Генерация тестов для модулей проверки статической семантики в компиляторах

М.В. Архипова

Аннотация. 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.

1. Введение

Для задач тестирования компилятор удобно разбивать на модули, имеющие различные функции: лексический и синтаксический анализаторы, семантический анализатор, оптимизатор и генератор кода. Так как описанное разбиение условно, а для тестировщика компилятор представляется единой системой, преобразующей программы, написанные на входном языке, то в качестве тестов для тестирования любого модуля компилятора могут использоваться только программы, написанные на входном языке[1]. Тестирование различных частей компилятора выдвигает различные требования к свойствам тестовых программ, их разнообразию и способу проверки корректности поведения тестируемого модуля. Например, при тестировании лексического и синтаксического анализаторов в составе компилятора существенным является синтаксическая корректность или некорректность тестов, при этом их семантика не важна.

Будем называть процесс тестирования модулей проверки статической семантики в компиляторах семантическим тестированием. Тестовые программы для семантического тестирования строятся с учетом семантических правил исходного языка. Естественным требованием для таких тестов является их синтаксическая корректность.

Семантический анализатор (или модуль проверки статической семантики в компиляторах) должен пропускать корректные с точки зрения статической

[1] Далее будем называть такие программы: *тестами*, или *тестовыми программами,* или *программами-тестами*.

семантики программы и обнаруживать некорректность для “неправильных” с точки зрения статической семантики программ, выдавая об этом сообщения, позволяющие локализовать проблему.

Различают тесты “позитивные” и “негативные”. Целевая система[2] должна выдавать сообщения об ошибках в ответ на “негативные” тесты и молча пропускать “позитивные”. При помощи набора “позитивных” тестов можно проверить, что все семантически правильные программы пропускаются, а при помощи набора “негативных” тестов — что все неправильные программы отвергаются. Такого рода семантические тесты являются самопроверяющимися [18], так как они не требуют особо проверять, что тестируемая система ведет себя корректно в ходе теста.

Для оценки полноты тестирования необходимо определить критерий покрытия. Такой критерий разбивает всю область входных данных на конечное число подобластей. Это разбиение строится таким образом, чтобы в одну подобласть попали данные, предназначенные для обнаружения сходных ошибок.

Задачи тестирования различных модулей компилятора исследовались неравномерно, и к настоящему моменту лучше всего изучен вопрос тестирования лексического и синтаксического анализаторов. Так произошло в основном благодаря появлению формы Бэкуса-Наура (BNF) [1], которая является общепризнанным способом формального описания синтаксиса языков программирования.

Среди работ, посвященных решению задачи формального описания семантики языков программирования, выделим следующие: определение Алгола 60 средствами расширенного алгоритма Маркова, данное Дебаккером [4], определение Алгола 60 с помощью λ-исчисления, принадлежащее Ландину [5, 6, 7] (см. также Бём [8, 9]), определение Микро-Алгола с помощью рекурсивных функций, применяемых к программе и к “векторам состояний”, принадлежащее Маккарти [10] (см. также Маккарти и Пэинтер [11]); определение языка Эйлер с помощью семантических правил, применяемых во время синтаксического анализа программы, предложенное Виртом и Вебером [12], и определение языка PL/I, данное Венской лабораторией фирмы IBM и основанное на работе Маккарти и Ландина [23], а также на понятии абстрактной машины, введённом Элготом [13, 14].

Особое место занимает подход к описанию семантики языка с помощью атрибутных грамматик, введенных Кнутом [2, 3]. Существенное различие между предшествующими методами и атрибутными грамматиками Кнута состоит в том, что остальные определения представляют собой довольно сложные процессы, применяемые ко всему языку; можно сказать, что человек, прежде чем он поймёт описание языка, должен будет понять, как устроен его компилятор. Напротив, в атрибутных грамматиках каждая конструкция языка

[2] *Целевой системой* будем называть программную систему, которая выступает в качестве объекта тестирования. В данной статье под целевой системой в основном будет подразумеваться модуль проверки статической семантики в компиляторе.

определяется только через её “непосредственное окружение”, тем самым взаимосвязи между определениями разных частей языка сводятся к минимуму. Например, определение составных операторов, операторов перехода и т.д. не влияет существенно на определение оператора печати; любое из правил можно выбросить, и получится строгое определение другого языка. Такая локализация и разделение семантических правил помогает сделать определение семантики языка более понятным и кратким.

Хотя определения семантики языков программирования большинства авторов не так сложны, как определение Дебаккера, в их работах присутствуют относительно сложные зависимости между отдельными частями определения.

На основании изложенного можно сделать вывод, что ни одна из ранее предложенных схем формального определения семантики не в состоянии дать такого же краткого и простого определения семантики, как атрибутные грамматики.

Отметим, что семантические правила в том виде, в котором они описываются атрибутными грамматиками, не зависят от выбранного метода синтаксического анализа. Они привязаны лишь к конкретным формам синтаксиса. Семантические правила зависят от имен нетерминалов в левой и правой частях синтаксических правил. Конкретные знаки пунктуации и порядок, в котором нетерминалы располагаются в правых частях правил, несущественны с точки зрения семантических правил. Таким образом, атрибутные грамматики хорошо сочетаются с идеей Маккарти об “абстрактном синтаксисе” [10, 11].

Формализм атрибутных грамматик оказался очень удобным средством для описания семантики языков программирования. 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.

Атрибутные грамматики широко использовались для описания семантики языков программирования. Было создано несколько систем автоматизации разработки трансляторов и расширений для них, основанных на формализме атрибутных грамматик: Yacc и Lex [15], LIGA [20], Ox [21]. Опыт их использования показал, что атрибутный формализм может быть успешно применен для описания семантики языка при создании транслятора. Но при попытке применить атрибутные грамматики для решения задачи автоматической генерации семантически корректных программ возникают трудности. Атрибутные грамматики содержат в себе мало информации, которая могла бы быть использована для упорядочивания генерации семантически корректных программ-тестов.

2. Атрибутные грамматики и их применимость к задаче генерации семантических тестов

Атрибутные грамматики в конце 60-х годов XX века ввел Кнут для описания статической семантики языков программирования [2, 3]. Напомним некоторые определения.

Атрибутной грамматикой называется тройка AG = (G, A, R), где

G — контекстно-свободная грамматика,
A — конечное множество атрибутов,
R — конечное множество семантических правил.

Контекстно-свободной грамматикой называется четверка *G = (N, T, P, D),* где

N — конечное множество нетерминальных символов,
T — конечное множество терминальных символов,
P — конечное множество правил вывода,
$D \in N$ — стартовый символ грамматики.

Элементы множества $V = N \cup T$ называются *грамматическими символами*.

Правило вывода — это пара вида $X \to \alpha$, где $X \in N$, а $\alpha \in V^*$, т.е. слева в правиле вывода находится нетерминал, а справа — строка, состоящая из грамматических символов.

Пустая строка обозначается символом ε.

С каждым символом $X \in V$ ассоциируется конечный набор *атрибутов A(X)*. Рас-сматриваются два класса атрибутов: *наследуемые I(X)* и *синтезируемые S(X).*

Для правила вывода $p \in P$, имеющего вид $p: X_0 \to X_1 \ldots X_n$ $(n \geq 0)$, обозначим через $X_i.a$ значения атрибутов его элементов. Здесь $a \in A(X_i)$, $0 \leq i \leq n$. Кроме того, правилу вывода *p* сопоставляется конечное множество *семантических правил* R_p. Множество R_p содержит по одному правилу для каждого синтезируемого атрибута X_0 и для каждого наследуемого атрибута X_i при $1 \leq i \leq n$.

Каждое правило в R_p, таким образом, имеет вид $X_i.a = f(y_1, \ldots y_k)$, $k \geq 0$, где $a \in S(X_i)$ при $i = 0$, $a \in I(X_i)$ при $1 \leq i \leq n$; *f* называется семантической функцией, все y_i являются атрибутами элементов правила, отличными от вычисляемых.

Деревом вывода для программы *L*, созданным по грамматике *G*, называется дерево, каждая вершина которого соответствует грамматическому символу $X \in V$ или ε; корень соответствует стартовому символу *D*; и, кроме того, если вершина, соответствующая символу *X*, имеет детей $X_1, \ldots, X_n$, то $X \to X_1 \ldots X_n$ — правило вывода грамматики *G*.

Атрибутированным деревом программы L называется дерево вывода программы *L* такое, что к каждой его вершине *N*, соответствующей символу *X*, присоединены атрибуты, соответствующие грамматическому символу *X*. Значение присоединенного атрибута $a \in A(X)$ в вершине *N* принято обозначать *N.a*.

Графом атрибутной зависимости для атрибутного дерева *T* программы *L* называется ориентированный граф, вершинами которого выступают атрибуты вершин дерева *T*, а дуги строятся по следующему правилу: из вершины, соответствующей атрибуту *N.b,* существует дуга, ведущая в вершину, соответствующую атрибуту *N.a*, тогда и только тогда, когда значение *N.a* в силу введенных семантических правил зависит от значения *N.b.*

Если значения всех атрибутов могут быть получены посредством вычисления семантических функций, то атрибутная грамматика называется *правильно определенной (well defined)*.

Кнут показал, что атрибутная грамматика является правильно определенной тогда и только тогда, когда соответствующий граф атрибутной зависимости программы L ацикличен [2].

Пример. Рассмотрим язык $L = \{a^n b^n c^n \mid n > 0\}$, т.е. состоящий из предложений вида *abc, aabbcc*, *aaabbbccc*, и т. д. Для его формального описания используем атрибутную грамматику. Синтаксис языка *L* в виде BNF будет выглядеть так:

```
S ::= A B C
A ::= a | aA
B ::= b | bB
C ::= c | cC
```

Дополним синтаксис семантической информацией: каждому грамматическому символу *x*, кроме стартового, соответствует атрибут *Nx*. Значение атрибута *Nx* есть число повторений в *x* соответствующего ему нетерминала. Например, атрибут *Na* соответствует символу *A*. Правило вычисления атрибута *Na* таково:

```
Na(a) = 1
Na(aA) = 1 + Na(A)
```

Атрибуты *Nb* и *Nc* соответствуют символам *B* и *C*, правила их вычисления выглядят аналогично правилу вычисления атрибута *Na*.

Кроме того, надо ввести семантическое правило, обеспечивающее одинаковое количество всех терминальных символов в предложениях языка:

```
Na(A) = Nb(B) = Nc(C)
```

Атрибутная грамматика для языка *L* имеет следующий вид:

```
S ::= ABC      Na(A) = Nb(B) = Nc(C)
A ::=   a      Na(A') := 1
      | aA     Na(A') := 1 + Na(A)
B ::=   b      Nb(B') := 1
      | bB     Nb(B') := 1 + Nb(B)
C ::=   c      Nc(C') := 1
      | cC     Nc(C') := 1 + Nc(C)
```

Как можно построить алгоритм для генерации синтаксически и семантически корректных предложений *L*, основывающийся на его формальном описании в виде атрибутной грамматики?

Правила вычисления атрибутов не могут быть применены, пока не будет готово предложение языка. При построении предложений можно руководствоваться только продукционными BNF-правилами, других указаний для способа генерации предложений такая грамматика не содержит. На Рис. 1 в качестве примера изображено дерево синтаксического разбора синтаксически корректного, но семантически неверного предложения языка *L*.

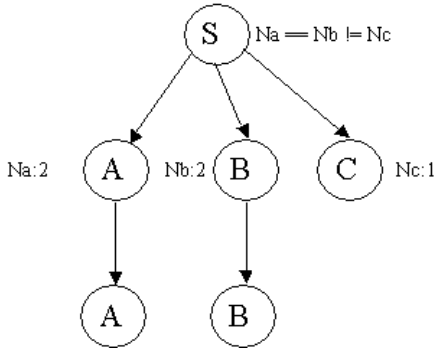

Рис. 1. Дерево синтаксического разбора предложения языка L aabbc.

Поэтому так устроенный алгоритм генерации требует фильтрации построенных предложений для выбора из них только семантически корректных, что эквивалентно написанию семантического анализатора.

Как показывает этот пример, для генерации семантически корректных программ тяжело использовать атрибутные грамматики в классическом виде, поэтому для описания статической семантики предлагается другой формализм, который для определенности будем называть *конструктивной грамматикой*.

3. Конструктивная грамматика

Удобной формой описания синтаксических конструкций языка является представление их в виде абстрактного синтаксиса, т.е. в виде *абстрактных синтаксических деревьев* (далее AST — Abstract Syntax Tree) [1]. Такое дерево может быть описано, например, с помощью TreeDL-нотации [19]. Далее для краткости будем называть абстрактные синтаксические деревья просто синтаксическими деревьями.

Анализируя статическую семантику языков программирования, можно заметить, что семантические ограничения описывают связи между узлами синтаксических деревьев, например, связь между описанием переменной или функции и их использованием. Если для одного из этих узлов известны его атрибуты, то в семантически корректной программе можно вычислить атрибуты для второго узла. Например, если в описании переменной известен

атрибут, соответствующий имени переменной, то известен атрибут-имя для синтаксической конструкции ее использования.

Для определенности будем называть узлы синтаксического дерева, связанные семантическим правилом, *источником* и *целью*. Под источником понимается узел, содержащий информацию, достаточную для определения атрибутов второго узла, который является целью.

Будем говорить, что семантическое правило *Sr относится к узлу T*, если *T* является целью для *Sr*. Чтобы программа была семантически корректной, для каждого узла синтаксического дерева этой программы должны быть выполнены все семантические правила, которые к нему относятся.

Исходя из вышесказанного, предлагается следующий новый подход к формализации статической семантики языков программирования для задач генерации программ-тестов.

Конструктивной грамматикой будем называть тройку

KG = (G, A, AR), где

G — *контекстно-свободная* грамматика, описывающая синтаксис целевого языка,
A — конечное *множество* атрибутов, использующихся для описания семантики
AR — конечное *множество* абстрактных семантических правил, которые описывают семантику языка.

Абстрактным семантическим правилом $ASr \in AR$ будем называть шестерку

ASr = (St, A(St), Tt, A(Tt), P, F), где

St — тип узла-источника дерева абстрактного синтаксиса,
A(St) — атрибуты источника,
Tt — тип узла-цели дерева абстрактного синтаксиса,
A(Tt) — атрибуты цели,
P — множество параметров семантического правила *ASr*,
F — семантическая функция, связывающая атрибуты цели с атрибутами источника.

В отличие от атрибутных грамматик, где семантические функции описываются, как $X_i.a = f(y_1,...y_k), k \geq 0$, где $1 \leq i \leq n$, и $\forall j\ 1 \leq j \leq k\ y_j$ — это атрибут в правиле вывода p: $X_0 \to X_1...X_n (\mathrm{n} \geq 0)$, в конструктивных грамматиках будем записывать семантические функции, как

Tt.a = f (a1St,…, akSt), где $k \geq 0$,

$\forall j\ 1 \leq j \leq k\ a_j St \in A(St)$ — множество независимых атрибутов источника,

$Tt.a \in A(Tt)$ — множество зависимых атрибутов цели,
St — тип узла-источника дерева абстрактного синтаксиса,
Tt — тип узла-цели дерева абстрактного синтаксиса,

$f \in F$ — семантическая функция.

Семантические правила, в отличие от их абстрактных описаний, соответствуют синтаксическим деревьям конкретных программ на целевом языке. Для семантических правил известны конкретные узлы (соответствующих типов, указанных в абстрактном описании), для которых должно выполняться это семантическое правило.

Семантическим правилом будем называть пятерку

Sr = (S, A(S), T, A(T), ASr), где

S — источник, узел синтаксического дерева программы,
A(S) — значения атрибутов источника,
T — цель, узел синтаксического дерева программы,
A(T) — значения атрибуты цели,
ASr — абстрактное описание семантического правила *Sr*.

Каждое семантическое правило в конструктивной грамматике определяется двумя узлами синтаксического дерева программы. Один из этих грамматических символов — *цель семантического правила* и обозначается *T* (от английского слова *target*), другой — *источник семантического правила* и обозначается *S* (от английского слова *source*).

Графом семантической зависимости для дерева *Tree* программы *L* называется ориентированный граф, вершинами которого выступают узлы-источники и узлы-цели дерева *Tree* для некоторых правил. Дуги в этом графе строятся так: из вершины *S* существует дуга в вершину *T* тогда и только тогда, когда для дерева *Tree* определено семантическое правило с источником *S* и целью *T*.

Графом атрибутной зависимости для атрибутного дерева *Tree* программы *L* называется ориентированный граф, вершинами которого выступают атрибуты источников и целей множества семантических правил дерева *Tree*. Дуги в этом графе строятся по следующему правилу: из вершины, соответствующей атрибуту *N.b*, существует дуга в вершину, соответствующую атрибуту *N.a*, тогда и только тогда, когда атрибут *N.a* зависит от *N.b*.

Граф семантической зависимости может быть получен из графа атрибутной зависимости путем склейки в графе атрибутной зависимости вершин, соответствующих атрибутам одного и того же узла.

Аналогично граф атрибутной зависимости может быть получен из графа семантической зависимости путем разбиения вершин графа семантической зависимости на группы вершин, представляющих собой зависимые и независимые атрибуты, и проведения между ними соответствующих дуг.

Конструктивная грамматика, в отличие от атрибутной, содержит информацию, которая может быть использована при построении синтаксических деревьев программ-тестов, что позволяет избежать полного перебора всех правил вывода.

Конструктивные грамматики позволяют отказаться от построения таблиц символов, так как вся необходимая для правильного использования имен

информация в них имеется и доступна на момент определения атрибутов очередного грамматического символа.

4. Генерация тестов с использованием конструктивной грамматики целевого языка

Если отталкиваться от формального описания синтаксиса и статической семантики целевого языка программирования, то естественно считать критерием покрытия (для тестирования семантического анализатора) покрытие каждого описанного семантического правила во всех возможных синтаксических контекстах. Это означает, что *для каждого семантического правила Sr во множестве тестов должны встречаться все возможные синтаксические конструкции, на которых действует семантическое правило Sr*. Алгоритм генерации должен обеспечить достижимость этого требования.

Процесс генерации семантических тестов состоит из следующих этапов:

1. Построение синтаксических конструкций, попадающих под действие семантических правил, и формирование из них синтаксических деревьев[3];
2. Достраивание деревьев программ-тестов, полученных на первом этапе, до синтаксической полноты;
3. Определение семантических зависимостей «источник-цель»: для каждой цели ищем источник, если источников не хватает, то достраиваем[4];
4. Определение порядка на множестве семантических функций, соответствующих рассматриваемому синтаксическому дереву[5];
5. Вычисление семантических атрибутов целей.

Далее следует подробное описание процесса семантически управляемой генерации позитивных тестов для тестирования семантического анализатора в компиляторе целевого языка.

Рассмотрим множество $\{Tt_k\}_{k=0...N}$ типов узлов дерева абстрактного синтаксиса целевого языка. Каждый элемент этого множества считаем потенциальной целью (в терминах конструктивной грамматики).

[3] На этом этапе деревья могут быть синтаксически неполными

[4] После первых трех этапов атрибуты абстрактных синтаксических деревьев (например, названия типов, имена переменных), остаются пустыми. Значения для них будут получены на 4 и 5 этапе алгоритма генерации.

[5] Этот порядок необходим для вычисления семантически зависимых атрибутов. Независимые атрибуты для одного семантического правила могут быть зависимыми для другого семантического правила.

Каждому типу узла соответствует множество $R^{Tt_k} = \left\{ R_i^{Tt_k} \in AR \right\}_{i=0...M^{Tt_k}}$ абст-рактных семантических правил, в которых он выступает в качестве цели.

Далее опишем действия, которые генератор повторяет для всех $R_i^{Tt_k}$ каждого Tt_k.

4.1. Построение синтаксического дерева программы-теста

На этапе построения семантически интересного синтаксического дерева тестовой программы входными данными является выбранное абстрактное семантическое правило $R_i^{Tt_k}$, имеющее в качестве:

- типа узла, обозначающего цель — Tt_k,
- типа узла, обозначающего источник — $St_i^{Tt_k}$.

Рассмотрим семантическое правило *"все поля-члены класса должны иметь различные имена"*. Синтаксические узлы, соответствующие полям-членам класса, по синтаксису не могут находиться ни в каком поддереве, кроме поддерева, корнем которого является узел, соответствующий объявлению класса. Для того, чтобы ускорить работу генератора, можно явно указать ему узел, начиная с которого имеет смысл строить цепочки к источнику и цели. Для рассмотренного семантического правила источником и целью будут узлы, обозначающие в грамматике целевого языка определения полей-членов класса, а начальным узлом для построения цепочек к источнику и цели — узел, обозначающий объявление класса.

Для оптимизации алгоритма построения синтаксических деревьев и для придания большей гибкости описанию семантических правил введем новый параметр в абстрактное описание семантического правила, а именно: тип узла, который является корнем поддерева области действия семантического правила. Далее в описании семантических правил будем называть такой узел просто корнем поддерева. Информация о типе *корня поддерева* неявно содержится в неформальном описании каждого семантического правила.

Будем строить синтаксическое дерево тестовой программы по трем вершинам узел-источник, узел-цель, узел-корень синтаксического поддерева.

Естественным способом построения всех видов синтаксически неполных деревьев тестовых программ, содержащих указанные вершины, является нахождение декартова произведения следующих множеств: множества цепочек от стартового узла до корня поддерева, множества цепочек от корня поддерева до источника и множества цепочек от корня поддерева до цели (Рис. 2). В общем случае мощности таких множеств могут быть больше единицы. Тогда для выбранного абстрактного семантического правила может быть построено не одно абстрактное синтаксическое дерево, а целое семейство деревьев.

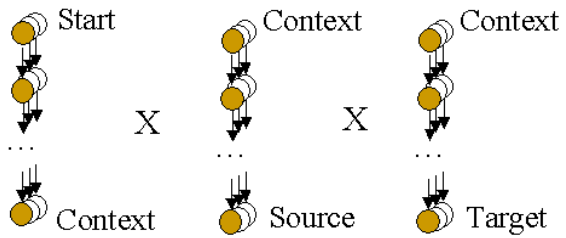

Рис. 2. Декартово произведение множества цепочек от стартового узла до корня поддерева, множества цепочек от корня поддерева до источника и множества цепочек от корня поддерева до цели.

Каждому синтаксическому дереву в этом семействе соответствует множество абстрактных семантических правил, выполнение которых обеспечит семантическую корректность программы, которой соответствует рассматриваемое синтаксическое дерево. Это множество содержит все семантические правила, в которых в качестве целей могут выступать узлы рассматриваемого дерева (Рис. 3). На следующем этапе на основе множества абстрактных семантических правил должно быть построено множество семантических правил, связывающих конкретные узлы синтаксического дерева тестовой программы.

4.2. Построение множества семантических правил синтаксического дерева программы

На предыдущем этапе процесса семантически управляемой генерации тестов были получены синтаксически полные деревья программ-тестов, сопровождаемые множеством абстрактных семантических правил.

Рис. 3. Построение синтаксических каркасов для семантически корректных тестов.

Следующим этапом генерации будет построение из множества абстрактных описаний множеств семантических правил для каждого синтаксического дерева.

В качестве исходных данных рассмотрим множество синтаксически полных деревьев и список абстрактных семантических правил, в соответствии с которыми должны быть вычислены атрибуты узлов деревьев.

Не теряя общности, рассмотрим синтаксическое дерево программы *Tree* и список абстрактных семантических правил *SRList*, выполнение которых обеспечит семантическую корректность. Для того, чтобы программа стала семантически корректной, необходимо правильно определить конкретные семантические правила.

В предыдущем параграфе описывался процесс построения семейства деревьев программ для выбранного семантического правила $R_i^{Tt_k}$. Будем называть такое семантическое правило *основным (main)*. Для основного семантического правила по построению известны узлы источник и цель. Основное семантическое правило становится первым во множестве конкретных семантических правил, соответствующих дереву программы-теста.

Рассмотрим правила заполнения неосновных семантических правил в соответствии с их типом.

Выделяются три типа семантических правил:

1. *"Один к одному"* — каждой цели соответствует только один источник и этот источник не может определять другую цель. Примером может служить требование о соответствии типов выражений слева и справа от оператора суммирования;
2. *"Один ко многим"* — каждой цели соответствует только один источник, который может определять больше одной цели. Например, статическая семантика языка Си позволяет множественное использование переменной, которая описана только один раз;
3. *"Многие ко многим"* — каждой цели должны соответствовать все существующие в данном дереве источники указанного в семантическом правиле типа, каждый из которых должен определять все цели указанного в семантическом правиле типа. Например, каждая конструкция определения переменной должна задавать уникальное имя переменной[6].

Семантические правила типа *"один к одному"* и *"один ко многим"* задают правила вычисления атрибутов цели через атрибуты источника, поэтому источник должен обязательно существовать в дереве.

Семантические правила типа *"многие ко многим"* не требуют обязательного наличия источника (или источников), но если он есть, то правило должно выполняться.

Рассмотрим некоторое абстрактное семантическое правило $R_i \in SRList$. В независимости от его типа находим в дереве все узлы, соответствующие типу узла-цели в семантическом правиле R_i, и все узлы, соответствующие типу узла-источника в семантическом правиле R_i. Найденные узлы дерева будем

[6] Такого рода правила являются псевдоопределениями цели. На самом деле с помощью таких правил описываются требования на уникальности каких-либо значений.

называть *кандидатами* на роль цели и источника соответственно.

Теперь необходимо расставить зависимости между узлами-кандидатами в соответствии с типом семантического правила:

1. Если рассматриваемое правило имеет тип *"многие ко многим"*, то ставим в соответствие по правилу R_i каждому узлу из множества кандидатов на роль цели каждый узел из множества кандидатов на роль источника. Отсутствие кандидатов-источников в дереве не имеет значения, в этом случае никаких зависимостей по правилу R_i не появляется.
2. Если правило отвечает типу *"один ко многим"*, то для каждого кандидата на роль цели подбираем источник из множества кандидатов и устанавливаем между ними зависимость по правилу R_i. Один и тот же источник может использоваться больше одного раза. Если множество кандидатов на роль источника пусто, тогда следует достроить в дереве необходимый узел, который мог бы выступить кандидатом на роль источника[7].
3. Если правило имеет тип *"один к одному"*, то подбор кандидатов осуществляется аналогично пункту 2, с учетом того, что источник может определять только одну цель.

Утверждение 1: Если позволяет синтаксис и семантические ограничения целевого языка, то для любого семантического правила может быть построен новый узел-источник.

Доказательство: Действительно, по свойствам грамматики целевого языка любой узел достижим из стартового узла, и это утверждение может быть нарушено только, если:

1. В качестве корня поддерева семантического правила (то есть корня поддерева, где должны находиться источник и цель) указан узел, из которого источник и цель одновременно недостижимы;
2. Описаны избыточные семантические ограничения для источника, отбрасывающие все возможные варианты.

Но оба эти случая означают, что допущены ошибки в описании правил статической семантики.

4.3. Вычисление атрибутов целей семантических зависимостей

На этом этапе упорядочим все дуги графа семантических правил следующим образом.

Будем называть граф семантической зависимости дерева *Tree упорядоченным*, если его дуги пронумерованы так, что при обходе графа атрибутной зависимости дерева *Tree* в соответствии с указанной нумерацией для каждого семантического правила атрибуты цели не будут посещены раньше атрибутов источника.

Введенный на графе семантических зависимостей порядок определяет порядок вычисления атрибутов целей в семантических правилах.

[7] Это возможно как показано в *утверждении 1.*

Рассмотрим пример, в котором участвуют три семантических правила:

1. Ограничение на повтор имен при описании переменных в одной области видимости: *переменные должны иметь различные имена в одной области видимости* (узлы синтаксического дерева, соответствующие описанию переменных, обозначим *Decl*).
2. Ограничение на использование переменной до ее описания: *переменная должна быть описана перед использованием*.
3. Ограничение на использование не инициализированных переменных: *переменная перед использованием должна быть проинициализирована* (узлы синтаксического дерева, соответствующие использованию переменных, обозначим *VarUse*, инициализации переменных — *Assign*).

Первое семантическое правило связывает узлы *Decl* в отношении «многие ко многим», второе — связывает *Assign* (источник) и *VarUse* (цель) в отношении «один ко многим», третье — *Decl* (источник) и *VarUse* (цель) в отношении «один ко многим». Предположим, что построено синтаксическое дерево, фрагмент которого изображен на Рис. 4.

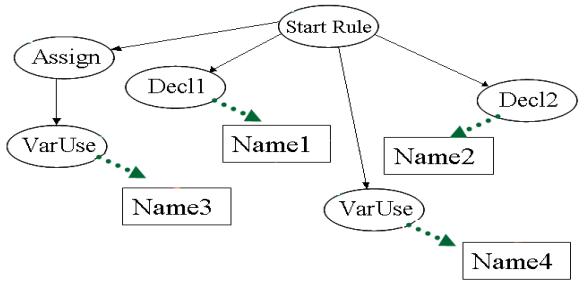

Рис. 4. Фрагмент синтаксического дерева.

Фрагменту синтаксического дерева с Рис. 4 может соответствовать фрагмент кода

```
...
int  <name1>;
float <name2>;
...
<name3> = 0;
...
if (<name4>)
{
...
}
...
```

Namei - узлы обозначают атрибуты соответствующих узлов *(Name1 - Decl1, Name2 — Decl2, Name3 — VarUse — Assign, Name4 — VarUse - ...).* На следующем рисунке (Рис. 5) жирными линиями изображен полученный граф

семантической зависимости, дуги которого пронумерованы, следуя ранее введенным правилам нумерации.

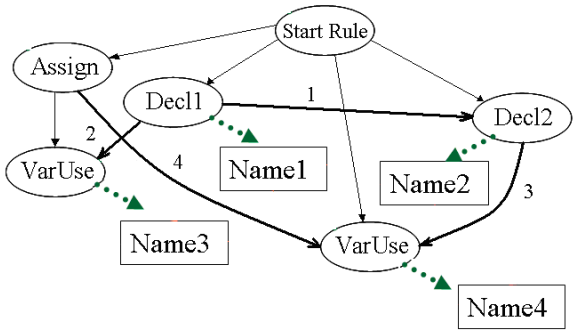

Рис. 5. Фрагмент синтаксического дерева с указанием семантических правил

Правила вычисления атрибутов для синтаксического дерева с Рис. 5 образуют систему (1).

$$\begin{cases} Name2 \neq Name1 \\ Name3 = Name1 \\ Name4 = Name2 \\ Name4 = Name3 \end{cases} \quad (1)$$

Очевидно, что система (1) не имеет решения. Посмотрим на рисунок 6. На нем жирными стрелками обозначены дуги графа атрибутной зависимости. Хорошо виден контур, в котором присутствует дуга, соответствующая правилу запрета эквивалентности значений атрибутов (*Name1* → *Name2*).

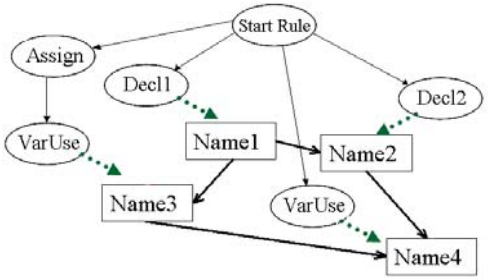

Рис. 6 Фрагмент синтаксического дерева с указанием семантических зависимостей между атрибутами.

Если бы, например, в графе атрибутной зависимости не было контура, или в контуре не присутствовала дуга запрета эквивалентности значений атрибутов, или все дуги контура были бы дугами запрета эквивалентности, тогда система имела бы решение.

Будем называть граф семантических зависимостей *разрешимым*, если система, образованная правилами вычисления атрибутов графа семантических зависимостей, имеет решение.

В случае, когда получена неразрешимая система, необходимо переопределить множество семантических правил для рассматриваемого дерева программы так, чтобы система для вычисления семантических атрибутов была разрешима. Пример такого переопределения приведен на Рис. 7.

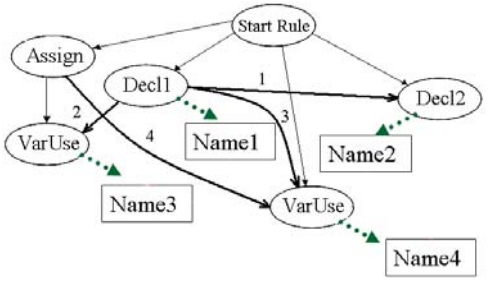

Рис. 7 Фрагмент синтаксического дерева с указанием семантических правил.

Семантическое правило *"переменная должна быть описана перед использованием"* имеет тип "один-ко-многим", поэтому можно использовать узел *Decl1* в качестве источника для определения узла *VarUse* (Рис. 7) вместо узла *Decl2*, как это было раньше (см. Рис. 5). Граф атрибутных зависимостей, соответствующий графу на Рис. 7, образует систему (2):

$$\begin{cases} Name2 \neq Name1 \\ Name3 = Name1 \\ Name4 = Name1 \\ Name4 = Name3 \end{cases} \quad (2)$$

Система (2) имеет решение. Следовательно, граф семантических зависимостей разрешим, и синтаксическое дерево, на котором определен этот граф, может быть преобразовано в корректную программу в соответствии с заданными правилами статической семантики целевого языка.

5. Реализация подхода

Описанная методика построения синтаксически и семантически корректных программ-тестов была реализована в инструменте Stg.

Stg (Semantic Test Generator) — java-инструмент для автоматической генерации семантически корректных тестовых программ на целевом языке программирования. Stg — инструмент, адаптируемый для разных языков программирования. Это значит, что инструмент состоит из неизменяемого ядра

и языково-зависимых частей, большинство из которых генерируются автоматически из формального описания грамматики целевого языка.

Инструмент Stg был использован для формального описания статической семантики языка Си с использованием стандарта языка C99 [16]. В таблице 1 приво-дятся среднестатистические данные по результатам генерации тестов для языка Си.

Количество сгенерированных программ	~ 10000
Средний размер сгенерированных программ	~ 30 (строк кода в одной программе)
Общий объем	~ 10 Mb
Время генерации	~ 60 мин
Время прогона всех программ через gcc-компилятор	~ 15 мин
Покрытие кода фронтенда gcc-компилятора сгенерированными программами	~ 27 % (для измере-ния использовался Clover)

Таб. 1. Результаты генерации Stg (Си)

Stg-алгоритм использует формальное описание целевого языка в качестве исходной информации для создания программ на целевом языке. Синтаксис и семантика целевого языка описываются отдельно.

Для описания абстрактного синтаксиса используется специальная нотация TreeDL[19].

Описание статической семантики задается в специальной нотации SRL (Semantic Relation Language).

Описанная технология генерации тестовых программ на целевом языке позволила реализовать пилотный проект для MPEG-21 [17]. MPEG-21 определяет XML-схемы, описывающие метаданные и семантические правила для их обработки. Для проведения пилотного проекта были предложены IPMP XML-схема и стандарт, неформально описывающий семантические требования [24]. STG-технология позволила формализовать описание семантических требований, XML-схема была переведена в TreeDL-представление, после чего автоматически было сгенерировано множество XML-документов, соответствующих IPMP — описанию.

В заключение, отметим, что работа над автоматизацией семантически управляемой генерации с использованием конструктивных грамматик продолжается. Как показали последние эксперименты, описанная технология позволяет создавать также и “негативные” тесты.

Литература

1. А.Ахо, Р.Сети, Д.Ульман. Компиляторы: принципы, технологии, инструменты // Москва–Санкт-Петербург–Киев, 2001.
2. Knuth D.E. Semantics of Context-Free Languages, Mathematical Systems Theory, 2, 2, 1968, 127-146.

3. Knuth D.E. Semantics of Context-Free Languages: Correction, Mathematical Systems Theory, 5, 1, 1971, 179.
4. de Bakker J.W., Formal definition of programming languages, with an approach to the definition of ALGOL 60, Math. Cent. Tracts 16. Mathematical Centrum. Amsterdam, 1967.
5. Landin P.J., “The mechanical evaluation of expressions’, Comp. J. 6 (1964), 308-320.
6. Landin P.J., A formal description of ALGOL 60, Formal Language Description for Computer Programming, pp. 266-294, Proc. IFIP working Conf. Vienna (1964), North Holland, 1966.
7. Landin P.J., A correspondence between ALGOL 60 and Church’s lambda notation, Comm. ACM 8 (1965), 89-101, 158-165.
8. Bohm C., The CUCH as a formal and description language, Formal language description languages for computer programming, pp. 266-294, Proc. IFIP working Conf. Vienna (1964), North Holland, 1966.
9. Corrado Bohm and Wolf Cross, “Introduction to the CUCH”, Automata Theory (ed/ by Caianiello E.R.), pp. 35-65, Academic press, 1966.
10. McCarthy J., A formal definition of a subset of ALGOL, Formal Language Description for Computer Programming, pp. 1-12, Proc. IFIP working Conf. Vienna (1964), North Holland, 1966.
11. McCarthy J., Painter J., Correctness of a compiler for arithmetic expressions, Proc. Sympos. Appl. Math., Vol 17, Amer. Math. Soc., Providence, R.I., 1967.
12. Wirth N., Weber H., Euler: A generalization of ALGOL, and its formal definition, Comm. ACM 9 (1966), 11-23, 89-99, 878.
13. Elgot C.C., “Machine species and their computation languages”, Formal Language Descripton Languages for Computer Programming, pp. 160-179, IFIP Working Conf., Vienna (1964), North Holland, 1966.
14. Elgot C.C., Robinson A., “Random-access, stored program machines, an approach to programming languages”, J. ACM 11 (1964), 365-399.
15. http://dinosaur.compilertools.net/
16. ISO/IEC 9899 - Programming languages — C. http://wwwold.dkuug.dk/JTC1/SC22/WG14/www/C99RationaleV5.10.pdf
17. http://www.iso.ch/iso/en/prods-services/popstds/mpeg.html
18. Ю.В.Баскаков. Принципы построения тестовых комплектов для тестирования конформ-ности компиляторов стандартам языков программирования. – в сб. трудов под ред. В.А. Сухомлина ‘Теоретические и прикладные проблемы ИТ’ Москва, ВМиК МГУ, 2001
19. http://sourceforge.net/projects/treedl
20. Kastens, U. "Attribute Grammars as a Specification Method", Proceedings of the International Summer School on Attribute Grammars, Lecture Notes in Computer Science 545, Springer-Verlag, New York-Heidelberg-Berlin, 1991a, 16-47.
21. Kurt M. Bischoff. 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
22. http://www.swif.uniba.it/lei/foldop/foldoc.cgi?abstract+syntax
23. PL/I-Definition Group of the Vienna Laboratory "Formal Definition of PL/I (U.L.D. No. 3)" Tech. Rep. TR 25.071, IBM Laboratory Vienna, Vienna, Austria, Dec. 1966.
24. ISO/IEC 13818-11:2004 Information technology -- Generic coding of moving pictures and associated audio information -- Part 11: IPMP on MPEG-2 systems; http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=37680&COMMID=&scopelist=