  absent
  target MethodModifiers {isAbstract}
  source MethodModifiers {accessibility {<AccesibilityModifiers>
"AccesibilityModifiers.PRIVATE"}}
  context c1 : same MethodHeader

}

// Line 1047
one_node relation R287_2_0
{
  comment "Abstract метод не может быть static"
  arbitrary
  absent
  target MethodModifiers {isAbstract}
  source MethodModifiers {isStatic}
  context c1 : same MethodHeader

}

// Line 1048
one_node relation R287_3_0
{
  comment "Abstract метод не может быть final"
  arbitrary
  absent
  target MethodModifiers {isAbstract}
  source MethodModifiers {isFinal}
  context c1 : same MethodHeader
```

```
}

// Line 1049
one_node relation R287_4_0
{
  comment "Abstract метод не может быть native"
  arbitrary
  absent
  target MethodModifiers {isAbstract}
  source MethodModifiers {isNative}
  context c1 : same MethodHeader

}

// Line 1050
one_node relation R287_5_0
{
  comment "Abstract метод не может быть strictfp"
  arbitrary
  absent
  target MethodModifiers {isAbstract}
  source MethodModifiers {isStrictfp}
  context c1 : same MethodHeader

}

// Line 1051
one_node relation R287_6_0
{
  comment "Native метод не может быть strictfp"
  arbitrary
  absent
  target MethodModifiers {isNative}
  source MethodModifiers {isStrictfp}
  context c1 : same MethodHeader

}

// Line 1078
one_node relation R309_ThrowsClauseTypes_0
{
  comment "Все типы указанные в throws clause должны приводиться к Throwable."
  arbitrary
  present
  target ThrowsClause.1.ExceptionTypeSpec {exceptionType}
  context c1 : same RootNode

}

// Line 1085
one_to_one relation R315_Method_Abstract_0
```

```
{
  comment "Абстрактный метод класса не должен иметь тело."
  arbitrary
  present
  target MethodDeclaration.1.MethodHeader.1.MethodModifiers[isAbstract!=null] {isAbstract}
  source this.target.^MethodDeclaration.1.EmptyMethodBody {dummy}
  context c1 : same ClassBody

}

// Line 1086
one_to_one relation R315_Method_Native_0
{
  comment "Native метод класса не должен иметь тело."
  arbitrary
  present
  target MethodDeclaration.1.MethodHeader.1.MethodModifiers[isNative!=null] {isNative}
  source this.target.^MethodDeclaration.1.EmptyMethodBody {dummy}
  context c1 : same ClassBody

}

// Line 1090
one_to_one relation R317_MandatoryReturn_0
{
  comment "Каждый метод возвращающий значение и имеющий тело должен содержать
оператор возврата значения на верхнем уровне (упрощенная версия)."
  arbitrary
  equal
  target MethodHeader[resType!=null] {resType}
  source
UsualMethodBody.1.Block.1.LastBlockStatement.1.CompoundReturnStatement.1.Expression
{type}
  context c1 : same MethodDeclaration[methodBody is UsualMethodBody]

}

// Line 1144
many_to_many relation R349ClassDecl_ConstructorName_0
{
  comment "Имя конструктора должно совпадать с именем класса в котором он объявлен."
  arbitrary
  equal
  target ClassDeclaration {ident}
  source this.target.2.ConstructorDeclaration.1.ConstructorDeclarator  {ident}
  context c1 : same RootNode

}

// Line 1145
one_node relation RFormalParameter_ParamInit_0
{
```

```
  comment "Параметры конструктора нельзя инициализировать."
  arbitrary
  present
  target FormalParameter.1.SimpleVariableDeclaratorWrapper {varDeclarator[is SimpleVariableDeclarator]}
  context c1 : same RootNode

}

// Line 1146
/*
many_to_many relation RConstructor_SignatureUniq_0
{
  comment "В одном классе нельзя объявлять методы с эквивалентными по переопределению (override) сигнатурами."
  arbitrary
  custom
  target this.target_context.1.ConstructorDeclaration.1.ConstructorDeclarator {varDeclarator[is SimpleVariableDeclarator]}
  source this.target_context.1.ConstructorDeclaration.1.ConstructorDeclarator {пусто}
  context c1 : same ClassBody
  context c2 : same InterfaceBody
  define {

  }
  synchronize {

  }

}
*/

// Line 1186
many_to_many relation RInterfaceEnclosingTypes_0
{
  comment "Имя вложенного интерфейса не должно совпадать с именами объемлющих классов или интерфейсов"
  arbitrary
  unequal
  target TypeDeclaration {ident}
  source this.target.00.InterfaceDeclaration {ident}
  context c1 : same RootNode

}

// Line 1187
one_node relation RInterfaceClassMembers_Protected_0
{
  comment "Модификаторы protected и private можно использовать только для интерфейсов вложенных в декларацию класса."
  arbitrary
  unequal
```

```
  target InterfaceDeclaration[^ClassDeclaration=null].1.Modifiers {accessibility
{<AccesibilityModifiers> not "AccesibilityModifiers.PROTECTED"}}
  context c1 : same RootNode

}

// Line 1188
one_node relation RInterfaceClassMembers_Private_0
{
  comment "Модификаторы protected и private можно использовать только для интерфейсов
вложенных в декларацию класса."
  arbitrary
  unequal
  target InterfaceDeclaration[^ClassDeclaration=null].1.Modifiers {accessibility
{<AccesibilityModifiers> not "AccesibilityModifiers.PRIVATE"}}
  context c1 : same RootNode

}

// Line 1189
one_node relation RInterfaceStaticMembers_0
{
  comment "Модификатор static может применяться только к вложенным (member)
интерфейсам."
  arbitrary
  absent
  target CompilationUnit.1.InterfaceDeclaration.1.Modifiers {isStatic}
  context c1 : same RootNode

}

// Line 1192
many_to_many relation RInterfTypeParam_ConstDecl_0
{
  comment "Типовые параметры нельзя использовать типовые парамеры интерфеса при
декларации полей."
  arbitrary
  unequal
  target ParameterizedInterfaceDeclaration.1.TypeParameters.00.TypeParameter {typeVar}
  source
this.target.^ParameterizedInterfaceDeclaration.2.ConstantDeclaration.1.VariableDeclaratorWrap
per.1.VariableDeclarator.1.SimpleVariableDeclaratorId {ident}
  context c1 : same RootNode

}

// Line 1193
many_to_many relation RInterfTypeParam_MemDecl_0
{
  comment "Типовые параметры нельзя использовать типовые парамеры интерфеса при
декларации внутреннего типа."
  arbitrary
```

```
  unequal
  target ParameterizedInterfaceDeclaration.1.TypeParameters.00.TypeParameter {typeVar}
  source this.target.^ParameterizedInterfaceDeclaration.2.TypeDeclaration {ident}
  context c1 : same RootNode

}

// Line 1207
/*
one_node relation RInterfaceFDecl_MayPublic_0
{
  comment "В декларации поля интерейса можно указывать public."
  arbitrary
  present
  target NormalInterfaceDeclaration.1.ConstantDeclaration.1.ConstantModifiers {isPublic}
  context c1 : same RootNode

}
*/

// Line 1208
/*
one_node relation RInterfaceFDecl_MayStatic_0
{
  comment "В декларации поля интерейса можно указывать static."
  arbitrary
  present
  target NormalInterfaceDeclaration.1.ConstantDeclaration.1.ConstantModifiers {isStatic}
  context c1 : same RootNode

}
*/

// Line 1209
/*
one_node relation RInterfaceFDecl_MayFinal_0
{
  comment "В декларации поля интерейса можно указывать final."
  arbitrary
  present
  target NormalInterfaceDeclaration.1.ConstantDeclaration.1.ConstantModifiers {isFinal}
  context c1 : same RootNode

}
*/

// Line 1213
/*
many_to_many relation RInterfaceFDecl_InitExpr_0
{
  comment "RInterfaceFDecl_InitExpr"
  arbitrary
```

```
  present
  target InterfaceDeclaration.2.ConstantDeclaration.1.VariableDeclaratorWrapper {isFinal}
  source this.target.1.VariableDeclarator[is CompoundVariableDeclarator] {пусто}
  context c1 : same RootNode

}
*/

// Line 1214
many_to_many relation RinterfaceFDecl_DefinedName_0
{
  comment "Простое имя поля не может использоваться в инициализаторах других полей
до его описания."
  semantic
  unequal
  target
ConstantDeclaration.1.VariableDeclaratorWrapper.1.VariableDeclarator.1.SimpleVariableDeclar
atorId {ident}
  source
ConstantDeclaration.1.VariableDeclaratorWrapper.1.CompoundVariableDeclarator.1.VariableIn
itializer.00.ExpressionName {ident}
  context c1 : same InterfaceDeclaration

}

// Line 1246
one_to_one relation RArrayInitElemType_0
{
  comment "Выражения элементов массива должны быть приводимы к типу элементов
массива."
  arbitrary
  compatible
  target this.target_context.00.SimpleVariableInitializer.1.Expression {type}
  source this.target_context {elemType}
  context c1 : same ArrayCreationExpression

}

// Line 1248
one_to_one relation RArrayInitRecursive_0
{
  comment "Тип вложенных инициализаторов массивов должен соответствовать типу
выражения."
  arbitrary
  equal
  target ArrayCreationExpression.1.ArrayVariableInitializer.(1.ArrayVariableInitializer)* {type}
  source this.target.^ArrayCreationExpression.type[is ArrayType].(1.ArrayType)* {type}
  context c1 : same RootNode

}

// Line 1251
```

```
many_to_many relation RLocalClassUniqueNames_0
{
  comment "Имена деклараций local классов дожны различаться в области видимости."
  semantic
  unequal
  target ClassDeclarationBlockStatement.00.ClassDeclaration {ident}
  source this.target.2.ClassDeclaration {ident}
  context c1 : same RootNode

}

// Line 1252
one_node relation RLocalClassModifiersAccess_0
{
  comment "Local class не может быть public, protected или private."
  arbitrary
  present
  target ClassDeclarationBlockStatement.1.ClassDeclaration.1.Modifiers {accessibility
{<AccesibilityModifiers> not "AccesibilityModifiers.PUBLIC"}}
  context c1 : same RootNode

}

// Line 1253
one_node relation RLocalClassModifiersStatic_0
{
  comment "Local class не может быть static."
  arbitrary
  absent
  target ClassDeclarationBlockStatement.1.ClassDeclaration.1.Modifiers {isStatic}
  context c1 : same RootNode

}

// Line 1256
many_to_many relation RLocalVarUniqueName_0
{
  comment "Локальная переменная должна иметь уникальное имя"
  arbitrary
  unequal
  target VariableDeclarator {varDeclId.ident}
  source VariableDeclarator {varDeclId.ident}
  context c1 : same Block


}

// Line 1259
many_to_many relation RLabelUniqueName_NormalNormal_0
{
  comment "Метки операторов должны быть уникальны внутри метода."
  arbitrary
```

```
  unequal
  target NormalLabeledStatement {label}
  source this.target.00.NormalLabeledStatement {label}
  context c1 : same Block


}

// Line 1260
many_to_many relation RLabelUniqueName_NormalAbrupt_0
{
  comment "Метки операторов должны быть уникальны внутри метода."
  arbitrary
  unequal
  target NormalLabeledStatement {label}
  source this.target.00.AbruptLabeledStatement {label}
  context c1 : same Block


}

// Line 1261
many_to_many relation RLabelUniqueName_AbruptNormal_0
{
  comment "Метки операторов должны быть уникальны внутри метода."
  arbitrary
  unequal
  target AbruptLabeledStatement {label}
  source this.target.00.NormalLabeledStatement {label}
  context c1 : same Block


}

// Line 1262
many_to_many relation RLabelUniqueName_AbruptAbrupt_0
{
  comment "Метки операторов должны быть уникальны внутри метода."
  arbitrary
  unequal
  target AbruptLabeledStatement {label}
  source this.target.00.AbruptLabeledStatement {label}
  context c1 : same Block


}

// Line 1263
one_to_one relation RLabelStmtNormalBreak_0
{
  comment "Оператор завершается нормально, если вложенный оператор завершается
abruptly по оператору break с совпадающей меткой."
```

```
  arbitrary
  equal
  target ANLabeledStatement {label}
  source this.target.00.AbruptGotoStatement {optLabel}
  context c1 : same Block


}

// Line 1265
one_node relation RNormalIfStmtExprType_0
{
  comment "Тип выражения в операторе if дожен быть boolean."
  arbitrary
  present
  target NormalIfStatement {condition.type[is BooleanType]}
  context c1 : same Block


}

// Line 1266
one_node relation RAbruptIfStmtExprType_0
{
  comment "Тип выражения в операторе if дожен быть boolean."
  arbitrary
  present
  target AbruptIfStatement {condition.type[is BooleanType]}
  context c1 : same Block


}

// Line 1268
one_node relation RAssertStmtExprType_0
{
  comment "Тип условия в операторе assert дожен быть boolean."
  arbitrary
  present
  target AssertStatement {condition.type[is BooleanType]}
  context c1 : same Block


}

// Line 1270
one_node relation RNormalSwitchExprType_0
{
  comment "Тип выражения в операторе switch может быть только char, byte, short, int или
enum."
  arbitrary
  present
```

```
  target NormalSwitchStatement {condition.type {<IntegralType> not "IntegralType.LONG"}}
  context c1 : same Block

}

// Line 1271
one_node relation RAbruptSwitchExprType_0
{
  comment "Тип выражения в операторе switch может быть только char, byte, short, int или
enum."
  arbitrary
  present
  target AbruptSwitchStatement {condition.type {<IntegralType> not "IntegralType.LONG"}}
  context c1 : same Block

}

// Line 1272
one_to_many relation RNormalSwitchLabelComp_0
{
  comment "Тип выражения любой метки должен приводиться к типу выражения условия."
  arbitrary
  compatible
  target NormalSwitchStatement.1.SwitchElement.1.SwitchLabel.1.ConstantExpression {type}
  source this.target.^NormalSwitchStatement {condition.type}
  context c1 : same Block

}

// Line 1273
one_to_many relation RAbruptSwitchLabelComp_0
{
  comment "Тип выражения любой метки должен приводиться к типу выражения условия."
  arbitrary
  compatible
  target AbruptSwitchStatement.1.SwitchElement.1.SwitchLabel.1.ConstantExpression {type}
  source this.target.^AbruptSwitchStatement {condition.type}
  context c1 : same Block

}

// Line 1274
one_node relation RNormalSwitchLabelNotNull_0
{
  comment "Метки не могут быть null."
  arbitrary
  present
  target NormalSwitchStatement.1.SwitchElement.1.SwitchLabel {constantExpr[notis
NullLiteral]}
  context c1 : same Block

}
```

```
// Line 1275
one_node relation RAbruptSwitchLabelNotNull_0
{
  comment "Метки не могут быть null."
  arbitrary
  present
  target AbruptSwitchStatement.1.SwitchElement.1.SwitchLabel {constantExpr[notis
NullLiteral]}
  context c1 : same Block

}

// Line 1276
many_to_many relation RNormalSwitchLabelUnique_0
{
  comment "Метки не могут быть одинаковыми."
  arbitrary
  unequal
  target this.target_context.1.SwitchElement.1.SwitchLabel {constantExpr}
  source this.target_context.1.SwitchElement.1.SwitchLabel {constantExpr}
  context c1 : same NormalSwitchStatement

}

// Line 1277
many_to_many relation RAbruptSwitchLabelUnique_0
{
  comment "Метки не могут быть одинаковыми."
  arbitrary
  unequal
  target this.target_context.1.SwitchElement.1.SwitchLabel {constantExpr}
  source this.target_context.1.SwitchElement.1.SwitchLabel {constantExpr}
  context c1 : same AbruptSwitchStatement

}

// Line 1279
one_node relation RNormalSwitchNoDefault_0
{
  comment "Оператор с отсутствующей default меткой может завершиться нормально"
  arbitrary
  absent
  target NoDefaultNormSwitchStatement {defaultLabel}
  context c1 : same Block

}

// Line 1280
one_to_one relation RNormalSwitchBreakable_0
{
  comment "Оператор с в котором есть оператор break может завершиться нормально"
```

```
  arbitrary
  present
  target BreakNormSwitchStatement {condition}
  source this.target.1.SwitchElement.1.BreakStatement[optLabel=null] {dummy}
  context c1 : same Block

}

// Line 1281
/*
one_to_one relation RNormalSwitchLeave_0
{
  comment "RNormalSwitchLeave"
  arbitrary
  present
  target LeaveNormSwitchStatement {condition}
  context c1 : same RootNode

}
*/

// Line 1282
one_to_one relation RAbruptSwitchNotBreakable_0
{
  comment "Оператор завершается abruptly если 1) есть default, 2) блоки не заканчиваются
break, 3) последний блок заканчивается Abrupt!=break"
  arbitrary
  present
  target AbruptSwitchStatement {condition}
  source this.target.1.SwitchElement.1.AbruptStatement[notis BreakStatement] {dummy}
  context c1 : same Block

}

// Line 1283
one_node relation RNormalWhileStmtExprType_0
{
  comment "Тип условия в операторе while дожен быть boolean."
  arbitrary
  present
  target NormalWhileStatement {condition.type[is BooleanType]}
  context c1 : same Block


}

// Line 1284
one_node relation RAbruptWhileStmtExprType_0
{
  comment "Тип условия в операторе while дожен быть boolean."
  arbitrary
  present
```

```
  target AbruptWhileStatement {condition.type[is BooleanType]}
  context c1 : same Block


}

// Line 1285
one_node relation RNormalDoStmtExprType_0
{
  comment "Тип условия в операторе do дожен быть boolean."
  arbitrary
  present
  target NormalDoStatement {condition.type[is BooleanType]}
  context c1 : same Block


}

// Line 1286
one_node relation RAbruptDoStmtExprType_0
{
  comment "Тип условия в операторе do дожен быть boolean."
  arbitrary
  present
  target AbruptDoStatement {condition.type[is BooleanType]}
  context c1 : same Block


}

// Line 1287
one_node relation RNormalBasicForStmtExprType_0
{
  comment "Тип условия в операторе basic for дожен быть boolean."
  arbitrary
  present
  target NormalBasicForStatement.optCondition {type[is BooleanType]}
  context c1 : same Block


}

// Line 1288
one_node relation RAbruptBasicForStmtExprType_0
{
  comment "Тип условия в операторе basic for дожен быть boolean."
  arbitrary
  present
  target AbruptBasicForStatement.optCondition {type[is BooleanType]}
  context c1 : same Block
```

```
}

// Line 1289
one_node relation RNormalEnhancedForStmtExprType_0
{
  comment "Тип выражения в операторе enhanced for дожен быть либо Iterable, либо
ArrayType."
  arbitrary
  present
  target NormalEnhancedForStatement.expression {type[is ArrayType]}
  context c1 : same Block


}

// Line 1290
one_node relation RAbruptEnhancedForStmtExprType_0
{
  comment "Тип выражения в операторе enhanced for дожен быть либо Iterable, либо
ArrayType."
  arbitrary
  present
  target AbruptEnhancedForStatement.expression {type[is ArrayType]}
  context c1 : same Block


}

// Line 1291
one_to_many relation RBreakNoLabelContext_0
{
  comment "Оператор break без метки должен находиться внутри операторов switch, while,
do, или for."
  arbitrary
  absent
  target BreakStatement[optLabel=null] {optLabel}
  source this.target.^NormalBreakableStatement {dummy}
  context c1 : same Block


}

// Line 1292
one_to_many relation RBreakLabelContext_0
{
  comment "Оператор break с меткой должен находиться внутри оператора помеченного
такой же меткой."
  arbitrary
  equal
  target BreakStatement[optLabel!=null] {optLabel}
  source this.target.^NormalLabeledStatement {label}
  context c1 : same Block
```

```
}

// Line 1293
one_to_many relation RContinueNoLabelContext_0
{
  comment "Оператор continue без метки должен находиться внутри операторов while, do,
или for."
  arbitrary
  absent
  target ContinueStatement[optLabel=null] {optLabel}
  source this.target.^NormalLoopStatement {dummy}
  context c1 : same Block


}

// Line 1294
one_to_many relation RContinueLabelContext_0
{
  comment "Оператор continue с меткой должен находиться внутри оператора while, do, или
for помеченного такой же меткой."
  arbitrary
  equal
  target ContinueStatement[optLabel!=null] {optLabel}
  source this.target.^NormalLabeledStatement.1.NormalLoopStatement {parent.label}
  context c1 : same Block


}

// Line 1296
many_to_many relation RCompoundReturn_0
{
  comment "Тип возвращаемого выражения должен приводиться к типу возвращаемого
значения метода"
  arbitrary
  equal
  target MethodHeader {resType[!=null]}
  source
this.source_context.1.UsualMethodBody.1.Block.1.LastBlockStatement.1.CompoundReturnStat
ement.1.Expression {type}
  context c1 : same MethodDeclaration

}

// Line 1297
many_to_many relation RVoidReturn_0
{
  comment "Если метод возвращает void, то все операторы return не должны содержать
выражение"
```

```
  arbitrary
  absent
  target MethodHeader {resType}
  source
this.source_context.1.UsualMethodBody.1.Block.1.LastBlockStatement.1.VoidReturnStatement
{dummy}
  context c1 : same MethodDeclaration

}

// Line 1298
one_node relation RThrowExprType_0
{
  comment "Выражение должно приводиться к типу Throwable."
  arbitrary
  present
  target ThrowStatement {expression.type[is ExceptionType]}
  context c1 : same Block

}

// Line 1299
one_node relation RRuntimeThrowExprType_0
{
  comment "Тип выбрасываемого исключения приводится к типу RuntimeException."
  arbitrary
  present
  target RuntimeExceptionThrowStatement {expression.type[is
JavaLangRuntimeExceptionType]}
  context c1 : same Block

}

// Line 1300
one_node relation RErrorThrowExprType_0
{
  comment "Тип выбрасываемого исключения приводится к типу Error."
  arbitrary
  present
  target ErrorThrowStatement {expression.type[is JavaLangErrorType]}
  context c1 : same Block

}

// Line 1302
one_node relation RThrowAssignableType_MethodThrows_0
{
  comment "Тип выражения оператора throw приводится к одному из типов throws списка
декларации метода, содержащего этот оператор."
  arbitrary
  compatible
  target MethodThrowsClauseThrowStatement {expression.type}
```

```
  source this.target_context.1.MethodHeader.1.ThrowsClause.ExceptionTypeSpec
{exceptionType}
  context c1 : same MethodDeclaration

}

// Line 1303
one_node relation RThrowAssignableType_AbstractMethodThrows_0
{
  comment "Тип выражения оператора throw приводится к одному из типов throws списка
декларации метода, содержащего этот оператор."
  arbitrary
  compatible
  target MethodThrowsClauseThrowStatement {expression.type}
  source this.target_context.1.ThrowsClause.ExceptionTypeSpec {exceptionType}
  context c1 : same AbstractMethodDeclaration

}

// Line 1304
one_node relation RThrowAssignableType_ConstructorThrows_0
{
  comment "Тип выражения оператора throw приводится к одному из типов throws списка
декларации метода, содержащего этот оператор."
  arbitrary
  compatible
  target MethodThrowsClauseThrowStatement {expression.type}
  source this.target_context.1.ThrowsClause.ExceptionTypeSpec {exceptionType}
  context c1 : same ConstructorDeclaration

}

// Line 1305
one_node relation RNormalSynchronizedExprType_0
{
  comment "Тип выражения в операторе synchronized должен быть ссылочным."
  arbitrary
  present
  target NormalSynchronizedStatement {expression.type[is ReferenceType]}
  context c1 : same Block

}

// Line 1306
one_node relation RAbruptSynchronizedExprType_0
{
  comment "Тип выражения в операторе synchronized должен быть ссылочным."
  arbitrary
  present
  target AbruptSynchronizedStatement {expression.type[is ReferenceType]}
  context c1 : same Block
```

```
}

// Line 1307
one_node relation RTryCatchParamType_Normal_0
{
  comment "Тип параметра исключения должен быть подклассом Throwable."
  arbitrary
  present
  target
NormalTryStatement.1.CatchClause.1.FormalParameter.1.SimpleVariableDeclaratorWrapper
{type[is JavaLangThrowableType]}
  context c1 : same RootNode

}

// Line 1308
one_node relation RTryCatchParamType_Abrupt_0
{
  comment "Тип параметра исключения должен быть подклассом Throwable."
  arbitrary
  present
  target
AbruptTryStatement.1.CatchClause.1.FormalParameter.1.SimpleVariableDeclaratorWrapper
{type[is JavaLangThrowableType]}
  context c1 : same RootNode

}

// Line 1309
many_to_many relation RTryCatchParamName_Normal_0
{
  comment "Параметр catch блока не должен совпадать с параметром метода или локальной
переменной."
  semantic
  unequal
  target
NormalTryStatement.1.CatchClause.1.FormalParameter.1.SimpleVariableDeclaratorWrapper.1.
VariableDeclarator {varDeclId.ident}
  source VariableDeclarator {varDeclId.ident}
  context c1 : same RootNode

}

// Line 1310
many_to_many relation RTryCatchParamName_Abrupt_0
{
  comment "Параметр catch блока не должен совпадать с параметром метода или локальной
переменной."
  semantic
  unequal
```

```
  target
AbruptTryStatement.1.CatchClause.1.FormalParameter.1.SimpleVariableDeclaratorWrapper.1.V
ariableDeclarator {varDeclId.ident}
  source VariableDeclarator {varDeclId.ident}
  context c1 : same RootNode

}

// Line 1312
many_to_many relation RTryCatchParamFinal_Normal_0
{
  comment "Параметр catch блока декларированный как final нельзя менять внутри catch
блока."
  arbitrary
  unequal
  target Assignment.1.ExprNameLeftHandSide.1.ExpressionName {ident}
  source
this.target_context.1.FormalParameter.modifiers[isFinal!=null].parent.1.SimpleVariableDeclarat
orWrapper.1.VariableDeclarator {varDeclId.ident}
  context c1 : same NormalTryStatement.1.CatchClause

}

// Line 1313
many_to_many relation RTryCatchParamFinal_Abrupt_0
{
  comment "Параметр catch блока декларированный как final нельзя менять внутри catch
блока."
  arbitrary
  unequal
  target Assignment.1.ExprNameLeftHandSide.1.ExpressionName {ident}
  source
this.target_context.1.FormalParameter.modifiers[isFinal!=null].parent.1.SimpleVariableDeclarat
orWrapper.1.VariableDeclarator {varDeclId.ident}
  context c1 : same AbruptTryStatement.1.CatchClause

}

// Line 1318
one_node relation RIntLiteralType_0
{
  comment "Тип IntegerLiteral есть int"
  arbitrary
  equal
  target IntegerLiteral {type {<IntegralType> "IntegralType.INT"}}
  context c1 : same RootNode

}

// Line 1319
one_node relation RLongLiteralType_0
{
```

```
 comment "Тип LongLiteral есть long"
 arbitrary
 equal
 target LongLiteral {type {<IntegralType> "IntegralType.LONG"}}
 context c1 : same RootNode

}

// Line 1320
one_node relation RFloatLiteralType_0
{
 comment "Тип FloatingPointLiteral есть float"
 arbitrary
 equal
 target FloatingPointLiteral {type {<FloatingPointType> "FloatingPointType.FLOAT"}}
 context c1 : same RootNode

}

// Line 1321
one_node relation RDoubleLiteralType_0
{
 comment "Тип DoublePointingLiteral есть double"
 arbitrary
 equal
 target DoublePointingLiteral {type {<FloatingPointType> "FloatingPointType.DOUBLE"}}
 context c1 : same RootNode

}

// Line 1322
one_node relation RBoolLiteralType_0
{
 comment "Тип BooleanLiteral есть boolean"
 arbitrary
 equal
 target BooleanLiteral {type {<BooleanType> ""}}
 context c1 : same RootNode

}

// Line 1323
one_node relation RCharLiteralType_0
{
 comment "Тип CharacterLiteral есть char"
 arbitrary
 equal
 target CharacterLiteral {type {<IntegralType> "IntegralType.CHAR"}}
 context c1 : same RootNode

}
```

```
// Line 1324
one_node relation RStringLiteralType_0
{
  comment "Тип StringLiteral есть java.lang.String"
  arbitrary
  equal
  target StringLiteral {type {<JavaLangStringType> ""}}
  context c1 : same RootNode

}

// Line 1325
one_node relation RNullLiteralType_0
{
  comment "Тип NullLiteral есть Null reference"
  arbitrary
  equal
  target NullLiteral {type {<NullType> ""}}
  context c1 : same RootNode

}

// Line 1326
one_node relation RClassLiteralType_0
{
  comment "Тип ClassLiteral есть java.lang.Class"
  arbitrary
  equal
  target ClassLiteral {type {<JavaLangClassType> ""}}
  context c1 : same RootNode

}

// Line 1328
one_to_many relation RThisLiteralUseInMethod_0
{
  comment "Метод в котором используется ключевое слово this не может быть
статическим"
  arbitrary
  absent
  target ThisAccessExpression {type}
  source this.target_context.1.MethodHeader.1.MethodModifiers {isStatic}
  context c1 : same MethodDeclaration

}

// Line 1329
/*
one_to_many relation RThisLiteralUseInConstructor_0
{
  comment "Ключевое слово this может использоваться в конструкторе"
  arbitrary
```

```
  present
  target ThisAccessExpression {type}
  source this.target_context {constructorBody}
  context c1 : same ConstructorDeclaration

}
*/

// Line 1330
/*
one_to_many relation RThisLiteralForbidden_StaticInitializer_0
{
  comment "Ключевое слово this не может использоваться в статическом инициализаторе"
  arbitrary
  present
  target ThisAccessExpression {type}
  source this.target.^ClassBodyDeclaration[notis StaticInitializer] {пусто}
  context c1 : same CompilationUnit

}
*/

// Line 1331
one_to_many relation RThisLiteralUseInFieldInitializer_0
{
  comment "Поле при инициализации которого используется ключевое слово this не может
быть статическим"
  arbitrary
  absent
  target ThisAccessExpression {type}
  source this.target_context.1.FieldModifiers {isStatic}
  context c1 : same FieldDeclaration

}

// Line 1332
one_to_many relation RThisLiteralType_Simple_0
{
  comment "Тип выражения this - это класс в котором он встретился"
  arbitrary
  equal
  target SimpleThisAccessExpression.00.TypeName {ident}
  source this.target.^TypeDeclaration {ident}
  context c1 : same CompilationUnit

}

// Line 1333
one_to_many relation RThisLiteralClassName_0
{
  comment "ClassName должен быть n-ым лексически enclosing классом в котором
встретилось выражение."
```

```
  arbitrary
  equal
  target QualifiedThisAccessExpression.1.ClassName {ident}
  source this.target.^TypeDeclaration.(^TypeDeclaration)* {ident}
  context c1 : same CompilationUnit

}

// Line 1334
one_to_one relation RThisLiteralType_Qualified_0: RThisLiteralClassName_0
{
  comment "Тип выражения ClassName.this - это класс ClassName"
  arbitrary
  equal
  target super.target.^QualifiedThisAccessExpression.00.TypeName {ident}
  source super.source {ident}
  context c1 : same RootNode

}

// Line 1335
one_node relation RParenthesizedExpressionType_0
{
  comment "Скобочное выражение имеет такой же тип как и выражение в скобках"
  arbitrary
  equal
  target ParenthesizedExpression {type}
  source this.target {expression.type}
  context c1 : same RootNode

}

// Line 1336
one_node relation RNewClassTypeArgWildcard_0
{
  comment "Типовые аргументы не могут быть wildcard."
  arbitrary
  present
  target ClassInstanceCreationExpression.1.TypeArguments.00.ActualTypeArgumentList
{argument[notis WildcardActualTypeArgument]}
  context c1 : same RootNode

}

// Line 1339
one_node relation RNewClassFinalDecl_0: R229_TD_0
{
  comment "Соответствующая декларация класса не должна быть final."
  arbitrary
  absent
```

```
  target
NewClassInstanceCreationExpression.fullTypeName.00.TypeName.R229_TD_0[target=prev].so
urce[is ClassDeclaration].1.Modifiers {isFinal}
  context c1 : same RootNode

}

// Line 1340
one_to_many relation RPrimaryClassDeclIdent_0: R229_TD_0
{
  comment "Идентификатору должен соответствовать внутренний non-final, доступный
класс внутри декларации типа Primary."
  arbitrary
  equal
  target PrimaryClassInstanceCreationExpression {ident}
  source
this.target.1.Primary.00.TypeName.R229_TD_0[target=prev].source.00.ClassDeclaration
{ident}
  context c1 : same RootNode

}

// Line 1341
one_node relation RPrimaryClassDeclFinal_0: RPrimaryClassDeclIdent_0
{
  comment "Внутренний тип Primary должен быть non-final."
  arbitrary
  absent
  target super.source.1.Modifiers {isFinal}
  context c1 : same RootNode

}

// Line 1343
one_to_many relation RNamedNewClassAbstract_0: R229_TD_0
{
  comment "Создавать можно только классы и притом не абстрактные."
  arbitrary
  present
  target NamedNewClassInstanceCreationExpression {fullTypeName}
  source this.target.fullTypeName.00.TypeName.R229_TD_0[target=prev].source[is
ClassDeclaration].1.Modifiers[isAbstract=null] {accessibility}
  context c1 : same RootNode

}

// Line 1344
one_node relation RNamedPrimaryClassAbstract_0: RPrimaryClassDeclIdent_0
{
  comment "Идентификатору должен соответствовать только класс внутренний для типа
Primary и притом не абстрактный."
  arbitrary
```

```
  absent
  target super.source.1.Modifiers {isAbstract}
  context c1 : same RootNode

}

// Line 1346
one_node relation RNewClassType_0
{
  comment "Тип выражения new совпадает с типом создаваемого класса."
  arbitrary
  equal
  target NewClassInstanceCreationExpression {type}
  source NewClassInstanceCreationExpression {fullTypeName}
  context c1 : same CompilationUnit

}

// Line 1347
one_node relation RPrimaryClassType_0
{
  comment "Тип выражения primary.new совпадает с типом создаваемого класса."
  arbitrary
  equal
  target PrimaryClassInstanceCreationExpression {ident}
  source PrimaryClassInstanceCreationExpression.type.00.TypeName {ident}
  context c1 : same CompilationUnit

}

// Line 1348
one_node relation RPrimaryClassType_Ref_0
{
  comment "Тип выражения PrimaryClassInstanceCreationExpression может быть только
ссылочным."
  arbitrary
  present
  target PrimaryClassInstanceCreationExpression {type[is ReferenceType]}
  context c1 : same CompilationUnit

}

// Line 1357
/*
one_to_one relation RNewClassAnonymConstructor_0
{
  comment "Анонимный класс не может иметь конструкторов."
  arbitrary
  present
  target ClassInstanceCreationExpression.1.ClassBody.1.ClassBodyDeclaration {parent}
  source this.target[notis ConstructorDeclaration] {пусто}
  context c1 : same RootNode
```

```
}
*/

// Line 1359
one_node relation RArrayCreationType_0
{
  comment "Тип выражения создания массива всегда массив."
  arbitrary
  present
  target ArrayCreationExpression {type[is ArrayType]}
  context c1 : same RootNode

}

// Line 1361
one_node relation RArrayCreationElementType_0
{
  comment "Тип элемента массива не может быть параметризованным типом."
  arbitrary
  present
  target ArrayCreationExpression {elemType[notis ParameterizedType]}
  context c1 : same RootNode

}

// Line 1362
one_to_one relation RArrayCreationTypeRecursion_0
{
  comment "Вложенность типа массива совпадает с количеством Dim выражений."
  arbitrary
  equal
  target ArrayCreationExpression.type.1.ArrayType.(1.ArrayType)* {parent.type}
  source this.target.^ArrayCreationExpression.1.DimExprs.1.DimExprs.(1.DimExprs)*
{auxType}
  context c1 : same RootNode

}

// Line 1363
one_to_one relation RArrayCreationTypeMiddle_0
{
  comment "Тип массива определяется количеством Dim выражений."
  arbitrary
  equal
  target DimExprs.1.DimExprs {auxType}
  source this.target.parent {auxType.type}
  context c1 : same ArrayCreationExpression

}

// Line 1364
```

```
/*
one_to_one relation RArrayCreationTypeStart_0
{
  comment "Тип массива определяется количеством Dim выражений."
  arbitrary
  equal
  target ArrayCreationExpression.1.DimExprs {auxType.type}
  source this.target.parent {type}
  context c1 : same RootNode

}
*/

// Line 1365
one_to_one relation RArrayCreationTypeEndElem1_0
{
  comment "Тип массива определяется количеством Dim выражений."
  arbitrary
  equal
  target ArrayCreationExpression.00.SimpleDim {auxType.type}
  source this.target.^ArrayCreationExpression {elemType}
  context c1 : same RootNode

}

// Line 1366
one_to_one relation RArrayCreationTypeEndElem2_0
{
  comment "Тип массива определяется количеством Dim выражений."
  arbitrary
  equal
  target ArrayCreationExpression.00.SimpleDimExpr {auxType.type}
  source this.target.^ArrayCreationExpression {elemType}
  context c1 : same RootNode

}

// Line 1367
one_node relation RArrayCreationExpressionType_0
{
  comment "Тип выражений определяющих размер массива должен быть интегральным."
  arbitrary
  present
  target ExprArrayCreationExpression.00.SimpleDimExpr.1.Expression {type[is IntegralType]}
  context c1 : same RootNode

}

// Line 1368
one_node relation RArrayCreationExpressionIntType_0
{
  comment "Тип выражений определяющих размер массива должен приводиться к int."
```

```
  arbitrary
  unequal
  target ExprArrayCreationExpression.00.SimpleDimExpr.1.Expression {type {<IntegralType>
"IntegralType.INT"}}
  context c1 : same RootNode

}

// Line 1370
one_node relation RPrimaryType_0
{
  comment "Тип выражения Primary должен быть ссылочным"
  arbitrary
  present
  target PrimaryFieldAccess.1.Primary {type[is ReferenceType]}
  context c1 : same RootNode

}

// Line 1371
one_to_one relation RFieldDeclExists_Primary_0: R229_TD_0
{
  comment "Поле доступа должно быть описано в соответствующем классе"
  arbitrary
  equal
  target super.target.^PrimaryFieldAccess {ident}
  source super.source.2.FieldDeclaration.2.VariableDeclarator {varDeclId.ident}
  context c1 : same RootNode

}

// Line 1372
one_to_one relation RFieldAccessType_Primary_0: RFieldDeclExists_Primary_0
{
  comment "Тип FieldAccess совпадает с типом поля [после capture conversion]"
  arbitrary
  equal
  target super.target {type}
  source super.source.^VariableDeclaratorWrapper {type}
  context c1 : same RootNode

}

// Line 1374
one_to_many relation RFieldDeclExists_Super_0: R229_TD_0
{
  comment "Поле доступа должно быть описано в соответствующем суперклассе "
  arbitrary
  equal
  target SuperFieldAccess {ident}
```

```
  source
this.target.^TypeDeclaration.1.Super.1.ClassOrInterfaceType.00.TypeName.R229_TD_0[target=
prev].source.2.FieldDeclaration.2.VariableDeclarator {varDeclId.ident}
  context c1 : same RootNode

}

// Line 1375
one_to_one relation RFieldAccessType_Super_0: RFieldDeclExists_Super_0
{
  comment "Тип FieldAccess совпадает с типом поля [после capture conversion]"
  arbitrary
  equal
  target super.target {type}
  source super.source.^VariableDeclaratorWrapper {type}
  context c1 : same RootNode

}

// Line 1376
one_to_many relation RFieldSuperClassName_0
{
  comment "Текущий класс должен быть подклассом или совпадать с указанным типом
класса."
  arbitrary
  equal
  target SuperStaticFieldAccess.1.ClassName {ident}
  source this.target.^TypeDeclaration.(^TypeDeclaration)* {ident}
  context c1 : same CompilationUnit

}

// Line 1377
one_to_one relation RFieldDeclExists_StaticSuper_0: RFieldSuperClassName_0
{
  comment "Поле доступа должно быть описано в соответствующем суперклассе "
  arbitrary
  equal
  target super.target.^SuperStaticFieldAccess {ident}
  source super.source.2.FieldDeclaration.2.VariableDeclarator {varDeclId.ident}
  context c1 : same RootNode

}

// Line 1378
one_to_one relation RFieldAccessType_StaticSuper_0: RFieldDeclExists_StaticSuper_0
{
  comment "Тип FieldAccess совпадает с типом поля [после capture conversion]"
  arbitrary
  equal
  target super.target {type}
  source super.source.^VariableDeclaratorWrapper {type}
```

```
  context c1 : same RootNode

}

// Line 1379
one_node relation RMethodExpr_VoidTypeUse_0
{
  comment "Вызов void метода может быть только на верхнем уровне"
  arbitrary
  present
  target ExpressionStatement.1.StatementExpression.>=2.MethodInvocation {resType}
  context c1 : same RootNode

}

// Line 1380
one_node relation RMethodExpr_VoidTypeUseAbrupt_0
{
  comment "Запрещено использование void методов везде кроме ExpressionStatement на верхнем уровне"
  arbitrary
  present
  target this.target_context.00.MethodInvocation {resType}
  context c1 : same NormalStatement[notis ExpressionStatement]
  context c2 : same AbruptStatement

}

// Line 1381
one_node relation RMethodExpr_ResType_0
{
  comment "Для методов не верхнего уровня тип выражения вызов метода совпадает с типом возвращаемого значения"
  arbitrary
  equal
  target ExpressionStatement.1.StatementExpression.>=2.MethodInvocation {resType}
  source ExpressionStatement.1.StatementExpression.>=2.MethodInvocation {type}
  context c1 : same RootNode

}

// Line 1382
one_node relation RMethodExpr_ResTypeAbrupt_0
{
  comment "Для методов вне ExpressionStatement тип выражения вызов метода совпадает с типом возвращаемого значения"
  arbitrary
  equal
  target this.target_context.00.MethodInvocation {resType}
  source this.target_context.00.MethodInvocation {type}
  context c1 : same NormalStatement[notis ExpressionStatement]
  context c2 : same AbruptStatement
```

```
}

// Line 1383
one_to_many relation RSimpleMethodName_0
{
  comment "Имя вызываемого метода должно быть доступно в области видимости"
  arbitrary
  equal
  target SimpleMethodName {ident}
  source this.target.^ClassBody.1.MethodDeclaration.1.MethodHeader.1.MethodDeclarator
{ident}
  context c1 : same CompilationUnit

}

// Line 1384
one_to_one relation RMethodExpr_Simple_0: RSimpleMethodName_0
{
  comment "Тип выражения вызов метода совпадает с типом возвращаемого значения
метода"
  arbitrary
  equal
  target super.target.^MethodInvocation {resType}
  source super.source.^MethodDeclaration.1.MethodHeader {resType}
  context c1 : same RootNode

}

// Line 1385
one_to_many relation RTypeMethodName_0: R229_TD_0
{
  comment "Имя вызываемого метода должно быть доступно в области видимости"
  arbitrary
  equal
  target TypeMethodName {ident}
  source
this.target.1.ClassOrInterfaceType.00.TypeName.R229_TD_0[target=prev].source.2.MethodDec
laration.1.MethodHeader.1.MethodDeclarator {ident}
  context c1 : same CompilationUnit

}

// Line 1386
one_to_one relation RMethodExpr_Type_0: RTypeMethodName_0
{
  comment "Тип выражения вызов метода совпадает с типом возвращаемого значения
метода"
  arbitrary
  equal
  target super.target.^MethodInvocation {resType}
  source super.source.^MethodDeclaration.1.MethodHeader {resType}
```

```
  context c1 : same RootNode

}

// Line 1388
one_to_many relation RFieldMethodName_0: R229_TD_0
{
  comment "Имя вызываемого метода должно быть доступно в области видимости"
  arbitrary
  equal
  target FieldMethodName {ident}
  source
this.target.2.ClassOrInterfaceType.00.TypeName.R229_TD_0[target=prev].source.2.MethodDec
laration.1.MethodHeader.1.MethodDeclarator {ident}
  context c1 : same CompilationUnit

}

// Line 1389
one_to_one relation RMethodExpr_Field_0: RFieldMethodName_0
{
  comment "Тип выражения вызов метода совпадает с типом возвращаемого значения
метода"
  arbitrary
  equal
  target super.target.^MethodInvocation {resType}
  source super.source.^MethodDeclaration.1.MethodHeader {resType}
  context c1 : same RootNode

}

// Line 1390
one_to_many relation RMethodDeclExists_Primary_0: R229_TD_0
{
  comment "Метод должен быть описан в классе соответствующем типу Primary
выражения"
  arbitrary
  equal
  target PrimaryMethodInvocation {name}
  source
this.target.2.ClassOrInterfaceType.00.TypeName.R229_TD_0[target=prev].source.2.MethodDec
laration.1.MethodHeader.1.MethodDeclarator {ident}
  context c1 : same RootNode

}

// Line 1391
one_to_one relation RMethodExpr_Primary_0: RMethodDeclExists_Primary_0
{
  comment "Тип выражения вызов метода совпадает с типом возвращаемого значения
метода"
  arbitrary
```

```
  equal
  target super.target.^MethodInvocation {resType}
  source super.source.^MethodDeclaration.1.MethodHeader {resType}
  context c1 : same RootNode

}

// Line 1392
one_to_many relation RMethodDeclExists_Super_0: R229_TD_0
{
  comment "Метод должен быть описан в соответствующем суперклассе"
  arbitrary
  equal
  target SuperMethodInvocation {name}
  source
this.target.^TypeDeclaration.1.Super.1.ClassOrInterfaceType.00.TypeName.R229_TD_0[target=
prev].source.2.MethodDeclaration.1.MethodHeader.1.MethodDeclarator {ident}
  context c1 : same RootNode

}

// Line 1393
one_to_one relation RMethodExpr_Super_0: RMethodDeclExists_Super_0
{
  comment "Тип выражения вызов метода совпадает с типом возвращаемого значения
метода"
  arbitrary
  equal
  target super.target.^MethodInvocation {resType}
  source super.source.^MethodDeclaration.1.MethodHeader {resType}
  context c1 : same RootNode

}

// Line 1394
one_to_many relation RMethodClassSuper_0
{
  comment "Текущий класс должен быть подклассом или совпадать с указанным типом
класса."
  arbitrary
  equal
  target ClassSuperMethodInvocation.1.ClassName {ident}
  source this.target.^TypeDeclaration.(^TypeDeclaration)* {ident}
  context c1 : same CompilationUnit

}

// Line 1395
one_to_many relation RMethodDeclExists_ClassSuper_0: RMethodClassSuper_0
{
  comment "Метод должен быть описан в соответствующем суперклассе"
  arbitrary
```

```
  equal
  target super.target.^ClassSuperMethodInvocation {name}
  source super.source.2.MethodDeclaration.1.MethodHeader.1.MethodDeclarator {ident}
  context c1 : same RootNode

}

// Line 1396
one_to_one relation RMethodExpr_ClassSuper_0: RMethodDeclExists_ClassSuper_0
{
  comment "Тип выражения вызов метода совпадает с типом возвращаемого значения
метода"
  arbitrary
  equal
  target super.target.^MethodInvocation {resType}
  source super.source.^MethodDeclaration.1.MethodHeader {resType}
  context c1 : same RootNode

}

// Line 1397
one_to_many relation RMethodDeclExists_Type_0: R229_TD_0
{
  comment "Метод должен быть описан в соответствующем классе"
  arbitrary
  equal
  target TypeMethodInvocation {name}
  source
this.target.1.ClassOrInterfaceType.00.TypeName.R229_TD_0[target=prev].source.2.MethodDec
laration.1.MethodHeader.1.MethodDeclarator {ident}
  context c1 : same RootNode

}

// Line 1398
one_to_one relation RMethodExpr_2Type_0: RMethodDeclExists_Type_0
{
  comment "Тип выражения вызов метода совпадает с типом возвращаемого значения
метода"
  arbitrary
  equal
  target super.target.^MethodInvocation {resType}
  source super.source.^MethodDeclaration.1.MethodHeader {resType}
  context c1 : same RootNode

}

// Line 1399
one_node relation RArrayAccess_ExprType_0
{
  comment "Тип выражения ссылки на массив должен быть ArrayType"
  arbitrary
```

```
  present
  target ExprNameArrayAccess.1.ExpressionName {type [is ArrayType]}
  context c1 : same RootNode

}

// Line 1400
one_node relation RArrayAccess_PrimaryType_0
{
  comment "Тип выражения ссылки на массив должен быть ArrayType"
  arbitrary
  present
  target PrimarynoNewArrayAccess.1.PrimaryNoNewArray {type [is ArrayType]}
  context c1 : same RootNode

}

// Line 1401
one_to_one relation RArrayAccessType_Expr_0
{
  comment "Тип выражения ArrayAccess это тип элементов массива."
  arbitrary
  equal
  target ExprNameArrayAccess {type}
  source this.target.1.ExpressionName.1.ArrayType {type}
  context c1 : same RootNode

}

// Line 1402
one_to_one relation RArrayAccessType_Primary_0
{
  comment "Тип выражения ArrayAccess это тип элементов массива."
  arbitrary
  equal
  target PrimarynoNewArrayAccess {type}
  source this.target.1.PrimaryNoNewArray.1.ArrayType {type}
  context c1 : same RootNode

}

// Line 1403
one_node relation RPostIncrement_Numeric_0
{
  comment "Результат postfix выражения должен быть convertible к numeric типу"
  arbitrary
  present
  target PostIncrementExpression.1.PostfixExpression {type [is NumericType]}
  context c1 : same RootNode

}
```

```
// Line 1404
one_to_one relation RPostIncrement_Type_0
{
  comment "Тип PostIncrementExpression совпадает с типом postfix выражения."
  arbitrary
  equal
  target PostIncrementExpression {type}
  source this.target.1.PostfixExpression {type}
  context c1 : same RootNode

}

// Line 1405
one_node relation RPostDecrement_Numeric_0
{
  comment "Результат postfix выражения должен быть convertible к numeric типу"
  arbitrary
  present
  target PostDecrementExpression.1.PostfixExpression {type [is NumericType]}
  context c1 : same RootNode

}

// Line 1406
one_to_one relation RPostDecrement_Type_0
{
  comment "Тип PostDecrementExpression совпадает с типом postfix выражения."
  arbitrary
  equal
  target PostDecrementExpression {type}
  source this.target.1.PostfixExpression {type}
  context c1 : same RootNode

}

// Line 1408
one_node relation RPreIncrement_Numeric_0
{
  comment "Результат unary выражения должен быть convertible к numeric типу"
  arbitrary
  present
  target PreIncrementExpression.1.UnaryExpression {type [is NumericType]}
  context c1 : same RootNode

}

// Line 1409
one_to_one relation RPreIncrement_Type_0
{
  comment "Тип PreIncrementExpression совпадает с типом unary выражения."
  arbitrary
  equal
```

```
  target PreIncrementExpression {type}
  source this.target.1.UnaryExpression {type}
  context c1 : same RootNode

}

// Line 1411
one_node relation RPreDecrement_Numeric_0
{
  comment "Результат unary выражения должен быть convertible к numeric типу"
  arbitrary
  present
  target PreDecrementExpression.1.UnaryExpression {type [is NumericType]}
  context c1 : same RootNode

}

// Line 1412
one_to_one relation RPreDecrement_Type_0
{
  comment "Тип PreDecrementExpression совпадает с типом unary выражения."
  arbitrary
  equal
  target PreDecrementExpression {type}
  source this.target.1.UnaryExpression {type}
  context c1 : same RootNode

}

// Line 1414
one_node relation RUnaryPlus_Numeric_0
{
  comment "Результат unary выражения должен быть convertible к numeric типу"
  arbitrary
  present
  target UnaryPlusExpression.1.UnaryExpression {type [is NumericType]}
  context c1 : same RootNode

}

// Line 1415
one_to_one relation RUnaryPlus_Type_0
{
  comment "Тип UnaryPlusExpression совпадает с типом unary выражения."
  arbitrary
  equal
  target UnaryPlusExpression {type}
  source this.target.1.UnaryExpression {type}
  context c1 : same RootNode

}
```

```
// Line 1417
one_node relation RUnaryMinus_Numeric_0
{
  comment "Результат unary выражения должен быть convertible к numeric типу"
  arbitrary
  present
  target UnaryMinusExpression.1.UnaryExpression {type [is NumericType]}
  context c1 : same RootNode

}

// Line 1418
one_to_one relation RUnaryMinus_Type_0
{
  comment "Тип UnaryMinusExpression совпадает с типом unary выражения."
  arbitrary
  equal
  target UnaryMinusExpression {type}
  source this.target.1.UnaryExpression {type}
  context c1 : same RootNode

}

// Line 1419
one_node relation RBitwiseComplement_Integral_0
{
  comment "Результат unary выражения должен быть convertible к integral типу"
  arbitrary
  present
  target BitwiseComplementExpression.1.UnaryExpression {type [is IntegralType]}
  context c1 : same RootNode

}

// Line 1420
one_to_one relation RBitwiseComplement_Type_0
{
  comment "Тип BitwiseComplementExpression совпадает с типом unary выражения."
  arbitrary
  equal
  target BitwiseComplementExpression {type}
  source this.target.1.UnaryExpression {type}
  context c1 : same RootNode

}

// Line 1421
one_node relation RLogicalComplement_Boolean_0
{
  comment "Результат unary выражения должен иметь тип boolean"
  arbitrary
  present
```

```
  target LogicalComplementExpression.1.UnaryExpression {type [is BooleanType]}
  context c1 : same RootNode

}

// Line 1422
one_to_one relation RLogicalComplement_Type_0
{
  comment "Тип LogicalComplementExpression совпадает с типом unary выражения (т.е.
boolean)."
  arbitrary
  equal
  target LogicalComplementExpression {type}
  source this.target.1.UnaryExpression {type}
  context c1 : same RootNode

}

// Line 1423
one_node relation RPrimitiveCast_Type_0
{
  comment "Тип cast выражения совпадает с типом указанным в скобках."
  arbitrary
  equal
  target PrimitiveTypeCastExpression {type}
  source PrimitiveTypeCastExpression {castType}
  context c1 : same RootNode

}

// Line 1424
one_node relation RReferenceCast_Type_0
{
  comment "Тип cast выражения совпадает с типом указанным в скобках."
  arbitrary
  equal
  target ReferenceTypeCastExpression {type}
  source ReferenceTypeCastExpression {castType}
  context c1 : same RootNode

}

// Line 1425
one_node relation RPrimitiveCast_TypeMatching_0
{
  comment "Тип primitive cast выражения всегда primitive."
  arbitrary
  present
  target PrimitiveTypeCastExpression {type[is PrimitiveType]}
  context c1 : same RootNode

}
```

```
// Line 1426
one_node relation RReferenceCast_TypeMatching_0
{
  comment "Тип reference cast выражения всегда reference."
  arbitrary
  present
  target ReferenceTypeCastExpression {type[is ReferenceType]}
  context c1 : same RootNode

}

// Line 1428
one_to_one relation RPrimitiveCast_Conversion_0
{
  comment "Должна существовать возможность преобразования типов по правилам casting
conversion."
  arbitrary
  compatible
  target PrimitiveTypeCastExpression.1.UnaryExpression {type}
  source this.target.^PrimitiveTypeCastExpression {type}
  context c1 : same RootNode

}

// Line 1429
one_to_one relation RReferenceCast_Conversion_0
{
  comment "Должна существовать возможность преобразования типов по правилам casting
conversion."
  arbitrary
  compatible
  target ReferenceTypeCastExpression.1.UnaryExpressionNotPlusMinus {type}
  source this.target.^ReferenceTypeCastExpression {type}
  context c1 : same RootNode

}

// Line 1430
one_node relation RMultExprLeft_Numeric_0
{
  comment "Тип обоих операндов должен приводиться к Numeric типу"
  arbitrary
  present
  target StrictMultiplicativeExpression.left {type [is NumericType]}
  context c1 : same RootNode

}

// Line 1431
one_node relation RMultExprRight_Numeric_0
{
```

```
  comment "Тип обоих операндов должен приводиться к Numeric типу"
  arbitrary
  present
  target StrictMultiplicativeExpression.right {type [is NumericType]}
  context c1 : same RootNode

}

// Line 1432
one_node relation RMultExpr_PromType_0
{
  comment "Над типами аргументов выполняется numeric promotion."
  arbitrary
  compatible
  target StrictMultiplicativeExpression {left.type}
  source StrictMultiplicativeExpression {right.type}
  context c1 : same RootNode

}

// Line 1433
one_to_one relation RMultExpr_Type_0
{
  comment "Тип выражения совпадает с типом полученным после numeric promotion."
  arbitrary
  equal
  target StrictMultiplicativeExpression {type}
  source this.target.left {type}
  context c1 : same RootNode

}

// Line 1434
one_node relation RStrConcAddExpr_OperandType_0
{
  comment "Если один из операндов имееет тип String то производится операция
конкатенации."
  arbitrary
  present
  target StringConcatenationAdditiveExpression.1.Expression {type [is JavaLangStringType]}
  context c1 : same RootNode

}

// Line 1435
one_node relation RAddExprLeft_Numeric_0
{
  comment "Тип левого операнда должен приводиться к Numeric типу"
  arbitrary
  present
  target NumericAdditiveExpression.left {type [is NumericType]}
  context c1 : same RootNode
```

```
}

// Line 1436
one_node relation RAddExprRight_Numeric_0
{
  comment "Тип правого операнда должен приводиться к Numeric типу"
  arbitrary
  present
  target NumericAdditiveExpression.right {type [is NumericType]}
  context c1 : same RootNode

}

// Line 1437
one_node relation RStrConcAddExpr_Type_0
{
  comment "Результат конкатенации имеет тип String."
  arbitrary
  present
  target StringConcatenationAdditiveExpression {type [is JavaLangStringType]}
  context c1 : same RootNode

}

// Line 1438
one_node relation RAddExpr_PromType_0
{
  comment "Над типами аргументов выполняется numeric promotion."
  arbitrary
  compatible
  target NumericAdditiveExpression {left.type}
  source NumericAdditiveExpression {right.type}
  context c1 : same RootNode

}

// Line 1439
one_to_one relation RAddExpr_Type_0
{
  comment "Тип выражения совпадает с типом полученным после numeric promotion."
  arbitrary
  equal
  target NumericAdditiveExpression {type}
  source this.target.left {type}
  context c1 : same RootNode

}

// Line 1440
one_node relation RShiftExprLeft_Integral_0
{
```

```
 comment "Тип левого операнда должен приводиться к Integral типу"
 arbitrary
 present
 target StrictShiftExpression.left {type [is IntegralType]}
 context c1 : same RootNode

}

// Line 1441
one_node relation RShiftExprRight_Integral_0
{
 comment "Тип правого операнда должен приводиться к Integral типу"
 arbitrary
 present
 target StrictShiftExpression.right {type [is IntegralType]}
 context c1 : same RootNode

}

// Line 1443
one_to_one relation RShiftExpr_Type_0
{
 comment "Тип выражения совпадает с типом левого операнда после numeric promotion."
 arbitrary
 equal
 target StrictShiftExpression {type}
 source this.target.left {type}
 context c1 : same RootNode

}

// Line 1444
one_node relation RRelationalStrictExpr_Type_0
{
 comment "Тип StrictRelationalExpression выражения всегда boolean."
 arbitrary
 present
 target StrictRelationalExpression {type [is BooleanType]}
 context c1 : same RootNode

}

// Line 1445
one_node relation RRelationalInstExpr_Type_0
{
 comment "Тип InstanceOfRelationalExpression выражения всегда boolean."
 arbitrary
 present
 target InstanceOfRelationalExpression {type [is BooleanType]}
 context c1 : same RootNode

}
```

```
// Line 1446
one_node relation RRelationalExprLeft_Numeric_0
{
  comment "Тип левого операнда должен приводиться к Numeric типу"
  arbitrary
  present
  target StrictRelationalExpression.left {type [is NumericType]}
  context c1 : same RootNode

}

// Line 1447
one_node relation RRelationalExprRight_Numeric_0
{
  comment "Тип правого операнда должен приводиться к Numeric типу"
  arbitrary
  present
  target StrictRelationalExpression.right {type [is NumericType]}
  context c1 : same RootNode

}

// Line 1448
one_node relation RRelationalInstExpr_Ref_0
{
  comment "Тип выражения должен быть reference или null типа."
  arbitrary
  present
  target InstanceOfRelationalExpression.expression {type [is ReferenceType]}
  context c1 : same RootNode

}

// Line 1449
one_node relation RRelationalInstType_Ref_0
{
  comment "Тип указанный в выражении должен быть reference."
  arbitrary
  present
  target InstanceOfRelationalExpression {refType [is ReferenceType]}
  context c1 : same RootNode

}

// Line 1451
one_node relation RRelationalInstCompType_0
{
  comment "Если соответствующее выражение приведения типов не верно, то и проверка
на этот тип тоже неверна."
  arbitrary
  compatible
```

```
  target InstanceOfRelationalExpression {expression.type}
  source this.target {refType}
  context c1 : same RootNode

}

// Line 1452
one_node relation REqualityExpr_Type_0
{
  comment "Тип equality выражения всегда boolean."
  arbitrary
  present
  target StrictEqualityExpression {type [is BooleanType]}
  context c1 : same RootNode

}

// Line 1454
one_node relation REqualityNumericType_0
{
  comment "У numeric equality выражения типы операндов должны быть numeric."
  arbitrary
  present
  target NumericEqualityExpression {left.type[is NumericType], right.type[is NumericType]}
  context c1 : same RootNode

}

// Line 1455
one_node relation REqualityBooleanType_0
{
  comment "У boolean equality выражения типы операндов должны быть boolean."
  arbitrary
  present
  target BooleanEqualityExpression {left.type[is BooleanType], right.type[is BooleanType]}
  context c1 : same RootNode

}

// Line 1456
one_node relation REqualityReferenceType_0
{
  comment "У reference equality выражения типы операндов должны быть reference."
  arbitrary
  present
  target ReferenceEqualityExpression {left.type[is ReferenceType], right.type[is
ReferenceType]}
  context c1 : same RootNode

}

// Line 1457
```

```
one_node relation REqualityReferenceComp_0
{
  comment "Один из операндов должен приводиться к другому"
  arbitrary
  compatible
  target ReferenceEqualityExpression {left.type}
  source this.target {right.type}
  context c1 : same RootNode

}

// Line 1458
one_node relation RBitwiseNumericType_0
{
  comment "У bitwise выражения типы операндов должны быть numeric."
  arbitrary
  present
  target IntegerBitwiseAndOrExpression {left.type[is NumericType], right.type[is
NumericType]}
  context c1 : same RootNode

}

// Line 1459
one_node relation RLogicalBooleanType_0
{
  comment "У logical выражения типы операндов должны быть boolean."
  arbitrary
  present
  target BooleanLogicalAndOrExpression {left.type[is BooleanType], right.type[is
BooleanType]}
  context c1 : same RootNode

}

// Line 1460
one_node relation RBitwiseExpr_PromType_0
{
  comment "Над типами аргументов bitwise выражения выполняется numeric promotion."
  arbitrary
  compatible
  target IntegerBitwiseAndOrExpression {left.type}
  source IntegerBitwiseAndOrExpression {right.type}
  context c1 : same RootNode

}

// Line 1461
one_to_one relation RBitwiseExpr_Type_0
{
  comment "Тип выражения совпадает с типом полученным после numeric promotion."
  arbitrary
```

```
  equal
  target IntegerBitwiseAndOrExpression {type}
  source this.target.left {type}
  context c1 : same RootNode

}

// Line 1462
one_node relation RLogicalExpr_Type_0
{
  comment "Тип logical выражения всегда boolean."
  arbitrary
  present
  target BooleanLogicalAndOrExpression {type [is BooleanType]}
  context c1 : same RootNode

}

// Line 1463
one_node relation RConditionalAOBooleanType_0
{
  comment "У conditional and/or выражения типы операндов должны быть boolean."
  arbitrary
  present
  target StrictConditionalAndOrExpression {left.type[is BooleanType], right.type[is
BooleanType]}
  context c1 : same RootNode

}

// Line 1464
one_node relation RConditionalAOExpr_Type_0
{
  comment "Тип conditional and/or выражения всегда boolean."
  arbitrary
  present
  target StrictConditionalAndOrExpression {type [is BooleanType]}
  context c1 : same RootNode

}

// Line 1466
one_node relation RConditionalQBooleanType_0
{
  comment "Тип первого выражения должен быть boolean."
  arbitrary
  present
  target StrictConditionalExpression {condition.type[is BooleanType]}
  context c1 : same RootNode

}
```

```
// Line 1468
one_to_one relation RConditionalQExpr_ThenPromType_0
{
  comment "Тип второго выражения приводим к типу conditional expression."
  arbitrary
  compatible
  target StrictConditionalExpression {type}
  source this.target.thenExpr {type}
  context c1 : same RootNode

}

// Line 1469
one_to_one relation RConditionalQExpr_ElsePromType_0
{
  comment "Тип третьего выражения приводим к типу conditional expression."
  arbitrary
  compatible
  target StrictConditionalExpression {type}
  source this.target.elseExpr {type}
  context c1 : same RootNode

}

// Line 1471
one_node relation RAssignmentType_0
{
  comment "Тип выражения Assignment совпадает с типом левой части"
  arbitrary
  present
  target Assignment {type}
  source Assignment {left.type}
  context c1 : same RootNode

}

// Line 1473
one_node relation RAssignmentCompatible_0
{
  comment "Тип правой части должен приводится к типу левой части"
  arbitrary
  compatible
  target Assignment {right.type}
  source Assignment {left.type}
  context c1 : same Block

}

// Line 1474
one_to_one relation RExprLeftHSType_0
{
```

```
  comment "Если левая часть это ExprNameLeftHandSide, то тип левой части совпадает с
типом ExpressionName."
  arbitrary
  equal
  target ExprNameLeftHandSide {type}
  source this.target.1.ExpressionName {type}
  context c1 : same RootNode

}

// Line 1475
one_to_one relation RFieldLeftHSType_0
{
  comment "Если левая часть это FieldAccessLeftHandSide, то тип левой части совпадает с
типом FieldAccess."
  arbitrary
  equal
  target FieldAccessLeftHandSide {type}
  source this.target.1.FieldAccess {type}
  context c1 : same RootNode

}

// Line 1476
one_to_one relation RArrayLeftHSType_0
{
  comment "Если левая часть это ArrayAccessLeftHandSide, то тип левой части совпадает с
типом ArrayAccess."
  arbitrary
  equal
  target ArrayAccessLeftHandSide {type}
  source this.target.1.ArrayAccess {type}
  context c1 : same RootNode

}

// Line 1479
one_to_many relation RDefiniteAssignment_0
{
  comment "Локальная переменная должна быть definitely assigned перед использованием"
  semantic
  equal
  target ExpressionName[^LeftHandSide = null] {ident}
  source ExprNameLeftHandSide.1.ExpressionName {ident}
  context c1 : differ source_context {this.target_context.^UsualMethodBody.00.Statement}
target_context {Statement}

}

// Line 1480
/*
many_to_many relation RDefiniteAssignment_Unequal_0
```

```
{
 comment "При объявлении переменная не может быть проинициализирована сама собой"
 arbitrary
 unequal
 target LeftHandSide.1.ExpressionName {ident}
 source ExpressionName {ident}
 context c1 : same Assignment

}
*/

// Line 1481
many_to_many relation RDefiniteAssignment_UnequalRecur_0
{
 comment "При объявлении переменная не может быть проинициализирована сама собой"
 arbitrary
 unequal
 target LeftHandSide.1.ExpressionName {ident}
 source ExpressionName {ident}
 context c1 : differ source_context {this.target_context.(1.Assignment)*} target_context
{Assignment}

}

type_set PrimitiveTypes {
   const IntegralType.BYTE, const IntegralType.SHORT,  {const IntegralType.CHAR, const
IntegralType.INT}, const IntegralType.LONG,  const FloatingPointType.FLOAT, const
FloatingPointType.DOUBLE
}
type_set ObjectTypes {
   ref NullType, ref ClassOrInterfaceType[notin
R229_TD_0[*].target.^ClassOrInterfaceType].(00.TypeName.R229_TD_0[target=prev].source.
1.Super.1.ClassOrInterfaceType)*
}
type_set BooleanPrimitiveTypes {
   const BooleanType.BOOL
}
type_set StringReferenceType {
   const JavaLangStringType.JAVA_STRING
}
type_set ClassReferenceType {
   const JavaLangClassType.JAVA_CLASS
}
node ThisAccessExpression
{
	ssconstraint
	{
		forbidden
		ancestor StaticInitializer
		vertical
	}
}
```

```
node ReturnStatement
{
	ssconstraint
	{
		forbidden
		ancestor StaticBlock
		vertical
	}
}

node CompoundReturnStatement
{
	ssconstraint
	{
		forbidden
		ancestor ConstructorDeclaration
		vertical
	}
}

node AbstractMethodDeclaration
{
	ssconstraint
	{
		forbidden
		ancestor ClassDeclaration
		vertical
	}
}

node ConstantDeclaration
{
	ssconstraint
	{
		forbidden
		ancestor ClassDeclaration
		vertical
	}
}

node FieldDeclaration
{
	ssconstraint
	{
		forbidden
		ancestor InterfaceDeclaration
		vertical
	}
}
```

```
node MethodDeclaration
{
        ssconstraint
        {
                forbidden
                ancestor InterfaceDeclaration
                vertical
        }
}

node FinalModifier
{
        ssconstraint
        {
                forbidden
                ancestor InterfaceDeclaration
                vertical
        }
}
node Literal
{
   ssconstraint
   {
      forbidden
      ancestor PrimaryFieldAccess
      irrelevant (Assignment)
      vertical
   }
}
node ConstructorDeclaration
{
   ssconstraint
   {
      forbidden
      ancestor ClassInstanceCreationExpression
      relevant (ClassBody)
      vertical
      depth 2
   }
}

node SimpleVariableDeclarator
{
   ssconstraint
   {
      forbidden
      ancestor InterfaceDeclaration
      vertical
         depth 4
   }
}
```

```
node ThisAccessExpression
{
   ssconstraint
   {
      forbidden
      ancestor InterfaceDeclaration
      vertical
   }
}

node SuperFieldAccess
{
   ssconstraint
   {
      forbidden
      ancestor InterfaceDeclaration
      vertical
   }
}

node SuperStaticFieldAccess
{
   ssconstraint
   {
      forbidden
      ancestor InterfaceDeclaration
      vertical
   }
}

node SuperMethodInvocation
{
   ssconstraint
   {
      forbidden
      ancestor InterfaceDeclaration
      vertical
   }
}

node ClassSuperMethodInvocation
{
   ssconstraint
   {
      forbidden
      ancestor InterfaceDeclaration
      vertical
   }
}
```