УДК 004.415.53

ГЕНЕРАЦИЯ ТЕСТОВ ДЛЯ СЕМАНТИЧЕСКИХ АНАЛИЗАТОРОВ

М. В. Архипова[1]

В статье исследуется проблема построения программ-тестов для модулей проверки статической семантики в компиляторах, рассматриваются существующие способы формального описания правил статической семантики языков программирования и обсуждаются причины, мешающие использованию существующих нотаций для решения задачи генерации семантически корректных программ-тестов. Кроме того, предлагается собственный подход к описанию правил статической семантики языков программирования, удобный для построения семантически управляемого генератора программ-тестов.

Ключевые слова: статическая семантика, семантический анализатор, автоматическая генерация тестов.

1. Введение. Практика и теория разработки компиляторов являются вполне зрелыми дисциплинами программной инженерии. Они развиваются уже несколько десятилетий, поэтому имеются широко известные теоретические результаты, практические методики и инструменты автоматизации разработки компиляторов. Вместе с тем, для современных языков программирования и для компиляторов, которые используются в средах разработки программ, нет готовых решений, которые позволяют автоматически построить компилятор, отвечающий всей совокупности требований. Примерами "неудобных" конструкций языка являются, например, средства объектно-ориентированного программирования (наследование, полиморфизм); примерами "неудобных" требований окружения являются диагностика и компенсация ошибок. Реализация этих требований становится особенно сложной, когда речь идет об оптимизирующих или распараллеливающих компиляторах, о расширяемых и специализированных языках, о многоязыковых системах.

Такого рода сложности приводят к тому, что даже при использовании средств автоматической генерации отдельных компонентов компиляторов, например парсеров и семантических анализаторов, результатом генерации является не готовая программа, а некоторый полуфабрикат, который требует ручной доводки. В процессе таких доводок нередко в реализацию вносятся ошибки. Большое число ошибок, как правило, встречается в блоках оптимизации и генерации кода. Кроме того, встречаются и системные ошибки, обусловленные сложностью интерфейсов между отдельными блоками компилятора.

Эти соображения объясняют, почему, несмотря на серьезные продвижения как собственно методов разработки компиляторов, так и методов верификации компиляторов, по-прежнему актуальной является задача тестирования компиляторов, в частности, задача автоматической генерации тестов, чему и посвящена наша работа.

Общеизвестны два подхода к тестированию: метод "белого ящика" и метод "черного ящика". В первом случае разработчик тестов нацеливается на то, чтобы выявить все ошибочные фрагменты реализации. Для этого он изучает эту реализацию и строит тесты, которые показывают, что тот или иной фрагмент дает неверный результат (тест нашел ошибку) или что фрагмент на представленных данных дал корректный результат (тест не нашел ошибку). Во втором случае тестировщик изучает спецификацию программы и старается испытать ее на разнообразных входных данных, которые в совокупности должны охватить если не все, то по крайней мере основные классы входных данных.

У каждого из этих двух подходов есть плюсы и минусы. Плюс метода "белого ящика" состоит в том, что основным входом для построения тестов является хорошо структурированный, даже формальный набор данных — реализация компилятора на некотором языке программирования. Это позволяет упростить задачу генерации тестов и анализ полноты тестирования, по крайней мере в терминах процента проверенных (покрытых) строк или операторов реализации. Плюс метода "черного ящика" состоит в том, что тесты нацелены непосредственно на проверку требований к компилятору, а это упрощает анализ правильности его работы (критерии правильности задаются описанием языка), тогда как при проверке правильности отдельных блоков реализации никаких готовых, предопределенных критериев правильности нет, поэтому, в частности, автоматизировать такую проверку далеко не всегда удается.

[1] Институт системного программирования РАН, Б. Коммунистическая, 25, 109004, Москва; e-mail: maryn@ispras.ru

В качестве компромисса между методами белого и черного ящика можно рассматривать тестирование на основе моделей. Модель задает требования к реализации (поэтому ее можно использовать для анализа корректности реализации) и может быть формально описана (на некотором языке спецификаций или при помощи другой машиночитаемой нотации), поэтому так же, как и программа, она может быть использована в качестве исходных данных для генерации тестов и для оценки тестового покрытия.

В общем случае нет ответа, какие тесты лучше: построенные на основе реализации и спецификации или на основе моделей и каких именно. В практическом плане важно, что модель позволяет строить тесты систематически и автоматически. Это, в свою очередь, означает, что можно ожидать повышения тщательности проверки при увеличении количества сгенерированных тестов (конечно, при некоторых разумных предположениях о схеме генерации). Для оценки качества полученных тестов необходимо разрабатывать критерии, требованиям которых должны удовлетворять хорошие тестовые наборы.

Модели для сложных программных систем также являются достаточно сложными системами. Поэтому на первый план выходят проблемы не только возможности компьютерной обработки моделей, но и проблемы построения моделей. В частности, это означает, что модели должны строиться на концепциях, хорошо понятных программистам и тестировщикам.

В контексте разработки тестов для компиляторов это означает, что модель по форме должна быть близка к определению грамматики языка. Использование описания языка в качестве основных данных для генерации тестов, во-первых, даст четкую привязку тестов к функциональным требованиям тестируемого компилятора (а не к особенностям реализации); во-вторых, позволит легко вносить изменения в тестовый набор при работе с диалектами и "расширенными подмножествами" языков, с которыми приходится иметь дело на практике. Кроме того, существуют дополнительные требования, обусловленные спецификой процессов, в которых эти тесты могут использоваться. В частности, рассматриваются требования разработчиков компиляторов, для которых тесты — это в первую очередь средство отладки. В этом случае конкретного разработчика обычно интересует не функционирование всего компилятора в целом, а корректность работы отдельных подсистем, реализующих те или иные возможности. В таком контексте важно уметь строить "целенаправленные" тесты, которые, с одной стороны, проверяют компилятор в целом, а с другой стороны, концентрируют проверки на тех свойствах работы компилятора (на тех подсистемах), которые сопряжены с фокусом интересов пользователя тестов.

По времени выполнения проверки семантических правил выделяют *статическую* и *динамическую* семантики языков программирования. В то время как правила статической семантики проверяются на этапе компиляции, правила динамической семантики могут быть проверены только на этапе выполнения программы. Поэтому вопросы, касающиеся динамической семантики, выходят за рамки обсуждаемой проблемы и в данной работе не рассматриваются.

Статическая семантика традиционно формулируется в форме ограничений (контекстных условий), что облегчает проверку этих условий при семантическом анализе, но затрудняет генерацию подходящих программ-тестов.

Поскольку функцией семантического анализатора компилятора является проверка выполнения правил статической семантики, то для тестирования семантического анализатора в компиляторе в качестве тестовых (исходных) данных нужны синтаксически корректные программы, в которых выполняются контекстные ограничения, описанные в спецификации исходного языка (*позитивные тесты*), и синтаксически корректные программы, в которых нарушены определенные контекстные ограничения (*негативные тесты*).

Позитивные тесты позволяют проверить правильность реализации контекстных условий в семантическом анализаторе. Негативные тесты могут использоваться для проверки правильности сообщений об ошибках, выдаваемых анализатором, и для проверки способности компилятора обнаруживать ошибочные места в программах.

В принципе для "целеуказания" (описания нужных тестов) естественно использовать описание статической семантики языка. Прямолинейное использование описания статической семантики в виде контекстных ограничений позволяет только строить фильтры, которые отбрасывают неподходящие программы из множества синтаксически корректных программ, которые генерировать относительно просто. Получается, что тесты строятся методом проб и ошибок, причем практика показывает, что КПД таких методов крайне низок. Повысить эффективность генерации тестов для семантических анализаторов можно было бы, если бы входные данные генератора позволяли непосредственно конструировать подходящие тесты. Таким образом, появляется потребность в таких описаниях статической семантики, которые облегчают целенаправленную генерацию, т.е. обеспечивают минимум непроизводительных затрат. Описание семантики такого рода в дальнейшем будем называть *конструктивным*.

В настоящей работе описывается метод генерации позитивных и негативных тестов для семантических анализаторов. В этом методе используется конструктивное описание статической семантики, которое заменяет и дополняет традиционные контекстные условия.

Статья состоит из введения, заключения и шести основных разделов. В разделе 2 описывается текущее состояние исследований, связанных с тестированием компиляторов и семантических анализаторов. В разделе 3 подробно рассматривается пример, демонстрирующий использование классических атрибутных грамматик (далее АГ) для решения задачи генерации тестов для семантического анализатора компилятора модельного языка. Данный пример показывает, что использование АГ для задачи генерации тестов для семантических анализаторов не эффективно. Раздел 4 посвящен описанию предлагаемого конструктивного метода формального задания правил статической семантики. В разделе 5 формулируются критерии покрытия, которым должны удовлетворять наборы позитивных и негативных тестов для семантических анализаторов. В разделе 6 описывается алгоритм генерации тестов на основе предлагаемого конструктивного описания. В разделе 7 приводятся результаты практического применения предложенного метода.

2. Обзор существующих подходов к задаче генерации тестов для компиляторов. Начиная с 60-х годов XX-века были опубликованы результаты многих исследований, посвященных вопросам генерации тестов для компиляторов [1 – 9, 23]. Большинство из предложенных способов не позволяет генерировать тесты с учетом контекстных ограничений целевого языка программирования, а те немногие, что позволяют, обладают существенными недостатками.

Рассмотрим основные существующие подходы.

В ранних исследованиях, посвященных генерации тестов для компиляторов, К. Ханфорд [1] и П. Пардом [2] представили методы, позволяющие генерировать синтаксически корректные, без учета правил статической семантики, программы для компиляторов процедурных языков.

Так, в 1970 году была опубликована работа К. Ханфорда [1], 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.

Работа П. Пардома [2] стала фундаментальной в области генерации тестов для компиляторов и послужила отправной точкой для дальнейших исследований.

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

В 1976 году была опубликована работа Б. Уичмана и Б. Джонса [3], в которой авторы обсуждали необходимость построения тестовых программ для компиляторов, позволяющих 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 году А. Челентано с соавторами в работе [4] предложили систему, частично автоматизирующую фазу тестирования при разработке компилятора. При этом синтаксически корректные программы генерировались по алгоритму П. Пардома. Грамматика целевого языка, подаваемая на вход генератору, дополнялась кодом, в котором производилась проверка семантических ограничений и преобразование синтаксически корректных программ в семантически корректные. 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 чрезмерных усилий.

В работе [5] А. Данкен и Дж. Хатчисон предложили использовать АГ 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 проводить 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 [6], авторами которого являются Е. Сирер и Б. Бершед, описывается спецификационный язык lava. Грамматика, описанная на lava, напоминает EBNF-грамматику, дополненную Java-кодом

для описания контекстных ограничений. Подход применялся авторами для генерации небольшого числа тестов ($\sim$ 6 тестов), имеющих большой размер ($\sim$ 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 [7 – 9], выполненные А. С. Косачевым, М. А. Посыпкиным и другими, посвящены генерации тестов для компиляторов на основе ASM-спецификации [24]. В работах рассматривается 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 неутешительными: генерация тестов для mpc-компилятора на втором шаге алгоритм заняла 63 часа, приблизительная оценка времени работы алгоритма на третьем шаге — один год [7]. Но позднее авторами были предложены оптимизации, позволившие генерировать тесты за приемлемое время [8, 9].

Причиной, по которой вплоть до настоящего времени не найден удобный с практической точки зрения метод генерации семантически корректных программ, видится отсутствие общепринятого, простого и естественного способа формального описания семантики языков программирования, удобного для использования генератором тестов.

Существуют различные способы формального описания семантики языков программирования. Среди них наиболее известными являются W-грамматики [10], аксиоматическое описание семантики [11], венский метод [12 – 15] и АГ [16, 17].

Формализм АГ оказался удобным средством для описания статической семантики языков программирования. Вместе с тем выяснилось, что реализация алгоритмов вычисления значений атрибутов для АГ общего вида сопряжена с большими трудностями. В связи с этим рассматривались различные классы АГ, обладающих “хорошими” свойствами. К числу таких свойств относятся, прежде всего, простота алгоритма проверки АГ на зацикливание и простота алгоритма вычисления атрибутов для АГ данного класса.

Было создано несколько систем для автоматизации разработки трансляторов и их расширений, основанных на формализме атрибутных грамматик: Yacc и Lex [18], LIGA [19], Ox [20]. Опыт их использования показал, что атрибутный формализм может быть применен для описания семантики языка при создании семантического анализатора компилятора.

Если бы удалось на основе АГ автоматически генерировать тесты для семантических анализаторов, это было бы хорошим результатом, так как, в принципе, разработка АГ хотя и не простая, но зато хорошо изученная задача. Однако как следует из ряда проведенных исследований, например [5], прямолинейное использование атрибутных грамматик для задачи генерации тестовых программ дает возможность только вести анализ выполненных контекстных условий. Это позволяет автоматизировать фильтрацию (отбрасывание семантически некорректных программ), выполняемую после массовой генерации синтаксически правильных программ. Вместе с тем, хотя и генерацию, и фильтрацию можно автоматизировать, в этой схеме имеется несколько недостатков. Во-первых, по сути, требуется разработка эталонного семантического анализатора (сложность АГ для современных языков программирования, особенно для объектно-ориентированных (далее ОО) языков, достаточно высока). Во-вторых, данный подход не позволяет нацелить генерацию на отдельные группы конструкций. В-третьих, традиционно АГ выглядят не очень прозрачно: их структура не обладает свойством прослеживаемости (traceability) ограничений статической семантики языка в тексте АГ, а затем и в тестах, которые генерируются на основе такой АГ.

На практике сложность АГ, в частности, выражается во введении в описание грамматики вспомогательных структур данных и функций на одном из языков программирования и большого числа технических операций для передачи контекста между “сферами ответственности” нетерминалов, соседствующих в дереве разбора. Так как сложность этих вспомогательных функций часто бывает достаточно высокой, автоматический анализ этих функций в общем случае невозможен. Заметим, что в современных ОО языках сложность таких функций существенно выше по сравнению с языками типа C и Pascal. Этот факт и проблемы, упомянутые во втором и третьем пунктах выше, приводят к тому, что в терминах структуры АГ (рассматривается полное описание, включающее и вспомогательные функции) не удается сформулировать критерии тестового покрытия, без чего, в свою очередь, автоматическая генерация тестов лишается формальных критериев качества и параметров, направляющих процесс генерации.

После введения Кнутом термина АГ другими исследователями рассматривались различные классы

атрибутных грамматик, обладающих “хорошими” свойствами. Однако, с точки зрения применения АГ к решению задачи генерации тестов, любые атрибутные грамматики ведут себя одинаково. В связи с этим в данной статье используется термин *классические атрибутные грамматики* для обозначения формализма, введенного Д. Кнутом в работе [16].

Несмотря на отмеченные недостатки, в технике классических атрибутных грамматик много полезного, что можно было бы использовать для генерации тестов для анализаторов семантики, в первую очередь мы имеем в виду хорошо проработанную теорию и известность данного подхода в среде разработчиков компиляторов. Поэтому, перед тем как предложить новый способ описания статической семантики, приспособленный для генерации тестов, мы предлагаем рассмотреть пример описания статической семантики в форме классической атрибутной грамматики, проанализировать, в чем, собственно, трудности генерации, а затем сделать улучшения для организации целенаправленной генерации тестов.

3. Генерация с использованием классических атрибутных грамматик. Рассмотрим метод построения позитивных тестов для семантического анализатора компилятора, в основе которого лежит алгоритм генерации, использующий принципы последовательного раскрытия продукционных правил грамматики исходного языка с фильтрацией по контекстным условиям на раннем этапе. Подобный алгоритм применялся в работе [5].

На вход алгоритму подается формальное описание грамматики исходного языка в виде EBNF и формальное описание контекстных ограничений в виде атрибутных грамматик. 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.

Для описания грамматики исходного языка используется форма записи EBNF, а для описания контекстных ограничений исходного языка в стиле атрибутных грамматик — форма записи, использованная в работе Ф. Пагана [21] и К. Слоннегера [22]. Приведем основные обозначения этой формы записи:

Nonterm.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 семантические проверки и вычисления более сложные, чем присваивание; параметрами функции могут быть атрибуты, относящиеся к нетерминалам;
=	обозначает операцию проверки эквивалентности значений;
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 может быть описана только один раз. Синоним может быть объявлен только для уже объявленной переменной или синонима.

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

program ::= “program” identifier “;” declarations

(1) program.symtab := emptysymtab()

declarations ::= (declaration)+

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>

Неформальное описание семантических ограничений приводится в табл. 1. Строки, отмеченные в

таблице восклицательными знаками, соответствуют собственно требованиям контекстных условий.

Таблица 1

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.
!	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.

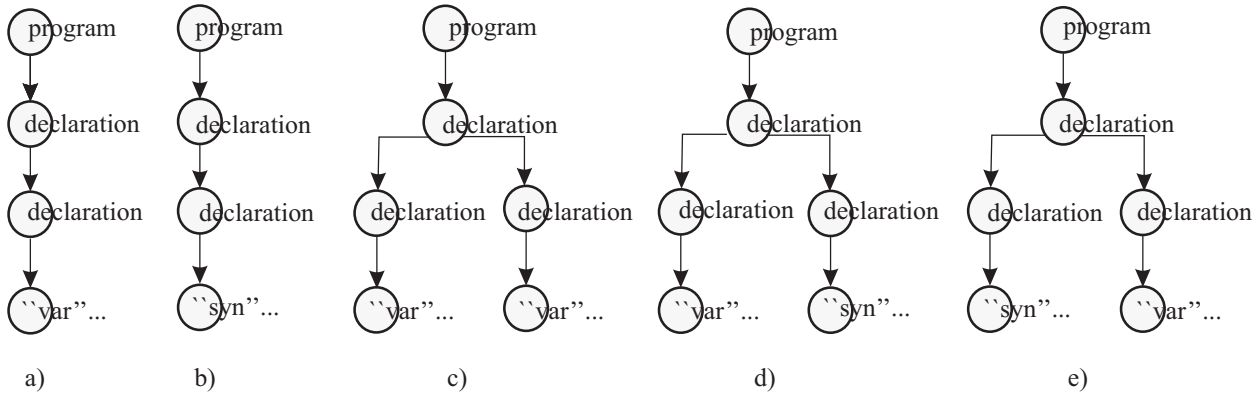

Рис. 1. Первые пять деревьев вывода

На рис. 1 представлены первые пять деревьев вывода, которые получены в результате раскрытия продукционных правил грамматики языка L, в соответствии с описанным ранее алгоритмом. Следующие деревья строятся аналогично.

Тестовые программы, соответствующие этим деревьям, приведены в табл. 2. Как мы видим, в случаях b) и e) семантически корректные программы построить не удалось, так как при раскрытии нетерминала declaration по второй альтернативе (“syn” new_syn _identifier“=” var_ identifier) нарушается семантическое ограничение (5), поскольку ни одна переменная не была объявлена и таблица символов пуста.

Таблица 2

Результаты генерации тестов для языка L

a)	b)	c)	d)	e)
program p1; var a;	Программа не будет построена, так как нарушается семантическое ограничение (5)	program p1; var a; var b;	program p1; var a; syn b = a;	Программа не будет построена, так как нарушается семантическое ограничение (5)

В данном примере были построены только две тестовые программы c) и d), “интересные с точки зрения статической семантики”. Тестовая программа c) проверяет выполнение семантического требования уникальности имен переменных, а d) проверяет сразу два правила: уникальность имен переменных и синонимов и объявление переменной перед объявлением синонима.

Построение двух из пяти деревьев вывода было прервано из-за нарушения контекстных условий. Программа a) семантически “неинтересна”, так как она не обязана удовлетворять никаким семантическим правилам.

Вернемся к случаям b) и e). Построенное синтаксическое (т.е. 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, т.е. 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 семантического правила.

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.

4. Конструктивное описание семантики. Резюмируя недостатки прямолинейного использования классических атрибутных грамматик для генерации тестов, мы пришли к выводу, что требования контекстных условий необходимо трансформировать в явные зависимости между отдельными конструкциями языка (нетерминалами или, что более точно, между атрибутами при нетерминалах классической атрибутной грамматики). Такая форма АГ позволяет не только описывать требования к семантически корректным программам, но и организовывать процесс построения семантически корректной программы некоторым регулярным образом, достраивая дерево разбора и устанавливая значения атрибутов в соответствии с требованиями АГ. Такую форму АГ мы называем *конструктивным описанием семантики.*

Покажем, как можно трансформировать АГ в конструктивное описание на примере языка L, который уже рассматривался выше.

Таблица 3

Синтаксис и статическая семантика языка L

Синтаксис	Семантика
program ::= “program” identifier “;” declarations declarations ::= (declaration)+ declaration ::= (“var” new_var _identifier) |(“syn” new_syn_ identifier “=” var_identifier) “;” new_var_identifier ::= <identifier> new_syn_identifier ::= <identifier> var_identifier ::= <identifier>	1. Имена любых двух переменных должны различаться. 2. Имена любых двух синонимов должны различаться. 3. Имя любой переменной должно отличаться от имени любого синонима. 4. Синоним может быть объявлен только для уже существующей переменной или синонима.

Далее для удобства будем считать, что каждый из нетерминалов обладает какими-то атрибутами, как это принято в атрибутных грамматиках. Например, у нетерминала *new_var_identifier* есть атрибут с именем *name*, множество значений которого совпадает с множеством значений терминального символа <*identifier*>. Пусть нетерминалы *var_identifier* и *new_syn_identifier* обладают такими же атрибутами.

Рассмотрим некоторое альтернативное описание правил статической семантики, представленных в табл. 3. Заметим, что в соответствии с семантикой языка все переменные в программе p на языке L находятся в одной области видимости. Семантическое правило 1, требующее уникальности имен переменных, означает, что для любой программы p на языке L должно выполняться $\forall i \neq j$, $0 < i, j \leqslant N$

$$\text{new\_var\_ identifier}_{\text{i}}\text{.name} \neq \text{new\_var\_ identifier}_{\text{j}}\text{.name}, \tag{1}$$

где под $\textit{new\_var\_identifier}_{\text{i}}$ понимается некоторое i-е вхождение нетерминала *new_var_identifier*, а N — это общее количество нетерминалов *new_var_identifier* в программе p.

Аналогично можно записать семантические правила 2 и 3: $\forall k \neq l$, $0 < k, l \leqslant M$, $\forall i : 0 < i, j \leqslant N$

$$\text{new\_syn\_identifier}_{\text{k}}\text{.name} \neq \text{new\_syn\_identifier}_{\text{l}}\text{.name}, \tag{2}$$

$$\text{new\_var\_identifier}_{\text{i}}\text{.name} \neq \text{new\_syn\_ identifier}_{\text{k}}\text{.name}, \tag{3}$$

где M — это количество конструкций *new_syn_identifier*, а N — это общее количество нетерминалов *new_var_identifier* в программе p.

Рассмотрим семантическое правило 4. В нем говорится о том, что терминальное значение <*identifier*>, стоящее в правой части продукционного правила *var_identifier*, должно совпадать со значением терминала <*identifier*>, соответствующего какому-нибудь раскрытию нетерминала *new_var_identifier* или *new_syn_identifier*. Таким образом, для любой программы p должно быть верно следующее: $\forall p\ (0 < p \leqslant K)$ $\exists i\ (0 < i \leqslant N)$

$$\text{var\_identifier}_\text{p}.\text{name} = \text{new\_var\_identifier}_\text{i}.\text{name}, \tag{4}$$

где K — количество конструкций *var_identifier*, а N — количество конструкций *new_var_identifier* в программе p.

Кроме того, декларация переменной должна предшествовать декларации ее синонима. Для описания этого правила введем для каждого нетерминала атрибут *pos*, значение которого совпадает с позицией в программе, в которой расположена конструкция, соответствующая раскрытию данного нетерминала. Тогда требование, фиксирующее порядок следования деклараций, можно выразить следующим образом:

$$\text{var\_identifier}_\text{p}.\text{pos} > \text{new\_var\_identifier}_\text{i}.\text{pos}. \tag{5}$$

Из соотношений (1) – (4) видно, что семантические правила языка L описывают связи между парами нетерминалов языка L. Если для одного из пары нетерминалов атрибуты известны, то можно вычислить атрибуты для второго нетерминала. Такое представление семантических правил содержит явные указания на семантически связанные синтаксические конструкции, что позволяет создать целенаправленный алгоритм генерации тестовых программ, семантическая корректность которых будет гарантироваться по построению.

На примере простого языка L мы показали, что существует схема пополнения синтаксического дерева узлами, позволяющая разрешить все требования статической семантики. Эта схема работает в случае, когда удается определить частичный порядок, задающий вычисление атрибутов синтаксического дерева.

Описанная идея была апробирована в ряде проектов. Опыт показал, что даже для таких языков как, Java и C#, описание классических атрибутных грамматик в конструктивном стиле возможно и автоматическая генерация по такой грамматике также возможна. Вместе с тем остается вопрос, какова сфера применимости этого подхода (когда удается описать АГ в форме попарных отношений между атрибутами нетерминалов), для какого класса языков такая генерация тестов возможна.

Для того чтобы ответить на этот вопрос, автор сформулировал и доказал следующую теорему.

Теорема 1. *Любая классическая атрибутная грамматика представима в виде системы правил, связывающих атрибуты пар нетерминальных символов соответствующего языка или нескольких атрибутов одного нетерминального символа.*

Замечание. Вообще говоря, одно семантическое правило в своем первоначальном виде может соответствовать нескольким правилам, связывающим пары нетерминалов.

Доказательство. Пусть имеется описание языка L в виде классической атрибутной грамматики. По определению в атрибутной грамматике [17] множество семантических правил представлено в виде системы уравнений вида $X.a = f(X_1.y_1, \ldots, X_n.y_n)$, где a, y_1, …, y_n — атрибуты нетерминалов X, X_1, …, X_n соответственно. Добавим в нетерминал X атрибуты $b_1, \ldots, b_n$, для которых должны выполняться следующие семантические правила: $X.b_i = X_i.y_i$, $\forall i$, $0 < i \leqslant n$. Тогда исходное уравнение можно записать в виде $X.a = f(X.b_1, \ldots, X.b_n)$. Таким образом, исходная система уравнений может быть представлена в виде системы, объединяющей уравнения

$$\begin{cases} X.b_1 = X_1.y_1, \\ \ldots, \\ X.b_n = X_n.y_n, \\ X.a \ = f(X.b_1, \ldots, X.b_n). \end{cases} \tag{6}$$

Первые n уравнений системы (6) представляют собой правила, связывающие атрибуты пар нетерминалов (X, X_i), $0 < i \leqslant n$. Последнее уравнение представляет собой правило, связывающее атрибуты a, b_1, …, b_n, относящиеся к одному узлу. Теорема доказана.

Следствием из этой теоремы служит вывод о том, что для любого языка, статическая семантика которого задана при помощи АГ, можно построить систему отношений между атрибутами нетерминалов в синтаксическом дереве разбора такую, что этому дереву разбора будет соответствовать семантически корректная программа.

Практика показывает, что требуется несколько видов отношений. Для их рассмотрения введем несколько определений. В каждой паре нетерминалов, связанных некоторым семантическим правилом, будем выделять *источник* и *цель*.

Определение 1. *Источником* семантического правила R будем называть нетерминал, атрибуты которого используются в семантическом правиле R для вычисления атрибутов другого нетерминала, который будем называть *целью*.

Определение 2. Будем называть атрибуты источника, участвующие в определении семантического правила R, *независимыми атрибутами* семантического правила R, а атрибуты цели, также участвующие в определении семантического правила, — *зависимыми атрибутами* R.

Каждое семантическое правило характеризуется типом отношения, возникающего между атрибутами источника и цели. Например, правила, описанные в (1) и (2), представляют отношение одного типа, которое обозначается знаком "$\neq$", а правило, описанное в (4), представляет отношение другого типа, которое обозначается знаком "=".

Кроме того, семантические правила характеризуются еще одним свойством. Правило, описанное в (1), ставит в соответствие каждому раскрытию нетерминала *new_var_identifier* другое раскрытие такого же нетерминала, а правило, описанное в (4), ставит в соответствие каждому нетерминалу *var_identifier* только один случай раскрытия *new_var_identifier*. Введем понятие *типа* семантического отношения. В данном примере есть семантические правила двух типов:

many-to-many — из множества нетерминалов, подпадающих под описание источника семантического правила, и множества нетерминалов, подпадающих под описание цели семантического правила, составляется декартово произведение; элементы этого декартова произведения составляют пары: источник и цель семантического правила;

one-to-many — для цели обязан существовать источник, при этом один источник может быть связан семантическим взаимоотношением с несколькими целями.

Так, правила, описанные в (1) и (2), соответствуют семантическим правилам типа *many-to-many*, а правило, описанное в (4), — семантическому правилу типа *one-to-many*.

Для конструктивного формального описания статической семантики вводим новый язык SRL (Semantic Relation Language) [26]. Язык SRL очень простой, тем не менее мы не будем давать его полное описание, а ограничимся только теми конструкциями, которые необходимы для разбора примеров, раскрывающих семантические правила (1) – (5).

```
relation ::= ("many-to-many"|"one-to-many")
        "relation" <new_rule_id> "{""
        (comment)
        ("unequal" | "equal")
        "target" nonterm_type ("|"nonterm_type)*                    (7)
        "source" nonterm_type ("|"nonterm_type)*
        "}";
nonterm_type ::= <nonterm_type> "{"
        <attr_name>("," "attr_name>)* "}";
```

Конструкция relation позволяет описывать семантические правила различных типов (*many-to-many, one-to-many*), связывающие упорядоченные/неупорядоченные (*ordered, unordered*) пары нетерминалов (*nonterm_type*), и отношения (*equal, unequal*), возникающие между атрибутами (<*attr_name*>) этих нетерминалов.

Опишем на языке SRL правила статической семантики для языка L из примера 1 (см. табл. 4).

Выделим основные особенности конструктивного задания АГ для задач генерации тестов для семантических анализаторов по сравнению с классическими атрибутными грамматиками:

1) нет необходимости писать пользовательские функции (так, в примере 1 пользователь должен был реализовать функции emptysymtab(), contains(), push(), being_ahead());

2) семантически зависимые синтаксические конструкции указаны явно;

3) подход позволяет локализовать формальное описание каждого неформального семантического требования в пределах одного формального семантического правила.

5. Критерий покрытия. При тестировании семантического анализатора компилятора необходимо проверить правильность реализации всех контекстных условий целевого языка, а также правильность диагностики ошибок. Для этого необходимо сгенерировать тестовые программы, удовлетворяющие всем описанным правилам статической семантики целевого языка (позитивные тесты), и тестовые программы, нарушающие определенные семантические требования (негативные тесты).

Таблица 4

Семантика языка L на языке SRL

№	Семантические правила
1	**many-to-many relation** R1 **{** “Уникальность имен переменных и синонимов” **unordered** **unequal** **target** new_var_identifier **{**name**}** | new_syn_identifier **{**name**}** **source** new_var_identifier **{**name**}** | new_syn_identifier **{**name**}** **}**
2	**one-to-many relation** R2 **{** “Синоним может быть определен только для уже определенной переменной или синонима” **ordered** **equal** **target** var_identifier **{**name**}** **source** new_var_identifier **{**name**}** | new_syn_identifier **{**name**}** **}**

Для оценки полноты тестирования необходимо определить *критерий покрытия.* Такой критерий разбивает всю область входных данных на конечное число подобластей (*классов эквивалентности*). Это разбиение строится таким образом, чтобы в одну подобласть попали данные, предназначенные для обнаружения сходных ошибок. Для удовлетворения условиям критерия покрытия достаточно построить тестовый набор, включающий в себя по одному представителю из каждого класса эквивалентности.

В данной работе для тестирования семантического анализатора используется тестирование на основе моделей. Говоря о тестировании программного обеспечения, под целевой системой понимается тестируемая система. Подход состоит в следующем: 1) строится модель целевой системы (в данном случае семантического анализатора); 2) строится исчерпывающий набор тестов в рамках полученной модели целевой системы. Предполагается, что тесты, построенные для того, чтобы покрыть модель целевой системы, будут достаточно хорошо покрывать его реализацию. Исходя из этих соображений, сформулируем критерии покрытия, которым должны удовлетворять наборы позитивных и негативных тестов для семантического анализатора компилятора соответственно.

Модель входных данных семантического анализатора описывается формально при помощи SRL. Каждое семантическое правило, формально заданное на языке SRL, явно зависит от пары синтаксических конструкций — “источник-цель”. Программа, выведенная из грамматики целевого языка и содержащая указанные синтаксические конструкции, покрывает данное семантическое правило. Следовательно, множество программ, покрывающих все семантические правила, покрывает модель входных данных. И, следовательно, это множество программ должно хорошо покрывать реализацию семантического анализатора.

Для тестирования семантического анализатора мы предлагаем использовать следующий критерий покрытия.

Набор позитивных тестов для семантического анализатора компилятора должен покрывать все тройки (R, S, T), где R — семантическое правило, S — вершина, соответствующая нетерминалу-источнику, описанному в R, T — вершина, соответствующая нетерминалу-цели, описанному в R.

Для набора негативных тестов критерий покрытия формулируется аналогично. Для генерации негативных тестов необходимо создать описания семантических правил обратные обычным правилам. Затем запустить процесс генерации тестов для каждого обращенного правила. Благодаря простоте описания правил на языке SRL, процесс обращения правил можно автоматизировать. Таким образом, требуется построить набор негативных тестов, покрывающий все тройки (R^{-1}, S, T), где R^{-1} — семантическое правило обратное R, S — вершина, соответствующая нетерминалу-источнику, описанному в R^{-1}, T — вершина, соответствующая нетерминалу-цели, описанному в R^{-1}.

6. Генерация тестов на основе конструктивного описания статической семантики. Использование конструктивного описания на языке SRL позволило нам разработать алгоритм семантически управляемой генерации, следуя которому можно целенаправленно построить позитивные или негативные тесты, удовлетворяющие описанным в предыдущем параграфе критериям покрытия.

В общем случае множество всех возможных программ, содержащих синтаксические конструкции,

соответствующие источнику и цели какого-либо семантического правила, бесконечно. Поэтому мы высказываем следующую гипотезу.

Гипотеза 1. *Разумное ограничение глубины рекурсии при раскрытии продукционных правил грамматики целевого языка позволит построить набор тестов, достаточный для проведения удовлетворительного тестирования семантического анализатора компилятора. Размер ограничения на глубину рекурсии в рамках данной гипотезы должен определяться опытным путем.*

Однако, даже после введения ограничения на глубину рекурсии, количество тестовых программ будет слишком велико.

Пусть T_R — конечное множество всех тестовых программ, соответствующих одному семантическому правилу R. Такое семантическое правило R будем называть *первичным семантическим правилом* для тестовых программ, принадлежащих 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: $T_R = \bigcup\limits_{S \in F_R} S$.

Определение 3. Подмножество S множества T_R будем называть *элементом семейства подмножеств* F_R, если S принадлежат только те программы, синтаксические деревья которых содержат эквивалентные поддеревья, состоящие из двух цепочек, соединяющих корень дерева программы с узлами, соответствующими источнику и цели первичного семантического правила R.

Определение 4. Поддерево, соединяющее корень дерева тестовой программы с источником и целью первичного семантического правила, будем называть *первичным поддеревом.*

Заметим, что при таком разбиении подмножества могут пересекаться, поэтому их нельзя считать классами эквивалентности. Однако тестовые программы, принадлежащие одному подмножеству, могут считаться эквивалентными с точки зрения проверки правильности интерпретации первичного семантического правила, соответствующего данному подмножеству, в семантическом анализаторе. Это возможно, поскольку каждая такая программа должна удовлетворять первичному семантическому правилу, примененному в фиксированном синтаксическом контексте, заданном первичным поддеревом.

Сформулируем еще одну гипотезу для ограничения количества тестов.

Гипотеза 2. *Для организации удовлетворительного тестирования семантического анализатора в компиляторе* (*в соответствии с полученным в предыдущем разделе критерием покрытия*) *достаточно построить по одному представителю* (*тестовой программе*) *каждого подмножества семейства* F_R. *Иными словами, для того чтобы добиться выполнения условий критерия покрытия, сформулированного в предыдущем разделе, необходимо построить набор тестовых программ следующего вида*: $\mathbb{T} = \{p_i \in S_i \,|\, S_i \in F_R,\, 1 \leqslant i \leqslant |F_R|\}$.

Пусть грамматика исходного языка задана в эквивалентном BNF виде, а контекстные ограничения — в виде SRL. Каждое семантическое правило привязывается к нетерминалу, соответствующему описанию цели этого семантического правила. В качестве внутреннего представления тестовых программ используются абстрактные синтаксические деревья.

Для того чтобы в соответствии с гипотезой 2 построить набор тестовых программ, необходимо на первом этапе для каждого семантического правила (исполняющего в данном случае роль первичного правила) построить множество всех первичных поддеревьев с учетом ограничения глубины рекурсии. В результате будет получено множество синтаксически неполных деревьев тестовых программ. Далее алгоритм работает отдельно с каждым первичным поддеревом.

Первичное поддерево достраивается минимальным образом до синтаксически полного дерева, т.е. продолжается раскрытие нетерминалов до тех пор, пока это позволяет грамматика языка. Достраивание дерева минимальным образом осуществляется за счет игнорирования построения всех дочерних вершин, которым соответствуют опциональные нетерминальные символы в правых частях соответствующих продукционных правил.

Каждому синтаксически полному дереву, полученному из первичного поддерева, соответствует *базовое множество* семантических правил, состоящее из семантических правил целевого языка, связанных с нетерминалами, которым соответствуют вершины данного синтаксического дерева.

Используя описания семантических правил из базового множества, строится разрешимая система правил вычисления атрибутов вершин данного синтаксического дерева. То есть для каждого семантического правила из базового множества определяются все пары вершин, соответствующих описаниям источника и цели данного правила такие, что вычисление атрибутов цели не противоречит правилам вычисления остальных атрибутов. Если какие-то вершины не удается найти, то они достраиваются в дерево.

Добавление в дерево новой вершины может повлечь за собой расширение базового множества семантических правил, соответствующих данному дереву. Если это происходит, то процесс подбора источника

и цели для каждого семантического правила из базового множества повторяется заново, и так до тех пор, пока не будет построена разрешимая система правил вычисления атрибутов или пока не будут перебраны все варианты пар вершин — тогда данное синтаксическое дерево объявляется семантически неразрешимым и отбрасывается. На практике семантически неразрешимых синтаксических деревьев оказывается порядка 2 %.

Полученная система правил вычисления атрибутов представляется в виде графа атрибутной зависимости. В случае если в графе атрибутной зависимости присутствует цикл, то делается вывод о некорректности описания семантических правил на основе теоремы *о корректности семантических правил*, доказанной Д. Кнутом [17]. В случае если получен ациклический граф, то организуется процесс вычисления атрибутов, аналогично тому, как это делается в атрибутных грамматиках. После того как будут вычислены все атрибуты, тестовая программа преобразуется из внутреннего представления в текст на целевом языке.

Описанный выше алгоритм построения семантически корректных тестовых программ конечен благодаря тому, что набор нетерминалов и семантических ограничений целевого языка конечен. Множество терминалов целевого языка также можно считать конечным, рассматривая все идентификаторы (кроме ключевых слов целевого языка) и числовые значения равнозначными с точки зрения статического анализа. И хотя, вообще говоря, множество всех семантически корректных программ на целевом языке может быть бесконечным (по причине существования рекурсии в продукционных правилах), достаточно, как показывает практика, построить множество программ при условии ограничения глубины рекурсии.

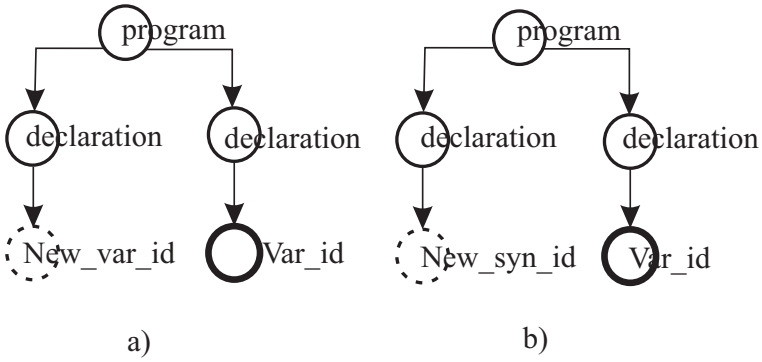

Рис. 2

Пример 2. Рассмотрим язык L из примера 1. Пусть семантика языка L представлена в виде SRL (см. табл. 4).

Пусть необходимо сгенерировать множество семантически корректных тестовых программ для компилятора с языка L, удовлетворяющее критерию покрытия, сформулированному в предыдущем разделе. Пусть семантическое правило $R1$ ставится в соответствие следующим нетерминалам: *new_var_identifier, new_syn_identifier*, так как эти нетерминалы указаны в описании цели $R1$. Аналогично, семантическое правило $R2$ ставится в соответствие нетерминалу *var_identifier*.

В соответствии с описанным выше алгоритмом построим множество первичных поддеревьев для каждого семантического правила.

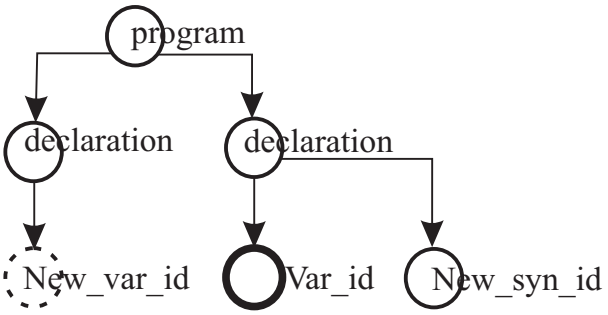

Рис. 3

На рис. 2 представлены поддеревья из множества первичных поддеревьев, соответствующих семантическому правилу $R2$ языка L. Вообще говоря, множество первичных поддеревьев, соответствующих семантическому правилу R2, бесконечно, так как в грамматике языка L есть рекурсивные продукции. Вершины, соответствующие описанию цели семантического правила, обозначены жирными окружностями, а вершины-источники обозначены пунктирными окружностями. Далее по алгоритму требуется достроить первичные поддеревья до синтаксически полных.

В данном случае все первичные поддеревья (рис. 2) достраиваются одинаковым способом до синтаксически полных деревьев, а именно: к родительским вершинам узлов *Var_id* присоединяются дочерние узлы, соответствующие нетерминалам *new_syn_identifier*, а к родительским вершинам узлов *New_syn_id* присоединяются дочерние узлы, соответствующие нетерминалу *var_identifier*. Добавление в деревья вершин, соответствующих *new_syn_identifier*, влечет добавление в базовые множества семантических правил правила $R1$, связанного с нетерминалом *new_syn_identifier*. Добавление в деревья вершин, соответствую-

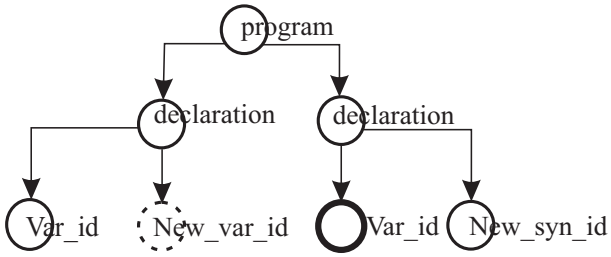

Рис. 4

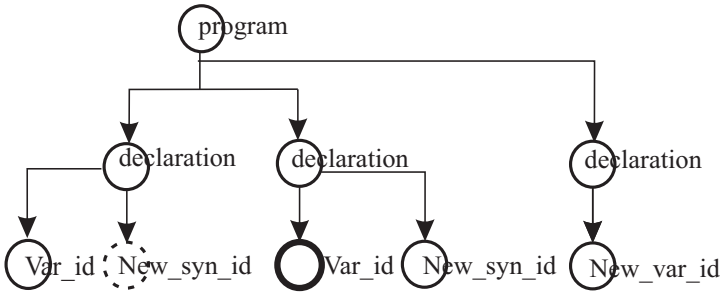

Рис. 5

щих нетерминалу *var_identifier*, не влечет расширения базовых множеств правил, так как соответствующее правило уже там есть — это первичное правило. Таким образом, базовое множество семантических правил, соответствующих каждому дереву, состоит из первичного правила $R2$ и правила $R1$.

Итак, после достраивания первое первичное поддерево (рис. 2 a) примет вид, представленный на рис. 3.

Дереву на рис. 3 соответствует разрешимая система правил вычисления атрибутов:

$$\begin{cases} \text{New\_var\_id.name} \neq \text{New\_syn\_id.name} \\ \text{Var\_id.name} = \text{New\_var\_id.name} \\ \text{Var\_id.pos} < \text{New\_var\_id.pos}. \end{cases}$$

Процесс вычисления значений атрибутов будет состоять из двух шагов. На первом шаге атрибуты *new_var_identifier* и *new_syn_identifier* инициализируются различными значениями, а на втором шаге вычисляется значение атрибута нетерминала *var_identifier*.

Первая тестовая программа выглядит следующим образом:

```
program p1;
var a;
syn s = a;
```

Первичному поддереву рис. 2 b соответствует синтаксически полное дерево, представленное на рис. 4.

Данному дереву соответствует следующая система правил вычисления атрибутов:

$$\begin{cases} \text{New\_syn\_id}_1\text{.name} \neq \text{New\_syn\_id}_2\text{.name} \\ \text{Var\_id}_1\text{.name} = \text{New\_syn\_id}_2\text{.name} \\ \text{Var\_id}_2\text{.name} = \text{New\_syn\_id}_1\text{.name} \\ \text{Var\_id}_1\text{.pos} < \text{New\_syn\_id}_2\text{.pos} \\ \text{Var\_id}_2\text{.pos} < \text{New\_syn\_id}_1\text{.pos}. \end{cases}$$

Данная система неразрешима из-за двух последних неравенств. Следовательно, в синтаксическое дерево тестовой программы, представленное на рис. 4, требуется достроить вершину-источник, соответ-

ствующую правилу $R2$, для вершины-цели *Var_id*, появившейся в дереве в результате построения синтаксически полного дерева. В результате дерево примет вид, отображенный на рис. 5. Дереву на рис. 5 соответствуют:

разрешимая система правил вычисления атрибутов	тестовая программа на языке L

$\begin{cases} \text{New\_var\_id.name} \neq \text{New\_syn\_id}_1\text{.name} \\ \text{New\_var\_id.name} \neq \text{New\_syn\_id}_2\text{.name} \\ \text{New\_syn\_id}_1\text{.name} \neq \text{New\_syn\_id}_2\text{.name} \\ \text{Var\_id}_1\text{.name} = \text{New\_syn\_id}_2\text{.name} \\ \text{Var\_id}_2\text{.name} = \text{New\_var\_id.name} \\ \text{Var\_id}_1\text{.pos} < \text{New\_syn\_id}_2\text{.pos} \\ \text{Var\_id}_2\text{.pos} < \text{New\_var\_id.pos.} \end{cases}$	program p1;
	var a;
	syn b = a;
	syn c = b;

Таким образом, мы получили две тестовые программы, нацеленные на тестирование реализации проверки семантического правила $R2$ в компиляторе с языка L. Аналогичным образом строятся тесты для правила $R1$.

7. Практические результаты. На основе алгоритма семантически управляемой генерации был разработан инструмент STG (Semantic Test Generator). В качестве входных данных для STG используется описание синтаксиса целевого языка в некотором BNF-подобном виде и описание правил статической семантики на языке SRL. STG полностью автоматически позволяет получить набор семантически корректных программ на целевом языке, о каждой из которых известно, для проверки какого контекстного условия она предназначена. Для проверки каждого семантического правила строится набор программ, в каждой из которых данное правило выполняется в различном синтаксическом контексте.

Тесты, полученные при помощи STG, документированы, т.е. в каждом тесте имеется информация о том, какое правило является первичным и какие правила составляют базовое множество семантических правил для данного теста. STG позволяет генерировать тесты сложной структуры. Например, для тестирования семантического анализатора компилятора с языка Java каждый тест должен представлять собой директорию, содержащую другие директории (Java-пакеты) и файлы с расширением java (Java-классы).

По сравнению с генерацией тестов с использованием фильтрации при помощи STG удается сгенерировать тесты на несколько порядков быстрее. Гипотетическое время генерации с использованием фильтрации 3792 семантически корректных тестовых программ на языке C для компилятора gcc составляет приблизительно 1811 часов. Гипотетическое время генерации получено нами как результат суммирования времени, затраченного на генерацию синтаксически корректных тестов при помощи инструмента, разработанного в ИСП РАН, со временем компиляции полученных тестов, так как это время совпадает со временем фильтрации (отбора семантически корректных тестов среди синтаксически корректных). При гипотетической генерации никакой критерий покрытия, касающийся правил статической семантики, не использовался. Вероятность случайного удовлетворения требованиям какого-либо критерия слишком мала.

В работе М. А. Посыпкина [25, с. 55] сформулирована теорема о том, что для любого критерия покрытия существует некоторое число I, являющееся верхней границей числа узлов в деревьях вывода тестов такое, что полученные при помощи предложенного им алгоритма тесты будут удовлетворять заданному критерию покрытия. Приводятся результаты практических экспериментов, из которых следует, что для построения семантически корректных выражений для языка калькулятора, состоящего из четырех продукций, т.е. $I \leqslant 4$, требуется более 36 часов. Причем в зависимости от количества нетерминалов в грамматике исходного языка время генерации увеличивается экспоненциально. Для сравнения выбрана работа М. Посыпкина, так как результаты, полученные им в области генерации семантически корректных тестов, являются наиболее актуальными на данный момент.

Для сравнения, грамматика языка C состоит более чем из 100 продукционных правил. При помощи STG примерно за 1 час было получено около 9000 семантически корректных тестовых программ, покрывающих 80 правил статической семантики языка C.

В настоящее время инструмент STG используется для генерации тестов для семантического анализатора транслятора с языка Java. Статическая семантика языка Java значительно сложнее статической семантики языка C. Уже разработаны тесты, охватывающие декларации пакетов, классов, полей, методов, параметризованных классов, выражения присваивания и др. Сгенерированные тесты позволили обнаружить несколько ошибок в трансляторе. К моменту написания данной статьи количество форма-

лизованных семантических правил языка Java составило 182, что соответствует приблизительно 41 % от общего числа семантических правил, описанных в стандарте. При этом было прочитано больше половины стандарта, а именно, 57 %.

Опыт показал, что при использовании предлагаемой методики разработчики тестов демонстрируют более высокую суммарную производительность, чем разработчики, создающие тесты вручную. Таким образом, результаты апробации STG на реальных языках программирования позволяют предполагать, что данная методика может применяться в проектах по разработке программного обеспечения для получения более эффективных результатов тестирования.

8. Заключение. В работе предлагается метод конструктивного описания правил статической семантики языков программирования, основанный на модельном подходе. Методика позволяет целенаправленно генерировать корректные с точки зрения статической семантики тестовые программы (позитивные тесты) и некорректные тестовые программы с известными ошибками (негативные тесты).

При использовании предложенной методики не требуется построения фильтра, проверяющего семантическую корректность построенных тестовых программ, что позволяет не только уменьшить время генерации семантически корректных тестовых программ, но и теоретически повысить эффективность полученных тестов, так как при их построении семантические правила рассматриваются с другой точки зрения, нежели при построении семантического анализатора.

Предложены критерии покрытия для наборов позитивных и негативных тестов для семантического анализатора в компиляторе.

На основе метода конструктивного описания был разработан генератор, позволяющий получить множества тестовых программ, удовлетворяющих предложенным критериям покрытия.

Практические результаты применения предложенного метода показывают ценность данного метода для формального описания семантики языков программирования и текстовых нотаций, а также целесообразность его использования для тестирования компиляторов и других текстовых процессоров.

СПИСОК ЛИТЕРАТУРЫ

1. *Hanford K.V.* Automatic generation of test cases // IBM System Journal. 1970. **9**, N 4. 242–257.
2. *Purdom P.* A sentence generator for testing parsers // Behavior and Information Technology. 1972. **12**, N 3. 366–375.
3. *Wichmann B.A., Jones B.* Testing ALGOL 60 compilers // Software — Practice and Experience. 1976. **6**, N 2. 261–270.
4. *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. 897–918.
5. *Duncan A.G., Hutchison J.S.* Using attributed grammars to test designs and implementation // Proceedings of the 5th International Conference on Software Engineering. Piscataway: IEEE Press, 1981. 170–178.
6. *Sirer E.G., Bershad B.N.* Using production grammars in software testing // Proceedings of the 2nd Conference on Domain-Specific Languages. New York: ACM Press, 1999. 1–13.
7. *Kalinov A., Kossatchev A., Petrenko A., Posypkin M., Shishkov V.* Using ASM specifications for automatic test suite generation for mpC parallel programming language compiler // Proceedings of the 4th International Workshop on Action Semantics. BRISC Notes Series. University of Aarhus (Denmark), 2002. 96–106.
8. *Kalinov A., Kossatchev A., Petrenko A., Posypkin M., Shishkov V.* Using ASM specifications for compiler testing // Proceedings of the 10th International Workshop on Abstract State Machines. Lecture Notes in Computer Science. Vol. 2589. New York–Heidelberg–Berlin: Springer-Verlag, 2003. 415.
9. *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. 218–229.
10. *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.
11. *Hoare C.A.R., Wirth N.* An axiomatic definition of the programming language PASCAL // Acta Informatica. 1973. **2**, N 4. 335–355.
12. *Lee J.A.N.* Computer semantics. New York: Van Nostrand Co., 1972.
13. *Lucas P., Lauer P., Stigleitner H.* Method and notation for the formal definition of programming languages. IBM Technical Report 25.087. IBM Lab. Vienna, 1968.
14. *Lucas P., Walk K.* On the formal description of PL/1 // Annual Review Automatic Programming. 1969. **6**, N 3. 105–182.
15. *Wegner P.* The Vienna definition language // Computer Surveys. 1972. **4**, N 1. 5–63.
16. *Knuth D.E.* Semantics of context-free languages // Mathematical Systems Theory. 1968. **2**, N 2. 127–146.
17. *Knuth D.E.* Semantics of context-free languages: Correction // Mathematical Systems Theory. 1971. **5**, N 1. 179.
18. Yacc and Lex. http://dinosaur.compilertools.net/

19. *Kastens U.* Attribute grammars as a specification method // Proceedings of the International Summer School on Attribute Grammars. Lecture Notes in Computer Science. Vol. 545. New York–Heidelberg–Berlin: Springer-Verlag, 1991. 16–47.
20. *Bischoff K.M.* Ox: An attribute-grammar compiling system based on yacc, lex and c: User reference manual. User Manual, 1993. http://citeseer.ist.psu.edu/bischoff93ox.html
21. *Pagan F.* Formal specification of programming languages: A panoramic primer. Englewood Cliffs: Prentice Hall, 1981.
22. *Slonneger K., Barry L.* Kurtz formal syntax and semantics of programming languages: A laboratory based approach. London: Addison-Wesley Longman Publishing Co., 1995.
23. *Зеленов С.В., Зеленова С.А., Косачев А.С., Петренко А.К.* Генерация тестов для компиляторов и других текстовых процессоров // Программирование. 2003. **29**, № 3. 104–111.
24. *Gurevich Y.* Abstract state machines: An overview of the project // Foundations of Information and Knowledge Systems. Lecture Notes in Computer Science. Vol. 2942. New York–Heidelberg–Berlin: Springer-Verlag, 2004. 6–13.
25. *Посыпкин М.А.* Применение формальных методов для тестирования компиляторов. Диссертационная работа на соискание ученой степени кандидата физико-математических наук. М., ИСП РАН, 2004.
26. *Архипова М.В.* Генерация тестов для семантических анализаторов. Препринт ИСП РАН. № 9. М., ИСП РАН, 2006.

Поступила в редакцию
10.07.2006