Глава 7

Формальная семантика языков программирования

Эта глава отведена обсуждению формальных подходов к описанию семантики языков программирования. Вначале поясняются синтезируемые атрибуты и атрибутные грамматики, которые являются расширениями обычных, синтаксических грамматик на область семантики. Затем дается характеристика операционной семантики, максимально задействующей воображение человека в роли виртуальной машины. Значительное место и внимание уделено аксиоматической семантике — наиболее развиваемому на практике подходу к автоматическому доказательству корректности программы. Рассматривается принятая в нем схема доказательства, использование правил вывода, опирающихся на постусловия и предусловия; приводятся примеры подобных правил для наиболее популярных конструкций языка. В частности, описывается понятие инварианта цикла и исследуются методики его определения. В заключение объясняются основные идеи денотационного подхода к заданию семантики, где во главу угла поставлены рекурсивные функции отображения отдельных конструкций языка в область семантических значений. Демонстрируются примеры функций семантического отображения для чисел, выражений, операторов присваивания и циклов.

Семантика языка программирования

Считается, что в настоящее время средства для определения синтаксиса языка достаточно удобны и развиты. Увы, но этого нельзя сказать о средствах формального представления семантики языка. Понятно, что для использования языка программирования следует описать каждую конструкцию языка в отдельности, а также ее применение в совокупности с другими конструкциями. В языке существует множество различных конструкций, точное определение которых необходимо как программисту, применяющему язык, так и разработчику компилятора для этого языка. Программисту эти знания позволяют прогнозировать вычисления, производимые операторами программы. Разработчику описания конструкций необходимы для создания правильной реализации компилятора.

Обычно семантика определяется с помощью обычного текста. Сначала при помощи какой-либо грамматики (скажем, BNF-грамматики) описывается синтак-

сис конструкции, а затем приводятся примеры и комментарии для разъяснения семантики. Очень часто смысл комментариев неоднозначен, поэтому различные читатели воспринимают его по-разному. Программист может получить ошибочное представление о работе создаваемой программы, а разработчик может сформировать ошибочную реализацию некоторой конструкции языка. Для исключения ошибок нужен инструмент для ясного, точного и лаконичного определения семантики всего языка.

Приемлемое решение в отношении семантики языка найти чрезвычайно трудно. Причина кроется в том, что содержание, смысл конструкций во много раз сложнее их формы. Был разработан целый ряд методов для формального определения семантики. Приведем описания наиболее популярных из них.

Синтезируемые атрибуты

Ранние попытки точно задать семантику делались путем добавления расширений к BNF-грамматике языка программирования. Дополнительную информацию о семантике можно было извлечь из дерева грамматического разбора. Идея, предложенная Дональдом Кнутом, заключалась в том, чтобы сопоставить каждому узлу дерева разбора программы некоторую функцию, определяющую семантику этого узла [70]. Подобная информация сохранялась в так называемых атрибутах.

Семантика конструкции может представляться некоторой величиной или набором величин, связанных с конструкцией. Например, семантика выражения `3 + 4` может быть целым значением `7`, типом `int` или строкой `+ 3 4`.

Величина, ассоциированная с конструкцией, называется *атрибутом*. Атрибут *a* для *X* будем записывать как *X.a*, где *X* считается нетерминалом или терминалом грамматики. *E.val* рассматривается как ссылка на атрибут *val* выражения *E*. На рис. 7.1 все символы грамматики имеют по атрибуту *val*. В общем случае каждый символ грамматики может иметь несколько атрибутов.

Атрибуты имеют значения. Точнее, каждое вхождение атрибута в дерево разбора имеет значение. Атрибут в корне дерева разбора на рис. 7.1 имеет значение `11`. Атрибут **num**.*val* слева внизу рисунка имеет значение `7`.

ПРИМЕЧАНИЕ

> В этой главе будем использовать упрощенное обозначение нетерминалов и терминалов грамматики. Нетерминалы будем записывать курсивом (без угловых скобок), а терминалы — полужирным шрифтом.

Будем полагать, что атрибуты для терминальных символов поступают вместе с символами. Когда лексический анализатор распознает на входе знак **num**, он также определяет значение знака — предполагаем, что это значение «приходит» вместе с **num** в атрибуте **num**.*val*.

Значения атрибутов для нетерминалов определяются семантическими правилами, добавленными к правилам подстановки грамматики. В этом разделе мы будем использовать синтезируемые атрибуты. Атрибут *N.a* называется *синтези-*

руемым, если правила, определяющие *N.a*, добавляются к правилам подстановки, в которых *N* появляется в левой части правила. Иными словами, синтезируемый атрибут в узле *N* дерева разбора определяется только с использованием значений атрибутов в дочерних по отношению к *N* узлах и в самом узле *N*. Семантическое правило, определяющее *N.a*, записывается как присваивание атрибуту *N.a* некоторого значения.

Использование синтезируемых атрибутов известно как синтаксически управляемый подход к определению семантики. Спецификация синтаксиса совместно со связанными с ней семантическими правилами называется *синтаксически управляемым определением*.

Синтаксически управляемое определение в табл. 7.1 основывается на грамматике для выражений. Каждый символ грамматики имеет синтезируемый атрибут *val*. С каждым правилом подстановки связано одно семантическое правило (это правило записано в той же строке). Здесь T и T_1 обозначают ссылки на два вхождения одного и того же нетерминального символа в правиле.

Аннотированное дерево разбора имеет атрибуты, добавленные к узлам. Аннотированное дерево разбора на рис. 7.1 создано применением правил подстановки и семантических правил из табл. 7.1.

Таблица 7.1. Синтаксически управляемое определение для вычисления арифметических выражений

Правило подстановки	Семантическое правило
$E ::= E_1 + T$	$E.val := E_1.val + T.val$
$E ::= E_1 - T$	$E.val := E_1.val - T.val$
$E ::= T$	$E.val := T.val$
$T ::= T_1 * F$	$T.val := T_1.val * F.val$
$T ::= F$	$T.val := F.val$
$F ::= (E)$	$F.val := E.val$
$F ::= \mathbf{num}$	$F.val := \mathbf{num}.val$

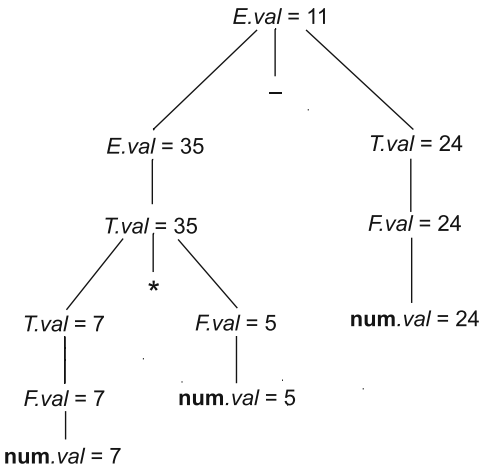

Рис. 7.1. Аннотированное дерево разбора с атрибутами

Листья дерева разбора помечены терминалами. Терминалы тоже могут иметь атрибуты; правда, семантических правил для определения значений терминалов нет. При обходе левой ветви дерева разбора снизу вверх правило

```
F.val := num.val
```

определяет, что значение атрибута *val* для самого левого узла *F* равно 7. Правило

```
T.val := F.val
```

связанное с правилом подстановки $T ::= F$, копирует это значение, так что *T.val* тоже равно 7.

Более высокий, родительский узел *T* генерирует правило подстановки

```
T ::= T1 * F
```

поэтому его значение является произведением значений T_1 и *F*. Следовательно, левый дочерний узел корня имеет значение 35. Семантическое правило, связанное с правилом подстановки для корня, определяет, что значение *E* должно быть разностью значений ее детей.

Порядок вычислений

При использовании синтезируемых атрибутов информация передается вверх по дереву разбора. Значение атрибута узла определяется в терминах атрибутов ее дочерних узлов. Именно поэтому значения атрибутов на рис. 7.1 могут быть вычислены обходом дерева от листьев к корню.

Выводы

Предусматривается следующий порядок использования синтезируемых атрибутов для определения смысла конструкций:

1. *Атрибуты* — атрибуты добавляются к символам грамматики.
2. *Семантические правила* — семантические правила добавляются к правилам подстановки. Синтезируемый атрибут *N.a* определяется по семантическому правилу, добавленному к такому правилу подстановки, в левой части которого находится *N*.

Атрибутные грамматики

Атрибутные грамматики можно рассматривать как обобщение аппарата синтезируемых атрибутов [70]. Они позволяют осмысливать конструкции в зависимости от окружающего контекста. В этом случае значения атрибутов передаются как вверх, так и вниз по дереву разбора. Дело в том, что смысл узла на дереве разбора может зависеть не только от его поддерева, но и от информации из другой части дерева.

Предусматривается следующий порядок использования атрибутных грамматик для определения смысла конструкций:

1. *Атрибуты* — атрибуты добавляются к символам грамматики. Для каждого атрибута определяется, является ли он синтезируемым или наследуемым.

2. *Семантические правила* — семантические правила добавляются к правилам подстановки.
3. Если нетерминал *N* появляется в левой части правила подстановки, то добавляемые семантические правила определяют *синтезируемые* атрибуты для *N*.
4. Если нетерминал *A* появляется в правой части правила подстановки, то добавляемые семантические правила определяют *наследуемые* атрибуты для *A*. Именно наследуемые атрибуты инициируют передачу информации вниз по дереву разбора или по горизонтали этого дерева (в этом случае говорят о передаче информации от «братских» узлов).

Рассмотрим проблему трансляции десятичных чисел в диапазоне от 0 до 99 в русские фразы (табл. 7.2).

Таблица 7.2. Трансляция десятичных чисел

Число	Фраза
0	Ноль
1	Один
10	Десять
19	Девятнадцать
20	Двадцать
29	Двадцать девять
30	Тридцать
31	Тридцать один

Трансляция `тридцать один` для числа `31` строится на основе фразы `тридцать`, трансляции цифры `3` в левой позиции числа и трансляции цифры `1` в правой позиции числа. Однако есть исключения из этого правила. Трансляцией числа `30` является `тридцать`, а не `тридцать ноль`. Трансляцией числа `19` является фраза `девятнадцать`, а не `десять девять`.

Проиллюстрируем применение наследуемых атрибутов для трансляции чисел от 0 до 99, генерируемых по следующей грамматике:

```
N :: = D|D P
P ::= D
D :: = 0|1|2|3|4|5|6|7|8|9
```

Дерево разбора для числа 68 представлено на рис. 7.2, *а*.

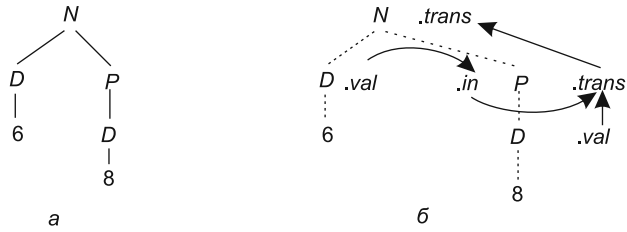

Рис. 7.2. Дерево разбора для числа 68

Нетерминал *D* имеет синтезируемый атрибут *val*, который представляет значение цифры, генерируемой по правилу подстановки для *D*. Нетерминал *N* имеет синте-

зируемый атрибут *trans*, который дает трансляцию числа, генерируемую по правилу для *N*. Нетерминал *P* имеет наследуемый атрибут *in* и синтезируемый атрибут *trans*.

Стрелки на рис. 7.2, *б* иллюстрируют, как синтезируемая для узла *P* трансляция представляется в виде атрибута *P.trans*. Для создания *P.trans* используются значения обеих цифр. С правилом подстановки *P* ::= *D* ассоциируется семантическое правило, которое определяет *P.trans* в терминах *P.in* (значения левой цифры) и *D.val*. На рис. 7.2, *б P.in* = 6 и *D.val* = 8, а *P.trans* является фразой для 68 = 10 * *P.in* + *D.val*.

Полная атрибутная грамматика представлена в табл. 7.3. С правилом подстановки *N* :: = *D P* связаны два семантических правила:

```
P.in := D.val
N.trans := P.trans
```

Словом, *P.in* наследует значение *D.val* от левой цифры, а *P.trans* становится трансляцией *N.trans* (почти волшебным образом превращается в эту трансляцию).

Таблица 7.3. Атрибутная грамматика

Правило подстановки	Семантические правила
`N ::= D`	`N.trans := spell(D.val)`
`N ::= D P`	`P.in := D.val` `N.trans := P.trans`

`P ::= D`	`P.trans := if D.val = 0 then decade(P.in)`
	`else if P.in ≤ 1 then spell(10*P.in + D.val)`
	`else decade(P.in) • spell(D.val)`
	здесь точка • обозначает операцию конкатенации (составления строки из отдельных частей)

`D ::= 0`	`D.val:= 0`
`D ::= 1`	`D.val:= 1`
…	…
`D ::= 9`	`D.val:= 9`

С правилом подстановки *P* :: = *D* связано семантическое правило, которое определяет *P.trans*. Это правило основывается на следующем псевдокоде для трансляции числа *n*:

```
if n is a multiple of 10 then decade(n div 10)
     else if n < 20 then spell(n)
          else decade(n div 10) • spell(n mod 10)
```

Функции *spell* и *decade* удовлетворяют следующим уравнениям:

```
spell(1) = один, spell(2) = два, …, spell(19) = девятнадцать
decade(0) = ноль, decade(1) = десять, …, decade(9) = девяносто
```

С правилом подстановки *P* :: = *D* связано следующее правило:

```
P.trans := if D.val = 0 then decade(P.in)
           else if P.in ≤ 1 then spell(10*P.in + D.val)
                else decade(P.in) • spell(D.val)
```

Синтезируемый атрибут *P.trans* определяется в терминах наследуемого атрибута *P.in* и синтезируемого атрибута *D.val*.

Наследуемые атрибуты придают потоку информации ясность и недвусмысленность. Впрочем, без них можно обойтись, поскольку все, что определяется с помощью

наследуемых и синтезируемых атрибутов, можно определить с использованием лишь синтезируемых атрибутов. Например, атрибутная грамматика из табл. 7.3 может быть модифицирована в эквивалентную грамматику, применяющую только синтезируемые атрибуты. Мы ввели эту грамматику лишь затем, чтобы проиллюстрировать, как можно использовать наследуемые атрибуты.

Операционная семантика

Операционное определение языка программирования описывает выполнение программы, составленной на данном языке, средствами виртуального компьютера [86]. Виртуальный компьютер определяется как абстрактный автомат. Внутренние состояния этого автомата моделируют состояния вычислительного процесса при выполнении программы. Автомат транслирует исходный текст программы в набор формально определенных операций. Этот набор задает переходы автомата из исходного состояния в последовательность промежуточных состояний, изменяя значения переменных программы. Автомат завершает свою работу, переходя в некоторое конечное состояние. Таким образом, здесь идет речь о достаточно прямой абстракции возможного использования языка программирования.

Итак, операционная семантика описывает смысл программы путем выполнения ее операторов на простой машине-автомате. Изменения, происходящие в состоянии машины при выполнении данного оператора, определяют смысл этого оператора.

С одной стороны, машина-автомат способна воспринимать задания, представляемые лишь на простом языке моделирования. С другой стороны, современный компьютер может служить универсальным интерпретатором такого языка. Подобный интерпретатор может быть реализован программой, которая становится виртуальной машиной для языка моделирования.

Таким образом, в операционной семантике требуются:

- транслятор, преобразующий операторы исходного языка программирования **L** в команды низкоуровневого языка моделирования;
- виртуальная машина (ВМ) для языка моделирования.

Например, язык моделирования может представляться набором простых команд, записанных в табл. 7.4.

Таблица 7.4. Язык моделирования виртуальной машины

Команды виртуальной машины
`ident := var`
`ident := ident + 1`
`ident := ident – 1`
`goto  label`
`if var relop var goto label`
`ident := var bin_op var`
`ident := un_op var`

Здесь приняты следующие обозначения: `var` — переменная, `ident` — имя переменной, `goto label` — команда безусловного перехода на символический адрес `label`,

`relop` — обозначение одной из операций сравнения (<, >, =, <>), `bin_op` — операция над двумя аргументами, `un_op` — операция над одним аргументом. Условимся, что символический адрес команды (метку) будем отделять от команды двоеточием.

Положим, что нам нужно выяснить смысл оператора `for` для языка программирования C. Мы представляем этот оператор с помощью некоторой программы для виртуальной машины, приведенной в табл. 7.5.

Таблица 7.5. Программа для виртуальной машины

Оператор языка C	Команды виртуальной машины
`for (expr1; expr2; expr3)`	`expr1;`
`{`	`loop: if expr2 = 0 goto out`
`...`	`...`
`}`	`expr3;`
	`goto loop;`
	`out: …`

После этого запускаем нашу виртуальную машину. Смысл оператора определяют изменения в состоянии виртуальной машины, происходящие при выполнении команд — результата трансляции `for`-оператора. Подобные программы могут составляться и для последовательности операторов. Главное, чтобы имеющийся транслятор был способен обрабатывать каждый из используемых операторов.

ПРИМЕЧАНИЕ

Человек, читающий программы для виртуальной машины, тоже может считать себя виртуальным компьютером, то есть полагают, что он может правильно выполнять «команды» и распознает эффекты «выполнения».

В 1972 году была разработана операционная модель языка под названием Vienna Definition Language (VDL) — это метаязык, предназначенный для описания других языков. В ее состав входили транслятор и виртуальная машина. С помощью данной модели был описан язык программирования PL/1.

Аксиоматическая семантика

Данный подход основывается на применении аппарата исчисления предикатов и теории доказательств. Семантику каждой конструкции языка определяют как некий набор аксиом или правил вывода, используемый для вывода результатов выполнения этой конструкции. Чтобы понять смысл всей программы (то есть разобраться, что и как она делает), эти аксиомы и правила вывода следует применять так же, как при доказательстве обычных математических теорем. Аксиомы и правила вывода используют для оценки значений переменных после выполнения каждого оператора программы. В итоге, когда программа выполнена, формируется доказательство того, что вычисленные результаты удовлетворяют заданным ограничениям на их значения относительно входных значений. Словом, доказывается,

что выходные данные являются корректными результатами вычисления функций, обрабатывающих соответствующие входные данные. Примером описанного подхода считается метод *аксиоматической семантики*, созданный Тони Хоаром [66–68].

Итак, метод аксиоматической семантики служит для доказательства правильности программы. Доказательство правильности демонстрирует, что программа выполняет вычисления, описываемые ее спецификацией. При доказательстве каждый оператор окружается логическими выражениями (условиями), которые задают ограничения на программные переменные. Они используются для определения смысла оператора. Программа представляется в виде *хоаровских* троек (рис. 7.3).

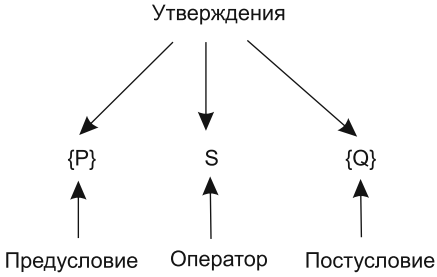

Рис. 7.3. Хоаровская тройка программы

Для получения тройки к каждому оператору нужно добавить слева предусловие, а справа — постусловие. Условия (утверждения, высказывания) записываются в фигурных скобках.

Предусловие описывает ограничения на программные переменные перед выполнением оператора и обозначается как `{P}`.

Постусловие задает новые ограничения на эти переменные после выполнения оператора и обозначается как `{Q}`.

Принято: предусловие вычисляется на основе постусловий операторов.

Среди множества предусловий особую роль играет слабейшее предусловие.

Слабейшее предусловие — это наименьшее ограничивающее предусловие, которое гарантирует правильность связанного постусловия.

Рассмотрим пример хоаровской тройки:

```
{x > 10}   result := 5 * x + 1   {result > 1}
{x > 50}
{x > 1000}
```

В примере записано, что при правильной работе оператора `result := 5 * x + 1` на результат накладывается ограничение `result > 1`.

Здесь перечислены три возможных предусловия, налагающих разные ограничения на значения «входной» переменной `x`, но слабейшее предусловие имеет следующий вид:

```
{x > 0}  --  слабейшее предусловие
```

Слабейшее предусловие все еще обеспечивает корректную работу оператора, при котором соблюдается постусловие, но при этом предоставляет наибольшую «свободу» переменной из предусловия. Ослабление предусловия можно проиллюстрировать рис. 7.4.

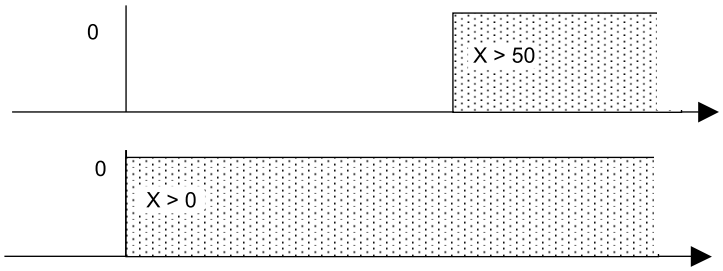

Рис. 7.4. Ослабление предусловия

Здесь показана числовая ось, на которой штриховкой обозначены области разрешенных значений для переменной x. Видим, что слабейшему предусловию соответствует максимальная область допустимых значений.

В аксиоматической семантике принята следующая схема доказательства программы:

1. Слабейшее предусловие оператора вычисляется на базе его постусловия.
2. Выполняется обратный проход по программе — доказательство начинается с последнего оператора, завершается первым оператором.
3. Первое предусловие задает условия, при которых программа формирует последнее постусловие (желаемые результаты).

Для некоторых операторов вычисление слабейшего предусловия выполняется просто и может быть определено аксиомой. В большинстве случаев слабейшее предусловие нужно вычислять по правилу вывода.

Аксиома — это логическое утверждение, принимаемое без доказательства в силу непосредственной убедительности.

Правило вывода — это метод получения истинного утверждения на основе других утверждений.

Аксиоматическая семантика требует, чтобы для каждого оператора языка программирования была определена или аксиома, или правило вывода.

Аксиома присваивания

Эта аксиома «обслуживает», то есть позволяет «доказать» оператор присваивания.

Пусть `x := E` — оператор присваивания с постусловием `Q`. Тогда его предусловие, определяемое по аксиоме

$P = Q_{X \to E}$

означает, что P вычисляется как Q, в котором все вхождения переменной x замещаются на E, то есть на выражение из правой части оператора.

Аксиома записывается в виде следующей тройки:

$\{P = Q_{X \to E}\}$ `x := E` $\{Q\}$

Повторим, что аксиоматическая семантика разрабатывается для доказательства правильности программ. С этой точки зрения оператор присваивания с предусло-

вием и постусловием может рассматриваться как теорема. Если аксиома присваивания при применении к постусловию и оператору вырабатывает «правильное» предусловие, то теорема доказана.

Применение аксиомы присваивания

Рассмотрим примеры применения аксиомы присваивания, то есть примеры доказательства теорем по поводу операторов присваивания.

Пусть исходные данные для доказательства теоремы имеют вид:

```
a := b/2 + 1 {a < 50}
```

Тогда для заданного оператора `a := b/2 + 1` слабейшее предусловие вычисляется подстановкой в постусловие `{a < 50}` вместо `a` выражения `b/2 + 1`:

```
{b/2 + 1 < 50} ⇒ {b < 98}
```

Как использовать этот результат? Достаточно проверить соблюдение предусловия и постусловия в ходе реальных вычислений.

Еще один пример. Для заданной пары

```
y := 5 * x – 10 {y > 95}
```

слабейшее предусловие равно:

```
{5 * x – 10 > 95} ⇒ {x > 17}
```

Правило консеквенции (упрощения)

Рассмотрим хоаровскую тройку `{a > 7} a := a – 2 {a > 0}`. В этом случае утверждение, вырабатываемое по аксиоме присваивания, не равно предусловию. И все же очевидно, что из `{a > 7}` следует `{a > 2}`. Для использования этого факта в доказательстве нам потребуется правило вывода «*правило консеквенции*», иначе называемое *правилом упрощения*.

Общая форма правила вывода имеет вид

$$\frac{S1, S2, ..., SN}{S},$$

где утверждается, что если исходные высказывания $S1$, $S2$, ..., SN истинны, то может быть выведена истина для заключения S.

Правило *консеквенции* записывается в форме

$$\frac{\{P\}S\{Q\}, P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\}S\{Q'\}},$$

где символ $\Rightarrow$ обозначает «следует», а S — любой оператор программы.

Правило означает: если логическое высказывание $\{P\}\ S\ \{Q\}$ — истина, а из утверждения P' следует утверждение P и из утверждения Q следует утверждение Q', то может быть выведено истинное логическое высказывание $\{P'\}\ S\ \{Q'\}$.

Это правило говорит, что *постусловие* может быть всегда *ослаблено*, а *предусловие* может быть всегда *усилено*.

Такое ослабление/усиление очень полезно в доказательстве программ. Например, это позволяет завершить доказательство начального логического высказывания.

Примем P = `{a > 2}`, P' = `{a > 7}`, $Q = Q'$ = `{a > 0}`,
тогда

$$\frac{\{a > 2\}\ \ a := a - 2\ \ \{a > 0\}, \{a > 7\} \Rightarrow \{a > 2\}, \{a > 0\} \Rightarrow \{a > 0\}}{\{a > 7\}\ \ a := a - 2\ \ \{a > 0\}}.$$

Первое исходное утверждение `{a > 2} a := a – 2 {a > 0}` доказывается по аксиоме присваивания. Второе и третье утверждения очевидны. Следовательно, по правилу консеквенции, такое упрощение является истинным.

Правило вывода для последовательности

Слабейшее предусловие для последовательности операторов не может описываться аксиомой, так как предусловие зависит от конкретных разновидностей операторов в последовательности. В этом случае предусловие может быть описано только с помощью правила вывода.

Пусть $S1$ и $S2$ — два смежных оператора, имеющих следующие пред- и постусловия: $\{P1\}\ S1\ \{P2\}$, $\{P2\}\ S2\ \{P3\}$. Отметим, что постусловие первого оператора совпадает с предусловием второго. Логически это вполне объяснимо: первый оператор формирует данные для успешного старта второго оператора. Тогда правило вывода для последовательности из двух операторов принимает вид:

$$\frac{\{P1\}\ \ S1\ \ \{P2\}, \{P2\}\ \ S2\ \ \{P3\}}{\{P1\}\ \ S1; S2\ \ \{P3\}}.$$

Для данного примера $\{P1\}\ S1; S2\ \{P3\}$ правило вывода предписывает аксиоматическую проверку последовательности операторов $S1$; $S2$: они должны иметь общее утверждение, иначе ограничение на переменные, — это $\{P2\}$. Мало того, предусловие первого оператора должно стать общим предусловием последовательности, а постусловие второго — общим постусловием.

Правило вывода для последовательности имеет очень большое практическое значение. Именно оно обеспечивает цепочку автоматического доказательства операторов программы. Для запуска цепочки достаточно знать лишь требуемый результат программы, то есть постусловие последнего оператора. При доказательстве последнего оператора будет вычислено его слабейшее предусловие. Это даст возможность перейти к доказательству предпоследнего оператора. Ведь правило вывода для последовательности гласит: предусловие последнего оператора равно постусловию предпоследнего. Далее шаги перехода от оператора к оператору повторяются. Процесс завершается доказательством первого оператора программы, то есть вычислением его слабейшего предусловия, которое определяет ограничения на исходные данные программы.