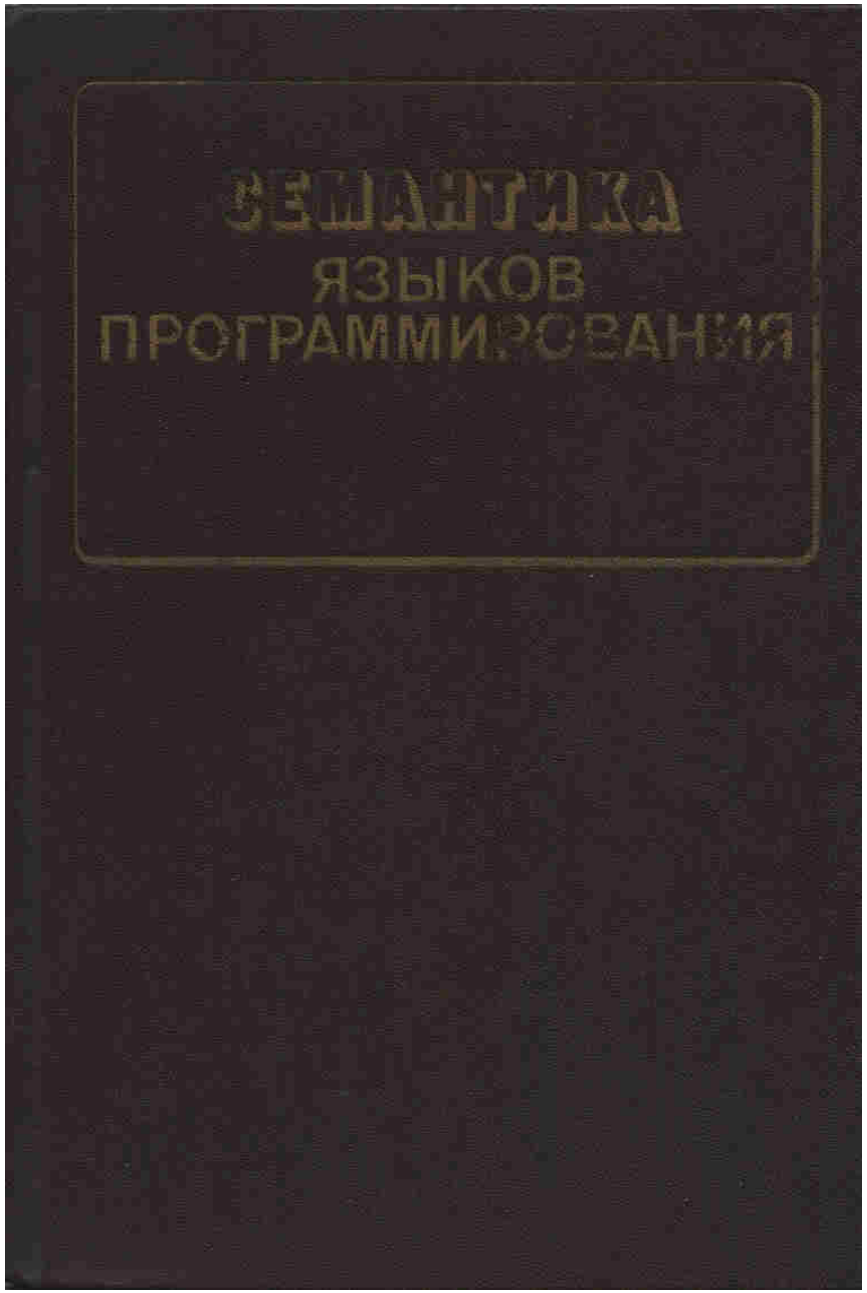


СЕМАНТИКА ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

СБОРНИК СТАТЕЙ

Перевод с английского
А. Н. Бирюкова
и
В. А. Серебрякова

под редакцией
В. М. Курочкина

ИЗДАТЕЛЬСТВО «МИР»
Москва 1980

УДК 681.142.2

Сборник содержит статьи зарубежных авторов по одному из интересных и важных разделов теории формальных языков — проблеме задания семантики. Среди авторов статей известные специалисты Д. Кнут, К. Хоор, П. Льюис, Д. Розенкранц, Д. Стирнз, Дж. Донаху и др. Успехи в этой области позволяют существенно улучшить качество многих компонентов математического обеспечения вычислительных машин, особенно тех, которые связаны с реализацией языков программирования.

Книга представляет значительный интерес для исследователей, разрабатывающих языки программирования, а также для аспирантов и студентов, специализирующихся в программировании.

Редакция литературы по математическим наукам

2405000000

$\text{С}\frac{20205\text{-}030}{041(01)\text{-}80}30-80$

От редактора перевода

Из большого круга вопросов, которые охватывает системное программирование, языки программирования — по-видимому, наиболее продвинутый в теоретическом отношении раздел. Формализация и использование математического аппарата начались здесь чуть ли не с появления первых более или менее сложных языков программирования. И с самого начала очень большое внимание стало уделяться проблеме точного описания (или определения) языка.

Важность этого вопроса с точки зрения практики не вызывает сомнений. Так называемым «пользователям», описывающим на том или ином языке программирования алгоритмы решения своих задач, необходимо понимать, правильно применять и записывать конструкции этого языка. Системным программистам точность описания нужна прежде всего при реализации языка на ЭВМ — вид и смысл языковых конструкций должны быть такими, какими их задумал и определил автор языка, и следовательно, так же должны понимать его и «заложить» в вычислительную машину реализаторы языка. Возникает, однако, целая серия каверзных вопросов. Где гарантия того, что пользователь правильно понял определение языка? Как проверить, соответствует ли запись на данном языке тому алгоритму, который был применен для решения задачи, т. е. правильна ли программа? Еще более сложные вопросы можно поставить перед системным программистом. Можно ли ему верить, если он утверждает, что понял описание языка и точно реализовал его? Получатся ли одинаковые результаты, если одну и ту же программу выполнить на разных машинах, т. е. воспользоваться разными реализациями одного и того же языка?

Пытаться ответить на эти и подобные им вопросы можно только располагая соответствующим формальным аппаратом. Если этот формальный аппарат к тому же и достаточно совершенен, то он поможет решить многие сложные задачи, до сих пор не решенные в системном программировании. Пусть, например, определен некоторый язык (т. е. задана некоторая совокупность формул), определена вычислительная машина (в принципе, посредством формул того же типа, описывающих систему команд машины, т. е. ее входной язык); если все точно определено, то значит можно механически получить компилятор с данного языка на данную машину или, например, осуществить перевод программ с одной машины на другую и т. п. Представляется, однако, что до этого еще очень далеко. Если подобные задачи сохранят свою актуальность в будущем, то решены они будут не скоро. Теория языков программирования сделала в этом направлении только первые шаги.

Наиболее значительную историю имеет задача формализации описания синтаксиса языка. Аппарат бэкусовских формул, используемых для задания контекстно-независимых свойств языка, получил широкое распространение, и по вопросам этого круга имеется богатая литература (в том числе уже и учебного характера). Несколько сложнее задача определения контекстно-зависимых свойств языка. Хотя ее также можно отнести к синтаксису, многие из аппаратов, предложенных для ее решения, в равной степени применимы для задания более сложных свойств языка, связанных

с определением смысла языковых конструкций, т. е. с тем, что принято квалифицировать как *семантику языка.* В настоящем сборнике представлены работы, характеризующие ряд подходов к задаче формального определения семантики языков программирования.

Сборник начинается обзорной статьей Маркотти, Ледгарда и Бохмана, в которой кратко описываются четыре метода задания семантики (W-грамматики, системы продукций, венский метод и атрибутные грамматики). Все эти методы используются для определения семантики одного и того же языка, после чего проводится сопоставление этих описаний, их сравнительный анализ и оценка. С оценкой методов, даваемой авторами, по-видимому, согласится не каждый — здесь неизбежно сказывается субъективный подход, вкусы, привычки и т. п.; тем не менее работа читается с большим интересом, наводит на размышления и помогает лучше осмыслить смежные вопросы.

По венскому методу и W-грамматикам на русском языке уже имеется достаточно литературы, и интересующийся читатель может более детально и глубоко ознакомиться с ней. Атрибутные грамматики подробно описаны в статье Льюиса, Розенкранца и Стирнза. Однако мы сочли целесообразным поместить в сборник также работу Кнута, где впервые было введено само понятие атрибутной грамматики — очень изящное и стройное. Следует сказать, что у Льюиса, Розенкранца и Стирнза это понятие несколько трансформировалось, приобрело черты практицизма и утратило свою оригинальную прелесть.

Две последние работы (Хоора — Лауэра и Донаху) посвящены вопросу одновременного использования нескольких методов для точного определения одного языка. Смысл такого подхода состоит в том, что разным группам специалистов (например, системным программистам и пользователям) нужны различные аспекты в описаниях языков, а следовательно, им удобнее пользоваться и разными описаниями. Основная проблема, которая здесь возникает, — это совместимость (или непротиворечивость) нескольких описаний. Кроме того, указанные работы могут проиллюстрировать, как усложняется формальное описание, когда в язык пытаются включить те или иные элементы и свойства, встречающиеся в реальных языках программирования. Может быть, имея в виду именно эти усложнения, авторы пытаются проводить мысль о том, что простота формального определения может служить критерием целесообразности использования языка (или отдельных его компонент).

В целом следует сказать, что хотя материал и труден, но очень интересен и, безусловно, будет полезен прежде всего системным программистам всех категорий, а также читателям, интересующимся теоретическими аспектами языков программирования. Хотелось бы надеяться, что появление широкодоступной литературы на русском языке по семантике языков программирования найдет отражение в круге вопросов, с которыми знакомят студентов вузов по специальностям, так или иначе связанным с системным программированием.

В. М. Курочкин.

СЕМАНТИКА КОНТЕКСТНО-СВОБОДНЫХ ЯЗЫКОВ [1)]

Д. Э. Кнут

California Institute of Technology

Найти «значение» цепочки КС-языка можно, вычислив в каждом узле дерева ее вывода так называемые «атрибуты». Атрибуты вычисляются с помощью функций, сопоставленных синтаксическим правилам грамматики. В статье изучается вопрос о том, как выглядит процесс вычисления атрибутов, когда некоторые атрибуты являются «синтезированными», т. е. зависят только от атрибутов *потомков* рассматриваемого нетерминала, а другие — «унаследованными», т.е. зависят от атрибутов *предков* данного нетерминала. Представлен алгоритм, проверяющий, не приводят ли семантические правила ^к циклическому определению некоторых атрибутов. Приведен пример простого языка программирования, в определении которого участвуют как синтезированные, так и унаследованные атрибуты. Метод определения, основанный на использовании атрибутов, сравнивается с другими методами формального описания семантики, имеющимися в литературе.

1. ВВЕДЕНИЕ

Допустим, что нам нужно дать точное определение двоичной системы записи чисел. Это можно сделать многими способами. В данном разделе мы рассмотрим метод, который может быть использован и для других систем счисления. В случае двоичной системы этот метод сводится к определению, основанному на следующей контекстно-свободной грамматике

$$
\begin{aligned}
&B \to 0\\
&B \to 1\\
&L \to B\\
&L \to LB\\
&N \to L\\
&N \to L\cdot L
\end{aligned}
\qquad (1.1)
$$

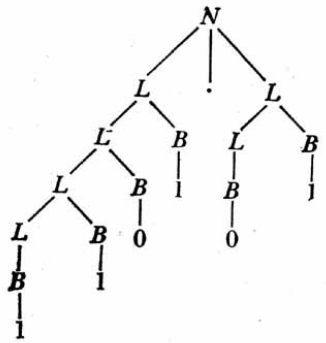
[1)] *Knuth D. E.* Semantics of Context-Free Languages, Mathematical Systems Theory, 2, 2, 1968, 127—146.

Knuth D. E. Semantics of Context-Free Languages: Correction, Mathematical Systems Theory; 5, 1, 1971, 179.

В первоначальном тексте алгоритм теоремы о проверке зацикленности был неверным, и позднее автор дал другой его вариант. Текст, приведенный в настоящем сборнике, скомпилирован из двух вышеуказанных статей. — *Прим. ред.*

(Здесь терминальными символами являются · 0 и 1, нетерминальными — B, L и N, обозначающие соответственно бит, список битов и число. Двоичным числом будем считать любую цепочку терминальных символов, выводимую из N при помощи правил (1.1).) Эта грамматика в действительности выражает тот факт, что двоичное число представляет собой последовательность из одного или более нулей и единиц, за которой может следовать точка и еще одна последовательность нулей и единиц. Кроме того, грамматика приписывает каждому двоичному числу определенную древовидную структуру. Например, цепочка 1101·01 получает следующую структуру:

(1.2)



Естественно определять значение двоичной записи (1.1) с помощью некоторого пошагового процесса, сопоставленного ее структуре (1.2); значение всей двоичной записи строится из значений отдельных частей. Это можно сделать, приписав каждому нетерминалу *атрибуты* следующим образом:

Каждый бит B имеет целочисленный атрибут «значение», обозначаемый $v(B)$.

Каждый список битов L имеет целочисленный атрибут «длина», обозначаемый $l(L)$.

Каждый список битов L имеет целочисленный атрибут «значение», обозначаемый $v(L)$.

Каждое число N имеет атрибут «значение», являющийся рациональным числом и обозначаемый $v(N)$.

(Заметим, что у всех нетерминалов L по два атрибута; вообще говоря, каждому нетерминалу можно приписывать любое желаемое число атрибутов.)

Грамматику (1.1) можно теперь расширить так, чтобы каждому синтаксическому правилу отвечали семантические правила.

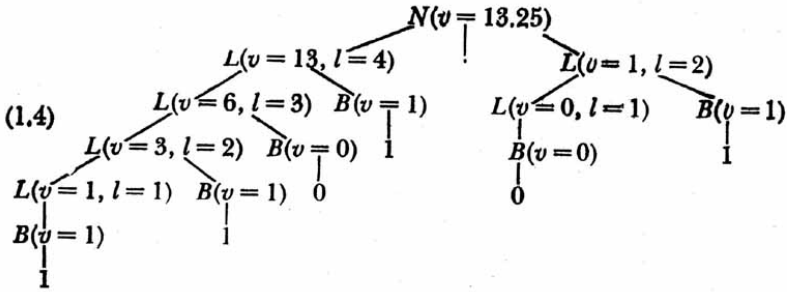
(1.3)

$B \to 0$	$v(B) = 0$	
$B \to 1$	$v(B) = 1$	
$L \to B$	$v(L) = v(B)$,	$l(L) = 1$
$L_1 \to L_2 B$	$v(L_1) = 2v(L_2) + v(B)$,	$l(L_1) = l(L_2) + 1$
$N \to L$	$v(N) = v(L)$	
$N \to L_1 \cdot L_2$	$v(N) = v(L_1) + v(L_2)/2^{l(L_2)}$	

(Индексы в четвертом и шестом правилах применяются для того, чтобы различать вхождения одноименных нетерминалов.) В этих семантических правилах значения атрибутов всех нетерминалов определяются через значения атрибутов их непосредственных потомков, так что окончательные значения определены для всех атрибутов. Предполагается, что смысл обозначений, использованных для записи семантических правил, понятен. Отметим, например, что символ «0» в семантическом правиле «$v(B) = 0$» интерпретируется не так, как символ «0» в синтаксическом правиле «$B \to 0$». В первом случае «0» обозначает математическое понятие, а именно число нуль, во втором — некоторый символ, имеющий эллиптическую форму. В каком-то смысле, то, что эти два символа выглядят одинаково, — не более чем простое совпадение.

Структуру (1.2) можно расширить, выписав явно атрибуты при всех узлах.

(1.4)



Таким образом, «1101·01» обозначает 13.25 (в десятичной записи).

Такой способ определения семантики контекстно-свободных языков в сущности хорошо известен, так как он уже использо-

вался несколькими авторами. Однако существует важное расширение этого метода. Именно это расширение и представляет для нас интерес.

Предположим, например, что мы хотим определить семантику двоичной записи другим способом, более близким к нашему обычному ее пониманию. Первая единица в записи «1101·01» на самом деле означает 8, хотя в соответствии с (1.4) ей приписывается значение 1. Возможно, поэтому будет лучше определять семантику таким образом, чтобы местоположение символа тоже играло определенную роль. Можно ввести следующие атрибуты:

Каждый символ B имеет атрибут «значение», являющийся рациональным числом и обозначаемый $v(B)$.
Каждый символ B имеет целочисленный атрибут «масштаб», обозначаемый $s(B)$.
Каждый символ L имеет атрибут «значение», являющийся рациональным числом и обозначаемый $v(L)$.
Каждый символ L имеет целочисленный атрибут «длина», обозначаемый $l(L)$.
Каждый символ L имеет целочисленный атрибут «масштаб», обозначаемый $s(L)$.
Каждый символ N имеет атрибут «значение», принимающий в качестве значений рациональные числа и обозначаемый $v(N)$.

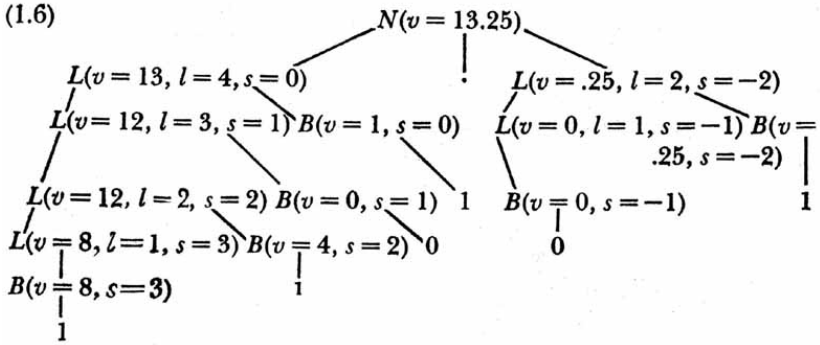
Эти атрибуты можно определить следующим образом:

	Синтаксические правила	Семантические правила
	$B \to 0$	$v(B) = 0$
	$B \to 1$	$v(B) = 2^{s(B)}$
	$L \to B$	$v(L) = v(B),\ s(B) = s(L),\quad l(L) = 1$
(1.5)	$L_1 \to L_2B$	$v(L_1) = v(L_2) + v(B)\quad s(B) = s(L_1),$ $s(L_2) = s(L_1) + 1,\quad l(L_1) = l(L_2) + 1$
	$N \to L$	$v(N) = v(L),\quad s(L) = 0$
	$N \to L_1 \cdot L_2$	$v(N) = v(L_1) + v(L_2),\quad s(L_1) = 0,$ $s(L_2) = -l(L_2)$

(Здесь при записи семантических правил принято следующее соглашение. Правая часть каждого правила представляет собой определение левой части, таким образом, «$s(B) = s(L)$» означает, что сначала должно быть вычислено $s(L)$, а затем полученное значение следует присвоить $s(B)$.)

Важным свойством грамматики (1.5) является то, что некоторые из атрибутов, которым присваиваются значения, приписаны нетерминалам, стоящим в правой части соответствующего синтаксического правила, в то время как в (1.3) атрибуты левых частей семантических правил относились только к нетерминалам, стоящим в левой части синтаксического правила. Здесь мы используем как *синтезированные атрибуты* (вычисляемые через атрибуты потомков данного нетерминала), так и *унаследованные атрибуты* (вычисляемые через атрибуты предков). Синтезированные атрибуты вычисляются в древовидной структуре снизу вверх, а унаследованные — сверху вниз. Грамматика (1.5) включает синтезированные атрибуты $v(B)$, $v(L)$, $l(L)$, $v(N)$ и унаследованные атрибуты $s(B)$ и $s(L)$, так что при их вычислении необходимо проходить по дереву в обоих направлениях. Вычисление на структуре, соответствующей цепочке 1101·01, имеет вид:

(1.6)



Можно заметить, что атрибуты «длина» символов L, стоящих справа от точки, должны быть вычислены снизу вверх до того, как будут вычислены (сверху вниз) атрибуты «масштаб» и атрибуты «значение» (снизу вверх).

Грамматика (1.5), вероятно, не является «наилучшей возможной» грамматикой для системы двоичной записи, но похоже, что она лучше согласуется с нашей интуицией, чем грамматика (1.3). (Грамматика, которая более точно соответствует нашему традиционному толкованию двоичной нотации, содержит другое множество правил вывода. Эти правила сопоставляют цепочке битов справа от точки иную структуру, вследствие чего атрибут «длина», не играющий принципиальной роли, становится ненужным.)

Наш интерес к грамматике (1.5) вызван не тем, что она представляет собой идеальное определение двоичной системы записи, а тем, что она демонстрирует взаимодействие унаследованных и синтезированных атрибутов. Тот факт, что семантические правила, подобные правилам в (1.5), не приводят к зацикленности определения атрибутов, не является очевидным, поскольку здесь атрибуты вычисляются не при однократном обходе дерева в одном направлении. Алгоритм, проверяющий семантические правила на зацикленность, будет описан ниже.

Важность унаследованных атрибутов состоит в том, что они естественно возникают в практике и в очевидном смысле «двойственны» синтезированным атрибутам. Хотя для определения смысла двоичной записи достаточно только синтезированных атрибутов, существует ряд языков, для которых такое ограничение приводит к неуклюжему и неестественному определению семантики. Ситуации, когда встречаются и унаследованные, и синтезированные атрибуты, представляют собой те самые случаи, которые в предшествующих определениях семантики вызывали серьезные трудности.

2. ФОРМАЛЬНЫЕ СВОЙСТВА

Придадим теперь идее использования синтезированных и унаследованных атрибутов более точную и более общую форму.

Пусть имеется контекстно-свободная грамматика $\mathscr{G} = (V, N, S, \mathscr{P})$, где V — (конечный) алфавит терминальных и нетерминальных символов; $N \subseteq V$ — множество нетерминальных символов; $S \in N$ — «начальный» символ, не входящий в правые части правил, и $\mathscr{P}$ — множество правил. Семантические правила дополняют $\mathscr{G}$ следующим образом. С каждым символом $X \in V$ связывается конечное множество атрибутов $A(X)$. $A(X)$ разбивается на два непересекающихся множества: множество синтезированных атрибутов $A_0(X)$ и множество унаследованных атрибутов $A_1(X)$. Множество $A_1(S)$ должно быть пустым (т. е. начальный символ S не должен иметь унаследованных атрибутов); аналогично, множество $A_0(X)$ пусто, если X — терминальный символ. Каждый атрибут α из множества $A(X)$ имеет (возможно, бесконечное) множество значений V_α. Для каждого вхождения X в дерево вывода семантические правила позволяют определить одно значение из множества V_α для соответствующего атрибута.

Пусть $\mathscr{P}$ состоит из m правил, и пусть p-е правило имеет вид

$$X_{p0} \to X_{p1} X_{p2} \ldots X_{pn_p}, \tag{2.1}$$

где $n_p \geqslant 0$, $X_{p0} \in N$ и $X_{pj} \in V$ для $1 \leqslant j \leqslant n_p$. Семантическими правилами называются функции $f_{pj\alpha}$, определенные для всех

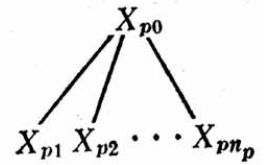
$1 \leqslant p \leqslant m$, $0 \leqslant j \leqslant n_p$ и некоторых $\alpha \in A_0(X_{pj})$, если $j = 0$, или $\alpha \in A_1(X_{pj})$, если $j > 0$. Каждая такая функция представляет собой отображение из $V_{\alpha_1} \times V_{\alpha_2} \times \ldots \times V_{\alpha_t}$ в V_α для некоторого $t = t(p, j, \alpha) \geqslant 0$, где все $\alpha_i = \alpha_i(p, j, \alpha)$ являются атрибутами некоторых X_{pk_i}, при $0 \leqslant k_i = k_i(p, j, \alpha) \leqslant n_p$, $1 \leqslant i \leqslant t$. Другими словами, каждое семантическое правило отображает значения некоторых атрибутов символов $X_{p_0}, X_{p_1}, \ldots, X_{pn_p}$ и значение некоторого атрибута символа X_{pj}.

Грамматика (1.5), например, представляется в виде $\mathscr{G} = (\{0, 1, \cdot, B, L, N\}, \{B, L, N\}, N, \{B \to 0, B \to 1, L \to B, L \to LB, N \to L, N \to L \cdot L\})$.

Атрибутами здесь являются $A_0(B) = \{v\}$, $A_1(B) = \{s\}$, $A_0(L) = \{v, l\}$, $A_1(L) = \{s\}$, $A_0(N) = \{v\}$, $A_1(N) = \varnothing$ и $A_0(x) = A_1(x) = \varnothing$ для $x \in \{0, 1, \cdot\}$. Множествами значений атрибутов будут $V_v = \{$рациональные числа$\}$, $V_s = V_l = \{$целые числа$\}$. Типичным примером правила вывода служит четвертое правило $X_{40} \to X_{41} X_{42}$, где $n_4 = 2$, $X_{40} = X_{41} = L$, $X_{42} = B$. Так же типично и семантическое правило f_{40v}, соответствующее этому правилу вывода. Оно определяет $v(X_{40})$ через другие атрибуты; в данном случае f_{40v} отображает $V_v \times V_v$ в V_v согласно формуле $f_{40v}(x, y) = x + y$. (Это есть не что иное, как правило «$v(L_1) = v(L_2) + v(B)$» из (1.5); используя довольно громоздкую запись, введенную в предыдущем абзаце, получим: $t(4, 0, v) = 2$, $\alpha_1(4, 0, v) = \alpha_2(4, 0, v) = v$, $k_1(4, 0, v) = 1$, $k_2(4, 0, v) = 2$.)

Семантические правила используются для сопоставления цепочкам контекстно-свободного языка «значения» следующим образом [1]). Для любого вывода терминальной цепочки t из S при помощи синтаксических правил построим обычное дерево вывода. А именно, корнем дерева будет S, а каждый узел помечается либо терминальным символом, либо нетерминалом X_{p0}, соответствующим применению p-го правила для некоторого p; в последнем случае у этого узла будет n_p непосредственных потомков.

(2)



[1]) На самом деле значение здесь приписывается дереву вывода цепочки, а не самой цепочке. Если грамматика неоднозначна, это не одно и то же (см. стр. 160). — *Прим. перев.*

Пусть теперь X — метка некоторого узла дерева и пусть $\alpha \in A(X)$ — атрибут символа X. Если $\alpha \in A_0(X)$, то $X = X_{p0}$ для некоторого p, если же $\alpha \in A_1(X)$, то $X = X_{pj}$ для некоторых j и p. В обоих случаях дерево «в районе» этого узла имеет вид (2.2). По определению атрибут α имеет в этом узле значение v, если в соответствующем семантическом правиле

$$f_{pj\alpha}\colon V_{\alpha 1} \times \ldots \times V_{\alpha t} \to V_\alpha \tag{2.3}$$

все атрибуты $\alpha_1, \ldots, \alpha_t$ уже определены и имеют в узлах с метками $X_{pk_1}, \ldots, X_{pk_t}$ значения $v_1, \ldots, v_t$ соответственно, а $v = f_{pj\alpha}(v_1, \ldots, v_t)$. Процесс вычисления атрибутов на дереве продолжается до тех пор, пока нельзя будет вычислить больше ни одного атрибута. Вычисленные в результате атрибуты корня дерева представляют собой «значение», соответствующее данному дереву вывода (1.6).

Естественно потребовать, чтобы семантические правила давали возможность вычислить все атрибуты произвольного узла в любом дереве вывода. Если это условие выполняется, будем говорить, что семантические правила *заданы корректно*[1]). Поскольку деревьев вывода, вообще говоря, бесконечно много, важно уметь определять по самой грамматике, являются ли корректными ее семантические правила. Алгоритм проверки этого свойства приведен в разд. 3.

Отметим, что этот метод определения семантики обладает такой же мощностью, как и всякий другой возможный метод, в том смысле, что значение любого атрибута в любом узле может произвольным образом зависеть от структуры всего дерева. Предположим, например, что в контекстно-свободной грамматике всем символам, кроме S, приписано по два унаследованных атрибута: l («положение») и t («дерево»), а всем нетерминалам, кроме того, по одному синтезированному атрибуту s («поддерево»). Значениями l будут конечные последовательности положительных целых чисел $\{a_1 \cdot a_2 \cdot \cdots \cdot a_k\}$, определяющие местонахождение узла в дереве в соответствии с системой обозначения Дьюи (см. [8], стр. 388—389)[2]). Атрибуты t и s представляют собой множество упорядоченных пар (l, X), где l — положение узла, а X — символ грамматики, обозначающий метку узла с положением l. Семантическими правилами

[1]) В оригинале well defined. — *Прим. ред.*

[2]) Тем, кто незнаком с этой системой обозначений, не обязательно обращаться за консультацией по указанному адресу: принцип системы легко усматривается из формул (2.4). — *Прим. ред.*

для каждого синтакенического правила (2.1) служат

$$l(X_{pj}) = \begin{cases} l(X_{p0}) \cdot j & \text{если } X_{p0} \neq S; \\ j & \text{если } X_{p0} = S; \end{cases}$$

$$t(X_{pj}) = \begin{cases} t(X_{p0}) & \text{если } X_{p0} \neq S; \\ s(X_{p0}) & \text{если } X_{p0} = S; \end{cases} \tag{2.4}$$

$$s(X_{p0}) = \{(l(X_{p0}), X_{p0}) \mid X_{p0} \neq S\} \cup \bigcup_{j=1}^{n_p} \{s(X_{pj}) \mid X_{pj} \in N\}.$$

Следовательно, для дерева (1.2), например, мы имеем

$$\begin{aligned} s(N) = \{&(1, L), (2, \cdot)(3, L), (1.1, L), (1.2, B), (3.1, L), (3.2, B), \\ &(1.1.1, L)(1.1.2, B), (1.2.1, 1), (3.1.1, B), (3.2.1, 1), \\ &(1.1.1.1, L), (1.1.1.2, B), (1.1.2.1, 0)(3.1.1.1, 0), \\ &(1.1.1.1.1, B), (1.1.1.2.1, 1), (1.1.1.1.2.1, 1)\}. \end{aligned}$$

Ясно, что эта запись содержит всю информацию о дереве вывода. Согласно семантическим правилам (2.4), атрибут t во всех узлах (кроме корня) представляет собой множество, характеризующее все дерево вывода; атрибут l определяет местонахождение этих узлов. Отсюда сразу следует, что любая мыслимая функция, определенная на дереве вывода, может быть представлена как атрибут произвольного узла, поскольку эта функция имеет вид $f(t, l)$, для некоторого f. Аналогично, можно показать, что для определения значения, связанного с произвольным деревом вывода, достаточно только синтезированных атрибутов, поскольку синтезированный атрибут w, вычисляемый по формуле

$$w(X_{p0}) = \{(0, X_{p0})\} \cup \bigcup_{j=1}^{n_p} \{(j \cdot \alpha, X) \mid (\alpha, X) \in w(X_{pj}),\ X_{pj} \in N\} \tag{2.5}$$

в корне дерева полностью определяет все дерево[1]). Каждое семантическое правило, определяемое методами этого раздела, можно рассматривать как функцию этого атрибута w. Следовательно, описанный общий метод по существу не более мощен, чем метод, вовсе не использующий унаследованных атрибутов. Правда, это утверждение не следует понимать как практическую рекомендацию, поскольку семантические правила, не использующие унаследованных атрибутов, будут зачастую го-

[1]) В правой части формулы необходимо добавить еще член $\bigcup_{j=1}^{n_p} \{(j \cdot 0, x) \mid x \notin N\}$. — *Прим. ред.*

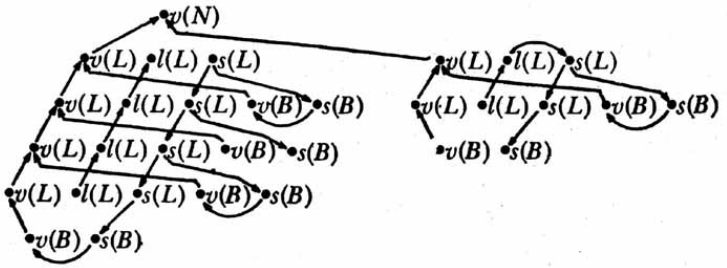
раздо более сложными (а также менее понимаемыми и практичными), чем правила, включающие атрибуты обоих типов. Если допустить, чтобы атрибуты в каждом узле дерева могли зависеть от всего дерева, то семантические правила часто могут стать проще и будут лучше соответствовать нашему пониманию процесса вычисления.

3. ПРОВЕРКА НА ЗАЦИКЛЕННОСТЬ

Рассмотрим теперь алгоритм, проверяющий, является ли корректной система семантических правил, определенная в предыдущем разделе. Другими словами, мы хотим знать, когда семантические правила позволяют вычислить значение любого атрибута любого узла произвольного дерева вывода. Можно считать, что грамматика не содержит «бесполезных» правил вывода, т. е. что каждое правило из $\mathscr{P}$ участвует в выводе хотя бы одной терминальной цепочки.

Пусть $\mathscr{T}$ — произвольное дерево вывода, соответствующее данной грамматике; метками концевых узлов могут быть только терминальные символы, корню же разрешим иметь меткой не только аксиому, но и любой символ из V. Тогда можно определить ориентированный граф $D(\mathscr{T})$, соответствующий $\mathscr{T}$, взяв в качестве его узлов упорядоченные пары (X, α), где X — узел дерева $\mathscr{T}$, а α — атрибут символа, служащего меткой узла X. Дуга из (X_1, α_1) в (X_2, α_2) проводится в том и только в том случае, когда семантическое правило, вычисляющее атрибут α_2, непосредственно использует значение атрибута α_1. Например, если $\mathscr{T}$ — дерево (1.2), а в качестве семантических правил взяты правила (1.5), то орграф $D(\mathscr{T})$ будет иметь такой вид:

(3.1)



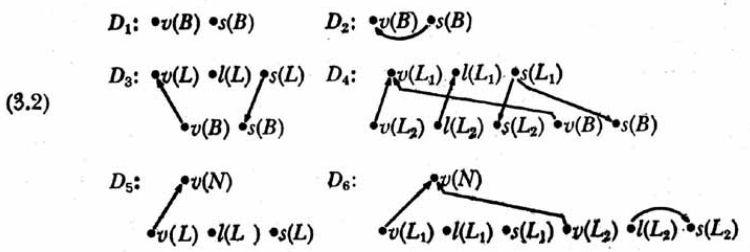
Другими словами, узлами графа $D(\mathscr{T})$ служат атрибуты, значения которых нужно вычислить, а дуги определяют зависимости, подразумевающие, какие атрибуты вычисляются раньше, а какие позже (1.6).

Ясно, что семантические правила являются корректными тогда и только тогда, когда ни один из орграфов $D(\mathscr{T})$ не

содержит ориентированного цикла. Дело в том, что если в графе нет ориентированных циклов, то можно применить хорошо известную процедуру, позволяющую присвоить значения всем атрибутам (см. [8], стр. 465-6). Если же в некотором графе $D(\mathscr{T})$ есть ориентированный цикл, то ввиду того что грамматика не содержит бесполезных правил, можно утверждать, что существует ориентированный цикл в некотором графе $D(\mathscr{T})$, у которого меткой корня дерева $\mathscr{T}$ служит S. Для такого дерева $\mathscr{T}$ все атрибуты вычислить не удается. Таким образом, задача «Являются ли семантические правила корректными?» сводится к задаче «Содержат ли орграфы $D(\mathscr{T})$ ориентированные циклы?»

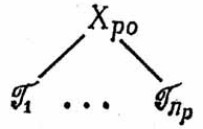
Каждый орграф $D(\mathscr{T})$ можно рассматривать как суперпозицию меньших орграфов D_p, соответствующих правилам $X_{p0} \to X_{p1} \ldots X_{pn_p}$ грамматики, $1 \leqslant p \leqslant m$. В обозначениях разд. 2 орграф D_p имеет узлы (X_{pj}, α) для $0 \leqslant j \leqslant n_p$, $\alpha \in A(X_{pj})$ и дуги из (X_{pk_i}, α_i) в (X_{pj}, α) для $0 \leqslant j \leqslant n_p$, $\alpha \in A_0(X_{pj})$, если $j = 0$, $\alpha \in A_1(X_{pj})$, если $j > 0$, $k_i = k_i(p, j, \alpha)$, $\alpha_i = \alpha_i(p, j, \alpha)$, $1 \leqslant i \leqslant t(p, j, \alpha)$. Другими словами, D_p отражает связи, которые порождают все семантические правила, соответствующие p-му синтаксическому правилу. Например, шести правилам грамматики (1.5) соответствуют шесть следующих орграфов:

(3.2)



Орграф (3.1) получается в результате «объединения» таких подграфов. Вообще, если $\mathscr{T}$ имеет в качестве метки корня терминал, $D(\mathscr{T})$ не содержит дуг. Если корень дерева $\mathscr{T}$ помечен нетерминальным символом, $\mathscr{T}$ имеет вид

(3.3)



для некоторого p, где $\mathscr{T}_j$ — дерево вывода, у которого корень помечен символом X_{pj}, где $1 \leqslant j \leqslant n_p$. В первом случае говорят,

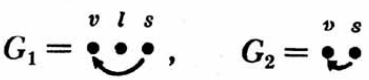
что $\mathscr{T}$ — дерево вывода типа 0, во втором случае $\mathscr{T}$ называется *деревом вывода типа p*. В соответствии с этим определением для того, чтобы по D_p, $D(\mathscr{T}_1)$, ..., $D(\mathscr{T}_{n_p})$ построить $D(\mathscr{T})$, нужно для всех j, $1 \leqslant j \leqslant n_p$ совместить узлы, соответствующие атрибутам символа X_{pj} графа D_p с соответствующими узлами (отвечающими тем же атрибутам корня дерева $\mathscr{T}_j$) в графе $D(\mathscr{T}_j)$.

Для проверки того, содержит ли граф $D(\mathscr{T})$ ориентированный цикл, нам понадобится еще одно понятие. Пусть p — номер правила вывода. Обозначим через G_j произвольный орграф $(1 \leqslant j \leqslant n_p)$, множество узлов которого является подмножеством множества $A(X_{pj})$ атрибутов символа X_{pj}. Пусть

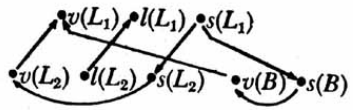
$$D_p[G_1, \ldots, G_{n_p}] \tag{3.4}$$

орграф, полученный из D_p добавлением дуг, идущих из (X_{pj}, α) в (X_{pj}, α'), если в графе G_j есть дуга из α в α'.

Например, если



и если D_4 — ориентированный граф из (3.2), то $D_4(G_1, G_2)$ имеет вид



Теперь можно воспользоваться следующим алгоритмом. Для любого $X \in V$ $S(X)$ будет некоторым множеством ориентированных графов с узлами из $A(X)$. Сначала для всех $X \in N$ $S(X)$ пусто, а для $X \notin N$ $S(X)$ состоит из единственного графа с множеством узлов $A(X)$ и не содержащего дуг. Будем добавлять к множествам $S(X)$ новые орграфы при помощи следующей процедуры до тех пор, пока в $S(X)$ не перестанут появляться новые элементы. Выберем целое p, $1 \leqslant p \leqslant m$ и для каждого j, $1 \leqslant j \leqslant n_p$, выберем орграф $D'_j \in S(X_{pj})$. Затем добавим в $S(X_{p0})$ орграф с множеством узлов $A(X_{p0})$, обладающий тем свойством, что в нем дуга от α к α' идет тогда и только тогда, когда в орграфе

$$D_p[D'_1, \ldots, D'_{n_p}] \tag{3.5}$$

существует ориентированный путь из (X_{p0}, α) в (X_{p0}, α'). Ясно, что этот процесс рано или поздно закончится и новые орграфы

перестанут порождаться, поскольку вообще существует лишь конечное число ориентированных графов.

В случае грамматики (1.5) алгоритм построит следующие множества:

Пусть $\mathscr{T}$ — дерево вывода с корнем X, и пусть $D'(\mathscr{T})$ — ориентированный граф с множеством узлов $A(X)$, у которого есть дуга из α в α' тогда и только тогда, когда в $D(\mathscr{T})$ существует ориентированный путь из (X, α) в (X, α'). Покажем, что после окончания работы описанного выше алгоритма для всех $X \in V$ $S(X)$ — *это множество всех* $D'(\mathscr{T})$, *где* $\mathscr{T}$ — *дерево вывода с корнем* X[1]). Действительно, построение не добавляет к $S(X)$ новых ориентированных графов, не являющихся $D'(\mathscr{T})$. Алгоритм можно даже легко обобщить так, чтобы для каждого графа из $S(X)$ он печатал на выходе соответствующее дерево вывода $\mathscr{T}$. Обратно, если $\mathscr{T}$ — дерево вывода, мы можем показать индукцией по числу узлов дерева $\mathscr{T}$, что $D'(\mathscr{T})$ принадлежит некоторому множеству $S(X)$. В противном случае $\mathscr{T}$ должно иметь вид (3.3) и $D(\mathscr{T})$ «составлен» из D_p $D(\mathscr{T}_1)$, ..., $D(\mathscr{T}_{n_p})$. По индукции и вследствие того, что при $j \neq j'$ из $D(\mathscr{T}_j)$ в $D(\mathscr{T}_{j'})$ не проходит дуг вне D_p, дуги в $D(\mathscr{T}_1)$, ..., $D(\mathscr{T}_{n_p})$, составляющие рассматриваемый путь графа $D(\mathscr{T})$, можно заменить соответствующими дугами в $D_p[D'_1, \ldots, D'_{n_p}]$, где $D'_j \in S(X_{pj})$, $1 \leqslant j \leqslant n_p$. Поэтому ориентированный граф, включаемый в $S(X_{p0})$ на базе $D_p[D'_1, \ldots, D'_{n_p}]$, просто совпадет с $D'(\mathscr{T})$.

Вышеприведенный алгоритм решает задачу, поставленную в этом разделе.

Теорема. *Семантические правила, добавленные к грамматике так, как это сделано в разд.* 2, *являются корректными тогда и только тогда, когда ни один из ориентированных графов* (3.5) *ни при каком выборе* p *и* $D'_1 \in S(X_{p_1}), \ldots, D'_{n_p} \in S(X_{n_p})$ *не содержит ориентированных циклов.*

---

[1]) Если, как определялось выше, концевыми узлами $\mathscr{T}$ могут быть только терминальные символы, то $S(X)$ будет *содержать* все $D'(\mathscr{T})$, а *не совпадать* со множеством всех $D'(\mathscr{T})$. Чтобы не вносить в последующие построения несущественных исправлений, проще в этом абзаце считать $\mathscr{T}$ любым деревом вывода с корнем X (т. е. не обязательно продолженным до терминалов). — *Прим. ред.*

Доказательство. Если (3.5) содержит ориентированный цикл, то, как было показано выше, некоторый $D(\mathscr{T})$ содержит ориентированный цикл. Наоборот, если $\mathscr{T}$ — дерево с наименьшим возможным числом узлов, такое, что $D(\mathscr{T})$ содержит ориентированный цикл, то $\mathscr{T}$ должно иметь вид (3.3), а $D(\mathscr{T})$ «составляется» из D_p; $D(\mathscr{T}_1), \ldots, D(\mathscr{T}_{np})$. Из минимальности $\mathscr{T}$ следует, что ориентированный цикл включает по меньшей мере одну дугу графа D_p, и, следовательно, можно, рассуждая, как выше, все дуги, образующие этот цикл и лежащие в одном из графов $D(\mathscr{T}_1), \ldots, D(\mathscr{T}_{np})$, заменить дугами графа (3.5).

4. ПРОСТОЙ ЯЗЫК ПРОГРАММИРОВАНИЯ

Сейчас мы продемонстрируем, как описанный выше метод семантического определения можно применять к языкам программирования. Для простоты изучим формальное определение небольшого языка, описывающего программы для машин Тьюринга.

Машина Тьюринга (в классическом смысле) работает с бесконечной лентой, которую можно представлять себе разделенной на клетки. Машина может считывать или записывать символы некоторого конечного алфавита в обозреваемую в некоторый момент клетку, а также сдвигать читающее устройство на одну клетку вправо или влево. Следующая программа, например, прибавляет единицу к целому числу, представленному в двоичном виде, и печатает точку справа от этого числа. Предполагается, что в начале и в конце работы программы читающее устройство находится на первой пустой клетке справа от числа.

Алфавит *пробел, единица, нуль, точка*;
печатать «*точка*»;
перейти на *выполнить*;
тест: **если символ на ленте** «*единица*» **то**
(4.1) {**печатать** «*нуль*»; *выполнить*: **сдвинуться влево на одну клетку; перейти на** *тест*};
печатать «*единица*»;
возврат: **сдвинуться вправо на одну клетку;**
если символ на ленте «*нуль*» **то перейти** *на возврат*.

(Читатель, по-видимому, найдет этот язык программирования достаточно прозрачным для того, чтобы понять его, прежде чем будет дано какое-либо формальное определение, хотя это и не обязательно. Приведенная выше программа не является примером искусного программирования. Она лишь иллюстрирует

некоторые черты простого языка, рассматриваемого в настоящем разделе.)

Поскольку каждый язык программирования нужно как-то называть, назовем наш язык Тьюринголом. Всякая правильная программа на Тьюринголе определяет программу для машины Тьюринга; будем говорить, что программа для машины Тьюринга состоит из

множества «состояний» Q,
множества «символов» Σ,
«начального состояния» $q_0 \in Q$,
«конечного состояния» $q_\infty \in Q$,

и «функции переходов» δ, отображающей множество $(Q - \{q_\infty\}) \times \Sigma$ в $\Sigma \times \{-1, 0, +1\} \times Q$. Если $\delta(q, s) = (s', k, q')$, то это означает, что если машина находится в состоянии q и обозревает символ s, то она печатает символ s', сдвигает читающее устройство на k клеток вправо (сдвигу на одну клетку влево соответствует случай $k = 1$) и переходит в состояние q'. Формально программа машины Тьюринга определяет вычисление для ленты с «любым начальным содержимым», т. е. для любой бесконечной в обе стороны последовательности

4.2) $$\ldots, a_{-3}, a_{-2}, a_{-1}, a_0, a_1, a_2, a_3, \ldots$$

элементов алфавита Σ следующим образом. В произвольный момент вычисления существует «текущее состояние» $q \in Q$ и целочисленная величина «положение читающего устройства» p. Вначале $q = q_0$ и $p = 0$. Если $q \neq q_\infty$ и если $\delta(q, a_p) = (s', k, q')$, то следующим шагом вычисления будет замена значения p на $p + k$, q на q' и a_p на s'. Если $q = q_\infty$, вычисление заканчивается. (Вычисление может не закончиться; для программы (4.1) это произойдет тогда и только тогда, когда $a_j =$ «единица» для всех $j < 0$.)

Теперь, когда у нас имеется точное определение программ машин Тьюринга, мы хотим определить программу машины Тьюринга, соответствующую произвольной программе на Тьюринголе (и одновременно определить синтаксис Тьюрингола). Для этого удобно принять некоторое соглашение о форме записи.

(1) Семантическое правило **«включить** x **в** B**»**, связанное с синтаксическим правилом, означает, что x должен стать элементом множества B, где B — атрибут аксиомы S грамматики. Значением B будет множество всех x, для которых существует такое семантическое правило, связанное с каждым применением соответствующего синтаксического правила в дереве вывода. (Это правило можно рассматривать как сокращенную запись

семантического правила

$$B(X_{p0}) = \bigcup_{j=1}^{n_p} B(X_{pj}) \cup \{x \mid \text{«включить } x \text{ в } B\text{»}$$

связано с p-м правилом$\}$ (4.3)

связанного с каждым синтаксическим правилом. Здесь B — синтезированный атрибут, имеющийся у всех нетерминальных символов. Для терминальных символов $B(x)$ пусто. Ясно, что эти правила позволяют получить нужное $B(S)$.)

(2) Семантическое правило «**определить** $f(x) = y$», связанное с синтаксическим правилом, означает, что значение функции f в точке x будет равно y; здесь f — атрибут аксиомы S грамматики. Если встречается два правила, задающих значение $f(x)$ для одного и того же значения x, то возникает ситуация ошибки, а дерево вывода, в котором она возникла, называется *неправильным*. Далее, к функции f можно обращаться в других семантических правилах при условии, что $f(x)$ будет использоваться только тогда, когда значение функции для аргумента x определено. Любое дерево вывода, для которого встретилось обращение к неопределенной величине $f(x)$, называется *неправильным*. (Правила такого типа важны, например, тогда, когда нужно обеспечить соответствие между описанием и использованием идентификаторов. В приведенном ниже примере в соответствии с этим соглашением неправильными будут программы, в которых один и тот же идентификатор дважды встречается в качестве метки, или в операторе перехода используется идентификатор, не являющийся меткой оператора. В сущности, это правило можно представлять себе, аналогично (1), как «**включить** (x, y) **в** f», если рассматривать f как множество упорядоченных пар; необходимо соответственно ввести дополнительные проверки на зацикленность. Признак «правильно или неправильно» можно считать атрибутом аксиомы S. Построение соответствующих семантических правил, аналогичных (4.3), которые аккуратно определяют запись «**определить** $f(x) = y$», несложно и предоставляется читателю.)

(3) Функция «**новсимвол**», в каком бы правиле она ни встретилась, будет вырабатывать некий абстрактный объект, причем при каждом обращении к «новсимвол» этот объект будет отличен от всех полученных при предшествовавших обращениях к **новсимвол**. (Эту функцию легко выразить при помощи других семантических правил, например используя атрибуты l из (2.3), принимающие в разных вершинах дерева разные значения. Функция **новсимвол** служит подходящим источником «сырья» для построения множеств.)

Мы видели, что соглашения (1), (2) и (3) можно заменить другими семантическими конструкциями, не использующими таких соглашений, следовательно, они не являются «базисными» для семантики. Но они чрезвычайно полезны, так как соответствуют понятиям, которыми часто пользуются, поэтому их можно считать принципиальными для метода описания семантики, представленного в настоящей статье. Эффект от введения таких соглашений состоит в том, что уменьшается общее количество атрибутов, явно присутствующих в правилах, и в том, что удается обойтись без неоправданно длинных правил.

Теперь уже несложно дать формальное определение синтаксиса и семантики Тьюрингола.

Нетерминальные символы: P (программа), S (оператор), L (список операторов), I (идентификатор), O (направление), A (символ алфавита), D (описание).

Терминальные символы: $a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ o\ p\ q\ r\ s\ t\ u\ v\ w\ x\ y\ z$. , : ; ‘ ’ { } **алфавит перейти на печатать если символ на ленте то сдвинуться на одну клетку влево вправо**

Начальный символ: p
Атрибуты:

Имя атрибута	*Тип значения*	*Цель введения*
Q	Множество	Состояния программы
Σ	Множество	Символы программы
q_0	Элемент множества Q	Начальное состояние
q_∞	Элемент множества Q	Конечное состояние
δ	Функция, отображающая $(Q - \{q_\infty\}) \times \Sigma$ в $\Sigma \times \{-1, 0, +1\} \times Q$	Функция переходов
метка	Функция, отображающая цепочки букв в элементы множества Q	Таблица состояний для операторных меток
символ	Функция, отображающая цепочки букв в элементы множества Σ	Таблица символов для символов ленты

следующее	Элемент множества Q	Состояние, непосредственно следующее за оператором или списком операторов
d	± 1	Направление
текст	Цепочка букв	Идентификатор
начало	Элемент множества Q	Состояние в начале выполнения оператора или списка операторов (унаследованный атрибут).

Синтаксические и семантические правила см. в табл. 1.

Отметим, что каждому оператору S соответствует два состояния: начало(S) — состояние, соответствующее первой команде, входящей в оператор (если таковая имеется), являющееся унаследованным атрибутом символа S, и следующее(S) — состояние, «следующее» за оператором, или состояние, в которое попадает машина после нормального выполнения оператора. В случае оператора перехода, однако, программа не попадет в состояние следующее(S), поскольку действие оператора состоит в передаче управления в другое место; о состоянии следующее(S) можно сказать, что оно следует за оператором «статически» или «текстуально», а не «динамически» в ходе выполнения программы.

В табл. 1 следующее(S) является синтезированным атрибутом; можно составить аналогичные семантические правила, в которых атрибут следующее(S) будет унаследованным. При этом, правда, программы, включающие пустые операторы, будут несколько менее эффективны (см. правило 4.4). Аналогично, оба атрибута начало(S) и следующее(S) могут быть сделаны синтезированными, но это будет стоить дополнительных инструкций в программе машины Тьюринга при реализации списка операторов.

Наш пример мог бы быть проще, если бы мы использовали менее традиционную форму инструкций машины Тьюринга. Принятое нами определение требует, чтобы каждая инструкция включала действия чтения, печати и сдвига читающего устройства. Машина Тьюринга представляется при этом в виде некоей одно-плюс-одно-адресной вычислительной машины, в которой каждая инструкция определяет местоположение (состояние) следующей инструкции. Метод определения семантических правил, использованный в этом примере, где атрибут следующее(S) является синтезированным, а начало(S) — унаследованным, го-

Таблица 1

Комментарий	*Номер*	*Синтаксическое правило*	*Пример*	*Семантическое правило*
Буквы	1.1	$A \to a$	*a*	**текст** (A) = *a*
	. . . .	. . . . .	(аналогично для всех букв)	
	1.26	$A \to z$	*z*	текст (A) = *z*.
Идентификаторы	2.1	$I \to A$	*m*	текст (I) = текст (A).
	2.2	$I \to IA$	*marilyn*	текст (I) = текст (I) текст (A).
Описания	3.1	$D \to$ **алфавит** I	**алфавит** *marilyn*	**определить символ** (текст (I)) = **новсимвол**; **включить** символ (текст I) **в** Σ.
	3.2	$D \to D, I$	**алфавит** *marilyn, jayne, birgitta*	**определить символ** (текст (I)) = **новсимвол**; **включить** символ (текст (I)) **в** Σ.
Оператор печати	4.1	$S \to$ **печатать** 'I'	**печатать** *'jayne'*	**определить** δ (начало (S), s) = (символ (текст (I)), 0, следующий (S)) для всех $s \in \Sigma$; следующий (S) = **новсимвол**; **включить следующий** (S) **в** Q.
Оператор сдвига	4.2	$S \to$ **сдвинуться** O **на одну клетку**	**сдвинуться влево на одну клетку**	**определить** δ (начало (S), s) = (s, d (0), следующий (S)) для всех $s \in \Sigma$; следующий (S) = **новсимвол**; **включить следующий** (S) **в** Q.
	4.2.1	$O \to$ **влево**	**влево**	d (O) = -1.
	4.2.2	$O \to$ **вправо**	**вправо**	d (O) = $+1$.
Оператор перехода	4.3	$S \to$ **перейти на** l	**перейти на** *boston*	**определить** δ (начало (S), s)) = (s, 0, метка (текст (I)) для всех $s \in \Sigma$; следующий (S) = **новсимвол**; **включить следующий** (S) **в** Q.

Продолжение таблицы 1

Комментарий	Номер	Синтаксическое правило	Пример	Семантическое правило
Пустой оператор	4.4	$S \to$		следующий (S) = начало (S).
Условный оператор	5.1	$S_1 \to$ если символ на ленте 'I' то S_2	**если символ на ленте** *'martlin'* **то печатать** *'jayne'*	**определить** δ (начало (S_1), s) = (s, 0, следующий (S_2)) для всех $s \in \Sigma$ — символ (текст (I)); **определить** δ (начало (S_1), s) =(s, 0, следующий (S_2)) для s = символ (текст (I)); начало (S_2) = **новсимвол**; следующий (S_1) = следующий (S_2); **включить** начало (S_2) **в** Q.
Помеченный оператор	5.2	$S_1 \to I : S_2$	*boston:* **сдвинуться влево на одну клетку**	определить метка (текст (I)) = начало (S_1); начало (S_2) = начало (S_1); следующий (S_1) = следующий (S_2).
Составной оператор	5.3	$S \to \{L\}$	{**печатать** *'jayne'* **перейти на** *boston*}	начало (L) = начало (S); следующий (S) = следующий (L).
Список операторов	6.1	$L \to S$	**печатать** *'jayne'*	начало (S) = начало (L); следующий (L) = следующий (S).
	6.2	$L_1 \to L_2 ; S$	**печатать** *'jayne'*	начало (L_2) = начало (L_1);
			перейти на *boston*	начало (S) = следующий (L_2); следующий (L_1) = следующий (S).
Программа	7	$P \to D ; L$	**алфавит** *martlin, jayne, birgitta*	q_0 = **новсимвол**; **включить** q_0 **в** Q;
			печатать *'jayne'*	начало (L) = q_0; q_∞ = следующий (L).

дится и для вычислительной машины или автомата, в котором $n+1$-я инструкция выполняется после n-й. В этом случае (следующее(S) — начало(S)) есть число инструкций, «скомпилированных» для оператора S.

Создается впечатление, что такое определение Тьюрингола приближает нас к желанной цели: придать точный смысл тем понятиям, которые встречаются в неформальном руководстве по языку для программиста, причем сделать это нужно совершенно формально и однозначно. Другими словами, это определение, возможно, отвечает нашему образу мышления при изучении языка. Определение 4.1 оператора печати, например, можно легко перевести на естественный язык, написав

«Оператор может иметь вид

печатать «I»

где I — идентификатор. Это означает, что всякий раз при выполнении этого оператора символ на обозреваемой клетке ленты будет заменен символом, обозначенным I, безотносительно к тому, какой символ находится в обозреваемой клетке. После этого выполнение программы продолжится с новой инструкции, которая определяется (другими правилами) как следующая за данным оператором».

5. ОБСУЖДЕНИЕ

Идея определения семантики с помощью синтезированных атрибутов, связанных с каждым нетерминальным символом, и семантических правил, сопоставленных каждому правилу вывода, принадлежит Айронсу [6, 7]. Первоначально каждый нетерминальный символ имел ровно один атрибут, называвшийся его «трансляцией». Эта идея использовалась Айронсом и позже другими авторами, особенно Макклюром [14] при построении «синтаксически управляемых компиляторов», переводивших языки программирования в машинный код.

Как мы видели в разд. 2, синтезированных атрибутов достаточно (в принципе) для определения любой функции на дереве вывода. Но на практике применение наряду с синтезированными и унаследованных атрибутов, как описано в данной статье, приводит к значительным упрощениям. Определение Тьюрингола, например, показывает, что легко учитывается согласованность описаний и использований символов, а также между метками и операторами. Другой общей особенностью языков программирования, определение которой значительно упрощается в результате применения унаследованных атрибутов,

является «блочная структура». Вообще говоря, унаследованные атрибуты полезны всякий раз, когда часть значения некоторой конструкции определяется контекстом, в котором находится эта конструкция. Метод, приведенный в разд. 2, показывает, как можно формально описывать унаследованные и синтезированные атрибуты, а в разд. 3 показано, что можно не принимать во внимание проблему зацикленности (являющуюся потенциальным источником трудностей при использовании атрибутов разных типов).

Автору к настоящему времени известно несколько работ, внесших принципиальный вклад в решение задачи формального описания семантики языков программирования. Это определение Алгола 60 средствами расширенного алгоритма Маркова, данное Дебаккером [1], определение Алгола 60 с помощью λ-исчисления, принадлежащее Ландину [9, 10, 11] (см. также Бём [2, 3]), определение Микро-Алгола с помощью рекурсивных функций, применяемых к программе и к «векторам состояний», принадлежащее Маккарти [12] (см. также Маккарти и Пэинтер [13]); определение языка Эйлер средствами семантических правил, применяемых во время синтаксического анализа программы, предложенное Виртом и Вебером [16], и определение языка ПЛ/I [15], данное Венской лабораторией фирмы IBM и основанное на работе Маккарти и Ландина, а также на понятии абстрактной машины, введенном Элготом [4, 5].

Наиболее существенная разница между предшествующими методами и описанием языка Тьюрингол, приведенным в табл. 1, состоит в том, что остальные определения представляют собой довольно сложные процессы, применяемые ко всей программе; можно сказать, что человек, прежде чем он поймет описание языка, должен будет понять, как устроен его компилятор. Эта трудность особенно ощутима в работе Дебаккера, определяющего машину, подобную марковским алгоритмам, но значительно более сложную. Эта машина имеет около 800 команд. На каждом шаге вычисления машины нужно выполнять последнюю применимую команду, так что мы не можем проверить, нужно ли выполнить команду номер 100, до тех пор, пока не убедимся, что остальные 700 команд неприменимы. Кроме того, в процессе работы машины список пополняется новыми командами. Ясно, что читателю чрезвычайно трудно понять работу такой машины или формально доказать ее основные свойства. Описание Тьюрингола, напротив, определяет каждую конструкцию языка только через ее «непосредственное окружение», сводя тем самым к минимуму взаимосвязи между определениями разных частей языка. Определение составных операторов, операторов перехода и т. д. не влияет существенно на определение оператора печати; например, любое из правил 4.1, 4.2, 4.3, 4.4, 5.1, 5.3 можно выбросить, и получится строгое определение другого языка. Такая локализация и разделение семантических правил помогает сделать определение более понятным и кратким.

Хотя определения остальных авторов, упомянутые выше, не так сложны, как определение Дебаккера, в их работах все-таки присутствуют относительно сложные зависимости между отдельными частями определения. Рассмотрим, например, формальное определение языка Эйлер, данное Виртом и Вебером [16]. Это краткое описание весьма сложного языка и потому, безусловно, оно является одним из наиболее удачных формальных определений. И все же, несмотря на то, что Вирт и Вебер проверили свое определение с помощью моделирования на вычислительной машине, весьма вероятно, что некоторые черты Эйлера удивят его создателей. Следующая программа на Эйлере синтаксически и семантически правильна, хотя после метки L нигде не встречается двоеточия:

```
⊥ begin label L; new A; A ← 0;
  if false then go to L else L;
  out 1; L; A ← A + 1; out 2;
  if false then go to L else
  if A < 2 then go to L else out 3; L end ⊥
```

Результатом работы этой программы будет **1, 2, 2, 3**! Промахи такого рода не являются неожиданностью при алгоритмическом определении языка. При использовании методов разд. 4 подобные ошибки менее вероятны.

Есть основания утверждать, что ни одна из предыдущих схем (формального определения семантики) не в состоянии дать такого же краткого и простого для понимания определения Тьюрингола, как то, которое представлено выше. Кроме того (хотя детали окончательно не проработаны), оказывается, что Алгол 60, Эйлер, Микро-Алгол и ПЛ/I также можно определить методами разд. 4, причем все преимущества по сравнению с остальными методами сохраняются. Правда, здесь автор не может быть беспристрастным судьей, поэтому для подтверждения такой точки зрения требуется некоторый дополнительный опыт.

Отметим, что семантические правила в том виде, в котором они даны в настоящей статье, не зависят от конкретно выбранного метода синтаксического анализа. На самом деле они не привязаны даже к конкретным формам синтаксиса. Единственное от чего зависят семантические правила — это имя нетерми-

нала в левой части синтаксического правила и имена нетерминалов в правой его части. Конкретные знаки пунктуации и порядок, в котором нетерминалы располагаются в правых частях правил; несущественны с точки зрения семантических правил. Таким образом, рассматриваемое здесь определение семантики хорошо сочетается с идеей Маккарти об «абстрактном синтаксисе» [12, 13].

Когда синтаксис неоднозначен в том смысле, что некоторые цепочки языка имеют более одного дерева вывода, семантические правила дают для каждого дерева вывода свое «значение». Предположим, например, что к грамматике (1.3) добавлены правила

$$L_1 \to BL_2 \quad v(L_1) = 2^{l(L_2)} v(B) + v(L_2), \qquad l(L_1) = l(L_2) + 1.$$

Грамматика в результате становится синтаксически неоднозначной, но остается по-прежнему семантически однозначной, поскольку атрибут $v(N)$ имеет одно и то же значение для всех деревьев вывода. С другой стороны, если изменить правило 5.2 определения Тьюрингола с $S \to I$: S на $S \to S$: I, грамматика станет неоднозначной как синтаксически, так и семантически.

СПИСОК ЛИТЕРАТУРЫ

1. 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.
2. Böhm 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.
3. Corrado Böhm and Wolf Cross, «Introduction to the CUCH», *Automata Theory* (ed. by Caianiello E. R.), pp. 35—65, Academic press, 1966.
4. Elgot C. C., «Machine species and their computation languages», *Formal Language Description Languages for Computer Programming*, pp. 160—179, IFIP Working Conf., Vienna (1964), North Holland, 1966.
5. Elgot C. C., Robinson A., «Random-access, stored program machines, an approach to programming languages», *J. ACM* **11** (1964), 365—399.
6. Irons E. T., A sintax directed compiler for ALGOL 60, *Comm. ACM* **4** (1961), 51—55.
7. Irons E. T., Towards more versatile mechanical translators, Proc. Sympos. Appl. Math., Vol. 15, pp. 41—50, Amer. Math. Soc., Providence, R. I., 1963.
8. Knuth D. E., *The Art of Computer Programming*, I, Addison-Wesley, 1968. (Русский перевод: Кнут Д. Искусство программирования для ЭВМ. Т. I. Основные алгоритмы. — М.: Мир, 1976.)
9. Landin P. J., «The mechanical evaluation of expressions», *Comp. J.* **6** (1964), 308—320.
10. Landin P. J., A formal description of ALGOL 60, *Formal Language Description Languages for Computer Programming*, pp. 266—294, Proc. IFIP Working Conf., Vienna (1964), North Holland, 1966.
11. Landin P. J., A correspondence between ALGOL 60 and Church's lambda notation, *Comm. ACM* **8** (1965), 89—101, 158—165.

12. McCarthy J., A formal definition of a subset of ALGOL, *Formal Language description Languages for Computer Programming*, pp. 1—12, Proc. IFIP Working Conf., Vienna (1964), North Holland, 1966.
13. McCarthy J., Painter J., Correctness of a compiler for arithmetic expressions, Proc. Sympos. Appl. Math., Vol. 17 Amer. Math. Soc., Providence, R. I., 1967.
14. McClure R. M., TMG — A syntax directed compiler, *Proc. ACM Nat. Conf.* **20** (1965), 262—274.
15. PL/I Definition Group of the Vienna Labratory, *Rormal Definition of PL/I*, IBM Technical Report TR 25.071 (1966).
16. Wirth N., Weber H., Euler: A generalization of ALGOL, and its formal definition, *Comm. ACM* **9** (1966), 11—23, 89—99, 878.

Содержание

От редактора перевода 5

М. Маркотти, Х. Ледгард, Г. Бохман. **Формальные описания языков программирования.** *Перевод А. Н. Бирюкова* 9

Д. Э. Кнут. **Семантика контекстно-свободных языков.** *Перевод А. Н. Бирюкова* 137

П. Льюис, Д. Розенкранц, Р. Стирнз. **Атрибутные трансляции.** *Перевод А. Н. Бирюкова* 162

Ч. Э. Р. Хоор. **Непротиворечивые взаимодополняющие теории семантики языков программирования.** *Перевод А. Н, Бирюкова.* . 196

Джеймс Донаху. **Взаимодополняющие определения семантики языка программирования.** *Перевод В. А. Серебрякова* 222

СЕМАНТИКА ЯЗЫКОВ ПРОГРАММИРОВАНИЯ
Сборник статей

Научный редактор К. Г. Батаев
Мл. научный редактор Н. С. Полякова
Художественный редактор В. И. Шаповалов
Технический редактор И. М. Кренделева
Корректор Н. И. Баранова

ИБ № 1805

Сдано в набор 01.10.79. Подписано к печати 15.07.80. Формат 60×90 $^{1}/_{16}$.
Бумага типографская № 2. Гарнитура латинская. Печать высокая.
Объем 12,5 бум. л. Усл. печ. л. 25. Уч.-изд. л. 22,27.
Изд. № 1/0317. Тираж 8000 экз. Зак. 415.
Цена 2 р. 30 к.

ИЗДАТЕЛЬСТВО «МИР»
129820, Москва, И ПО, ГСП,
1-й Рижский пер., 2.

Ленинградская типография № 2, головное предприятие ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Союзполиграф-прома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 198052, г. Ленинград, Л-52, Измайловский проспект, 29.