THE THEORY OF PARSING, TRANSLATION AND COMPILING

volume 2: COMPILING

ALFRED V. AHO
Bell Telephone Laboratories, Inc.
Murray Hill, N. J.

JEFFREY D. ULLMAN
Department of Electrical Engineering
Princeton University

Prentice-Hall, Inc.
Englewood Cliffs, N. J.
1973

А. АХО, ДЖ. УЛЬМАН

ТЕОРИЯ СИНТАКСИЧЕСКОГО АНАЛИЗА, ПЕРЕВОДА И КОМПИЛЯЦИИ

Том 2
Компиляция

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

Под редакцией
В. М. Курочкина

ИЗДАТЕЛЬСТВО «МИР»
МОСКВА 1978

УДК 519.682.1+681.142.2

Второй том фундаментальной монографии известных американских ученых посвящен методам оптимизации синтаксических анализаторов, теории синтаксически управляемого перевода, а также способам организации памяти при переводе. Большое внимание уделяется методам оптимизации объектной программы. Авторы проделали значительную работу по отбору и систематизации многочисленных результатов, полученных в последние годы; они строят изложение на едином подходе к задачам перевода и задачам оптимизации программы.

Книга предназначена тем, кто работает в области системного и теоретического программирования, преподает или изучает эти дисциплины, а также разработчикам математического обеспечения ЭВМ.

Редакция литературы по математическим наукам

А $\frac{20205-017}{041(01)-78}$ 17—78 © Перевод на русский язык, «Мир», 1978

ПРЕДИСЛОВИЕ

Конструирование компиляторов — один из первых больших разделов системного программирования, которые получают сейчас строгое теоретическое обоснование. В т. 1 этой книги изложены необходимые для такого обоснования сведения из математики и теории языков и основные методы проведения быстрого синтаксического анализа. Том 2 служит продолжением т. 1, но, за исключением гл. 7 и 8, он ориентирован на несинтаксические аспекты в конструировании компиляторов.

Материал в т. 2 излагается в основном так же, как и в т. 1, но доказательства здесь несколько более схематичны. Мы попытались по возможности облегчить чтение, включив в книгу много примеров, каждый из которых иллюстрирует одно или два понятия.

Так как акцент делается на идеи, а не на языковые или машинные подробности, то основанный на этой книге курс должен сопровождаться лабораторными занятиями по программированию, чтобы студент мог получить некоторые навыки в применении обсуждаемых идей к практическим задачам. Для таких занятий можно порекомендовать упражнения на программирование, приведенные в конце разделов. Часть лабораторного курса должна быть посвящена вопросу генерации кода для таких конструкций языка программирования, как рекурсия, пересылка параметров, взаимодействие подпрограмм, адресация массивов, циклы и т. д.

О пользовании книгой

Книга возникла на основе записей лекций, прочитанных на старших курсах Принстонского университета и Стивенсовского технологического института. Материал т. 2 использовался в Стивенсовском институте как семестровый курс конструирования компиляторов, следующий за семестровым курсом, основанным на т. 1.

С точки зрения изучения конструирования компиляторов одни разделы книги, как мы полагаем, важнее других. При первом

чтении можно пропустить все доказательства, а также гл. 8 и разд. 7.4.3, 7.5.3, 9.3.3, 10.2.3 и 10.2.4.

Как и в т. 1, в конце каждого раздела помещены задачи и замечания по литературе. Задачи, не являющиеся ни проблемами для исследования, ни открытыми проблемами, мы приблизительно упорядочили в соответствии с уровнем их сложности. Для этого использовали звездочки: упражнения, не помеченные звездочкой, служат для проверки понимания основных определений; для решения упражнений с одной звездочкой требуется одна существенная догадка, а упражнения с двумя звездочками значительно сложнее.

Благодарности

В дополнение к благодарностям, выраженным в предисловии к т. 1, мы хотели бы поблагодарить Карела Чулика, Амелию Фонг, Майка Хэммера и Стива Джонсона за полезные замечания.

Альфред В. Ахо
Джефри Д. Ульман

7 МЕТОДЫ ОПТИМИЗАЦИИ СИНТАКСИЧЕСКИХ АНАЛИЗАТОРОВ

В этой главе мы рассмотрим различные методы, с помощью которых можно уменьшать размер и/или повышать скорость работы синтаксических анализаторов.

Сначала исследуем вопрос об уменьшении объема памяти, требующейся для матриц предшествования. Мы увидим, что в определенных случаях, среди которых многие представляют практический интерес, матрицу предшествования размера $m \times n$ удается заменить двумя векторами длины m и n соответственно. Кроме того, покажем, как видоизменять матрицу предшествования, не затрагивая построенный по ней алгоритм разбора типа „перенос—свертка".

Затем мы опишем, как по грамматике слабого предшествования можно механически получить синтаксический анализатор на языке Флойда—Эванса, а потом рассмотрим различные приемы, применяемые для уменьшения размера полученного анализатора.

Наконец, подробно изучим различные преобразования, с помощью которых можно уменьшить размер LR-анализатора без ослабления его способности обнаруживать ошибки. Детально исследуются метод „простых LR" Де Ремера и метод расщепления грамматики Кореньяка.

Методы, описанные в этой главе, показывают, какие типы оптимизации возможны для анализаторов, построенных с помощью приемов гл. 5 тома 1. Возможны и многие другие типы оптимизации, но законченного „каталога" их не существует. Читателям, желающим получить только общее представление о методах оптимизации синтаксических анализаторов, рекомендуем помещенный в конце главы обзор ее содержания.

7.1. ФУНКЦИИ ПРЕДШЕСТВОВАНИЯ

Матрицу с элементами —1, 0, +1 или „пусто" будем называть *матрицей предшествования*. Матрицы предшествования очевидным образом применяются при построении алгоритмов

разбора, использующих технику предшествования. Например, можно применить матрицу предшествования для представления отношений предшествования Вирта—Вебера для грамматики предшествования, сопоставив

-1	с	$\lessdot$
0	с	$\doteq$
$+1$	с	$\gtrdot$
пусто	с	**ошибкой**

Матрицей предшествования можно также воспользоваться для того, чтобы представить шаги алгоритма разбора типа „перенос—свертка". Одно из таких представлений можно получить, сопоставив

-1	с	**переносом**
0	с	**ошибкой**
$+1$	с	**сверткой**

В этом разделе мы покажем, как матрицу предшествования часто удается представить в сжатой форме парой векторов, называемых функциями предшествования.

7.1.1. Теорема о представлении матрицы

Пусть M—матрица предшествования размера $m \times n$. Будем говорить, что пара (f, g) векторов с целочисленными компонентами *представляет* M, если

(1) $f=(f_1, f_2, \ldots, f_m)$,
(2) $g=(g_1, g_2, \ldots, g_n)$,
(3) $f_i < g_j$, если $M_{ij}=-1$,
$f_i = g_j$, если $M_{ij}=0$,
$f_i > g_j$, если $M_{ij}=+1$.

Можно использовать f и g вместо M следующим образом. Для того чтобы определить M_{ij}, отыщем f_i и g_j. Если $f_i < g_j$, $f_i = g_j$ или $f_i > g_j$, положим $M_{ij}=-1$, 0 или $+1$ соответственно. Заметим, что при таком использовании f и g вместо M у матрицы не будет пустых элементов, потому что между f_i и g_j всегда выполняется одно из отношений $<$, $=$ или $>$.

Векторы f и g будем называть *функциями предшествования* для M. Используя f и g для представления M, можно сократить объем памяти, требующейся для матрицы предшествования, с $m \times n$ до $m+n$ элементов. Следует, однако, отметить, что не для всякой матрицы предшествования функции предшествования существуют.

Пример 7.1. Рассмотрим грамматику простого предшествования G с правилами

$$S \to aSc \mid bSc \mid c$$

Отношения предшествования Вирта—Вебера для G образуют матрицу, изображенную на рис. 7.1. Мы будем называть ее *матрицей отношений предшествования Вирта—Вебера*, чтобы избежать путаницы с термином „матрица предшествования". Отношения предшествования, показанные на рис. 7.1, можно представить также в виде матрицы предшествования (рис. 7.2). Можно далее

	S	a	b	c	\$
S				$\doteq$	
a	$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$	
b	$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$	
c				$\gtrdot$	$\gtrdot$
\$		$\lessdot$	$\lessdot$	$\lessdot$	

Рис. 7.1. Матрица отношений предшествования Вирта — Вебера.

		1	2	3	4	5
		S	a	b	c	\$
1	S				0	
2	a	0	-1	-1	-1	
3	b	0	-1	-1	-1	
4	c				$+1$	$+1$
5	\$		-1	-1	-1	

Рис. 7.2. Матрица предшествования M.

представить эту матрицу предшествования функциями предшествования

$$f=(1, 0, 0, 2, 0)$$
$$g=(0, 1, 1, 1, 0)$$

Нетрудно убедиться, что это действительно функции предшествования для M. Например, $f_4=2$, $g_5=0$ и, значит, поскольку $f_4 > g_5$, f и g действительно представляют элемент M_{45} со значением $+1$.

Элемент M_{41} матрицы предшествования имеет значение „пусто". Однако $f_4=2$, а $g_1=0$. Таким образом, если мы будем использовать f и g для представления M, надо будет заменить значение M_{41} на $+1$ (так как $f_4 > g_1$). Аналогично пустые элементы M_{11}, M_{15}, M_{42} и M_{43} все будут иметь значение $+1$, а M_{12}, M_{13}, M_{25}, M_{35}, M_{51}, M_{55} получат значение 0.

Пустые элементы в исходной матрице предшествования указывают на ошибочные ситуации. Поэтому, если использовать функции предшествования для представления отношений предшествования, утратится возможность обнаружения ошибки в том случае, когда не выполняется ни одно из трех отношений предшествования. Но эта ошибка все равно выявится при попытке

выполнить свертку, когда окажется, что нет правила с такой правой частью, которая находится в верхушке магазина. Тем не менее такая задержка в обнаружении ошибки может оказаться неприемлемо дорогой платой за удобство использования функций предшествования вместо матриц предшествования; это зависит от того, насколько важно раннее обнаружение ошибки в конкретном компиляторе. □

Пример 7.2. Потерю возможности своевременно обнаруживать ошибки можно в значительной степени компенсировать, построив для грамматики предшествования алгоритм разбора типа „перенос—свертка", в котором с отношениями $\lessdot$ и $\doteq$ будет связана

		1	2	3	4
		a	b	c	\$
1	S			−1	
2	a	−1	−1	−1	
3	b	−1	−1	−1	
4	c			+1	+1
5	\$	−1	−1	−1	

Рис. 7.3. Матрица предшествования M'.

операция **перенос**, а с отношением $\gtrdot$ — операция **свертка**. Кроме того, для построения функции действия алгоритма разбора типа „перенос—свертка" требуются только отношения предшествования с областью определения $N \cup \Sigma \cup \{\$\}$ и множеством значений $\Sigma \cup \{\$\}$. Например, можно сопоставить $\lessdot$ и $\doteq$ с -1, $\gtrdot$ с $+1$ и получить из матрицы, приведенной на рис. 7.1, матрицу предшествования M' (рис. 7.3). Пустые элементы обозначают ошибочные ситуации. Можно показать, что векторы

$$f=(0,\ 0,\ 0,\ 2,\ 0) \quad \text{и} \quad g=(1,\ 1,\ 1,\ 0)$$

— функции предшествования для M'. Эти функции предшествования обладают тем преимуществом, что для пустых элементов M_{14}, M_{24}, M_{34} и M_{54} обе имеют значение 0 ($f_1=f_2=f_3=f_5=g_4=0$). Поэтому, обозначая нулем ошибочную ситуацию, мы сохраняем возможность обнаружения ошибки, заложенную в исходной матрице M'. Подробнее мы рассмотрим эту проблему в разд. 7.1.3. □

Сначала мы дадим алгоритм, который по данной матрице предшествования M находит функции предшествования для M,

если они существуют. В следующем разделе опишем модификацию этого алгоритма, где по данной матрице предшествования с элементами -1, $+1$ и „пусто" строятся функции предшествования для этой матрицы, которые по возможности чаще представляют пустые элементы нулями.

Прежде всего заметим, что если две строки матрицы предшествования M совпадают, то их можно слить в одну строку, и это не повлияет на факт существования функций предшествования для M. По той же причине можно слить совпадающие столбцы. Матрицу предшествования, в которой все совпадающие строки и столбцы слиты, назовем *приведенной* матрицей предшествования. Ясно, что, если сначала привести матрицу предшествования, легче будет строить функции предшествования.

Алгоритм 7.1. Вычисление функций предшествования.

Вход. Матрица M размера $m \times n$ с элементами -1, 0, $+1$ и „пусто".

Выход. Два целочисленных вектора $f=(f_1,\ \ldots,\ f_m)$ и $g=(g_1,\ \ldots,\ g_n)$, у которых

$$\begin{aligned} f_i &< g_j \quad \text{при} \quad M_{ij}=-1 \\ f_i &= g_j \quad \text{при} \quad M_{ij}=0 \\ f_i &> g_j \quad \text{при} \quad M_{ij}=+1 \end{aligned}$$

или „нет", если такие векторы не существуют.

Метод.

(1) Построим ориентированный граф, содержащий не более $m+n$ вершин и называемый *графом линеаризации* для M. Сначала пометим m вершин буквами F_1, F_2, ..., F_m, а оставшиеся n вершин буквами G_1, G_2, ..., G_n. В дальнейшем граф будет преобразовываться, причем в каждый момент будут существовать некоторая вершина $\hat{F}_i$, *представляющая* F_i, и некоторая вершина $\hat{G}_j$, *представляющая* G_j. Вначале $\hat{F}_i=F_i$ и $\hat{G}_j=G_j$ для всех i и j. Затем для всех i и j выполним шаг (2) или (3).

(2) Если $M_{ij}=0$, построим новую вершину N, сливая $\hat{F}_i$ с $\hat{G}_j$. Теперь N представляет все те вершины, которые ранее представлялись вершинами $\hat{F}_i$ и $\hat{G}_j$.

(3) Если $M_{ij}=+1$, проведем дугу из $\hat{F}_i$ в $\hat{G}_j$. Если $M_{ij}=-1$, проведем дугу из $\hat{G}_j$ в $\hat{F}_i$.

(4) Если полученный граф содержит циклы, выдаем сообщение „нет".

(5) Если граф линеаризации ациклический, положим значение f_i равным длине самого длинного пути, начинающегося в $\hat{F}_i$, а g_j — длине самого длинного пути, начинающегося в $\hat{G}_j$. □

На шаге (4) алгоритма 7.1 для выяснения, циклический или нет ориентированный граф G, можно применить следующий общий метод:

(1) Пусть G — исходный граф.

(2) Найти в данном графе вершину N, не имеющую прямых потомков. Если таких вершин нет, граф G — циклический. В противном случае удалить N [1]).

(3) Если полученный граф пуст, то граф G — ациклический. В противном случае повторить шаг (2).

Установив в некоторый момент, что граф ациклический, воспользуемся для нахождения на шаге (5) алгоритма 7.1 длины самого длинного пути, начинающегося в произвольной вершине, следующим методом разметки вершин.

Пусть G — ориентированный ациклический граф.

(1) Сначала пометить все вершины графа G нулями.

(2) Повторять шаг (3) до тех пор, пока метки графа G не перестанут изменяться. Метка при каждой вершине будет равна длине самого длинного пути, начинающегося в этой вершине.

(3) Выбрать в G некоторую вершину N. Пусть N имеет прямых потомков $N_1, N_2, \ldots, N_k$ с метками $l_1, l_2, \ldots, l_k$. Заменить метку вершины N на $\max\{l_1, l_2, \ldots, l_k\}+1$. (Если $k=0$, то меткой вершины N оставить 0.)

Проделать этот шаг для всех вершин графа G.

Ясно, что шаг (3) для каждой вершины повторяется не более l раз, где l — длина самого длинного пути в G.

Пример 7.3. Рассмотрим матрицу предшествования M, изображенную на рис. 7.4.

	1	2	3	4	5
1	−1	−1	0		−1
2			+1	−1	0
3	−1		+1	0	
4		−1	+1		
5				+1	+1

Рис. 7.4. Матрица предшествования.

Граф линеаризации, построенный по M, показан на рис. 7.5. Заметим, что на шаге (2) алгоритма 7.1 сливаются три пары вершин: (F_3, G_4), (F_2, G_5) и (F_1, G_3).

[1]) Строго говоря, надо удалить также и дуги, входящие в N. — *Прим. перев.*

Граф линеаризации — ациклический. В результате выполнения шага (5) алгоритма 7.1 получаются функции предшествования $f=(0, 1, 2, 1, 3)$ и $g=(3, 2, 0, 2, 1)$. Например, $f_5=3$, поскольку длина самого длинного пути, начинающегося в вершине F_5, равна 3. □

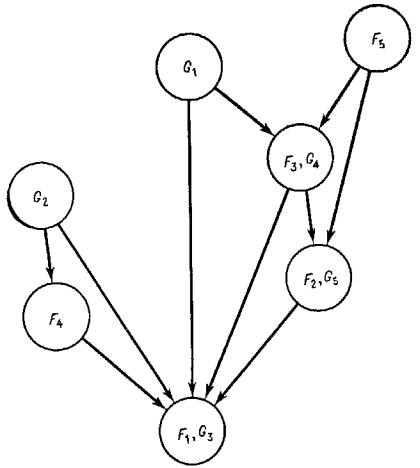


Рис. 7.5. Граф линеаризации.

Теорема 7.1. *Матрица предшествования имеет функции предшествования тогда и только тогда, когда ее граф линеаризации ациклический.*

Доказательство. *Достаточность.* Сначала отметим, что алгоритм 7.1 выдает на выходе функции f и g только тогда, когда граф линеаризации ациклический. Достаточно показать, что если f и g вычисляются при помощи алгоритма 7.1, то

(1) $M_{ij}=0$ означает, что $f_i=g_j$,

(2) $M_{ij}=+1$ означает, что $f_i>g_j$,

(3) $M_{ij}=-1$ означает, что $f_i<g_j$.

Утверждение (1) сразу следует из шага (2) алгоритма 7.1. Для того чтобы доказать утверждение (2), заметим, что при $M_{ij}=+1$ к графу линеаризации добавляется дуга $(\hat{F}_i, \hat{G}_j)$. Следовательно, $f_i>g_j$, так как, если граф линеаризации ацикличе-

ский, длина самого длинного пути, начинающегося в вершине $\hat{F}_i$, должна быть по крайней мере на единицу больше длины самого длинного пути, начинающегося в $\hat{G}_j$. Утверждение (3) доказывается аналогично.

Необходимость. Допустим, что матрица предшествования M имеет функции предшествования f и g, но граф линеаризации содержит цикл, состоящий из последовательности вершин $N_1, \ldots, N_k, N_{k+1}$, где $N_{k+1} = N_1$ и $k \geqslant 1$. Тогда на шаге (3) для всех i, $1 \leqslant i \leqslant k$, можно найти такие вершины H_i и I_{i+1}, что

(1) H_i и I_{i+1} — вершины, вначале помеченные буквами F и G,
(2) H_i и I_{i+1} представляются вершинами N_i и N_{i+1} соответственно,
(3) либо H_i есть F_r, I_{i+1} есть G_s и $M_{rs} = +1$, либо H_i есть G_r, I_{i+1} есть F_s и $M_{sr} = -1$.

Из правила (2) алгоритма 7.1 видно, что если вершины F_r и G_s представляются одной и той же вершиной N_i, то f_r должно равняться g_s, если f и g — функции предшествования для M. Предположим, что f и g — функции предшествования для M. Пусть $h_i = f_r$, если H_i есть F_r, и $h_i = g_r$, если H_i есть G_r. Положим h'_i равным f_r, если I_i есть F_r, и равным g_r, если I_i есть G_r. Тогда

$$h_1 > h'_2 = h_2 > h'_3 = \ldots = h_k > h'_{k+1}$$

Но так как N_{k+1} совпадает с N_1, то $h'_{k+1} = h_1$. Однако уже было показано, что $h_1 > h'_{k+1}$. Таким образом, матрица предшествования с циклическим графом линеаризации не может иметь функций предшествования. □

Следствие. *Алгоритм 7.1 вычисляет функции предшествования для M, если они существуют, и выдает ответ „нет" в противном случае.* □

7.1.2. Применения к разбору, основанному на операторном предшествовании

Для любой матрицы, элементы которой принимают не более трех значений, можно попытаться найти функции предшествования. Описанный метод применим независимо от того, что именно представляют элементы. Чтобы проиллюстрировать это, покажем сейчас, как применяются функции операторного предшествования для представления отношений операторного предшествования.

Пример 7.4. Рассмотрим нашу любимую грамматику G_0 с правилами

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid a$$

Матрица отношений операторного предшествования для G_0 изображена на рис. 7.6.

	$	(	+	*	a	)
$		⋖	⋖	⋖	⋖	
(		⋖	⋖	⋖	⋖	≐
+	⋗	⋖	⋗	⋖	⋖	⋗
*	⋗	⋖	⋗	⋗	⋖	⋗
a	⋗		⋗	⋗		⋗
)	⋗		⋗	⋗		⋗

Рис. 7.6. Матрица отношений операторного предшествования для грамматики G_0.

	1	2	3	4	5	
1		-1	-1	-1		$
2		-1	-1	-1	0	(
3	+1	-1	+1	-1	+1	+
4	+1	-1	+1	+1	+1	*
5	+1		+1	+1	+1	a,)
	$	(	+	*	)	
		a				

Рис. 7.7. Приведенная матрица предшествования M'.

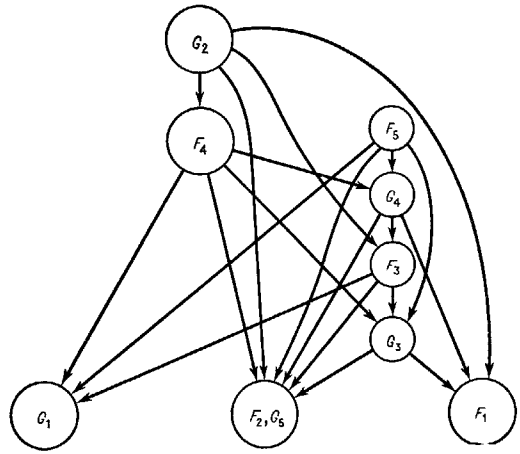


Рис. 7.8. Граф линеаризации для M'.

Если заменить ⋖ на -1, ≐ на 0 и ⋗ на $+1$, то получится приведенная матрица предшествования M' (рис. 7.7). В ней

слиты строки, помеченные символами a и), и столбцы, помеченные символами (и a. Граф линеаризации для M' изображен на рис. 7.8. С помощью этого графа находим функции предшествования $f'=(0, 0, 2, 4, 4)$ и $g'=(0, 5, 1, 3, 0)$ для матрицы M'. Следовательно, функциями предшествования для исходной матрицы будут $f=(0, 0, 2, 4, 4, 4)$ и $g=(0, 5, 1, 3, 5, 0)$. □

7.1.3. Функции слабого предшествования

Как уже отмечалось, с элементами -1, 0 и $+1$ матрицы из алгоритма 7.1 можно связать отношения предшествования Вирта—Вебера $\lessdot$, $\doteq$ и $\gtrdot$ соответственно. Если функции предшествования найдены, то отношение предшествования между символами X и Y можно определить, применив первую функцию к X, а вторую к Y. В этом случае для всех пар X и Y между символами X и Y выполняется какое-нибудь отношение предшествования, вследствие чего обнаружение ошибки задерживается до тех пор, пока не будет достигнут конец входной цепочки или пока не потребуется проделать невозможную свертку.

Однако метод функций предшествования можно применить для представления шагов работы алгоритма разбора типа „перенос — свертка", сохранив некоторую возможность обнаруживать ошибки, которая отражается пустыми элементами исходной матрицы отношений предшествования. Определим *матрицу слабого предшествования* M как $(m\times n)$-матрицу с элементами -1, $+1$ и „пусто". Элементы со значением -1, вообще говоря, обозначают переносы, элементы со значением $+1$ обозначают свертки, а пустые элементы—ошибки. С помощью такой матрицы можно описать функцию „перенос—свертка" алгоритма разбора типа „перенос—свертка" для грамматики слабого предшествования, грамматики (1,1)-предшествования и простой грамматики со смешанной стратегией предшествования.

Назовем векторы f и g *функциями слабого предшествования* для матрицы слабого предшествования M, если $f_i<g_j$ всякий раз, когда $M_{ij}=-1$, а $f_i>g_j$, когда $M_{ij}=+1$.

Условие $f_i=g_j$ можно использовать как указание на ошибочную ситуацию, представляемую пустым элементом M_{ij}. Вообще говоря, не всегда, когда M_{ij}—пустой элемент, удается удовлетворить условию $f_i=g_j$, но желательно сохранить, насколько это возможно, способность обнаружения ошибок, заложенную в исходной матрице.

Таким образом, можно рассматривать проблему нахождения функций слабого предшествования для матрицы слабого предшествования M как задачу отыскания функций, которые в наибольшем числе случаев порождают нули для пустых элементов матрицы M. Мы предпочитаем не заменять сразу все пустые

элементы матрицы слабого предшествования нулями, поскольку это ограничило бы число матриц слабого предшествования, имеющих функции слабого предшествования. Некоторые пустые элементы, возможно, придется заменить на -1 или $+1$ для того, чтобы существовали функции слабого предшествования (см. упр. 7.19). Кроме того, может оказаться, что к некоторым пустым элементам анализатор никогда не обращается, поэтому нет необходимости заменять их нулями.

В связи с этим приобретает важность понятие независимых вершин в ориентированном ациклическом графе. Говорят, что две вершины N_1 и N_2 ориентированного ациклического графа *независимы*, если нет пути из N_1 в N_2 или из N_2 в N_1.

Алгоритм 7.1 можно было бы непосредственно применять для построения функции слабого предшествования для матрицы M, но в том виде, в каком он описан, он не позволяет максимизировать число нулей, порождаемых для пустых элементов. Тем не менее мы используем три первых шага алгоритма 7.1 для построения графа линеаризации для M.

Из теоремы 7.1 известно, что M имеет функции слабого предшествования тогда и только тогда, когда граф линеаризации для M ациклический. Независимые вершины графа линеаризации определяют, какие пустые элементы матрицы M можно сохранить. Другими словами, равенство $f_i=g_j$ должно выполняться только для независимых вершин F_i и G_j. Конечно, если предпочтение отдано условию $f_i=g_j$, то могут быть другие пары независимых вершин, для которых отвечающие им числа нельзя сделать равными.

	a	(	)	+	*	$
E			$\doteq$	$\doteq$		
T			$\gtrdot$	$\gtrdot$	$\doteq$	$\gtrdot$
F			$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
a			$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
)			$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$				
+	$\lessdot$	$\lessdot$				
*	$\lessdot$	$\lessdot$				
$	$\lessdot$	$\lessdot$				

Рис. 7.9. Матрица отношений предшествования Вирта—Вебера для G_0.

	1	2	3	4	
1		-1			E
2		$+1$	-1	$+1$	T
3		$+1$	$+1$	$+1$	$F, a,)$
4	-1				$(, +, *, \$$
	a	)	*	$	
		+			

Рис. 7.10. Приведенная матрица слабого предшествования.

Пример 7.5. Матрица отношений предшествования Вирта—Вебера для грамматики G_0 показана на рис. 7.9. Столбцы, отве-

чающие нетерминалам, выброшены, поскольку мы будем пользоваться этой матрицей только в применении к алгоритму типа „перенос—свертка". Соответствующая приведенная матрица слабого предшествования показана на рис. 7.10, а построенный по ней граф линеаризации изображен на рис. 7.11. В этом графе вершины, помеченные буквами F_1 и G_4, независимы. Независимы также G_2 и G_4, а вершины F_1 и G_3 не являются независимыми. □

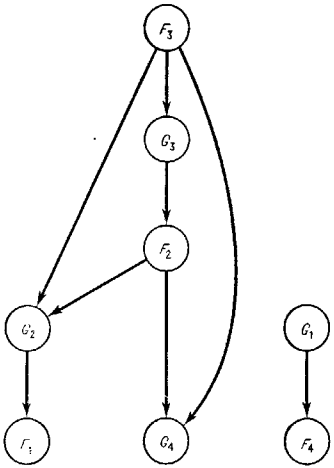


Рис. 7.11. Граф линеаризации.

Можно обобщить шаг (5) алгоритма 7.1 так, чтобы получались функции предшествования, максимизирующие число нулей, отвечающих пустым элементам. Определение компонент векторов предшествования состоит в присваивании числовых меток вершинам графа линеаризации. Всем вершинам из любого множества попарно независимых вершин присваивается одно и то же число, но у вершины, являющейся предком одной или более вершин, присваемое число должно быть больше, чем у любого из ее потомков.

Числа присваиваются вершинам следующим образом. Во-первых, множество вершин графа линеаризации разбивается на кисти независимых вершин так, чтобы общее число пар F, G, объединенных в кисть, было по возможности бо́льшим. Вообще говоря, различных разбиений множества на кисти может быть много и некоторые пары F, G могут оказаться предпочтительнее других. Эта часть процесса вполне может оказаться большой комбинаторной задачей.

Как только граф разбит на множество кистей, на последнем можно[1]) ввести линейный порядок $<$, при котором $C < C'$ для кистей C и C' тогда и только тогда, когда C содержит вершину, являющуюся потомком некоторой вершины из C'. Если $C_0, C_1, \ldots, C_k$ — последовательность кистей, расположенных в описанном порядке, то всем вершинам из C_0 присваивается 0, всем вершинам из C_1 присваивается 1 и т. д.

[1]) Не всегда. — *Прим. перев.*

Пример 7.6. Рассмотрим граф линеаризации для G_0, изображенный на рис. 7.11. Множество $\{F_1, F_4, G_4\}$ — одна из кистей независимых вершин. То же можно сказать и о множествах $\{F_4, G_2, G_4\}$, $\{F_2, G_1\}$, $\{G_1, G_3\}$ и $\{F_3, G_1\}$. Но кисть $\{G_1, G_3\}$ не подходит, так как обе вершины в ней помечены буквой G, и это не дает нулевого элемента в матрице слабого предшествования. Кисть $\{F_3, G_1\}$ может оказаться предпочтительнее, чем $\{F_2, G_1\}$, потому что соотношение $f_3 = g_1$ позволяет обнаружить ошибку, когда во входной цепочке появляются комбинации символов aa, $a($, $)a$ или $)($, в то время как с помощью соотношения $f_2 = g_1$ удается обнаружить ошибку только для пар Ta и $T($. Кроме того, заметим, что если мы обнаружим ошибку, когда появится aa, нам не удастся свернуть a в F, и потому комбинации символов Fa и Ta никогда не встретятся.

	a	(	)	+	*	$
E	−1	−1	−1	−1	−1	−1
T	−1	−1	+1	+1	−1	+1
F	0	0	+1	+1	+1	+1
a	0	0	+1	+1	+1	+1
)	0	0	+1	+1	+1	+1
(	−1	−1	0	0	−1	0
+	−1	−1	0	0	−1	0
*	−1	−1	0	0	−1	0
$	−1	−1	0	0	−1	0

Рис. 7.12. Окончательный вид матрицы предшествования для G_0.

Таким образом, одним из возможных вариантов объединения вершин в кисти будет $\{F_1\}$, $\{F_4, G_2, G_4\}$, $\{F_2\}$, $\{G_3\}$, $\{F_3, G_1\}$. Расположив кисти слева направо в описанном порядке, получим функции слабого предшествования $f = (0, 2, 4, 1)$ и $g = (4, 1, 3, 1)$. Эти функции определяют матрицу предшествования, изображенную на рис. 7.12. □

7.1.4. Преобразование матриц предшествования

Пример 7.6 показывает, что к некоторым элементам матрицы предшествования Вирта—Вебера, соответствующим ошибочным ситуациям, алгоритмы разбора типа „перенос—свертка" для грамматик простого и слабого предшествования (алгоритмы 5.12 и 5.14) никогда не обращаются. Если удастся выделить эти элементы и чем-нибудь заменить, можно будет при нахождении функций слабого предшествования, покрывающих максимально возможное число элементов, которые отвечают ошибочным ситуациям, не принимать их во внимание.

Чтобы понять, как можно модифицировать матрицы отношений предшествования, сначала договоримся, какие два алгоритма разбора типа „перенос—свертка" мы будем считать строго эквивалентными. Для записи алгоритмов разбора воспользуемся обозначениями, введенными в разд. 5.3 (том 1).

Определение. Пусть $\mathcal{A}_1 = (f_1, g_1)$[1]) и $\mathcal{A}_2 = (f_2, g_2)$ — два алгоритма разбора типа „перенос — свертка" для КС-грамматики $G = (N, \Sigma, P, S)$. Будем называть $\mathcal{A}_1$ и $\mathcal{A}_2$ *строго эквивалентными*, если их поведение одинаково для всех входных цепочек. Другими словами, если входная цепочка w принадлежит языку $L(G)$, то оба алгоритма разбора допускают w. Если w не принадлежит $L(G)$, то оба алгоритма выдают сообщение об ошибке после одинакового числа шагов и на одной и той же фазе. Если это произошло на фазе свертки, то ошибочное отношение находится после просмотра одного и того же числа символов в магазинах.

Проанализируем, какие пустые элементы матрицы канонических отношений предшествования Вирта — Вебера можно изменять, не влияя на поведение алгоритма разбора, построенного по этой матрице согласно алгоритму 5.12. Основная польза от такого анализа состоит в том, что удается найти хорошие кисти для функций слабого предшествования, подобно тому, как это делалось в предыдущем разделе. Пустые элементы, которые нельзя изменять, будем называть *существенными*. Доказываемая ниже теорема характеризует существенные пустые элементы.

Сначала введем некоторые обозначения. Предположим, что $G = (N, \Sigma, P, S)$ — КС-грамматика. Пусть M_c — матрица канонических отношений предшествования Вирта — Вебера для G. (Каноническими мы называем отношения, построенные прямо по определению.) Будем отмечать такие отношения предшествования нижним индексом c. Если между символами X и Y не выполняется ни одно из отношений предшествования Вирта — Вебера, будем писать $X ?_c Y$.

Затем можно построить произвольную матрицу M, состоящую из $\lessdot$, $\doteq$, $\gtrdot$ и пустых элементов. Если M имеет те же размеры, что и M_c, то будем называть M *матрицей отношений предшествования* для G. Если элемент (X, Y) матрицы M пустой, будем писать $X ? Y$.

Для построения алгоритма разбора типа „перенос — свертка" $\mathcal{A}_c = (f_c, g_c)$ для грамматики G, использующего отношения предшествования Вирта — Вебера, которые содержатся в M_c, можно применить алгоритм 5.12. Применяя алгоритм 5.12, можно также построить для G другой алгоритм разбора типа „перенос — свертка" $\mathcal{A} = (f, g)$, использующий отношения предшествования, которые содержатся в M. Корректность алгоритма разбора $\mathcal{A}_c$ гарантирует теорема 5.15, а аналогичной гарантии для $\mathcal{A}$ нет. Однако в теореме 7.2 устанавливается необходимое и достаточное условие для того, чтобы алгоритм $\mathcal{A}$ был строго эквивалентен алгоритму $\mathcal{A}_c$.

[1]) Здесь f_1 — это функция переноса — свертки, а g_1 — функция свертки.

Теорема 7.2. *Алгоритм $\mathcal{A}$ строго эквивалентен алгоритму $\mathcal{A}_c$ тогда и только тогда, когда для всех X и Y из $N \cup \Sigma \cup \{\$\}$, a, b из $\Sigma \cup \{\$\}$ и A из N удовлетворяются следующие четыре условия:*

(1) (а) *Если $X \lessdot_c Y$, то $X \lessdot Y$,*
(б) *если $X \doteq_c Y$, то $X \doteq Y$,*
(в) *если $X \gtrdot_c a$, то $X \gtrdot a$.*

(2) *Если $b ?_c a$, то $b ? a$.*

(3) *Если $A ?_c a$, то либо*
(а) *$A ? a$, либо*
(б) *для всех Z из $N \cup \Sigma$, для которых $A \to \alpha Z$ — правило из P, утверждение $Z \gtrdot_c a$ ложно.*

(4) *Если $X ?_c A$, то либо*
(а) *$X ? A$, либо*
(б) *для всех Z из $N \cup \Sigma$, для которых $A \to Z\alpha$ — правило из P, утверждение $X \lessdot_c Z$ ложно.*

Доказательство. *Достаточность.* Согласно условию (1), шаги алгоритмов $\mathcal{A}$ и $\mathcal{A}_c$ должны совпадать до тех пор, пока последний не обнаружит ошибку. Следовательно, достаточно показать, что если два алгоритма разбора достигают конфигурации $Q = (X_1 \ldots X_m, a_1 \ldots a_r, \pi)$ и $Q \vdash_{\mathcal{A}_c}$ **ошибка**, то $Q \vdash_{\mathcal{A}}$ **ошибка**. Более того, механизмы обнаружения ошибки в алгоритмах $\mathcal{A}$ и $\mathcal{A}_c$ совпадают.

Сначала предположим, что в конфигурации Q $f_c(X_m, a_1) =$ **ошибка**, но $f(X_m, a_1) \neq$ **ошибка**. Покажем, что это приводит к противоречию. Итак, предположим, что $X_m ?_c a_1$ верно, а $X_m ? a_1$ не выполняется. По условию (2) символ X_m должен быть нетерминалом. По условию (3) для всех Y, для которых правило $X_m \to \alpha Y$ принадлежит P, утверждение $Y \gtrdot_c a_1$ ложно.

Из анализа алгоритма разбора, использующего технику предшествования, видно, что нетерминал помещается в верхушку магазина только в том случае, если предыдущим шагом была свертка. Тогда в P есть такое правило $X_m \to \alpha Y$, что прежде, чем попасть в конфигурацию Q, оба алгоритма сделали шаг $(X_1 \ldots X_{m-1}\alpha Y, \alpha_1 \ldots a_r, \pi') \vdash Q$. Но это означает, что $Y \gtrdot_c a_1$; получили противоречие.

Другая возможность состоит в том, что в конфигурации Q $g_c(X_1 \ldots X_m, e) =$ **ошибка**, но $g(X_1 \ldots X_m, e) \neq$ **ошибка**. Надо рассмотреть лишь случай, когда $X_m \gtrdot_c a_1$, $X_m \gtrdot a_1$ и существует такое число s, что $X_s ?_c X_{s+1}$, причем $X_i \doteq_c X_{i+1}$ и $X_i \doteq X_{i+1}$ для $s < i < m$, но $X_s ? X_{s+1}$ не выполняется. Считаем, что X_{s+1} — нетерминал, поскольку $\mathcal{A}_c$ может поместить в магазин после X_s терминал только в том случае, когда $X_s \lessdot_c X_{s+1}$ или $X_s \doteq_c X_{s+1}$.

По условию (4), если $X_{s+1} \to Y\alpha$ принадлежит P, то не может быть $X_s \lessdot_c Y$. Но X_{s+1} мог бы появиться в магазине вслед за X_s, только если бы была возможна свертка некоторой цепочки $Y\alpha$

в X_{s+1}. Другими словами, должна существовать конфигурация $(X_1 \ldots X_s Y\alpha,\ b_1 \ldots b_k,\ \pi'')$, приводящая к Q и такая, что

$$(X_1 \ldots X_s Y\alpha,\ b_1 \ldots b_k,\ \pi'') \vdash_{\mathcal{A}_c} (X_1 \ldots X_s X_{s+1},\ b_1 \ldots b_k,\ \pi''i).$$

Но тогда было бы $X_s \lessdot_c Y$ вопреки условию (4).

Необходимость. Нетрудно показать, что если не удовлетворяется условие (1), алгоритмы разбора не являются строго эквивалентными. Поэтому опустим эту часть доказательства и займемся более сложными случаями.

Случай 1: Пусть не удовлетворяется условие (2). Это значит, что для некоторой пары b, a верно $b ?_c a$ и неверно $b ? a$. Так как G — грамматика простого предшествования (и, следовательно, приведенная), в $L(G)$ найдется слово вида wbx. Посмотрим, как происходит разбор цепочки wba согласно алгоритмам $\mathcal{A}_c$ и $\mathcal{A}$. Поскольку $wbx \in L(G)$, ни тот, ни другой алгоритм не сможет обнаружить ошибку, прежде чем символ a из wba не станет следующим входным символом. Таким образом, оба анализатора должны достичь некоторой конфигурации $(\$\alpha,\ ba\$,\ \pi)$, когда b помещается в верхушку магазина и образуется конфигурация $(\$\alpha b,\ a\$,\ \pi)$. Так как $b ?_c a$ верно, а $b ? a$ нет, алгоритмы $\mathcal{A}_c$ и $\mathcal{A}$ не являются строго эквивалентными.

Случай 2: Пусть нарушено (3). Это значит, что верно $A ?_c a$, неверно $A ? a$ и в P есть такое правило $A \to \alpha X$, что $X \gtrdot_c a$.

Так как G — приведенная грамматика, найдутся правовыводимая в G цепочка $\beta A w$ и цепочка $x \in \Sigma^*$, для которых $A \Rightarrow_r \alpha X \Rightarrow_r \beta_1 \Rightarrow_r \ldots \Rightarrow_r \beta_n \Rightarrow_r x$. Кроме того, найдется такая цепочка $y \in \Sigma^*$, что $\beta \Rightarrow^* y$. Согласно лемме 5.3, если Y — последний символ какой-либо из цепочек $\beta_1, \ldots, \beta_n$ или x, то $Y \gtrdot_c a$.

Заметим, что при разборе цепочки yxw первый символ цепочки w не будет перенесен в магазин, пока не выполнится свертка yx в βA. Поэтому разбор цепочки yxa будет происходить точно так же, как и разбор цепочки yxw, пока не будет достигнута конфигурация $(\$\beta A,\ a\$,\ \pi')$. Но $A ?_c a$ верно, а $A ? a$ нет, так что эти два алгоритма не являются строго эквивалентными.

Случай 3: Пусть нарушено условие (4). Другими словами, верно $X ?_c A$, неверно $X ? A$ и в P есть такое правило $A \to X\alpha$, что $X \lessdot_c Y$. Пусть $\beta A w$ — правовыводимая цепочка и $A \Rightarrow_r Y\alpha \Rightarrow_r \gamma_1 \Rightarrow_r \ldots \Rightarrow_r \gamma_n \Rightarrow x$. Далее, пусть $\delta X u$ — такая правовыводимая цепочка, что $\delta \Rightarrow^* y$ и $X \Rightarrow^* z$. Тогда, согласно лемме 5.3, для X и первых символов цепочек $\gamma_1, \ldots, \gamma_n$ и x выполняется отношение $\lessdot_c$. Кроме того, для последнего символа каждой правовыводимой цепочки в выводе $X \Rightarrow_r^* z$ и первого символа цепочки x выполняется отношение $\gtrdot_c$.

При разборе цепочки $yzxw$ будет достигнута конфигурация $(\$\delta X,\ xw\$,\ \pi)$, а затем конфигурация $(\$\delta XA,\ w\$,\ \pi')$. Анализаторы попытаются здесь произвести свертку по правилу, в результате применения которого в цепочке $\beta A w$ появился символ A. Если $X ?_c A$ верно, а $X ? A$ нет, снова приходим к противоречию тому, что анализаторы строго эквивалентны. □

Пример 7.7. Рассмотрим грамматику простого предшествования G с правилами

$$\begin{aligned} E &\to E + A \mid A \\ A &\to T \\ T &\to T * F \mid F \\ F &\to (B \mid a \\ B &\to E) \end{aligned}$$

Ясно, что $L(G) = L(G_0)$. Матрица канонических отношений предшествования Вирта — Вебера для G приведена на рис. 7.13.

	E	A	T	F	B	a	(	)	+	*	$
E								≐	≐		
A								≻	≻		≻
T								≻	≻	≐	≻
F								≻	≻	≻	≻
B								≻	≻	≻	≻
a								≻	≻	≻	≻
)								≻	≻	≻	≻
(	≺	≺	≺	≺	≐	≺	≺				
+		≐	≺	≺		≺	≺				
*				≐		≺	≺				
$	≺	≺	≺	≺		≺	≺				

Рис. 7.13. Матрица канонических отношений предшествования Вирта — Вебера для G_0.

Выясним, какие элементы на рис. 7.13 можно изменить в соответствии с теоремой 7.2. Условие (1) гласит, что непустые элементы не изменяются. Условие (2) утверждает, что все пустые элементы, расположенные на пересечении последних шести столбцов и последних шести строк, существенны.

По условию (3) $(E,\ \$)$ — существенный пустой элемент, так

как есть правило $E \to A$ и $A \gtrdot \$$ верно. Оставшиеся в последних шести столбцах пустые элементы несущественны, и, значит, они могут меняться произвольным образом.

По условию (4) пустой элемент ($, B) существен, так как есть правило $B \to E$) и $\$ \lessdot E$ верно. Оставшиеся в первых пяти столбцах пустые элементы можно произвольно изменять. □

Если алгоритм разбора типа „перенос—свертка" для обратимой грамматики слабого предшествования строить с помощью алгоритма 5.14, то можно показать, что анализаторы, аналогичные анализаторам $\mathcal{A}$ и $\mathcal{A}_c$ из теоремы 7.2, строго эквивалентны тогда и только тогда, когда удовлетворяются первые три условия теоремы 7.2[1]).

Пример 7.8. Применим условия (1)—(3) теоремы 7.2 к матрице отношений слабого предшествования для G_0, приведенной на рис. 7.9. Легко видеть, что все пустые элементы, стоящие в последних шести строках, существенны. Кроме них есть только один существенный пустой элемент, а именно (E, $), так как правило $E \to T$ принадлежит P и $T \gtrdot \$$ верно.

Изучим граф линеаризации, изображенный на рис. 7.11. Мы видим, что для него нет таких функций предшествования, что каждому существенному пустому элементу соответствует 0. В противном случае вершины F_4, G_2, G_3 и G_4, например, были бы объединены в одну кисть.

На этом можно закончить описание использования функций предшествования для построения анализаторов. Однако мы могли бы воспользоваться несколько более слабым определением эквивалентности анализаторов.

Строгая эквивалентность—очень сильное условие. На практике желательно, чтобы два алгоритма разбора типа „перенос—свертка" считались эквивалентными, если они допускают одну и ту же входную цепочку или оба выдают сообщение об ошибке в одном и том же месте ошибочной входной цепочки. Таким образом, может оказаться, что один анализатор сразу сообщил об ошибке, в то время как другому понадобилось сделать несколько сверток (но не переносов из входной цепочки), прежде чем выдать аналогичное сообщение. При таком условии, которое будет называться просто *эквивалентностью*, можно значительнее видоизменять отношения предшествования, сохраняя эквивалентность анализаторов (см. упр. 7.13).

При таком расширенном определении эквивалентности алгоритм разбора типа „перенос—свертка", использующий функции предшествования, приведенные в табл. 7.1, эквивалентен анализатору, построенному при помощи алгоритма 5.14 по матрице отношений слабого предшествования, приведенной на рис. 7.9. □

[1]) Напомним, что свертки в анализаторе слабого предшествования не зависят от матрицы предшествования.

Подробно эта более слабая форма эквивалентности будет изучаться в разд. 7.2—7.4.

Таблица 7.1

	E	T	F	a	(	)	+	*	$
f	0	2	5	5	4	5	4	4	4
g				5	5	1	1	3	0

УПРАЖНЕНИЯ

7.1.1. Для следующих грамматик найдите функции слабого предшествования или докажите, что таких функций нет:

(а) $S \to SA \mid A$
$A \to (S) \mid (\ )$

(б) $E \to E+T \mid +T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid a$

7.1.2. Покажите, что если M'—матрица, полученная из M перестановкой некоторых строк и/или столбцов, то векторы f и g, построенные для M' алгоритмом 7.1, будут получаться перестановкой компонент векторов, построенных для M.

7.1.3. Найдите функции предшествования для матрицы, изображенной на рис. 7.14.

			+1	+1	+1
			+1	+1	+1
−1	−1		+1	+1	+1
−1	−1	−1	+1	+1	+1
−1	−1	−1	−1	+1	+1
−1	−1	−1	−1	−1	0

Рис. 7.14. Матрица предшествования.

7.1.4. Постройте алгоритм, определяющий, имеет ли матрица такие фунции предшествования f и g, что $f=g$.

***7.1.5.** (а) Покажите, что описанный после алгоритма 7.1 метод, определяющий, является ли ориентированный граф ациклическим, действительно работает.

(б) Покажите, что можно сделать так, чтобы этот метод работал за время $0(n)+0(e)$, где n — число вершин, а e — число дуг данного графа. *Указание:* Выберите некоторую вершину. Раскрасьте все вершины, лежащие на пути, ведущем из этой вершины либо к листу, либо к уже раскрашенной вершине. Если встретится лист, удалите его, поднимитесь к его прямому предку, а затем продолжите процесс раскраски.

7.1.6. (а) Покажите, что метод разметки, приведенный после алгоритма 7.1, позволяет найти длину самого длинного пути, начинающегося в произвольной вершине.

(б) Покажите, что этот метод можно определить так, чтобы он требовал времени $0(n)+0(e)$, где n — число вершин, а e — число дуг данного графа.

7.1.7. Придумайте алгоритм, который по матрице M с элементами $-1, 0, +1$, „пусто" и по константе k выясняет, существуют ли такие векторы f и g, что

(1) если $M_{ij}=-1$, то $f_i+k<g_j$,
(2) если $M_{ij}=0$, то $|f_i-g_j|\leqslant k$,
(3) если $M_{ij}=+1$, то $f_i>g_j+k$.

Определение. Пусть M — матрица слабого предшествования. Последовательность целых чисел $i_1, i_2, \ldots, i_k$, где k — четное число, бо́льшее 3, назовем *циклом* матрицы M, если

(1) $M_{i_j i_{j+1}}=-1$ для нечетных j, $M_{i_{j+1} i_j}=+1$ для четных j и $M_{i_1 i_k}=+1$

или

(2) $M_{i_j i_{j+1}}=+1$ для нечетных j, $M_{i_{j+1} i_j}=-1$ для четных j и $M_{i_1 i_k}=-1$.

7.1.8. Покажите, что функции слабого предшествования для матрицы слабого предшествования существуют тогда и только тогда, когда она не содержит циклов.

7.1.9. Пусть M — матрица слабого предшествования и i, j, k, l — такие индексы, что

(1) $M_{lk}=M_{jl}=-1$, $M_{jk}=+1$ и M_{il} есть „пусто",

или

(2) $M_{ik}=M_{jl}=+1$, $M_{jk}=-1$ и M_{il} есть „пусто".

Пусть M' получается в результате замены в матрице M элемента M_{il} на -1 в случае (1) или на $+1$ в случае (2). Покажите, что f и g — функции слабого предшествования для M тогда и только тогда, когда они являются функциями слабого предшествования для M'.

Определение. Будем говорить, что две строки (два столбца) матрицы предшествования *совместимы*, если во всех случаях, когда их соответствующие элементы не совпадают, один из них пустой. Совместимые строки (столбцы) можно слить, заменив их новой строкой (столбцом) так, что непустые элементы каждой из прежних строк (столбцов) совпадают с соответствующими непустыми элементами новой строки (столбца).

7.1.10. Покажите, что операции слияния строк и столбцов сохраняют свойство матрицы не иметь линеаризующих функций.

Функции предшествования можно применять также для представления отношений $\lessdot$ и $\doteq$, используемых функцией свертки в алгоритме разбора типа „перенос — свертка", построенном по алгоритму 5.12. Сначала найдем матрицу слабого предшествования M, в которой -1 представляет $\lessdot$, $+1$ представляет $\doteq$, а пустые элементы представляют и $\gtrdot$, и **ошибку**. Затем попытаемся отыскать для M функции предшествования, опять пробуя заменить возможно большее число пустых элементов нулями.

7.1.11. Представьте отношения $\lessdot$ и $\doteq$, приведенные на рис. 7.13, функциями предшествования. С помощью теоремы 7.2 найдите существенные пустые элементы и попытайтесь сохранить их.

***7.1.12.** Покажите, что если принять определение строгой эквивалентности анализаторов слабого предшествования, то пустой элемент (X, Y) матрицы отношений предшествования Вирта — Вебера будет существенным тогда и только тогда, когда удовлетворяется одно из следующих условий:

(1) X и Y принадлежат $\Sigma \cup \{\$\}$,
(2) X принадлежит N, Y принадлежит $\Sigma \cup \{\$\}$ и есть такое правило $X \to \alpha Z$, что $Z \gtrdot_c Y$.

В следующих задачах термин „эквивалентность" понимается в смысле примера 7.8.

***7.1.13.** Пусть $\mathcal{A}_c$ и $\mathcal{A}$ — те же алгоритмы разбора типа „перенос — свертка" для грамматики простого предшествования, что и в теореме 7.2. Докажите, что они эквивалентны тогда и только

тогда, когда удовлетворяются следующие условия:

(1) а) Если $X \lessdot_c Y$, то $X \lessdot Y$,
(б) если $X \doteq_c Y$, то $X \doteq Y$,
(в) если $X \gtrdot_c a$, то $X \gtrdot a$.
(2) Если $b ?_c a$, то $b \lessdot a$ ложно.
(3) Если $A ?_c a$ и $A \lessdot a$ или $A \doteq a$, то нет такого вывода $A \Rightarrow_r \alpha_1 X_1 \Rightarrow_r \ldots \Rightarrow_r \alpha_m X_m$, $m \geqslant 1$, что $X_i ?_c a$ и $X_i \gtrdot a$ для $1 \leqslant i < m$ и $X_m \gtrdot_c a$ или X_m — терминал и $X_m \gtrdot a$.
(4) Если $A_1 \lessdot a$ или $A_1 \doteq a$ для некоторого a, то нет вывода $A_1 \Rightarrow A_2 \Rightarrow \ldots \Rightarrow A_m \Rightarrow B\alpha$, $m \geqslant 1$, символа X и правила $B \to Y\beta$, для которых

(а) $X ?_c A_i$, но $X \lessdot A_i$, где $2 \leqslant i \leqslant m$;
(б) $X ?_c B$, но $X \lessdot B$;
(в) $X \lessdot Y$.

7.1.14. Покажите, что анализатор, использующий функции предшествования из примера 7.8, эквивалентен каноническому анализатору предшествования для G_0.

7.1.15. Пусть M — матрица отношений предшествования, полученная из матрицы канонических отношений предшествования Вирта — Вебера M_c в результате замены некоторых пустых элементов на $\gtrdot$. Покажите, что анализаторы, построенные по M и M_c с помощью алгоритма 5.12 (или 5.14), эквивалентны.

***7.1.16.** Рассмотрим алгоритм разбора типа „перенос — свертка" для грамматики простого предшествования, в котором после каждой свертки проверяется, какое из отношений $\lessdot$ или $\doteq$ выполняется между символом, стоящим непосредственно слева от основы, и символом, в который свертывается основа. При каких условиях произвольная матрица отношений предшествования будет порождать анализатор, строго эквивалентный (или эквивалентный) анализатору такого типа, построенному по матрице канонических отношений предшествования Вирта — Вебера?

****7.1.17.** Покажите, что всякий КС-язык порождается грамматикой предшествования (не обязательно обратимой), для которой можно найти функции предшествования.

Проблемы для исследования

7.1.18. Придумайте эффективный алгоритм нахождения функций предшествования для грамматики слабого предшествования G, позволяющий построить анализатор, эквивалентный каноническому анализатору предшествования для G.

7.1.19. Разработайте хорошие процедуры нейтрализации ошибок, которыми можно было бы воспользоваться при разборе с помощью функций предшествования.

Упражнения на программирование

7.1.20. Разработайте программу, реализующую алгоритм 7.1.

7.1.21. Напишите программу, реализующую алгоритм разбора типа „перенос — свертка", который использует в качестве функций f и g функции предшествования.

7.1.22. Напишите программу, определяющую, является ли КС-грамматика грамматикой предшествования, для которой существуют функции предшествования.

7.1.23. Напишите программу, которая получает на вход грамматику простого предшествования G, имеющую функции предшествования, и выдает для G анализатор типа „перенос — свертка", использующий функции предшествования.

Замечания по литературе

Флойд [1963] применил функции предшествования для представления матрицы отношений операторного предшествования. Вирт и Вебер [1966] предложили применять их для представления отношений предшествования Вирта — Вебера. Алгоритмы вычисления функций предшествования были даны Флойдом [1963], Виртом [1965], Беллом [1969], Мартином [1972], а также Ахо и Ульманом [1972а].

Теорема 7.2 взята из работы Ахо и Ульмана [1972б], где содержатся также ответы на вопросы, сформулированные в упр. 7.1.13 и 7.1.15. Упр. 7.1.17 заимствовано у Мартина [1972].

7.2. ОПТИМИЗАЦИЯ АНАЛИЗАТОРОВ ФЛОЙДА — ЭВАНСА

Алгоритм разбора типа „перенос — свертка" демонстрирует принципиально простой метод разбора. Однако при реализации двух функций анализатора мы сталкиваемся с проблемами эффективности. В настоящем разделе будет исследован вопрос о том, как сконструировать алгоритм разбора типа „перенос — свертка" для обратимой грамматики слабого предшествования, пользуясь языком Флойда — Эванса. В центре внимания будут методы, позволяющие уменьшать объем получаемой программы на языке Флойда — Эванса, не затрагивая поведения анализатора. Хотя здесь рассматриваются только грамматики предшествования, методы этого раздела применимы также при реализации анализаторов для всех остальных классов грамматик, описанных в гл. 5 (том 1).

7.2.1. Механическое построение анализаторов Флойда — Эванса для грамматик слабого предшествования

Начнем с того, что покажем, как можно механически построить анализатор на языке Флойда—Эванса для обратимой грамматики слабого предшествования. Язык Флойда—Эванса описан в разд. 5.4.4. Алгоритм проиллюстрируем примером.

Как всегда, возьмем для примера грамматику слабого предшествования G_0 с правилами

$$
\begin{array}{ll}
(1) & E \to E + T \\
(2) & E \to T \\
(3) & T \to T * F \\
(4) & T \to F \\
(5) & F \to (E) \\
(6) & F \to a
\end{array}
$$

Отношения предшествования Вирта—Вебера для G_0 приведены на рис. 7.9. По каждой строке этой матрицы предшествования будем генерировать операторы анализатора на языке Флойда—Эванса. Будем пользоваться четырьмя типами операторов: оператор переноса, оператор свертки, оператор проверки и вычисляемый оператор перехода [1]). Операторам будем присваивать символьные метки, указывающие как на тип оператора, так и на верхний символ магазина. В таких метках буква S означает перенос, R—свертку, C—проверку и G—переход. За этими буквами в метках стоит символ, находящийся в верхушке магазина.

Сначала сгенерируем операторы переноса, а затем свертки.

Для грамматики слабого предшествования отношения предшествования $\lessdot$ и $\doteq$ означают перенос, а $\gtrdot$—свертку.

E-строка на рис. 7.9 порождает операторы

$$
\begin{array}{llll}
(7.2.1) & SE: & E\,|\,) \to E\,)\,| & *S) \\
 & & E\,|\,+ \to E\,+\,| & *S+ \\
 & & \$E\,|\,\$ \quad\quad | & \textbf{допуск} \\
 & & E\,| \quad\quad\quad | & \textbf{ошибка}
\end{array}
$$

[1]) Введение вычисляемого оператора перехода означает расширение языка Флойда—Эванса по сравнению с описанием его в разд. 5.4.4. Это расширение заключается в том, что следующая метка может быть выражением, содержащим знак #, который, как и в разд. 5.4.4, представляет неизвестный символ, соответствующий символу, расположенному в указанной позиции в стеке, или символу, на который происходит заглядывание вперед. Мы не будем останавливаться на деталях реализации такого оператора, но читатель может сам убедиться, что введенные здесь вычисляемые операторы перехода легко реализовать на доступной ему вычислительной машине.

Первый оператор означает, что если E находится в верхушке магазина и текущий входной символ есть), то) помещается в верхушку магазина, считывается следующий символ и происходит переход на оператор, помеченный S). Если первый оператор неприменим, проверяем, является ли текущий входной символ знаком +. Если второй оператор неприменим, опять проверяем, совпадает ли текущий входной символ с символом \$. Соответствующим действием анализатора будет тогда переход в заключительное состояние **допуск**, если в магазине содержится цепочка \$$E$. В противном случае выдается сообщение об ошибке. Заметим, что если в верхушке магазина находится E, то никакие свертки невозможны.

Поскольку вторая компонента метки представляет собой символ, находящийся в верхушке магазина, во многих случаях удается избежать его ненужной проверки, если известно, что́ это за символ. Зная, что в верхушке магазина находится символ E, можно было бы заменить операторы (7.2.1) на

$$
\begin{array}{llll}
SE: & |\,) \to)\,| & & *S) \\
 & |\,+ \to +\,| & & *S+ \\
 & \$\#\,|\,\$ \quad\quad | & & \textbf{допуск} \\
 & | \quad\quad\quad | & & \textbf{ошибка}
\end{array}
$$

Здесь оператор ошибки стоит последним не случайно. Когда E находится в верхушке магазина, текущим входным символом должен быть один из символов), + или \$. В противном случае мы получаем ошибку. Соответственно упорядочив операторы, можно сначала провести проверку на), потом на +, а затем на \$ и, если ни один из этих символов не является текущим входным символом, сообщить об ошибке.

Строка, отвечающая на рис. 7.9 символу T, порождает операторы

$$
\begin{array}{llll}
(7.2.2) & ST: & |\,* \to *\,| & *S* \\
 & RT: & E+T\,|\,\quad \to E\,| & CT \\
 & & T\,|\,\quad \to E\,| & CT \\
 & CT: & |\,) \quad\quad | & SE \\
 & & |\,+ \quad\quad | & SE \\
 & & |\,\$ \quad\quad | & SE \\
 & & | \quad\quad\quad | & \textbf{ошибка}
\end{array}
$$

Здесь $T \doteq *$ порождает первый оператор, $T \gtrdot)$, $T \gtrdot +$ и $T \gtrdot \$$ указывают, что, если в верхушке магазина находится символ T, надо выполнить свертку. Так как мы рассматриваем грамматику слабого предшествования, то в соответствии с леммой 5.4 при свертке всегда применяется правило с самой длин-

ной правой частью их всех применимых правил. Таким образом, сначала выясняем, содержится ли в верхушке магазина комбинация $E+T$. Если да, то заменяем $E+T$ на E. В противном случае свертываем T в E. Какой бы из RT-операторов ни применялся, известно, что в верхушке магазина находится символ T. Поэтому можно написать

$$RT:\quad E+\#| \to E| \qquad CT$$
$$\#| \to E| \qquad CT$$

и тем самым вновь избежать ненужной проверки верхнего символа магазина.

После выполнения свертки проверяется, была ли она законной. Иными словами, выясняется, что представляет собой текущий входной символ:), + или $. Для этого используется группа операторов проверки, помеченных CT. Если текущий входной символ — не), не + и не $, то выдается сообщение об ошибке. Порядок действий, при котором вначале делается свертка, а затем проверяется, надо ли было ее делать, может не всегда оказаться желательным, но, выполняя эти действия именно в таком порядке, мы получаем возможность совместить общие операции проверки.

Для проведения такой проверки введем вычисляемый оператор перехода вида

$$G:\quad \#| \quad | \quad S\#$$

указывающий, что верхний символ магазина должен стать последним символом метки.

Можно заменить операторы проверки в (7.2.2) последовательностью операторов

CT:	$\|)\ \|$		G
	$\|+\|$		G
	$\|\$\|$		G
	$\|\ \|$	**ошибка**	
G:	$\#\|\ \|$		$S\#$

Теперь можно применять эти операторы проверки и в других последовательностях. Например, если в G_0 свертка выполняется, когда в верхушке магазина находится T, то новым верхним символом магазина должен быть символ E. Таким образом, все операторы в группе CT передают управление на метку SE. Однако, вообще говоря, возможны свертки и в несколько различных нетерминалов, и тогда тот же вычисляемый оператор перехода работает с другими верхними символами магазина.

Наконец, ради удобства позволим операторам иметь более одной метки. Польза от такого допущения, которое несложно

реализовать, станет ясной в дальнейшем. Приведем алгоритм, использующий изложенные идеи.

Алгоритм 7.2. Построение анализатора Флойда — Эванса по обратимой грамматике слабого предшествования.

Вход. Обратимая грамматика слабого предшествования $G = (N, \Sigma, P, S)$.

Выход. Анализатор на языке Флойда — Эванса для G.

Метод.

(1) Найти для G отношения предшествования Вирта — Вебера.

(2) Линейно упорядочить множество $N \cup \Sigma \cup \{\$\}$; например, так: $\{X_1, X_2, \ldots, X_m\}$.

(3) Порождать операторы для $X_1, X_2, \ldots, X_m$ следующим образом. Допустим, что X_i не является начальным символом и либо $X_i \lessdot a$, либо $X_i \doteq a$ для всех a из $\{a_1, a_2, \ldots, a_j\}$ и $X_i \gtrdot b$ для всех b из $\{b_1, b_2, \ldots, b_l\}$. Кроме того, предположим, что $A_1 \to \alpha_1 X_i$, $A_2 \to \alpha_2 X_i$, $\ldots$, $A_k \to \alpha_k X_i$ — правила, в которых X_i — последний символ правой части, расположенные так, что $\alpha_p X_i$ не является суффиксом цепочки $\alpha_q X_i$ при $p < q$. Считаем, что $A_h \to \alpha_h X_i$ имеет номер p_h, $1 \leqslant h \leqslant k$. Затем порождаем операторы

SX_i:	$\|a_1 \to a_1\ \|$		$*Sa_1$
	$\|a_2 \to a_2\ \|$		$*Sa_2$
	.		
	.		
	.		
	$\|a_j \to a_j\ \|$		$*Sa_k$
RX_i:	$\alpha_1\#\| \to A_1\|$	**выдача** p_1	CX_i
	$\alpha_2\#\| \to A_2\|$	**выдача** p_2	CX_i
	.		
	.		
	.		
	$\alpha_k\#\| \to A_k\|$	**выдача** p_k	CX_i
	$\|\ \|$	**ошибка**	
CX_i:	$\|b_1\ \|$		G
	$\|b_2\ \|$		G
	.		
	.		
	.		
	$\|b_l\ \|$		G
	$\|\ \|$	**ошибка**	

Если $j=0$, то первый оператор группы RX_i имеет также метку SX_i. Если $k=0$, то оператор ошибки в группе RX_i имеет метку RX_i. Если X_i — начальный символ, то делаем все, как раньше, и добавляем в конец группы SX_i оператор

$$S\#|\$\ |\quad \textbf{допуск}$$

$S\$$ — начальное состояние анализатора.

(4) Добавить вычисляемый оператор перехода

$$G:\quad \#|\ \ |S\#\quad \square$$

Пример 7.9. Рассмотрим грамматику G_0. По строке F матрицы предшествования можно получить операторы

SF:	RF[1]):	$T*\#\|\to T\|$	**выдача 3**	CF
		$\#\|\to T\|$	**выдача 4**	CF
		$\|\ \ \|$	**ошибка**	
	CF:	$\|)\ \ \|$		G
		$\|+\ \ \|$		G
		$\|*\ \ \|$		G
		$\|\$\ \ \|$		G
		$\|\ \ \|$	**ошибка**	

Заметим, что третий оператор не нужен, потому что сравнение, выполняемое вторым оператором, всегда дает положительный результат. В принципе можно включить в алгоритм 7.2 проверку, позволяющую не порождать ненужные операторы. В дальнейшем мы будем считать, что ненужные операторы не порождаются.

По строке a получаем операторы

Sa:	Ra:	$\#\|\ \ \to F\|$	**выдача 6**	Ca
	Ca:	$\|)\ \ \|$		G
		$\|+\ \ \|$		G
		$\|*\ \ \|$		G
		$\|\$\ \ \|$		G
		$\|\ \ \|$	**ошибка**	

Отметим, что операторы проверки для a совпадают с операторами проверки для F.

В следующем разделе мы опишем в общих чертах алгоритм, объединяющий излишние операторы. На самом деле операторы

[1]) Отметим использование нескольких меток у одного оператора. Здесь группа SF пуста.

проверки, помеченные меткой CT, всегда можно слить с операторами, помеченными меткой CF, написав

Ca:	CF:	$\|*\ \ \|$	G
	CT:	$\|)\ \ \|$	G
		$\|+\ \ \|$	G
		$\|\$\ \ \|$	G
		$\|\ \ \|$	**ошибка**

В нашем алгоритме слияния будут также выполняться частные объединения такого рода.

Строка матрицы предшествования, помеченная символом (, порождает операторы

$S($:	$\|(\ \to (\|$	$*S($
	$\|a\to a\|$	$*Sa$
	$\|\ \ \|$	**ошибка**

Аналогичные операторы порождаются строками, помеченными символами $+$, $*$ и \$. $\square$

В качестве упражнения предлагаем проверить, что алгоритм 7.2 порождает для G корректный правый анализатор.

7.2.2. Усовершенствование анализаторов Флойда — Эванса

В данном разделе мы изучим методы, применяемые для уменьшения числа операторов переноса и проверки в анализаторе Флойда — Эванса, построенном с помощью алгоритма 7.2. Наш основной метод состоит в слиянии общих операторов переноса и общих операторов проверки. По ходу дела могут добавляться дополнительные операторы, вызывающие безусловные переходы, но мы будем считать, что их относительный вес сравнительно мал. Обсудим, как выполняется слияние операторов переноса; тот же метод применим и для слияния операторов проверки.

Пусть $G=(\mathrm{N}, \Sigma, P, S)$ — обратимая грамматика слабого предшествования и M — ее матрица отношений предшествования Вирта — Вебера. Матрица M определяет операторы переноса и проверки, возникающие в алгоритме 7.2.

По матрице предшествования M построим *совмещенную матрицу переносов* M_s следующим образом:

(1) Выбросим все элементы $\gtrdot$ и заменим $\doteq$ на $\lessdot$. (Так как мы интересуемся только переносами, отношения $\lessdot$ и $\doteq$ можно отождествить.)

(2) Если две или более строки полученной матрицы совпадают, заменим их одной строкой в M_s, причем свяжем с этой новой

строкой множество тех символов из $N \cup \Sigma \cup \{\$\}$, с которыми были связаны исходные строки.

(3) Выбросим все строки, не содержащие элементов $\lessdot$; полученную матрицу обозначим M_s.

Пример 7.10. Совмещенная матрица переносов для G_0, полученная из матрицы рис. 7.9, изображена на рис. 7.15. Эта сов-

	(	a	)	+	*	$
E			$\lessdot$	$\lessdot$		
T					$\lessdot$	
{(,+,*,$}	$\lessdot$	$\lessdot$				

Рис. 7.15. Совмещенная матрица переносов.

мещенная матрица переносов точно отражает ситуации, в которых анализатор выполняет перенос. □

По совмещенной матрице переносов M_s построим неупорядоченный помеченный ориентированный граф (A, R), называемый *графом переносов*, связанным с M_s:

(1) Для каждой строки матрицы M_s, помеченной символом Y, в A найдется вершина, помеченная символом Y.

(2) Существует одна дополнительная вершина, помеченная символом $\varnothing$, которая представляет фиктивную пустую строку.

(3) Если строка Y матрицы M_s *покрывается* строкой Z матрицы M_s (т. е. в каждом столбце, где Y имеет элемент $\lessdot$, строка Z тоже имеет элемент $\lessdot$), то дуга (Y, Z) содержится в R. Эта дуга помечена числом, равным числу столбцов, в которых Z имеет элемент $\lessdot$, а Y — нет. Заметим, что строка Y может быть пустой. Метку дуги (Y, Z) обозначим через $l(Y, Z)$.

Пример 7.11. Рассмотрим матрицу переносов M_s, приведенную на рис. 7.16. Граф переносов, связанный с M_s, изображен на рис. 7.17. □

Ясно, что граф переносов представляет собой ориентированный ациклический граф с единственным корнем $\varnothing$. Число операторов переноса, порождаемых алгоритмом 7.2, равно числу элементов переноса ($\lessdot$ или $\doteq$) матрицы предшествования M. Используя матрицу переносов M_s и сливая строки с совпадающими элементами переноса, можно уменьшить число требуемых операторов переноса. Метод заключается в построении для графа переносов *ориентированного остовного дерева* (*остова*) (т. е. подмножества дуг, которое образует дерево и включает в себя все вершины графа) наименьшего веса. Здесь *весом* остова называется сумма меток его дуг.

Путь из $\varnothing$ в Z, проходящий через Y, в графе переносов допускает следующую интерпретацию. Метка $l(\varnothing, Y)$ равна числу операторов переносов, построенных для строки Y матрицы M_s. Таким образом, число операторов переноса, порождаемых для строк Y и Z, равно $l(\varnothing, Y) + l(\varnothing, Z)$. Однако, если в графе есть

	a_1	a_2	a_3	a_4	a_5	a_6
Y_1	$\lessdot$		$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$
Y_2	$\lessdot$		$\lessdot$	$\lessdot$		$\lessdot$
Y_3	$\lessdot$			$\lessdot$		
Y_4	$\lessdot$					$\lessdot$
Y_5		$\lessdot$				

Рис. 7.16. Матрица переносов M_s.

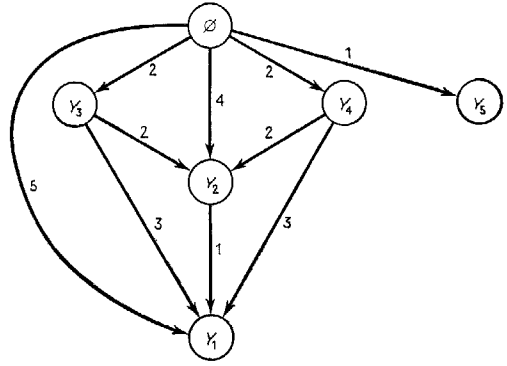


Рис. 7.17. Граф переносов, связанный с M_s.

путь из $\varnothing$ в Z, проходящий через Y, можно сначала сгенерировать операторы переноса для Y. Для того чтобы построить операторы переноса, соответствующие Z, можно взять операторы переноса для Y и написать перед ними те из операторов переноса для Z, которые еще не написаны. Тогда для строк Y и Z будет порождена такая последовательность операторов переноса:

SZ: Операторы переноса для элементов строки Z, не содержащихся в строке Y.

SY: Операторы переноса для элементов строки Y.

Число операторов переноса для Y и Z будет, таким образом, равно $l(\varnothing, Y)+l(Y, Z)=l(\varnothing, Z)$, а не $l(\varnothing, Y)+l(\varnothing, Z)$. Мы попутно получили операторы переноса для Y.

Можно обобщить этот прием и построить алгоритм, который, получая на вход произвольный ориентированный остов графа переносов, выдает множество операторов переноса, „соответствующих" этому остову. Число операторов переноса равно сумме меток всех дуг остова. Метод можно представить в виде следующего алгоритма.

Алгоритм 7.3. Построение множества операторов переноса по остову.

Вход. Матрица переносов M_s и остов ее графа переносов.

Выход. Последовательность операторов языка Флойда—Эванса.

Метод. Для каждой вершины Y остова, за исключением корня, строим последовательность операторов

$$
\begin{array}{lll}
L: & |a_1 \to a_1| & *Sa_1 \\
 & |a_2 \to a_2| & *Sa_2 \\
 & \vdots & \\
 & |a_n \to a_n| & *Sa_n \\
 & | \qquad | & L'
\end{array}
$$

Здесь L—метка строки Y в матрице M_s (т. е. множество меток $SX_1, \ldots, SX_m$, где $X_1, \ldots, X_m$—символы, строки которых в матрице предшествования образуют строку Y в M_s). L'—метка прямого предка вершины Y в остове; $a_1, \ldots, a_n$—столбцы, покрываемые строкой Y матрицы M_s, но не ее прямым предком.

Для вершины $\varnothing$ добавляем новый вычисляемый оператор перехода:

$$\varnothing: \quad \#| \qquad | \quad R\#$$

Операторы, соответствующие вершинам, можно располагать в любом порядке, но если оператор

$$| \qquad |' \quad L'$$

непосредственно предшествует оператору с меткой L', то его можно выбросить. □

Пример 7.12. Дерево, изображенное на рис. 7.18, является остовом графа переносов, изображенного на рис. 7.17.

С помощью алгоритма 7.3 по дереву рис. 7.18 порождается приведенная ниже последовательность операторов. Конечно, последовательность операторов, порождаемая алгоритмом 7.3, не

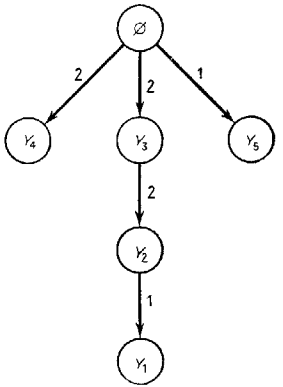


Рис. 7.18. Остовное дерево.

единственно возможная; допустимы и другие последовательности. Под SY_i подразумевается множество меток, соответствующее строке Y_i матрицы переносов.

$$
\begin{array}{lll}
SY_4: & |a_1 \to a_1| & *Sa_1 \\
 & |a_6 \to a_6| & *Sa_6 \\
 & | \qquad | & \varnothing \\
SY_1: & |a_5 \to a_5| & *Sa_5 \\
SY_2: & |a_3 \to a_3| & *Sa_3 \\
 & |a_6 \to a_6| & *Sa_6 \\
SY_3: & |a_1 \to a_1| & *Sa_1 \\
 & |a_4 \to a_4| & *Sa_4 \\
 & | \qquad | & \varnothing \\
SY_5: & |a_2 \to a_2| & *Sa_2 \\
\varnothing: & \#| \qquad | & R\#
\end{array}
$$

□

Теорема 7.3. *Алгоритм 7.3 строит такую последовательность операторов языка Флойда—Эванса, что замена ею последовательности операторов переноса, построенных алгоритмом 7.2, не отражается на поведении анализатора Флойда—Эванса.*

Доказательство. Заметим, что если сначала вычисляется последовательность операторов, полученная для вершины Y по алгоритму 7.3, то затем будут выполняться в точности те операторы, которые были сгенерированы для предков вершины Y. Легко показать, что эти операторы служат для проверки того,

что в следующей позиции входной цепочки стоит один из символов, столбцы которых покрываются строкой Y матрицы переносов.

Оператор с меткой $\varnothing$ гарантирует, что если не выполняется перенос, то происходит переход на соответствующую R-группу. □

Если дан граф переносов, то найти его остов, для которого алгоритм 7.3 порождает наименьшее число операторов переноса, удивительно просто. Легко видеть, что, не считая операторов безусловного перехода, число операторов, вырабатываемых алгоритмом 7.3 (каждый из них имеет вид $|a \to a|$ $*Sa$ для некоторого a), в точности равно сумме меток дуг дерева.

Алгоритм 7.4. Построение остова наименьшего веса по графу переносов.

Вход. Граф переносов (A, R) для матрицы предшествования M.

Выход. Остов (A, R'), для которого сумма $\sum_{(X, Y) \in R'} l(X, Y)$ принимает наименьшее значение.

Метод. Для каждой вершины Y из A, отличной от корня, выбрать такую вершину X, что величина $l(X, Y)$ принимает наименьшее значение для всех дуг, входящих в Y. Включить (X, Y) в R'. □

Пример 7.13. Остов на рис. 7.18 получен по графу переносов рис. 7.17 с помощью алгоритма 7.4. □

Теорема 7.4. *Число операторов переноса, которые строит алгоритм 7.3 по остову, минимально для данного графа переносов, если в качестве остова выбрано дерево, порождаемое алгоритмом 7.4.*

Доказательство. Поскольку каждая вершина, кроме корня графа (A, R), имеет единственного прямого предка, граф (A, R') должен быть деревом. В каждом остове графа (A, R) в каждую отличную от корня вершину входит одна дуга — отсюда сразу следует минимальность дерева (A, R'). □

Заметим, что, как и в случае матрицы переносов, по матрице предшествования M можно построить *матрицу сверток* M_r, выбросив из M все элементы, кроме $\lessdot$, и слив совпадающие строки. После этого полностью аналогично графу переносов определяется граф сверток и минимизируется число операторов проверки с помощью алгоритма, аналогичного алгоритму 7.4. Детали построения мы оставляем читателю. Приведем пример минимизации полного анализатора Флойда—Эванса для грамматики G_0.

Пример 7.14. Матрицы M_s и M_r для G_0 показаны в табл. 7.2. Здесь $Y = \{(, +, *, \$\}$ и $Z = \{F, a,)\}$.

Таблица 7.2

	(	a	)	+	*
E			$\lessdot$	$\lessdot$	
T					$\lessdot$
Y	$\lessdot$	$\lessdot$			

M_s

	)	+	*	$
T	$\gtrdot$	$\gtrdot$		$\gtrdot$
Z	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$

M_r

Применяя алгоритм 7.4 для слияния операторов переноса и операторов проверки, получаем следующий анализатор Флойда—Эванса для G_0. Начальный оператор помечен меткой $S\$$.

$S($:	$S+$:	$S*$:	$S\$$:			$(\to($			$*S($
						$a \to a$			$*Sa$
									$\varnothing$
			SE:			$)\to)$			$*S)$
						$+\to+$			$*S+$
					$\$\#$[1]	$\$$		допуск	
									$\varnothing$
			ST:			$*\to*$			$*S*$
			$\varnothing$:		$\#$				$R\#$
			RT:		$E+\#$		$\to E$	выдача 1	CT
					$\#$		$\to E$	выдача 2	CT
		SF:	RF:		$T*\#$		$\to T$	выдача 3	CF
					$\#$		$\to T$	выдача 4	CF
		Sa:	Ra:		$\#$		$\to F$	выдача 6	Ca
		$S)$:	$R)$:		$(E\#$		$\to F$	выдача 5	$C)$
RE:	$R($:	$R+$:	$R*$:	$R\$$:				ошибка	
		CF:	Ca:	$C)$:		$*$			G
				CT:					G
						$+$			G
						$\$$			G
								ошибка	
				G:	$\#$				$S\#$

[1]) Здесь этот оператор выполняется только в том случае, если в верхушке магазина находится E. Если бы это было не так, пришлось бы заменить $\#$ символом E (или, в общем случае, начальным символом).

Предпоследний оператор играет роль $\varnothing$ для операторов проверки. □

Можно улучшать анализаторы на языке Флойда—Эванса, исходя и из других соображений. Одним из таких улучшений может служить преобразование, при котором два оператора объединяются в один. Например, второй оператор анализатора для G_0 можно объединить с оператором, помеченным меткой Sa, и написать

$$|a \to F| \quad \textbf{выдача } 6 \quad Ca$$

Если надо слегка изменить поведение анализатора, можно внести в него дальнейшие изменения. Одним из них может быть задержка в обнаружении ошибок. Например, анализатор для G_0 из разд. 5.4.3 использует только 11 операторов, но в некоторых случаях обнаружение ошибок в нем задерживается.

Необходимо подчеркнуть, что на языке Флойда—Эванса можно записать любой из рассмотренных в гл. 5 анализаторов. Для того чтобы сделать это эффективно, можно применить приемы, аналогичные тем, которыми мы пользовались в настоящем разделе.

Наконец, встает вопрос реализации анализатора, записанного на языке Флойда—Эванса. Оператор языка может вызвать одну из следующих элементарных операций: чтение входного символа, сравнение символов, запись символов в магазин, выталкивание символов из магазина, порождение выходной цепочки. Реализовать эти операции весьма просто. Следовательно, можно написать программу, которая будет отображать анализатор на языке Флойда—Эванса в последовательность таких элементарных операций. Для выполнения этой последовательности элементарных операций достаточно небольшого интерпретатора.

УПРАЖНЕНИЯ

7.2.1. С помощью алгоритма 7.2 постройте анализаторы на языке Флойда—Эванса для следующих грамматик слабого предшествования:

(а) $S \to S+I \mid I$
$\quad I \to (S) \mid a(S) \mid a$

(б) $S \to 0S1 \mid 01$

(в) $E \to E+T \mid T$
$\quad T \to T*F \mid F$
$\quad F \to F\uparrow P \mid P$
$\quad P \to (E) \mid (a)$

7.2.2. Воспользуйтесь методами этого раздела для улучшения анализаторов, построенных в упр. 7.2.1.

7.2.3. Найдите матрицы переносов и сверток для матрицы слабого предшествования, приведенной на рис. 7.19.

	X_1	X_2	X_3	X_4	X_5	X_6
X_1	$\lessdot$	$\lessdot, \doteq$	$\doteq$	$\gtrdot$		$\gtrdot$
X_2		$\lessdot$	$\lessdot$	$\doteq$	$\gtrdot$	$\gtrdot$
X_3	$\gtrdot$	$\doteq$	$\lessdot, \doteq$	$\gtrdot$		$\gtrdot$
X_4	$\gtrdot$	$\doteq$	$\doteq$	$\doteq$	$\lessdot$	
X_5		$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
X_6	$\lessdot$	$\lessdot$	$\gtrdot$	$\doteq$	$\doteq$	
X_7	$\doteq$	$\lessdot, \doteq$	$\gtrdot$		$\gtrdot$	$\gtrdot$
X_8		$\lessdot$	$\gtrdot$			$\lessdot$

Рис. 7.19. Матрица слабого предшествования.

7.2.4. Постройте графы переносов и сверток для матриц переносов и сверток, найденных в упр. 7.2.3.

7.2.5. Примените алгоритм 7.3 для нахождения кратчайшей последовательности операторов переноса и проверки для графов, построенных в упр. 7.2.4.

***7.2.6.** Разработайте алгоритм построения детерминированного левого анализатора на языке Флойда—Эванса для LL(1)-грамматики.

7.2.7. С помощью алгоритма, разработанного в упр. 7.2.6, напишите на языке Флойда—Эванса левый анализатор для LL(1)-грамматики

$$E \to TE'$$
$$E' \to +TE' \mid e$$
$$T \to FT'$$
$$T' \to *FT' \mid e$$
$$F \to (E) \mid a$$

***7.2.8.** Пользуясь методами этого раздела, улучшите анализатор, построенный в упр. 7.2.7.

***7.2.9.** Разработайте алгоритм построения детерминированного правого анализатора на языке Флойда—Эванса для LR(1)-грамматики.

7.2.10. С помощью алгоритма, разработанного в упр. 7.2.9, постройте правые анализаторы для G_0 и грамматики из упр. 7.2.7.

***7.2.11.** Пользуясь методами этого раздела, улучшите анализаторы, построенные в упр. 7.2.10. Сравните полученные анализаторы с анализаторами, разработанными в примерах 5.47 и 7.14, а также с анализатором из упр. 7.2.8.

***7.2.12.** Можно ли по анализатору на языке Флойда—Эванса определить, является ли он корректным анализатором слабого предшествования для данной грамматики?

***7.2.13.** Разработайте алгоритм порождения программы на языке Флойда—Эванса, моделирующей детерминированный МП-преобразователь. Как можно улучшить полученную программу?

Проблемы для исследования

Ясно, что настоящий раздел не дает полного представления об изучаемом предмете и анализаторы, написанные на языке Флойда—Эванса, можно и далее существенно улучшать. Поэтому мы предлагаем следующие направления дальнейших исследований.

7.2.14. Изучите, как можно оптимизировать различные анализаторы типа „перенос—свертка" при реализации их на языке Флойда—Эванса. В частности, можно рассмотреть алгоритмы разбора LL(k)-, ОПК-грамматик, грамматик расширенного предшествования, простых грамматик со смешанной стратегией предшествования и LR(k)-грамматик.

7.2.15. Обобщите методы оптимизации анализаторов, допустив отсрочку в обнаружении ошибок, слияние операторов и/или другие разумные изменения программ на языке Флойда—Эванса.

7.2.16. Разработайте вариант языка Флойда—Эванса для реализации алгоритмов разбора. Ваш язык должен обладать следующим свойством: каждый оператор языка реализуется с помощью некоторого постоянного числа машинных команд на символ оператора языка. При разработке языка ориентируйтесь на „обыкновенную" вычислительную машину с прямым доступом.

Упражнения на программирование

7.2.17. Постройте набор элементарных операций, используемых для реализации операторов языка Флойда—Эванса. Напишите интерпретатор, выполняющий эти операции.

7.2.18. Постройте компилятор, принимающий на вход программу на языке Флойда—Эванса и выдающий для нее последовательность элементарных операций, которую может выполнить интерпретатор из упр. 7.2.17.

7.2.19. Напишите программу, которая строила бы анализаторы на языке Флойда—Эванса для некоторого полезного класса КС-грамматик.

7.2.20. Сконструируйте анализатор на языке Флойда—Эванса для одной из грамматик, данных в приложении к тому 1. Включите в него программу нейтрализации ошибок, вызываемую каждый раз, когда обнаруживается ошибка. Программа нейтрализации ошибок должна корректировать магазин и/или входную цепочку так, чтобы можно было возобновить нормальный разбор.

Замечания по литературе

Язык Флойда—Эванса и его варианты пользуются популярностью при написании синтаксических анализаторов. Методы построения анализаторов на этом языке разрабатывались рядом авторов, в том числе в работах Билза [1969], Билза и др. [1969], Де Ремера [1968], Эрли [1966], Хейнса и Шютте [1970]. Приемы построения анализаторов на языке Флойда—Эванса, описанные в этом разделе, впервые встречаются у Читэма [1967]. Ихбиа и Морзе [1970] предложили использовать в языке Флойда—Эванса вычисляемые операторы перехода и метод оптимизации, описанный в разд. 7.2.2. В работе Лафранса [1970] приведены некоторые методы нейтрализации ошибок для анализаторов на языке Флойда—Эванса.

7.3. ПРЕОБРАЗОВАНИЯ, ОПРЕДЕЛЕННЫЕ НА МНОЖЕСТВАХ LR(k)-ТАБЛИЦ

Оставшаяся часть главы посвящена обсуждению вопроса оптимизации LR(k)-анализаторов. Мы уделяем так много внимания LR(k)-анализаторам по двум причинам. Во-первых, LR(k)-грамматики представляют собой наиболее широкий естественный класс однозначных грамматик, для которых удается построить детерминированные анализаторы. Во-вторых, применяя методы оптимизации, можно строить LR(k)-анализаторы, успешно конкурирующие с анализаторами других типов.

В гл. 5 был дан алгоритм разбора для LR(k)-грамматик. Ядром алгоритма служит множество LR(k)-таблиц, управляющих работой анализатора. В разд. 5.2.5 приведен алгоритм,

применяемый для автоматического построения канонического множества LR(k)-таблиц для LR(k)-грамматики (алгоритм 5.7).

В то же время, как отмечалось, для грамматики, представляющей практический интерес, это каноническое множество LR(k)-таблиц при $k \geqslant 1$ может оказаться неприемлемо большим. Тем не менее LR(k)-алгоритм разбора, использующий каноническое множество LR(k)-таблиц (канонический LR(k)-анализатор), обладает рядом привлекательных свойств.

(1) Анализатор быстро работает. Входная цепочка длины n анализируется за cn тактов, где c — малая константа.

(2) Анализатор обладает хорошей способностью к обнаружению ошибок. Например, предположим, что xa — префикс некоторой цепочки рассматриваемого языка, а xab не является префиксом никакой цепочки. Обрабатывая входную цепочку вида $xaby$, канонический LR(1)-анализатор произведет разбор подцепочки x, прочтет символ a, а потом сообщит об ошибке. Таким образом, он объявит об ошибке, как только аванцепочкой впервые станет входной символ b. Вообще канонический LR(k)-анализатор при разборе входной цепочки слева направо сообщает об ошибке при первой же возможности.

Анализаторы предшествования не обладают способностью обнаруживать ошибки так рано. Например, при разборе цепочки $xaby$ анализатор предшествования прежде, чем сообщить об ошибке, может просмотреть сколь угодно много символов подцепочки y. (См. упр. 7.3.5.)

LL(k)-анализаторы совмещают высокое быстродействие и хорошую способность к обнаружению ошибок, свойственные LR(k)-анализаторам. Однако не каждый детерминированный язык имеет LL-грамматику, и, вообще говоря, для описания языка программирования и его перевода часто удается найти более „естественную" LR-грамматику. По этой причине вплоть до конца настоящей главы мы будем изучать методы построения компактных LR(k)-анализаторов. Многие из этих методов применимы также и к LL(k)-грамматикам.

В данном разделе мы рассмотрим LR(k)-анализаторы с точки зрения общей теории. Будем говорить, что два LR(k)-анализатора *эквивалентны*, если, получив на вход цепочку w, они оба либо допускают w, либо сообщают об ошибке, находясь на одном и том же символе цепочки w. Такая эквивалентность очень близка к „эквивалентности", с которой мы встретились в примере 7.8.

В данном разделе мы приведем несколько преобразований, применяемых для уменьшения размера LR(k)-анализатора и порождающих эквивалентный LR(k)-анализатор. В разд. 7.4 изложим методы, позволяющие по некоторым типам LR(k)-грамматик

непосредственно получать LR(k)-анализаторы, эквивалентные каноническому LR(k)-анализатору, но имеющие существенно меньшие размеры. Методы, рассматриваемые в этом разделе, можно применять и к каноническим анализаторам. Наконец, в разд. 7.5 мы опишем более подробную реализацию LR-анализатора, в которой общие операции просмотра можно совмещать.

7.3.1. ОБЩЕЕ ПОНЯТИЕ LR(k)-ТАБЛИЦЫ

Основу LR(k)-алгоритма разбора (алгоритма 5.7) образует множество LR(k)-таблиц. Вообще говоря, существует много различных множеств таблиц, которыми можно воспользоваться для построения эквивалентных анализаторов для одной и той же LR(k)-грамматики. Следовательно, можно поставить задачу отыскания множества таблиц, обладающего некоторыми нужными свойствами, например минимальностью.

Для того чтобы понять, как можно изменять множество LR(k)-таблиц, разберем поведение LR(k)-алгоритма разбора во всех подробностях. Этот алгоритм помещает LR(k)-таблицы в магазин. LR(k)-таблица, находящаяся в верхушке магазина, управляет поведением алгоритма разбора. Каждая таблица представляет собой пару функций $\langle f, g \rangle$. Напомним, что функция действия f по аванцепочке устанавливает, какое действие надо предпринять. Действием может быть: (1) перенос очередного входного символа в магазин, (2) свертка содержимого верхней части магазина в соответствии с известным правилом, (3) сообщение об окончании разбора, (4) сообщение о том, что во входной цепочке найдена синтаксическая ошибка. Вторая функция, а именно функция перехода g, вызывается после каждого переноса и каждой свертки. Функция перехода по символу грамматики либо определяет имя другой таблицы, либо выдает код ошибки.

В качестве примера рассмотрим LR(1)-таблицу, приведенную на рис. 7.20. В LR(k)-алгоритме разбора таблица может воздействовать на поведение анализатора двумя способами. Допустим, что таблица T_1 находится в верхушке магазина. Функция действия таблицы определяет работу анализатора. Например, если аванцепочкой является символ b, то анализатор выполняет свертку, применяя правило 3. Если же аванцепочкой является символ a, то анализатор помещает текущий входной символ (в данном случае a) в магазин и, поскольку $g(a) = T_7$, записывает в верхушку магазина вслед за a имя таблицы T_7[1]).

[1]) Как отмечалось в гл. 5, не обязательно записывать в магазин символы грамматики. Но так как поведение LR(k)-анализатора становится понятнее, если в магазине есть символы грамматики, мы будем считать в этом разделе, что символы грамматики тоже помещаются в магазин.

Сразу после свертки таблица задает другой тип работы анализатора. Предположим, что содержимое магазина есть $\alpha AT_2bT_2ST_3$ и таблица T_3 вызывает свертку согласно правилу $S \to bS$. Тогда анализатор удалит из магазина четыре символа (два символа грамматики и две таблицы), оставив в нем цепочку αAT [1]). Теперь

	Действие			*Переход*			
T_1:	a	b	e	S	A	a	b
	S	3	X	T_4	X	T_7	X

Рис. 7.20. LR(1)-таблица.

управление передано таблице T_1. В верхушку магазина помещается нетерминал S и вызывается функция перехода таблицы T_1, которая определяет, что в верхушку магазина нужно поместить таблицу $T_4 = g(S)$.

Мы будем считать, что узловыми при LR-разборе являются моменты, когда в верхушку магазина помещается новая таблица. Такую таблицу назовем *управляющей*. Изучим характеристики LR-анализатора в терминах последовательности управляющих таблиц. Если управляющая таблица T вызывает перенос, то следующая управляющая таблица определяется непосредственно с помощью функции переходов таблицы T. Если, с другой стороны, управляющая таблица T вызывает свертку, то следующая управляющая таблица определяется с помощью функции переходов $(i+1)$-й сверху таблицы, содержащейся в магазине, где i — длина правой части правила, применяемого для выполнения свертки. Может показаться, что трудно определить, какая таблица находится в указанном месте, однако можно построить алгоритм, определяющий множество возможных таблиц.

Попытаемся установить, в каком случае два множества LR(k)-таблиц определяют эквивалентные анализаторы. Мы сформулируем критерии поведения анализаторов, из которых должно быть ясно, что понимать под эквивалентностью. Сначала дадим определение множества LR(k)-таблиц, обобщающее определение, приведенное в разд. 5.2.5. Теперь как элемент возможного действия, так и элемент возможного перехода может быть специальным символом φ, который можно интерпретировать как „несущественный" элемент. Мы увидим, что многие из элементов типа „ошибка" в LR(k)-таблицах никогда не используются, т. е. независимо от входной цепочки к некоторым элементам типа

[1]) Заметим, что когда каноническому LR(k)-анализатору предстоит выполнить свертку по правилу i, правая часть правила i всегда будет суффиксом цепочки символов грамматики, находящейся в магазине. Таким образом, в процессе разбора не возникает необходимости сравнивать правую часть правила с находящейся в магазине цепочкой символов грамматики.

„ошибка" LR(k)-анализатор никогда не обращается. Таким образом, можно произвольно менять эти элементы; при этом анализатор будет работать по-прежнему.

Определение. Пусть G — КС-грамматика. *Множеством* LR(k)-*таблиц* для G назовем пару $(\mathscr{T}, T_0)$, где $\mathscr{T}$ — множество таблиц для G и таблица T_0, называемая *начальной*, принадлежит $\mathscr{T}$. *Таблица* для G — это пара функций $\langle f, g\rangle$, где

(1) f — отображение из Σ^{*k} в множество, состоящее из элементов φ, **ошибка, перенос, допуск** и **свертка** i для всех правил с номерами i,

(2) g отображает $N \cup \Sigma$ в $\mathscr{T} \cup \{\varphi$, **ошибка**$\}$.

Если ясно, о какой таблице T_0 идет речь, будем писать вместо $(\mathscr{T}, T_0)$ просто $\mathscr{T}$. Каноническое множество LR(k)-таблиц, построенное в разд. 5.2.5, удовлетворяет данному здесь определению множества LR(k)-таблиц. Заметим, что в каноническом множестве LR(k)-таблиц элемент φ никогда не встречается.

Пример 7.15. Пусть G определяется правилами

(1) $S \to SA$
(2) $S \to A$
(3) $A \to aA$
(4) $A \to b$

Множество LR(1)-таблиц для G приведено на рис. 7.21.

Мы увидим, что анализатор, использующий множество таблиц на рис. 7.21, проводит разбор, не привлекая для этого

	Действие			*Переход*			
	a	b	e	S	A	a	b
T_1	S	S	X	X	T_3	T_1	T_2
T_2	4	4	4	φ	φ	φ	φ
T_3	2	φ	φ	T_1	X	T_2	φ

Рис. 7.21. Множество LR(1)-таблиц.

непосредственно грамматику G. Просто таблицы „подходят" грамматике, т. е. в них фигурируют только символы из G, и вызываемые свертки выполняются по правилам, действительно существующим в G. □

Переопределим LR(k)-алгоритм разбора, опираясь на понятие множества LR(k)-таблиц, введенное выше. Этот алгоритм по существу тот же, что и алгоритм 5.7, если элемент φ рассматривать как элемент ошибки. Для удобства определим алгоритм заново.

Определение. Пусть $(\mathscr{T}, T_0)$ — множество LR(k)-таблиц для КС-грамматики $G=(\mathrm{N}, \Sigma, P, S)$. *Конфигурацией* LR($k$)-анализатора для $(\mathscr{T}, T_0)$ назовем тройку $(T_0X_1T_1 \ldots X_mT_m, w, \pi)$, где

(1) T_i принадлежит $\mathscr{T}$, $0 \leqslant i \leqslant m$, и T_0 — начальная таблица;
(2) X_i принадлежит $\mathrm{N} \cup \Sigma$, $1 \leqslant i \leqslant m$;
(3) w принадлежит Σ^*;
(4) π — выходная цепочка, состоящая из номеров правил.

Первая компонента конфигурации представляет содержимое магазина, вторая — непросмотренную часть входной цепочки, третья — построенный к этому моменту разбор.

Начальная конфигурация анализатора — это конфигурация вида (T_0, w, e) для некоторой цепочки w из Σ^*. Как и раньше, такт алгоритма разбора будем представлять с помощью отношения $\vdash$, заданного на множестве конфигураций.

Предположим, что анализатор находится в конфигурации $(T_0X_1T_1 \ldots X_mT_m, w, \pi)$, где $T_m=\langle f, g\rangle$ — таблица из $\mathscr{T}$. Пусть $w \in \Sigma^*$ и $u=\mathrm{FIRST}_k(w)$. Это означает, что w представляет непросмотренную часть входной цепочки, а u — аванцепочку.

(1) Если $f(u)=$ **перенос** и $w=aw'$, где $a \in \Sigma$, то

$$(T_0X_1T_1 \ldots X_mT_m, aw', \pi) \vdash (T_0X_1T_1 \ldots X_mT_maT, w', \pi)$$

где $T=g(a)$. В этом случае выполняется перенос; при этом следующий входной символ a записывается в магазин, а за ним в верхушку магазина помещается таблица $g(a)$.

(2) Пусть $f(u)=$ **свертка** i и $A \to \gamma$ — правило с номером i, $|\gamma|=r$. Будем считать, что $r \leqslant m$ и $T_{m-r}=\langle f', g'\rangle$. Таблица T_{m-r} — это таблица, которой передается управление, когда из магазина удаляется цепочка $X_{m-r+1}T_{m-r+1} \ldots X_mT_m$. Тогда

$$(T_0X_1T_1 \ldots X_mT_m, w, \pi) \vdash (T_0X_1T_1 \ldots X_{m-r}T_{m-r}AT, w, \pi i)$$

где $T=g'(A)$. В этом случае выполняется свертка по правилу $A \to \gamma$. Номер этого правила приписывается к выходной цепочке, цепочка длины $2|\gamma|$ удаляется из верхушки магазина и заменяется цепочкой AT, где $T=g'(A)$ и g' — функция переходов верхней оставшейся в магазине таблицы. Заметим, что при свертке символы, удаляемые из магазина, не просматриваются. Поэтому возможна такая свертка, когда удаляемые символы грамматики не составляют правой части правила, управляющего сверткой [1]).

(3) Если $f(u)=\varphi$, **ошибка** или **допуск**, то нет такой очередной конфигурации C, что $(T_0X_1T_1 \ldots X_mT_m, w, \pi) \vdash C$.

[1]) Это не случится, если используется каноническое множество таблиц. Вообще говоря, такая ситуация нежелательна, и ее можно допустить только в том случае, когда есть уверенность, что ошибка будет вскоре все равно обнаружена.

Очередной конфигурации нет также и в следующих случаях: в правиле (1) $w=e$ (т. е. входная цепочка исчерпана) или $g(a)$ не является именем таблицы, в правиле (2) $r>m$ (т. е. в магазине недостаточно символов) или $g'(A)$ не является именем таблицы.

Назовем конфигурацию $(T_0X_1T_1 \ldots X_mT_m, w, \pi)$ *сигналом ошибки*, если очередной конфигурации нет. Исключение составляет конфигурация (T_0ST_1, e, π), которую в случае $T_1=\langle f, g\rangle$ и $f(e)=$ **допуск** будем называть *допускающей*.

Отношения $\vdash^i$, $\vdash^*$ и $\vdash^+$ определяются как обычно.

Конфигурацию C назовем *достижимой*, если $C_0 \vdash^* C$ для некоторой начальной конфигурации C_0. Приведем теперь алгоритм разбора.

Алгоритм 7.5. LR(k)-алгоритм разбора.

Вход. КС-грамматика $G=(\mathrm{N}, \Sigma, P, S)$, множество $(\mathscr{T}, T_0)$ LR(k)-таблиц для G и входная цепочка $w \in \Sigma^*$.

Выход. Последовательность правил π или сигнал ошибки.

Метод.

(1) Построить начальную конфигурацию (T_0, w, e).

(2) Пусть C — последняя построенная конфигурация. Построить такую очередную конфигурацию C', что $C \vdash C'$, и затем повторить шаг (2). Если очередной конфигурации нет, перейти к шагу (3).

(3) Пусть $C=(\alpha, x, \pi)$ — последняя построенная конфигурация. Если C — допускающая конфигурация, выдать π и остановиться. В противном случае сообщить об ошибке. □

Ясно, что этот алгоритм можно реализовать с помощью детерминированного МП-преобразователя с правым концевым маркером.

Если алгоритм 7.5 достигает допускающей конфигурации, то выходная цепочка π называется *разбором* для входной цепочки w. Разбор π называют *корректным*, если π — правый разбор для w в соответствии с грамматикой G. Подобно этому, множество LR(k)-таблиц называют *корректным для грамматики* G, если алгоритм 7.5 порождает корректный разбор для любой цепочки из $L(G)$ и не порождает разбора ни для какой цепочки w, не принадлежащей $L(G)$.

По теореме 5.12 каноническое множество LR(k)-таблиц для LR(k)-грамматики G корректно для G. Однако произвольное множество LR(k)-таблиц для грамматики G, разумеется, не обязательно корректно для G.

Пример 7.16. Проследим последовательность шагов алгоритма 7.5 на входной цепочке ab при условии, что алгоритм исполь-

зует множество LR(1)-таблиц, приведенное на рис. 7.21, а начальной таблицей служит T_1.

Начальная конфигурация есть (T_1, ab, e). Действием таблицы T_1 на аванцепочке a будет **перенос**, а переходом — T_1, так что

$$(T_1, ab, e) \vdash (T_1aT_1, b, e)$$

Действием таблицы T_1 на символе b также будет **перенос**, но переходом — T_2, поэтому

$$(T_1aT_1, b, e) \vdash (T_1aT_1bT_2, e, e)$$

Действием таблицы T_2 на символе e будет **свертка 4**; правило 4 — это $A \to b$. Таким образом, верхняя таблица и символ грамматики удаляются из $T_1aT_1bT_2$, в результате чего остается цепочка T_1aT_1. Переход таблицы T_1 на символе A есть T_3, так что

$$(T_1aT_1bT_2, e, e) \vdash (T_1aT_1AT_3, e, 4)$$

Действие таблицы T_3 на аванцепочке e есть φ. Другими словами, построить очередную конфигурацию не удается. Так как $(T_1aT_1AT_3, e, 4)$ не является допускающей конфигурацией, она — сигнал ошибки.

Поскольку $ab \in L(G)$, это множество таблиц, очевидно, не корректно для грамматики из примера 7.15. □

7.3.2. Эквивалентность множеств таблиц

Опишем теперь, что значит, что два множества LR(k)-таблиц эквивалентны. Самая слабая эквивалентность, которая может нас заинтересовать, означает, что алгоритм 7.5, работающий с двумя множествами таблиц, порождает один и тот же разбор для цепочек, принадлежащих языку $L(G)$, и что цепочки, не анализируемые при использовании одного множества таблиц, не анализируются и при использовании другого множества таблиц. Ошибочные ситуации могут обнаруживаться в разное время.

Самая сильная эквивалентность, которую можно рассматривать, требует, чтобы при работе с двумя множествами таблиц порождались одинаковые последовательности шагов разбора. Иными словами, пусть (T_0, w, e) и (T'_0, w, e) — начальные конфигурации для двух множеств таблиц $\mathscr{T}$ и $\mathscr{T}'$. Тогда для всех $i \geqslant 0$ при использовании таблиц из $\mathscr{T}$

$$(T_0, w, e) \vdash^i (T_0X_1T_1 \ldots X_mT_m, x, \pi)$$

тогда и только тогда, когда при использовании таблиц из $\mathscr{T}'$

$$(T'_0, w, e) \vdash^i (T'_0X'_1T'_1 \ldots X'_nT'_n, x', \pi')$$

причем $m = n$, $x = x'$, $\pi = \pi'$ и $X_i = X'_i$ для $1 \leqslant i \leqslant m$.

Каждое из этих определений позволяет разработать такие методы преобразований множеств таблиц, что соответствующие эквивалентности сохраняются. Здесь мы будем рассматривать некоторую промежуточную эквивалентность. Конечно, требуется, чтобы алгоритм 7.5, использующий одно множество таблиц, находил для входной цепочки w разбор тогда и только тогда, когда он находит этот же разбор для w, используя другое множество таблиц. Более того, как и в определении самой сильной эквивалентности, потребуем, чтобы алгоритм 7.5 выполнял одну и ту же последовательность шагов разбора для каждой входной цепочки независимо от того, какое множество таблиц определяет эти шаги. Однако разрешим при работе с одним множеством еще производить свертки, в то время как при работе с другим разбор уже может закончиться. Такое определение оправдывается следующими соображениями.

Если во входной цепочке есть ошибка, желательно, чтобы это обнаруживалось при использовании любого из множеств таблиц и чтобы при этом местонахождение ошибки было определено как можно точнее. Нежелательно, чтобы в то время, как одно множество таблиц позволяет обнаружить ошибку, другое множество требовало бы просмотра еще многих входных символов. Это связано с тем, что для создания разумных и понятных методов диагностики нужно, чтобы ошибка обнаруживалась как можно ближе к тому месту, где она находится.

В практических ситуациях, когда встречается ошибка, управление передается программе нейтрализации ошибок, которая изменяет оставшуюся часть входной цепочки и/или содержимое магазина так, чтобы анализатор мог продолжить разбор остатка входной цепочки и за один просмотр входа найти возможно больше ошибок. Было бы очень жаль, если бы работа анализатора, обработавшего значительные части входной цепочки до обнаружения ошибки, оказалось бессмысленной. Исходя из этого, введем понятие эквивалентности множеств таблиц. Оно представляет собой частный случай того неформального понятия эквивалентности, которое обсуждалось в разд. 7.1.

Определение. Пусть $(\mathscr{T}, T_0)$ и $(\mathscr{T}', T'_0)$ — два множества LR(k)-таблиц для КС-грамматики $G = (N, \Sigma, P, S)$. Пусть w — входная цепочка из Σ^*, $C_0 = (T_0, w, e)$, $C'_0 = (T'_0, w, e)$, а $C_0 \vdash C_1 \vdash C_2 \vdash \ldots$ и $C'_0 \vdash C'_1 \vdash \ldots$ — соответствующие последовательности конфигураций, построенные алгоритмом 7.5. Назовем множества $(\mathscr{T}, T_0)$ и $(\mathscr{T}', T'_0)$ *эквивалентными*, если для всех $i \geqslant 0$ и для произвольной цепочки w удовлетворяются следующие четыре условия:

(1) Если C_i и C'_i обе существуют, то их можно записать в виде $C_i = (T_0X_1T_1 \ldots X_mT_m, x, \pi)$ и $C'_i = (T'_0X_1T'_1 \ldots X_mT'_m, x, \pi)$,

т. е. пока существуют обе последовательности конфигураций, они совпадают во всем, за исключением имен таблиц.

(2) C_i — допускающая конфигурация тогда и только тогда, когда C_i' — допускающая конфигурация.

(3) Если конфигурация C_i определена, а C_i' нет, то вторые компоненты у C_{i-1} и C_i совпадают.

(4) Если конфигурация C_i' определена, а C_i нет, то вторые компоненты у C_{i-1}' и C_i' совпадают.

Условия (3) и (4) утверждают, что если при работе с одним множеством таблиц обнаруживается ошибка, то при работе с другим множеством после этого не должно быть чтения символов на входе, т. е. не выполняются действия **перенос**. Но условия (3) и (4) допускают, чтобы в то время, как одно из множеств вызывало остановку на несущественном действии или на действии **ошибка**, другое требовало выполнить одно или более действий **свертка**.

Отметим, что ни одно из множеств таблиц не обязано быть корректным для G. Тем не менее, если два множества таблиц эквивалентны и одно из них корректно для G, то другое также будет корректным для G.

Пример 7.17. Рассмотрим LR(1)-грамматику с правилами

$$(1)\quad S \to aSb$$
$$(2)\quad S \to ab$$

Множество $(\mathscr{T}, T_0)$ — каноническое множество LR(1)-таблиц для G — показано на рис. 7.22. На рис. 7.23 приведено другое множество LR(1)-таблиц для G — множество $(\mathscr{U}, U_0)$. Исследуем поведение LR(1)-алгоритма разбора, использующего $\mathscr{T}$ и $\mathscr{U}$, на входной цепочке abb. Используя $\mathscr{T}$, алгоритм разбора делает последовательность тактов

$$(T_0, abb, e) \vdash (T_0aT_2, bb, e)$$
$$\vdash (T_0aT_2bT_5, b, e)$$

Последняя конфигурация — сигнал ошибки. Используя множество таблиц $\mathscr{U}$, алгоритм разбора действует так:

$$(U_0, abb, e) \vdash (U_0aU_2, bb, e)$$
$$\vdash (U_0aU_2bU_4, b, e)$$
$$\vdash (U_0SU_1, b, 2)$$

Последняя конфигурация — ошибочная. Заметим, что канонический анализатор сообщает об ошибке, как только он впервые обнаружит во входной цепочке второй символ b. Анализатор, использующий множество таблиц $\mathscr{U}$, прежде чем сообщить об ошибке,

свертывает ab в S. Правда, второй символ b не записывается в магазин, поэтому условия эквивалентности не нарушаются.

Нетрудно показать, что множества $\mathscr{T}$ и $\mathscr{U}$ действительно эквивалентны. Существует алгоритм, позволяющий определить, эквивалентно ли произвольное множество LR(k)-таблиц для LR(k)-грамматики каноническому множеству LR(k)-таблиц для этой грамматики. Построение такого алгоритма оставляем в качестве упражнения. □

	Действие			Переход		
	a	b	e	S	a	b
T_0	S	X	X	T_1	T_2	X
T_1	X	X	A	X	X	X
T_2	S	S	X	T_3	T_4	T_5
T_3	X	S	X	X	X	T_6
T_4	S	S	X	T_7	T_4	T_8
T_5	X	X	2	X	X	X
T_6	X	X	1	X	X	X
T_7	X	S	X	X	X	T_9
T_8	X	2	X	X	X	X
T_9	X	1	X	X	X	X

Рис. 7.22. Множество таблиц $(\mathscr{T}, T_0)$.

	Действие			Переход		
	a	b	e	S	a	b
U_0	S	X	X	U_1	U_2	φ
U_1	φ	X	A	φ	φ	φ
U_2	S	S	X	U_3	U_2	U_4
U_3	φ	S	X	φ	φ	U_5
U_4	X	2	2	φ	φ	φ
U_5	X	1	1	φ	φ	φ

Рис. 7.23. Множество таблиц $(\mathscr{U}, U_0)$.

7.3.3. φ-недостижимые множества таблиц

Многие из элементов **ошибка** канонического множества LR(k)-таблиц никогда не используются LR(k)-алгоритмом разбора. Такие элементы **ошибка** можно заменить элементами φ, действительно несущественными в том смысле, что ни в какой достижимой конфигурации они не влияют на вычисление очередной конфигурации. Мы покажем, что все символы **ошибка** у функции переходов канонического множества таблиц можно заменить элементами φ и, если данная таблица может стать управляющей только сразу после свертки, аналогичную замену можно проделать и для функции действия.

Определение. Пусть $(\mathscr{T}, T_0)$ — множество LR(k)-таблиц, $k \geqslant 1$ и

$$C = (T_0X_1T_1X_2T_2 \ldots X_mT_m,\ w,\ \pi)$$

— любая достижимая конфигурация. Пусть $T_m = \langle f, g\rangle$ и $u = \mathrm{FIRST}_k(w)$. Будем говорить, что в $\mathscr{T}$ *нет достижимых эле-*

ментов φ, или, короче, $\mathscr{T}$ φ-*недостижимо*, если для любой конфигурации C справедливы следующие утверждения:

(1) $f(u) \neq \varphi$.

(2) Если $f(u) =$ **перенос**, то $g(a) \neq \varphi$, где a — первый символ цепочки u.

(3) Если $f(u) =$ **свертка** i, правило с номером i есть $A \to Y_1 \ldots Y_r$, $r \geqslant 0$ и $T_{m-r} = \langle f', g' \rangle$, то $g'(A) \neq \varphi$.

Неформально можно сказать, что множество LR(k)-таблиц φ-недостижимо, если все элементы φ, появляющиеся в таблицах, никогда не используются алгоритмом 7.5 при разборе цепочек. Приведем теперь алгоритм, заменяющий в каноническом множестве LR(k)-таблиц по возможности большее число элементов **ошибка** на φ так, чтобы полученное множество таблиц осталось φ-недостижимым.

Применяя этот алгоритм к каноническому множеству LR(k)-таблиц, можно отыскать и заменить на φ все элементы **ошибка**, которые никогда не используются LR(k)-алгоритмом разбора.

Алгоритм 7.6. Построение φ-недостижимого множества LR(k)-таблиц с максимально возможным числом элементов φ.

Вход. LR(k)-грамматика $G = (N, \Sigma, P, S)$, где $k \geqslant 1$, и каноническое множество $(\mathscr{T}, T_0)$ LR(k)-таблиц для G.

Выход. Эквивалентное множество $(\mathscr{T}', T_0')$ LR(k)-таблиц, где все неиспользуемые элементы **ошибка** заменены символами φ.

Метод.

(1) Для каждой таблицы $T = \langle f, g \rangle$ из $\mathscr{T}$ построить новую таблицу $\langle f', g' \rangle$, где

$$g'(X) = \begin{cases} g(X), & \text{если } g(X) \neq \textbf{ошибка} \\ \varphi & \text{в противном случае} \end{cases}$$

Пусть $(\mathscr{T}_1, T_0')$ — множество построенных таким образом таблиц.

(2) Множество таблиц $(\mathscr{T}', T_0')$ строится затем следующим образом:

(а) T_0' принадлежит $\mathscr{T}'$;

(б) для каждой таблицы $T = \langle f, g \rangle$ из $\mathscr{T}_1 - \{T_0'\}$ включить в $\mathscr{T}'$ таблицу $T' = \langle f', g \rangle$, где f' определяется так:

(i) $f'(u) = f(u)$ всякий раз, когда $f(u) \neq$ **ошибка**;

(ii) если $f(vb) =$ **ошибка** для некоторых $v \in \Sigma^{k-1}$ и $b \in \Sigma$ и для некоторого $a \in \Sigma$ существует такая таблица $\langle f_1, g_1 \rangle$ из $\mathscr{T}_1$, что $f_1(av) =$ **перенос** и $g_1(a) = T$, то $f'(vb) =$ **ошибка**;

(iii) если $f(u) =$ **ошибка** для некоторой цепочки $u \in \Sigma^{*(k-1)}$ и для некоторого $a \in \Sigma$ существует такая таблица $\langle f_1, g_1 \rangle$ из $\mathscr{T}_1$, что $f_1(au) =$ **перенос** и $g_1(a) = T$, то $f'(u) =$ **ошибка**;

(iv) в противном случае $f'(u) = \varphi$. □

На шаге (1) алгоритма 7.6 все элементы **ошибка** в функциях переходов заменяются на элементы φ, так как при $k \geqslant 1$ канонический LR(k)-анализатор всегда обнаруживает ошибку сразу после операции переноса. Следовательно, значение **ошибка** у функции переходов никогда не используется.

Шаг (2) алгоритма 7.6 заменяет на φ значение **ошибка** у функции действия таблицы T, если нет такой таблицы $\langle f_1, g_1 \rangle$ и такой аванцепочки au, что $f_1(au) =$ **перенос** и $g_1(a) = T$. При таких условиях таблица T может появиться в верхушке магазина только сразу после свертки. Однако, если ошибка есть, канонический LR(1)-анализатор сообщит о ней перед сверткой. Таким образом, в таблицах, аналогичных T, никакие элементы **ошибка** не запрашиваются и, значит, могут рассматриваться как несущественные.

Отметим еще раз, что алгоритм 7.6 в том виде, как он был определен, работает только для множеств LR(k)-таблиц с $k \geqslant 1$. В случае LR(0)-грамматики аванцепочка всегда пуста. Поэтому все ошибки должны распознаваться при обращении к функции переходов. Следовательно, в множестве LR(0)-таблиц не все элементы **ошибка** у функций переходов несущественны. В качестве упражнения предлагаем выяснить, какие из этих элементов несущественны.

Пример 7.18. Пусть G — LR(1)-грамматика с правилами

(1) $S \to SaSb$

(2) $S \to e$

Каноническое множество LR(1)-таблиц для G показано на рис. 7.24, *а*, а множество таблиц, полученное в результате применения алгоритма 7.6, — на рис. 7.24, *б*.

Заметим, что на рис. 7.24, *б* все элементы **ошибка** в правых половинах таблиц (у функций переходов) заменены элементами φ. У функций действия таблица T_0, согласно правилу (2а) алгоритма 7.6, оставлена без изменений. Действия **перенос** встречаются только в таблицах T_3 и T_6; они приводят к тому, что управляющей становится одна из таблиц T_4, T_5 или T_7. Поэтому элементы **ошибка** у функций действия этих таблиц не меняются. В ряде других мест **ошибка** заменена на φ. □

Теорема 7.5. *Множество таблиц $\mathscr{T}'$, построенное алгоритмом 7.6, φ-недостижимо и эквивалентно каноническому множеству $\mathscr{T}$.*

Доказательство. Эквивалентность множеств $\mathscr{T}$ и $\mathscr{T}'$ очевидна, так как единственное изменение, которое вносится в множество $\mathscr{T}$, состоит в том, что элементы **ошибка** заменяются на φ,

	Действие			Переход		
	a	b	e	S	a	b
T_0	2	X	2	T_1	X	X
T_1	S	X	A	X	T_2	X
T_2	2	2	X	T_3	X	X
T_3	S	S	X	X	T_4	T_5
T_4	2	2	X	T_6	X	X
T_5	1	X	1	X	X	X
T_6	S	S	X	X	T_4	T_7
T_7	1	1	X	X	X	X

а

	Действие			Переход		
	a	b	e	S	a	b
T_0	2	X	2	T_1	φ	φ
T_1	S	φ	A	φ	T_2	φ
T_2	2	2	X	T_3	φ	φ
T_3	S	S	φ	φ	T_4	T_5
T_4	2	2	X	T_6	φ	φ
T_5	1	X	1	φ	φ	φ
T_6	S	S	φ	φ	T_4	T_7
T_7	1	1	X	φ	φ	φ

б

Рис. 7.24. Множество LR(1)-таблиц до и после применения алгоритма 7.6: *а* — каноническое множество LR(1)-таблиц; *б* — φ-недостижимое множество таблиц.

а LR(k)-алгоритм разбора не отличает ошибку от φ[1]). Покажем теперь, что если алгоритм 7.5, использующий множество таблиц $\mathscr{T}'$, обращается к элементу φ, то множество $\mathscr{T}'$ не было правильно построено по $\mathscr{T}$.

Пусть $\mathscr{T}'$ не φ-недостижимо. Тогда должно найтись такое наименьшее число i, что $C \vdash^i C$, где C_0 — начальная конфигурация и в конфигурации C алгоритм 7.5 запрашивает элемент φ множества $\mathscr{T}'$. Поскольку T_0 не изменяется на шаге (2а), должно быть $i > 0$. Пусть $C = (T_0X_1T_1 \ldots X_mT_m, w, \pi)$, где $T_m = \langle f, g\rangle$ и $\mathrm{FIRST}_k(w) = u$. Элемент φ может встретиться в трех случаях.

Случай 1: Предположим, что $f(u) = \varphi$. Тогда, согласно шагам (2б ii) и (2б iii) алгоритма 7.6, предыдущим тактом анализатора должна была быть свертка, а не перенос. Значит, $C_0 \vdash^{i-1} C' \vdash C$, где

$$C' = (T_0X_1T_1 \ldots X_{m-1}T_{m-1}Y_1U_1 \ldots Y_rU_r,\ x,\ \pi')$$

[1]) φ-элементы введены только для того, чтобы отметить элементы, которые можно менять.

и свертка выполнялась по правилу $X_m \to Y_1 \ldots Y_r$ (π представляет собой цепочку π', к которой приписан номер этого правила).

Рассмотрим множество ситуаций, по которому построена таблица U_r. (Если $r = 0$, вместо U_r читай T_{m-1}.) Это множество должно включать ситуацию $[X_m \to Y_1 \ldots Y_r \cdot, u]$. В определении допустимой ситуации требовалось, чтобы существовала такая цепочка $y \in \Sigma^*$, что $X_1 \ldots X_m uy$ — правовыводимая цепочка. Предположим, что нетерминал X_m появляется в результате применения правила $A \to \alpha X_m \beta$. Иными словами, в пополненной грамматике есть вывод

$$S' \Rightarrow_r^* \gamma A x \Rightarrow_r \gamma\alpha X_m \beta x \Rightarrow_r^* \gamma\alpha X_m uy$$

где $\gamma\alpha = X_1 \ldots X_{m-1}$. Так как $u \in \mathrm{FIRST}(\beta x)$, ситуация $[A \to \alpha X_m \cdot \beta, v]$ должна быть допустима для $X_1 \ldots X_m$, если $v = \mathrm{FIRST}(x)$.

Можно заключить, что в каноническом множестве таблиц действие таблицы T_m на цепочке u не „ошибка", и, значит, вопреки предположению в алгоритме 7.6 оно не могло оказаться замененным на φ.

Случай 2: Предположим, что $f(u) =$ **перенос**, a — первый символ цепочки u, $g(a) = \varphi$. Поскольку $f(u) =$ **перенос**, в множестве ситуаций, связанном с таблицей T_m, найдется такая ситуация вида $[A \to \alpha \cdot a\beta, v]$, что $u \in \mathrm{EFF}(a\beta v)$ и $[A \to \alpha \cdot a\beta, v]$ — допустимая ситуация для активного префикса $X_1 \ldots X_m$ (см. упр. 7.3.8). Но тогда ситуация $[A \to \alpha a \cdot \beta, v]$ допустима для $X_1 \ldots X_m a$ и $X_1 \ldots X_m a$ — также активный префикс. Следовательно, множество допустимых ситуаций для $X_1 \ldots X_m a$ непусто, и $g(a)$ не может быть элементом φ вопреки предположению.

Случай 3: Предположим, что $f(u) =$ **свертка** p, где правило с номером p есть $A \to X_r \ldots X_m$, $T_{r-1} = \langle f', g'\rangle$ и $g'(A) = \varphi$. Тогда ситуация $[A \to X_r \ldots X_m \cdot, u]$ допустима для $X_1 \ldots X_m$, а ситуация $[A \to \cdot X_r \ldots X_m, u]$ допустима для $X_1 \ldots X_{r-1}$. Как и в случае 1, приходим к выводу, что найдется ситуация $[B \to \alpha A \cdot \beta, v]$, допустимая для $X_1 \ldots X_{r-1}A$, и, значит, $g'(A)$ не может быть элементом φ. □

Можно также показать, что алгоритм 7.6 заменяет на φ максимально возможное число элементов **ошибка** в каноническом множестве таблиц. Поэтому, если произвольный элемент **ошибка** из $\mathscr{T}'$ заменить на φ, полученное множество таблиц не будет более φ-недостижимым.

7.3.4. Слияние таблиц с помощью совместимых разбиений

В этом разделе мы опишем важный метод, применяемый для уменьшения множества LR(k)-таблиц. Этот метод состоит в том, что две таблицы сливаются в одну, когда это не отражается на

поведении LR(k)-алгоритма разбора, использующего рассматриваемое множество таблиц. Пусть дано φ-недостижимое множество таблиц и T_1, T_2—две таблицы из этого множества. Предположим, что всякий раз, когда элементы действия или перехода таблиц не совпадают, один из них есть φ. В этом случае будем говорить, что таблицы T_1 и T_2 *совместимы*, и их можно слить, рассматривая T_1 и T_2 как одну таблицу.

Далее, предположим, что T_1 и T_2 не совпадают только в одном элементе перехода, который в одной таблице представляет собой T_3, а в другой T_4. Если T_3 и T_4 совместимы, можно одновременно слить T_3 с T_4 и T_1 с T_2. Слияние таблиц можно еще выполнить в том случае, когда T_3 и T_4 несовместимы только из-за того, что несовпадающими элементами перехода служат T_1 и T_2.

Опишем такой алгоритм слияния, определив понятие совместимого разбиения множества таблиц. Затем покажем, что все члены каждого блока совместимого разбиения можно одновременно объединить в одну таблицу.

Определение. Пусть $(\mathscr{T}, T_0)$ — φ-недостижимое множество LR(k)-таблиц и $\Pi = \{\mathscr{S}_1, \ldots, \mathscr{S}_p\}$ — *разбиение* на $\mathscr{T}$. Другими словами, $\mathscr{S}_1 \cup \mathscr{S}_2 \cup \ldots \cup \mathscr{S}_p = \mathscr{T}$ и $\mathscr{S}_i \cap \mathscr{S}_j = \varnothing$ для всех $i \neq j$. Будем называть Π *совместимым разбиением ширины* p, если для всех блоков $\mathscr{S}_i$, $1 \leqslant i \leqslant p$, из принадлежности элементов $\langle f_1, g_1\rangle$ и $\langle f_2, g_2\rangle$ блоку $\mathscr{S}_i$ вытекает, что

(1) при $f_1(u) \neq f_2(u)$ хотя бы одно из значений $f_1(u)$ и $f_2(u)$ есть φ,

(2) при $g_1(X) \neq g_2(X)$ либо

(а) хотя бы одно из значений $g_1(X)$ и $g_2(X)$ есть φ, либо

(б) $g_1(X)$ и $g_2(X)$ принадлежат одному и тому же блоку разбиения Π.

Совместимые разбиения множества LR(k)-таблиц можно найти методами, сходными с методами нахождения неразличимых состояний не полностью определенного конечного автомата. Наша цель состоит в нахождении совместимых разбиений наименьшей ширины. Изложим алгоритм, показывающий, как с помощью совместимых разбиений ширины p на множестве LR(k)-таблиц найти эквивалентное множество, содержащее p таблиц.

Алгоритм 7.7. Слияние с помощью совместимых разбиений.

Вход. φ-недостижимое множество $(\mathscr{T}, T_0)$ LR(k)-таблиц и совместимое разбиение $\Pi = \{\mathscr{S}_1, \ldots, \mathscr{S}_p\}$ на $\mathscr{T}$.

Выход. Эквивалентное φ-недостижимое множество $(\mathscr{T}', T_0')$ LR(k)-таблиц, для которого $\# \mathscr{T} = p$.

Метод.

(1) Для каждого i, $1 \leqslant i \leqslant p$, по блоку разбиения Π построить таблицу $U_i = \langle f, g\rangle$ следующим образом:

(а) Если $\langle f', g'\rangle \in \mathscr{S}_i$ и $f'(u) \neq \varphi$ для аванцепочки u, положить $f(u) = f'(u)$. Если такой таблицы в $\mathscr{S}_i$ нет, положить $f(u) = \varphi$.

(б) Если $\langle f', g'\rangle \in \mathscr{S}_i$ и $g'(X) \in \mathscr{S}_j$, положить $g(X) = U_j$. Если в $\mathscr{S}_i$ нет такой таблицы, что $g'(X) \in \mathscr{S}_j$, положить $g(X) = \varphi$.

(2) T_0' — таблица, построенная по блоку, содержащему T_0. □

Корректность построения, выполняемого алгоритмом 7.7, следует из определения совместимого разбиения.

Пример 7.19. Рассмотрим нашу обычную грамматику арифметических выражений G_0:

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to a$

φ-недостижимое множество LR(1)-таблиц для G_0 показано на рис. 7.25.

Легко видеть, что как таблицы T_3 и T_{10}, так и таблицы T_{14} и T_{20} совместимы. Поэтому можно построить совместимое разбиение, где блоками будут $\{T_3, T_{10}\}$, $\{T_{14}, T_{20}\}$ и все остальные таблицы. После замены $\{T_3, T_{10}\}$ на U_1 и $\{T_{14}, T_{20}\}$ на U_2 множество таблиц станет таким, как оно представлено на рис. 7.26. Заметим, что элементы перехода T_3, T_{10}, T_{14} и T_{20} соответственно изменены на U_1 или U_2.

Совместимое разбиение, приведенное выше, наилучшее из всех, которые можно найти. Например, T_{16} и T_{17} почти совместимы, и можно было бы сгруппировать их в один блок разбиения, если бы удалось сгруппировать в один блок этого же разбиения таблицы T_{10} и T_{20}. Но T_{10} и T_{20} не согласуются в действиях для $+$, $*$ и). □

Докажем, что алгоритм 7.7 порождает на выходе эквивалентное φ-недостижимое множество LR(k)-таблиц.

Теорема 7.6. *Пусть $\mathscr{T}'$ — множество LR(k)-таблиц, построенное по $\mathscr{T}$ алгоритмом 7.7. Тогда $\mathscr{T}$ и $\mathscr{T}'$ эквивалентны и оба φ-недостижимы.*

Доказательство. Пусть таблица T' из $\mathscr{T}'$ построена по блоку совместимого разбиения, содержащему таблицу T из $\mathscr{T}$.

	Действие						Переход							
	a	+	*	(	)	e	E	T	F	a	+	*	(	)
T_0	S	X	X	S	X	X	T_1	T_2	T_3	T_4	φ	φ	T_5	φ
T_1	φ	S	φ	φ	X	A	φ	φ	φ	φ	T_6	φ	φ	φ
T_2	φ	2	S	φ	2	2	φ	φ	φ	φ	φ	T_7	φ	φ
T_3	φ	4	4	φ	4	4	φ	φ	φ	φ	φ	φ	φ	φ
T_4	X	6	6	X	6	6	φ	φ	φ	φ	φ	φ	φ	φ
T_5	S	X	X	S	X	X	T_8	T_9	T_{10}	T_{11}	φ	φ	T_{12}	φ
T_6	S	X	X	S	X	X	φ	T_{13}	T_3	T_4	φ	φ	T_5	φ
T_7	S	X	X	S	X	X	φ	φ	T_{14}	T_4	φ	φ	T_5	φ
T_8	φ	S	φ	φ	S	X	φ	φ	φ	φ	T_{16}	φ	φ	T_{15}
T_9	φ	2	S	φ	2	2	φ	φ	φ	φ	φ	T_{17}	φ	φ
T_{10}	φ	4	4	φ	4	4	φ	φ	φ	φ	φ	φ	φ	φ
T_{11}	X	6	6	X	6	6	φ	φ	φ	φ	φ	φ	φ	φ
T_{12}	S	X	X	S	X	X	T_{18}	T_9	T_{10}	T_{11}	φ	φ	T_{12}	φ
T_{13}	φ	1	S	φ	1	1	φ	φ	φ	φ	φ	T_7	φ	φ
T_{14}	φ	3	3	φ	3	3	φ	φ	φ	φ	φ	φ	φ	φ
T_{15}	X	5	5	X	5	5	φ	φ	φ	φ	φ	φ	φ	φ
T_{16}	S	X	X	S	X	X	φ	T_{19}	T_{10}	T_{11}	φ	φ	T_{12}	φ
T_{17}	S	X	X	S	X	X	φ	φ	T_{20}	T_{11}	φ	φ	T_{12}	φ
T_{18}	φ	S	φ	φ	S	X	φ	φ	φ	φ	T_{16}	φ	φ	T_{21}
T_{19}	φ	1	S	φ	1	1	φ	φ	φ	φ	φ	T_{17}	φ	φ
T_{20}	φ	3	3	φ	3	3	φ	φ	φ	φ	φ	φ	φ	φ
T_{21}	X	5	5	X	5	5	φ	φ	φ	φ	φ	φ	φ	φ

Рис. 7.25. φ-недостижимое множество таблиц для G_0.

	Действие						Переход							
	a	+	*	(	)	e	E	T	F	a	+	*	(	)
T_0	S	X	X	S	X	X	T_1	T_2	U_1	T_4	φ	φ	T_5	φ
T_1	φ	S	φ	φ	φ	A	φ	φ	φ	φ	T_6	φ	φ	φ
T_2	φ	2	S	φ	φ	2	φ	φ	φ	φ	φ	T_7	φ	φ
T_4	X	6	6	X	X	6	φ	φ	φ	φ	φ	φ	φ	φ
T_5	S	X	X	S	X	X	T_8	T_9	U_1	T_{11}	φ	φ	T_{12}	φ
T_6	S	X	X	S	X	X	φ	T_{13}	U_1	T_4	φ	φ	T_5	φ
T_7	S	X	X	S	X	X	φ	φ	U_2	T_4	φ	φ	T_5	φ
T_8	φ	S	φ	φ	S	φ	φ	φ	φ	φ	T_{16}	φ	φ	T_{15}
T_9	φ	2	S	φ	2	φ	φ	φ	φ	φ	φ	T_{17}	φ	φ
T_{11}	X	6	6	X	6	X	φ	φ	φ	φ	φ	φ	φ	φ
T_{12}	S	X	X	S	X	X	T_{18}	T_9	U_1	T_{11}	φ	φ	T_{12}	φ
T_{13}	φ	1	S	φ	φ	1	φ	φ	φ	φ	φ	T_7	φ	φ
T_{15}	X	5	5	X	X	5	φ	φ	φ	φ	φ	φ	φ	φ
T_{16}	S	X	X	S	X	X	φ	T_{19}	U_1	T_{11}	φ	φ	T_{12}	φ
T_{17}	S	X	X	S	X	X	φ	φ	U_2	T_{11}	φ	φ	T_{12}	φ
T_{18}	φ	S	φ	φ	S	φ	φ	φ	φ	φ	T_{16}	φ	φ	T_{21}
T_{19}	φ	1	S	φ	1	φ	φ	φ	φ	φ	φ	T_{17}	φ	φ
T_{21}	X	5	5	X	5	X	φ	φ	φ	φ	φ	φ	φ	φ
U_1	φ	4	4	φ	4	4	φ	φ	φ	φ	φ	φ	φ	φ
U_2	φ	3	3	φ	3	3	φ	φ	φ	φ	φ	φ	φ	φ

Рис. 7.26. Совмещенное множество таблиц.

Пусть $C_0=(T_0, w, e)$ и $C_0'=(T_0', w, e)$ — начальные конфигурации LR(k)-анализатора, использующего $\mathscr{T}$ и $\mathscr{T}'$ соответственно. Покажем, что

(7.3.1) $C_0 \vdash^i (T_0X_1T_1 \ldots X_mT_m, x, \pi)$ для анализатора, использующего $\mathscr{T}$, тогда и только тогда, когда
$C_0' \vdash^i (T_0'X_1T_1' \ldots X_mT_m', x, \pi)$ для анализатора, использующего $\mathscr{T}'$.

Это означает, что единственное различие между случаями, когда LR(k)-анализатор использует $\mathscr{T}$ и $\mathscr{T}'$, состоит в том, что, используя $\mathscr{T}'$, анализатор заменяет таблицу T из $\mathscr{T}$ представителем содержащего ее блока разбиения П.

Докажем утверждение (7.3.1) индукцией по i. Начнем с необходимости условия. Базис, $i=0$, тривиален. Для проведения шага индукции предположим, что утверждение (7.3.1) верно для i. Докажем, что оно верно для $i+1$. Так как множество $\mathscr{T}$ φ-недостижимо, действия таблиц T_m и T_m' на $\mathrm{FIRST}_k(x)$ совпадают. Пусть это общее действие — **перенос**, a — первый символ цепочки x и элемент перехода таблицы T_m на a есть T. В соответствии с алгоритмом 7.7 и определением совместимого разбиения переходом таблицы T_m' на a будет T' тогда и только тогда, когда T' — представитель содержащего T блока разбиения П.

Если действие — **свертка** по некоторому правилу с r символами в правой части, то, сравнивая таблицы T_{m-r} и T_{m-r}', убеждаемся, что утверждение верно для $i+1$.

Для доказательства достаточности нужно только заметить, что, если в T_m' у функции действия для цепочки $\mathrm{FIRST}(x)$ стоит элемент, отличный от φ, таблица T_m' должна совпадать с T_m, поскольку множество $\mathscr{T}$ φ-недостижимо. □

Итак, алгоритм 7.7 сохраняет эквивалентность в самом сильном смысле. Анализатор, использующий два таких множества таблиц, всегда вычисляет очередные конфигурации за одно и то же число шагов независимо от того, имеет ли входная цепочка разбор.

7.3.5. Отсрочка в обнаружении ошибок

Наш основной метод уменьшения множества LR(k)-таблиц должен сливать таблицы всегда, когда это возможно. Однако две таблицы можно слить в одну только тогда, когда они совместимы. В данном разделе мы обсудим метод, применяемый для замены в некоторых таблицах существенных элементов **ошибка** элементами **свертка** с тем, чтобы увеличить число совместимых пар таблиц в множестве LR(k)-таблиц.

В качестве примера рассмотрим таблицы T_4 и T_{11}, приведенные на рис. 7.25. Для удобства представим их функции действия отдельно в виде рис. 7.25(а). Если изменить действие таблицы T_4 на аванцепочке) с **ошибки** на **свертку** 6 и действие таблицы T_{11} на аванцепочке e с **ошибки** на **свертку** 6, то T_4 и T_{11} станут совместимыми и можно будет слить их в одну таблицу. Однако, прежде чем выполнять такие замены, хотелось бы удостовериться, что ошибки, обнаруживаемые таблицей T_4 на) и таблицей T_{11} на e, будут обнаружены на одном из последующих шагов перед операцией переноса. В этом разделе мы получим

Действие

	a	+	*	(	)	e
T_4:	X	6	6	X	X	6
T_{11}:	X	6	6	X	6	X

Рис. 7.25(а)

условия, при которых можно отсрочить обнаружение ошибки, не изменив позиции во входной цепочке, в которой LR(k)-анализатор сообщает об ошибке. В частности, легко показать, что в каноническом множестве таблиц любая такая замена допустима.

Предположим, что LR(k)-анализатор находится в конфигурации $(T_0X_1T_1 \ldots X_mT_m, w, \pi)$ и действием таблицы T_m на аванцепочке $u=\mathrm{FIRST}_k(w)$ служит **ошибка**. Предположим далее, что этот элемент **ошибка** заменяется в T_m на **свертку** p, где p — номер правила $A \to Y_1 \ldots Y_r$. Эту ошибку впоследствии можно обнаружить двумя способами.

При свертке из магазина удаляются $2r$ символов и управление передается таблице T_{m-r}. Если у функции переходов таблицы T_{m-r} символу A отвечает элемент φ, то можно сообщить об ошибке, заменив φ элементом **ошибка**. С другой стороны, символу A у функции переходов таблицы T_{m-r} может соответствовать некоторая таблица T. Если действием таблицы T на аванцепочке u служит **ошибка** или элемент φ (который можно заменить на **ошибку**, чтобы сохранить свойство φ-недостижимости), то можно обнаружить ошибку и на этой стадии. Удастся обнаружить ошибку и в том случае, когда действием таблицы T на аванцепочке u будет **свертка** p' и описанный выше процесс повторится. Короче говоря, мы хотим, чтобы таблицы, которые становятся управляющими после свертки по правилу p, не вызывали на аванцепочке u переноса (или допуска).

Заметим, что для того, чтобы заменить **ошибку** на **свертку**, нам приходится во всей широте использовать определение эквивалентности множеств LR(k)-таблиц. При разборе с новым множеством таблиц последующие конфигурации могут вычисляться

на несколько шагов позже, чем при разборе с помощью старого множества, но входные символы при этом считываться не будут.

Чтобы описать условия, при которых допустима такая замена, нужно для любой таблицы, появляющейся при разборе в верхушке магазина, знать, какие таблицы могут встретиться в магазине в качестве $(r+1)$-й таблицы сверху. Определим сначала три функции на таблицах и цепочках символов грамматики.

Определение. Пусть $(\mathscr{T}, T_0)$ — множество LR(k)-таблиц для грамматики $G=(N, \Sigma, P, S)$. Расширим область определения функции GOTO из разд. 5.2.3. Функция GOTO отображает $\mathscr{T}\times(N\cup\Sigma)^*$ в $\mathscr{T}$ следующим образом:

(1) $\mathrm{GOTO}(T, e)=T$ для всех T из $\mathscr{T}$.

(2) Если $T=\langle f, g\rangle$, то $\mathrm{GOTO}(T, X)=g(X)$ для всех X из $N\cup\Sigma$ и T из $\mathscr{T}$.

(3) $\mathrm{GOTO}(T, \alpha X)=\mathrm{GOTO}(\mathrm{GOTO}(T, \alpha), X)$ для всех α из $(N\cup\Sigma)^*$ и T из $\mathscr{T}$.

Весом таблицы T из $(\mathscr{T}, T_0)$ будем называть такое наибольшее число r, что если $\mathrm{GOTO}(T_0, \alpha)=T$, то $|\alpha|\geqslant r$.

Нам придется также пользоваться функцией GOTO^{-1}, „обратной" к GOTO. Она отображает множество $\mathscr{T}\times(N\cup\Sigma)^*$ в множество всех подмножеств множества $\mathscr{T}$. Положим $\mathrm{GOTO}^{-1}(T, \alpha)=\{T' \mid \mathrm{GOTO}(T', \alpha)=T\}$.

Наконец, определим функцию $\mathrm{NEXT}(T, p)$, где $T\in\mathscr{T}$, а p — номер правила $A\to X_1\ldots X_r$:

(1) Если вес таблицы T меньше r, то $\mathrm{NEXT}(T, p)$ не определено.

(2) Если вес таблицы T не меньше r, то $\mathrm{NEXT}(T, p)=\{T' \mid$ существуют такие $T''\in\mathscr{T}$ и $\alpha\in(N\cup\Sigma)^r$, что $T''\in\mathrm{GOTO}^{-1}(T, \alpha)$ и $T'=\mathrm{GOTO}(T'', A)\}$.

Таким образом, $\mathrm{NEXT}(T, p)$ дает все таблицы, которые могут стать управляющими после T, если T находится в верхушке магазина и вызывает свертку по правилу p. Заметим, что не требуется, чтобы верхними r символами магазина были $X_1\ldots X_r$. Единственное требование: чтобы в магазине было не меньше r символов. Если таблицы канонические, можно показать, что среди всех цепочек длины r множество $\mathrm{GOTO}^{-1}(T, \alpha)$ будет непустым только для $\alpha=X_1\ldots X_r$.

В качестве упражнений (см. конец разд. 7.3) предлагаем установить некоторые алгебраические свойства функций GOTO и NEXT.

Функция GOTO для множества LR(k)-таблиц хорошо описывается с помощью помеченного графа. Вершины графа GOTO помечены именами таблиц, и если $\mathrm{GOTO}(T_i, X)=T_j$, то из вершины, помеченной T_i, в вершину, помеченную T_j, проведена дуга,

помеченная X. Следовательно, если $\mathrm{GOTO}(T, X_1X_2\ldots X_r)=T'$, то существует такой путь из вершины T в вершину T', что метки дуг, составляющих этот путь, образуют цепочку $X_1X_2\ldots X_r$. Вес таблицы T можно интерпретировать как длину кратчайшего пути из T_0 в T в графе GOTO.

По графу GOTO легко вычисляется функция NEXT. Для того чтобы найти $\mathrm{NEXT}(T, i)$, где правило с номером i есть $A\to X_1X_2\ldots X_r$, найдем в графе GOTO все такие вершины T'', что из T'' в T проходит путь длины r. Затем для каждой такой вершины T'' включим $\mathrm{GOTO}(T'', A)$ в $\mathrm{NEXT}(T, i)$.

Пример 7.20. Граф GOTO для множества таблиц рис. 7.25 приведен на рис. 7.27.

Из этого графа получаем, что $\mathrm{GOTO}(T_6, (E))=T_{15}$, так как $\mathrm{GOTO}(T_6, (\ )=T_5$, $\mathrm{GOTO}(T_5, E)=T_8$ и $\mathrm{GOTO}(T_8,))=T_{15}$.

Таблица T_6 имеет вес 2, поэтому $\mathrm{NEXT}(T_6, 5)$, где правило 5 есть $F\to(E)$, не определено.

Вычислим теперь $\mathrm{NEXT}(T_{15}, 5)$. Путь длины 3 в T_{15} выходит только из трех таблиц, а именно из T_0, T_6 и T_7. Поэтому $\mathrm{GOTO}(T_0, F)=T_3$, $\mathrm{GOTO}(T_6, F)=T_3$ и $\mathrm{GOTO}(T_7, F)=T_{14}$, так что $\mathrm{NEXT}(T_{15}, 5)=\{T_3, T_{14}\}$. □

Дадим теперь алгоритм, преобразующий φ-недостижимое множество таблиц так, чтобы некоторые ошибки обнаруживались позже во времени, но на том же месте входной цепочки. Наш алгоритм не будет самым общим, но из него будет ясно, как осуществляются и более общие преобразования.

Мы будем заменять некоторые элементы **ошибка** и элементы φ у функций действия на элементы **свертка**. Для каждого изменяемого элемента мы точно определяем правило, которое нужно применить при новой свертке. Допустимые замены образуют так называемое множество отсрочки. Каждый элемент множества отсрочки представляет собой тройку (T, u, i), где T — имя таблицы, u — аванцепочка и i — номер правила. Элемент (T, u, i) указывает, что нужно изменить действие таблицы T на аванцепочке u на **свертку** i.

Определение. Пусть $(\mathscr{T}, T_0)$ — множество LR(k)-таблиц для грамматики $G=(N, \Sigma, P, S)$. Назовем подмножество $\mathscr{P}$ множества $\mathscr{T}\times\Sigma^{*k}\times P$ *множеством отсрочки для* $(\mathscr{T}, T_0)$, если удовлетворяются следующие условия:

Если $(T, u, i)\in\mathscr{P}$ и $T=\langle f, g\rangle$, то

(1) $f(u)=$ **ошибка** или φ;

(2) если правило i есть $A\to\alpha$ и $T=\mathrm{GOTO}(T_0, \beta)$, то α — суффикс цепочки β;

(3) нет такого правила i', что (T, u, i') также принадлежит $\mathscr{P}$;

(4) если $T'\in\mathrm{NEXT}(T, i)$ и $T'=\langle f', g'\rangle$, то $f'(u)=$ **ошибка** или φ.

Условие (1) утверждает, что на **свертку** заменяются только элементы **ошибка** и φ. Условие (2) гарантирует, что свертка по правилу i возможна только тогда, когда в верхушке магазина

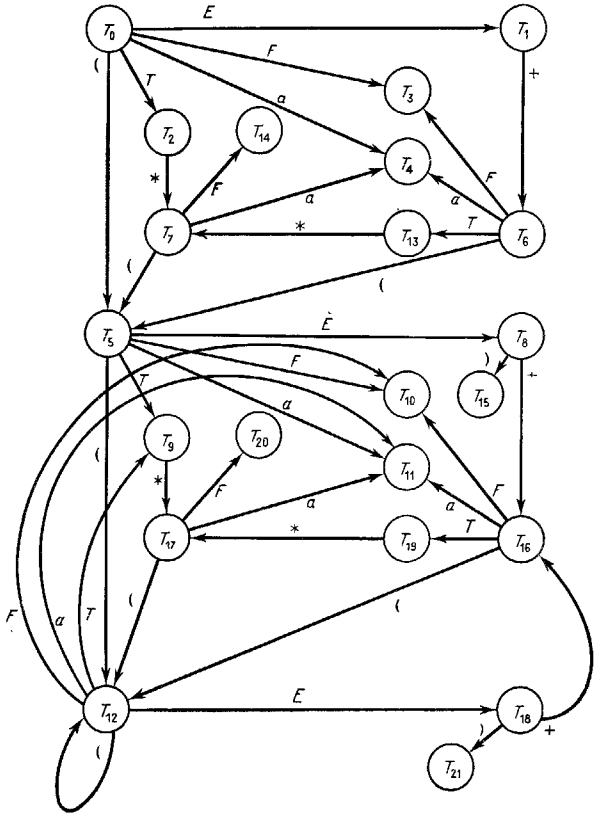


Рис. 7.27. Граф GOTO.

находится цепочка α. Условие (3) обеспечивает однозначность, а (4) означает, что свертки, вызванные появлением дополнительных **сверток**, в конечном итоге будут проделаны без **переносов**.

По поводу условия (4) заметим, что (T', u, j) также может входить в $\mathcal{P}$. В этом случае значение $f'(u)$ также будет изменено с **ошибки** или φ на **свертку** j. Поэтому, прежде чем будет выдано сообщение об ошибке, возможно, придется выполнить последовательно несколько сверток.

Нахождение множества отсрочки, позволяющего максимизировать общее число совместимых таблиц в множестве LR(k)-таблиц, составляет большую комбинаторную задачу. В одном из приведенных ниже примеров мы коснемся некоторых эвристических методов нахождения соответствующих множеств отсрочки. Но сначала покажем, как используется множество отсрочки при преобразовании данного множества LR(k)-таблиц.

Алгоритм 7.8. Отсрочка в обнаружении ошибок.

Вход. LR(k)-грамматика $G=(\mathrm{N}, \Sigma, P, S)$, φ-недостижимое множество $(\mathcal{T}, T_0)$ LR(k)-таблиц для G и множество отсрочки $\mathcal{P}$.

Выход. φ-недостижимое множество $\mathcal{T}'$ LR(k)-таблиц, эквивалентное $\mathcal{T}$.

Метод.

(1) Для каждого элемента (T, u, i) из $\mathcal{P}$, где $T=\langle f, g\rangle$, заменить $f(u)$ на **свертку** i.

(2) Предположим, что $(T, u, i) \in \mathcal{P}$ и правило i есть $A \to \alpha$. Для всех $T'=\langle f', g'\rangle$, для которых $\mathrm{GOTO}(T', \alpha)=T$ и $g'(A)=\varphi$, заменить $g'(A)$ на **ошибку**.

(3) Предположим, что (T, u, i) принадлежит $\mathcal{P}$ и $T'=\langle f', g'\rangle$ принадлежит $\mathrm{NEXT}(T, i)$. Если $f'(u)=\varphi$, заменить $f'(u)$ на **ошибку**.

(4) Полученное множество таблиц, имена которых сохранены, представляет собой $\mathcal{T}'$. □

Пример 7.21. Рассмотрим грамматику G с правилами

(1) $S \to AS$
(2) $S \to b$
(3) $A \to aB$
(4) $B \to aB$
(5) $B \to b$

φ-недостижимое множество LR(1)-таблиц для G, построенное в результате применения алгоритма 7.6 к каноническому множеству LR(1)-таблиц для G, приведено на рис. 7.28.

Можно, например, заменить оба элемента **ошибка** у функций действия таблицы T_4 на **свертку** 5 и элемент **ошибка** в таблице T_8 на **свертку** 2. Другими словами, мы выбираем множество отсрочки $\mathcal{P}=\{(T_4, a, 5), (T_4, b, 5), (T_8, e, 2)\}$. Правило 5 есть $B \to b$ и $\mathrm{GOTO}(T_0, b)=\mathrm{GOTO}(T_2, b)=T_4$. Таким образом, эле-

	Действие			Переход				
	a	b	e	S	A	B	a	b
T_0	S	S	X	T_1	T_2	φ	T_3	T_4
T_1	φ	φ	A	φ	φ	φ	φ	φ
T_2	S	S	φ	T_5	T_2	φ	T_3	T_4
T_3	S	S	X	φ	φ	T_6	T_7	T_8
T_4	X	X	2	φ	φ	φ	φ	φ
T_5	φ	φ	1	φ	φ	φ	φ	φ
T_6	3	3	φ	φ	φ	φ	φ	φ
T_7	S	S	X	φ	φ	T_9	T_7	T_8
T_8	5	5	X	φ	φ	φ	φ	φ
T_9	4	4	φ	φ	φ	φ	φ	φ

Рис. 7.28. φ-недостижимое множество таблиц для G_0.

	Действие			Переход				
	a	b	e	S	A	B	a	b
T_0	S	S	X	T_1	T_2	X	T_3	T_4
T_1	φ	φ	A	φ	φ	φ	φ	φ
T_2	S	S	φ	T_5	T_2	X	T_3	T_4
T_3	S	S	X	X	φ	T_6	T_7	T_8
T_4	5	5	2	φ	φ	φ	φ	φ
T_5	φ	φ	1	φ	φ	φ	φ	φ
T_6	3	3	φ	φ	φ	φ	φ	φ
T_7	S	S	X	X	φ	T_9	T_7	T_8
T_8	5	5	2	φ	φ	φ	φ	φ
T_9	4	4	φ	φ	φ	φ	φ	φ

Рис. 7.29. Множество таблиц после применения алгоритма отсрочки в обнаружении ошибок.

менты, расположенные в T_0 и T_2 под B, нужно изменить с φ на **ошибку**. Аналогично элементы, расположенные в T_3 и T_7 под S, заменяются на **ошибку**. Поскольку NEXT(T_4, 5) и NEXT(T_8, 2) пусты, никакие элементы φ у функций действия заменить на **ошибку** нельзя. Полученное множество таблиц показано на рис. 7.29.

Можно, если угодно, воспользоваться алгоритмом 7.7, взяв в качестве совместимого разбиения разбиение, в котором таблица T_4 сгруппирована с T_8, T_1 с T_2 и T_5 с T_6. (Возможны также три другие попарные комбинации.) Новое множество таблиц приведено на рис. 7.30. □

	Действие			Переход				
	a	b	e	S	A	B	a	b
T_0	S	S	X	T_1	T_1	X	T_3	T_4
T_1	S	S	A	T_5	T_1	X	T_3	T_4
T_3	S	S	X	X	φ	T_5	T_7	T_4
T_4	5	5	2	φ	φ	φ	φ	φ
T_5	3	3	1	φ	φ	φ	φ	φ
T_7	S	S	X	X	φ	T_9	T_7	T_4
T_9	4	4	φ	φ	φ	φ	φ	φ

Рис. 7.30. Совмещенное множество таблиц.

Теорема 7.7. *Алгоритм 7.8 порождает φ-недостижимое множество таблиц $\mathscr{T}'$, эквивалентное множеству $\mathscr{T}$.*

Доказательство. Пусть $C_0=(T_0, w, e)$ — начальная конфигурация LR(k)-анализатора (использующего либо $\mathscr{T}$, либо $\mathscr{T}'$ — имена таблиц совпадают). Пусть $C_0 \vdash C_1 \vdash \ldots \vdash C_n$ — вся последовательность тактов, выполненных анализатором, работающим с $\mathscr{T}$, и $C_0 \vdash C_1' \vdash \ldots \vdash C_m'$ — соответствующая последовательность для $\mathscr{T}'$. Так как $\mathscr{T}'$ образуется из $\mathscr{T}$ в результате замены только элементов **ошибка** и φ, должно быть $m \geqslant n$ и $C_i'=C_i$ для $1 \leqslant i \leqslant n$. (Это означает, что хотя имеются в виду разные таблицы, имена их совпадают.)

Покажем, что либо $m=n$, либо, если $m>n$, после того как достигается конфигурация C_n', не выполняются переносы и не будет допуска.

Если $m=n$, то утверждение теоремы очевидно, и если C_n — допускающая конфигурация, то $m=n$. Поэтому предположим,

что C_n сигнализирует об ошибке и $m > n$. Согласно определению множества отсрочки, поскольку в конфигурации C_n действием является **ошибка**, а в конфигурации C'_n нет, действием в C'_n должна быть свертка. Поэтому пусть s — наименьшее целое, бо́льшее n и такое, что действие в конфигурации C'_s есть **перенос** или **допуск**. (Если такого s не существует, то теорема доказана.)

Итак, найдется такое наименьшее r, $n < r \leqslant s$, что элемент у функции действия, к которому происходит обращение в конфигурации C'_r, присутствует в одной из таблиц множества $\mathscr{T}$. Случай $r = s$ не исключается, поскольку, безусловно, в $\mathscr{T}$ был **перенос** или **допуск** конфигурации C'_r. Элемент действия, запрашиваемый в конфигурации C'_{r-1}, имел вид **свертка** i для некоторого i. Согласно нашему определению r, этот элемент должен был появиться в результате применения алгоритма 7.8.

Пусть T_1 и T_2 — управляющие таблицы в конфигурациях C'_{r-1} и C'_r соответственно. Тогда $T_2 \in \mathrm{NEXT}(T_1, i)$, что противоречит условию (4) определения множества отсрочки. □

Приведем теперь достаточно полный пример, который демонстрирует, как находить множества отсрочки и совместимые разбиения. Здесь во многом используются эвристические методы. Поскольку эти методы нигде далее не описываются, мы настоятельно рекомендуем читателю внимательно разобраться в данном примере.

Пример 7.22. Рассмотрим множество таблиц для G_0, показанное на рис. 7.25. Наш подход заключается в том, чтобы для увеличения числа таблиц с аналогичными функциями действия применить алгоритм 7.8, заменяющий действия **ошибка** действиями **свертка**. В частности, попытаемся слить в одну все таблицы, вызывающие одинаковые свертки.

Попробуем слить таблицы T_{15} и T_{21}, поскольку они обе свертывают по правилу 5, и таблицы T_4 и T_{11}, свертывающие по правилу 6.

Для того чтобы слить таблицы T_{15}, T_{21}, нужно сделать так, чтобы действием таблицы T_{15} на) и таблицы T_{21} на e стала **свертка** 5. Затем нужно проверить, какие действия отвечают символам) и e в таблицах из множеств $\mathrm{NEXT}(T_{15}, 5) = \{T_3, T_{14}\}$ и $\mathrm{NEXT}(T_{21}, 5) = \{T_{10}, T_{20}\}$. Поскольку T_3 и T_{14} на) обе имеют действием φ, нужно заменить эти φ на **ошибку** и на этом закончить преобразование. Правда, тогда T_3 и T_{10} не будут более совместимыми, как и таблицы T_{14} и T_{20}; поэтому мы заменим действия таблиц T_3 и T_{14} на символе) на **свертку** 4 и **свертку** 3 соответственно.

Теперь проанализируем множества таблиц $\mathrm{NEXT}(T_3, 4) = \mathrm{NEXT}(T_{14}, 3) = \{T_2, T_{13}\}$. Аналогичные рассуждения приводят нас к выводу, что действия таблиц T_2 и T_{13} на) нужно заменить не на **ошибку**, а на **свертку** 2 и **свертку** 1 соответственно. Далее видим, что $\mathrm{NEXT}(T_2, 2) = \{T_1\} = \mathrm{NEXT}(T_{13}, 1)$. Нужно заменить действие таблицы T_1 на символе) на **ошибку**, и тогда будут учтены все изменения, необходимые для замены действия T_{15} на символе) на **свертку** 5.

Посмотрим, что произойдет, если заменить действие таблицы T_{21} на символе e на **свертку** 5. $\mathrm{NEXT}(T_{21}, 4) = \{T_{10}, T_{20}\}$, но заменять действия таблиц T_{10} и T_{20} на символе e на **ошибку** нежелательно, поскольку, возможно, не удастся слить эти таблицы с T_3 и T_{11}. Поэтому заменим действия таблиц T_{10} и T_{20} на символе e на **свертку** 4 и **свертку** 3 соответственно. Далее находим, что

$$\mathrm{NEXT}(T_{10}, 4) = \mathrm{NEXT}(T_{20}, 3) = \{T_9, T_{19}\}$$

Мы не будем заменять действия таблиц T_9 и T_{19} на символе e на **ошибку**, так что пусть действием таблицы T_9 на символе e будет **свертка** 2, а таблицы T_{19} — **свертка** 1. Затем находим, что

$$\mathrm{NEXT}(T_9, 2) = \mathrm{NEXT}(T_{19}, 1) = \{T_8, T_{18}\}$$

В этих таблицах заменим действия на символе e на **ошибку**.

Мы сделали таблицы T_{15} и T_{21} совместимыми, и это не повлияло на возможную совместимость таблиц T_3 и T_{10}, T_{14} и T_{20}, T_2 и T_9, T_{13} и T_{19}, T_{14} и T_{20}. Исследуем возможность сделать T_4 и T_{11} совместимыми, заменив действие таблицы T_4 на символе) и таблицы T_{11} на символе e на **свертку** 6. Поскольку $\mathrm{NEXT}(T_4, 6) = \{T_3, T_{14}\}$ и $\mathrm{NEXT}(T_{11}, 6) = \{T_{10}, T_{20}\}$, изменения, внесенные в T_3, T_{14} и T_{10}, T_{20}, без затруднений переносятся на таблицы T_4 и T_{11}. Все множество отсрочки состоит из элементов

$[T_2, \ ), 2]$	$[T_9, e, 2]$
$[T_3, \ ), 4]$	$[T_{10}, e, 4]$
$[T_4, \ ), 6]$	$[T_{11}, e, 6]$
$[T_{13}, \ ), 1]$	$[T_{19}, e, 1]$
$[T_{14}, \ ), 3]$	$[T_{20}, e, 3]$
$[T_{15}, \ ), 5]$	$[T_{21}, e, 5]$

Результат применения алгоритма 7.8 к множеству таблиц рис. 7.25 с таким множеством отсрочки показан на рис. 7.31. Заметим, что к функциям переходов не добавлено ни одного элемента **ошибка**.

Из рис. 7.31 видно, что следующие пары таблиц совместимы:

$$\begin{array}{ll} T_3, & T_{10} \\ T_4, & T_{11} \\ T_{11}, & T_{20} \\ T_{15}, & T_{21} \end{array}$$

	Действие						Переход							
	a	+	*	(	)	e	E	T	F	a	+	*	(	)
T_0	S	X	X	S	X	X	T_1	T_2	T_3	T_4	φ	φ	T_5	φ
T_1	φ	S	φ	φ	φ	A	φ	φ	φ	φ	T_6	φ	φ	φ
T_2	φ	2	S	φ	φ	2	φ	φ	φ	φ	φ	T_7	φ	φ
T_3	φ	4	4	φ	φ	4	φ	φ	φ	φ	φ	φ	φ	φ
T_4	X	6	6	X	X	6	φ	φ	φ	φ	φ	φ	φ	φ
T_5	S	X	X	S	X	X	T_8	T_9	T_{10}	T_{11}	φ	φ	T_{12}	φ
T_6	S	X	X	S	X	X	φ	T_{13}	T_3	T_4	φ	φ	T_5	φ
T_7	S	X	X	S	X	X	φ	φ	T_{14}	T_4	φ	φ	T_5	φ
T_8	φ	S	φ	φ	S	φ	φ	φ	φ	φ	T_{16}	φ	φ	T_{15}
T_9	φ	2	S	φ	2	φ	φ	φ	φ	φ	φ	T_{17}	φ	φ
T_{10}	φ	4	4	φ	4	φ	φ	φ	φ	φ	φ	φ	φ	φ
T_{11}	X	6	6	X	6	X	φ	φ	φ	φ	φ	φ	φ	φ
T_{12}	S	X	X	S	X	X	T_{18}	T_9	T_{10}	T_{11}	φ	φ	T_{12}	φ
T_{13}	φ	1	S	φ	φ	1	φ	φ	φ	φ	φ	T_7	φ	φ
T_{14}	φ	3	3	φ	φ	3	φ	φ	φ	φ	φ	φ	φ	φ
T_{15}	X	5	5	X	X	5	φ	φ	φ	φ	φ	φ	φ	φ
T_{16}	S	X	X	S	X	X	φ	T_{19}	T_{10}	T_{11}	φ	φ	T_{12}	φ
T_{17}	S	X	X	S	X	X	φ	φ	T_{20}	T_{11}	φ	φ	T_{12}	φ
T_{18}	φ	S	φ	φ	S	φ	φ	φ	φ	φ	T_{16}	φ	φ	T_{21}
T_{19}	φ	1	S	φ	1	φ	φ	φ	φ	φ	φ	T_{17}	φ	φ
T_{20}	φ	3	3	φ	3	φ	φ	φ	φ	φ	φ	φ	φ	φ
T_{21}	X	5	5	X	5	X	φ	φ	φ	φ	φ	φ	φ	φ

Рис 7.31. Результат применения алгоритма отсрочки в обнаружении ошибок.

Более того, если эти пары образуют блоки совместимого разбиения, можно сгруппировать еще и такие пары:

$$T_8,\quad T_{18}$$
$$T_5,\quad T_{12}$$
$$T_7,\quad T_{17}$$
$$T_{13},\quad T_{19}$$
$$T_2,\quad T_9$$
$$T_6,\quad T_{16}$$

Если алгоритм 7.7 применить к разбиению, блоками которого служат приведенные выше пары и одноэлементные множества $\{T_0\}$ и $\{T_1\}$, то получится множество таблиц, показанное

	Действие						Переход							
	a	+	*	(	)	e	E	T	F	a	+	*	(	)
T_0	S	X	X	S	X	X	T_1	T_2	T_3	T_4	φ	φ	T_5	φ
T_1	φ	S	φ	φ	X	A	φ	φ	φ	φ	T_6	φ	φ	φ
T_2	φ	2	S	φ	2	2	φ	φ	φ	φ	φ	T_7	φ	φ
T_3	φ	4	4	φ	4	4	φ	φ	φ	φ	φ	φ	φ	φ
T_4	X	6	6	X	6	6	φ	φ	φ	φ	φ	φ	φ	φ
T_5	S	X	X	S	X	X	T_8	T_2	T_3	T_4	φ	φ	T_5	φ
T_6	S	X	X	S	X	X	φ	T_{13}	T_3	T_4	φ	φ	T_5	φ
T_7	S	X	X	S	X	X	φ	φ	T_{14}	T_4	φ	φ	T_5	φ
T_8	φ	S	φ	φ	S	X	φ	φ	φ	φ	T_6	φ	φ	T_{15}
T_{13}	φ	1	S	φ	1	1	φ	φ	φ	φ	φ	T_7	φ	φ
T_{14}	φ	3	3	φ	3	3	φ	φ	φ	φ	φ	φ	φ	φ
T_{15}	X	5	5	X	5	5	φ	φ	φ	φ	φ	φ	φ	φ

Рис. 7.32. Множество таблиц после применения алгоритма слияния.

на рис. 7.32. Представителем в каждом случае будет та из пары таблиц, которая имеет меньший индекс. Интересно отметить, что множество таблиц на рис. 7.32 получается непосредственно из G_0 с помощью метода „SLR“, излагаемого в разд. 7.4.1.

Для того чтобы проиллюстрировать эффект отсрочки в обнаружении ошибок, разберем ошибочную входную цепочку a). Используя множество таблиц рис. 7.25, канонический анализа-

тор выполнит только один такт

$$[T_0, a), e] \vdash [T_0 a T_4), e]$$

Действием таблицы T_4 на символе) будет **ошибка**.

С другой стороны, используя для разбора этой же входной цепочки множество таблиц рис. 7.32, анализатор выполнит последовательность тактов

$$\begin{aligned}[T_0, a), e] &\vdash [T_0 a T_4, \ ), e] \\ &\vdash [T_0 F T_3, \ ), 6] \\ &\vdash [T_0 T T_2, \ ), 64] \\ &\vdash [T_0 E T_1, \ ), 642]\end{aligned}$$

Здесь, прежде чем сообщить об ошибке, анализатор сделает еще три свертки. □

7.3.6. Исключение сверток по цепным правилам

Изучим теперь важное преобразование множества LR(k)-таблиц, не сохраняющее эквивалентности в смысле предыдущих разделов. Это преобразование не приведет к возникновению дополнительных операций переноса и не вызовет свертки, если исходное множество таблиц обнаружит ошибку. Напротив, в результате этого преобразования некоторые свертки будут пропускаться. Это приведет к тому, что таблица, появляющаяся в магазине, не всегда будет связана с записанной под ней цепочкой символов грамматики (хотя она всегда будет совместима с таблицей, связанной с этой цепочкой). Преобразование, рассматриваемое в этом разделе, должно осуществляться над цепными правилами; мы изучим его несколько менее формально, чем предыдущие преобразования.

Правило вида $A \to B$, где A и B — нетерминалы, называется *цепным*. Правила такого вида часто встречаются в грамматиках, описывающих языки программирования. Так, цепные правила часто возникают при использовании контекстно-свободной грамматики для описания приоритетов операций в языках программирования. Например, если цепочку $a_1 + a_2 * a_3$ следует трактовать как $a_1 + (a_2 * a_3)$, то говорят, что операция $*$ имеет *более высокий приоритет*, чем операция $+$.

Наша грамматика G_0, описывающая арифметические выражения, вводит для $*$ более высокий приоритет, чем для $+$. Грамматика G_0 состоит из правил

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$

(5) $F \to (E)$
(6) $F \to a$

Можно считать, что нетерминалы E, T и F порождают выражения различных приоритетов в соответствии с приоритетами операций. E порождает выражения, приоритет которых равен 1. Это цепочки символов T, разделенные знаками $+$. Операция $+$ имеет приоритет 1. T порождает выражения приоритета 2; они состоят из символов F, разделенных знаками $*$. Выражения приоритета 3 — это выражения, порожденные нетерминалом F, и их можно считать первичными выражениями.

Таким образом, при разборе цепочки $a_1 + a_2 * a_3$ в соответствии с грамматикой G_0 сначала нужно свернуть $a_2 * a_3$ в T, а затем свернуть это T вместе с a_1 в выражение E.

Единственная цель, которой служат два цепных правила $E \to T$ и $T \to F$, состоит в том, чтобы позволить тривиально свертывать выражения более высокого приоритета в выражения более низкого приоритета. В компиляторе правила перевода, связанные обычно с такими цепными правилами, означают просто, что переводы нетерминалов в левой и правой частях правила совпадают. При таком условии можно, если угодно, исключить свертки по цепным правилам.

Некоторые языки программирования имеют 12 и более приоритетов операций. Поэтому, если разбор ведется в соответствии с грамматикой, отражающей иерархию приоритетов, то анализатор часто будет выполнять последовательности сверток по цепным правилам. Если удастся исключить такие последовательности сверток, скорость разбора значительно возрастет. В большинстве практических случаев это удается сделать, не изменив получаемого перевода.

В данном разделе мы опишем преобразование на множестве LR(k)-таблиц, позволяющее исключить свертки по цепным правилам везде, где это надо.

Пусть $(\mathscr{T}, T_0)$ — φ-недостижимое множество LR(k)-таблиц для LR(k)-грамматики $G = (N, \Sigma, P, S)$. Предположим, что $\mathscr{T}$ содержит максимально возможное число элементов φ. Пусть $A \to B$ — цепное правило из P.

Допустим, что LR(k)-алгоритм разбора, использующий это множество таблиц, обладает следующим свойством: всякий раз, когда основа $Y_1 Y_2 \ldots Y_r$ свертывается в B при аванцепочке u, очередным шагом будет свертка B в A. Часто оказывается возможным изменить множество таблиц так, чтобы основа $Y_1 Y_2 \ldots Y_r$ свертывалась в A за один шаг. Выясним, при каких условиях это можно сделать.

Пусть p — номер правила $A \to B$. Предположим, что $T = \langle f, g \rangle$ — такая таблица, что $f(u) =$ **свертка** p для некоторой

аванцепочки u. Пусть $\mathscr{T}' = \mathrm{GOTO}^{-1}(T, B)$ и $\mathscr{U} = \{U \mid U = \mathrm{GOTO}(T', A),\ T' \in \mathscr{T}'\}$. Множество $\mathscr{T}'$ состоит из таблиц, которые могут появиться в магазине непосредственно под B, при условии, что над B находится таблица T. Если $(\mathscr{T}, T_0)$ — каноническое множество LR(k)-таблиц, то $\mathscr{U} = \mathrm{NEXT}(T, p)$ (упр. 7.3.19).

Для того чтобы исключить свертку по правилу p, заменим для каждой таблицы $\langle f', g'\rangle$ из $\mathscr{T}'$ значение элемента $g'(B)$ с имени таблицы T на $g'(A)$. Тогда вместо двух тактов, а именно

$$(\gamma T' Y_1 U_1 Y_2 U_2 \ldots Y_r U_r, w, \pi) \vdash (\gamma T' BT, w, \pi i)$$
$$\vdash (\gamma T' AU, w, \pi i p)$$

анализатор сделает только один такт

$$(\gamma T' Y_1 U_1 Y_2 U_2 \ldots Y_r U_r, w, \pi) \vdash (\gamma T' AU, w, \pi i)\ ^{1)}$$

Такая замена элемента $g'(B)$ возможна при условии, что элементы таблицы T и всех таблиц из $\mathscr{U}$ совпадают всегда, кроме тех случаев, когда аванцепочка вызывает свертку по правилу p. Другими словами, пусть $T = \langle f, g\rangle$ и $\mathscr{U} = \{\langle f_1, g_1\rangle, \langle f_2, g_2\rangle, \ldots, \langle f_m, g_m\rangle\}$. Тогда требуется, чтобы

(1) для всех u из Σ^{*k} если $f(u)$ — не φ и не **свертка** p, то $f_i(u)$ — либо φ, либо совпадает с $f(u)$ при $1 \leqslant i \leqslant m$,

(2) для всех X из $\mathrm{N} \cup \Sigma$ если $g(X) \neq \varphi$, то $g_i(X)$ — либо φ, либо совпадает с $g(X)$ при $1 \leqslant i \leqslant m$.

Если оба эти требования удовлетворяются, преобразуем таблицы в множествах $\mathscr{T}'$ и $\mathscr{U}$ так:

(3) положим $g'(B) = g'(A)$ для всех $\langle f', g'\rangle$ из $\mathscr{T}'$,

(4) при $1 \leqslant i \leqslant m$

(а) для каждого $u \in \Sigma^{*k}$ заменим $f_i(u)$ на $f(u)$, если $f_i(u) = \varphi$ и $f(u)$ — не φ и не **свертка** p,

(б) для каждого $X \in \mathrm{N} \cup \Sigma$ заменим $g_i(X)$ на $g(X)$, если $g(X) \neq \varphi$.

Изменение, обусловленное применением правила (3), может сделать таблицу T недостижимой, если ее можно достичь только в результате обращения к элементам $g'(B)$ таблиц $\langle f', g'\rangle$ из множества $\mathscr{T}'$.

Заметим, что измененный анализатор может помещать в магазин символы и таблицы, которые не записывались туда исходным анализатором. Например, предположим, что исходный анализатор выполняет свертку в нетерминал B и затем перенос:

$$(\gamma T' Y_1 U_1 Y_2 U_2 \ldots Y_r U_r, aw, \pi) \vdash (\gamma T' BT, aw, \pi i)$$
$$\vdash (\gamma T' BTaT'', w, \pi i)$$

[1]) На самом деле в результате описанных замен анализатор достигает конфигурации $(\gamma T' BU, w, \pi i)$, а не $(\gamma T' AU, w, \pi i)$. Впрочем, это замечание не влияет на ход дальнейших рассуждений.— *Прим. перев.*

Новый анализатор выполняет ту же последовательность тактов, но в магазине будут появляться другие символы. В этом случае будет

$$(\gamma T' Y_1 U_1 Y_2 U_2 \ldots Y_r U_r, aw, \pi) \vdash (\gamma T' AU', aw, \pi i)$$
$$\vdash (\gamma T' AU' aT'', w, \pi i)$$

Пусть $T = \langle f, g\rangle$, $T' = \langle f', g'\rangle$ и $U = \langle f_i, g_i\rangle$. Тогда $U = g'(A)$. Таблица U' была построена по U в соответствии с правилом (4), сформулированным выше. Поэтому, если $f(v) =$ **перенос**, где $v = \mathrm{FIRST}(aw)$, то ясно, что $f_i'(v) =$ **перенос**. Более того, известно, что $g_i(a)$ совпадает с $g(a)$. Значит, новый анализатор выполняет правильную последовательность тактов, за исключением того, что он игнорирует вопрос о том, была ли в действительности произведена свертка по правилу $A \to B$.

На последующих тактах анализатор не просматривает символы грамматики, находящиеся в магазине. Так как элементы перехода таблиц U' и T совпадают, можно быть уверенным в том, что поведение обоих анализаторов будет одинаковым (за исключением сверток по цепному правилу $A \to B$).

Это преобразование можно повторить с новым множеством таблиц, чтобы исключить возможно больше сверток по несущественным с точки зрения семантики цепным правилам.

Пример 7.23. Исключим возможно больше сверток по цепным правилам в множестве LR(1)-таблиц для грамматики G_0, представленной на рис. 7.32. Таблица T_2 вызывает свертку по правилу 2, т. е. по правилу $E \to T$. Множеством таблиц, которые могут появиться в магазине непосредственно под T_2, является $\{T_0, T_5\}$, так как $\mathrm{GOTO}(T_0, E) = T_1$ и $\mathrm{GOTO}(T_5, E) = T_8$.

Нужно проверить, что таблицы T_2 и T_1, а также T_2 и T_8 совместимы везде, кроме элементов **свертка** 2. Действие таблицы T_2 на $*$ есть **перенос**. Действие таблиц T_1 и T_8 на $*$ есть φ. Переход таблицы T_2 на $*$ есть T_7. Переход таблиц T_1 и T_8 на $*$ есть φ. Следовательно, T_2 и T_1, а также T_2 и T_8 совместимы. Поэтому можно заменить переход таблицы T_0 на нетерминале T с T_2 на T_1 и переход таблицы T_5 на нетерминале T с T_2 на T_8. Нужно также заменить действия таблиц T_1 и T_8 на символе $*$ с φ на **перенос**, поскольку действие T_2 на $*$ есть **перенос**. Наконец, заменяем переходы таблиц T_1 и T_8 на символе $*$ с φ на T_7, потому что переход таблицы T_2 на $*$ есть T_7.

Таблица T_2 теперь недостижима из T_0, и, значит, ее можно исключить.

Рассмотрим операции **свертка** 4, содержащиеся в таблице T_3. (Правило 4 — это правило $T \to F$.) Множеством таблиц, которые могут появиться непосредственно под T_3, является $\{T_0, T_5, T_6\}$. Теперь $\mathrm{GOTO}(T_0, T) = T_1$, $\mathrm{GOTO}(T_5, T) = T_8$ и $\mathrm{GOTO}(T_6, T) = T_{13}$. (До описанного преобразования $\mathrm{GOTO}(T_0, T) = \mathrm{GOTO}(T_5, T) = T_2$.)

Нужно проверить, что таблица T_3 совместима с каждой из таблиц T_1, T_8 и T_{13}. Ясно, что это так, поскольку действие таблицы T_3—есть либо φ, либо **свертка** 4, а переходы ее все равны φ.

Таким образом, можно заменить переходы таблиц T_6, T_5 и T_6 на символе F на T_1, T_8 и T_{13} соответственно. Это делает таблицу T_3 недостижимой из T_0.

Полученное множество таблиц показано на рис. 7.33,*а*. Таблицы T_2 и T_3 исключены.

Заметим, что в столбцах, расположенных под символами E, T и F, все элементы перехода оказались совместимыми. Поэтому можно слить эти три столбца в один. Пометим этот новый столбец символом E. Новое множество таблиц приведено на рис. 7.33,*б*.

В алгоритм разбора осталось внести лишь одно изменение, а именно при вычислении элементов перехода вместо символов T и F использовать E. В результате множество таблиц рис. 7.33,*б* приводит к разбору согласно остовной грамматике

$$
\begin{array}{ll}
(1) & E \to E+E \\
(3) & E \to E*E \\
(5) & E \to (E) \\
(6) & E \to a
\end{array}
$$

определенной в разд. 5.4.3.

Разберем, например, входную цепочку $(a+a)*a$ с помощью множества таблиц рис. 7.33,*б*. LR(1)-анализатор выполняет такую последовательность тактов:

$$
\begin{aligned}
[T_0, (a+a)*a, e] &\vdash [T_0(T_5, a+a)*a, e] \\
&\vdash [T_0(T_5aT_4, +a)*a, e] \\
&\vdash [T_0(T_5ET_8, +a)*a, 6] \\
&\vdash [T_0(T_5ET_8+T_6, a)*a, 6] \\
&\vdash [T_0(T_5ET_8+T_6aT_4,)*a, 6] \\
&\vdash [T_0(T_5ET_8+T_6ET_{13},)*a, 66] \\
&\vdash [T_0(T_5ET_8,)*a, 661] \\
&\vdash [T_0(T_5ET_8)T_{15}, *a, 661] \\
&\vdash [T_0ET_1, *a, 6615] \\
&\vdash [T_0ET_1*T_7, a, 6^{\wedge}15] \\
&\vdash [T_0ET_1*T_7aT_4, e, 6615] \\
&\vdash [T_0ET_1*T_7ET_{14}, e, 66156] \\
&\vdash [T_0ET_1, e, 661563]
\end{aligned}
$$

Последняя конфигурация—допускающая. Канонический LR(1)-анализатор для G_0 при разборе этой же входной цепочки сделал

	Действие						Переход							
	a	+	*	(	)	e	E	T	F	a	+	*	(	)
T_0	S	X	X	S	X	X	T_1	T_1	T_1	T_4	φ	φ	T_5	φ
T_1	φ	S	S	φ	X	A	φ	φ	φ	φ	T_6	T_7	φ	φ
T_4	X	6	6	X	6	6	φ	φ	φ	φ	φ	φ	φ	φ
T_5	S	X	X	S	X	X	T_8	T_8	T_8	T_4	φ	φ	T_5	φ
T_6	S	X	X	S	X	X	φ	T_{13}	T_{13}	T_4	φ	φ	T_5	φ
T_7	S	X	X	S	X	X	φ	φ	T_{14}	T_4	φ	φ	T_5	φ
T_8	φ	S	S	φ	S	X	φ	φ	φ	φ	T_6	T_7	φ	T_{15}
T_{13}	φ	1	S	φ	1	1	φ	φ	φ	φ	φ	T_7	φ	φ
T_{14}	φ	3	3	φ	3	3	φ	φ	φ	φ	φ	φ	φ	φ
T_{15}	X	5	5	X	5	5	φ	φ	φ	φ	φ	φ	φ	φ

а

	Действие						Переход					
	a	+	*	(	)	e	E	a	+	*	(	)
T_0	S	X	X	S	X	X	T_1	T_4	φ	φ	T_5	φ
T_1	φ	S	S	φ	X	A	φ	φ	T_6	T_7	φ	φ
T_4	X	6	6	X	6	6	φ	φ	φ	φ	φ	φ
T_5	S	X	X	S	X	X	T_8	T_4	φ	φ	T_5	φ
T_6	S	X	X	S	X	X	T_{13}	T_4	φ	φ	T_5	φ
T_7	S	X	X	S	X	X	T_{14}	T_4	φ	φ	T_5	φ
T_8	φ	S	S	φ	S	X	φ	φ	T_6	T_7	φ	T_{15}
T_{13}	φ	1	S	φ	1	1	φ	φ	φ	T_7	φ	φ
T_{14}	φ	3	3	φ	3	3	φ	φ	φ	φ	φ	φ
T_{15}	X	5	5	X	5	5	φ	φ	φ	φ	φ	φ

б

Рис. 7.33. Множество LR(1)-таблиц после исключения цепных правил: *а*—до слияния столбцов; *б*—после слияния столбцов.

бы пять дополнительных тактов, соответствующих сверткам по цепным правилам. □

Метод исключения цепных правил формально описан в алгоритме 7.9. Хотя мы и не будем подробно доказывать его корректность, этот алгоритм позволит нам исключить все свертки по цепным правилам, если для каждого нетерминала грамматика содержит не более одного цепного правила. Даже если какой-то нетерминал имеет более одного цепного правила, этот алгоритм все-таки будет работать достаточно хорошо.

Алгоритм 7.9. Исключение сверток по цепным правилам.

Вход. LR(k)-грамматика $G=(\mathrm{N}, \Sigma, P, S)$ и φ-недостижимое множество $(\mathscr{T}, T_0)$ таблиц для G.

Выход. Измененное множество таблиц для G, „эквивалентное" множеству $(\mathscr{T}, T_0)$ в том смысле, что оно позволяет так же рано обнаруживать ошибки, но может привести к неудаче при свертке по некоторым цепным правилам.

Метод.

(1) Упорядочить нетерминалы так, чтобы $\mathrm{N}=\{A_1, \ldots, A_n\}$ и если $A_i \to A_j$ — цепное правило, то $i<j$. Возможность такого упорядочения обеспечивается однозначностью грамматики.

(2) Выполнять шаг (3) последовательно для $j=1, 2, \ldots, n$.

(3) Пусть $A_i \to A_j$ — цепное правило с номером p и T_1 — таблица, вызывающая действие **свертка** p на одной или более аванцепочках. Предположим, что T_2 принадлежит множеству $\mathrm{NEXT}(T_1, p)$ [1]) и для всех аванцепочек либо действия таблиц T_1 и T_2 совпадают, либо одно из них есть φ, либо действием таблицы T_1 служит **свертка** p. Наконец, допустим, что элементы перехода таблиц T_1 и T_2 также либо совпадают, либо один из них есть φ. Построим теперь новую таблицу T_3, элементы которой всегда совпадают с отличными от φ элементами таблиц T_1 и T_2. Исключение составляют элементы, соответствующие тем аванцепочкам, которым в таблице T_1 отвечает **свертка** p. В этом случае действия таблицы T_3 будут совпадать с соответствующими действиями таблицы T_2. Затем преобразуем каждую таблицу $T=\langle f, g\rangle$, для которой $g(A_i)=T_2$ и $g(A_j)=T_1$, положив $g(A_i)=g(A_j)=T_3$.

(4) По окончании преобразований шага (3) устраним все таблицы, не являющиеся более достижимыми из начальной таблицы. □

Установим ряд результатов, необходимых для доказательства того, что для LR(1)-грамматики алгоритм 7.9 полностью исключает свертки по цепным правилам, если нет двух правил вида $A \to B$ и $A \to C$, при условии что ни из какого нетерминала грамматики не выводится только пустая цепочка. (Такой нетерминал легко убрать, причем после этого грамматика останется LR(1)-грамматикой.)

[1]) Следует отметить, что всякий раз, когда применение шага (3) влечет за собой изменение множества таблиц, соответственно изменяется и связанная с этим множеством функция NEXT.

Лемма 7.1. *Пусть $G=(\mathrm{N}, \Sigma, P, S)$ — LR(1)-грамматика, обладающая тем свойством, что для каждого $C \in \mathrm{N}$ существует такая цепочка $w \neq e$ из Σ^*, что $C \Rightarrow^* w$. Допустим, что $A \Rightarrow^+ B$ с помощью последовательности цепных правил. Пусть $\mathcal{A}_1$ и $\mathcal{A}_2$ — множества LR(1)-ситуаций, допустимых для активных префиксов γA и γB соответственно. Тогда удовлетворяются следующие условия:*

(1) *Если $[C \to \alpha_1 \cdot X\beta_1, a] \in \mathcal{A}_1$ и $[D \to \alpha_2 \cdot Y\beta_1, b] \in \mathcal{A}_2$, то $X \neq Y$.*

(2) *Если $[C \to \alpha_1 \cdot \beta_1, a] \in \mathcal{A}_1$ и $b \in \mathrm{EFF}(\beta_1 a)$* [1]), *то в $\mathcal{A}_2$ нет такой ситуации вида $[D \to \alpha_2 \cdot \beta_2, c]$, что $b \in \mathrm{EFF}(\beta_2 c)$, кроме, быть может, ситуации $[E \to B \cdot, \beta]$, где $A \Rightarrow^* E \Rightarrow B$ с помощью последовательности цепных правил.*

Доказательство. Предоставляем читателю в качестве упражнения убедиться в том, что нарушение условия (1) или (2) приводит к противоречию с LR(1)-условием. □

Следствие. LR(1)-*таблицы, построенные по $\mathcal{A}_1$ и $\mathcal{A}_2$, совпадают на всех элементах перехода и действия, кроме случаев, когда один из них есть φ или когда таблица, построенная по $\mathcal{A}_2$, вызывает на этом элементе свертку по цепному правилу.*

Доказательство. Согласно теореме 7.7, поскольку две таблицы построены по множествам допустимых ситуаций для цепочек, оканчивающихся нетерминалом, все элементы **ошибка** несущественны. Выполнение условия (1) леммы 7.1 обеспечивает отсутствие конфликтов в элементах перехода; условие (2) утверждает, что конфликтов нет и в элементах действия, за исключением разрешимых конфликтов, относящихся к цепным правилам. □

Лемма 7.2. *В ходе работы алгоритма 7.9 каждая таблица является результатом слияния списка (возможно, длины 1) LR(1)-таблиц $T_1, \ldots, T_n$, построенных по множествам допустимых ситуаций для некоторых активных префиксов $\gamma A_1, \gamma A_2, \ldots, \gamma A_n$, где $A_i \to A_{i+1}$ принадлежит P для $1 \leqslant i \leqslant n$.*

Доказательство. Оставляем в качестве упражнения. □

[1]) Заметим, что здесь β_1 может равняться e; тогда $\mathcal{A}_1$ на аванцепочке b вызывает свертку. В противном случае $\mathcal{A}_1$ вызывает перенос.

Теорема 7.8. *Пусть алгоритм 7.9 применяется к* LR(1)-*грамматике G и ее каноническому множеству* LR(1)-*таблиц $\mathscr{T}$. Если G имеет для каждого нетерминала не более одного цепного правила и ни из какого нетерминала не выводится только e, то полученное множество таблиц не содержит сверток по цепным правилам.*

Доказательство. Интуитивно ясно, что леммы 7.1 и 7.2 гарантируют, что все пары таблиц T_1 и T_2, рассмотренные на шаге (3), действительно удовлетворяют условиям этого шага. Формальное доказательство оставляем в качестве упражнения. □

УПРАЖНЕНИЯ

7.3.1. Постройте канонические множества LR(1)-таблиц для следующих грамматик:

(а) $S \to ABAC$
$A \to aD$
$B \to b \mid c$
$C \to c \mid d$
$D \to D0 \mid 0$

(б) $S \to aSS \mid b$

(в) $S \to SSa \mid b$

(г) $E \to E + T \mid T$
$T \to T * F \mid F$
$F \to P \uparrow F \mid P$
$P \to (E) \mid a \mid a(L)$
$L \to L, E \mid E$

7.3.2. Используйте каноническое множество таблиц из упр. 7.3.1(а) и алгоритм 7.5 для разбора входной цепочки $a0ba00c$.

7.3.3. Покажите, как можно реализовать алгоритм 7.5
(а) с помощью детерминированного МП-преобразователя,
(б) в виде анализатора на языке Флойда—Эванса.

7.3.4. Постройте анализатор на языке Флойда—Эванса для грамматики G_0 по
(а) множеству LR(1)-таблиц, приведенному на рис. 7.33, *а*,
(б) множеству LR(1)-таблиц, приведенному на рис. 7.33, *б*.
Сравните полученные анализаторы с анализатором из упр. 7.2.11.

7.3.5. Рассмотрим грамматику G, порождающую язык

$$L = \{a^n 0 a^i b^n \mid i,\ n \geqslant 0\} \cup \{0 a^n 1 a^i c^n \mid i,\ n \geqslant 0\}:$$

$$S \to A \mid 0B$$
$$A \to aAb \mid 0 \mid 0C$$
$$B \to aBc \mid 1 \mid 1C$$
$$C \to aC \mid a$$

Сконструируйте для G анализатор простого предшествования и LR(1)-анализатор. Покажите, что анализатор простого предшествования при разборе цепочки $a^n 1 a^i b$, прежде чем сообщить об ошибке, прочтет все символы a, следующие за символом 1. Покажите, что канонический LR(1)-анализатор сообщит об ошибке, как только прочтет символ 1.

7.3.6. С помощью алгоритма 7.6 постройте φ-недостижимые множества LR(1)-таблиц для каждой грамматики из упр. 7.3.1.

7.3.7. Пользуясь методами настоящего раздела, постройте наименьшее эквивалентное множество LR(1)-таблиц для каждой грамматики из упр. 7.3.1.

7.3.8. Пусть $\mathscr{S}$ — каноническая система множеств LR(k)-ситуаций для LR(k)-грамматики $G = (N, \Sigma, P, S)$. Пусть $\mathscr{A}$ — множество ситуаций, принадлежащее $\mathscr{S}$. Покажите, что

(а) если $[A \to \alpha \cdot \beta,\ u] \in \mathscr{A}$, то $u \in \mathrm{FOLLOW}_k(A)$,

(б) если $\mathscr{A}$ не является начальным множеством ситуаций, то $\mathscr{A}$ содержит по меньшей мере одну ситуацию вида $[A \to \alpha X \cdot \beta,\ u]$ для некоторого X из $N \cup \Sigma$,

(в) если $[B \to \cdot \beta,\ v] \in \mathscr{A}$ и $B \neq S'$, то в $\mathscr{A}$ есть ситуация вида $[A \to \alpha \cdot B\gamma,\ u]$,

(г) если $[A \to \alpha \cdot B\beta,\ u] \in \mathscr{A}$ и $\mathrm{EFF}_1(B\beta u)$ содержит символ a, то $\mathscr{A}$ есть ситуация вида $[C \to \cdot a\gamma,\ v]$ для некоторых γ и v. (Этот результат позволяет получить простой метод вычисления элементов **перенос** в LR(1)-анализаторе.)

***7.3.9.** Покажите, что если в множестве $\mathscr{T}'$ LR(k)-таблиц, построенном алгоритмом 7.6, заменить какой-нибудь элемент **ошибка** на φ, то новое множество таблиц не будет более φ-недостижимым.

***7.3.10.** Покажите, что канонический LR(k)-анализатор сообщает об ошибке либо в начальной конфигурации, либо сразу после операции переноса.

****7.3.11.** Пусть G — LR(k)-грамматика. Найдите верхнюю и нижнюю границы для числа таблиц в каноническом множестве LR(k)-таблиц для G. Можете ли Вы найти разумные верхнюю

и нижнюю границы числа таблиц в произвольном допустимом множестве LR(k)-таблиц для G?

***7.3.12.** Видоизмените алгоритм 7.6 так, чтобы им можно было воспользоваться для построения φ-недостижимого множества LR(0)-таблиц для LR(0)-грамматики.

***7.3.13.** Разработайте алгоритм нахождения всех элементов φ в произвольном множестве LR(k)-таблиц.

***7.3.14.** Разработайте алгоритм нахождения всех элементов φ в произвольном множестве LL(k)-таблиц.

****7.3.15.** Разработайте алгоритм нахождения всех элементов φ в LC(k)-таблице разбора.

***7.3.16.** Разработайте удобный алгоритм нахождения совместимых разбиений множества LR(k)-таблиц.

7.3.17. Найдите совместимые разбиения множеств LR(1)-таблиц из упр. 7.3.1.

7.3.18. Покажите, что отношение совместимости LR(k)-таблиц рефлексивно и симметрично, но не транзитивно.

7.3.19. Пусть $(\mathscr{T}, T_0)$ — каноническое множество LR(k)-таблиц для LR(k)-грамматики G. Покажите, что множество GOTO (T_0, α) непусто тогда и только тогда, когда α — активный префикс грамматики G. Верно ли это утверждение для произвольного допустимого множества LR(k)-таблиц для G?

***7.3.20.** Пусть $\mathscr{T}$ — каноническое множество LR(k)-таблиц для G. Найдите верхнюю границу веса произвольной таблицы из $\mathscr{T}$ (как функцию от G).

7.3.21. Пусть $\mathscr{T}$ — каноническое множество LR(k)-таблиц для LR(k)-грамматики $G = (N, \Sigma, P, S)$. Покажите, что NEXT(T, p) для всех $T \in \mathscr{T}$ равно $\{\text{GOTO}(T', A) \mid T' \in \text{GOTO}^{-1}(T, \alpha)$, правило p есть $A \to \alpha\}$.

7.3.22. Дайте алгоритм вычисления NEXT(T, p) для произвольного множества LR(k)-таблиц для G.

***7.3.23.** Пусть $\mathscr{T}$ — каноническое множество LR(k)-таблиц. Предположим, что для каждой таблицы $T \in \mathscr{T}$, содержащей одно или более действий свертки, выбирается одно правило с номером p, по которому T производит свертку, и все действия **ошибка** и φ заменяются на **свертку** p. Покажите, что полученное множество таблиц эквивалентно $\mathscr{T}$.

***7.3.24.** Покажите, что с помощью алгоритма 7.9 не всегда удается исключить свертки по цепным правилам из множества LR(k)-таблиц для LR(k)-грамматики.

***7.3.25.** Докажите, что алгоритм 7.9 строит множество LR(k)-таблиц, эквивалентное исходному, не считая сверток по цепным правилам.

7.3.26. Будем говорить, что бинарная операция θ *связывает слева направо*, если цепочку $a\theta b\theta c$ следует понимать как $((a\theta b)\theta c)$. Постройте LR(1)-грамматику, порождающую выражения над алфавитом, состоящим из символов $\{a, (,)\}$ и знаков операций $\{+, -, *, /, \uparrow\}$. Все операции бинарные, за исключением $+$ и $-$, которые и бинарны, и (префиксно) унарны. Все бинарные операции, кроме $\uparrow$, связывают слева направо. Бинарные операции $+$ и $-$ имеют приоритет 1, операции $*$ и $/$ — приоритет 2, операция $\uparrow$ и две унарные операции — приоритет 3.

****7.3.27.** Разработайте метод автоматического построения LR(1)-анализатора для языка выражений. Выражения определяются над множеством операций со связками и приоритетами, как в упр. 7.3.26.

***7.3.28.** Докажите, что множества таблиц $\mathscr{T}$ и $\mathscr{U}$ из примера 7.17 эквивалентны.

****7.3.29.** Покажите, что проблема определения эквивалентности двух множеств LR(k)-таблиц разрешима.

Определение. Следующие правила порождают арифметические выражения, в которых $\theta_1, \theta_2, \ldots, \theta_n$ представляют бинарные операции n различных приоритетов. θ_1 имеет самый низкий приоритет, а θ_n — самый высокий. Операции связывают слева направо

$$
\begin{aligned}
E_0 &\to E_0\theta_1E_1 \mid E_1\\
E_1 &\to E_1\theta_2E_2 \mid E_2\\
&\;\;\vdots\\
E_{n-1} &\to E_{n-1}\theta_nE_n \mid E_n\\
E_n &\to (E_1) \mid a
\end{aligned}
$$

Это же множество выражений можно получить с помощью так называемой „LR(1)-грамматики с индикаторами“:

$$
\begin{array}{lll}
(1) & E_i \to E_i\theta_jE_j & 0 \leqslant i < j \leqslant n\\
(2) & E_i \to (E_0) & 0 \leqslant i \leqslant n\\
(3) & E_i \to a & 0 \leqslant i \leqslant n
\end{array}
$$

В этих правилах индексы можно рассматривать как индикаторы нетерминала E и терминала θ. Условия, налагаемые на индикаторы, отражают приоритет операций. Например, первое правило

указывает, что выражение приоритета i — это выражение приоритета i, за которым идет операция приоритета j, а за ней — выражение приоритета j при условии $0 \leqslant i \leqslant j \leqslant n$. Начальный символ имеет приоритет 0.

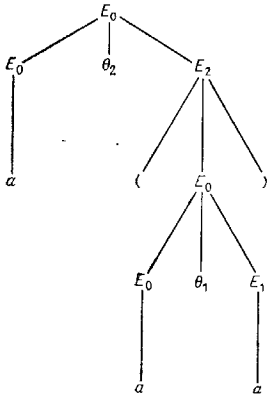


Рис. 7.34. Дерево разбора

Дерево разбора для выражения $a\theta_2(a\theta_1 a)$, аналогичного $a*(a+a)$, изображено на рис. 7.34. На дереве показаны значения индикаторов, связанных с нетерминалами.

Хотя грамматика с индикаторами при отсутствии индикаторов неоднозначна, можно построить LR(1)-подобный анализатор, использующий для правильного разбора входной цепочки индикаторы с LR(1)-таблицами. Такой LR(1)-анализатор представлен на рис. 7.35.

Чтобы проиллюстрировать поведение этого анализатора, разберем входную цепочку $a\theta_2(a\theta_1 a)$. Анализатор начинает работу в конфигурации $([T_0, 0], a\theta_2(a\theta_1 a), e)$, в которой с начальной таблицей T_0 связан индикатор 0. Действием пары $[T_0, i]$ на входном символе a служит перенос, поэтому анализатор переходит в конфигурацию $([T_0, 0]aT_2, \theta_2(a\theta_1 a), e)$.

	Действие					*Переход*				
	a	θ_j	(	)	e	E	a	θ_j	(	)
$[T_0,i]$	S	X	S	X	X	$[T_1,i]$	T_2	X	$[T_3,0]$	X
$[T_1,i]$	X	S	X	X	A	X	X	$[T_4,j]$	X	X
T_2	X	3	X	3	3	X	X	X	X	X
$[T_3,0]$	S	X	S	X	X	$[T_5,i]$	T_2	X	$[T_3,0]$	X
$[T_4,j]$	S	X	S	X	X	$[T_6,i]$	T_2	X	$[T_3,0]$	X
$[T_5,i]$	X	S	X	S	X	X	X	$[T_4,j]$	X	T_7
$[T_6,i]$	X	$R1$	X	$R2$	$R2$	X	X	$[T_4,j]$	X	X
T_7	X	2	X	2	2	X	X	X	X	X

Рис. 7.35. LR(1)-анализатор с индикаторами из упр. 7.3.30.

Действием таблицы T_2 на входном символе 0_2 является свертка по правилу 3, т. е. по правилу $E \to a$. Переход пары $[T_0, i]$ на символе E есть $[T_1, i]$. Значение индикатора передается от T_0 к T_1. Таким образом, анализатор переходит в конфигурацию

$$([T_0, 0]E[T_1, 0],\ \theta_2(a\theta_1 a),\ 3)$$

Законченная последовательность шагов анализатора выглядит так:

$$\begin{aligned}
&([T_0, 0],\ a\theta_2(a\theta_1 a),\ e)\\
\vdash\ &([T_0, 0]aT_2,\ \theta_2(a\theta_1 a),\ e)\\
\vdash\ &([T_0, 0]E[T_1\ 0],\ 0_2(a\theta_1 a),\ 3)\\
\vdash\ &([T_0, 0]E[T_1, 0]\theta_2[T_4, 2],\ (a\theta_1 a),\ 3)\\
\vdash\ &([T_0, 0]E[T_1, 0]\theta_2[T_4,2]([T_3, 0],\ a\theta_1 a),\ 3)\\
\vdash\ &([T_0, 0]E[T_1, 0]\theta_2[T_4, 2]([T_3, 0]aT_2\ \theta_1 a),\ 3)\\
\vdash\ &([T_0, 0]E[T_1, 0]\theta_2[T_4, 2]([T_3, 0]E[T_5, 0],\ \theta_1 a),\ 33)\\
\vdash\ &([T_0, 0]E[T_1, 0]\theta_2[T_4, 2]([T_3, 0]E[T_5, 0]\theta_1[T_4, 1],\ a),\ 33)\\
\vdash\ &([T_0, 0]E[T_1, 0]\theta_2[T_4, 2]([T_3, 0]E[T_5, 0]0_1[T_4, 1]aT_2,\ ),\ 33)\\
\vdash\ &([T_0, 0]E[T_1, 0]\theta_2[T_4, 2]([T_3, 0]E[T_5, 0]0_1[T_4, 1]E[T_6, 1],\ ),\ 333)\\
\vdash\ &([T_0, 0]E[T_1, 0]\theta_2[T_4, 2]([T_3, 0]E[T_5, 0]\ ),\ 3331_1)\\
\vdash\ &([T_0, 0]E[T_1, 0]\theta_2[T_4,2]([T_3, 0]E[T_5, 0])T_7,\ e,\ 3331_1)\\
\vdash\ &([T_0, 0]E[T_1, 0]\theta_2[T_4, 2]E[T_6, 2],\ e,\ 3331_12)\\
\vdash\ &([T_0, 0]E[T_1, 0],\ e,\ 3331_121_2)
\end{aligned}$$

**7.3.30. Покажите, что анализатор, приведенный на рис. 7.35, правильно разбирает все выражения, порождаемые грамматикой с индикаторами.

7.3.31. Постройте LR(1) — анализатор для грамматики без индикаторов с операциями n различных приоритетов. Каков размер этого анализатора по сравнению с анализатором с индикаторами, приведенным на рис. 7.35? Сравните быстродействия двух таких анализаторов.

*7.3.32. Постройте LR(1) — подобный анализатор с индикаторами для языка выражений, включающих бинарные операции, причем одни операции связывают слева направо, другие — справа налево.

**7.3.33. Следующая грамматика с индикаторами порождает выражения, включающие бинарные операции n различных приоритетов:

(1)	$E_i \to (E_0)R_{i,n}$	$0 \leqslant i \leqslant n$
(2)	$E_i \to aR_{i,n}$	$0 \leqslant i \leqslant n$
(3)	$R_{i,k} \to 0_j E_j R_{i,j-1}$	$0 \leqslant i < j \leqslant k \leqslant n$
(4)	$R_{i,j} \to e$	$0 \leqslant i \leqslant j \leqslant n$

Постройте для этой грамматики LL(1)-подобный анализатор с индикаторами. *Указание*: Хотя в этой грамматике у R два указателя, анализатору нужен только первый из них.

7.3.34. Закончите доказательство леммы 7.1 и ее следствия.

7.3.35. Докажите лемму 7.2.

7.3.36. Закончите доказательство теоремы 7.8.

Открытая проблема

7.3.37. При каких условиях можно после исключения сверток по цепным правилам слить все столбцы перехода для нетерминалов, как это делалось для грамматики G_0? Предлагаем читателю исследовать возможность связать этот вопрос с операторным предшествованием. Напомним, что G_0 — грамматика операторного предшествования.

Проблемы для исследования

7.3.38. Разработайте дополнительные методы преобразования множеств LR-таблиц, сохраняющие „эквивалентность" в понимаемом нами смысле.

7.3.39. Разработайте методы компактного представления LR-таблиц, использующие преимущества элементов φ.

Упражнения на программирование

7.3.40. Придумайте элементарные операции, которыми можно пользоваться при реализации LR(1)-анализатора. Такими операциями могут быть чтение входного символа, запись символа в магазин, выталкивание определенного числа символов из магазина, посылка в выходную цепочку и т.д. Постройте интерпретатор, выполняющий эти элементарные операции.

7.3.41. Постройте программу, получающую на вход множество LR(1)-таблиц и порождающую на выходе последовательность элементарных команд, которая моделирует LR(1)-анализатор, работающий с этим множеством таблиц.

7.3.42. Постройте программу, получающую на вход множество LL(1)-таблиц и порождающую на выходе последовательность элементарных команд, которая моделирует LL(1)-анализатор, работающий с этим множеством таблиц.

7.3.43. Напишите программу, добавляющую к каноническому множеству LR-таблиц несущественные элементы.

***7.3.44.** Напишите программу, которая для построения компактных множеств таблиц применяет эвристические методы реализации отсрочки в обнаружении ошибок и выполнении слияния таблиц.

7.3.45. Реализуйте алгоритм 7.9, исключающий, когда это возможно, свертки по цепным правилам.

Замечания по литературе

Преобразования, описанные в этом разделе, были предложены Ахо и Ульманом [1972в, 1972г]. Пейджер [1970] рассмотрел другой подход к задаче упрощения LR(k)-анализаторов, при котором анализатор изменяется так, чтобы ошибки обнаруживались не в той же позиции входной цепочки, что и в случае канонического анализатора; кроме того, для определения нужной свертки требуется просматривать содержимое магазина. Идея использования в LL-грамматиках и анализаторах индикаторов принадлежит Льюису, Розенкранцу и Стирнзу. В работе Льюиса и Розенкранца [1971] сообщается, что использование индикаторов при обработке выражений и условных операторов позволило уменьшить синтаксический анализатор в их компиляторе Алгола 60 с 37 до 29 LL(1)-таблиц разбора.

7.4. МЕТОДЫ ПОСТРОЕНИЯ LR(k)-АНАЛИЗАТОРОВ

Объем работы, выполняемой при построении множеств LR(k)-ситуаций (и, следовательно, канонического LR(k)-анализатора), резко возрастает с увеличением размера грамматики и с ростом длины аванцепочки k. Для больших грамматик объем вычислений, требующихся для построения канонического множества LR(k)-таблиц, оказывается неприемлемо большим даже при $k=1$. В этом разделе мы изучим ряд более практичных методов построения допустимых множеств LR(1)-таблиц для некоторых LR(1)-грамматик.

Первый рассматриваемый нами метод заключается в построении для грамматики G канонической системы множеств LR(0)-ситуаций. Если каждое множество LR(0)-ситуаций непротиворечиво [1]), то для G можно построить допустимое множество LR(0)-таблиц. Если множество LR(0)-ситуаций противоречиво, то для разрешения конфликтов, возникающих между действиями при разборе, имеет смысл попытаться воспользоваться в этом множестве ситуаций аванцепочками. Экономия при таком подходе достигается за счет того, что заглядывание вперед применяется только тогда, когда оно необходимо. Для многих грамматик этот подход позволяет получить множество таблиц, значительно меньшее, чем каноническое множество LR(k)-таблиц для G.

[1]) Множество LR(k)-ситуаций называют непротиворечивым, если по нему можно построить LR(k)-таблицу, у которой действия разбора определены однозначно.

Правда, для некоторых LR(k)-грамматик этот метод не работает.

Мы обсудим также другой подход к задаче построения LR(k)-анализаторов. При таком подходе большая грамматика расщепляется на части и для этих частей строятся множества LR(k)-ситуаций, а затем для получения бо́льших множеств ситуаций эти множества объединяются. Однако не всякое расщепление LR(k)-грамматики дает такие части, по которым можно построить допустимое множество таблиц для G.

7.4.1. Простые LR-грамматики

В данном разделе мы попытаемся построить анализатор для LR(k)-грамматики, строя сначала для G систему множеств LR(0)-ситуаций. Излагаемый метод работает для грамматик, составляющих подкласс LR-грамматик и называемых простыми LR-грамматиками.

Определение. Пусть $G = (\mathrm{N}, \Sigma, P, S)$ — КС-грамматика (не обязательно LR(0)). Пусть $\mathscr{S}_0$ — каноническая система множеств LR(0)-ситуаций для G, а $\mathscr{A}$ — произвольное множество ситуаций из $\mathscr{S}_0$. Предположим, что если $[A \to \alpha\cdot\beta, e]$ и $[B \to \gamma\cdot\delta, e]$ — две различные ситуации из $\mathscr{A}$, то удовлетворяется одно из условий:

(1) ни β, ни δ не равны e,
(2) $\beta \neq e$, $\delta = e$ и $\mathrm{FOLLOW}_k(B) \cap \mathrm{EFF}_k(\beta\ \mathrm{FOLLOW}_k(A)) = \varnothing$ [1]),
(3) $\beta = e$, $\delta \neq e$ и $\mathrm{FOLLOW}_k(A) \cap \mathrm{EFF}_k(\delta\ \mathrm{FOLLOW}_k(B)) = \varnothing$,
(4) $\beta = \delta = e$ и $\mathrm{FOLLOW}_k(A) \cap \mathrm{FOLLOW}_k(B) = \varnothing$.

Тогда G называется *простой* LR(k)-грамматикой (или, короче, SLR(k)-грамматикой).

Пример 7.24. Пусть, как обычно, G_0 — грамматика с правилами

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid a$$

Каноническая система множеств LR(0)-ситуаций для G_0 приведена на рис. 7.36; вторые компоненты, равные везде e, опущены.

G_0 — не SLR(0)-грамматика, потому что $\mathscr{A}_1$, например, содержит такие две ситуации $[E' \to E\cdot]$ и $[E \to E\cdot + T]$, что

$$\mathrm{FOLLOW}_0(E') = \{e\} = \mathrm{EFF}_0[+\,T\mathrm{FOLLOW}_0(E)]\ ^{2)}$$

[1]) Здесь можно воспользоваться функцией $\mathrm{FOLLOW}_{k-1}(A)$, поскольку цепочка β должна порождать хотя бы один символ любой цепочки, принадлежащей $\mathrm{EFF}_k(\beta\ \mathrm{FOLLOW}_k(A))$.

[2]) Заметим, что $\mathrm{FIRST}_0(\alpha) = \mathrm{EFF}_0(\alpha) = \mathrm{FOLLOW}_0(\alpha) = \{e\}$ для всех α.

Однако G_0 — SLR(1)-грамматика. Для того чтобы проверить выполнение SLR(1)-условия, достаточно исследовать множества,

(1) содержащие не менее двух ситуаций,

(2) содержащие ситуацию, в которой точка стоит справа от всех символов.

$\mathscr{A}_0$: $E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot a$

$\mathscr{A}_1$: $E' \to E\cdot$
$E \to E\cdot + T$

$\mathscr{A}_2$: $E \to T\cdot$
$T \to T\cdot * F$

$\mathscr{A}_3$: $T \to F\cdot$

$\mathscr{A}_4$: $F \to a\cdot$

$\mathscr{A}_{11}$: $F \to (E)\cdot$
$E \to \cdot E \mid T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot a$

$\mathscr{A}_6$: $E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot a$

$\mathscr{A}_7$: $T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot a$

$\mathscr{A}_8$: $F \to (E\cdot)$
$E \to E\cdot + T$

$\mathscr{A}_9$: $E \to E + T\cdot$
$T \to T\cdot * F$

$\mathscr{A}_{10}$: $T \to T * F\cdot$

$\mathscr{A}_5$: $F \to (\cdot E)$

Рис. 7.36. Множество LR(0)-ситуаций для G_0.

Итак, нас могут интересовать только множества $\mathscr{A}_1$, $\mathscr{A}_2$ и $\mathscr{A}_9$. Для $\mathscr{A}_1$ имеем $\mathrm{FOLLOW}_1(E') = \{e\}$ и $\mathrm{EFF}_1[+T\ \mathrm{FOLLOW}_1(E)] = \{+\}$. Так как $\{e\} \cap \{+\} = \varnothing$, то $\mathscr{A}_1$ удовлетворяет условию (3) определения SLR(1)-грамматики. По той же причине условию (3) удовлетворяют множества $\mathscr{A}_2$ и $\mathscr{A}_9$. Следовательно, можно заключить, что G_0 есть SLR(1)-грамматика. □

Теперь попытаемся построить множество LR(1)-таблиц для SLR(1)-грамматики G, начиная с канонической системы $\mathscr{S}_0$ множеств LR(0)-ситуаций для G. Пусть некоторое множество $\mathscr{A}$ из $\mathscr{S}_0$ содержит только две ситуации: $[A \to \alpha\cdot, e]$ и $[B \to \beta\cdot\gamma, e]$. По этому множеству можно построить LR(1)-таблицу. Элементы перехода строятся очевидным образом, как и для LR(0)-таблиц. Но какое действие нужно совершить, если аванцепочкой служит символ a: свертку по правилу $A \to \alpha$ или перенос? Ответить на этот вопрос можно, выяснив, верно ли, что $a \in \mathrm{FOLLOW}_1(A)$. Если $a \in \mathrm{FOLLOW}_1(A)$, то a не может принадлежать $\mathrm{EFF}_1(\gamma)$ по

определению SLR(1)-грамматики. Поэтому действием здесь будет **свертка**. Если же $a \notin \mathrm{FOLLOW}_1(A)$, то **свертка** не подходит. Если $a \in \mathrm{EFF}(\gamma)$, то действие — **перенос**, в противном случае — **ошибка**. В законченном виде этот алгоритм приведен ниже.

Алгоритм 7.10. Построение множества LR(k)-таблиц для SLR(k)-грамматики.

Вход. SLR(k)-грамматика $G=(\mathrm{N}, \Sigma, P, S)$ и каноническая система $\mathscr{S}_0$ множеств LR(0)-ситуаций для G.

Выход. Множество $(\mathscr{T}, T_0)$ LR(k)-таблиц для G, называемое SLR(k)-*множеством таблиц* для G.

Метод. Пусть $\mathscr{A}$ — множество LR(0)-ситуаций из $\mathscr{S}_0$. LR(k)-таблица T, связанная с $\mathscr{A}$, представляет собой пару $\langle f, g\rangle$, построенную так:

(1) Для всех u из Σ^{*k}
  (а) $f(u)=$ **перенос**, если $[A \to \alpha\cdot\beta, e]$ принадлежит $\mathscr{A}$, $\beta \neq e$ и $u \in \mathrm{EFF}_k(\beta\ \mathrm{FOLLOW}_k(A))$,
  (б) $f(u)=$ **свертка** i, если $[A \to \alpha\cdot, e] \in \mathscr{A}$, правило i есть $A \to \alpha$ и $u \in \mathrm{FOLLOW}_k(A)$,
  (в) $f(e)=$ **допуск**, если $[S' \to S\cdot, e] \in \mathscr{A}$ [1]),
  (г) $f(u)=$ **ошибка** в остальных случаях.

(2) $g(X)$ для всех X из $\mathrm{N} \cup \Sigma$ представляет собой таблицу, построенную по $\mathrm{GOTO}(\mathscr{A}, X)$. Начальная таблица T_0 — это таблица, связанная с множеством, содержащим ситуацию $[S' \to \cdot S, e]$. □

Алгоритм 7.10 аналогичен нашему первоначальному методу построения множества таблиц по системе множеств ситуаций, приведенному в разд. 5.2.5. Пусть $\mathscr{A}'$ — множество таких ситуаций $[A \to \alpha\cdot\beta, u]$, что $[A \to \alpha\cdot\beta, e] \in \mathscr{A}$ и $u \in \mathrm{FOLLOW}_k(A)$, и пусть $\mathscr{S}'_0=\{\mathscr{A}' \mid \mathscr{A} \in \mathscr{S}_0\}$. Тогда в результате применения к $\mathscr{S}'_0$ алгоритма 7.10 получается то же множество таблиц, что и в результате применения к $\mathscr{S}'_0$ метода, описанного в разд. 5.2.5.

Из определения сразу видно, что каждое множество ситуаций из $\mathscr{S}'_0$ непротиворечиво тогда и только тогда, когда G — SLR(k)-грамматика.

Пример 7.25. Построим SLR(1)-множество таблиц для множества ситуаций, показанного на рис. 7.36. Таблицу, построенную по $\mathscr{A}_i$, назовем T_i. Мы рассмотрим только построение таблицы T_2.

Множество $\mathscr{A}_2$ — это $\{[E \to T\cdot], [T \to T\cdot * F]\}$. Пусть $T_2 = \langle f, g\rangle$. Так как $\mathrm{FOLLOW}(E)=\{+,), e\}$, то $f(+)=f(\,)=f(e)=$ **свертка** 2.

[1]) Каноническая система множеств ситуаций строится по пополненной грамматике.

(Правила перенумерованы как обычно.) Так как $\mathrm{EFF}(*F\ \mathrm{FOLLOW}(T))=\{*\}$, то $f(*)=$ **перенос**. Для остальных аванцепочек $f(a)=f(\,)=$ **ошибка**.

$g(X)$ определено только для $X=*$. Из рис. 7.36 легко видеть, что $g(*)=T_7$. Все множество таблиц показано на рис. 7.37.

	Действие						Переход							
	a	+	*	(	)	e	E	T	F	a	+	*	(	)
T_0	S	X	X	S	X	X	T_1	T_2	T_3	T_4	X	X	T_5	X
T_1	X	S	X	X	X	A	X	X	X	X	T_6	X	X	X
T_2	X	2	S	X	2	2	X	X	X	X	X	T_7	X	X
T_3	X	4	4	X	4	4	X	X	X	X	X	X	X	X
T_4	X	6	6	X	6	6	X	X	X	X	X	X	X	X
T_5	S	X	X	S	X	X	T_8	T_2	T_3	T_4	X	X	T_5	X
T_6	S	X	X	S	X	X	X	T_9	T_3	T_4	X	X	T_5	X
T_7	S	X	X	S	X	X	X	X	T_{10}	T_4	X	X	T_5	X
T_8	X	S	X	X	S	X	X	X	X	X	T_6	X	X	T_{11}
T_9	X	1	S	X	1	1	X	X	X	X	X	T_7	X	X
T_{10}	X	3	3	X	3	3	X	X	X	X	X	X	X	X
T_{11}	X	5	5	X	5	5	X	X	X	X	X	X	X	X

Рис. 7.37. Множество SLR(1)-таблиц для G_0.

С точностью до имен таблиц и элементов φ это множество совпадает с множеством, приведенным на рис. 7.32. □

Докажем, что алгоритм 7.10 для SLR(k)-грамматики G всегда строит допустимое множество LR(k)-таблиц. На самом деле получаемое множество таблиц эквивалентно каноническому множеству LR(k)-таблиц для G.

Теорема 7.9. *Если G — SLR(k)-грамматика, то SLR(k)-множество таблиц $(\mathscr{T}, T_0)$, построенное алгоритмом 7.10, эквивалентно каноническому множеству $(\mathscr{T}_c, T_c)$ LR(k)-таблиц для G.*

Доказательство. Пусть $\mathscr{S}_k$ — каноническая система множеств LR(k)-ситуаций для G и $\mathscr{S}_0$ — система множеств LR(0)-ситуаций для G. Определим *ядро* множества ситуаций $\mathscr{A}$ как множество заключенных в скобки первых компонент ситуаций из

этого множества. Например, ядром множества $[A \to \alpha \cdot \beta, u]$ является $[A \to \alpha \cdot \beta]$[1]). Обозначим ядро множества $\mathcal{A}$ через $\mathrm{CORE}(\mathcal{A})$.

Все множества ситуаций в системе $\mathcal{S}_0$ различны, однако некоторые множества из $\mathcal{S}_k$ могут иметь одинаковые ядра. Легко показать, что $\mathcal{S}_0 = \{\mathrm{CORE}(\mathcal{A}) \mid \mathcal{A} \in \mathcal{S}_k\}$.

Определим на множестве таблиц функцию h, соответствующую функции CORE, определенной на множествах ситуаций. Положим $h(T) = T'$, если T — каноническая LR(k)-таблица, связанная с $\mathcal{A}$, а T' — SLR(k)-таблица, полученная с помощью алгоритма 7.10 из множества $\mathrm{CORE}(\mathcal{A})$. Легко проверить, что h коммутирует с GOTO, т. е. $\mathrm{GOTO}(h(T), X) = h(\mathrm{GOTO}(T, X))$.

Как и раньше, пусть

$$\mathcal{A}' = \{[A \to \alpha \cdot \beta,] \mid [A \to \alpha \cdot \beta, e] \in \mathcal{A} \text{ и } u \in \mathrm{FOLLOW}_k(A)\}$$

Пусть $\mathcal{S}'_0 = \{ l \mid \mathcal{A}' \in \mathcal{S}_0\}$. Мы уже знаем, что $(\mathcal{T}, T_0)$ совпадает с множеством LR(k)-таблиц, построенным по $\mathcal{S}'_0$ методом, описанным в разд. 5.2.5. Покажем, что $(\mathcal{T}, T_0)$ всегда можно получить, последовательно преобразуя каноническое множество $(\mathcal{T}_c, T_c)$ LR(k)-таблиц для G. Для этого нужно проделать следующее.

(1) Пусть $\mathcal{P}$ — множество отсрочки, состоящее из таких троек (T, u, i), что действие таблицы T на аванцепочке u есть **ошибка**, а действие $h(T)$ на u есть **свертка** i. Применяя к $\mathcal{P}$ и $(\mathcal{T}_c, T_c)$ алгоритм 7.8, получим множество таблиц $(\mathcal{T}'_c, T'_c)$.

(2) Воспользуемся теперь алгоритмом 7.7 для слияния всех таких пар таблиц T_1 и T_2, что $h(T_1) = h(T_2)$. Новым множеством таблиц и будет $(\mathcal{T}, T_0)$.

Пусть (T, u, i) — элемент множества $\mathcal{P}$. Чтобы доказать, что $\mathcal{P}$ удовлетворяет требованиям, предъявляемым к множеству отсрочки для $\mathcal{T}_c$, нужно показать, что если $T'' = \langle f'', g'' \rangle$ принадлежит $\mathrm{NEXT}(T, i)$, то $f''(u) =$ **ошибка**. Для этого предположим, что правило i есть $A \to \alpha$ и $T'' = \mathrm{GOTO}(T_0, \beta A)$ для некоторого активного префикса βA. Тогда $T = \mathrm{GOTO}(T_0, \beta\alpha)$.

Предположим противное, а именно, что $f''(u) \neq$ **ошибка**. Тогда найдется ситуация $[B \to \gamma \cdot \delta, v]$, допустимая для βA, причем $u \in \mathrm{EFF}(\delta v)$[2]). Каждое множество ситуаций, за исключением начального, содержит ситуацию, в которой слева от точки стоит хотя бы один символ (см. упр. 7.3.8). Начальное множество допустимо только для e. Поэтому без потери общности можно считать, что $\gamma = \gamma' A$ для некоторой цепочки γ'. Тогда ситуация $[B \to \gamma' \cdot A\delta, v]$ допустима для β. То же справедливо и в отношении ситуации $[A \to \cdot\alpha, u]$. Следовательно, ситуация $[A \to \alpha \cdot, u]$

[1]) Далее мы не будем каждый раз оговаривать различие между $[A \to \alpha \cdot \beta]$ и $[A \to \alpha \cdot \beta, e]$.

[2]) Заметим, что это утверждение справедливо независимо от того, является δ пустой цепочкой или нет.

допустима для $\beta\alpha$, и $f(u)$ вопреки предположению не есть **ошибка**. Отсюда заключаем, что $\mathcal{P}$ действительно является множеством отсрочки для $\mathcal{T}_c$.

Множество, полученное в результате применения к $\mathcal{T}_c$ алгоритма 7.8 с использованием множества отсрочки $\mathcal{P}$, обозначим через $\mathcal{T}_1$. Пусть T — таблица из $\mathcal{T}_c$, связанная с множеством ситуаций $\mathcal{A}$, и T' — соответствующая видоизмененная таблица из $\mathcal{T}_1$. Тогда T и T' отличаются только тем, что в то время как T сообщает об ошибке, T' может вызвать свертку. Это происходит, если $u \in \mathrm{FOLLOW}(A)$ и $v \neq u$ только для ситуации из $\mathcal{A}$ вида $[A \to \alpha \cdot, v]$. Утверждение вытекает из того, что по правилу (1б) алгоритма 7.10 таблица T' вызывает свертку при всех таких u, что $u \in \mathrm{FOLLOW}(A)$ и $[A \to \alpha \cdot] \in \mathrm{CORE}(\mathcal{A})$.

Определим теперь на $\mathcal{T}_1$ разбиение $\Pi = \{\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_r\}$, группирующее таблицы T_1 и T_2 в один блок тогда и только тогда, когда $h(T_1) = h(T_2)$. Так как h коммутирует с GOTO, разбиение Π совместимо. Слияние таблиц в каждом блоке, проведенное с помощью алгоритма 7.7, дает множество $\mathcal{T}$.

Так как алгоритмы 7.7 и 7.8 сохраняют эквивалентность множества таблиц, мы показали, что множество $\mathcal{T}$ LR(k)-таблиц для G эквивалентно каноническому множеству $\mathcal{T}_c$ LR(k)-таблиц для G. □

Прежде чем закончить изучение SLR-разбора, отметим, что методы оптимизации, описанные в разд. 7.3, применимы также и к SLR-таблицам. В упр. 7.4.16 выясняется, какие элементы ошибки в SLR(1)-множестве таблиц несущественны.

7.4.2. Распространение SLR-подхода на грамматики, не являющиеся SLR-грамматиками

Метод построения анализатора для грамматики по канонической системе множеств $\mathcal{S}_0$ LR(0)-ситуаций обладает двумя значительными преимуществами. Во-первых, для данной грамматики объем вычислений, необходимых для построения $\mathcal{S}_0$, вообще говоря, гораздо меньше объема работы по построению системы $\mathcal{S}_1$ множеств LR(1)-ситуаций. Во-вторых, число множеств LR(0)-ситуаций обычно гораздо меньше числа множеств LR(1)-ситуаций.

Правда, возникает вопрос: что делать, если грамматика такова, что множества FOLLOW недостаточно для разрешения конфликта, возникающего между действиями при разборе из-за противоречивости множеств LR(0)-ситуаций? Известно несколько методов, и мы их изучим, прежде чем закончить исследование LR(0)-подхода к построению анализаторов. Один из методов состоит в том, что для разрешения неоднозначности можно по-

пытаться использовать локальный контекст. Если этот метод не приведет к успеху, можно попробовать расщепить какое-то множество ситуаций на несколько подмножеств. В каждом из этих подмножеств для однозначного определения действия можно воспользоваться локальным контекстом. Проиллюстрируем на примерах оба метода.

Пример 7.26. Рассмотрим LR(1)-грамматику с правилами

(1) $S \to Aa$
(2) $S \to dAb$
(3) $S \to cb$
(4) $S \to dca$
(5) $A \to c$

Несмотря на то, что язык $L(G)$ состоит всего из четырех цепочек, грамматика G не является SLR(k)-грамматикой ни для какого $k \geqslant 0$. Каноническая система множеств LR(0)-ситуаций для G приведена на рис. 7.38. Вторые компоненты опущены и

$\mathcal{A}_0$: $S' \to \cdot S$
$S \to \cdot Aa \mid \cdot dAb \mid \cdot cb \mid \cdot dca$
$A \to \cdot c$
$\mathcal{A}_1$: $S' \to S\cdot$
$\mathcal{A}_2$: $S \to A\cdot a$
$\mathcal{A}_3$: $S \to d\cdot Ab \mid d\cdot ca$
$A \to \cdot c$
$\mathcal{A}_4$: $S \to c\cdot b$
$A \to c\cdot$
$\mathcal{A}_5$: $S \to Aa\cdot$
$\mathcal{A}_6$: $S \to dA\cdot b$
$\mathcal{A}_7$: $S \to dc\cdot a$
$A \to c\cdot$
$\mathcal{A}_8$: $S \to cb\cdot$
$\mathcal{A}_9$: $S \to dAb\cdot$
$\mathcal{A}_{10}$: $S \to dca\cdot$

Рис. 7.38. Каноническая система множеств LR(0)-ситуаций.

ситуаций $[A \to \alpha_1\cdot\beta_1]$, $[A \to \alpha_2\cdot\beta_2]$, . . ., $[A \to \alpha_n\cdot\beta_n]$ для краткости обозначены $A \to \alpha_1\cdot\beta_1 \mid \alpha_2\cdot\beta_2 \mid \cdots \mid \alpha_n\cdot\beta_n$. Здесь два противоречивых множества ситуаций: $\mathcal{A}_4$ и $\mathcal{A}_7$. Более того, поскольку FOLLOW(A) $= \{a, b\}$, алгоритм 7.10 не построит по $\mathcal{A}_4$ и $\mathcal{A}_7$ однозначные действия для аванцепочек b и a соответственно.

Тем не менее исследуем функцию GOTO, определенную на множествах ситуаций и изображенную графически на рис. 7.39[1]).

[1]) Заметим, что этот граф — ациклический, хотя в общем случае граф переходов содержит циклы.

Мы видим, что достичь $\mathcal{A}_4$ из $\mathcal{A}_0$ можно, только имея в магазине c. Если свернуть c в A, то по правилам грамматики полу-

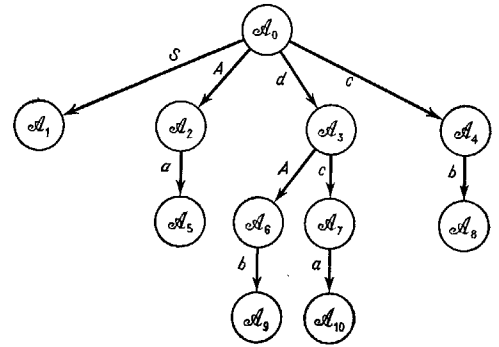


Рис. 7.39. Граф GOTO.

чим, что за A может идти только a. Поэтому таблица T_4, построенная по $\mathcal{A}_4$, содержит однозначную функцию действия:

	a	b	c	d
T_4:	свертка 5	перенос	ошибка	ошибка

Аналогично из графа GOTO видно, что достичь $\mathcal{A}_7$ из $\mathcal{A}_0$ можно, только имея в магазине dc. В таком контексте, если c свернуть к A, то за A может идти только b. Поэтому таблица T_7, построенная по $\mathcal{A}_7$, содержит функцию действия

	a	b	c	d
T_7:	перенос	свертка 5	ошибка	ошибка

Остальные LR(1)-таблицы для G можно получить непосредственно по алгоритму 7.10. □

Грамматика из примера 7.26 не является SLR-грамматикой. Однако для разрешения всех неоднозначностей при разборе мы смогли использовать в множестве LR(0)-ситуаций заглядывание вперед. Класс LR(k)-грамматик, для которых всегда можно таким

способом построить LR-анализаторы, называется классом LR(k)-*грамматик с заглядыванием* или, короче, классом LALR(k)-грамматик [1]) (более точное определение дано в упр. 7.4.11). LALR(k)-грамматики образуют наиболее широкий естественный подкласс LR(k)-грамматик, для которого заглядывание на k символов позволяет разрешить конфликты, возникающие при разборе, с помощью канонической системы $\mathscr{S}_0$ множеств LR(0)-ситуаций. Аванцепочки можно найти либо непосредственно по графу GOTO для $\mathscr{S}_0$, либо слив множества LR(k)-ситуаций с одинаковыми ядрами. LALR-грамматики включают в себя все SLR-грамматики, но не всякая LR-грамматика является LALR-грамматикой.

Приведем теперь пример, когда для получения однозначных действий разбора множество ситуаций „расщепляется".

Пример 7.27. Рассмотрим LR(1)-грамматику с правилами

$$
\begin{array}{ll}
(1) & S \to Aa \\
(2) & S \to dab \\
(3) & S \to Bb \\
(4) & S \to dBa \\
(5) & A \to c \\
(6) & B \to c
\end{array}
$$

Эта грамматика совершенно аналогична грамматике из примера 7.26, но она не является LALR-грамматикой. Каноническая система множеств LR(0)-ситуаций для пополненной грамматики

$$
\begin{array}{llll}
\mathcal{A}_0: & S' \to \cdot S & \mathcal{A}_5: & A \to c\cdot \\
 & S \to \cdot Aa & & B \to c\cdot \\
 & S \to \cdot dAb & \mathcal{A}_6: & S \to Aa\cdot \\
 & S \to \cdot Bb & \mathcal{A}_7: & S \to Bb\cdot \\
 & S \to \cdot dBa & \mathcal{A}_8: & S \to dA\cdot b \\
 & A \to \cdot c & \mathcal{A}_9: & S \to dB\cdot a \\
 & B \to \cdot c & \mathcal{A}_{10}: & S \to dAb\cdot \\
\mathcal{A}_1: & S' \to S\cdot & \mathcal{A}_{11}: & S \to dBA\cdot \\
\mathcal{A}_2: & S \to A\cdot a & & \\
\mathcal{A}_3: & S \to B\cdot b & & \\
\mathcal{A}_4: & S \to d\cdot Ab & & \\
 & S \to d\cdot Ba & & \\
 & A \to \cdot c & & \\
 & B \to \cdot c & &
\end{array}
$$

Рис. 7.40. Каноническая система множеств LR(0)-ситуаций.

приведена на рис. 7.40. Множество $\mathcal{A}_0$ противоречиво, потому что неизвестно, по какому правилу нужно свертывать: по $A \to c$ или по $B \to c$. Так как FOLLOW(A) = FOLLOW(B) = $\{a, b\}$, нам

[1]) От англ. LookAhead LR(k).— *Прим. перев.*

не удастся устранить неоднозначность, используя в качестве аванцепочек эти множества. Следовательно, G не является SLR(1)-грамматикой.

Анализ правил грамматики показывает, что если в магазине содержится только c и очередной входной символ есть a, то для свертки символа c нужно применить правило $A \to c$. Если очередным входным символом служит b, то нужно воспользоваться правилом $B \to c$. Но если в магазине записана цепочка dc и очередной входной символ есть a, то для свертки c нужно применить правило $B \to c$, а если очередным входным символом служит b — правило $A \to c$.

Функция GOTO для множества ситуаций приведена на рис. 7.41. К сожалению, $\mathcal{A}_5$ достижимо из $\mathcal{A}_0$ и тогда, когда

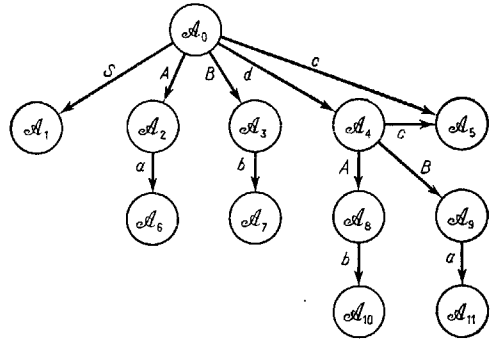


Рис. 7.41. Граф GOTO.

в магазине содержится c, и тогда, когда там записана цепочка dc. Поэтому $\mathcal{A}_5$ не уточняет, какая из этих двух цепочек записана в магазине, и, значит, G не является LALR(1)-грамматикой.

Тем не менее для G можно построить LR(1)-анализатор, заменив $\mathcal{A}_5$ такими двумя одинаковыми множествами $\mathcal{A}_5'$ и $\mathcal{A}_5''$, что $\mathcal{A}_5'$ достижимо только из $\mathcal{A}_4$, а $\mathcal{A}_5''$ — только непосредственно из $\mathcal{A}_0$. Эти новые множества ситуаций дают необходимую дополнительную информацию о том, что содержится в магазине.

По $\mathcal{A}_5'$ и $\mathcal{A}_5''$ можно построить такие таблицы с однозначными функциями действия:

	a	b	c	d
T_5':	свертка 6	свертка 5	ошибка	ошибка
T_5'':	свертка 5	свертка 6	ошибка	ошибка

Функции перехода в таблицах T_5' и T_5'' всегда принимают значения **ошибка**. □

7.4.3. Расщепление грамматики

В настоящем разделе мы изучим другой метод построения LR-анализаторов. Он не так прост в применении, как метод SLR, но работает в случаях, когда метод SLR не приводит к успеху. Мы будем расщеплять грамматику $G=(\mathrm{N}, \Sigma, P, S)$ на несколько грамматик-компонент, рассматривая некоторые нетерминальные символы как терминальные. Пусть $\mathrm{N}' \subseteq \mathrm{N}$ — такое множество „расщепляющих" нетерминалов. Для каждого A из N' можно найти *грамматику-компоненту* G_A *с начальным символом* A, воспользовавшись следующим алгоритмом.

Алгоритм 7.11. Расщепление грамматики.

Вход. КС-грамматика $G=(\mathrm{N}, \Sigma, P, S)$ и подмножество N' множества N.

Выход. Множество грамматик-компонент G_A для каждого $A \in \mathrm{N}'$.

Метод. Для каждого $A \in \mathrm{N}'$ построить G_A так:

(1) В правых частях всех правил из P заменить каждый нетерминал $B \in \mathrm{N}'$ на $\hat{B}$. Обозначим через $\hat{\mathrm{N}}$ множество $\{\hat{B} \mid B \in \mathrm{N}'\}$ и через $\hat{P}$ новое множество правил.

(2) Положим $G_A' = (\mathrm{N} - \mathrm{N}' \cup \{A\}, \Sigma \cup \hat{\mathrm{N}}, \hat{P}, A)$.

(3) Применить алгоритм 2.9 для исключения из G_A' бесполезных нетерминалов и правил. Получившаяся в результате грамматика и есть G_A. □

Мы займемся сейчас задачей построения LR(1)-анализаторов для всех грамматик-компонент и объединения этих анализаторов. Для различных компонент можно, вообще говоря, строить анализаторы разных типов. Например, для разбора выражений можно применить анализатор операторного предшествования, а все остальное разбирать с помощью LL(1)-анализатора. Представляет интерес вопрос о том, как можно распространить методы настоящего раздела на различные типы анализаторов.

Пример 7.28. Пусть G_0 — наша обычная грамматика и $\mathrm{N}' = \{E, T\}$ — множество расщепляющих нетерминалов. Тогда $\hat{P}$ состоит из правил

$$E \to \hat{E} + \hat{T} \mid \hat{T}$$
$$T \to \hat{T} * F \mid F$$
$$F \to (\hat{E}) \mid a$$

Таким образом, $G_E = (\{E\}, \{\hat{E}, \hat{T}, +\}, \{E \to \hat{E} + \hat{T} \mid \hat{T}\}, E)$ и $G_T = (\{T, F\}, \{\hat{T}, \hat{E}, (,), a, *\}, \{T \to \hat{T} * F \mid F, F \to (\hat{E}) \mid a\}, T)$. □

Опишем теперь метод построения LR(1)-анализаторов для некоторых больших грамматик. Этот метод состоит в том, что сначала данная грамматика расщепляется на ряд меньших грамматик. Если для каждой грамматики-компоненты удается найти систему непротиворечивых множеств LR(1)-ситуаций и если удовлетворяются некоторые условия, связывающие эти множества, то для исходной грамматики множество LR(1)-ситуаций строится путем объединения множеств ситуаций для грамматик-компонент. Основная идея метода такова: построение систем множеств LR(1)-ситуаций для нескольких меньших грамматик и последующее их объединение обычно требует значительно меньшей работы, чем построение канонической системы множеств LR(1)-ситуаций для одной большой грамматики.

В алгоритме расщепления грамматики нетерминал A рассматривается как начальный символ соответствующей грамматики, а FOLLOW(A) — как множество возможных аванцепочек для начального множества ситуаций, отвечающего подграмматике G_A. Основная цель — объединение некоторых множеств ситуаций, имеющих одинаковые ядра. Сходство этого алгоритма с SLR(1)-алгоритмом очевидно. В самом деле, как мы увидим, SLR(1)-алгоритм представляет собой алгоритм расщепления грамматики с $\mathrm{N}' = \mathrm{N}$.

В законченном виде метод выглядит следующим образом.

(1) Для данной грамматики $G=(\mathrm{N}, \Sigma, P, S)$ находим подходящее множество расщепляющих нетерминалов $\mathrm{N}' \subseteq \mathrm{N}$. Включаем S в N'. Это множество должно быть достаточно велико для того, чтобы грамматики-компоненты были небольшими и чтобы для каждой компоненты легко строилось множество LR(1)-таблиц. В то же время число компонент не должно быть слишком большим, чтобы при построении множества таблиц метод не оказался непригодным. (Последнее замечание относится только к грамматикам, не являющимся SLR-грамматиками. Если грамматика является SLR-грамматикой, то подойдет любой выбор множества N', а наименьшее множество таблиц будет получено при $\mathrm{N} = \mathrm{N}'$.)

(2) После выбора множества N' находим грамматики-компоненты по алгоритму 7.11.

(3) Применяя описанный ниже алгоритм 7.12, вычисляем множества LR(1)-ситуаций для каждой из грамматик-компонент.

(4) Воспользовавшись затем алгоритмом 7.13, объединяем множества-компоненты в систему $\mathscr{S}$ множеств ситуаций для исходной грамматики. Этот прием не всегда дает систему непроти-

воречивых множеств ситуаций для исходной грамматики. Однако, если система $\mathscr{S}$ непротиворечива, множество LR(1)-таблиц строится затем по $\mathscr{S}$ обычным способом.

Алгоритм 7.12. Построение множеств LR(1)-ситуаций для грамматик-компонент исходной грамматики.

Вход. Грамматика $G=(\mathrm{N}, \Sigma, P, S)$, подмножество N' множества N, содержащее S, и грамматики-компоненты G_A для всех $A \in \mathrm{N}'$

Выход. Множества LR(1)-ситуаций для всех грамматик-компонент.

Метод. Для удобства обозначений положим $\mathrm{N}'=\{S_1, S_2, \ldots, S_m\}$. Будем обозначать G_{S_i} как G_i.

Пусть $\mathcal{A}$ — множество LR(1)-ситуаций. Вычислим *замыкание* $\mathcal{A}'$ *множества* $\mathcal{A}$ *относительно* G_A. Метод вычисления напоминает алгоритм 5.8, но не совпадает с ним. $\mathcal{A}'$ определяется следующим образом:

(1) $\mathcal{A} \subseteq \mathcal{A}'$ (т. е. все элементы множества $\mathcal{A}$ принадлежат $\mathcal{A}'$).

(2) Если $[B \to \alpha \cdot C\beta,\ u] \in \mathcal{A}'$ и $C \to \gamma$ — правило из G_A, то $[C \to \cdot\gamma,\ v] \in \mathcal{A}'$ для всех $v \in \mathrm{FIRST}_1^G(\beta' u)$, где β' — это цепочка β, в которой все символы из $\hat{\mathrm{N}}$ заменены соответствующими символами из N.

Таким образом, в ситуациях все аванцепочки принадлежат Σ^*, а первые компоненты соответствуют правилам грамматики G_A.

Для каждой грамматики G_i построим систему $\mathscr{S}_i$ множеств LR(1)-ситуаций для G_i:

(1) Пусть $\mathcal{A}_0^i$ — замыкание (относительно G_i) множества $\{[S_i \to \cdot\alpha,\ a] \mid S_i \to \alpha$ — правило грамматики G_i и $a \in \mathrm{FOLLOW}_1^G(S_i)\}$. Положим $\mathscr{S}_i = \{\mathcal{A}_0^i\}$.

(2) Повторять шаг (3) до тех пор, пока к $\mathscr{S}_i$ не перестанут добавляться новые множества ситуаций.

(3) Пусть X принадлежит $\mathrm{N} \cup \Sigma \cup \hat{\mathrm{N}}$. Если $\mathcal{A} \in \mathscr{S}_i$, положим $\mathcal{A}' = \{[A \to \alpha X \cdot \beta,\ u] \mid [A \to \alpha \cdot X\beta,\ u] \in \mathcal{A}\}$. Добавим к $\mathscr{S}_i$ замыкание $\mathcal{A}''$ (относительно G_i) множества $\mathcal{A}'$. Итак, $\mathcal{A}''=\mathrm{GOTO}(\mathcal{A}, X)$. □

Отметим, что результат не изменится, если пополнить каждую грамматику-компоненту нулевым правилом вместо того, чтобы включать $\mathrm{FOLLOW}(A)$ в множество аванцепочек начального множества ситуаций.

Пример 7.29. Применим алгоритм 7.12 к грамматике G_0 с $\mathrm{N}'=\{E, T\}$. Находим, что $\mathrm{FOLLOW}(E)=\{+,), e\}$ и $\mathrm{FOLLOW}(T)=\{+, *,), e\}$. Таким образом, согласно шагу (1), $\mathcal{A}_0^E$ состоит

из ситуаций

$$[E \to \cdot\hat{E}+\hat{T},\ +/)/e]$$
$$[E \to \cdot\hat{T},\ +/)/e]$$

Аналогично $\mathcal{A}_0^T$ состоит из ситуаций

$$[T \to \cdot\hat{T} * F,\ +/*/)/e]$$
$$[T \to \cdot F,\ +/*/)/e]$$
$$[F \to \cdot(\hat{E}),\ +/*/)/e]$$
$$[F \to \cdot a,\ +/*/)/e]$$

Вся система множеств ситуаций, построенных для G_E, показана на рис. 7.42, а такая же система для G_T — на рис. 7.43.

$$\begin{array}{ll}
\mathcal{A}_0^E: & \left\{\begin{array}{l}[E \to \cdot\hat{E}+\hat{T},\ +/)/e]\\ [E \to \cdot\hat{T},\ +/)/e]\end{array}\right.\\
\mathcal{A}_1^E: & [E \to \hat{E}\cdot+\hat{T},\ +/)/e]\\
\mathcal{A}_2^E: & [E \to \hat{T}\cdot,\ +/)/e]\\
\mathcal{A}_3^E: & [E \to \hat{E}+\cdot\hat{T},\ +/)/e]\\
\mathcal{A}_4^E: & [E \to \hat{E}+\hat{T}\cdot,\ +/)/e]
\end{array}$$

Рис. 7.42. Система множеств ситуаций для G_E.

$$\begin{array}{ll}
\mathcal{A}_0^T: & \left\{\begin{array}{l}[T \to \cdot\hat{T} * F,\ +/*/)/e]\\ [T \to \cdot F,\ +/*/)/e]\\ [F \to \cdot(\hat{E}),\ +/*/)/e]\\ [F \to \cdot a,\ +/*/)/e]\end{array}\right.\\
\mathcal{A}_1^T: & [T \to \hat{T}\cdot * F,\ +/*/)/e]\\
\mathcal{A}_2^T: & [T \to F\cdot,\ +/*/)/e]\\
\mathcal{A}_3^T: & [F \to (\cdot\hat{E}),\ +/*/)/e]\\
\mathcal{A}_4^T: & [F \to a\cdot,\ +/*/)/e]\\
\mathcal{A}_5^T: & \left\{\begin{array}{l}[T \to \hat{T} * \cdot F,\ +/*/)/e]\\ [F \to \cdot(\hat{E}),\ +/*/)/e]\\ [F \to \cdot a,\ +/*/)/e]\end{array}\right.\\
\mathcal{A}_6^T: & [F \to (\hat{E}\cdot),\ +/*/)/e]\\
\mathcal{A}_7^T: & [T \to \hat{T} * F\cdot,\ +/*/)/e]\\
\mathcal{A}_8^T: & [F \to (\hat{E})\cdot,\ +/*/)/e]
\end{array}$$

Рис. 7.43. Система множеств ситуаций для G_T.

Заметим, что когда $\mathcal{A}_3^E$ строится по $\mathcal{A}_1^E$, символ $\hat{T}$, например, является терминалом, и операция замыкания не дает новых ситуаций. □

Приведем теперь алгоритм, который при определенных условиях строит по множествам ситуаций, образованным для грам-

матик-компонент с помощью алгоритма 7.12, множества LR(1)-ситуаций для исходной грамматики.

Алгоритм 7.13. Построение множества LR(1)-таблиц по множествам LR(1)-ситуаций для грамматик-компонент.

Вход. КС-грамматика $G=(N, \Sigma, P, S)$, расщепляющее множество нетерминалов $N'=\{S_1, S_2, \ldots, S_m\}$ и система $\{\mathcal{A}_0^i, \mathcal{A}_1^i, \ldots, \mathcal{A}_{n_i}^i\}$ множеств LR(1)-ситуаций для всех грамматик-компонент G_i.

Выход. Допустимое множество LR(1)-таблиц для G или сообщение о том, что данные множества ситуаций не позволяют получить допустимое множество таблиц.

Метод.

(1) В первых компонентах всех ситуаций заменить символы вида $\hat{S}_i$ на S_i. Все S_i принадлежат N'. Для измененных таким образом множеств ситуаций сохраним прежние обозначения.

(2) Пусть $\mathcal{I}_0=\{[S_1' \to \cdot S_1, e]\}$. Применить к $\mathcal{I}_0$ операцию пополнения и обозначить новое множество также через $\mathcal{I}_0$. В объединенной системе множеств ситуаций $\mathcal{I}_0$ будет „начальным" множеством.

Операция пополнения. Если множество $\mathcal{A}$ содержит ситуацию, у которой первая компонента имеет вид $A \to \alpha \cdot B\beta$ и $B \Rightarrow_G^* S_j\gamma$ для некоторых $S_j \in N'$ и $\gamma \in (N \cup \Sigma)^*$, то к $\mathcal{A}$ добавить $\mathcal{A}_0^j$. Повторять эту процедуру до тех пор, пока к $\mathcal{A}$ будут добавляться новые множества ситуаций.

(3) Построим систему $\mathcal{S}$ множеств ситуаций, достижимых из $\mathcal{I}_0$. Сначала $\mathcal{S}=\{\mathcal{I}_0\}$. Затем повторяем шаг (4) до тех пор, пока к $\mathcal{S}$ не перестанут добавляться новые множества ситуаций.

(4) Пусть $\mathcal{I} \in \mathcal{S}$. Можно представить $\mathcal{I}$ в виде $\mathcal{A} \cup \mathcal{A}_{i_1}^{j_1} \cup \mathcal{A}_{i_2}^{j_2} \cup \ldots \cup \mathcal{A}_{i_n}^{j_n}$, где $\mathcal{A}$ — либо пустое множество, либо одно из множеств $\{[S_1' \to \cdot S_1, e]\}$ или $\{[S_1' \to S_1 \cdot, e]\}$. Для всех X из $N \cup \Sigma$ положим $\mathcal{A}'=\mathrm{GOTO}(\mathcal{A}, X)$ и $\mathcal{A}_{h_n}^{j_n}=\mathrm{GOTO}(\mathcal{A}_{i_n}^{j_n}, X)$[1]. Обозначим через $\mathcal{I}'$ объединение $\mathcal{A}'$ и таких $\mathcal{A}_{h_n}^{j_n}$. Применим операцию пополнения к $\mathcal{I}'$ и обозначим новое множество также через $\mathcal{I}'$. Пусть KGOTO[2] — такая функция, что $\mathrm{KGOTO}(\mathcal{I}, X)=\mathcal{I}'$, если $\mathcal{I}$, X и $\mathcal{I}'$ связаны описанным выше образом. Добавим множество $\mathcal{I}'$ к системе $\mathcal{S}$, если его там еще нет. Будем повторять этот процесс до тех пор, пока какие-то $\mathcal{I} \in \mathcal{S}$ и $X \in N \cup \Sigma$ позволяют добавить к $\mathcal{S}$ новое множество $\mathrm{KGOTO}(\mathcal{I}, X)$.

(5) Когда к $\mathcal{S}$ перестанут добавляться новые множества ситуаций, построим по $\mathcal{S}$ множество LR(1)-таблиц, пользуясь методами разд. 5.2.5. Если таблица $T=\langle f, g\rangle$ была построена по множеству $\mathcal{I}$, то $g(X)=\mathrm{KGOTO}(\mathcal{I}, X)$. Если хотя бы одно из множеств ситуаций порождает конфликты действий при разборе, сообщаем об ошибке. □

[1]) Здесь имеется в виду функция GOTO для G_{j_n}. Если X — расщепляющий нетерминал, то вместо X берем $\hat{X}$.

[2]) Буква K обязана своим появлением А. Дж. Кореньяку, автору описываемого метода.

Пример 7.30. Применим алгоритм 7.13 к множествам ситуаций, приведенным на рис. 7.42 и 7.43. Результат выполнения шага (1) очевиден. В процессе выполнения шага (2) сначала образуется множество $\mathcal{I}_0=\{[E' \to \cdot E, e]\}$, а после применения операции пополнения — множество $\mathcal{I}_0=\{[E' \to \cdot E, e]\} \cup \mathcal{A}_0^E \cup \mathcal{A}_0^T$.

В начале шага (3) $\mathcal{S}=\{\mathcal{I}_0\}$. Переходя к шагу (4), сначала вычисляем

$$\mathcal{I}_1=\mathrm{GOTO}(\mathcal{I}_0, E)=\{[E' \to E \cdot, e]\} \cup \mathcal{A}_1^E$$

Другими словами, $\mathrm{GOTO}(\{[E' \to \cdot E, e]\}, E)=\{[E' \to E \cdot, e]\}$ и $\mathrm{GOTO}(\mathcal{A}_0^E, E)=\mathcal{A}_1^E$. Множество $\mathrm{GOTO}(\mathcal{A}_0^T, E)$ пусто. Операция пополнения не расширяет $\mathcal{I}_1$. Затем вычисляем $\mathcal{I}_2=\mathrm{GOTO}(\mathcal{I}_0, T)=\mathcal{A}_2^E \cup \mathcal{A}_1^T$. Операция пополнения не расширяет $\mathcal{I}_2$. Продолжая в том же духе, получаем систему множеств ситуаций для $\mathcal{S}$:

$$\begin{aligned}
\mathcal{I}_0 &= \{[E' \to \cdot E, e]\} \cup \mathcal{A}_0^E \cup \mathcal{A}_0^T \\
\mathcal{I}_1 &= \{[E' \to E \cdot, e]\} \cup \mathcal{A}_1^E \\
\mathcal{I}_2 &= \mathcal{A}_2^E \cup \mathcal{A}_1^T \\
\mathcal{I}_3 &= \mathcal{A}_2^T \\
\mathcal{I}_4 &= \mathcal{A}_4^T \\
\mathcal{I}_5 &= \mathcal{A}_0^E \cup \mathcal{A}_0^T \cup \mathcal{A}_3^T \\
\mathcal{I}_6 &= \mathcal{A}_3^E \cup \mathcal{A}_0^T \\
\mathcal{I}_7 &= \mathcal{A}_5^T \\
\mathcal{I}_8 &= \mathcal{A}_1^E \cup \mathcal{A}_6^T \\
\mathcal{I}_9 &= \mathcal{A}_4^E \cup \mathcal{A}_1^T \\
\mathcal{I}_{10} &= \mathcal{A}_7^T \\
\mathcal{I}_{11} &= \mathcal{A}_8^T
\end{aligned}$$

Все множества ситуаций из $\mathcal{S}$ непротиворечивы, и, значит, по $\mathcal{S}$ можно построить множество LR(1)-таблиц, приведенное на рис. 7.44. Таблица T_i соответствует $\mathcal{I}_i$.

Это множество таблиц совпало с множеством на рис. 7.37. Далее мы увидим, что это произошло не случайно. □

Покажем теперь, что такой подход позволяет получить множество LR(1)-таблиц, эквивалентное каноническому множеству

LR(1)-таблиц. Начнем с того, что охарактеризуем объединенную систему множеств ситуаций, построенную на шаге (4) алгоритма 7.13.

	Действие						*Переход*							
	a	+	*	(	)	e	E	T	F	a	+	*	(	)
T_0	S	X	X	S	X	X	T_1	T_2	T_3	T_4	X	X	T_5	X
T_1	X	S	X	X	X	A	X	X	X	X	T_6	X	X	X
T_2	X	2	S	X	2	2	X	X	X	X	X	T_7	X	X
T_3	X	4	4	X	4	4	X	X	X	X	X	X	X	X
T_4	X	6	6	X	6	6	X	X	X	X	X	X	X	X
T_5	S	X	X	S	X	X	T_8	T_2	T_3	T_4	X	X	T_5	X
T_6	S	X	X	S	X	X	X	T_9	T_3	T_4	X	X	T_5	X
T_7	S	X	X	S	X	X	X	X	T_{10}	T_4	X	X	T_5	X
T_8	X	S	X	X	S	X	X	X	X	X	T_6	X	X	T_{11}
T_9	X	1	S	X	1	1	X	X	X	X	X	T_7	X	X
T_{10}	X	3	3	X	3	3	X	X	X	X	X	X	X	X
T_{11}	X	5	5	X	5	5	X	X	X	X	X	X	X	X

Рис. 7.44. Множество таблиц для G_0, построенное алгоритмом 7.13.

Определение. Пусть KGOTO — функция, определенная на шаге (4) алгоритма 7.13. Расширим ее область определения, очевидным образом включив в нее цепочки символов, т. е.

(1) $\mathrm{KGOTO}(\mathcal{I}, e) = \mathcal{I}$,

(2) $\mathrm{KGOTO}(\mathcal{I}, \alpha X) = \mathrm{KGOTO}(\mathrm{KGOTO}(\mathcal{I}, \alpha), X)$.

Пусть $G = (\mathrm{N}, \Sigma, P, S)$ — КС-грамматика и N' — расщепляющее множество. Будем называть ситуацию $[A \to \alpha \cdot \beta, a]$ *квазидопустимой* для цепочки γ, если она допустима (в смысле разд. 5.2.3) или в пополненной грамматике есть такой вывод $S' \Rightarrow_r^* \delta_1 Bx \Rightarrow_r^* \delta_1\delta_2 Ax \Rightarrow_r \delta_1\delta_2\alpha\beta$, что

(1) $\delta_1\delta_2\alpha = \gamma$,

(2) $B \in \mathrm{N}'$,

(3) $a \in \mathrm{FOLLOW}(B)$.

Отметим, что если ситуация $[A \to \alpha \cdot \beta, a]$ квазидопустима для γ, то существует такая аванцепочка b, что ситуация $[A \to \alpha \cdot \beta, b]$ допустима для γ (b может совпасть с a).

Лемма 7.3. *Пусть* $G = (\mathrm{N}, \Sigma, P, S)$ — *КС-грамматика, упоминавшаяся выше, и* $\mathrm{KGOTO}(\mathcal{I}_0, \gamma) = \mathcal{I}$ *для некоторой цепочки* γ *из* $(\mathrm{N} \cup \Sigma)^*$. *Тогда* $\mathcal{I}$ — *множество квазидопустимых ситуаций для* γ.

Доказательство. Докажем лемму индукцией по $|\gamma|$. Базис, $\gamma = e$, опустим, так как при проведении шага индукции он станет очевидным.

Пусть $\gamma = \gamma' X$ и $\mathcal{K}$ — множество квазидопустимых ситуаций для γ', равное $\mathrm{KGOTO}(\mathcal{I}_0, \gamma')$. Пусть $\mathcal{K} = \mathcal{I}_{i_1}^{j_1} \cup \ldots \cup \mathcal{I}_{i_k}^{j_k}$. Случай, когда $[S' \to \cdot S, e]$ или $[S' \to S \cdot e]$ принадлежит $\mathcal{K}$, проанализировать легко, поэтому подробности мы опустим. Предположим, что $[A \to \alpha \cdot \beta, a]$ принадлежит $\mathcal{I} = \mathrm{KGOTO}(\mathcal{I}_0, \gamma' X)$. Ситуация $[A \to \alpha \cdot \beta, a]$ могла попасть в $\mathcal{I}$ тремя путями.

Случай 1: Допустим, что $[A \to \alpha \cdot \beta, a] \in \mathrm{GOTO}(\mathcal{I}_{i_p}^{j_p}, X)$ для некоторых p, $\alpha = \alpha' X$ и $[A \to \alpha' \cdot X\beta, a] \in \mathcal{I}_{i_p}^{j_p}$. Тогда ситуация $[A \to \alpha' \cdot X\beta, a]$ квазидопустима для γ', а отсюда следует, что ситуация $[A \to \alpha \cdot \beta, a]$ квазидопустима для γ.

Случай 2: Допустим, что $[A \to \alpha \cdot \beta, a] \in \mathrm{GOTO}(\mathcal{I}_{i_p}^{j_p}, X)$ и $\alpha = e$. Тогда найдется ситуация $[B \to \delta_1 X \cdot C\delta_2, b]$, принадлежащая $\mathrm{GOTO}(\mathcal{I}_{i_p}^{j_p}, X)$, и вывод $C \Rightarrow_r^* Aw$, где $a \in \mathrm{FIRST}(w\delta_2 b)$. Следовательно, ситуация $[B \to \delta_1 \cdot XC\delta_2, b]$ квазидопустима для γ', а ситуация $[B \to \delta_1 X \cdot C\delta_2 b]$ квазидопустима для γ. Если первым символом цепочки w служит a или $w = e$ и a появляется в δ_2, то ситуация $[A \to \alpha \cdot \beta, a]$ допустима для γ. Аналогично, если ситуация $[B \to \delta_1 X \cdot C\delta_2, b]$ допустима для γ, то допустимой для γ будет и ситуация $[A \to \alpha \cdot \beta, a]$.

Поэтому предположим, что $w = e$, $\delta_2 \Rightarrow^* e$, $a = b$ и ситуация $[B \to \delta_1 X \cdot C\delta_2, b]$ квазидопустима для γ. Тогда существует вывод

$$S' \Rightarrow_r^* \delta_2 Dx \Rightarrow_r^* \delta_3\delta_4 Bx \Rightarrow_r \delta_3\delta_4\delta_1 XC\delta_2 x \Rightarrow_r^* \delta_3\delta_4\delta_1 XAx$$

где $\delta_3\delta_4\delta_1 X = \gamma$, $D \in \mathrm{N}'$ и $b \in \mathrm{FOLLOW}(D)$. Таким образом, поскольку $a = b$, ситуация $[A \to \alpha \cdot \beta, a]$ квазидопустима для γ.

Случай 3: Допустим, что $[A \to \alpha \cdot \beta, a]$ попадает в $\mathcal{I}$ на шаге (4) при выполнении операции пополнения. Тогда $\alpha = e$ и в $\mathrm{GOTO}(\mathcal{I}_{i_p}^{j_p}, X)$ должна быть такая ситуация $[B \to \delta_1 X \cdot C\delta_2, b]$, что $C \Rightarrow_r^* Dw_1 \Rightarrow_r^* Aw_2 w_1$, $D \in \mathrm{N}'$ и $a \in \mathrm{FIRST}(w_2 c)$ для некоторого $c \in \mathrm{FOLLOW}(D)$. Другими словами, $D = S_r$, а $[A \to \alpha \cdot \beta, a]$ попадает в $\mathcal{K}$ при присоединении множества $\mathcal{I}_0^r$. Далее все аналогично случаю 2.

Теперь докажем утверждение, обратное к сформулированному выше, а именно: если ситуация $[A \to \alpha \cdot \beta, a]$ квазидопустима

для γ, то она принадлежит $\mathcal{I}$. Мы не рассматриваем более простой случай, когда ситуация допустима для γ. Предположим поэтому, что ситуация $[A \to \alpha \cdot \beta,\ a]$ квазидопустима, но не допустима. Тогда существует вывод

$$S' \Rightarrow_r^* \delta_1 Bx \Rightarrow_r^* \delta_1\delta_2 Ax \Rightarrow_r \delta_1\delta_2\alpha\beta x$$

где $\gamma = \delta_1\delta_2\alpha$, $B \in N'$ и $a \in \mathrm{FOLLOW}(B)$. Если $\alpha \neq e$, то можно записать α в виде $\alpha' X$. Тогда ситуация $[A \to \alpha' \cdot X\beta,\ a]$ квазидопустима для γ' и, значит, принадлежит $\mathcal{K}$. Отсюда сразу следует, что $[A \to \alpha \cdot \beta,\ a] \in \mathcal{I}$.

Поэтому будем считать, что $\alpha = e$. Рассмотрим два случая, соответствующие $\delta_2 \neq e$ и $\delta_2 = e$.

Случай 1: $\delta_2 \neq e$. Тогда существуют выводы $B \Rightarrow_r^* \delta_3 C \Rightarrow_r \delta_3\delta_4 X\delta_5$ и $\delta_5 \Rightarrow_r^* A$. Значит, ситуация $[C \to \delta_4 \cdot X\delta_5,\ a]$ квазидопустима для γ' и потому принадлежит $\mathcal{K}$. Отсюда вытекает, что $[C \to \delta_4 X \cdot \delta_5,\ a] \in \mathcal{I}$. Так как $\delta_5 \Rightarrow_r^* A$, нетрудно показать, что либо при выполнении операции замыкания, либо при выполнении операции пополнения ситуация $[A \to \alpha \cdot \beta,\ a]$ попадает в $\mathcal{I}$.

Случай 2: $\delta_2 = e$. В этом случае существует вывод $S' \Rightarrow_r^* \delta_3 Cy \Rightarrow_r \delta_3\delta_4 X\delta_5 y$, где $\delta_5 y \Rightarrow_r^* Bx$. Тогда ситуация $[C \to \delta_4 \cdot X\delta_5,\ c]$, где $c = \mathrm{FIRST}(y)$, допустима для γ'. Следовательно, $[C \to \delta_4 X \cdot \delta_5,\ c] \in \mathcal{I}$. Так как $\delta_5 y \Rightarrow_r^* Bx$, то при выполнении операции пополнения в $\mathcal{I}$ включаются ситуации вида $[B \to \cdot\varepsilon,\ b]$, где $B \to \varepsilon$ — правило, а b принадлежит $\mathrm{FOLLOW}(B)$. Поскольку $B \Rightarrow_r^* A$, ситуация $[A \to \alpha \cdot \beta,\ a]$ добавляется к $\mathcal{I}$ либо при замыкании (в смысле определения, данного при описании алгоритма 7.12) множества, содержащего $[B \to \cdot\varepsilon,\ b]$, либо при последующем выполнении операции пополнения. □

Теорема 7.10. *Пусть* (N, Σ, P, S) — *КС-грамматика и* $(\mathcal{T}, T_0)$ — *множество* LR(1)-*таблиц для* G, *построенное алгоритмом* 7.13. *Пусть* $(\mathcal{T}_c, T_c)$ — *каноническое множество. Эти два множества таблиц эквивалентны.*

Доказательство. Из леммы 7.3 вытекает, что таблица из множества $\mathcal{T}$, связанная с цепочкой γ, совпадает в действиях с таблицей из $\mathcal{T}_c$, связанной с γ, везде, где в $\mathcal{T}_c$ стоит действие, отличное от **ошибка**. Поэтому, если два множества не эквивалентны, можно найти такую входную цепочку w, что при работе с $\mathcal{T}_c$ анализатор делает последовательность тактов $(T_c, w) \vdash^* (T_c X_1 T_1 \ldots X_m T_m,\ x)$ [1]), после чего сообщает об ошибке, а при работе с $\mathcal{T}$ он делает последовательность тактов

$$\begin{aligned}(T_0, w) &\vdash^* (T_0 X_1 T_1' \ldots X_m T_m',\ x)\\ &\vdash^* (T_0 Y_1 U_1 \ldots Y_n U_n,\ x)\\ &\vdash^* (T_0 Y_1 U_1 \ldots Y_n U_n aU,\ x')\end{aligned}$$

[1]) Для простоты мы опустили в этих конфигурациях выходные цепочки.

Предположим, что таблица U_n строится по множеству ситуаций $\mathcal{I}$. Тогда $\mathcal{I}$ содержит такую ситуацию $[A \to \alpha \cdot \beta,\ b]$, что $\beta \neq e$ и $a \in \mathrm{EFF}(\beta)$. Так как ситуация $[A \to \alpha \cdot \beta,\ e]$ квазидопустима для $Y_1 \ldots Y_n$, то по лемме 7.3 существует вывод $S' \Rightarrow_r^* \gamma Ay \Rightarrow_r \gamma\alpha\beta y$ для некоторого y, где $\gamma\alpha = Y_1 \ldots Y_n$. Так как есть вывод $Y_1 \ldots Y_n \Rightarrow_r^* X_1 \ldots X_m$, то $\mathcal{I}$ содержит ситуацию $[B \to \cdot\varepsilon,\ c]$, допустимую для $X_1 \ldots X_m$, где $a \in \mathrm{EFF}(\varepsilon c)$. (Случай $\varepsilon = e$ и $a = c$ не исключается. В самом деле, это будет в том случае, если последовательность тактов

$$(T_0 X_1 T_1' \ldots X_m T_m',\ x) \vdash^* (T_0 Y_1 U_1 \ldots Y_n U_n,\ x)$$

имеет ненулевую длину. Если длина равна нулю, то роль ситуации $[B \to \cdot\varepsilon,\ c]$ играет $[A \to \alpha \cdot \beta,\ b]$.)

Так как $a = \mathrm{FIRST}(x)$, предположение о том, что при работе с множеством $\mathcal{T}'$ анализатор сообщает об ошибке в конфигурации $(T_0 X_1 T_1' \ldots X_m T_m',\ x)$, неверно. Теорема доказана. □

Сравним алгоритм расщепления грамматики с SLR-алгоритмом. Алгоритм расщепления представляет собой обобщение SLR-метода в следующем смысле.

Теорема 7.11. *Пусть* $G = (N, \Sigma, P, S)$ — *КС-грамматика.* G *является* SLR(1)-*грамматикой тогда и только тогда, когда алгоритм* 7.13 *работает успешно с расщепляющим множеством* N. *В этом случае множества таблиц, порождаемые этими двумя методами, совпадают.*

Доказательство. Если $N' = N$, то ситуация $[A \to \alpha \cdot \beta,\ a]$ квазидопустима для γ тогда и только тогда, когда ситуация $[A \to \alpha \cdot \beta,\ b]$ допустима для некоторых b и a из $\mathrm{FOLLOW}(A)$. Это следует из того, что если $B \Rightarrow_r^* \delta C$, то $\mathrm{FOLLOW}(B) \subseteq \mathrm{FOLLOW}(C)$. Из леммы 7.3 вытекает, что SLR-множества ситуаций совпадают с множествами, построенными алгоритмом 7.13. □

Теорема 7.9 представляет собой, таким образом, следствие теорем 7.10 и 7.11.

В алгоритме 7.13 законченная процедура построения канонического LR(1)-анализатора применяется к каждой грамматике-компоненте. Поэтому интуитивно, если грамматика не является SLR-грамматикой, желательно собрать в одной из компонент все те особенности данной грамматики, из-за которых она не является SLR-грамматикой. Проиллюстрируем это рассуждение примером.

Пример 7.31. Рассмотрим LR(1)-грамматику

(1) $S \to Aa$
(2) $S \to dAb$
(3) $S \to cb$
(4) $S \to BB$
(5) $A \to c$
(6) $B \to Bc$
(7) $B \to b$

	Действие					Переход						
	a	b	c	d	e	S	A	B	a	b	c	d
T_0	X	S	S	S	X	T_1	T_2	T_3	X	T_4	T_5	T_6
T_1	X	X	X	X	A	X	X	X	X	X	X	X
T_2	S	X	X	X	X	X	X	X	T_7	X	X	X
T_3	X	S	S	X	X	X	X	T_8	X	T_4	T_9	X
T_4	X	7	7	X	7	X	X	X	X	X	X	X
T_5	5	S	X	X	X	X	X	X	X	T_{10}	X	X
T_6	X	X	S	X	X	X	T_{11}	X	X	X	T_{12}	X
T_7	X	X	X	X	1	X	X	X	X	X	X	X
T_8	X	X	S	X	4	X	X	X	X	X	T_9	X
T_9	X	6	6	X	6	X	X	X	X	X	X	X
T_{10}	X	X	X	X	3	X	X	X	X	X	X	X
T_{11}	X	S	X	X	X	X	X	X	X	T_{13}	X	X
T_{12}	X	5	X	X	X	X	X	X	X	X	X	X
T_{13}	X	X	X	X	2	X	X	X	X	X	X	X

Рис. 7.45. Множество LR(1)-таблиц.

Из-за правил (1)—(3) и (5) она не является SLR-грамматикой. Взяв в качестве расщепляющего множества $\{S, B\}$, объединим эти четыре правила в одну грамматику-компоненту, а затем применим к ней LR(1)-метод. Алгоритм 7.13 применительно к такому расщепляющему множеству выдаст множество LR(1)-таблиц, показанное на рис. 7.45. □

Наконец, отметим, что ни алгоритм 7.10, ни алгоритм 7.13 не используют в полной мере технику отсрочки в обнаружении ошибок и слияния таблиц. Например, каноническое множество SLR(1)-грамматики из упр. 7.3.1(а) состоит из 18 LR(1)-таблиц. Алгоритм 7.13 даст множество, содержащее не менее 14 LR(1)-таблиц, алгоритм 7.10—множество из 14 SLR(1)-таблиц. В то же время, разумно используя отсрочку в обнаружении ошибок и выполняя слияние таблиц, можно получить множество из 7 LR(1)-таблиц.

УПРАЖНЕНИЯ

***7.4.1.** Рассмотрите класс $\{G_1, G_2, \ldots\}$ LR(0)-грамматик, где G_n содержит правила

$$S \to A_i \qquad 1 \leqslant i \leqslant n$$
$$A_i \to a_j A_i \qquad 1 \leqslant i \neq j \leqslant n$$
$$A_i \to a_i B_i \mid b_i \qquad 1 \leqslant i \leqslant n$$
$$B_i \to a_j B_i \mid b_i \qquad 1 \leqslant i,\ j \leqslant n$$

Покажите, что число таблиц в каноническом множестве LR(0)-таблиц для G_n экспоненциально зависит от n.

7.4.2. Покажите, что все грамматики из упр. 7.3.1 являются SLR(1)-грамматиками.

7.4.3. Покажите, что каждая LR(0)-грамматика является SLR(0)-грамматикой.

***7.4.4.** Покажите, что каждая простая ССП-грамматика является SLR(1)-грамматикой.

7.4.5. Покажите, что грамматика из примера 7.26 не является SLR(k)-грамматикой ни для какого $k \geqslant 0$.

***7.4.6.** Покажите, что всякая LL(1)-грамматика является SLR(1)-грамматикой. Всякая ли LL(2)-грамматика является SLR(2)-грамматикой?

7.4.7. С помощью алгоритма 7.10 постройте анализаторы для всех грамматик из упр. 7.3.1.

7.4.8. Пусть $\mathscr{S}_c$—каноническая система множеств LR(k)-ситуаций для G. Пусть $\mathscr{S}_0$—система множеств LR(0)-ситуаций для G. Покажите, что $\mathscr{S}_c$ и $\mathscr{S}_0$ имеют одинаковые множества ядер. *Указание:* Положите $\mathcal{A} = \mathrm{GOTO}(\mathcal{A}_0, \alpha)$, где $\mathcal{A}_0$—начальное множество системы $\mathscr{S}_c$, и проведите индукцию по $|\alpha|$.

7.4.9. Покажите, что $\mathrm{CORE}(\mathrm{GOTO}(\mathcal{A}, \alpha)) = \mathrm{GOTO}(\mathrm{CORE}(\mathcal{A}), \alpha)$, где, как и ранее, $\mathcal{A} \in \mathscr{S}_c$.

Определение. Грамматика $G=(N, \Sigma, P, S)$ называется LR(k)-грамматикой *с заглядыванием* и обозначается LALR(k), если следующий алгоритм позволяет получить множество LR(k)-таблиц:

(1) Построить для G каноническую систему $\mathscr{S}_c$ множеств LR(k)-ситуаций.

(2) Для каждого $\mathcal{A} \in \mathscr{S}_c$ обозначим через $\mathcal{A}'$ объединение таких $\mathcal{B} \in \mathscr{S}_c$, что $\mathrm{CORE}(\mathcal{B})=\mathrm{CORE}(\mathcal{A})$.

(3) Пусть $\mathscr{S}$ — множество всех $\mathcal{A}'$, построенных на шаге (2). Множество LR(k)-таблиц строится по $\mathscr{S}$ обычным образом.

7.4.10. Покажите, что если G — SLR(k)-грамматика, то она является LALR(k)-грамматикой.

7.4.11. Покажите, что описанный выше алгоритм построения LALR-таблиц порождает множество таблиц, эквивалентное каноническому множеству.

7.4.12. (а) Покажите, что грамматика из примера 7.26 является LALR(1)-грамматикой.

(б) Покажите, что грамматика из примера 7.27 не является LALR(k)-грамматикой ни для какого k.

7.4.13. Пусть G — грамматика с правилами

$$S \to L=R \mid R$$
$$L \to * \; R \mid a$$
$$R \to L$$

Покажите, что G является LALR(1)-грамматикой и не является SLR(1)-грамматикой.

7.4.14. Покажите, что существуют LALR-грамматики, не являющиеся LL-грамматиками, и обратно.

***7.4.15.** Пусть G — LALR-грамматика, а $\mathscr{S}_0$ — каноническая система множеств LR(0)-ситуаций для G. Будем говорить, что $\mathscr{S}_0$ имеет *конфликт „перенос — свертка“*, если некоторое множество $\mathcal{A} \in \mathscr{S}_0$ содержит ситуации $[A \to \alpha \cdot]$ и $[B \to \beta \cdot a\gamma]$, где $a \in \mathrm{FOLLOW}(A)$. Покажите, что в случае такого конфликта LALR-анализатор всегда выполняет перенос.

***7.4.16.** Пусть $(\mathscr{T}, T_0)$ — множество SLR(1)-таблиц для SLR(1)-грамматики $G=(N, \Sigma, P, S)$. Покажите, что

(1) все элементы ошибки у функций переходов несущественны,

(2) все элементы ошибки, соответствующие входному символу a у функции действия таблицы T, существенны тогда и только тогда, когда удовлетворяется одно из следующих условий:

(а) T — начальная таблица T_0,

(б) в $\mathscr{T}$ есть такая таблица $T'=\langle f, g\rangle$, что $T=g(b)$ для некоторого $b \in \Sigma$,

(в) в $\mathscr{T}$ есть такая таблица $T'=\langle f, g\rangle$, что T принадлежит $\mathrm{NEXT}(T', i)$ и $f(a)=$ **свертка**.

***7.4.17.** Пусть $G=(N, \Sigma, P, S)$ — КС-грамматика. Покажите, что G является LR(k)-грамматикой тогда и только тогда, когда для каждого расщепляющего множества $N' \in N$ все грамматики-компоненты G_A являются LR(k)-грамматиками, $A \in N'$. (*Замечание:* G может не быть LR(k)-грамматикой, но все же иметь такое расщепляющее множество N', что для $A \in N'$ все G_A будут LR(k)-грамматиками.)

7.4.18. Выполните упр. 7.4.17 для случая LL(k)-грамматик.

7.4.19. Воспользуйтесь алгоритмами 7.12 и 7.13 для построения множества LR(1)-таблиц для G_0, взяв в качестве расщепляющего множества $\{E, T, F\}$. Сравните полученное множество таблиц с множеством на рис. 7.44.

7.4.20. При каких условиях алгоритмы 7.12 и 7.13 для всех расщепляющих множеств нетерминалов будут порождать для G одно и то же множество LR(1)-таблиц?

***7.4.21.** Предположим, что с помощью LALR(1)-алгоритма не удается построить множество LR(1)-таблиц для грамматики $G=(N, \Sigma, P, S)$ из-за того, что возникает множество ситуаций, содержащее такие ситуации $[A \to \alpha \cdot, a]$ и $[B \to \beta \cdot \gamma, b]$, что $\gamma \neq e$ и $a \in \mathrm{EFF}(\gamma b)$. Покажите, что если $N' \subseteq N$ — расщепляющее множество и $A \in N'$, то алгоритм 7.13 также не даст множества LR(1)-таблиц с однозначно определенными действиями разбора.

7.4.22. Попытайтесь с помощью алгоритмов 7.12 и 7.13 построить множество LR(1)-таблиц для грамматики из упр. 7.27, взяв в качестве расщепляющего множества $\{S, A\}$.

7.4.23. Примените алгоритмы 7.12 и 7.13 для построения множества LR(1)-таблиц для грамматики из упр. 7.3.8 (а), имеющей 7 таблиц.

***7.4.24.** Воспользуйтесь методами отсрочки в обнаружении ошибок и слияния таблиц для нахождения эквивалентного множества из 7 LR(1)-таблиц для грамматики из упр. 7.3.1 (а).

***7.4.25.** Приведите пример (1,1)-ОПК-грамматики, которая не является SLR(1)-грамматикой.

***7.4.26.** Покажите, что если применить алгоритм 7.9 к множеству SLR(1)-таблиц для грамматики, имеющей не более одного цепного правила для каждого нетерминала, то все свертки по цепным правилам будут исключены.

Проблемы для исследования

7.4.27. Найдите дополнительные способы построения небольших множеств LR(k)-таблиц для LR(k)-грамматик, не использу-

ющие преобразований, подробно описанных в разд. 7.3. Ваши методы не обязательно должны работать для всех LR(k)-грамматик, но они должны быть применимы хотя бы к грамматикам, представляющим практический интерес, например к тем, которые приведены в приложении к тому 1.

7.4.28. Когда LR(k)-анализатор достигает ошибочной конфигурации, на практике обычно вызывается программа нейтрализации ошибок, изменяющая входную цепочку и содержимое магазина так, чтобы можно было продолжить нормальный разбор. Один из методов преобразования ошибочной конфигурации LR(1)-анализатора состоит в том, что на входной ленте разыскивается один из определенных входных символов. После того как такой входной символ a найден, в магазине разыскивается таблица $T=\langle f, g\rangle$, построенная по множеству ситуаций $\mathcal{A}$, содержащему ситуацию вида $[A\to\cdot\alpha, a]$, $A\neq S$. Процедура нейтрализации ошибок состоит в удалении всех входных символов вплоть до a и стирании в магазине всех символов и таблиц, расположенных выше T. Затем в магазин помещается нетерминал A и вслед за ним таблица $g(A)$. Из упр. 7.3.8(в) следует, что $g(A)\neq$ **ошибка**. Суть такого метода нейтрализации ошибок в том, что символы грамматики, расположенные в магазине выше T, вместе с входными символами вплоть до a рассматриваются как порождение символа A. Оцените эмпирически или теоретически эффективность такой процедуры нейтрализации ошибок. Разумным критерием эффективности может служить вероятность правильной нейтрализации ошибок из множества „обычных" ошибок программиста.

7.4.29. При расщеплении грамматики грамматики-компоненты можно анализировать по-разному. Исследуйте методы объединения различных типов анализаторов для грамматик-компонент. В частности, можно ли разбирать одну компоненту снизу вверх, а другую сверху вниз?

Упражнения на программирование

7.4.30. Напишите программу, строящую по SLR(1)-грамматике множество SLR(1)-таблиц.

7.4.31. Напишите программу, которая в множестве LR(1)-таблиц находит все недостижимые элементы ошибки.

7.4.32. Напишите программу, строящую по LALR(1)-грамматике LALR(1)-анализатор.

7.4.33. Постройте SLR(1)-анализатор с нейтрализацией ошибок для одной из грамматик, приведенных в приложении к тому 1.

Замечания по литературе

Простые LR(k)-грамматики и LALR(k)-грамматики впервые изучались Де Ремером [1969, 1971]. Им же предложен метод построения для грамматики канонического множества LR(0)-ситуаций и использования аванцепочек для разрешения неоднозначностей при разборе. Прием расщепления грамматики применительно к LR-анализаторам впервые применил Кореньяк [1969].

Упр. 7.4.1 заимствовано у Эрли [1968]. Процедура нейтрализации ошибок, описанная в упр. 7.4.28, предложена Лейниусом [1970]. Упр. 7.4.26 взято из работы Ахо и Ульмана [1972 г].

7.5. АНАЛИЗИРУЮЩИЕ АВТОМАТЫ

Вместо того чтобы смотреть на LR-анализатор как на программу с LR-таблицами в качестве данных, мы будем считать в этом разделе, что LR-таблицы управляют анализатором. Такой взгляд на LR-разбор позволяет разработать другой подход к задаче упрощения LR-анализаторов.

Основная идея этого раздела заключается в том, что если LR-таблицы размещаются в управляющем устройстве анализатора, то каждую таблицу можно рассматривать как состояние автомата, реализующего LR-алгоритм разбора. Такой автомат можно рассматривать как использующий магазин конечный автомат с „побочным эффектом". Методом, аналогичным алгоритму 2.2, можно минимизировать число состояний этого автомата.

7.5.1. Канонический анализирующий автомат

В LR-алгоритме разбора решение о переносе или свертке есть результат просмотра k следующих входных символов и обращения к управляющей таблице, которой служит таблица, расположенная в верхушке магазина. Если выполняется свертка, то новая управляющая таблица определяется после исследования таблицы, содержавшейся в магазине под свертываемой основой.

Вполне можно считать, что сами таблицы являются частями программы, а управление программой в свою очередь представляет собой управляющую таблицу. Типичным примером может служить программа, написанная на некотором легко интерпретируемом языке, так что различие между множеством таблиц, управляющих программой разбора, и самой интерпретируемой программой несущественно.

Пусть G — LR(0)-грамматика и $(\mathcal{T}, T_0)$ — множество LR(0)-таблиц для G. По множеству таблиц построим для G анализирующий автомат, который имитирует поведение LR(0)-алгоритма разбора для G, работающего с множеством таблиц $\mathcal{T}$.

Отметим, что если $T=\langle f, g\rangle$ — LR(0)-таблица из $\mathcal{T}$, то $f(e)$ — это **перенос**, **свертка** или **допуск**. Поэтому можно называть таблицы таблицами переноса или свертки в зависимости от значе-

ния функции действия. Вначале для каждой таблицы есть одна программа. Эти программы интерпретируются как *состояния* анализирующего автомата для G.

В *состоянии переноса* $T = \langle f, g \rangle$ выполняются такие действия:

(1) Имя состояния T помещается в магазин.

(2) Следующий входной символ, скажем a, удаляется из входной цепочки, и управляющее устройство переходит в состояние $g(a)$.

В рассмотренных ранее вариантах LR-анализа входной символ a также помещался в магазин вслед за таблицей T. Но, как уже отмечалось, нет необходимости запоминать в магазине символы грамматики, поэтому вплоть до конца раздела никакие символы грамматики помещать в магазин мы не будем.

В *состоянии свертки* происходит следующее:

(1) Пусть $A \to \alpha$ — правило i, по которому нужно свернуть. Верхние $|\alpha| - 1$ символов удаляются (выталкиваются) из магазина[1]). (Если $\alpha = e$, то в верхушке магазина находится управляющее состояние.)

(2) Определяется имя состояния, находящегося теперь в верхушке магазина. Пусть это состояние $T = \langle f, g \rangle$. Управляющее устройство переходит в состояние $g(A)$, и выдается номер правила i.

Особо отметим случай, когда действие „свертка" на самом деле означает допуск. В этом случае весь процесс заканчивается и автомат переходит в (допускающее) заключительное состояние.

Легко показать, что алгоритм с определенными так состояниями работает с магазином в точности так же, как LR(k)-анализатор (не считая того, что здесь мы не записываем в магазин символы грамматики), если отождествить имена состояний с именами таблиц, из которых состояния получаются. Единственное исключение состоит в том, что LR(k)-алгоритм разбора помещает в верхушку магазина имя управляющей таблицы, а здесь имя этой таблицы не записывается в магазин, поскольку оно является именем таблицы, с которой работает устройство управления программой.

Определим теперь анализирующий автомат, непосредственно выполняющий эти действия. Для каждого состояния (т. е. таблицы) в смысле, определенном выше, существует состояние автомата. Входными символами для такого автомата служат терминалы грамматики и сами имена состояний. В состоянии переноса выполняются действия только над терминалами, а в состоянии свертки — только над именами состояний. В самом деле, переход из состояния T' в состояние T, в котором вызывается свертка по правилу $A \to \alpha$, должен существовать только тогда, когда T принадлежит GOTO(T', α).

Следует помнить, что такой автомат нечто большее, чем конечный автомат, в котором состояния имеют побочные эффекты, связанные с магазином. Другими словами, каждый раз, когда возникает переход в новое состояние, с магазином что-то происходит — свойство, которого лишена модель системы, представляющая собой конечный автомат. Тем не менее можно уменьшить число состояний анализирующего автомата, применяя алгоритм, аналогичный алгоритму 2.2. Отличие в этом случае состоит в том, что мы должны быть уверены, что, если два состояния попадают в один и тот же класс эквивалентности, все их последующие побочные эффекты совпадут. Дадим теперь формальное определение анализирующего автомата.

Определение. Пусть $G = (N, \Sigma, P, S)$ — LR(0)-грамматика и $(\mathscr{T}, T_0)$ — ее множество LR(0)-таблиц. Определим не полностью определенный автомат M, называемый *каноническим анализирующим автоматом*, как пятерку $(\mathscr{T}, \Sigma \cup \mathscr{T} \cup \{\$\}, \delta, T_0, \{T_1\})$, где

(1) $\mathscr{T}$ — множество состояний,

(2) $\Sigma \cup \mathscr{T}$ — множество возможных входных символов (символы из Σ находятся на входной ленте, а символы из $\mathscr{T}$ — в магазине, поэтому $\mathscr{T}$ — не только множество состояний, но и подмножество входов анализирующего автомата),

(3) δ — отображение множества $\mathscr{T} \times (\Sigma \cup \mathscr{T})$ в $\mathscr{T}$, определяемое так:

(а) если $T \in \mathscr{T}$ — состояние переноса, то $\delta(T, a) = \mathrm{GOTO}(T, a)$ для всех $a \in \Sigma$,

(б) если $T \in \mathscr{T}$ — состояние свертки, вызывающее свертку по правилу $A \to \alpha$, и $T' \in \mathrm{GOTO}^{-1}(T, \alpha)$ (т. е. $T \in \mathrm{GOTO}(T', \alpha)$), то $\delta(T, T') = \mathrm{GOTO}(T', A)$,

(в) в остальных случаях $\delta(T, X)$ не определено.

Канонический анализирующий автомат является конечным преобразователем с побочными эффектами, связанными с магазином. Его поведение можно описать в терминах конфигураций, представляющих собой четверки вида (α, T, w, π), где

(1) α — содержимое магазина (верхний символ расположен справа),

(2) T — управляющее состояние,

[1]) Мы убираем из магазина $|\alpha| - 1$, а не $|\alpha|$ символов потому, что таблица, соответствующая самому правому символу цепочки α, находится в управляющем устройстве, а не была помещена в магазин.

(3) w — непросмотреиная часть входной цепочки,
(4) π — порожденная к этому моменту выходная цепочка.

Шаги автомата можно изобразить с помощью отношения $\vdash$, заданного на множестве конфигураций. Если T — состояние переноса и $\delta(T, a) = T'$, будем писать $(\alpha, T, aw, \pi) \vdash (\alpha T, T', w, \pi)$. Если T вызывает свертку по правилу $A \to \gamma$ с номером i и $\delta(T, T') = T''$, то пишем $(\alpha T'\beta, T, w, \pi) \vdash (\alpha T', T'', w, \pi i)$ для всех β длины $|\gamma| - 1$. Если $|\gamma| = 0$, то $(\alpha, T, w, \pi) \vdash (\alpha T, T'', w, \pi i)$. В этом случае T и T' совпадают. Заметим, что если считать, что в верхушке магазина может находиться символ управляющего состояния, то получатся конфигурации обычного LR-алгоритма разбора.

Отношения $\vdash^i$, $\vdash^*$ и $\vdash^+$ определяются обычным образом. *Начальная* конфигурация — это конфигурация вида (e, T_0, w, e), а *допускающая* конфигурация имеет вид (T_0, T_1, e, π). Если $(e, T_0, w, e) \vdash^* (T_0, T_1, e, \pi)$, то π называется *разбором, порожденным автоматом M для цепочки w*.

Пример 7.32. Рассмотрим LR(0)-грамматику G с правилами

(1) $S \to aA$
(2) $S \to aB$
(3) $A \to bA$
(4) $A \to c$
(5) $B \to bB$
(6) $B \to d$

порождающую регулярное множество $ab^*(c+d)$. На рис. 7.46 представлено множество из десяти LR(0)-таблиц для G.

T_0, T_2 и T_5 — состояния переноса. Таким образом, у нас получился анализирующий автомат с правилами переноса

$$\delta(T_0, a) = T_2$$
$$\delta(T_2, b) = T_5$$
$$\delta(T_2, c) = T_6$$
$$\delta(T_2, d) = T_7$$
$$\delta(T_5, b) = T_5$$
$$\delta(T_5, c) = T_6$$
$$\delta(T_5, d) = T_7$$

Найдем правила перехода для состояний свертки T_3, T_4, T_6, T_7, T_8 и T_9. Таблица T_3 вызывает свертку по правилу $S \to aA$. Так как $\text{GOTO}^{-1}(T_3, aA) = \{T_0\}$ и $\text{GOTO}(T_0, S) = T_1$, то единственным правилом для T_3 будет $\delta(T_3, T_0) = T_1$. Таблица T_7

вызывает свертку по правилу $B \to d$. Так как $\text{GOTO}^{-1}(T_7, d) = \{T_2, T_5\}$, $\text{GOTO}(T_2, B) = T_4$ и $\text{GOTO}(T_5, B) = T_9$, то правилами для T_7 будут $\delta(T_7, T_2) = T_4$ и $\delta(T_7, T_5) = T_9$.

	Действие	*Переход*						
	e	S	A	B	a	b	c	d
T_0	S	T_1	X	X	T_2	X	X	X
T_1	A	X	X	X	X	X	X	X
T_2	S	X	T_3	T_4	X	T_5	T_6	T_7
T_3	1	X	X	X	X	X	X	X
T_4	2	X	X	X	X	X	X	X
T_5	S	X	T_8	T_9	X	T_5	T_6	T_7
T_6	4	X	X	X	X	X	X	X
T_7	6	X	X	X	X	X	X	X
T_8	3	X	X	X	X	X	X	X
T_9	5	X	X	X	X	X	X	X

Рис. 7.46. Множество LR(0)-таблиц.

Полностью правила свертки таковы:

$$\delta(T_3, T_0) = T_1$$
$$\delta(T_4, T_0) = T_1$$
$$\delta(T_6, T_2) = T_3 \qquad \delta(T_6, T_5) = T_8$$
$$\delta(T_7, T_2) = T_4 \qquad \delta(T_7, T_5) = T_9$$
$$\delta(T_8, T_2) = T_3 \qquad \delta(T_8, T_5) = T_8$$
$$\delta(T_9, T_2) = T_4 \qquad \delta(T_9, T_5) = T_9$$

Граф переходов анализирующего автомата изображен на рис. 7.47.

Если на вход канонического автомата подать цепочку abc, то он пройдет последовательность конфигураций

$$\begin{aligned}(e, T_0, abc, e) &\vdash (T_0, T_2, bc, e)\\ &\vdash (T_0T_2, T_5, c, e)\\ &\vdash (T_0T_2T_5, T_6, e, e)\\ &\vdash (T_0T_2T_5, T_8, e, 4)\\ &\vdash (T_0T_2, T_3, e, 43)\\ &\vdash (T_0, T_1, e, 431)\end{aligned}$$

Таким образом, для входной цепочки *abc* анализирующий автомат порождает разбор 431. □

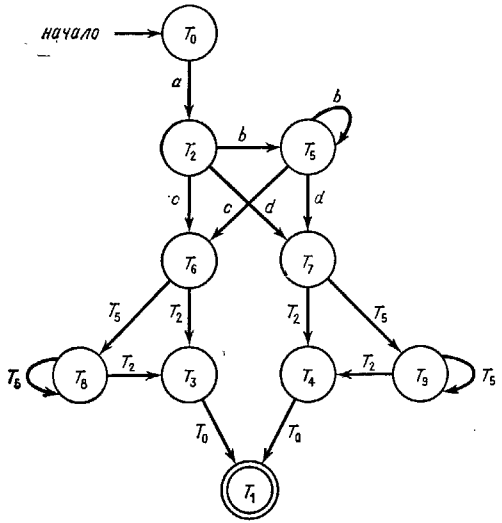


Рис. 7.47. Граф переходов канонического автомата.

7.5.2. Расщепление функций состояний

Использование анализирующего автомата для построения анализаторов имеет ряд преимуществ перед другими подходами. Например, часто удается расщепить функции некоторых состояний и связать их с двумя различными последовательными состояниями. Если состояние A расщепляется на два состояния A_1 и A_2, а B—на B_1 и B_2, то иногда можно слить, например, A_2 и B_2, в то время как слить A и B нельзя. Так как объем работы, выполненной в состояниях A_1 и A_2 (или B_1 и B_2), равен объему работы, выполненной в состоянии A (или B), то общее количество работы в результате расщепления не увеличивается. Вместе с тем, если удается слить состояния, то можно достичь экономии. В этом разделе мы исследуем, как расщепить функции некоторых состояний для того, чтобы затем попытаться совместить общие действия.

Каждое состояние свертки расщепим на два состояния: *состояние выталкивания* и следующее за ним *состояние опроса*. Предположим, что T—состояние свертки, вызывающее свертку по правилу $A \to \alpha$ с номером i. При расщеплении состояния T образуется состояние выталкивания, единственная функция которого—удалить верхние $|\alpha|-1$ символов из магазина. Если $\alpha = e$, то в состоянии выталкивания в верхушку магазина записывается имя T. Кроме того, в состоянии выталкивания порождается номер правила i.

После этого управляющее устройство переходит в состояние опроса, в котором выясняется имя состояния, находящегося в данный момент в верхушке магазина, например U, и происходит переход в состояние GOTO(U, A).

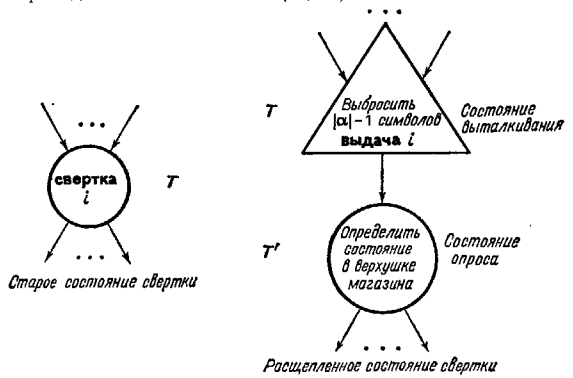


Рис. 7.48. Расщепление состояния свертки

В графе переходов состояние свертки T можно заменить состоянием выталкивания с тем же именем T и состоянием опроса, обозначенным T'. Все дуги, входящие в старое T, будут входить также и в новое T, а дуги, выходящие из старого T, теперь будут выходить из T'. Одна непомеченная дуга идет из нового T в T'. Это преобразование отражено на рис. 7.48, где правило i есть $A \to \alpha$.

Состояния переноса и состояние допуска не расщепляются.

Автомат, построенный по каноническому анализирующему автомату описанным способом, называется *расщепленным каноническим анализирующим автоматом*.

Пример 7.33. На рис. 7.49 показан расщепленный автомат, построенный по анализирующему автомату рис. 7.47. Состояния переноса изображены квадратами, состояния выталкивания—треугольниками, а состояния опроса и допуска—кружками.

Для того чтобы сравнить поведение этого расщепленного автомата с поведением автомата из примера 7.32, рассмотрим по-

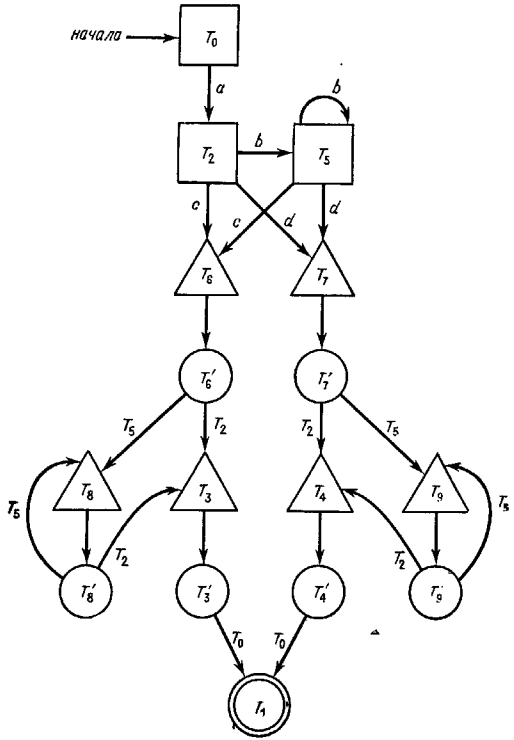


Рис. 7.49. Расщепленный канонический автомат.

следовательность тактов, пройденную расщепленным автоматом при разборе входной цепочки abc:

$$
\begin{aligned}
(e,\ T_0,\ abc,\ e) &\vdash (T_0,\ T_2,\ bc,\ e)\\
&\vdash (T_0T_2,\ T_5,\ c,\ e)\\
&\vdash (T_0T_2T_5,\ T_6,\ e,\ e)\\
&\vdash (T_0T_2T_5,\ T_6',\ e,\ 4)\\
&\vdash (T_0T_2T_5,\ T_8,\ e,\ 4)
\end{aligned}
$$

$$
\begin{aligned}
&\vdash (T_0T_2,\ T_3',\ e,\ 43)\\
&\vdash (T_0T_2,\ T_3,\ e,\ 43)\\
&\vdash (T_0,\ T_3',\ e,\ 431)\\
&\vdash (T_0,\ T_1,\ e,\ 431) \quad \square
\end{aligned}
$$

Пусть M_1 и M_2 — два анализирующих автомата для грамматики G. Будем называть M_1 и M_2 *эквивалентными*, если для любой входной цепочки w они порождают один и тот же разбор или оба выдают сигнал ошибки, прочитав одинаковое число входных символов. Таким образом, мы пользуемся тем же определением эквивалентности, что и в случае двух множеств LR(k)-таблиц.

Если канонический и расщепленный канонический автоматы работают параллельно, то легко видеть, что их магазины совпадают всякий раз, когда расщепленный автомат переходит в состояние переноса или опроса. Поэтому должно быть ясно, что эти два автомата эквивалентны.

Расщепленный анализирующий автомат можно упрощать двумя способами. Первый состоит в том, что некоторые состояния, действия которых не нужны, полностью исключаются. При втором способе сливаются неразличимые состояния. В первом случае исключаются некоторые состояния опроса.

Автомат, полученный из расщепленного канонического автомата после таких упрощений, будем называть *полуприведенным*.

Пример 7.34. Рассмотрим расщепленный анализирующий автомат, изображенный на рис. 7.49. Состояния T_3' и T_4' имеют только по одному переходу — этот переход помечен символом T_0. В результате наших преобразований эти состояния и переходы в T_0 будут исключены. Полученный полуприведенный автомат показан на рис. 7.50. □

Теорема 7.12. *Расщепленный канонический анализирующий автомат M_1 и его полуприведенный автомат M_2 эквивалентны.*

Доказательство. В результате действий, производимых в состоянии опроса, содержимое магазина не изменяется. Более того, если из состояния опроса T есть только один переход, то состояние, которым помечен этот переход, должно находиться в верхушке магазина всякий раз, когда M_1 переходит в состояние T. Это утверждение вытекает из определений канонического автомата и функции GOTO. Таким образом, первое преобразование не влияет на последовательности операций с магазином, входом или выходом, совершаемых автоматом. □

Займемся теперь задачей минимизации числа состояний полуприведенного автомата. Будем сливать состояния, побочные эф-

фекты которых (отличные от записи их имен в магазин) совпадают и переходы из которых по соответствующим дугам приводят к неразличимым состояниям. Алгоритм минимизации аналогичен по духу алгоритму 2.2, хотя и отличается от него ввиду того, что надо принять во внимание операции с магазином.

Определение. Пусть $M=(Q, \Sigma, \delta, q_0, \{q_1\})$ — полуприведенный анализирующий автомат, где q_0 — начальное состояние, а q_1 — состояние допуска. Заметим, что $Q \subseteq \Sigma$. Символом e будем „помечать" не помеченные еще переходы. Будем говорить, что p и q из Q *0-неразличимы*, и писать $p \equiv^0 q$, если граф переходов для M удовлетворяет одному из следующих условий (случай $p=q$ не исключается):

(1) p и q оба являются состояниями переноса;
(2) p и q оба являются состояниями опроса;
(3) p и q оба являются состояниями выталкивания, и в этих состояниях автомат удаляет из магазина одинаковое число символов и порождает один и тот же номер правила (другими словами, в состояниях p и q автомат свертывает по одному и тому же правилу);
(4) $p=q=q_1$ (заключительное состояние).

В остальных случаях p и q 0-различимы. В частности, состояния разных типов всегда 0-различимы.

Будем говорить, что p и q *k-неразличимы*, и писать $p \equiv^k q$, если они $(k-1)$-неразличимы и удовлетворяется одно из следующих условий:

(1) p и q являются состояниями переноса и
(а) для любого $a \in \Sigma \cup \{e\}$ дуги, помеченные символом a, либо выходят и из p и из q, либо не выходят ни из p, ни из q; если дуги, помеченные символом a, выходят из p и q и входят в p' и q' соответственно, то p' и q' $(k-1)$-неразличимы;
(б) нет состояния опроса с переходами, помеченными буквами p и q, в $(k-1)$-неразличимые состояния;
(2) p и q являются состояниями выталкивания и выходящие из них дуги ведут в $(k-1)$-неразличимые состояния;
(3) p и q являются состояниями опроса и для всех состояний s либо p и q оба имеют переход на s, либо оба его не имеют (в первом случае переходы происходят в $(k-1)$-неразличимые состояния).

В остальных случаях состояния p и q будем называть *k-различимыми*. Будем говорить, что p и q *неразличимы*, и писать $p \equiv q$, если они k-неразличимы для любого $k \geqslant 0$. В противном случае p и q будем называть *различимыми*.

Лемма 7.4. *Пусть $M=(Q, \Sigma, \delta, q_0, \{q_1\})$ — полуприведенный автомат. Тогда*
(1) *$\equiv^k$ для всех k есть отношение эквивалентности на Q,*
(2) *если $\equiv^k = \equiv^{k+1}$, то $\equiv^{k+1} = \equiv^{k+2} = \ldots$.*

Доказательство. Предлагаем в качестве упражнения (аналог леммы 2.11). □

Пример 7.35. Рассмотрим полуприведенный автомат, изображенный на рис. 7.50. Вспомнив множество LR(0)-таблиц, по

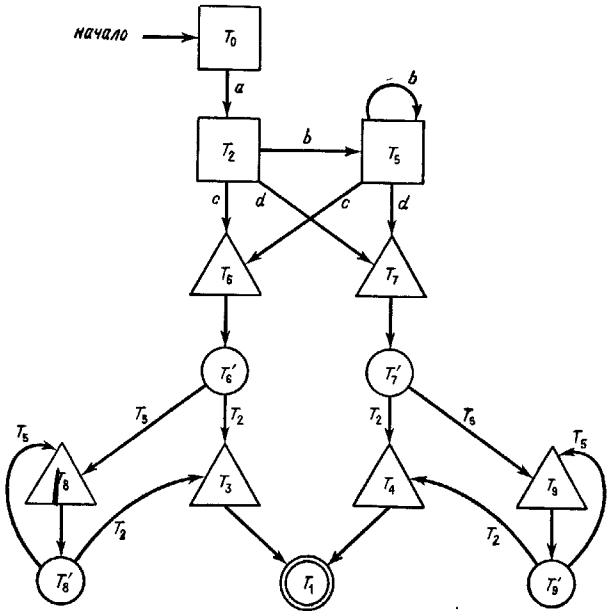


Рис. 7.50. Полуприведенный автомат.

которому был построен этот автомат (рис. 7.46), видим, что все шесть состояний выталкивания вызывают свертки по разным правилам, и значит, они 0-различимы. Состояния остальных типов по определению 0-неразличимы с состояниями того же типа, так что классы эквивалентности отношения $\equiv^0$ таковы: $\{T_0, T_2, T_5\}$, $\{T_1\}$, $\{T_3\}$, $\{T_4\}$, $\{T_6\}$, $\{T_7\}$, $\{T_8\}$, $\{T_9\}$, $\{T_6', T_7', T_8', T_9'\}$.

Для того чтобы найти $\equiv^1$, заметим, что состояния T_2 и T_5 1-различимы, так как из T'_6 в 0-различимые состояния T_3 и T_8 есть переходы с метками T_2 и T_5 соответственно. Далее, T_0 1-различимо с T_2 и T_5, поскольку из первого есть переход с меткой a, а из последних нет. T'_6 и T'_7 1-различимы, так как из них есть переходы в 0-различимые состояния, помеченные именем T_2. Аналогично пары состояний T'_6 и T'_9, T'_7 и T'_8, T'_8 и T'_9 1-различимы. Остальные пары, являющиеся 0-неразличимыми, также и 1-неразличимы. Таким образом, классы эквивалентности отношения $\equiv^1$, содержащие более одного элемента, — это $\{T'_6, T'_8\}$ и $\{T'_7, T'_9\}$. Находим далее, что $\equiv^2 = \equiv^1$. □

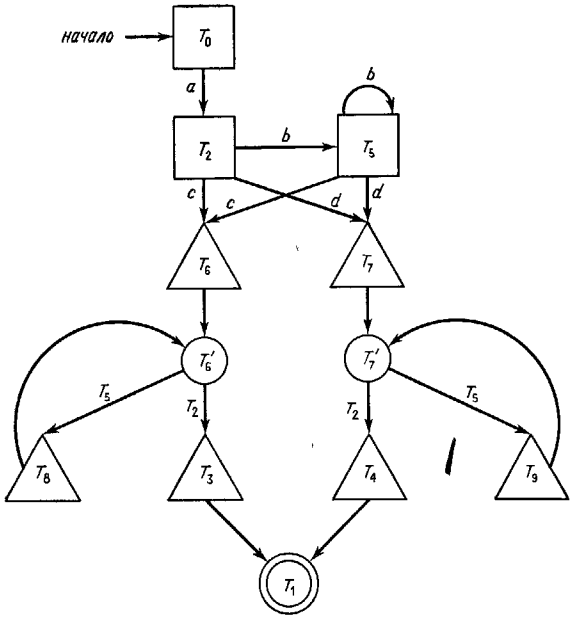


Рис. 7.51. Приведенный автомат.

Определение. Пусть $M_1 = (Q, \Sigma, \delta, q_0, \{q_1\})$ — полуприведенный автомат. Обозначим через Q' множество классов эквивалентности отношения $\equiv$. Класс эквивалентности, содержащий q, будем обозначать $[q]$. *Приведенным автоматом* для M_1 назовем

автомат $M_2 = (Q', \Sigma', \delta', [q_0], \{[q_1]\})$, где

(1) $\Sigma' = (\Sigma - Q) \cup Q'$,

(2) $\delta'([q], a) = [\delta(q, a)]$ для всех $q \in Q$ и $a \in \Sigma \cup \{e\} - Q$,

(3) $\delta'([q], [p]) = [\delta(q, p)]$ для всех q и $p \in Q$.

Пример 7.36. В примере 7.35 мы нашли, что $T'_6 \equiv T'_8$ и $T'_7 \equiv T'_9$. Приведенный автомат для автомата на рис. 7.50 изображен на рис. 7.51.

Состояния T'_6 и T'_7 выбраны в качестве представителей двух классов эквивалентности, содержащих более одного элемента. □

Из определения отношения $\equiv$ следует, что определение приведенного автомата корректно, т. е. правила (2) и (3) определения не зависят от выбора представителя класса эквивалентности.

Нетрудно также показать, что приведенный и полуприведенный автоматы эквивалентны. По существу, эти два автомата всегда будут совершать аналогичные последовательности тактов. Состояние приведенного автомата представляет класс эквивалентности, содержащий соответствующее состояние полуприведенного автомата. Формально это соответствие описано ниже.

Теорема 7.13. *Пусть M_1 — полуприведенный автомат и M_2 — соответствующий приведенный автомат. Тогда для всех $i \geqslant 0$ существует такая последовательность $T_1, \ldots, T_m$, T, что*

$$(e, T_0, w, e) \vdash^i_{M_1} (T_0 T_1 \ldots T_m, T, x, \pi)$$

тогда и только тогда, когда

$$(e, [T_0], w, e) \vdash^i_{M_2} ([T_0]\,[T_1] \ldots [T_m], [T], x, \pi)$$

где T_0 — начальное состояние автомата M_1, а $[u]$ обозначает класс эквивалентности, содержащий состояние u.

Доказательство. Элементарная индукция по i. □

Следствие. *M_1 и M_2 эквивалентны.* □

7.5.3. Обобщение на LR(k)-анализаторы

Канонический анализирующий автомат можно построить также и по множеству LR(k)-таблиц для LR(k)-грамматики с $k > 0$. Здесь мы исследуем случай $k = 1$. Анализирующий автомат в этом случае работает во многом аналогично каноническому анализирующему автомату для LR(0)-грамматики, не считая того, что о каждой таблице в отдельности нельзя сказать, является ли она таблицей переносов, сверток или допускающей таблицей.

Как и ранее, каждой таблице будет соответствовать состояние автомата. Находясь в конфигурации $(T_0 T_1 \ldots T_m, T, w, \pi)$, автомат действует так:

(1) Находит аванцепочку $a = \mathrm{FIRST}(w)$.

(2) Принимает решение о том, нужно ли выполнить перенос или свертку, т. е. если $T=\langle f, g\rangle$, то вычисляется $f(a)$.

(а) Если $f(a)=$ **перенос**, то автомат переходит в конфигурацию $(T_0T_1 \ldots T_mT,\ T',\ w',\ \pi)$, где $T'=g(a)$ и $aw'=w$.

(б) Если $f(a)=$ **свертка** i и правило i есть $A\to\alpha$, где $|\alpha|=r>0$, то автомат переходит в конфигурацию $(T_0T_1 \ldots T_{m-r+1},\ T',\ w,\ \pi i)$, где $T'=g'(A)$, если $T_{m-r+1}=\langle f', g'\rangle$. (Если правило имеет вид $A\to e$, то возникает конфигурация $(T_0T_1 \ldots T_mT,\ T',\ w,\ \pi i)$, где $T'=g(A)$, $T=\langle f, g\rangle$.)

(в) Если $f(a)=$ **допуск** или **ошибка**, то автомат останавливается и сообщает, что входная цепочка либо допущена, либо отвергнута.

Можно по-разному расщепить состояния автомата так, чтобы операции с магазином оказались выделенными, имея при этом в виду слить в дальнейшем общие операции. Здесь мы изучим следующую схему расщепления состояний.

Пусть T — состояние, соответствующее таблице $T=\langle f, g\rangle$. Расщепим его на состояния чтения, заталкивания, выталкивания и опроса:

(1) Образуем состояние *чтения* T, в котором считывается следующий входной символ. (Состояния чтения изображены на рисунках кружками.)

(2) Если $f(a)=$ **перенос**, образуем состояние *заталкивания* T^a и проведем дугу, помеченную символом a, из состояния чтения T в это состояние заталкивания. Если $g(a)=T'$, проведем непомеченную дугу из T^a в состояние чтения T'. Состояние заталкивания T^a имеет двойной эффект. Входной символ a удаляется из входной цепочки, и имя таблицы T заталкивается в магазин. (Состояния заталкивания изображены на рисунках квадратами.)

(3) Если $f(a)=$ **свертка** i, образуем состояние *выталкивания* T_1 и состояние *опроса* T_2. Из T в T_1 проведена дуга, помеченная символом a. Если правило i есть $A\to\alpha$, то в состоянии T_1 из магазина удаляются верхние $|\alpha|-1$ символов и порождается номер правила i. Если $\alpha=e$, то в состоянии T_1 в магазин помещается имя исходной таблицы T. (Состояния выталкивания изображены треугольниками.) Затем из T_1 в T_2 проведем непомеченную дугу. В состоянии T_2 выясняется, какой символ находится в данный момент в верхушке магазина. Если $\mathrm{GOTO}^{-1}(T, \alpha)$ содержит T', а $\mathrm{GOTO}(T', A)=T''$, проведем из T_2 в состояние чтения T'' дугу, помеченную символом T'. (Состояния опроса также изображаем кружками. Метки на дугах отличают эти состояния от состояний чтения.)

Таким образом, если $f(a)=$ **перенос** и $f(b)=$ **свертка** i, то состояние T будет выглядеть так, как показано на рис. 7.52.

(4) Состояние допуска не расщепляется.

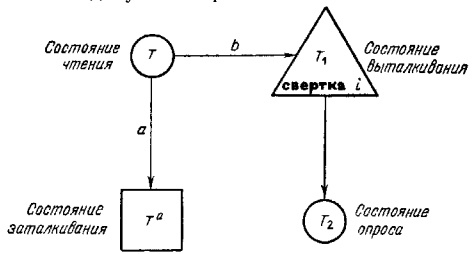


Рис. 7.52. Возможное представление состояния T.

Пример 7.37. Рассмотрим LR(1)-грамматику с правилами

(1) $S\to AB$
(2) $A\to aAb$
(3) $A\to e$
(4) $B\to bB$
(5) $B\to b$

Множество LR(1)-таблиц для G приведено на рис. 7.53.

	Действие			*Переход*				
	a	b	e	S	A	B	a	b
T_0	S	3	X	T_1	T_2	X	T_3	X
T_1	X	X	A	X	X	X	X	X
T_2	X	S	X	X	X	T_4	X	T_5
T_3	S	3	X	X	T_6	X	T_3	X
T_4	X	X	1	X	X	X	X	X
T_5	X	S	5	X	X	T_7	X	T_5
T_6	X	S	X	X	X	X	X	T_8
T_7	X	X	4	X	X	X	X	X
T_8	X	2	X	X	X	X	X	X

Рис. 7.53. Множество LR(1)-таблиц.

Анализирующий автомат, получающийся в результате применения описанной выше процедуры расщепления состояний, изображен на рис. 7.54. □

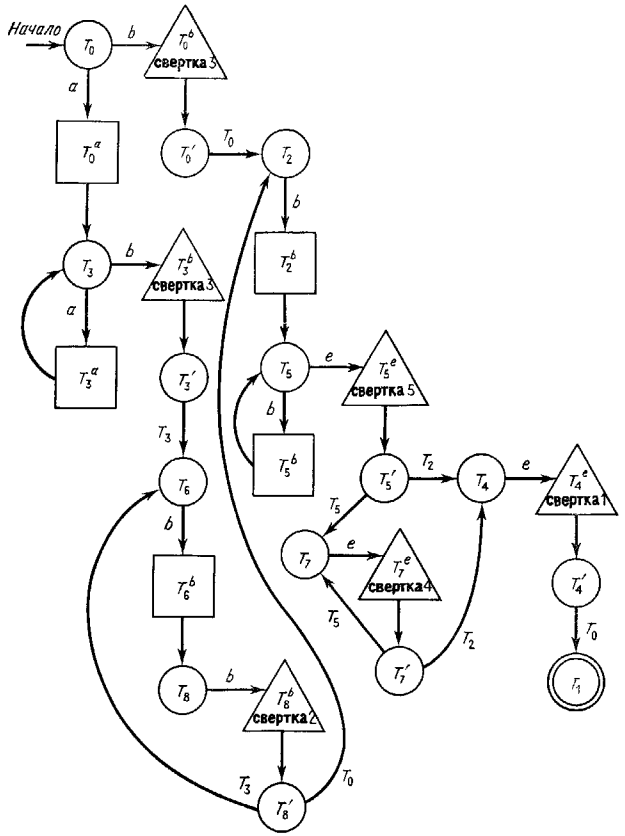


Рис. 7.54. Расщепленный автомат для LR(1)-грамматики.

Число состояний расщепленного автомата для LR(1)-грамматики можно уменьшить несколькими способами:

(1) Если из состояния опроса выходит только одна дуга, это состояние опроса можно выбросить. Это упрощение в точности аналогично соответствующему LR(0)-упрощению.

(2) Пусть T — такое состояние чтения, что в каждом пути, ведущем в T, последняя дуга, входящая в состояние выталкивания, всегда помечена одним и тем же входным символом или символом e. Тогда состояние чтения можно исключить. (Можно видеть, что в этом случае состояние выталкивания имеет только одну исходящую дугу и она помечена этим символом.)

Пример 7.38. Рассмотрим автомат, показанный на рис. 7.54. У него три состояния опроса со степенью по выходу 1, а имен-

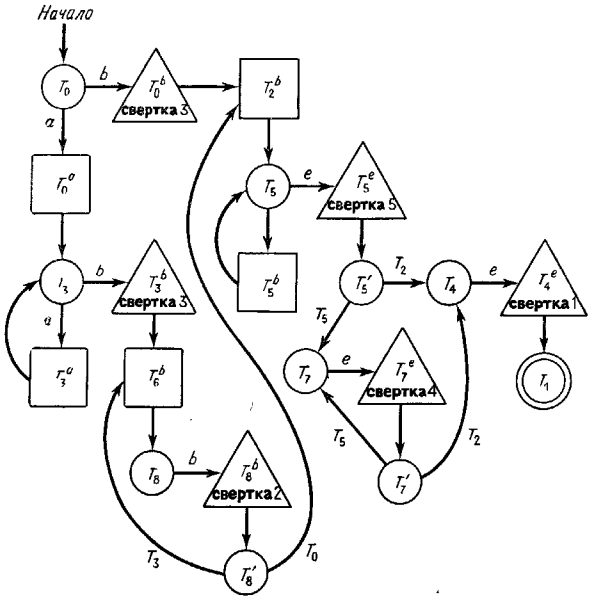


Рис. 7.55. Полуприведенный автомат.

но T_0', T_3' и T_4'. Эти состояния и исходящие из них дуги можно выбросить.

Исследуем состояние чтения T_6. В состояние T_6 можно попасть только по путям, выходящим из T_3 и T_3' или из T_8 и T_8'. В обоих случаях меткой предыдущего входного символа служит b, т. е. если в состоянии T_3 или T_8 на входе появляется b, то автомат переходит в состояние T_3^b или T_8^b для выполнения свертки. Символ b остается во входной цепочке до тех пор, пока

автомат, находясь в состоянии T_6, не прочтет его и не решит перейти в состояние T_6^b для выполнения переноса. Так как известно, что b на самом деле появляется во входной цепочке, состояние T_6 лишнее; в состоянии T_6^b автомат может поместить имя состояния T_6 в магазин, не считывая следующего входного символа, так как если автомат достиг состояния T_6, то этим входным символом должен быть символ b.

Аналогично можно исключить состояние T_2. Полученный в результате автомат показан на рис. 7.55. □

Как и в случае полуприведенного LR(0)-автомата, можно слить совместимые состояния, не затронув поведения автомата. Предоставляем читателю рассмотреть этот вопрос самостоятельно. На рис. 7.55 можно слить только пару T_5' и T_7'.

7.5.4. Обзор содержания главы

В этой главе мы изложили ряд методов, применяемых для уменьшения размера и повышения скорости работы синтаксических анализаторов. Как же нужно действовать при построении анализатора для данной грамматики с точки зрения всех предоставляемых возможностей?

Во-первых, надо принять решение о том, каким методом разбора воспользоваться — восходящим или нисходящим. Соответствующее решение принимается в зависимости от того, какой перевод нужно получить. Этот вопрос мы рассмотрим в гл. 9.

Если удобно воспользоваться нисходящим анализатором, то рекомендуется LL(1)-анализатор. Для построения такого анализатора требуется проделать следующее:

(1) Преобразовать грамматику в LL(1)-грамматику. Исходная грамматика очень редко оказывается LL(1)-грамматикой. Основными средствами такого преобразования служат левая факторизация и исключение левой рекурсии, но гарантировать, что эти преобразования обязательно приведут к успеху, нельзя. (См. примеры 5.10 и 5.11 в томе 1.)

(2) Если же удается получить эквивалентную LL(1)-грамматику G, то с помощью методов разд. 5.1 (в частности, с помощью алгоритма 5.1) легко построить для G LL(1)-таблицу разбора. Элементами этой таблицы на практике будут обращения к программам, которые оперируют с магазином, генерируют выходную цепочку или выдают сообщения об ошибках.

(3) Для уменьшения размера таблицы разбора применяются два метода:

(а) Если правило начинается с терминального символа, то этот терминальный символ нет необходимости помещать в магазин, если входной указатель перемещается. (Другими словами, если нужно воспользоваться правилом $A \to a\alpha$ и a — текущий входной символ, то α помещается в магазин, а входная головка сдвигается на один символ вправо.) Этот метод позволяет уменьшить число различных символов, появляющихся в магазине и, следовательно, число строк в LL(1)-таблице разбора.

(б) Некоторые нетерминалы с аналогичными действиями разбора можно объединить в один нетерминал с „индикатором", указывающим, какой нетерминал он представляет. Такому объединению легко поддаются нетерминалы, представляющие выражения. (См. упр. 7.3.28 и 7.3.31.)

Если удобно воспользоваться восходящим анализатором, то рекомендуется детерминированный алгоритм разбора типа „перенос — свертка", такой, как SLR(1)-анализатор или LALR(1)-анализатор. Синтаксис большинства языков программирования легко описать с помощью SLR(1)-грамматики, так что предварительно требуется лишь немного видоизменить исходную грамматику. Применение методов оптимизации позволяет существенно уменьшить размер SLR(1)- или LALR(1)-анализатора. Обычно имеет смысл исключить свертки по цепным правилам.

Дальнейшее уменьшение размера анализатора возможно, если LALR(1)-анализатор реализуется в виде анализатора на языке Флойда — Эванса из разд. 7.2. Элементы действия каждой LR(1)-таблицы можно представить в виде последовательности операторов переноса, за которыми следуют операторы свертки, за которыми в свою очередь следует один безусловный оператор ошибки. Если все операторы свертки включают одно и то же правило, то всех их вместе с последующим оператором ошибки можно заменить одним оператором, свертывающим в соответствии с этим правилом независимо от входной цепочки. Способность анализатора обнаруживать ошибки при такой оптимизации не затрагивается. См. упр. 7.3.23 и 7.5.13. Элементы перехода каждой LR(1)-таблицы, отличные от φ, запоминаются в виде списка пар (A, T), означающих, что на нетерминале A в верхушку магазина помещается таблица T. Переходы, отвечающие терминалам, можно закодировать в самих операторах переноса. Заметим, что элементы φ запоминать не требуется. После этого можно пользоваться методами оптимизации, описанными в разд. 7.2 и состоящими в том, что общие последовательности операторов сливаются.

Такие подходы к построению анализаторов обладают рядом практических преимуществ. Во-первых, полученный анализатор можно чисто механически отладить, порождая входные цепочки, позволяющие проверить поведение анализатора. Например, легко построить входные цепочки, проверяющие каждый полезный элемент LL(1)- или LR(1)-таблицы. Другое преимущество LL(1)- и LR(1)-анализаторов, особенно первого из них, связано с тем,

что небольшие изменения в синтаксис или семантику можно внести, просто изменив соответствующие элементы таблицы разбора.

Наконец, читатель может убедиться в том, что некоторые неоднозначные грамматики имеют "LL"- или "LR"-анализаторы, образованные в результате того, что конфликты разбора разрешаются некоторым произвольным образом (упр. 7.5.14). Конструирование анализаторов такого типа составляет предмет дальнейших исследований.

УПРАЖНЕНИЯ

7.5.1. Постройте для грамматики G_0 анализирующий автомат M. Постройте по M эквивалентные расщепленный, полуприведенный и приведенный автоматы.

7.5.2. Постройте анализирующие автоматы для всех грамматик из упр. 7.3.1.

7.5.3. Расщепите состояния автоматов из упр. 7.5.2 для построения приведенных анализирующих автоматов.

7.5.4. Докажите лемму 7.4.

7.5.5. Докажите, что определение приведенного автомата, данное в разд. 7.5.2, корректно, т. е. если $p \equiv q$, то $\delta(p, a) \equiv \delta(q, a)$ для всех a из $\Sigma' \cup \{e\}$.

7.5.6. Докажите теорему 7.13.

Определение. Пусть $G = (N, \Sigma, P, S')$ — пополненная КС-грамматика, правила которой занумерованы числами 0, 1, ... так, что нулевое правило есть $S' \to S$. Пусть $\Sigma' = \{\#_0, \#_1, \ldots, \#_p\}$ — множество специальных символов, не принадлежащих $N \cup \Sigma$. Пусть i-е правило имеет вид $A \to \beta$ и $S' \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha\beta w$. Тогда $\alpha\beta\#_i$ называется *характеристической цепочкой* правовыводимой цепочки $\alpha\beta w$.

*7.5.7. Покажите, что множество характеристических цепочек для КС-грамматики регулярно.

7.5.8. Покажите, что КС-грамматика G однозначна тогда и только тогда, когда каждая правовыводимая цепочка грамматики G, за исключением S', имеет единственную характеристическую цепочку.

7.5.9. Покажите, что КС-грамматика является LR(k)-грамматикой тогда и только тогда, когда любая правовыводимая цепочка $\alpha\beta w$, для которой $S' \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha\beta w$, имеет характеристическую цепочку, которую можно найти, зная только $\alpha\beta$ и $\mathrm{FIRST}_k(w)$.

7.5.10. Пусть $G = (N, \Sigma, P, S)$ — LR(0)-грамматика и $(\mathscr{T}, T_0)$ — ее каноническое множество LR(0)-таблиц. Пусть $M = (\mathscr{T}, \Sigma \cup \mathscr{T}, \delta, T_0, \{T_1\})$ — канонический анализирующий автомат для G. Пусть $M' = (\mathscr{T} \cup \{q_f\}, \Sigma \cup \Sigma', \delta', T_0, \{q_f\})$ — детерминированный конечный автомат, построенный по M следующим образом:

(1) $\delta'(T, a) = \delta(T, a)$ для всех $T \in \mathscr{T}$ и $a \in \Sigma$,

(2) $\delta'(T, \#_i) = q_f$, если $\delta(T, T')$ определено и T' — состояние свертки, вызывающее свертку по правилу i.

Покажите, что $L(M')$ — множество характеристических цепочек для G.

7.5.11. Напишите алгоритм слияния „эквивалентных" состояний полуприведенного автомата, построенного для LR(1)-грамматики. Эквивалентными здесь называются состояния, переходы из которых помечены одним и тем же множеством символов, причем одинаково помеченные переходы ведут в эквивалентные состояния [1]).

7.5.12. Предположим, что мы изменили определение эквивалентности, данное в упр. 7.5.11, так, что эквивалентными теперь считаются и те состояния, переходы из которых, помеченные символом a, всегда, когда они существуют, ведут в эквивалентные состояния. Будет ли полученный автомат эквивалентен (в формальном смысле, т. е. при условии, что перенос запрещен, если исходный автомат сообщил об ошибке) полуприведенному автомату?

7.5.13. Предположим, что в LR(1)-анализирующем автомате из состояния чтения T переходы происходят только в состояния выталкивания, свертывающие по одному и тому же правилу. Покажите, что если исключить T и слить все эти состояния выталкивания в одно состояние, то новый автомат будет свертывать независимо от аванцепочки, но останется эквивалентным исходному автомату.

7.5.14. Пусть G — однозначная грамматика с правилами

$$S \to \textbf{if } b \textbf{ then } SE \mid a$$
$$E \to \textbf{else } S \mid e$$

(а) Покажите, что $L(G)$ не является LL-языком.

(б) Постройте для G 1-предсказывающий анализатор, считая, что всякий раз, когда в верхушке магазина находится символ E, а следующим входным символом является **else**, нужно применить правило $E \to \textbf{else}\, S$.

(в) При аналогичных предположениях постройте для G LR(1)-анализатор.

[1]) Это определение можно сделать точным, задав отношения $\equiv^0$, $\equiv^1$, ..., как это делалось в лемме 7.2.

Проблемы для исследования

7.5.15. Примените используемый метод, а именно разбиение анализатора на ряд активных компонент и слияние или исключение некоторых из них, к анализаторам, отличным от LR-анализаторов. Например, в методе, описанном в разд. 7.2, активными компонентами были строки матрицы предшествования. Разработайте методы, применимые к LL-анализаторам и различным типам анализаторов предшествования.

7.5.16. Некоторые состояния канонического анализирующего автомата обладают тем свойством, что автомат с такими состояниями распознает только регулярные множества. Рассмотрим задачу расщепления состояний переноса анализирующего автомата на состояния просмотра и заталкивания. В состоянии просмотра могут просматриваться входные символы и генерироваться выходная цепочка, но не производится никаких действий с магазином. После состояния просмотра управление передается в другое состояние просмотра или в состояние заталкивания. Таким образом, множество состояний просмотра работает как конечный преобразователь. В состоянии заталкивания в магазин помещается имя текущего состояния. Разработайте преобразования, с помощью которых можно оптимизировать анализирующий автомат с состояниями просмотра и заталкивания, а также с расщепленными состояниями свертки.

7.5.17. Опишите методы оптимизации из разд. 7.3 в терминах методов из разд. 7.5 и наоборот.

Упражнения на программирование

7.5.18. Разработайте элементарные операции, необходимые для реализации расщепленного канонического автомата. Постройте для них интерпретатор.

7.5.19. Напишите программу, получающую на вход расщепленный канонический автомат и строящую по нему последовательность элементарных операций, моделирующую поведение анализирующего автомата.

7.5.20. Постройте для одной из грамматик из приложения к тому 1 два LR(1)-анализатора. Один LR(1)-анализатор должен быть LR(1)-анализатором интерпретирующего типа, работающим с множеством LR(1)-таблиц. Другой — последовательностью элементарных операций, моделирующей анализирующий автомат. Сравните размеры и скорости работы этих анализаторов.

Замечания по литературе

Подход, использующий анализирующий автомат, предложен Де Ремером [1969]. Определение характеристической цепочки, предшествующее упр. 7.5.7, заимствовано из того же источника. Классы неоднозначных грамматик, разбираемых LL- или LR-анализаторами, рассматривались Ахо и др. [1972].

8 ТЕОРИЯ ДЕТЕРМИНИРОВАННОГО РАЗБОРА

В гл. 5 мы познакомились с различными классами грамматик, для которых удается построить эффективные детерминированные синтаксические анализаторы. Были продемонстрированы некоторые отношения включения на классах грамматик. Например, мы показали, что каждая (m, k)-ОПК-грамматика является LR(k)-грамматикой. В настоящей главе мы пополним иерархию соотношений между различными классами грамматик.

Можно задаться вопросом о том, какой класс языков порождается грамматиками из данного класса. В этой главе мы увидим, что большинство классов грамматик из гл. 5 порождает в точности детерминированные КС-языки. Точнее, будет показано, что каждый из следующих классов грамматик порождает в точности детерминированные КС-языки:

(1) LR(1)-грамматики,
(2) (1,1)-ОПК-грамматики,
(3) обратимые грамматики (2,1)-предшествования,
(4) простые грамматики со смешанной стратегией предшествования.

При выводе этих результатов применяются алгоритмы преобразования грамматик одного типа в грамматики другого типа. Таким образом, для каждого детерминированного КС-языка можно найти грамматику, анализируемую с помощью алгоритма обратимого (2,1)-предшествования или алгоритма со смешанной стратегией предшествования. Однако многие из этих преобразующих алгоритмов зачастую дают грамматики, слишком большие для практического использования.

Интерес представляют три собственных подкласса детерминированных КС-языков:

(1) языки простого предшествования,
(2) языки операторного предшествования,
(3) LL-языки.

Языки операторного предшествования образуют собственный подкласс языков простого предшествования, и он не сравним с классом LL-языков. В гл. 5 мы видели, что класс языков, порождаемых обратимыми грамматиками слабого предшествования, совпадает с классом языков простого предшествования. Эта иерархия языков отражена на рис. 8.1.

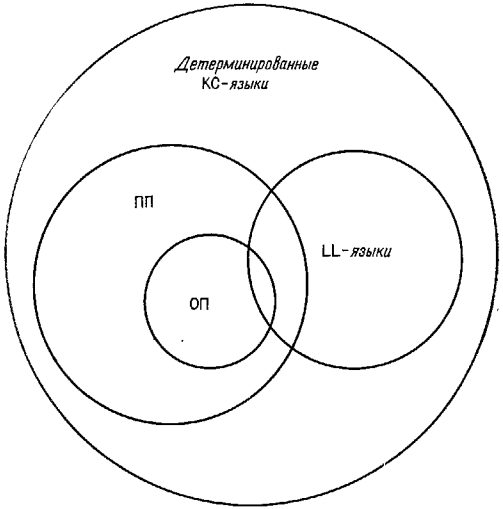


Рис. 8.1. Иерархия детерминированных КС-языков. (ПП — языки простого предшествования, ОП — языки операторного предшествования.)

В данной главе мы получим эту иерархию и отметим наиболее характерные черты каждого класса языков. Глава состоит из трех разделов. В первом изучаются LL-языки и их свойства. Во втором рассматривается класс детерминированных языков, а в третьем — языки простого и операторного предшествования.

Эта глава является своего рода лакомством — при строгой диете теории компиляции она не существенна, и при первом чтении ее можно опустить [1]).

[1]) Впрочем, читатели, которые не любят лакомств, могут вовсе к ней не возвращаться.

8.1. ТЕОРИЯ LL-ЯЗЫКОВ

Начнем с основных результатов, касающихся LL-языков и грамматик. Мы докажем следующие шесть утверждений:

(1) Каждая LL(k)-грамматика является LR(k)-грамматикой (теорема 8.1).

(2) Для каждого LL(k)-языка существует LL($k+1$)-грамматика в нормальной форме Грейбах (теорема 8.5).

(3) Проблема, эквивалентны ли две произвольные LL-грамматики, разрешима (теорема 8.6).

(4) Язык, не содержащий пустого слова, является LL(k)-языком тогда и только тогда, когда он имеет неукорачивающую (т. е. без e-правил) LL($k+1$)-грамматику (теорема 8.7).

(5) Для $k \geqslant 0$ множество LL(k)-языков является собственным подмножеством множества LL($k+1$)-языков (теорема 8.8).

(6) Существуют LR-языки, не являющиеся LL-языками (упр. 8.1.11).

8.1.1. LL- и LR-грамматики

Докажем сначала, что каждая LL(k)-грамматика является LR(k)-грамматикой. Это утверждение неформально можно обосновать так. Рассмотрим дерево вывода, изображенное схематически на рис. 8.2. Согласно LR(k)-условию, при просмотре входной цепочки wxy правило $A \to \alpha$ должно распознаваться после того, как станут известны подцепочка wx и множество $\mathrm{FIRST}_k(y)$. С другой стороны, в LL(k)-условии для распознавания правила $A \to \alpha$ требуется знать только w и $\mathrm{FIRST}_k(xy)$. Поэтому LL(k)-условие должно быть более сильным, чем LR(k)-условие, так что каждая LL(k)-грамматика должна быть LR(k)-грамматикой. Опишем теперь это формально.

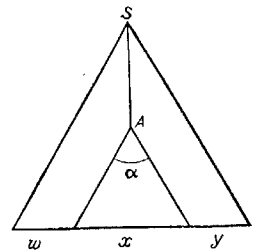


Рис. 8.2. Набросок дерева вывода.

Пусть дана LL(k)-грамматика G и в ней два дерева разбора. Допустим, что первые m символов крон этих двух деревьев совпадают. Тогда каждой вершине одного дерева, слева от которой находится не более $m-k$ листьев, помеченных терминалами, соответствует „идентичная“ вершина в другом дереве. Эта связь изображена на рис. 8.3, где заштрихованная область представляет „идентичные“ вершины. Мы считаем, что $|w| = m-k$ и $\mathrm{FIRST}_k(x_1) = \mathrm{FIRST}_k(x_2)$. Сформулируем наш результат в виде леммы:

Лемма 8.1. *Пусть G — LL(k)-грамматика и*

$$S \Rightarrow_l^{m_1} w_1 A\alpha \Rightarrow_l^* w_1 x_1 \quad \text{и} \quad S \Rightarrow_l^{m_2} w_2 B\beta \Rightarrow_l^* w_2 x_2$$

— два таких левых вывода, что $\mathrm{FIRST}_l(w_1x_1) = \mathrm{FIRST}_l(w_2x_2)$ *при* $l = k + \max(|w_1|, |w_2|)$. *(Другими словами, по крайней мере k символов, стоящих в w_1x_1 и w_2x_2 после w_1 и w_2, совпадают.)*

(1) *Если* $m_1 = m_2$, *то* $w_1 = w_2$, $A = B$ *и* $\alpha = \beta$.

(2) *Если* $m_1 < m_2$, *то* $S \Rightarrow_l^{m_1} w_1 A\alpha \Rightarrow_l^{m_2 - m_1} w_2 B\beta \Rightarrow^* w_2 x_2$.

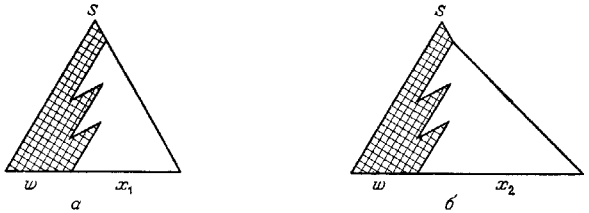


Рис. 8.3. Два дерева разбора для LL(k)-грамматики.

Доказательство. По определению LL(k)-грамматики при $m_1 \leqslant m_2$ первые m_1 шагов вывода $S \Rightarrow_l^{m_2} w_2 B\beta$ представляют собой вывод $S \Rightarrow_l^{m_1} w_1 A\alpha$, так как на каждом шаге этих выводов аванцепочки совпадают. Отсюда сразу следуют утверждения (1) и (2). □

Теорема 8.1. *Всякая* LL(k)-*грамматика*[1]) *является* LR(k)-*грамматикой.*

Доказательство. Пусть $G = (\mathrm{N}, \Sigma, P, S)$ — LL(k)-грамматика. Предположим, что она не LR(k)-грамматика. Тогда в пополненной грамматике существуют такие два правых вывода

(8.1.1) $$S' \Rightarrow_r^i \alpha A x_1 \Rightarrow_r \alpha\beta x_1$$

(8.1.2) $$S' \Rightarrow_r^j \gamma B y \Rightarrow_r \gamma\delta y$$

что $\gamma\delta y = \alpha\beta x_2$ для некоторой цепочки x_2, для которой $\mathrm{FIRST}_k(x_2) = \mathrm{FIRST}_k(x_1)$. Поскольку G по предположению не LR(1)-грамматика, можно считать, что $\alpha A x_2 \neq \gamma B y$.

Можно считать, что в обоих этих выводах i и j оба больше нуля. В противном случае было бы, например,

$$S' \Rightarrow^0 S' \Rightarrow S$$
$$S \Rightarrow^+ By \Rightarrow S\beta y$$

[1]) Везде в этой книге предполагается, что грамматика не имеет бесполезных правил.

а это означало бы, что G леворекурсивна и, следовательно, не является LL-грамматикой. Поэтому в оставшейся части доказательства мы будем считать, что в выводах (8.1.1) и (8.1.2) S' можно заменить на S.

Пусть x_α, x_β, x_γ и x_δ — терминальные цепочки, выводимые из α, β, γ и δ соответственно, причем $x_\alpha x_\beta x_2 = x_\gamma x_\delta y$. Рассмотрим левые выводы, соответствующие выводам

(8.1.3) $$S \Rightarrow_r^* \alpha A x_1 \Rightarrow_r \alpha\beta x_1 \Rightarrow_r^* x_\alpha x_\beta x_1$$

(8.1.4) $$S \Rightarrow_r^* \gamma B y \Rightarrow_r \gamma\delta y \Rightarrow_r^* x_\gamma x_\delta y$$

Точнее, пусть

(8.1.5) $$S \Rightarrow_l^* x_\alpha A\eta \Rightarrow_l x_\alpha\beta\eta \Rightarrow_l^* x_\alpha x_\beta\eta \Rightarrow_l^* x_\alpha x_\beta x_1$$

(8.1.6) $$S \Rightarrow_l^* x_\gamma B\theta \Rightarrow_l x_\gamma\delta\theta \Rightarrow_l^* x_\gamma x_\delta\theta \Rightarrow_l^* x_\gamma x_\delta y$$

где η и θ — подходящие цепочки из $(\mathrm{N} \cup \Sigma)^*$.

Согласно лемме 8.1, либо последовательность шагов в выводе $S \Rightarrow_l^* x_\alpha A\eta$ представляет собой начальную последовательность шагов в выводе $S \Rightarrow_l^* x_\gamma B\theta$, либо обратно. Исследуем первый случай; второй исследуется аналогично. Итак, вывод 8.1.6 можно записать в виде

(8.1.7) $$S \Rightarrow_l^* x_\alpha A\eta \Rightarrow_l x_\alpha\beta\eta \Rightarrow_l^* x_\gamma B\theta \Rightarrow_l x_\gamma\delta\theta \Rightarrow_l^* x_\gamma x_\delta\theta \Rightarrow_l^* x_\gamma x_\delta y$$

Рассмотрим дерево разбора T, соответствующее выводу (8.1.7). Пусть n_A — вершина, отвечающая в цепочке $x_\alpha A\eta$ символу A, а n_B — вершина, отвечающая в цепочке $x_\gamma B\theta$ символу B. Эти вершины показаны на рис. 8.4. Заметим, что n_B может быть потомком вершины n_A. Цепочки x_β и x_δ не могут перекрываться частично: либо они не пересекаются, либо x_δ — подцепочка цепочки x_β. Мы изобразили их на рис. 8.4 частично перекрывающимися лишь для того, чтобы легче было рассуждать в обоих случаях.

Рассмотрим теперь два правых вывода, связанных с деревом разбора T. В первом T расширяется вправо до вершины n_A включительно; во втором T расширяется вплоть до вершины n_B. Последний вывод можно записать так:

$$S \Rightarrow_r^* \gamma B y \Rightarrow_r \gamma\delta y$$

На самом деле это вывод (8.1.2). Правый вывод вплоть до вершины n_A имеет вид

(8.1.8) $$S \Rightarrow_r^* \alpha' A x_2 \Rightarrow_r \alpha'\beta x_2$$

для некоторой цепочки α'. Потом мы докажем, что $\alpha' = \alpha$, а сейчас примем это на веру и, получив противоречие LL(k)-условию, докажем тем самым теорему.

Итак, пусть $\alpha' = \alpha$. Тогда $\gamma\delta y = \alpha'\beta x_2 = \alpha\beta x_2$. Поэтому для продолжения выводов (8.1.2) и (8.1.8) до терминальной цепочки

$x_\gamma x_\delta y$ можно использовать один и тот же правый вывод. Но поскольку мы считаем вершины n_A и n_B различными, выводы (8.1.2) и (8.1.8) различаются, а, следовательно, различаются и законченные выводы. Отсюда вытекает, что цепочка $x_\gamma x_\delta y$ имеет два разных правых вывода и, значит, грамматика G неоднозначна. Согласно упр. 5.1.3, LL-грамматика не может быть неоднозначной. Поэтому G вопреки допущению не является LL-грамматикой.

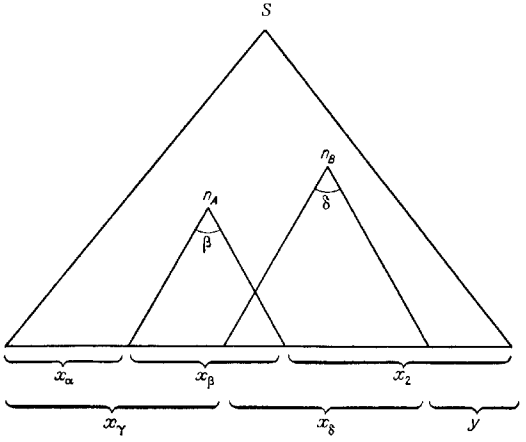


Рис. 8.4. Дерево разбора T.

Теперь докажем, что $\alpha' = \alpha$. Заметим, что цепочка α' составлена из выписанных слева направо меток тех вершин дерева T, прямой предок которых является предком вершины n_A. (Читатель может самостоятельно проверить это свойство правых выводов.) Рассмотрим опять левый вывод (8.1.5), имеющий то же самое дерево разбора, что и правый вывод (8.1.3). Пусть T' — дерево разбора, связанное с выводом (8.1.3). Шаги вывода (8.1.5) вплоть до получения цепочки $x_\alpha A\eta$ совпадают с шагами вывода (8.1.7) вплоть до получения цепочки $x_\alpha A\eta$. Пусть n'_A — вершина дерева T', соответствующая нетерминалу, который в выводе (8.1.7) заменяется на шаге $x_\alpha A\eta \Rightarrow_l x_\alpha\beta\eta$. Пусть П — прямо упорядоченное[1]) множество внутренних вершин дерева разбора T вплоть до вершины n_A, а П′ — прямо упорядоченное множество внутренних вершин дерева T' вплоть до вершины n'_A. Тогда i-я

[1]) П — это последовательность внутренних вершин дерева T, взятых в том порядке, в котором нетерминалы, отвечающие этим вершинам, разворачиваются в левом выводе.

вершина множества П согласуется с i-й вершиной множества П′ в том смысле, что обе они имеют одинаковые метки и их соответствующие потомки либо согласуются, либо расположены справа от n_A и n'_A соответственно.

Вершины дерева T', прямым предком которых является предок вершины n'_A, имеют такие метки, что если их выписать слева направо, они образуют цепочку α. Но эти вершины согласуются с теми вершинами дерева T, которые образуют α', поэтому $\alpha' = \alpha$. Теорема доказана. □

Грамматика
$$S \to A \mid B$$
$$A \to aAb \mid 0$$
$$B \to aBbb \mid 1$$

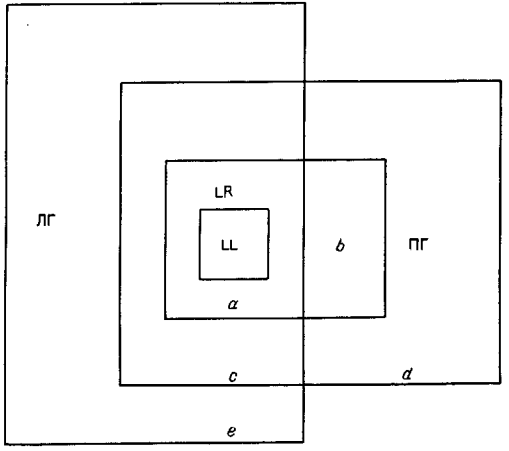


Рис. 8.5. Соотношения между классами грамматик. (ЛГ — левоанализируемые грамматики, ПГ — правоанализируемые грамматики.)

является LR(0)-грамматикой, но не LL-грамматикой (пример 5.4), так что включение класса LL-грамматик в класс LR-грамматик собственное. На самом деле, если рассмотреть классы LL- и LR-грамматик вместе с классами левоанализируемых и правоанализируемых (при помощи ДМП-автомата с концевым маркером) грамматик, то, согласно теоремам 5.5, 5.12 и 8.1, получится картина, изображенная на рис. 8.5.

Мы утверждаем, что каждый из шести классов грамматик на рис. 8.5 непуст. Мы знаем, что LL-грамматика существует, так что надо доказать следующее:

Теорема 8.2. *Существуют грамматики, которые являются*

(1) LR *и левоанализируемыми, но не* LL,
(2) LR, *но не левоанализируемыми,*
(3) *лево- и правоанализируемыми, но не* LR,
(4) *правоанализируемыми, но не* LR *и не левоанализируемыми,*
(5) *левоанализируемыми, но не правоанализируемыми.*

Доказательство. Каждая из следующих грамматик принадлежит соответствующему классу.

(1) Грамматика G_a с правилами

$$S \to A \mid B$$
$$A \to aaA \mid aa$$
$$B \to aaB \mid a$$

является LR(1) и левоанализируемой, но не LL.

(2) Грамматика G_b с правилами

$$S \to Ab \mid Ac$$
$$A \to AB \mid a$$
$$B \to a$$

является LR(1), но не левоанализируемой. (См. пример 3.27, том 1.)

(3) Грамматика G_c с правилами

$$S \to Ab \mid Bc$$
$$A \to Aa \mid a$$
$$B \to Ba \mid a$$

является лево- и правоанализируемой, но не LR.

(4) Грамматика G_d с правилами

$$S \to Ab \mid Bc$$
$$A \to Ac \mid a$$
$$B \to Bc \mid a$$
$$C \to a$$

является правоанализируемой, но не LR и не левоанализируемой.

(5) Грамматика G_e с правилами

$$S \to BAb \mid CAc$$
$$A \to BA \mid a$$
$$B \to a$$
$$C \to a$$

является левоанализируемой, но не правоанализируемой. (См. пример 3.26, том 1.) □

8.1.2. LL-грамматики в нормальной форме Грейбах

В настоящем разделе мы рассмотрим преобразования LL-грамматик, сохраняющие LL-свойство. Наши первые результаты касаются e-правил. Мы дадим два алгоритма, применяя которые последовательно, можно преобразовать LL(k)-грамматику в эквивалентную LL($k+1$)-грамматику, не содержащую e-правил. Первый алгоритм изменяет грамматику так, чтобы каждая правая часть либо стала равна e, либо начиналась с символа (возможно, терминального), из которого не выводится пустая цепочка. Второй алгоритм преобразует грамматику, удовлетворяющую этому условию, в грамматику без e-правил. Оба алгоритма сохраняют LL-свойство грамматики.

Определение. Пусть $G=(N, \Sigma, P, S)$ — КС-грамматика. Будем называть нетерминал A *исчезающим*, если из A выводится пустая цепочка. В противном случае символ из $N \cup \Sigma$ назовем *неисчезающим*. Таким образом, каждый терминал — неисчезающий символ. Будем говорить, что грамматика G *удовлетворяет условию* (*), если каждое правило из P имеет вид $A \to e$ или $A \to X_1 \ldots X_k$, где X_1 — неисчезающий символ[1]).

Алгоритм 8.1. Преобразование грамматики в грамматику, удовлетворяющую условию (*).

Вход. КС-грамматика $G=(N, \Sigma, P, S)$.

Выход. КС-грамматика $G_1=(N_1, \Sigma, P_1, S_1)$, удовлетворяющая условию (*), для которой $L(G_1)=L(G)-\{e\}$[2]).

Метод.

(1) Пусть $N' = N \cup \{\bar{A} \mid A$ — исчезающий нетерминал из $N\}$. Из нетерминала $\bar{A}$ будут выводиться те же цепочки, что и из A, за исключением e. Следовательно, $\bar{A}$ будет неисчезающим.

(2) Если $e \in L(G)$, положим $S_1=\bar{S}$. В противном случае $S_1=S$.

(3) Каждое правило из P, не являющееся e-правилом, можно единственным образом представить в виде $A \to B_1 \ldots B_m X_1 \ldots X_n$ (где $m \geqslant 0$, $n \geqslant 0$ и $m+n>0$), причем каждый символ B_i исчезающий и, если $n>0$, X_1 — неисчезающий символ. Остальные $X_j (1<j \leqslant n)$ могут быть как исчезающими, так и неисчезающими. Таким образом, X_1 — самый левый неисчезающий символ в правой части правила. Для каждого правила $A \to B_1 \ldots B_m X_1 \ldots X_n$, не являющегося e-правилом, построим P':

[1]) В оригинале такая грамматика называется nonnullable. — *Прим. перев.*

[2]) Получаемая по алгоритму 8.1 грамматика G_1 не эквивалентна исходной. Следуя примеру авторов, рекомендуем читателю самому в качестве упражнения найти и исправить ошибки алгоритма 8.1. — *Прим. ред.*

(а) если $m \geqslant 1$, включим в P' m правил

$$A \to \overline{B}_1 B_2 \ldots B_m X_1 \ldots X_n$$
$$A \to \overline{B}_2 B_3 \ldots B_m X_1 \ldots X_n$$
$$\vdots$$
$$A \to \overline{B}_m X_1 \ldots X_n$$

(б) если $n \geqslant 1$ и $m \geqslant 0$, включим в P' правило

$$A \to X_1 \ldots X_n$$

(в) кроме того, если A — исчезающий символ, включим в P' все правила из (а) и (б), где слева вместо A стоит $\overline{A}$.

(4) Если правило $A \to e$ принадлежит P, включим в P' правило $A \to e$.

(5) Положим $G_1 = (\mathrm{N}', \Sigma, P_1, S_1)$, где G_1 — это грамматика $G' = (\mathrm{N}', \Sigma, P', S_1)$, из которой выброшены все бесполезные символы и правила. □

Пример 8.1. Пусть G — LL(1)-грамматика с правилами

$$S \to AB$$
$$A \to aA \mid e$$
$$B \to bA \mid e$$

Здесь все нетерминалы исчезающие, так что введем новые нетерминалы $\overline{S}$, $\overline{A}$ и $\overline{B}$; первый из них будет новым начальным символом. Правила $A \to aA$ и $B \to bA$ оба начинаются с неисчезающего символа, поэтому их правые части имеют вид $X_1 X_2$ (для шага (3) алгоритма 8.1). Таким образом, эти правила будут сохранены, а к множеству правил добавятся еще правила

$$\overline{A} \to aA \qquad \overline{B} \to bA$$

В правиле $S \to AB$ все символы правой части исчезающие, так что ее можно записать в виде $B_1 B_2$. Это правило заменяется на четыре правила

$$S \to \overline{A}B \mid \overline{B}$$
$$\overline{S} \to \overline{A}B \mid \overline{B}$$

Так как $\overline{S}$ — новый начальный символ, находим, что символ S теперь недостижим. Окончательное множество правил, построенное по алгоритму 8.1, таково:

$$\overline{S} \to \overline{A}B \mid \overline{B}$$
$$\overline{A} \to aA$$
$$A \to aA \mid e$$
$$\overline{B} \to bA$$
$$B \to bA \mid e \qquad \square$$

Докажем, что алгоритм 8.1 сохраняет свойство грамматики быть LL(k)-грамматикой.

Теорема 8.3. *Пусть G_1 — грамматика, полученная из G по алгоритму 8.1. Тогда*

(1) $L(G_1) = L(G) - \{e\}$,

(2) *если G — LL(k)-грамматика, то G_1 — тоже LL(k)-грамматика.*

Доказательство. (1) Индукцией по длине цепочки можно показать, что для всех A из N

(а) $A \Rightarrow^*_{G_1} w$ тогда [1]) и только тогда, когда $A \Rightarrow^*_G w$,

(б) $\overline{A} \Rightarrow^*_{G_1} w$ тогда [1]) и только тогда, когда $w \neq e$ и $A \Rightarrow^*_G w$.

Подробное доказательство оставляем читателю в качестве упражнения.

Утверждение (2) докажем от противного. Допустим, что оно неверно. Тогда в G_1 можно найти выводы

$$S_1 \Rightarrow^*_{G_1 l} w\hat{A}\alpha \Rightarrow_{G_1 l} w\beta\alpha \Rightarrow^*_{G_1 l} wx$$
$$S_1 \Rightarrow^*_{G_1} w\hat{A}\alpha \Rightarrow_{G_1 l} w\gamma\alpha \Rightarrow^*_{G_1 l} wy$$

где $\mathrm{FIRST}_k(x) = \mathrm{FIRST}_k(y)$, $\beta \neq \gamma$ и $\hat{A}$ — либо A, либо $\overline{A}$.

По этим двум выводам можно построить соответствующие выводы цепочек wx и wy в грамматике G. Начнем с того, что определим $h(A) = h(\overline{A}) = A$ для $A \in \mathrm{N}$ и $h(a) = a$ для $a \in \Sigma$.

Для каждого правила $\hat{B} \to \delta$ из P_1 найдем в P правило $B \to \delta' h(\delta)$, из которого оно было построено на шаге (3) алгоритма 8.1. Другими словами, $B = h(\hat{B})$, а δ' — некоторая (возможно, пустая) цепочка исчезающих символов. Всякий раз, когда правило $\hat{B} \to \delta$ применяется в каком-нибудь из выводов в грамматике G_1, в соответствующем выводе в грамматике G заменим его правилом $B \to \delta' h(\delta)$, а затем произведем левый вывод цепочки e из δ', если $\delta' \neq e$. Таким образом, получаем

$$S \Rightarrow^*_{Gl} wAh(\alpha) \Rightarrow_{Gl} w\beta' h(\beta)h(\alpha) \Rightarrow^*_{Gl} wh(\beta)h(\alpha) \Rightarrow^*_{Gl} wx$$
$$S \Rightarrow^*_{Gl} wAh(\alpha) \Rightarrow_{Gl} w\gamma' h(\gamma)h(\alpha) \Rightarrow^*_{Gl} wh(\gamma)h(\alpha) \Rightarrow^*_{Gl} wy$$

[1]) Это неверно. См. предыдущую сноску. — *Прим. ред.*

Шаги вывода от $w\beta' h(\beta)h(\alpha)$ до $wh(\beta)h(\alpha)$ можно записать в виде

$$w\beta' h(\beta)h(\alpha) = w\delta_1 \Rightarrow_l w\delta_2 \Rightarrow_l \ldots \Rightarrow_l w\delta_n = wh(\beta)h(\alpha)$$

а шаги от $w\gamma' h(\gamma)h(\alpha)$ до $wh(\gamma)h(\alpha)$ — в виде

$$w\gamma' h(\gamma)h(\alpha) = w\varepsilon_1 \Rightarrow_l w\varepsilon_2 \Rightarrow_l \ldots \Rightarrow_l w\varepsilon_m = wh(\gamma)h(\alpha)$$

Пусть $z = \mathrm{FIRST}(x) = \mathrm{FIRST}(y)$. Мы утверждаем, что z принадлежит $\mathrm{FIRST}(\delta_i)$ и $\mathrm{FIRST}(\varepsilon_i)$ для всех i, поскольку β' и γ' состоят только из исчезающих символов (если β' и γ' непусты). Так как G — LL(k)-грамматика, то $\delta_i = \varepsilon_i$ для всех i. В частности, $\beta' h(\beta) = \gamma' h(\gamma)$. Следовательно, β и γ получаются по алгоритму 8.1 из одного и того же правила. Если $\beta' \neq \gamma'$, то в одном из приведенных выше выводов из нетерминала выводится e, а в другом не выводится. Так как это противоречит LL(k)-условию, заключаем, что $\beta' = \gamma'$. Так как цепочки β и γ предполагаются различными, они не могут одновременно быть пустыми, и значит, обе начинаются с одного и того же неисчезающего символа. Итак, $m = n$, а это противоречит предположению о том, что $\beta \neq \gamma$. □

Следующее наше преобразование полностью исключает из LL-грамматики e-правила. Будем считать, что $e \notin L(G)$. Идея заключается в том, чтобы сначала применить к LL(k)-грамматике алгоритм 8.1, а затем, скомбинировав в выводах неисчезающие символы со всеми последующими исчезающими символами, заменить полученную цепочку одним символом. Если грамматика является LL(k)-грамматикой, то число последовательных исчезающих символов, которые могут стоять за неисчезающими символами, ограничено.

Определение. Пусть $G = (\mathrm{N}, \Sigma, P, S)$ — КС-грамматика. Обозначим через V_G множество символов вида $[XB_1 \ldots B_n]$, где

(1) X — неисчезающий символ грамматики G (возможно, терминальный),

(2) $B_1, \ldots, B_n$ — исчезающие символы (следовательно, нетерминалы),

(3) если $i \neq j$, то $B_i \neq B_j$ (т. е. список $B_1, \ldots, B_n$ не содержит повторений).

Зададим гомоморфизм g из множества V_G в множество $(\mathrm{N} \cup \Sigma)^*$, положив $g([\alpha]) = \alpha$.

Лемма 8.2. *Пусть $G = (\mathrm{N}, \Sigma, P, S)$ — LL(k)-грамматика, удовлетворяющая условию (*) и такая, что из каждого нетерминала выводится хотя бы одна непустая терминальная цепочка, а V_G и g определены выше. Тогда для каждой левовыводимой в G непустой цепочки α существует единственная цепочка β из V_G^*, для которой $h(\beta) = \alpha$.*

Доказательство. Цепочку α можно единственным образом записать в виде $\alpha_1\alpha_2 \ldots \alpha_m$, $m \geqslant 1$, где α_i для всех i — это неисчезающий символ, за которым следует цепочка исчезающих символов (поскольку в G каждая непустая выводимая цепочка начинается с неисчезающего символа). Достаточно показать, что $[\alpha_i]$ для всех i принадлежит V_G. Если это не так, то α_i можно представить в виде $X\beta B\gamma B\delta$, где B — некоторый исчезающий символ, а β, γ и δ состоят только из исчезающих символов, т. е. α_i не удовлетворяет условию (3) определения V_G. Пусть w — такая непустая цепочка, что $B \Rightarrow_G^* w$. Тогда найдутся два различных левых вывода

$$\beta B\gamma B\delta \Rightarrow_l^* B\gamma B\delta \Rightarrow_l^* w\gamma B\delta \Rightarrow_l^* w$$
$$\beta B\gamma B\delta \Rightarrow_l^* B\delta \Rightarrow_l^* w\delta \Rightarrow_l^* w$$

Отсюда сразу получаем, что грамматика G неоднозначна и, следовательно, не является LL-грамматикой. □

Чтобы доказать, что всякий LL(k)-язык, не содержащий e, имеет LL($k+1$)-грамматику без e-правил, применим следующий алгоритм.

Алгоритм 8.2. Исключение из LL(k)-грамматики e-правил.

Вход. LL(k)-грамматика $G_1 = (\mathrm{N}_1, \Sigma, P_1, S_1)$.

Выход. LL($k+1$)-грамматика $G = (\mathrm{N}, \Sigma, P, S)$, для которой $L(G) = L(G_1) - \{e\}$.

Метод.

(1) Сначала воспользуемся алгоритмом 8.1 для построения LL(k)-грамматики $G_2 = (\mathrm{N}_2, \Sigma, P_2, S_2)$, удовлетворяющей условию (*).

(2) Исключим из G_2 все нетерминалы A, из которых выводятся только пустые цепочки, выбросив A из правых частей правил, где они встречаются, а затем исключим все A-правила. Обозначим полученную грамматику $G_3 = (\mathrm{N}_3, \Sigma, P_3, S_2)$.

(3) Построим грамматику $G = (\mathrm{N}, \Sigma, P, S)$:

(a) Пусть N — множество таких символов $[X\alpha]$, что
- (i) X — неисчезающий символ грамматики G_3,
- (ii) α — цепочка исчезающих символов,
- (iii) $X\alpha \notin \Sigma$ (т. е. не может быть одновременно $X \in \Sigma$ и $\alpha = e$),
- (iv) α не имеет повторяющихся символов,
- (v) $X\alpha$ действительно встречается в качестве подцепочки некоторой левовыводимой в G_3 цепочки.

(б) $S=[S_2]$.

(в) Пусть g — такой гомоморфизм, что $g([\alpha])=\alpha$ для всех $[\alpha]$ из N, и пусть $g(a)=a$ для a из Σ. Поскольку $g^{-1}(\beta)$ содержит не больше одного элемента, мы будем писать непосредственно $g^{-1}(\beta)$, если $g^{-1}(\beta)\neq\varnothing$. Построим множество P так:

(i) Пусть $[A\alpha]\in$ N и $A\to B$ — правило из P_3. Тогда $[A\alpha]\to g^{-1}(\beta\alpha)$ — правило из P.

(ii) Пусть $[a\alpha A\beta]\in$ N, где $a\in$ N, $A\in$ N$_3$. Пусть $A\to\gamma$ — правило из P_3, причем $\gamma\neq e$. Тогда $[a\alpha A\beta]\to ag^{-1}(\gamma\beta)$ — правило из P.

(iii) Пусть $[a\alpha A\beta]\in$ N, где $a\in$ N. Тогда $[a\alpha A\beta]\to a$ — правило из P. □

Пример 8.2. Рассмотрим грамматику из примера 8.1. Алгоритм 8.1 в этом примере уже применялся. Шаг (2) алгоритма 8.2 не изменяет грамматики. Будем порождать правила грамматики G по мере надобности, чтобы убедиться, что каждый рассматриваемый нетерминал встречается в некоторой левовыводимой цепочке. Начальным символом служит $[\bar{S}]$. Существуют два $\bar{S}$-правила: с правой частью $\bar{A}B$ и с правой частью $\bar{B}$. Так как $\bar{A}$ и $\bar{B}$ — неисчезающие символы, а B — исчезающий, то $g^{-1}(\bar{A}B)=[\bar{A}B]$ и $g^{-1}(\bar{B})=[\bar{B}]$. Следовательно, согласно пункту (i) шага (3в), получаем правила

$$[\bar{S}]\to[\bar{A}B]\,|\,[\bar{B}]$$

Рассмотрим нетерминал $[\bar{A}B]$. $\bar{A}$ имеет одно правило, а именно $\bar{A}\to aA$. Так как $g^{-1}(aAB)=[aAB]$, добавим правило

$$[\bar{A}B]\to[aAB]$$

Рассмотрев нетерминал $[\bar{B}]$, добавим еще правило

$$[\bar{B}]\to[bA]$$

Применим теперь к нетерминалу $[aAB]$ пункты (ii) и (iii). Для A и B существует по одному правилу, отличному от e-правила. Так как $g^{-1}(aAB)=[aAB]$, добавим правило

$$[aAB]\to a[aAB]$$

соответствующее A-правилу. Так как $g^{-1}(bA)=[bA]$, добавим правило

$$[aAB]\to a[bA]$$

соответствующее B-правилу. На шаге (iii) добавится правило

$$[aAB]\to a$$

Аналогично нетерминал $[bA]$ дает правила

$$[bA]\to b[aA]\,|\,b$$

Рассмотрев затем новый нетерминал $[aA]$, добавляем правила

$$[aA]\to a[aA]\,|\,a$$

Таким образом, для всех новых нетерминалов введены правила, и построение грамматики G закончено. □

Теорема 8.4. *Грамматика G, построенная по G_1 алгоритмом 8.2, обладает следующими свойствами:*

(1) $L(G)=L(G_1)-\{e\}$,

(2) *если G_1 — LL(k)-грамматика, то G — LL($k+1$)-грамматика.*

Доказательство. Пусть g — гомоморфизм, определенный на шаге (3в) алгоритма 8.2.

(1) По индукции непосредственно получаем, что $A\Rightarrow^*_{G_3}\beta$ (где $A\in$ N и $\beta\in(\text{N}\cup\Sigma)^*$) тогда и только тогда, когда $g(A)\Rightarrow^*_G g(\beta)$ и $\beta\neq e$. Следовательно, $[S_3]\Rightarrow^*_G w$ для w из Σ^* тогда и только тогда, когда $S_2\Rightarrow^*_{G_1} w$ и $w\neq e$. Значит, $L(G)=L(G_3)$. Равенство $L(G_2)=L(G_1)-\{e\}$ представляет собой утверждение (1) теоремы 8.3, и легко видеть, что шаг (2) алгоритма 8.2, превращающий G_2 в G_3, не изменяет порождаемого языка.

(2) Второе свойство доказывается аналогично. По данному левому выводу в G найдем соответствующий вывод в G_3 и покажем, что LL($k+1$)-конфликт в первом вызывает LL(k)-конфликт во втором. Параметр $k+1$ вместо k интуитивно можно обосновать так: если применяется правило грамматики G, построенное на шаге (3в) ((ii) или (iii)), то терминал a все еще остается частью аванцепочки, когда нужно выяснить, какое правило применить для $\alpha A\beta$. Допустим, что G не является LL($k+1$)-грамматикой. Тогда существуют выводы

$$S\Rightarrow^*_{Gl} wA\alpha\Rightarrow_{Gl} w\beta\alpha\Rightarrow^*_{Gl} wx$$
$$S\Rightarrow^*_{Gl} wA\alpha\Rightarrow_{Gl} w\gamma\alpha\Rightarrow^*_{Gl} wy$$

где $\beta\neq\gamma$, но $\text{FIRST}_{k+1}(x)=\text{FIRST}_{k+1}(y)$. Построим соответствующие выводы в G_3:

(а) Всякий раз, когда применяется правило $[A\alpha]\to g^{-1}(\beta\alpha)$, введенное в P на шаге (3вi), применяем правило $A\to\beta$ из G_3.

(б) Всякий раз, когда применяется правило $[a\alpha A\beta]\to ag^{-1}(\gamma\beta)$, введенное в P на шаге (3вii), строим левый вывод e из α, после чего применяем правило $A\to\gamma$.

(в) Всякий раз, когда применяется правило $[a\alpha A\beta]\to a$, строим левый вывод e из $\alpha A\beta$. Заметим, что этот вывод состоит не менее чем из одного шага, так как $\alpha A\beta\neq e$.

Таким образом, выводу $S \Rightarrow_{GI}^* wA\alpha$ соответствует единственный вывод $g(S) \Rightarrow_{G_3 I}^* wg(A)g(\alpha)$. Если A — заключенная в скобки цепочка символов, начинающаяся с терминала, например a, то рубежом[1]) цепочки $wg(A)g(\alpha)$ служит символ, расположенный через один от w вправо. В противном случае рубеж расположен непосредственно справа от w. В обоих случаях, поскольку $k+1$ символов цепочек x и y совпадают и G_3 — LL(k)-грамматика, шаги вывода в G_3, соответствующие применению правил $A \to \beta$ и $A \to \gamma$ в G, должны быть одинаковыми.

Чтобы показать, что вопреки предположению должно быть $\beta = \gamma$, достаточно исследовать три различных источника появления правил грамматики G и их связь с соответствующими выводами в G_3. Итак, пусть $A = [\delta]$. Если δ начинается с нетерминала, скажем $\delta = C\delta'$, то оба вывода строятся согласно пункту (а). Тогда в G_3 есть правило, скажем $C \to \delta''$, такое, что $\beta = \gamma = g^{-1}(\delta''\delta')$.

Если δ начинается с терминала, например $\delta = a\delta'$, то построение происходит согласно (б) или (в). Два вывода в G_3 заменяют некоторый префикс цепочки δ на e, а затем применяется правило, отличное от e-правила, из пункта (2б). Легко доказать, что в обоих случаях $\beta = \gamma$. □

Докажем теперь, что каждый LL(k)-язык порождается LL($k+1$)-грамматикой в нормальной форме Грейбах. Эта теорема находит важные применения, мы будем использовать ее для получения дальнейших результатов. Предварительно докажем две леммы.

Лемма 8.3. LL(k)-*грамматика не может быть леворекурсивной.*

Доказательство. Предположим, что $G = (N, \Sigma, P, S)$ имеет леворекурсивный нетерминал A. Тогда существует вывод $A \Rightarrow^+ A\alpha$. Если $\alpha \Rightarrow^* e$, то легко показать, что G неоднозначна и, значит, не может быть LL-грамматикой. Поэтому будем считать, что $\alpha \Rightarrow^* v$ для некоторой цепочки $v \in \Sigma^+$. Можно далее предположить, что $A \Rightarrow^* u$ для некоторой цепочки $u \in \Sigma^*$ и существует вывод

$$S \Rightarrow_l^* wA\delta \Rightarrow_l^* wA\alpha^k\delta \Rightarrow_l^* wuv^kx$$

Следовательно, существует и вывод

$$S \Rightarrow_l^* wA\delta \Rightarrow_l^* wA^k\delta \Rightarrow_l^* wA\alpha^{k+1}\delta \Rightarrow_l^* wuv^{k+1}x$$

Так как $\mathrm{FIRST}_k(uv^kx) = \mathrm{FIRST}_k(uv^{k+1}x)$, то для любого k приходим к противоречию с определением LL(k)-грамматики. □

Лемма 8.4. *Пусть* $G = (N, \Sigma, P, S)$ — *КС-грамматика, у которой в множестве* P *есть правило* $A \to B\alpha$, *где* $B \in N$. *Пусть* $B \to \beta_1 | \beta_2 | \ldots | \beta_m$ — *все* B-*правила грамматики* G, *а* $G_1 = (N, \Sigma, P_1, S)$ — *грамматика, полученная в результате удаления из* P *правила* $A \to B\alpha$ *и замены его правилами* $A \to \beta_1\alpha | \beta_2\alpha | \ldots | \beta_m\alpha$. *Тогда* $L(G_1) = L(G)$, *и если* G — LL(k)-*грамматика, то и* G_1 — *тоже* LL(k)-*грамматика.*

Доказательство. Согласно лемме 2.14, $L(G_1) = L(G)$. Чтобы показать, что LL(k)-свойство сохраняется, заметим, что левые выводы в G_1 по существу совпадают с левыми выводами в G, за исключением того, что последовательное применение правил $A \to B\alpha$ и $B \to B_i$ в грамматике G осуществляется теперь за один шаг. Говоря неформально, поскольку $B\alpha$ начинается с нетерминала, два шага вывода в грамматике G вызваны одной и той же k-символьной аванцепочкой. Поэтому при разборе в соответствии с грамматикой G_1 эта аванцепочка заставляет применить правило $A \to \beta_i\alpha$. Более подробное доказательство предлагаем в качестве упражнения. □

Алгоритм 8.3. Преобразование LL(k)-грамматики в LL($k+1$)-грамматику в нормальной форме Грейбах.

Вход. LL(k)-грамматика $G_1 = (N_1, \Sigma, P_1, S_1)$.

Выход. LL($k+1$)-грамматика $G = (N, \Sigma, P, S)$ в нормальной форме Грейбах, для которой $L(G) = L(G_1) - \{e\}$.

Метод.

(1) С помощью алгоритма 8.2 построить по G_1 LL($k+1$) грамматику $G_2 = (N_2, \Sigma, P_2, S)$, не содержащую e-правил.

(2) Перенумеровать нетерминалы из множества N_2, скажем $N_2 = \{A_1, \ldots, A_m\}$, так, чтобы выполнялось условие: если $A_i \to A_j\alpha$ принадлежит P_2, то $j > i$. Согласно лемме 2.16, это можно сделать, так как по лемме 8.3 грамматика G_2 не леворекурсивна.

(3) Для $i = m-1, m-2, \ldots, 1$ последовательно заменить все правила вида $A_i \to A_j\alpha$ теми из правил вида $A_i \to \beta\alpha$, для которых $A_j \to B$ уже отнесено к числу новых правил. Мы покажем, что эта операция приведет к тому, что у всех правил правые части будут начинаться с терминалов. Полученную грамматику обозначим $G_3 = (N_3, \Sigma, P_3, S)$.

(4) Пусть X_a для всех a из Σ будет новым нетерминальным символом. Положим

$$N = N_3 \cup \{X_a \mid a \in \Sigma\}$$

Пусть P образовано из P_3 заменой всех (кроме самых левых) вхождений терминала a на X_a и добавлением правил $X_a \to a$. Обозначим $G = (N, \Sigma, P, S)$. G — грамматика в нормальной форме Грейбах. □

[1]) Определение понятия рубежа см. в разд. 5.1.1.

Теорема 8.5. *Каждый* LL(k)*-язык порождается* LL($k+1$)*-грамматикой в нормальной форме Грейбах.*

Доказательство. Достаточно показать, что грамматика G, построенная алгоритмом 8.3, является LL($k+1$)-грамматикой, если G_1 — LL(k)-грамматика. По лемме 8.4 G_3 — LL($k+1$)-грамматика. Мы утверждаем, что правые части всех правил в G_3 начинаются с терминалов. Поскольку G_1 не леворекурсивна, это доказывается точно так же, как в алгоритме 2.14.

Легко показать, что все операции шага (4) сохраняют LL($k+1$)-свойство и не изменяют порождаемого грамматикой языка. Ясно также, что G — грамматика в нормальной форме Грейбах. Доказательства этих утверждений оставляем в качестве упражнений. □

8.1.3. Проблема эквивалентности для LL-грамматик

Теперь мы уже подготовлены к тому, чтобы изложить алгоритм, проверяющий, эквивалентны ли две LL(k)-грамматики. Однако надо ввести одно вспомогательное понятие.

Определение. Пусть $G=(N, \Sigma, P, S)$ — КС-грамматика. Для α из $(N \cup \Sigma)^*$ определим *мощность* [1]) цепочки α в G (и обозначим ее $TH^G(\alpha)$) как длину кратчайшей цепочки $w \in \Sigma^*$, выводимой из α. В качестве упражнения предлагаем читателю убедиться в том, что $TH^G(\alpha\beta) = TH^G(\alpha) + TH^G(\beta)$ и если $\alpha \Rightarrow^* \beta$, то $TH^G(\alpha) \leqslant TH^G(\beta)$.

Определим далее $TH_k^G(\alpha, w)$, где $\alpha \in (N \cup \Sigma)^*$ и $w \in \Sigma^{*k}$, как длину кратчайшей цепочки $x \in \Sigma^*$, для которой $\alpha \Rightarrow_G^* x$ и $w = FIRST_k(x)$. Если такой цепочки x нет, то значение $TH_k^G(\alpha, w)$ не определено. Когда ясно, о чем идет речь, символы k и G в обозначениях TH_k^G или TH^G будем опускать.

Алгоритм, проверяющий, эквивалентны ли две LL(k)-грамматики, основан на следующей лемме.

Лемма 8.5. *Пусть* $G_1=(N_1, \Sigma, P_1, S_1)$ *и* $G_1=(N_2, \Sigma, P_2, S_2)$ — *две* LL(k)*-грамматики в нормальной форме Грейбах и* $L(G_1)=L(G_2)$. *Тогда существует такая константа* p (*зависящая от* G_1 *и* G_2), *что если* $S_1 \Rightarrow_{G_1 l}^* w\alpha \Rightarrow_{G_1 l}^* wx$, $S_2 \Rightarrow_{G_2 l}^* w\beta \Rightarrow_{G_2 l}^* wy$, *где* α *и* β — *незаконченные части цепочек* $w\alpha$ *и* $w\beta$, *и* $FIRST_k(x) = FIRST_k(y)$, *то* $|TH^{G_1}(\alpha) - TH^{G_2}(\beta)| \leqslant p$ [2]).

[1]) В оригинале thickness.— *Прим. перев.*

[2]) Вертикальными прямыми | | здесь обозначена абсолютная величина числа, а не длина цепочки.

Доказательство. Пусть t — большее из чисел $TH^{G_1}(\gamma)$ и $TH^{G_2}(\gamma)$, где γ — правая часть правила из P_1 и P_2 соответственно. Положим $p=t(k+1)$ и допустим, что вопреки утверждению леммы

$$TH^{G_1}(\alpha) - TH^{G_2}(\beta) > p \tag{8.1.9}$$

Покажем, что при таком допущении $L(G_1) \neq L(G_2)$.

Обозначим $z = FIRST(x) = FIRST(y)$. Тогда $TH^{G_2}(\beta, z) \leqslant TH^{G_2}(\beta) + p$. Действительно, существует вывод $\beta \Rightarrow_{G_2 l}^* y$, и, следовательно, поскольку G_2 — грамматика в нормальной форме Грейбах, для некоторой цепочки δ найдется вывод $\beta \Rightarrow_{G_2 l}^* z\delta$, состоящий не более чем из k шагов. Легко показать, что

$$TH^{G_2}(\delta) \leqslant TH^{G_2}(\beta) + kt$$

так как „в худшем случае" δ — это цепочка β, к которой добавлены правые части k правил. Отсюда заключаем, что

$$TH^{G_2}(\beta, z) \leqslant k + TH^{G_2}(\delta) \leqslant TH^{G_2}(\beta) + p \tag{8.1.10}$$

Легко показать, что $TH^{G_1}(\alpha, z) \geqslant TH^{G_1}(\alpha)$, и потому из (8.1.9) и (8.1.10) вытекает, что

$$TH^{G_1}(\alpha, z) > TH^{G_2}(\beta, z) \tag{8.1.11}$$

Обозначим через u кратчайшую цепочку, выводимую из δ; тогда $TH^{G_2}(\beta, z) \leqslant |zu|$. Цепочка wzu принадлежит $L(G_2)$, так как $S \Rightarrow_{G_2}^* w\beta \Rightarrow_{G_2}^* wz\delta \Rightarrow_{G_2}^* wzu$. Но $\alpha \Rightarrow_{G_2}^* zu$ неверно, потому что, согласно (8.1.11), $TH^{G_1}(\alpha, z) > |zu|$. Так как G_1 — LL(k)-грамматика, то если в G_1 есть какой-то левый вывод цепочки wzu, он начинается с вывода $S_1 \Rightarrow_{G_1 l}^* w\alpha$. Таким образом, wzu не принадлежит $L(G_1)$ вопреки условию $L(G_1) = L(G_2)$. Отсюда получаем, что $TH^{G_1}(\alpha) - TH^{G_2}(\beta) \leqslant p = t(k+1)$. Случай $TH^{G_2}(\beta) - TH^{G_1}(\alpha) > p$ рассматривается симметрично. □

Лемма 8.6. *Для ДМП-автомата* P *проблема, допускает ли* P *все цепочки над своим алфавитом, разрешима.*

Доказательство. По теореме 2.23 дополнение $\overline{L(P)}$ языка $L(P)$ является детерминированным языком и, значит, КС-языком. Далее, можно эффективно построить такую КС-грамматику G, что $L(G) = \overline{L(P)}$. Для проверки равенства $L(G) = \varnothing$ можно применить алгоритм 2.7. Таким образом, можно узнать, допускает ли P все цепочки над своим входным алфавитом. □

Наконец, мы готовы к тому, чтобы описать алгоритм проверки двух LL(k)-грамматик на эквивалентность.

Теорема 8.6. *Для двух* LL(k)*-грамматик* $G_1=(N_1, \Sigma_1, P_1, S_1)$ *и* $G_2=(N_2, \Sigma_2, P_2, S_2)$ *проблема* „$L(G_1)=L(G_2)$?" *разрешима.*

Доказательство. Сначала построим по алгоритму 8.3 грамматики G_1' и G_2' в нормальной форме Грейбах, эквивалентные G_1 и G_2 соответственно (за исключением, возможно, пустой цепочки, которую несложно учесть). Затем построим ДМП-автомат P, допускающий входную цепочку w из $(\Sigma_1 \cup \Sigma_2)^*$ тогда и только тогда, когда

(1) w принадлежит как $L(G_1)$, так и $L(G_2)$,

или

(2) w не принадлежит ни $L(G_1)$, ни $L(G_2)$.

Следовательно, $L(G_1) = L(G_2)$ тогда и только тогда, когда $L(P) = (\Sigma_1 \cup \Sigma_2)^*$. Для проверки этого условия можно воспользоваться леммой 8.6.

Таким образом, для завершения доказательства осталось показать, как строится ДМП-автомат P. Его магазин состоит

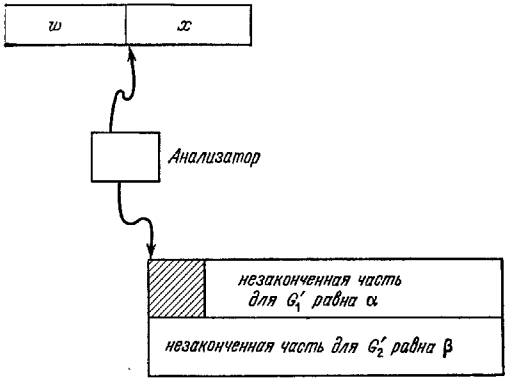


Рис. 8.6. Представление левовыводимых цепочек $w\alpha$ и $w\beta$.

из двух параллельных дорожек. P обрабатывает входную цепочку, разбирая ее одновременно в соответствии с G_1' и G_2'. Разбор проводится сверху вниз.

Предположим, что входная цепочка автомата P имеет вид wx. После выполнения $|w|$ шагов левого вывода в G_1' и в G_2' в его магазине будет записано содержимое каждого из магазинов k-предсказывающих анализаторов для G_1' и G_2', как показано на рис. 8.6. Из описания алгоритма 5.3 известно, что содержимое магазина в каждом случае представляет собой незаконченные части двух текущих левовыводимых цепочек, а также некоторую дополнительную информацию, записываемую вместе с нетерми-

налами для управления разбором. Можно считать, таким образом, что магазин содержит символы грамматик G_1' и G_2'. Дополнительная информация добавляется автоматически.

Заметим, что эти две незаконченные части могут занимать не одинаковый объем. Однако при $L(G_1) = L(G_2)$ мы можем, исходя из сказанного выше, считать, что разность их мощностей ограничена, а тогда P может моделировать оба вывода, производя считывание и запись в магазин на фиксированное число ячеек ниже его верхушки. Так как G_1' и G_2' — грамматики в нормальной форме Грейбах, P чередует шаги вывода в G_1' и G_2', а затем сдвигает свою входную головку на одну позицию. Если в одном из разборов достигается ошибочная конфигурация, то разбор ведется в соответствии с оставшейся грамматикой и продолжается до тех пор, пока не достигается ошибочная или допускающая конфигурация.

Необходимо только объяснить, как, предположив, что $L(G_1) = L(G_2)$, поместить в магазин обе незаконченные части так, чтобы они имели приблизительно одинаковую длину. По лемме 8.5 найдется такая константа p, что мощности двух незаконченных частей, возникающих при разборе префикса произвольной входной цепочки, различаются не более чем на p.

Для каждого символа грамматики мощности t автомат P резервирует t ячеек на соответствующей дорожке своего магазина, помещая этот символ в одну из них. Поскольку G_1' и G_2' — грамматики в нормальной форме Грейбах, ни в одной из них нет исчезающих символов и в любом случае $t \geqslant 1$. Так как две цепочки α и β, показанные на рис. 8.6, различаются по мощности самое большее на p, их представления в магазине автомата P различаются по длине самое большее на p ячеек.

Для завершения доказательства примем, что P отвергает входную цепочку, если две незаконченные части, записанные в его магазин, различаются по мощности более чем на p. По лемме 8.5 в этом случае $L(G_1) \neq L(G_2)$. Кроме того, если мощности никогда не различаются более чем на p, автомат P допускает входную цепочку тогда и только тогда, когда он находит разбор входной цепочки как в соответствии с грамматикой G_1', так и в соответствии с грамматикой G_2' или же не находит его ни в G_1', ни в G_2'. Таким образом, P допускает все цепочки над своим входным алфавитом тогда и только тогда, когда $L(G_1) = L(G_2)$. □

8.1.4. Иерархия LL-языков

Покажем, что LL(k)-языки для каждого $k \geqslant 0$ образуют собственное подмножество множества LL($k+1$)-языков. Как мы увидим, эта ситуация прямо противоположна ситуации с LR-языками, где для каждого LR-языка можно найти LR(1)-грамматику.

Рассмотрим последовательность языков $L_1, L_2, \ldots, L_k, \ldots$, где

$$L_k = \{a^n w \mid n \geqslant 1 \text{ и } w \in \{b, c, b^k d\}^n\}$$

В этом разделе мы покажем, что L_k является LL(k)-языком и не является LL($k-1$)-языком, и тем самым докажем существование бесконечной последовательности вложенных классов LL(k)-языков. Язык L_k порождается LL(k)-грамматикой

$$\begin{aligned} S &\to aT \\ T &\to SA \mid A \\ A &\to bB \mid c \\ B &\to b^{k-1}d \mid e \end{aligned}$$

Докажем, что каждая LL(k)-грамматика, порождающая язык L_k, должна содержать хотя бы одно e-правило.

Лемма 8.7. *Язык L_k не порождается никакой* LL(k)-*грамматикой без e-правил.*

Доказательство. Допустим противное. Тогда, воспользовавшись шагами (2)—(4) алгоритма 8.3, можно найти такую LL(k)-грамматику $G = (\mathrm{N}, \{a, b, c, d\}, P, S)$ в нормальной форме Грейбах, что $L(G) = L_k$. Докажем, что всякая такая грамматика порождает слова, не принадлежащие L_k.

Рассмотрим такую последовательность цепочек α_i, $i = 1, 2, \ldots$, что

$$S \Rightarrow_l^i a^i \alpha_i \Rightarrow_l^{i-1} a^{i+k-1}\delta$$

для некоторой цепочки δ. Так как G—LL(k)-грамматика в нормальной форме Грейбах, то α_i для каждого i определяется однозначно. В самом деле, если это не так, положим $\alpha_i = \alpha_j$. Тогда легко показать, что $a^{i+k}b^{j+k} \in L(G)$, а этого не может быть при $i \neq j$. Значит, можно найти такое i, что $|\alpha_i| \geqslant 2k-1$.

Выберем такое значение i, что $\alpha_i = \beta B \gamma$ для некоторых β и γ из N^* и $B \in \mathrm{N}$, причем $|\beta|$ и $|\gamma|$ равны по меньшей мере $k-1$. Поскольку G—LL(k)-грамматика, вывод слова $a^{i+k-1}b^{i+k-1}$ имеет вид

$$S \Rightarrow_l^* a^i \beta B \gamma \Rightarrow_l^* a^{i+k-1}b^{i+k-1}$$

Так как G—грамматика в нормальной форме Грейбах и $|\beta| \geqslant k-1$, то $\beta \Rightarrow^* a^{k-1}b^j$, $B \Rightarrow^* b^l$ и $\gamma \Rightarrow^* b^m$ для некоторых $j \geqslant 0$, $l \geqslant 1$ и $m \geqslant k-1$, причем $i+k-1 = j+l+m$.

Если рассмотреть вывод слова $a^{i+k-1}c^{i+k-1}$, можно также заключить, что $B \Rightarrow^* c^n$ для некоторого $n \geqslant 1$.

Наконец, рассмотрев вывод слова $a^{i+k-1}b^{j+l+k-1}db^m$, находим, что

$$S \Rightarrow_l^* a^i \beta B \gamma \Rightarrow_l^* a^{i+k-1}b^j B \gamma \Rightarrow_l^* a^{i+k-1}b^{j+l}\gamma \Rightarrow_l^* a^{i+k-1}b^{j+l+k-1}db^m$$

Существование последнего вывода вытекает из того, что у слов $a^{i+k-1}b^{j+l+k-1}db^m$ и $a^{i+k-1}b^{i+k-1}$ совпадают $(i+k-1) + (j+l+k-1)$ символов. Поэтому $\gamma \Rightarrow^* b^{k-1}db^m$.

Объединяя эти частичные выводы, получаем вывод

$$S \Rightarrow_l^* a^i \beta B \gamma \Rightarrow_l^* a^{i+k-1}b^j B \gamma \Rightarrow_l^* a^{i+k-1}b^j c^n \gamma \Rightarrow_l^* a^{i+k-1}b^j c^n b^{k-1}db^m$$

Но его результат не принадлежит L_k, так как он содержит подцепочку вида $cb^{k-1}d$. (Перед d должно стоять k символов b.) Итак, язык L_k не порождается никакой LL(k)-грамматикой без e-правил. □

Покажем теперь, что если язык L порождается LL(k)-грамматикой без e-правил, то он является LL($k-1$) языком.

Теорема 8.7. *Если язык L порождается* LL(k)-*грамматикой без e-правил, $k \geqslant 2$, то он порождается* LL($k-1$)-*грамматикой.*

Доказательство. В соответствии с шагом (3) алгоритма 8.3 для L можно найти LL(k)-грамматику в нормальной форме Грейбах. Пусть $G = (\mathrm{N}, \Sigma, P, S)$—такая грамматика. По ней можно построить LL($k-1$)-грамматику $G_1 = (\mathrm{N}_1, \Sigma, P_1, S)$, где

(1) $\mathrm{N}_1 = \mathrm{N} \cup \{[A, a] \mid A \in \mathrm{N},\ a \in \Sigma \text{ и } A \to a\alpha \in P$ для некоторой цепочки $\alpha\}$,

(2) $P_1 = \{A \to a[A, a] \mid A \to a\alpha \in P\} \cup \{[A, a] \to \alpha \mid A \to a\alpha \in P\}$.

В качестве упражнения предлагаем доказать, что G_1—LL($k-1$)-грамматика. Заметим, что приведенное здесь построение служит примером левой факторизации. □

Пример 8.3. Рассмотрим естественную LL($k+1$)-грамматику G_k для языка L_k из леммы 8.7:

$$\begin{aligned} S &\to aSA \mid aA \\ A &\to b^k d \mid b \mid c \end{aligned}$$

Построим для L_k LL(k)-грамматику G'_k, добавив новые символы $[S, a]$, $[A, b]$ и $[A, c]$. Правила грамматики G'_k таковы:

$$\begin{aligned} S &\to a[S, a] \\ A &\to b[A, b] \mid c[A, c] \\ [S, a] &\to SA \mid A \\ [A, b] &\to b^{k-1}d \mid e \\ [A, c] &\to e \end{aligned}$$

В качестве упражнения предлагаем доказать, что G_k—LL($k+1$)-грамматика, а G'_k—LL(k)-грамматика. □

Теорема 8.8. *Для всех $k \geqslant 1$ класс* LL($k-1$)-*языков содержится строго внутри класса* LL(k)-*языков.*

Доказательство. Ясно, что LL(0)-языки образуют собственное подмножество множества LL(1)-языков.

По лемме 8.7 язык L_k для $k>1$ не порождается никакой LL(k)-грамматикой без e-правил. Следовательно, по теореме 8.4 он не порождается никакой LL($k-1$)-грамматикой. Однако, как мы видели, он порождается LL(k)-грамматикой (с e-правилами). □

УПРАЖНЕНИЯ

8.1.1. Приведите дополнительные примеры грамматик, являющихся

(а) LR и левоанализируемыми (детерминированно), но не LL,
(б) правоанализируемыми и левоанализируемыми, но не LR,
(в) LR, но не левоанализируемыми.

8.1.2. Преобразуйте LL(1)-грамматику

$$\begin{aligned} E &\to TE' \\ E' &\to +TE' \mid e \\ T &\to FT' \\ T' &\to *FT' \mid e \\ F &\to a \mid (E) \end{aligned}$$

в LL(2)-грамматику без e-правил.

8.1.3. Преобразуйте грамматику из упр. 8.1.2 в LL(2)-грамматику в нормальной форме Грейбах.

8.1.4. Докажите утверждение (1) теоремы 8.3.

8.1.5. Дополните доказательство леммы 8.4.

8.1.6. Дополните доказательство теоремы 8.5.

8.1.7. Покажите, что грамматика G_1, построенная в доказательстве теоремы 8.7, является LL($k-1$)-грамматикой.

8.1.8. Покажите, что язык является LL(0)-языком тогда и только тогда, когда он либо пуст, либо состоит из одного элемента.

8.1.9. Покажите, что грамматики G_k и G_k' из примера 8.3 являются LL($k+1$)- и LL(k)-грамматикой соответственно.

***8.1.10.** Докажите, что каждая из грамматик в теореме 8.2 обладает приписанными ей там свойствами.

***8.1.11.** Покажите, что язык $L=\{a^nb^n \mid n\geqslant 1\} \cup \{a^nc^n \mid n\geqslant 1\}$ детерминированный, но не LL. *Указание:* Предположите, что L порождается LL(k)-грамматикой G в нормальной форме Грейбах. Покажите, что язык $L(G)$ должен содержать цепочки, не входящие в L, рассмотрев для этого левовыводимые цепочки вида $a^i\alpha$ для $i\geqslant 1$.

8.1.12. Покажите, что язык $L=\{a^nb^m \mid 1\leqslant m\leqslant n\}$ детерминированный, но не LL. Обратите внимание на то, что L представляет собой конкатенацию двух LL(1)-языков: a^* и $\{a^nb^n \mid n\geqslant 1\}$.

***8.1.13.** Покажите, что каждый LL(k)-язык порождается LL($k+1$)-грамматикой в нормальной форме Хомского.

****8.1.14.** Пусть $L=L_1 \cup L_2 \cup \ldots \cup L_m$, где L_i для каждого $1\leqslant i\leqslant m$ является LL-языком. Покажите, что если язык L регулярен, то регулярны все L_i.

****8.1.15.** Покажите, что если L — LL-язык, но не регулярный, то L — не LL-язык.

***8.1.16.** Покажите, что множество LL-языков не замкнуто относительно объединения, пересечения, дополнения, конкатенации, обращения и e-свободного гомоморфизма. *Указание:* См. упр. 8.1.11, 8.1.12 и 8.1.15.

8.1.17. Докажите, что $\mathrm{TH}^G(\alpha\beta)=\mathrm{TH}^G(\alpha)+\mathrm{TH}^G(\beta)$, и если $\alpha\Rightarrow^*\beta$, то $\mathrm{TH}^G(\alpha)\leqslant \mathrm{TH}^G(\beta)$.

8.1.18. Дайте алгоритмы вычисления $\mathrm{TH}^G(\alpha)$ и $\mathrm{TH}^G(\alpha, z)$.

8.1.19. Покажите, что грамматика G_1 из алгоритма 8.1 левопокрывает грамматику G из этого же алгоритма.

8.1.20. Покажите, что каждая LL(k)-грамматика G левопокрывается некоторой LL($k+1$)-грамматикой в нормальной форме Грейбах.

8.1.21. Покажите, что для $k\geqslant 2$ каждая LL(k)-грамматика без e-правил левопокрывается некоторой LL($k-1$)-грамматикой.

****8.1.22.** Покажите, что проблема существования для данной LR(k)-грамматики G такого числа k', что G является LL(k')-грамматикой, разрешима.

Замечания по литературе

Теорема 8.1 предложена Кнутом [1967]. Результаты разд. 8.1.2 и 8.1.3 впервые появились в работе Розенкранца и Стирнза [1970]. Там же приведены решения задач, сформулированных в упр. 8.1.14—8.1.16 и 8.1.22. Иерархия LL(k)-языков отмечена впервые Курки-Суонио [1969].

Результаты о разрешимости, связанные с теоремой 8.6, содержатся в более ранних статьях. Корсиьяк и Хопкрофт [1966] доказали разрешимость проблемы эквивалентности двух простых LL(1)-грамматик. Мак-Нотон [1967] показал, что проблема эквивалентности разрешима для *скобочных грамматик*, представляющих собой грамматики, в которых правые части всех правил заключены в круглые скобки, не встречающиеся нигде внутри правил. Независимо от него Паул и Ангер [1968а] показали, что разрешима проблема

структурной эквивалентности двух грамматик, понимая под этим, что эквивалентные грамматики порождают одни и те же цепочки, а соответствующие деревья разбора совпадают во всем, кроме меток. (Две грамматики структурно эквивалентны тогда и только тогда, когда построенные по ним скобочные грамматики эквивалентны.)

8.2. КЛАССЫ ГРАММАТИК, ПОРОЖДАЮЩИЕ ДЕТЕРМИНИРОВАННЫЕ ЯЗЫКИ

В настоящем разделе мы увидим, какие классы грамматик порождают в точности детерминированные языки. Среди них — классы LR(1)-грамматик, (1,1)-ОПК-грамматик, простых ССП-грамматик и обратимых грамматик (2,1)-предшествования. Кроме того, если детерминированный язык обладает префиксным свойством, то он порождается LR(0)- или (1,0)-ОПК-грамматикой. Заметим, что для любого языка можно сделать так, чтобы он обладал префиксным свойством: достаточно добавить к каждому слову данного языка концевой маркер.

8.2.1. ДМП-автоматы в нормальной форме и канонические грамматики

Общая стратегия разд. 8.2 заключается в построении грамматик по ДМП-автоматам, обладающим специальными свойствами. Эти грамматики или простые их модификации принадлежат упомянутым выше классам. Определим вначале специальные свойства, наличие которых у ДМП-автомата для нас желательно.

Определение. ДМП-автомат $P=(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ называется ДМП-*автоматом в нормальной форме*, если он обладает следующими свойствами:

(1) P — автомат без циклов. Это означает, что при любой входной цепочке P делает только ограниченное число шагов.

(2) F состоит из единственного состояния q_f, и если $(q_0, w, Z_0) \vdash^* (q_f, e, \gamma)$, то $\gamma = Z_0$. Другими словами, когда P допускает входную цепочку, он находится в заключительном состоянии q_f, а в магазине содержится только начальный символ.

(3) Q можно представить в виде $Q = Q_s \cup Q_w \cup Q_e \cup \{q_f\}$, где Q_s, Q_w и Q_e — попарно непересекающиеся множества, называемые множествами состояний *чтения*, *записи* и *стирания* соответственно, и q_f не принадлежит ни одному из них. Состояния обладают такими свойствами:

(а) Если $q \in Q_s$, то для каждого $a \in \Sigma$ существует такое состояние p_a, что $\delta(q, a, Z) = (p_a, Z)$ для всех Z. Таким образом, если P находится в состоянии чтения, то следующий такт состоит в чтении входного символа. Кроме того, этот такт никогда не зависит от символа, записанного в верхушке магазина.

(б) Если $q \in Q_w$, то $\delta(q, e, Z) = (p, YZ)$ для некоторых p и Y и для всех Z. В состоянии записи автомат всегда пишет в верхушку магазина новый символ, причем этот такт не зависит от текущего входного символа и от символа, содержавшегося в верхушке магазина.

(в) Если $q \in Q_e$, то для каждого $Z \in \Gamma$ существует такое состояние p_Z, что $\delta(q, e, Z) = (p_Z, e)$. В состоянии стирания автомат всегда убирает из магазина самый верхний символ, не считывая нового входного символа.

(г) $\delta(q_f, a, Z) = \varnothing$ для всех a из $\Sigma \cup \{e\}$ и $Z \in \Gamma$. В заключительном состоянии никакие переходы невозможны.

(4) Если $(q, w, Z) \vdash^+ (p, e, Z)$, то $w \neq e$. Другими словами, последовательность тактов, в результате которой длина цепочки, содержащейся в магазине, (возможно) увеличивается, а затем вновь становится прежней, не может реализоваться при пустой входной цепочке. Последовательность тактов $(q, w, Z) \vdash^+ (p, e, Z)$ называется *траверсом*. Заметим, что возможность или невозможность траверса для данных q, p и w не зависит от символа Z, записанного в верхушке магазина.

Короче говоря, в состоянии чтения автомат считывает следующий входной символ, в состоянии записи пишет новый символ в магазин, а в состоянии стирания исследует верхний магазинный символ и затем стирает его. Входная головка сдвигается только тогда, когда автомат находится в состоянии чтения.

Теорема 8.9. *Если $L \subseteq \Sigma^*$ — детерминированный язык и символ $¢$ не принадлежит Σ, то $L¢ = L(P)$ для некоторого ДМП-автомата P в нормальной форме.*

Доказательство. Построим последовательность из шести ДМП-автоматов $P_1 - P_6$, получая P_{i+1} по P_i так, чтобы у P_{i+1} было больше свойств ДМП-автомата в нормальной форме, чем у P_i. В результате P_6 будет нужным ДМП-автоматом в нормальной форме.

Сначала с помощью леммы 2.28 построим дочитывающий, а значит, и бесцикловый ДМП-автомат P_1, для которого $L = L(P_1)$. Затем преобразуем P_1 в P_2, воспользовавшись конструкцией из леммы 2.21 и считая P_1 расширенным ДМП-автоматом. ДМП-автомат P_2 будет дочитывающим и в каждом состоянии либо будет действовать как в состоянии стирания или записи, либо не будет изменять магазин, но в любом состоянии автомата P_2 будет возможен сдвиг входной цепочки (т. е. чтение).

Далее по теореме 2.23 получим P_3 из P_2. Новый автомат будет обладать всеми свойствами автомата P_2 и, кроме того, не будет делать e-тактов в заключительном состоянии.

Затем построим по P_3 автомат P_4 так, чтобы у него были все упомянутые свойства автомата P_3, но чтобы он допускал язык $L¢$, где $¢$ не принадлежит алфавиту языка L. У автомата P_4 будет единственное заключительное состояние q_f, и он будет допускать входную цепочку только тогда, когда в магазине находится лишь начальный символ. P_4 имеет два нижних маркера: Z_0 и Z_1. Z_0 служит начальным символом и за первые два такта P_4 записывает над Z_0 символ Z_1 и над Z_1 начальный символ автомата P_3. Потом P_4 моделирует P_3. Если P_3 допускает входную цепочку, то при входном символе $¢$ автомат P_4 переходит в новое состояние q_e и стирает все содержимое магазина вплоть до Z_1. Затем P_4 стирает Z_1, переходит в состояние q_f и останавливается.

Для того чтобы сделать P_4 автоматом в нормальной форме, осталось

(1) отделить операции чтения входной цепочки от операций с магазином,

(2) исключить траверсы для пустой входной цепочки.

Чтобы удовлетворить требованию (1), образуем из P_4 автомат P_5. Для каждого состояния q автомата P_4, в котором невозможен e-такт, кроме $q=q_f$, построим новые состояния q_a для всех a из Σ. Из состояния q при входном символе a происходит переход в состояние q_a, после чего P_5 в состоянии q_a при входе e выполняет тот шаг, который автомат P_4 выполнял в состоянии q при входе a.

Наконец, видим, что проблема, верно ли, что $(q,e,Z)\vdash^+_{P_5}(p,e,Z)$, разрешима для всех q и p. Из конструкции автомата P_2 вытекает, что разрешимость этой проблемы не зависит от Z. Построение алгоритма, решающего такую задачу, оставляем в качестве упражнения. Отметим, что все построенные ДМП-автоматы, включая и P_5, не имеют циклов. Следовательно, для каждого состояния q существует такое единственное состояние q' (возможно, $q'=q$), что $(q, e, Z)\vdash^*(q', e, Z)$, но ни для какого q'' неверно, что $(q', e, Z)\vdash^+(q'', e, Z)$. Последний ДМП-автомат P_6 в нашей последовательности построим из автомата P_5, приписывая состоянию q все переходы, которые можно сделать из q' в таких ситуациях. P_6 и будет нужным ДМП-автоматом P.

Детальное построение, опирающееся на эти интуитивные соображения, оставляем в качестве упражнений. □

Изложим теперь метод построения по ДМП-автомату в нормальной форме грамматики, которую мы назовем канонической. Этот метод отличается от метода, использованного в лемме 2.26, тем, что здесь мы опираемся на специальные свойства ДМП-автомата в нормальной форме.

Определение. Пусть $M=(Q, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_f\})$ — ДМП-автомат в нормальной форме, у которого верхушка магазина расположена слева. *Канонической грамматикой* для P называется грамматика $G=(\mathrm{N}, \Sigma, P, [q_0\ q_f])$, где

(1) N' — множество таких пар $[qp]$ из $Q\times Q$, что q — состояние чтения или записи, а p — произвольное состояние. Из нетерминала $[qp]$ выводятся в точности такие терминальные цепочки w, что, получая их на вход, автомат M делает траверс из состояния q в состояние p. Другими словами, $[qp]\Rightarrow^* w$ тогда и только тогда, когда $(q, w, Z)\vdash^+(p, e, Z)$ для всех $Z\in\Gamma$.

(2) Множество правил P' строится так:

(а) Если $\delta(q, a, Z)=(q', Z)$, включим в P' правило

$$[qq']\to a$$

Кроме того, для всех $r\in Q_s\cup Q_w$ включим в P' правила

$$[rq']\to[rq]a$$

Заметим, что здесь q — состояние чтения.

(б) Если $\delta(q, e, Z)=(s, YZ)$ и $\delta(p, e, Y)=(q', e)$, включим в P' правило

$$[qq']\to[sp]$$

и для всех $r\in Q_s\cup Q_w$ — правила

$$[rq']\to[rq][sp]$$

Здесь q — состояние записи, а p — состояние стирания.

(3) Множества N и P образуются в результате исключения из N' и P' соответственно бесполезных нетерминалов и правил. Правила в P будут иметь вид

$$\begin{array}{ll}(1) & [qq']\to a\\ (2) & [qq']\to[pp']a\\ (3) & [qq']\to[pp']\\ (4) & [qq']\to[pp'][rr']\end{array}$$

Назовем правило, имеющее в этом списке номер (i), правилом *типа* i, $1\leqslant i\leqslant 4$.

Заметим, что в канонических грамматиках правила обладают такими свойствами:

(1) если $[qq']\to a\in P$, то q — состояние чтения,

(2) если $[qq']\to[pp']a\in P$, то p' — состояние чтения,

(3) если $[qq']\to[pp']\in P$, то q — состояние записи, а p' — состояние стирания,

(4) если $[qq']\to[pp'][rr']\in P$, то p' — состояние записи, а r' — состояние стирания.

Полезно также заметить следующее. Пусть q — произвольное состояние записи автомата M. В состоянии q автомат M, прежде чем считать следующий входной символ, записывает в магазин только фиксированное число символов. Другими словами, существует такая конечная последовательность состояний $q_1, \ldots, q_k$, что $q_1 = q$, $\delta(q_i, e, Z) = (q_{i+1}, Y_iZ)$ для $1 \leqslant i < k$ и всех Z, причем q_k — состояние чтения. Последовательность не содержит повторяющихся элементов, и, возможно, $k = 1$. Основанием для такого утверждения служит тот факт, что если бы в последовательности были повторяющиеся элементы, то автомат M не был бы бесцикловым; если длина последовательности больше $\# Q$, то последовательность должна содержать повторения. Назовем эту последовательность состояний *пишущей последовательностью для состояния* q.

Теорема 8.10. *Если* $G = (\mathrm{N}, \Sigma, P, S)$ — *каноническая грамматика, построенная по* ДМП-*автомату в нормальной форме* $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_f\})$, *то* $L(G) = L(M) - \{e\}$.

Доказательство. Покажем, что символ $[qq']$ порождает точно такие входные цепочки, для которых возможен траверс из q в q'. Для этого докажем индукцией, что

(8.2.1) $[qq'] \Rightarrow^n w$ для некоторого n и $w \neq e$ тогда и только тогда, когда $(q, w, Z) \vdash^m (q', e, Z)$ для некоторого $m > 0$ и любого Z.

Достаточность. Базис, $m = 1$, проверяется тривиально. В этом случае цепочка w должна быть символом из Σ, а в P должно быть правило $[qq'] \to w$. Для проведения шага индукции предположим, что (8.2.1) истинно для значений, меньших m, и что $(q, w, Z) \vdash^m (q', e, Z)$. Тогда конфигурация, непосредственно предшествующая (q', e, Z), должна иметь вид (p, a, Z) или (p, e, YZ).

В первом случае p — состояние чтения и $(q, w, Z) \vdash^{m-1} (p, a, Z)$. Следовательно, $(q, w', Z) \vdash^{m-1} (p, e, Z)$, если $w'a = w$. Согласно предположению индукции, $[qp] \Rightarrow^* w'$. По определению грамматики G правило $[qq'] \to [qp]a$ принадлежит P, и потому $[qq'] \Rightarrow^* w$.

Во втором случае надо цепочку w представить в виде w_1w_2 и найти такие состояния r и s, что

$$\begin{aligned}(q, w_1w_2, Z) &\vdash^{m_1} (r, w_2, Z)\\ &\vdash (s, w_2, YZ)\\ &\vdash^{m_2} (p, e, YZ)\\ &\vdash (q', e, Z)\end{aligned}$$

где $m_1 < m$, $m_2 < m$ и последовательность переходов $(s, w_2, YZ) \vdash^{m_2} (p, e, YZ)$ никогда не стирает выделенный символ Y. Если

$m_1 = 0$, то $r = q$ и $w_2 = w$. Таким образом, правило $[qq'] \to [sp]$ принадлежит P. Из вида шагов автомата заключаем, что $(s, w_2, Y) \vdash^{m_2} (p, e, Y)$ и, значит, $[sp] \Rightarrow^* w_2$. Итак, $[qq'] \Rightarrow^* w$.

Если $m_1 > 0$, то $(q, w_1, Z) \vdash^{m_1} (r, e, Z)$ и, значит, согласно предположению, $[qr] \Rightarrow^* w_1$. Как и раньше, $[sp] \Rightarrow^* w_2$. Грамматика G устроена так, что правило $[qq'] \to [qr][sp]$ принадлежит P; следовательно, $[qq'] \Rightarrow^* w$.

Необходимость. Эту часть легко доказать индукцией; мы предоставляем это читателю.

Наша теорема — частный случай утверждения (8.2.1) при $q = q_0$ и $q' = q_f$. □

Следствие 1. *Если* L — *детерминированный язык, обладающий префиксным свойством, и* $e \notin L$ [1]), *то* L *порождается канонической грамматикой.*

Доказательство. Метод построения для такого языка ДМП-автомата в нормальной форме аналогичен методу, описанному в теореме 8.9. □

Следствие 2. *Если* $L \subseteq \Sigma^*$ — *детерминированный язык и* $¢ \notin \Sigma$, *то* $L¢$ *порождается канонической грамматикой.* □

8.2.2. Простые ССП-грамматики и детерминированные языки

Приступим теперь к доказательству того, что каждая каноническая грамматика является простой ССП-грамматикой. Напомним, что простая ССП-грамматика — это такая грамматика (не обязательно обратимая) слабого предшествования $G = (\mathrm{N}, \Sigma, P, S)$, что если $A \to \alpha$ и $B \to \alpha$ принадлежат P, то $l(A) \cap l(B) = \varnothing$, где $l(C)$ — множество нетерминальных или терминальных символов, которые могут появиться непосредственно слева от C в правовыводимой цепочке, т. е. $l(C) = \{X \mid X \lessdot C$ или $X \doteq C\}$.

Начнем с того, что покажем, что каноническая грамматика является грамматикой (не обязательно обратимой) (1, 1)-предшествования.

Лемма 8.8. *Каноническая грамматика является приведенной грамматикой (т. е. не имеет бесполезных символов, e-правил и циклов).*

Доказательство. Метод построения канонической грамматики исключает бесполезные символы и e-правила. Достаточно показать, что в ней нет циклов. Цикл может быть только тогда,

[1]) Заметим, что если L обладает префиксным свойством и $e \in L$, то $L = \{e\}$.

когда есть последовательность правил типа 3, скажем $[q_i q_i'] \to [q_{i+1} q_{i+1}']$, $1 \leqslant i < j$, где $[q_1 q_1'] = [q_j q_j']$. Но тогда из правил построения канонической грамматики вытекает, что пишущая последовательность для состояния q_1 начинается с $q_1, q_2, \ldots, q_j$ и, значит, содержит повторяющиеся элементы. Это в свою очередь означает, что соответствующий ДМП-автомат в нормальной форме имеет цикл. Отсюда заключаем, что на самом деле циклов в грамматике нет. □

Лемма 8.9. *Каноническая грамматика является (не обязательно обратимой) грамматикой* (1, 1)-*предшествования*[1]).

Доказательство. Пусть $G = (N, \Sigma, P, S)$ — каноническая грамматика. Рассмотрим три возможных конфликта отношений предшествования и покажем, что ни один из них не реализуется.

Случай 1: Предположим, что $X \lessdot Y$ и $X \doteq Y$. Поскольку $X \doteq Y$, должно быть правило $A \to XY$ типа 2 или 4. Поэтому $X = [qq']$ и либо $Y \in \Sigma$ и q' — состояние чтения, либо $Y = [pp']$ и p' — состояние стирания.

Поскольку $X \lessdot Y$, должно быть правило $B \to XC$ типа 4, где $C \Rightarrow^+ Y\alpha$ для некоторой цепочки α. Пусть, как и раньше, $X = [qq']$. Тогда q' должно быть состоянием записи, потому что $B \to XC$ — правило типа 4. Более того, Y должен иметь вид $[pp']$, где p' — состояние стирания, и, значит, $A \to XY$ — правило типа 4. Из вида правил типа 4 можно заключить, что p — второе состояние в пишущей последовательности для q'. Так как $B \to XC$ — правило типа 4, то $C = [pp'']$ для некоторого p''.

Рассмотрим теперь вывод $C \Rightarrow_l^+ Y\alpha$, который можно записать в виде

$$[s_1 s_1'] \Rightarrow_l [s_2 s_2'] \alpha_2 \Rightarrow_l \ldots \Rightarrow_l [s_n s_n'] \alpha_n$$

где $[s_1 s_1'] = [pp'']$ и $[s_n s_n'] = [pp']$. Из вида правил заключаем, что для каждого i либо $s_{i+1} = s_i$ (если $[s_i s_i']$ заменяется по правилу типа 2 или 4), либо s_{i+1} — состояние, которое в пишущей последовательности для q' следует за состоянием s_i (если $[s_i s_i']$ заменяется по правилу типа 3). Только в последнем случае s_{i+1}' будет состоянием стирания, и, значит (поскольку $s_n' (= p')$ есть состояние стирания), $s_n (= p)$ в пишущей последовательности для q' следует за s_{n-1}. Так как s_{n-1} либо есть p, либо следует за p в этой последовательности, то p появляется в пишущей последовательности для q' дважды. Это означает существование цикла, и мы заключаем, что в канонической грамматике конфликтов между отношениями $\lessdot$ и $\doteq$ нет.

[1]) Для того чтобы доказать, что каноническая грамматика является простой ССП-грамматикой, достаточно показать, что она — грамматика слабого предшествования (а не (1,1)-предшествования). Однако это дополнительное условие представляет определенный интерес, а доказательства не усложняет.

Случай 2: $X \lessdot Y$ и $X \gtrdot Y$. Поскольку $X \lessdot Y$, мы, как и в случае 1, выводим, что $X = [qq']$, где q' — состояние записи. С другой стороны, если $X \gtrdot Y$, то есть правило $A \to BZ$, где $B \Rightarrow^+ \alpha X$ и $Z \Rightarrow^+ Y\beta$. Вид правил таков, что если $B \Rightarrow^+ \alpha[qq']$, то q' — состояние стирания. Но мы уже нашли, что q' — состояние записи. Отсюда заключаем, что конфликтов между $\lessdot$ и $\gtrdot$ нет.

Случай 3: $X \doteq Y$ и $X \gtrdot Y$. Поскольку $X \doteq Y$, мы, как и в случае 1, выводим, что $X = [qq']$, где q' — состояние чтения или записи. Но из $X \gtrdot Y$, как в случае 2, заключаем, что q' — состояние стирания.

Таким образом, каноническая грамматика является грамматикой предшествования. □

Теорема 8.11. *Каноническая грамматика является простой грамматикой со смешанной стратегией предшествования.*

Доказательство. В соответствии с теоремой 8.10 и леммами 8.8 и 8.9 достаточно показать, что для каждой канонической грамматики $G = (N, \Sigma, P, S)$

(1) если $A \to \alpha XY\beta$ и $B \to Y\beta$ принадлежат P, то $X \notin l(B)$,

(2) если $A \to \alpha$ и $B \to \alpha$ принадлежат P, $A \neq B$, то $l(A) \cap l(B) = \varnothing$.

Справедливость утверждения (1) сразу следует из того, что если $X \lessdot B$ или $X \doteq B$, то $X \lessdot Y$. Но если существует правило $A \to \alpha XY\beta$, то $X \doteq Y$, и потому есть конфликт отношений предшествования, что противоречит лемме 8.9.

Рассмотрим теперь утверждение (2). $A \to \alpha$ и $B \to \alpha$ не могут быть различными правилами типа 2, так как если $A = [qq']$, $B = [pp']$, $\alpha = [rr'] a$ и грамматика G построена по ДМП-автомату $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \{q_f\})$, то

$$\delta(r', a, Z) = (q', Z) = (p', Z)$$

Следовательно, $q' = p'$. Но из вида правил типа 2 вытекает, что $q = p = r$, так что $A = B$ вопреки условию. Аналогичное рассуждение, которое предлагаем читателю в качестве упражнения, показывает, что $A \to \alpha$ и $B \to \alpha$ не могут быть различными правилами типа 4.

Разберем теперь случай $\alpha = a$, т. е. случай, когда $A \to \alpha$ и $B \to \alpha$ — правила типа 1. Вообще, если $X \lessdot Y$ или $X \doteq Y$, то, как мы видели (доказательство теоремы 8.9, случай 1), X должен быть нетерминалом. Итак, допустим, что C принадлежит и $l(A)$, и $l(B)$. Тогда существуют правила $D_1 \to CD_2$ и $E_1 \to CE_2$, где $D_2 \Rightarrow^* A\beta$ и $E_2 \Rightarrow^* B\gamma$. Случай $A = D_2$ или $B = E_2$ не исключается. Положим $C = [qq']$, $A = [pp']$ и $B = [rr']$. Тогда p и r — состояния чтения, так как $[pp'] \to a$ и $[rr'] \to a$ — правила типа 1. Согласно предыдущему рассуждению, каждое из состояний p и r должно появиться в пишущей последовательности для q' и,

следовательно, должно заканчивать эту последовательность. Значит, $p=r$. Так как

$$\delta(p, a, Z)=(p', Z)=(r', Z)$$

то $p'=r'$, $A=B$ вопреки условию. Таким образом, $A\to\alpha$ и $B\to\alpha$ не могут быть правилами типа 1.

Наконец, предположим, что $A\to\alpha$ и $B\to\alpha$ — правила типа 3. Как и ранее, пусть C принадлежит $l(A)\cap l(B)$, $C=[qq']$, $A=[pp']$ и $B=[rr']$. Тогда каждое из состояний p и r должно появиться в пишущей последовательности для q'. Если $p=r$, положим $\delta(p, e, Z)=(s, Y)$. В этом случае $\alpha=[ss']$ для некоторого s' и $\delta(s', e, Y)=(p', e)=(r', e)$. Следовательно, $r'=p'$ и опять $A=B$ вопреки условию.

Пусть тогда $p\neq r$. Если $\alpha=[ss']$, то s стоит в пишущей последовательности для q' и после p, и после r, а, значит, появляется там дважды. Отсюда выводим, что $A\to\alpha$ и $B\to\alpha$ не являются правилами типа 3. Таким образом, утверждение (2) верно, и G — простая ССП-грамматика. □

Из теоремы 8.11 следует, что язык $L¢$ порождается простой ССП-грамматикой, если L — детерминированный язык, а ¢ — концевой маркер. На самом деле верно более сильное утверждение: если ¢ убрать с конца каждой цепочки, порождаемой канонической грамматикой, то измененная таким образом грамматика останется простой ССП-грамматикой. Учитывая, что построение, убирающее концевые маркеры, будет проводиться неоднократно, мы представим его здесь в виде алгоритма.

Алгоритм 8.4. Удаление правого концевого маркера из слов, порождаемых приведенной грамматикой.

Вход. Приведенная грамматика $G=(N, \Sigma\cup\{¢\}, P, S)$, где $¢\notin\Sigma$ и $L(G)=L¢$ для некоторого $L\subseteq\Sigma^+$.

Выход. Грамматика $G_1=(N_1, \Sigma, P_1, S)$, для которой $L(G_1)=L$.

Метод.

(1) Убрать из P все правила вида $A\to¢$.

(2) Заменить в P все правила вида $A\to\alpha¢$, где $\alpha\neq e$, на $A\to\alpha$.

(3) Если правило $A\to\alpha B$ принадлежит P, $\alpha\neq e$, и $B\overset{*}{\Rightarrow}_G ¢$, добавить к P правило $A\to\alpha$.

(4) Исключить из N и из полученного множества правил бесполезные нетерминалы и правила. Обозначить новые множества нетерминалов и правил N_1 и P_1 соответственно. □

Теорема 8.12. *Если G_1 — грамматика, построенная алгоритмом 8.4, то $L(G_1)=L$.*

Доказательство. Поскольку каждое слово w языка $L(G)$ имеет вид $x¢$ для $x\in\Sigma^+$, то для любого $A\in N$ из $A\overset{*}{\Rightarrow}_G u$ следует, что либо $u\in\Sigma^+$, либо $u=v¢$, где $v\in\Sigma^*$. Назовем нетерминалы первого типа *промежуточными*, а нетерминалы второго типа — *завершающими*. Простая индукция по длине выводов показывает, что если A — промежуточный нетерминал, то $A\overset{*}{\Rightarrow}_G u$ тогда и только тогда, когда $A\overset{*}{\Rightarrow}_{G_1} u$, $u\in\Sigma^*$. Аналогично, если A — завершающий нетерминал, то $A\overset{*}{\Rightarrow}_G v¢$ тогда и только тогда, когда $A\overset{*}{\Rightarrow}_{G_1} v$. Доказательство оставляем в качестве упражнения.

Итак, $S\overset{*}{\Rightarrow}_G w¢$ тогда и только тогда, когда $S\overset{*}{\Rightarrow}_{G_1} w$. Следовательно, $L(G_1)=L$. □

Теорема 8.13. *Если L — детерминированный язык и $e\notin L$, то L порождается простой ССП-грамматикой.*

Доказательство. Пусть $L\subseteq\Sigma^+$ и $¢\notin\Sigma$. Тогда, согласно следствию 2 теоремы 8.10, $L¢$ порождается канонической грамматикой $G=(N, \Sigma, P, S)$, которая по теореме 8.11 является простой ССП-грамматикой. Пусть $G_1=(N_1, \Sigma, P_1, S)$ — грамматика, построенная алгоритмом 8.4 из грамматики G. Тогда $L(G_1)=L$. Мы докажем, что G_1 — также простая ССП-грамматика.

Легко показать, что G_1 не имеет e-правил и бесполезных символов. Если бы грамматика G_1 содержала цикл, в ней было бы правило $A\to B$, входящее в P_1, но не в P. Допустим, что правило $A\to BX$ принадлежит P для некоторого X, для которого $X\overset{+}{\Rightarrow}_G ¢$. Предположим далее, что $A\Rightarrow_{G_1} B\overset{+}{\Rightarrow}_{G_1} A$. Тогда $A\overset{+}{\Rightarrow}_G Aw¢$ для некоторой цепочки w из $(\Sigma\cup\{¢\})^*$; следовательно, G порождает цепочки, содержащие более одного символа ¢. Поскольку это, как мы знаем, неверно, заключаем, что G_1 — приведенная грамматика.

Теперь надо показать, что G_1 — грамматика предшествования. Так как ¢ встречается только на правом конце цепочек, порождаемых в G символом S, легко показать, что в G не выполняются только те соотношения, выполняющиеся в G_1, которые содержат справа символ \$ (концевой маркер, встречавшийся при описании метода предшествования). Но $X\mathrel{\dot{>}}\$$ — единственное соотношение, имеющее справа \$, которое может выполняться, поэтому G_1 должна быть грамматикой предшествования.

Далее, покажем, что в G_1 нет новых конфликтов $A\to\alpha XY\beta$ и $B\to Y\beta$, где $X\in l(B)$. Как и в теореме 8.11, свойства простого предшествования исключают эту возможность.

Наконец, мы должны доказать, что в P_1 нет пары правил $A\to\alpha$ и $B\to\alpha$, где $A\neq B$. Рассмотрим отдельно три случая.

Случай 1: Допустим, что $\alpha=C$ и правила $A\to CX$ и $B\to CY$, где $X\overset{*}{\Rightarrow}_G ¢$ и $Y\overset{*}{\Rightarrow}_G ¢$, принадлежат P. Пусть $C=[qq']$. Если q' — состояние чтения, то $X=Y=¢$. Как мы видели при доказательстве теоремы 8.11, левая часть правила типа 2 однозначно

определяется его правой частью, поэтому вопреки условию $A=B$. Если q' — состояние записи, обозначим через p единственное состояние чтения в пишущей последовательности для q' и положим $\delta(p, ¢, Z)=(p', Z)$, где δ — функция переходов ДМП-автомата, по которому была построена грамматика G. После того как ДМП-автомат попадает в состояние p', он не может выполнять никаких операций, кроме стирания содержимого магазина, потому что если он будет читать входную цепочку или писать в магазин, то либо он допустит цепочку, содержащую в середине символ ¢, либо X и Y окажутся бесполезными символами. Существует единственное состояние p'', в котором ДМП-автомат находится после стирания символов, записанных в магазин при прохождении пишущей последовательности для q'. Поэтому опять $X=Y$, и мы заключаем, что $A=B$.

Случай 2: Предположим, что $\alpha=C$ и правила $A\to C$ и $B\to CX$, где $X\Rightarrow^* ¢$, принадлежат P. Если $C=[qq']$, то q' — состояние стирания, так как $A\to C$ — правило типа 3. Но поскольку $B\to CX$ — правило типа 2 или 4, q' должно быть состоянием записи или чтения. Мы пришли к противоречию.

Случай 3: Если $\alpha=C$ и правила $A\to CX$ и $B\to C$ принадлежат P, то возникает ситуация, симметричная той, что рассматривалась в случае 2.

Следовательно, G_1 — простая ССП-грамматика. □

8.2.3. ОПК-грамматики, LR-грамматики и детерминированные языки

Покажем, что каноническая грамматика является также (1, 0)-ОПК-грамматикой, а значит, согласно теореме 5.18, и LR(0)-грамматикой. Интуитивно причина того, что для разбора канонической грамматики при помощи алгоритма типа „перенос — свертка" не требуется заглядывать вперед, состоит в том, что каждый терминал и каждый нетерминал $[qq']$, где q' — состояние стирания, определяют правый конец основы. Дадим формальное доказательство, основанное на этой идее.

Теорема 8.14. *Каждая каноническая грамматика является* (1, 0)-ОПК-*грамматикой.*

Доказательство. Пусть $G=(N, \Sigma, P, S)$ — каноническая грамматика. Предположим, что G не является (1, 0)-ОПК-грамматикой. Тогда в пополненной грамматике $G'=(N\cup\{S'\}, \Sigma, P\cup\{S'\to S\}, S')$ можно найти выводы $\$S'\Rightarrow_r^*\alpha XAw\Rightarrow_r\alpha X\beta w$ и $\$S'\Rightarrow_r^*\gamma Bx\Rightarrow_r\gamma\delta x$, где $\gamma\delta x$ можно записать в виде $\alpha' X\beta y$, причем $|x|\leqslant|y|$, но $\alpha' XAy\neq\gamma Bx$. Заметим, что всякая правовыводимая в G цепочка имеет открытую часть, состоящую только из

нетерминалов, т. е. $X\in N$, а α, α' и γ принадлежат N^*. В зависимости от типа правила $A\to\beta$ рассмотрим четыре случая.

Случай 1: Допустим, что $\beta=a$, где $a\in\Sigma$. Так как $|x|\leqslant|y|$, то должно быть $x=y$ и либо $\delta=a$, либо $\delta=Xa$. Если $\delta=a$, то $X\in l(A)\cap l(B)$, а это может быть, только если $A=B$, поскольку G — простая ССП-грамматика. Если $\delta=Xa$, то в P есть правила $A\to a$ и $B\to Xa$, где $X\in l(A)$; это опять противоречит тому, что G — простая ССП-грамматика.

Случай 2: Допустим, что $\beta=Ca$, где $C\in N$. Тогда, поскольку $|x|\leqslant|y|$, должно быть $x=y$ и либо $\delta=a$, либо $\delta=Ca$. Если $\delta=Ca$, то $A=B$, так как правила типа 2 обратимы (теорема 8.11). Отсюда вытекает, что $\alpha' XAy=\gamma Bx$ вопреки предположению. Если $\delta=a$, то из $A\to Ca$ следует $C\doteq a$. Так как символ C должен быть последним символом цепочки γ, то $C\lessdot B$ или $C\doteq B$ и, следовательно, $C\lessdot a$, а это противоречит тому, что G — грамматика предшествования.

Случай 3: Допустим, что $\beta=C$, где $C\in N$. Здесь четыре возможности: $\delta=C$, $\delta=a$, $\delta=Ca$ для некоторого $a\in\Sigma$, $\delta=XC$. Если $\delta=C$, то X принадлежит $l(A)\cap l(B)$, а это противоречит тому, что G — простая ССП-грамматика. Пусть $C=[qq']$. Тогда q' — состояние стирания. Если $\delta=a$, то C — последний символ цепочки γ. Поскольку B в правовыводимой цепочке стоит справа от C, q' должно быть состоянием записи. Итак, пришли к противоречию. Если $\delta=Ca$, то q' должно быть состоянием чтения, а это не так. Если $\delta=XC$, то, поскольку $X\in l(A)$, наличие правил $A\to C$ и $B\to XC$ противоречит тому, что G — простая ССП-грамматика.

Случай 4: Допустим, что $\beta=CD$ для C и D из N. Тогда δ есть либо D, либо a, либо Da для некоторого $a\in\Sigma$, либо CD. Так как правила типа 4 обратимы (доказательство теоремы 8.11), то при $\delta=CD$ имеем $A=B$ и $\alpha' XAy=\gamma Bx$. Пусть $D=[qq']$. Из вида правила $A\to CD$ следует, что q' — состояние стирания. Если $\delta=Da$, то q' должно быть состоянием чтения, и потому этот случай исключается. Если $\delta=a$, то, как и в случае 3, можно показать, что q' — состояние записи. Если $\delta=D$, то последним символом цепочки γ является C, а значит, $C\in l(B)$. Наличие правил $A\to CD$ и $B\to D$ противоречит тому, что G — простая ССП-грамматика. □

Следствие 1. *Всякий детерминированный язык L, обладающий префиксным свойством, имеет* (1, 0)-ОПК-*грамматику.*

Доказательство. Если $e\notin L$, результат очевиден. Если $e\in L$, то $L=\{e\}$. Для этого языка легко найти (1, 0)-ОПК-грамматику. □

Следствие 2. *Если $L\subseteq\Sigma^*$ — детерминированный язык и ¢ не принадлежит Σ, то $L¢$ имеет* (1, 0)-ОПК-*грамматику.* □

Теперь мы можем доказать теорему о произвольных детерминированных языках, представляющую собой по существу перефразировку следствия 2.

Теорема 8.15. *Всякий детерминированный язык порождается некоторой* (1, 1)-ОПК-*грамматикой.*

Доказательство. Пусть G — каноническая грамматика. С помощью алгоритма 8.4 построим по G грамматику G_1. Надо показать, что G_1 — это (1, 1)-ОПК-грамматика. Интуитивно проблема состоит в том, чтобы распознать, когда для свертки нужно применить правило $A \to B$ грамматики G_1, не являющееся правилом грамматики G. Заглядывание на один символ вперед позволяет выполнить такую свертку только тогда, когда аванцепочка состоит только из концевого маркера $\$$. Формальное доказательство оставляем в качестве упражнения. □

Теорема 8.16. (1) *Всякий детерминированный язык, обладающий префиксным свойством, имеет* LR(0)-*грамматику.*

(2) *Всякий детерминированный язык имеет* LR(1)-*грамматику.*

Доказательство. Согласно теореме 5.18, каждая (m, k)-ОПК-грамматика является LR(k)-грамматикой. Результат вытекает из теоремы 5.18 и следствия 1 теоремы 8.14. □

8.2.4. Грамматики расширенного предшествования и детерминированные языки

К настоящему моменту мы установили, что если L — детерминированный КС-язык, то

(1) существует такой ДМП-автомат P в нормальной форме, что $L(P) = L¢$,

(2) существует такая простая грамматика со смешанной стратегией предшествования G, что $L(G) = L - \{e\}$,

(3) существует такая (1, 1)-ОПК-грамматика G, что $L(G) = L$.

Покажем теперь, что существует также обратимая грамматика (2, 1)-предшествования G, для которой $L(G) = L - \{e\}$. Нам уже известно, что каноническая грамматика является грамматикой (1, 1)-предшествования, хотя она не обязательно обратима. Для того чтобы получить нужный результат, выполним некоторые преобразования канонической грамматики, превращающие ее в обратимую грамматику (2, 1)-предшествования. Эти преобразования выглядят следующим образом:

(1) Каждый нетерминал пополняется так, чтобы он „помнил" символ, стоящий слева от него в правом выводе.

(2) Цепные правила исключаются в результате замены правил вида $A \to B$ и $B \to \alpha_1 | \ldots | \alpha_m$ правилами $A \to \alpha_1 | \ldots | \alpha_m$.

(3) Нетерминалы „расщепляются" так, чтобы каждый нетерминал либо имел только одно правило вида $A \to a$, либо не имел правил такого вида.

(4) Наконец, нетерминалы „растягиваются" так, чтобы устранить необратимость, возникшую в результате появления правил вида $A \to a$. Другими словами, множество правил $A_1 \to a, \ldots, A_k \to a$ заменяется множеством $A_1 \to A_2, \ldots, A_{k-1} \to A_k$ и $A_k \to a$.

Каждая из первых трех операций сохраняет свойство грамматики быть грамматикой (1, 1)-предшествования. Четвертая может в худшем случае сделать ее грамматикой (2, 1)-предшествования, но она необходима для того, чтобы обеспечить обратимость. Дадим законченный алгоритм.

Алгоритм 8.5. Превращение канонической грамматики в обратимую грамматику (2, 1)-предшествования.

Вход. Каноническая грамматика $G = (N, \Sigma, P, S)$.

Выход. Эквивалентная обратимая грамматика (2, 1)-предшествования $G_4 = (N_4, \Sigma, P_4, S_4)$.

Метод.

(1) Построить по G грамматику $G_1 = (N_1, \Sigma, P_1, S_1)$:

(а) пусть N_1' — множество символов A_X, где $A \in N$ и $X \in N \cup \{\$\}$;

(б) положим $S_1 = S_\$$;

(в) если $A \to B$, $A \to BC$, $A \to Ba$ и $A \to a$ — правила из P, то в P_1' включаются соответственно правила $A_X \to B_X$, $A_X \to B_X C_B$, $A_X \to B_X a$ и $A_X \to a$ для всех $X \in N \cup \{\$\}$;

(г) N_1 и P_1 — это соответственно N_1' и P_1' после удаления бесполезных символов и правил.

(2) Построить по G_1 грамматику $G_2 = (N_2, \Sigma, P_2, S_1)$:

(а) убрать из P_1 все цепные правила, выполняя следующую операцию до тех пор, пока грамматика не перестанет изменяться: если $A \to B$ — рассматриваемое правило из P_1, добавить правила $A \to \alpha$ для всех правил $B \to \alpha$, принадлежащих P_1, и выбросить правило $A \to B$;

(б) обозначим через N_2 и P_2 полученные в результате множества полезных нетерминалов и правил.

(3) Построить по G_2 грамматику $G_3 = (N_3, \Sigma, P_3, S_1)$:

(а) пусть N_3' — множество N_2 вместе со всеми символами A^a, такими, что правило $A \to a$ принадлежит P_2;

(б) для каждого A^a из N_3' включить в P_3' правило $A^a \to a$;

(в) если $A \in N_2$, $A \to \alpha \in P_2$ и α не является одиночным терминальным символом, добавить к P_3' правило $A \to \alpha$

для каждого α' из $h^{-1}(\alpha)$, где h — гомоморфизм, определенный равенствами

$$h(a) = a \quad \text{для всех} \quad a \in \Sigma$$
$$h(B^a) = B \quad \text{для всех} \quad B^a \in N_3'$$
$$h(B) = B \quad \text{для всех} \quad B \in N_2$$

(г) обозначим через N_3 и P_3 полезные подмножества множеств N_3' и P_3' соответственно.

(4) Построить по G_3 грамматику $G_4 = (N_4, \Sigma, P_4, S_4)$:

(а) $N_4 = N_3$ и $S_4 = S_1$;

(б) P_4 — это множество P_3, измененное так: если $A_1 \to a$, $A_2 \to a$, ..., $A_k \to a$ — все правила из P_3 с правой частью a, взятые в некотором порядке, и $k > 1$, то в P_4 они заменяются на $A_1 \to A_2$, $A_2 \to A_3$, ..., $A_{k-1} \to A_k$, $A_k \to a$. □

Пример 8.4. Пусть G — грамматика с правилами

$$S \to AB$$
$$A \to a \mid b$$
$$B \to AC$$
$$C \to D$$
$$D \to a$$

Тогда G_1 определяется правилами

$$S_\$ \to A_\$ B_A$$
$$A_\$ \to a \mid b$$
$$B_A \to A_A C_A$$
$$A_A \to a \mid b$$
$$C_A \to D_A$$
$$D_A \to a$$

На шаге (2) алгоритма 8.5 правило $C_A \to D_A$ заменяется на $C_A \to a$. Символ D_A, таким образом, бесполезен.

На шаге (3) добавляются нетерминалы $A^a_\$$, $A^b_\$$, A^a_A, A^b_A и C^a_A. Тогда $A_\$$, A_A и C_A становятся бесполезными, и в результате правила грамматики G_3 принимают вид

$$S_\$ \to A^a_\$ B_A \mid A^b_\$ B_A$$
$$A^a_\$ \to a$$
$$A^b_\$ \to b$$
$$B_A \to A^a_A C^a_A \mid A^b_A C^a_A$$
$$A^a_A \to a$$
$$A^b_A \to b$$
$$C^a_A \to a$$

На шаге (4) растягиваются $A^a_\$$, A^a_A и C^a_A, а также $A^b_\$$ и A^b_A. Окончательно правила грамматики G_4 выглядят так:

$$S_\$ \to A^a_\$ B_A \mid A^b_\$ B_A$$
$$B_A \to A^a_A C^a_A \mid A^b_A C^a_A$$
$$A^a_\$ \to A^a_A$$
$$A^a_A \to C^a_A$$
$$C^a_A \to a$$
$$A^b_\$ \to A^b_A$$
$$A^b_A \to b \quad \square$$

Покажем, что G_4 — обратимая грамматика (2, 1)-предшествования, доказав предварительно ряд лемм.

Лемма 8.10. *В алгоритме* 8.5

(1) $L(G_1) = L(G)$,

(2) G_1 — *грамматика* (1, 1)-*предшествования*,

(3) *если правила* $A \to \alpha$ *и* $B \to \alpha$ *принадлежат* P_1 *и* α *не является одиночным терминальным символом, то* $A = B$.

Доказательство. Утверждение (1) доказывается простой индукцией по длине выводов. Для доказательства утверждения (2) заметим, что если X и Y связаны в G_1 одним из отношений $\lessdot$, $\doteq$ или $\gtrdot$, то X' и Y', представляющие собой X и Y с отброшенными индексами (если они были), связаны тем же отношением в G. Таким образом, G_1 — грамматика (1, 1)-предшествования, поскольку такова грамматика G.

Чтобы доказать (3), отметим, что правила типов 2 и 4 обратимы в G. Иными словами, если $A \to CX$ и $B \to CX$ принадлежат P, то $A = B$. Отсюда, таким образом, следует, что если $A_Y \to C_Y X_C$ и $B_Y \to C_Y X_C$ принадлежат P_1, то $A_Y = B_Y$. Аналогично, если X — терминал, а $A_Y \to C_Y X$ и $B_Y \to C_Y X$ принадлежат P_1, то $A_Y = B_Y$.

Мы должны рассмотреть правила в P_1, получаемые из правил типа 3. Допустим, что у нас есть правила $A_X \to C_X$ и $B_X \to C_X$. Так как бесполезные правила из G_1 исключены, в G должны быть правила с такими правыми частями XD и XE, что $D \Rightarrow^*_G A\alpha$ и $E \Rightarrow^*_G B\beta$ для некоторых α и β. Пусть $X = [qq']$, $A = [pp']$, $B = [rr']$ и $C = [ss']$. Тогда p и r принадлежат пишущей последовательности для q', и после каждого из них стоит s. Отсюда заключаем, что $p = r$. Так как правила $A \to C$ и $B \to C$ оба являются правилами типа 3, то $p' = r'$. Иначе говоря, пусть δ — функция переходов ДМП-автомата M, по которому была построена грамматика G. Тогда $\delta(p, e, Z) = (s, YZ)$ для некоторого Y, и $\delta(s', e, Y) = (p', e) = (r', e)$. Поэтому $A = B$ и $A_X = B_X$.

Случай $X=\$$ рассматривается аналогично с той лишь разницей, что роль q' играет начальное состояние q_0 ДМП-автомата M. □

Лемма 8.11. *Грамматика G_2 из алгоритма* 8.5 *обладает теми же тремя свойствами, что и грамматика G_1 из леммы* 8.10.

Доказательство. Свойство (1) опять доказывается простой индукцией. Чтобы доказать (2), заметим, что на шаге (2) алгоритма 8.5 не возникает новых правых частей и, следовательно, новых пар, связанных отношением $\doteq$. Кроме того, каждая правовыводимая цепочка в G_2 правовыводима и в G_1, а потому легко показать, что не возникает и новых отношений $\lessdot$ и $\gtrdot$.

Для доказательства (3) достаточно показать, что если правило $A_X \to B_X$ принадлежит P_1, то B_X не встречается в правой части никакого другого правила и, значит, является бесполезным символом, который удаляется на шаге (2б). Нам уже известно, что в P_1 нет правила $C_X \to B_X$, если $C \neq A$. Возможны только правила $C_X \to B_X a$, $C_X \to B_X D_B$ и $C_Y \to X_Y B_X$. Далее, $A \to B$ — правило типа 3, и потому если $B=[qq']$, то q' — состояние стирания. Таким образом, в P_1 не может быть правил $C_X \to B_X a$ и $C_X \to B_X D_B$, поскольку $C \to Ba$ и $C \to BD$ — правила типов 2 и 4 соответственно.

Рассмотрим правило $C_Y \to X_Y B_X$. Пусть $X=[pp']$ и $A=[rr']$. Тогда q — второе состояние в пишущей последовательности для p', потому что $C \to XB$ — правило типа (4). Кроме того, поскольку $A \to B$ — правило типа 3, q следует за r в любой пишущей последовательности, где встречается r. Но так как X в правовыводимой цепочке в G_1 стоит непосредственно слева от A, r встречается в пишущей последовательности для p', причем $r \neq p'$. Таким образом, в пишущей последовательности для p' состояние q попадается дважды, что невозможно. □

Лемма 8.12. *Грамматика G_3 из алгоритма* 8.5 *обладает теми же тремя свойствами, что и грамматика G_1 из леммы* 8.10.

Доказательство. Свойство (1) снова доказывается „в лоб". Для доказательства (2) заметим, что если X и Y связаны в G_3 одним из отношений $\lessdot$, $\doteq$ или $\gtrdot$, то X' и Y', представляющие собой X и Y с отброшенными верхними индексами (если они были), точно так же связаны в G_2. Таким образом, G_3 — грамматика (1,1)-предшествования и свойством (2) обладает. Свойство (3) очевидно. □

Теорема 8.17. *Грамматика G_4 из алгоритма* 8.5 *является обратимой грамматикой* (2,1)-*предшествования.*

Доказательство. Так как P_4 для каждого a из Σ содержит не более одного правила вида $A \to a$, то ясно, что грамматика G_4 обратима. Легко показать, что новые конфликты отношений (1,1)-предшествования могут возникнуть только для новых связей: (1) $A_X^a \gtrdot b$ и (2) $C \lessdot A_X^a$.

Конфликты вида (1) возникают из-за того, что в G_3 выполняются некоторые из соотношений $B_Y^a \gtrdot b$ или $B_Y^a \doteq b$ и, кроме того, $B_Y^a \Rightarrow_{G_4}^+ A_X^a$. В G_3, а значит, и в G_4 могут выполняться также соотношения $A_X^a \doteq b$ или $A_X^a \lessdot b$.

Конфликты вида (2) обусловлены сходными причинами. В G_3 могут выполняться соотношения $C \doteq B_C^a$ или $C \lessdot B_{C'}^a$ и, кроме того, $B_{C'}^a \Rightarrow_{G_4}^+ A_X^a$. В G_3, а значит, и в G_4 возможно также $C \doteq A_X^a$. (C' — это C без индексов.)

Покажем, что потенциальные конфликты вида (1) разрешаются с помощью отношений (2,1)-предшествования. Конфликты вида (2) невозможны.

Случай 1: Предположим, что $CA_X^a \gtrdot b$ в G_4 для некоторого $C \in N_4$ и как в G_3, так и в G_4 либо $CA_X^a \doteq b$, либо $CA_X^a \lessdot b$. Тогда $C=X_Z^d$ для некоторых Z и d; последнего символа в действительности может и не быть. Заметим, что, поскольку $A_X^a \gtrdot b$ выполняется в G_4, но не выполняется в G_3, должен найтись такой вывод $B_Y^a \Rightarrow_{G_4}^+ A_X^a$, что C в правом выводе в грамматике G_3 появляется слева от B_Y^a. Поэтому $Y=X$.

Теперь нужно показать, что в N_3 не может быть двух разных нетерминалов B_X^a и A_X^a. Другими словами, в N_2 нет таких B_X и A_X, что $A \neq B$, $A \Rightarrow_G^* a$, $B \Rightarrow_G^* a$. Пусть $A=[qq']$, $B=[pp']$ и $X=[rr']$. Заметим, что q и p принадлежат пишущей последовательности для r'. Кроме того, если $[ss'] \Rightarrow_G^* a$ и $[ss''] \Rightarrow_G^* a$, то, рассмотрев ДМП-автомат, лежащий в основе G, заключаем, что $s'=s''$. Таким образом, q' однозначно определяется состоянием q, а p' — состоянием p, при условии что $[qq'] \Rightarrow_G^* a$ и $[pp'] \Rightarrow_G^* a$.

Отсюда следует, что поскольку q и p принадлежат пишущей последовательности для r' и $q \neq p$, то либо B встречается в качестве выводимой цепочки в выводе $A \Rightarrow_G^* a$, либо A встречается в выводе $B \Rightarrow_G^* a$.

Без потери общности предположим, что осуществляется первая возможность. Тогда в G существует нетривиальный вывод B из A, и в нем применяются только правила типа (3). Поэтому $A_X \Rightarrow_{G_1}^* B_X$, и символ B_X должен был быть удален на шаге (2) алгоритма 8.5. Таким образом, заключаем, что в G_4 нет символа B_Y^a.

Случай, когда вместо C в приведенном выше рассуждении стоит \$, рассматривается аналогично. Здесь роль r' будет играть начальное состояние соответствующего ДМП-автомата q_0.

Случай 2: Допустим, что для некоторого C в G_3 выполняется соотношение $C \lessdot A_X^a$, а в G_3 и в G_4 — соотношение $C \doteq A_X^a$. Тогда

найдется такой символ B_X^a, что $C \lessdot B_X^a$ или $C \doteq B_X^a$ в G_3 и $B_X^a \Rightarrow_{G_3}^* A_X^a$. Рассуждая, как в случае 1, заключаем, что $C = X_Z^d$ и $A_X \Rightarrow_{G_3}^+ B_X$ или наоборот. Таким образом, A_X или B_X были удалены на шаге 2 алгоритма 8.5. Заметим, что здесь не может быть конфликта отношений (1,1)-предшествования и тем более (2,1)-предшествования.

Отсюда заключаем, что G_4 — обратимая грамматика (2,1)-предшествования. □

Следствие 1. *Каждый детерминированный язык L, обладающий префиксным свойством и не содержащий e, имеет обратимую грамматику* (2,1)-*предшествования.* □

Следствие 2. *Если $L \subseteq \Sigma^*$ — детерминированный язык и $¢$ не принадлежит Σ, то язык $L¢$ имеет обратимую грамматику* (2,1)-*предшествования.* □

Теорему 8.17 можно усилить, отбросив концевой маркер, упомянутый в следствии 2.

Теорема 8.18. *Каждый детерминированный язык, не содержащий e, имеет обратимую грамматику* (2,1)-*предшествования.*

Доказательство. Пусть $L¢$ имеет каноническую грамматику G. Применим к G алгоритм 8.5, а к полученной грамматике G_4 — алгоритм 8.4. Мы утверждаем, что грамматика, построенная алгоритмом 8.4, также является обратимой грамматикой (2,1)-предшествования. Доказательство оставляем в качестве упражнения. □

УПРАЖНЕНИЯ

8.2.1. Постройте ДМП-автоматы в нормальной форме, допускающие

(а) $\{wcw^R \mid w \in (a+b)^*\}$,
(б) $\{a^m b^n a^n b^m \mid m,\ n \geqslant 1\}$,
(в) $L(G_0)$.

8.2.2. Найдите каноническую грамматику для ДМП-автомата в нормальной форме

$$P = (\{q_0,\ q_1,\ q_2,\ q_3,\ q_f\},\ \{0,\ 1\},\ \{Z_0,\ Z_1,\ X\},\ \delta,\ q_0,\ Z_0,\ \{q_f\})$$

где δ для всех Y определяется равенствами

$$\delta(q_0,\ e,\ Y) = (q_1,\ Z_1 Y)$$
$$\delta(q_1,\ 0,\ Y) = (q_2,\ Y)$$
$$\delta(q_1,\ 1,\ Y) = (q_3,\ Y)$$
$$\delta(q_2,\ e,\ Y) = (q_1,\ XY)$$

$$\delta(q_3,\ e,\ X) = (q_1,\ e)$$
$$\delta(q_3,\ e,\ Z_1) = (q_f,\ e)$$
$$\delta(q_3,\ e,\ Z_0) = (q_f,\ e)\ ^{1)}$$

8.2.3. Найдите в упр. 8.2.2 состояния записи, чтения и стирания.

8.2.4. Дайте формальное доказательство теоремы 8.9.

В следующих трех упражнениях имеется в виду каноническая грамматика $G = (N,\ \Sigma,\ P,\ S)$.

8.2.5. Покажите, что

(а) если $[qq'] \to a \in P$, то q — состояние чтения,
(б) если $[qq'] \to [pp']a \in P$, то p' — состояние чтения,
(в) если $[qq'] \to [pp'] \in P$, то q — состояние записи, а p' — состояние стирания,
(г) если $[qq'] \to [pp'][rr'] \in P$, то p' — состояние записи, а r' — состояние стирания.

8.2.6. Покажите, что если $[qq'][pp']$ встречается в качестве подцепочки правовыводимой в G цепочки, то

(а) q' — состояние записи,
(б) $q' \neq p$,
(в) p принадлежит пишущей последовательности для q'.

8.2.7. Покажите, что если $[qq'] \Rightarrow^+ \alpha[pp']$, то p' — состояние стирания.

8.2.8. Докажите необходимость условия в теореме 8.10.

8.2.9. Дайте формальное доказательство теоремы 8.15.

8.2.10. С помощью алгоритма 8.5 найдите обратимые грамматики (2,1)-предшествования для следующих детерминированных языков:

(а) $\{a0^n c0^n \mid n \geqslant 0\} \cup \{b0^n c0^{2n} \mid n \geqslant 0\}$,
(б) $\{0^n a 1^n 0^m \mid n \geqslant 0,\ m \geqslant 0\} \cup \{0^m b 1^n 0^n \mid n \geqslant 0,\ m \geqslant 0\}$.

8.2.11. Рассмотрите полностью случай $X = \$$ в лемме 8.10.

8.2.12. Закончите доказательство теоремы 8.17.

8.2.13. Докажите теорему 8.18.

***8.2.14.** Покажите, что КС-язык имеет LR(0)-грамматику тогда и только тогда, когда он детерминированный и обладает префиксным свойством.

---

[1]) Это правило никогда не используется. Оно требуется для того, чтобы P был ДМП-автоматом в нормальной форме.

8.2.15. Покажите, что КС-язык имеет (1,0)-ОПК-грамматику тогда и только тогда, когда он детерминированный и обладает префиксным свойством.

****8.2.16.** Покажите, что каждый детерминированный язык имеет LR(1)-грамматику
(а) в нормальной форме Хомского,
(б) в нормальной форме Грейбах.

8.2.17. Покажите, что если $A \to \alpha$ и $B \to \alpha$ — правила типа 4 в канонической грамматике, то $A = B$.

***8.2.18.** Если грамматика G из алгоритма 8.4 каноническая, то верно ли, что грамматика G_1, построенная в этом алгоритме, правопокрывает грамматику G? Что будет в случае, когда G — произвольная грамматика?

8.2.19. Закончите доказательство теоремы 8.12.

***8.2.20.** Покажите, что каждая LR(k)-грамматика правопокрывается некоторой (1, k)-ОПК-грамматикой. *Указание*: Видоизмените LR(k)-грамматику, заменив в правых частях правил все терминалы a новыми нетерминалами X_a и добавив новые правила вида $X_a \to a$. Затем измените нетерминалы грамматики, чтобы запомнить множество допустимых ситуаций для активных префиксов, расположенных справа от них.

****8.2.21.** Покажите, что каждая LR(k)-грамматика правопокрывается LR(k)-грамматикой, которая является в то же время грамматикой (1,1)-предшествования (не обязательно обратимой).

***8.2.22.** Покажите, что грамматика G_4 из алгоритма 8.5 правопокрывает грамматику G из этого алгоритма.

Открытые проблемы

8.2.23. Верно ли, что каждая LR(k)-грамматика покрывается некоторой LR(1)-грамматикой?

8.2.24. Верно ли, что каждая LR(k)-грамматика покрывается обратимой грамматикой (2,1)-предшествования? Утвердительный ответ на этот вопрос означал бы также утвердительный ответ на вопрос, сформулированный в упр. 8.2.23.

8.2.25. Эта проблема уже была поставлена в гл. 2, но решение ее пока не получено, поэтому сформулируем ее еще раз. Разрешима ли проблема эквивалентности для ДМП-автоматов? Поскольку все конструкции этого раздела эффективно реализуются, мы получаем много эквивалентных формулировок этой задачи. Например, можно было бы доказывать, что разрешима проблема эквивалентности для простых ССП-грамматик или для обратимых грамматик (2,1)-предшествования.

Замечания по литературе

Теорема 8.11 впервые была доказана Ахо и др. [1972]. Теоремы 8.15 и 8.16 первоначально появились в работе Кнута [1965]. Теорема 8.18 заимствована у Грэхема [1970]. Упр. 8.2.21 взято из работы Грэя и Харрисона [1969].

8.3. ТЕОРИЯ ЯЗЫКОВ ПРОСТОГО ПРЕДШЕСТВОВАНИЯ

Мы уже видели, что многие классы грамматик порождают в точности множество детерминированных языков. Однако существует несколько важных классов грамматик, не порождающих всех детерминированных языков. Один из таких классов образуют LL-грамматики. В настоящем разделе мы изучим другой такой класс, а именно класс грамматик простого предшествования. Мы покажем, что множество языков простого предшествования является собственным подмножеством множества детерминированных языков и несравнимо с множеством LL-языков. Будет показано также, что множество языков операторного предшествования является собственным подмножеством множества языков простого предшествования.

8.3.1. Класс языков простого предшествования

Как отмечалось в гл. 5, каждый КС-язык имеет обратимую грамматику и грамматику предшествования. Если грамматика обладает обоими этими свойствами, то у нас — грамматика и язык простого предшествования. Интересно изучить порождающую способность грамматик простого предшествования. Они могут порождать только детерминированные языки, поскольку всякая грамматика простого предшествования имеет детерминированный анализатор. Докажем два основных результата, относящихся к языкам простого предшествования. Сначала покажем, что язык

$$L_1 = \{a0^n1^n \mid n \geqslant 1\} \cup \{b0^n1^{2n} \mid n \geqslant 1\}$$

не является языком простого предшествования.

Теорема 8.19. *Языки простого предшествования образуют собственное подмножество множества детерминированных языков.*

Доказательство. Ясно, что каждая грамматика простого предшествования является LR(1)-грамматикой. Для того чтобы доказать, что включение собственное, покажем, что не существует грамматики простого предшествования, порождающей детерминированный язык L_1.

Интуитивно причина этого состоит в том, что никакой анализатор простого предшествования для L_1 не может содержать счетчика нулей, встретившихся во входной цепочке, и в то же время помнить, встретился ему в начале входной цепочки сим-

вол a или b. Если анализатор запоминает в магазине первый входной символ, за которым следует строка нулей, то, как только во входной цепочке встретится цепочка единиц, анализатор не сможет определить, одна или две единицы соответствуют каждому записанному в магазине нулю, не стерев предварительно все нули, содержащиеся в магазине. С другой стороны, если анализатор попытается поместить в верхушку магазина указание на то, какой символ — a или b — встретился раньше, то он должен будет в процессе чтения нулей выполнить последовательность сверток, разрушающую счетчик нулей, встретившихся во входной цепочке.

Приведем теперь формальное доказательство, основанное на этом интуитивном рассуждении. Предположим, что $G=(\mathrm{N}, \Sigma, P, S)$ — грамматика простого предшествования и $L(G)=L_1$. Покажем, что этого не может быть, т. е. что всякая грамматика простого предшествования должна порождать цепочки, не принадлежащие L_1.

Допустим, что входную цепочку $a0^n w$, $w \in 1^*$, надо проанализировать с помощью анализатора простого предшествования, построенного для G алгоритмом 5.12. Так как $a0^n$ для любого n является префиксом некоторой цепочки из L_1, каждый нуль должен в конце концов оказаться в магазине. Обозначим через α_i содержимое магазина после переноса i-го нуля. Если $\alpha_i=\alpha_j$ для некоторого $i<j$, то цепочки $a0^j1^i$ и $a0^j1^j$ либо обе допускаются, либо обе отвергаются анализатором, и потому если $i \neq j$, то $\alpha_i \neq \alpha_j$.

Таким образом, для любой константы c можно найти такую цепочку α_i, что $|\alpha_i|>c$ и α_i — префикс любой цепочки α_j при $j>i$ (если это не так, можно построить такую последовательность $\alpha_{s_1}, \alpha_{s_2}, \ldots$ произвольной длины, что $|\alpha_{s_t}|=|\alpha_{s_{t-1}}|$ для $t \geqslant 2$, и тем самым найти две совпадающие цепочки α). Выберем наименьшее i. Тогда должна найтись такая кратчайшая цепочка $\beta \neq e$, что для каждого k $\alpha_i\beta^k=\alpha_{i+mk}$ для некоторого $m>0$. Это следует из того, что поскольку α_i не стирается, пока на входе появляются нули, символы, записываемые анализатором в магазин, не зависят от α_i. Поведение анализатора при входной цепочке $a0^n$ должно быть цикличным, и он должен либо увеличивать длину цепочки, содержащейся в магазине, либо повторно приходить к ситуации с одной и той же магазинной цепочкой (последнее, как мы уже доказали, невозможно).

Рассмотрим теперь поведение анализатора при входной цепочке вида $b0^n x$. Обозначим содержимое магазина после чтения подцепочки $b0^k$ через γ_k. Как и раньше, можно доказать, что для некоторой цепочки γ_j, где j выбрано наименьшим, найдется такая кратчайшая цепочка $\delta \neq e$, что для каждого k $\gamma_j\delta^k=\gamma_{j+qk}$

для некоторого $q>0$. На самом деле, поскольку γ_j никогда не стирается, должно быть $\delta=\beta$ и $q=m$. Другими словами, простой индукцией по $r \geqslant 0$ можно показать, что если после чтения $a0^{j+r}$ магазин содержит $\alpha_i e_r$, то после чтения $b0^{j+r}$ он будет содержать $\gamma_j e_r$.

Изучим поведение анализатора, обрабатывающего входную цепочку $a0^{j+mk}1^{j+mk}$. После чтения $a0^{j+mk}$ магазин будет содержать $\alpha_i\beta^k$. Пусть s — наибольшее из чисел, обладающих тем свойством, что после чтения 1^{j+mk-s} в магазине остается $\alpha_i\zeta$ для некоторой цепочки ζ из $(\mathrm{N} \cup \Sigma)^*$ (т. е. если после этого на входе появляется 1, то один из символов цепочки α_i участвует в свертке). Ясно, что s существует; в противном случае анализатор допустил бы цепочку, не принадлежащую L_1.

Аналогично пусть r — наибольшее из чисел, таких, что анализатор, начав работать с цепочкой $\gamma_j\beta^k$ в магазине и получив на вход $1^{2(j+mk)-r}$, придет к ситуации с цепочкой $\gamma_j\zeta$ в магазине. Как и раньше, ясно, что r существует.

Таким образом, для любого k входная цепочка $b0^{j+mk}1^{j+mk-s+r}$ должна допускаться, поскольку после чтения $b0^{j+mk}$ в магазине содержится цепочка $\alpha_j\beta^k$, а после чтения 1^{j+mk-s} — цепочка $\alpha_j\zeta$. Цепочки β стираются независимо от того, что находится ниже: α_i или α_j, а 1^r приводит к допуску. Но так как $m \neq 0$, то для всех k не может выполняться равенство $i+mk-s+r=2(j+mk)$.

Отсюда заключаем, что L_1 не является языком простого предшествования. □

Теорема 8.20. *Классы* LL-*языков и языков простого предшествования несравнимы.*

Доказательство. Язык L_1 из теоремы 8.19 представляет собой LL(1)-язык, не являющийся языком простого предшествования. Естественная грамматика для L_1 такова:

$$
\begin{aligned}
S &\to aA \mid bB \\
A &\to 0A1 \mid 01 \\
B &\to 0B11 \mid 011
\end{aligned}
$$

Легко видеть, что это LL(2)-грамматика. Левая факторизация превращает ее в LL(1)-грамматику.

Язык $L_2=\{0^n a1^n \mid n \geqslant 1\} \cup \{0^n b2^n \mid n \geqslant 1\}$ не является LL-языком (см. упр. 8.3.2). L_2 — это язык простого предшествования с грамматикой простого предшествования

$$
\begin{aligned}
S &\to A \mid B \\
A &\to 0A1 \mid a \\
B &\to 0B2 \mid b \qquad \square
\end{aligned}
$$

8.3.2. Языки операторного предшествования

Исследуем класс языков, порождаемых грамматиками операторного предшествования. Мы покажем, что хотя существуют неоднозначные грамматики операторного предшествования (простым примером такой грамматики служит грамматика $S \to A \mid B$, $A \to a$, $B \to a$), языки операторного предшествования образуют собственное подмножество множества языков простого предшествования. Начнем с того, что покажем, что каждый язык операторного предшествования порождается обратимой операторной грамматикой без цепных правил.

Лемма 8.13. *Всякая (не обязательно обратимая) грамматика операторного предшествования эквивалентна грамматике операторного предшествования без цепных правил.*

Доказательство. Легко видеть, что алгоритм 2.11, устраняющий цепные правила, сохраняет отношения операторного предшествования между терминалами. □

Дадим теперь алгоритм, превращающий произвольную КС-грамматику без цепных правил в обратимую грамматику без цепных правил.

Алгоритм 8.6. Превращение КС-грамматики, не содержащей цепных правил, в эквивалентную обратимую грамматику.

Вход. КС-грамматика $G = (\mathrm{N}, \Sigma, P, S)$ без цепных правил.

Выход. Эквивалентная обратимая грамматика $G_1 = (\mathrm{N}_1, \Sigma, P_1, S_1)$.

Метод.

(1) Нетерминалами новой грамматики будут непустые подмножества множества N. Формально положим

$$\mathrm{N}_1' = \{M \mid M \subseteq \mathrm{N} \text{ и } M \neq \varnothing\} \cup \{S_1\}.$$

(2) Для всех $w_0, w_1, \ldots, w_k$ из Σ^* и $M_1, \ldots, M_k$ из N_1' включить в P_1' правила $M \to w_0 M_1 w_1 \ldots M_k w_k$, где $M = \{A \mid$ в P существует правило $A \to w_0 B_1 w_1 \ldots B_k w_k$, $B_i \in M_i$ для $1 \leqslant i \leqslant k\}$, $M \neq \varnothing$. Заметим, что таким способом порождается только конечное число правил.

(3) Для всех $M \subseteq \mathrm{N}$, содержащих S, включить в P_1' правила $S_1 \to M$.

(4) Устранить все бесполезные нетерминалы и правила. Полученные полезные подмножества множеств N_1' и P_1' обозначить N_1 и P_1 соответственно. □

Пример 8.5. Рассмотрим грамматику

$$S \to a \mid aAbS$$
$$A \to a \mid aSbA$$

На шаге (2) включаем в P_1' правила

$$\{S\} \to a\{A\}b\{S\} \mid a\{S, A\}b\{S\} \mid a\{A\}b\{S, A\}$$
$$\{A\} \to a\{S\}b\{A\} \mid a\{S, A\}b\{A\} \mid a\{S\}b\{S, A\}$$
$$\{S, A\} \to a \mid a\{S, A\}b\{S, A\}$$

На шаге (3) добавляем

$$S_1 \to \{S\} \mid \{S, A\}$$

На шаге (4) обнаруживаем, что все $\{S\}$- и $\{A\}$-правила бесполезны, и получаем, таким образом, грамматику

$$S_1 \to \{S, A\}$$
$$\{S, A\} \to a \mid a\{S, A\}b\{S, A\} \quad \square$$

Лемма 8.14. *Грамматика G_1, построенная по G в алгоритме 8.6, обратима, если G не содержит цепных правил, и является грамматикой операторного предшествования, если G — грамматика операторного предшествования. Кроме того, $L(G_1) = L(G)$.*

Доказательство. Обратимость гарантируется шагом (2), а поскольку G не содержит цепных правил, на шаге (3) не могут появиться необратимые правые части. Доказательство того, что алгоритм 8.6 сохраняет отношения операторного предшествования между терминалами, очевидно; мы оставляем его в качестве упражнения. Наконец, легко проверить индукцией по длине вывода, что если $A \in M$, то $A \Rightarrow_G^* w$ тогда и только тогда, когда $M \Rightarrow_{G_1}^* w$. □

Итак, язык операторного предшествования порождается грамматикой в следующей нормальной форме.

Теорема 8.21. *Если L — язык операторного предшествования, то $L = L(G)$ для некоторой грамматики операторного предшествования $G = (\mathrm{N}, \Sigma, P, S)$, такой, что*

(1) *G — обратимая грамматика,*
(2) *S не встречается в правых частях правил,*
(3) *цепные правила обязательно имеют S в левой части.*

Доказательство. Достаточно применить алгоритмы 2.11 и 8.6 к произвольной грамматике операторного предшествования. □

Превратим теперь грамматику операторного предшествования G в нормальной форме, описанной в теореме 8.21, в обратимую грамматику слабого предшествования G_1, для которой $L(G_1) = ¢L(G)$, где ¢ — левый концевой маркер. Потом построим обратимую грамматику слабого предшествования G_2, для которой $L(G_2) =$

$= L(G)$. Тем самым мы покажем, что множество языков операторного предшествования является подмножеством множества языков простого предшествования.

Алгоритм 8.7. Превращение грамматики операторного предшествования в грамматику слабого предшествования.

Вход. Обратимая грамматика операторного предшествования $G = (\mathrm{N}, \Sigma, P, S)$, удовлетворяющая условиям теоремы 8.21.

Выход. Обратимая грамматика слабого предшествования $G_1 = (\mathrm{N}_1, \Sigma \cup \{¢\}, P_1, S_1)$, для которой $L(G_1) = ¢L(G)$, где $¢ \notin \Sigma$.

Метод.

(1) Пусть N_1' состоит из всех таких символов $[XA]$, что $X \in \Sigma \cup \{¢\}$ и $A \in \mathrm{N}$.

(2) Положим $S_1 = [¢S]$.

(3) Зададим гомоморфизм h из $\mathrm{N}_1' \cup \Sigma \cup \{¢\}$ в $\mathrm{N} \cup \Sigma$ равенствами

(а) $h(a) = a$ для $a \in \Sigma \cup \{¢\}$,

(б) $h([aA]) = aA$.

$h^{-1}(\alpha)$ определено только для цепочек $\alpha \in (\mathrm{N} \cup \Sigma)^*$, начинающихся символом из $\Sigma \cup \{¢\}$ и не имеющих рядом стоящих нетерминалов. Кроме того, если значение $h^{-1}(\alpha)$ определено, то оно единственно. Другими словами, h^{-1} объединяет нетерминал со стоящим слева от него терминалом. Пусть P_1' состоит из всех таких правил $[aA] \to h^{-1}(a\alpha)$, что $A \to \alpha \in P$ и $a \in \Sigma \cup \{¢\}$.

(4) Полезные подмножества множеств N_1' и P_1' обозначим N_1 и P_1 соответственно. □

Пример 8.6. Пусть G определяется правилами

$$S \to A$$
$$A \to aAbAc \mid aAd \mid a$$

Образуем только полезные подмножества множеств N_1' и P_1' из алгоритма 8.7. Начнем с нетерминала $[¢S]$. Ему отвечает правило $[¢S] \to [¢A]$. Правилами для $[¢A]$ будут $[¢A] \to ¢[aA][bA]c$, $[¢A] \to ¢[aA]d$ и $[¢A] \to ¢A$. Правила для $[aA]$ и $[bA]$ строятся аналогично. Таким образом, правилами грамматики G_1 будут

$$[¢S] \to [¢A]$$
$$[¢A] \to ¢[aA][bA]c \mid ¢[aA]d \mid ¢a$$
$$[aA] \to a[aA][bA]c \mid a[aA]d \mid aa$$
$$[bA] \to b[aA][bA]c \mid b[aA]d \mid ba \quad \square$$

Лемма 8.15. *Грамматика G_1 из алгоритма 8.7 является обратимой грамматикой слабого предшествования и $L(G_1) = ¢L(G)$.*

Доказательство. Обратимость доказать легко, и мы не будем этого делать. Чтобы показать, что G_1 — грамматика слабого предшествования, надо установить, что отношения $\lessdot$ и $\doteq$ в G_1 не пересекаются с $\gtrdot$ [1]). Зададим гомоморфизм g равенствами

(1) $g(a) = a$ для всех $a \in \Sigma$,

(2) $g(¢) = \$$,

(3) $g([aA]) = a$.

Достаточно показать, что

(1) если $X \lessdot Y$ в G_1, то $g(X) \lessdot g(Y)$ или $g(X) \doteq g(Y)$ в G,

(2) если $X \doteq Y$ в G_1, то $g(X) \lessdot g(Y)$ или $g(X) \doteq g(Y)$ в G,

(3) если $X \gtrdot Y$ в G_1, то $g(X) \gtrdot g(Y)$ в G.

Случай 1: Пусть $X \lessdot Y$ в G_1, где $X \neq ¢$ и $X \neq \$$. Тогда в P_1 есть правило с такой правой частью, скажем $\alpha X[aA]\beta$, что $[aA] \Rightarrow^+_{G_1} Y\gamma$ для некоторой цепочки γ. Если $\alpha \neq e$, легко показать, что в P есть правило с правой частью, содержащей в качестве подцепочки $h(X[aA])$ [2]), и значит, $g(X) \doteq a$ в G. Но из вида правил в P_1 следует, что $g(Y) = a$. Таким образом, $g(X) \doteq g(Y)$.

Если $\alpha = e$ и $X \in \Sigma$, то левая часть правила с правой частью $\alpha X[aA]\beta$ имеет вид $[XB]$ для некоторого $B \in \mathrm{N}$. В этом случае в P должно быть правило, правая часть которого содержит в качестве подцепочки XB. Более того, $B \to aAh(\beta)$ — правило из P. Следовательно, $X \lessdot a$ в G. Так как в этом случае $g(X) = X$, мы заключаем, что $g(X) \lessdot g(Y)$. Если $\alpha = e$ и $X = [bB]$ для некоторых $b \in \Sigma$ и $B \in N$, обозначим левую часть, соответствующую правой части $\alpha X[aA]\beta$, через $[bC]$. Тогда в P есть правило с правой частью, содержащей в качестве подцепочки bC, и $C \to BaAh(\beta)$ принадлежит P. Поэтому $b \lessdot a$ в G и, значит, $g(X) \lessdot g(Y)$. Случай $\alpha = e$ и $X = [¢B]$ для некоторого $B \in \mathrm{N}$ (а также $X = ¢$ или $X = \$$) легко анализируется.

Случай 2: Случай $X \doteq Y$ рассматривается аналогично случаю 1.

Случай 3: $X \gtrdot Y$. Предположим, что $Y \neq \$$. Тогда в P_1 есть правило с такой правой частью, скажем $\alpha[aA]Z\beta$, что $[aA] \Rightarrow^+_{G_1} \gamma X$ и $Z \Rightarrow^*_{G_1} Y\delta$ для некоторых γ и δ. Вид правил в P_1 таков, что либо $Z = Y$ и оба принадлежат $\Sigma \cup \{\$\}$, либо $Z = [aB]$ и $Y = [aC]$ или $Y = a$ для некоторых B и C из N. В любом случае $g(Z) = g(Y)$.

В P должно быть правило с правой частью, содержащей подцепочку $Ag(Z)$, поскольку $\alpha[aA]Z\beta$ служит правой частью

[1]) Здесь и далее символы $\lessdot$, $\doteq$ и $\gtrdot$ обозначают отношения операторного предшествования в G и отношения предшествования Вирта — Вебера в G_1.

[2]) h — гомоморфизм, определенный в алгоритме 8.7.

некоторого правила из P_1. Так как G не содержит цепных правил, за исключением начального, $\gamma \neq e$, и значит, в выводе $[aA] \Rightarrow^{+}_{G_1} \gamma X \quad g(X)$ выводится не из a, а из A. Поэтому $g(X) \lessdot g(Z)$ и $g(X) \gtrdot g(Y)$ в G.

Случай $Y = \$$ не вызывает затруднений.

Для завершения доказательства того, что G_1 — грамматика слабого предшествования, нужно показать, что если правила $A \to \alpha X \beta$ и $B \to \beta$ принадлежат P_1, то не выполняется ни одно из соотношений $X \lessdot B$ и $X \doteq B$. Положим $B = [aC]$ и временно допустим, что $C \neq S$. Тогда, поскольку G не содержит цепных правил, у которых в левой части стоит символ, отличный от S, можно утверждать, что $\beta = Y\beta'$ для некоторой цепочки $\beta' \neq e$, где $h(Y) = a$. Так как $\beta' \neq e$, то $h(\beta')$ содержит хотя бы один терминал. Пусть b — самый левый из этих терминалов. Тогда, поскольку a может встретиться в правовыводимой в G цепочке слева от C, можно считать, что $a \lessdot b$ в G. Но так как $\alpha X \beta$ служит правой частью некоторого правила из P_1, то $h(\beta)$ — подцепочка правой части некоторого правила из P, и значит, $a \doteq b$ в G. Тем самым возможность $C \neq S$ исключается.

Если $C = S$, то по теореме 8.21(2) должно быть $a = ¢$. Но в этом случае, поскольку $\alpha X \beta$ — правая часть некоторого правила из P_1, ¢ должно встретиться в правой части некоторого правила из P, что по нашему допущению неверно.

Отсюда заключаем, что G_1 — грамматика слабого предшествования. Доказательство равенства $L(G_1) = ¢L(G)$ очевидно, и мы его опускаем. □

Теорема 8.22. *Всякий язык операторного предшествования является языком простого предшествования.*

Доказательство. Согласно теореме 8.21 и лемме 8.15, если $L \subseteq \Sigma^+$ — язык операторного предшествования, то $¢L$ — обратимый язык слабого предшествования, $¢ \notin \Sigma$. Легко обобщить алгоритм 8.4 так, чтобы устранить из грамматики языка $¢L$, построенной алгоритмом 8.7, левый концевой маркер. (Вид правил в алгоритме 8.7 гарантирует, что полученная грамматика будет приведенной и обратимой.) Детали доказательства оставляем читателю в качестве упражнения. По теореме 5.16 L — язык простого предшествования. □

8.3.3. Обзор содержания главы

Отношения включения для классов языков, установленные в гл. 8, можно изобразить в виде рис. 8.7. Все включения собственные.

Приведем примеры языков из классов, показанных на рис. 8.7:

(*a*) $\{0^n1^n \mid n \geqslant 1\}$ — LL(1)-язык и в то же время язык операторного предшествования,

(*b*) $\{0^n1^n \mid n \geqslant 1\} \cup \{0^n2^n \mid n \geqslant 1\}$ — язык операторного предшествования, но не LL-язык,

(*c*) $\{a0^n1^m0^m \mid m, n \geqslant 1\} \cup \{b0^m1^n0^n \mid m, n \geqslant 1\} \cup \{a0^n2^n0^m \mid m, n \geqslant 1\}$ — язык простого предшествования, не являющийся ни языком операторного предшествования, ни LL-языком,

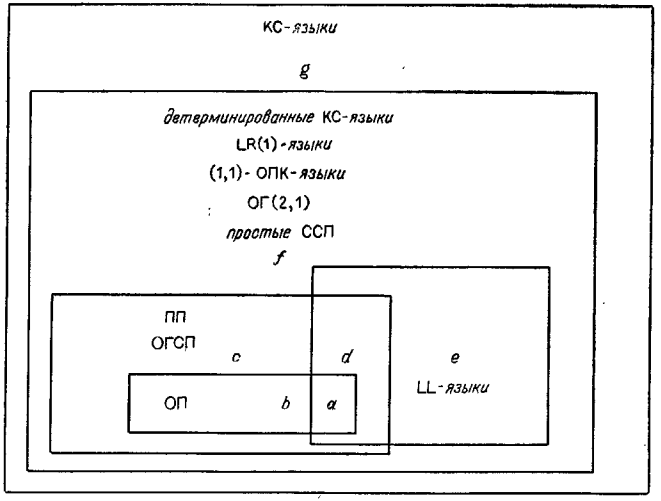


Рис. 8.7. Подклассы класса детерминированных языков. (ОГ (2,1) — языки, порождаемые обратимыми грамматиками (2,1)-предшествования, простые ССП — языки, порождаемые простыми ССП-грамматиками, ПП — языки простого предшествования, ОГСП — языки, порождаемые обратимыми грамматиками слабого предшествования, ОП — языки операторного предшествования.)

(*d*) $\{a0^n1^n0^m \mid m, n \geqslant 1\} \cup \{b0^m1^n0^n \mid m, n \geqslant 1\}$ — LL(1)-язык и в то же время язык простого предшествования, но не операторного предшествования,

(*e*) $\{a0^n1^n \mid n \geqslant 1\} \cup \{b0^n1^{2n} \mid n \geqslant 1\}$ — LL(1)-язык, но не язык простого предшествования,

(*f*) $\{a0^n1^n \mid n \geqslant 1\} \cup \{a0^n2^{2n} \mid n \geqslant 1\} \cup \{b0^n1^{2n} \mid n \geqslant 1\}$ — детерминированный язык, не являющийся ни языком простого предшествования, ни LL-языком,

(*g*) $\{0^n1^n \mid n \geqslant 1\} \cup \{0^n1^{2n} \mid n \geqslant 1\}$ — КС-язык, но не детерминированный.

Доказательства этих утверждений предлагаем в качестве упражнений.

УПРАЖНЕНИЯ

8.3.1. Покажите, что грамматика языка L_1, приведенная в теореме 8.20, является LL(2)-грамматикой. Найдите для L_1 LL(1)-грамматику.

***8.3.2.** Докажите, что язык $L_2 = \{0^n a 1^n \mid n \geqslant 1\} \cup \{0^n b 2^n \mid n \geqslant 1\}$ не является LL-языком. *Указание:* Предположите, что L_2 имеет LL(k)-грамматику в нормальной форме Грейбах.

***8.3.3.** Покажите, что язык L_2 из упр. 8.3.2 является языком операторного предшествования.

8.3.4. Докажите, что алгоритм 2.11 (исключение цепных правил) сохраняет свойства

(а) операторного предшествования,
(б) (m, n)-предшествования,
(в) (m, n)-ОПК.

8.3.5. Докажите лемму 8.14.

***8.3.6.** Почему алгоритм 8.6 не всегда превращает произвольную грамматику предшествования в грамматику простого предшествования?

8.3.7. Превратите следующую грамматику операторного предшествования в эквивалентную грамматику простого предшествования:

$$S \to \textbf{if } b \textbf{ then } S_1 \textbf{ else } S \mid \textbf{if } b \textbf{ then } S \mid a$$
$$S_1 \to \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_1 \mid a$$

8.3.8. Проведите доказательство в случае 2 из леммы 8.15.

8.3.9. Дайте алгоритм устранения из языка, порождаемого некоторой грамматикой, левого концевого маркера. Покажите, что если Ваш алгоритм применить к грамматике, построенной алгоритмом 8.7, то полученная грамматика останется обратимой грамматикой слабого предшествования.

8.3.10. Покажите, что язык простого предшествования

$$L = \{a0^n1^n0^m \mid n \geqslant 1,\ m \geqslant 1\} \cup \{b0^m1^n0^n \mid n \geqslant 1,\ m \geqslant 1\}$$

не является языком операторного предшествования. *Указание:* Покажите, что во всякой грамматике для L будут одновременно выполняться отношения операторного предшествования $1 \lessdot 1$ и $1 \gtrdot 1$.

***8.3.11.** Покажите, что язык L из упр. 8.3.10 является языком простого предшествования.

***8.3.12.** Докажите, что языки, описанные в разд. 8.3.3, обладают всеми приписанными им там свойствами.

8.3.13. Приведите дополнительные примеры языков, принадлежащих разным классам, указанным на рис. 8.7.

8.3.14. Обобщите теорему 8.18 так, чтобы можно было показать, что язык L_1 из этой теоремы не является обратимым языком (1, k)-предшествования ни для какого k.

8.3.15. Покажите, что грамматика G_1 из алгоритма 8.7 правопокрывает грамматику G из этого алгоритма.

8.3.16. Верно ли, что грамматика G_1, построенная в алгоритме 8.6, правопокрывает грамматику G из этого алгоритма, если G — приведенная грамматика?

****8.3.17.** Пусть L — язык простого предшествования

$$\{a^n0a^ib^n \mid i,\ n \geqslant 0\} \cup \{0a^n1a^ic^n \mid i,\ n \geqslant 0\}$$

Покажите, что для L не существует анализатора простого предшествования, сообщающего об ошибке сразу после прочтения символа 1 из входной цепочки вида a^n1a^ib (см. упр. 7.3.5).

Открытая проблема

8.3.18. Каким образом класс обратимых языков (1, k)-предшествования, $k > 1$, связан с классами, показанными на рис 8.7? Читатель должен обратить внимание на упр. 8.3.14.

Замечания по литературе

Строгое включение множества языков простого предшествования в множество детерминированных КС-языков, а также множества языков операторного предшествования в множество языков простого предшествования впервые доказано Фишером [1969].

9 ПЕРЕВОД И ГЕНЕРАЦИЯ КОДА

Разработчику компилятора обычно предоставляется неполное описание языка, для которого он должен построить компилятор. Бо́льшую часть синтаксиса языка можно представить в виде КС-грамматики. В то же время точно описать объектный код, который необходимо сгенерировать для произвольной входной программы, значительно труднее. Широко применимые формальные методы определения семантики языка программирования до сих пор остаются предметом исследований.

В настоящей главе мы введем формальные соглашения, которые можно использовать для точного описания переводов, выполняемых в компиляторе, и дадим примеры таких формализмов. Затем разработаем методы машинного выполнения переводов, определяемых этими формализмами.

9.1. РОЛЬ ПЕРЕВОДА В ПРОЦЕССЕ КОМПИЛЯЦИИ

Напомним, что в гл. 1 компилятор определялся как транслятор, отображающий цепочки в цепочки. Входом компилятора служит цепочка символов, составляющая *исходную программу*. Выход компилятора, называемый *объектной программой*, есть также цепочка символов. Объектная программа может быть

(1) последовательностью абсолютных машинных команд,
(2) последовательностью перемещаемых машинных команд,
(3) программой на языке ассемблера,
(4) программой на некотором другом языке.

Обсудим вкратце характеристики каждой из приведенных форм объектной программы.

(1) Метод отображения исходной программы в абсолютную программу на машинном языке, которую можно непосредственно выполнить, является одним из способов, позволяющих добиться очень быстрой компиляции. Примером компилятора такого рода служит WATFOR. Такой тип компиляции больше всего подходит

для небольших программ, не использующих независимо компилируемых подпрограмм.

(2) Перемещаемая машинная команда представляет собой команду, в которой участвуют адреса ячеек памяти относительно некоторого подвижного начала. Объектная программа в форме последовательности перемещаемых машинных команд обычно называется перемещаемым *объектным сегментом*. Этот сегмент может быть связан с другими объектными сегментами, такими, как независимо компилируемые подпрограммы пользователя, программы ввода—вывода и библиотечные функции; при этом получается единый перемещаемый объектный сегмент, часто называемый *модулем загрузки*. Программа, собирающая модуль загрузки из совокупности сегментов, называется *редактором связей*. Модуль загрузки затем размещается в памяти программой, называемой *загрузчиком*, которая превращает перемещаемые адреса в абсолютные. После этого объектная программа готова к выполнению.

Хотя редактирование связей и загрузка требуют времени, большинство промышленных компиляторов вырабатывает перемещаемые сегменты, чтобы получить возможность гибко использовать большие библиотечные подпрограммы и независимо компилируемые подпрограммы.

(3) Подход, при котором выполняется перевод исходной программы в программу на языке ассемблера, которая в свою очередь обрабатывается ассемблером, упрощает конструирование компилятора. Однако общее время, требуемое для того, чтобы выработать программу на машинном языке, достаточно велико, так как обработка выхода такого компилятора ассемблером может занимать столько же времени, сколько и сама компиляция.

(4) Некоторые компиляторы (например, Снобол) преобразуют исходную программу в другую программу на специальном внутреннем языке. Эта внутренняя программа затем выполняется путем моделирования последовательности команд внутренней программы. Такие компиляторы обычно называются *интерпретаторами*. Мы можем, однако, рассматривать как пример собственно компиляции только отображение исходной программы во внутренний язык.

В этой главе выходу компилятора не будет приписываться никакого фиксированного формата, хотя во многих примерах в качестве объектного кода берется язык ассемблера. В настоящем разделе мы рассмотрим в общих чертах процесс компиляции и основные методы преобразования исходной программы в объектный код.

9.1.1. Фазы компиляции

В гл. 1 мы видели, что компилятор можно расчленить на несколько промежуточных трансляторов, каждый из которых выполняет перевод некоторого представления исходной программы, пока, наконец, не будет получена объектная программа. Каждый промежуточный транслятор можно считать преобразованием, определяющим одну *фазу* компиляции, а композиция всех фаз дает компиляцию в целом.

Выбор того, что назвать фазой, достаточно произволен. Удобно считать основными фазами компиляции лексический анализ, синтаксический анализ и генерацию кода. Тем не менее во многих развитых компиляторах эти фазы часто разбиваются на несколько подфаз; могут также быть и другие фазы (например, оптимизация кода).

Входом компилятора служит цепочка символов. Лексический анализатор — это первая фаза компиляции. Он отображает свой вход в цепочку, состоящую из лексем и элементов таблицы имен. В ходе лексического анализа размеры первоначальной, исходной программы несколько уменьшаются, так как идентификаторы заменяются лексемами, а ненужные пробелы и комментарии выбрасываются. После лексического анализа перерабатываемая информация остается по существу цепочкой, но некоторые части этой цепочки являются указателями на информацию в таблице имен.

Хорошая модель лексического анализатора — конечный преобразователь. Мы обсуждали конечные преобразователи и их применение к лексическому анализу в разд. 3.3. Методы, которыми можно пользоваться для помещения информации в таблицы имен и извлечения ее оттуда, будут изложены в гл. 10.

Синтаксический анализатор воспринимает выход лексического анализатора и разбирает его в соответствии с некоторой входной грамматикой. Эта грамматика аналогична грамматике, использованной при описании входного языка. Однако грамматика входного языка обычно не уточняет, какие конструкции следует считать лексемами. Примерами некоторых конструкций, которые обычно распознаются во время лексического анализа, служат ключевые слова, константы и такие идентификаторы, как метки и имена переменных. Но эти же конструкции могут распознаваться и синтаксическим анализатором, и на практике не существует жесткого правила, определяющего, какие конструкции должны распознаваться на лексическом уровне, а какие надо оставить синтаксическому анализатору.

После синтаксического анализа можно считать, что исходная программа преобразована в дерево, называемое *синтаксическим*. Синтаксическое дерево тесно связано с деревом вывода исходной

программы и, как правило, является просто деревом вывода с удаленными цепочками цепных правил. В синтаксическом дереве внутренние вершины в основном соответствуют операциям, а листья представляют операнды, состоящие из указателей входов в таблицу имен. Структура синтаксического дерева отражает синтаксические правила языка программирования, на котором написана исходная программа. Для физического представления синтаксического дерева существует несколько способов, которые мы разберем в следующем разделе. Представление синтаксического дерева мы будем называть *промежуточной программой*.

Фактическим выходом синтаксического анализатора может быть последовательность команд, необходимых для того, чтобы строить промежуточную программу, обращаться к таблице имен, изменять ее и выдавать, когда это требуется, диагностические сообщения. В гл. 4—7 мы рассмотрели модель синтаксического анализатора, порождающую левый или правый разбор. В то же время свойства синтаксических деревьев, используемых на практике, позволяют легко заменить номера правил в правом или левом разборе командами построения синтаксического дерева и выполнения операций над таблицей имен. Поэтому имеет смысл считать левые и правые разборы, порождаемые анализаторами, описанными в гл. 4—7, промежуточными программами, а между синтаксическим деревом и деревом разбора не делать существенного различия.

Компилятор должен также проверять, соблюдены ли определенные семантические соглашения входного языка. Приведем несколько общих примеров таких соглашений:

(1) каждая операторная метка, на которую есть ссылка, должна фактически появляться в качестве метки некоторого оператора в исходной программе,

(2) никакой идентификатор не может быть описан более одного раза,

(3) все переменные должны определяться перед использованием,

(4) при вызове функции число аргументов и их атрибуты должны быть согласованы с определением функции.

Процесс проверки компилятором этих соглашений называется *семантическим анализом*. Часто семантический анализ производится сразу после синтаксического анализа, но его также можно выполнить и на более поздних фазах компиляции. Например, проверку того, что переменные определяются перед использованием, можно произвести во время оптимизации кода, если такая фаза есть. Проверку согласованности атрибутов операндов можно отложить до фазы генерации кода.

В гл. 10 мы изучим грамматики свойств, служащие формальным аппаратом, с помощью которого можно моделировать

различные аспекты синтаксического и семантического анализов.

После синтаксического анализа промежуточная программа отображается генератором кода в объектную программу. Однако для того, чтобы сгенерировать правильный объектный код, генератор кода должен еще иметь доступ к информации, хранящейся в таблице имен. Например, свойства операндов данной операции определяют код, который нужно сгенерировать для этой операции. Так, если A и B — переменные с плавающей точкой, то объектный код, сгенерированный для $A+B$, будет иным, нежели в случае, когда A и B — целые.

Во время генерации кода (или на некоторой ее подфазе) происходит и распределение памяти. Поэтому генератор кода должен знать, представляет ли переменная скаляр, массив или структуру. Эта информация хранится в таблице имен.

В некоторых компиляторах генерации кода предшествует фаза оптимизации. На этой фазе промежуточная программа подвергается преобразованиям с целью привести ее к такой форме, из которой можно получить более эффективную объектную программу. Часто бывает трудно отличить некоторые преобразования оптимизации от хороших приемов генерации кода. Несколько приемов оптимизации, применимых после построения промежуточной программы или в процессе ее генерации, мы рассмотрим в гл. 11.

Реальный компилятор может осуществлять фазу компиляции за один или более проходов. *Проход* состоит в чтении входа из внешней памяти с последующей записью промежуточных результатов во внешнюю память. Объем работы, выполняемой за один проход, является функцией размера машины, на которой будет работать компилятор, языка, для которого разрабатывается компилятор, числа людей, занятых в разработке компилятора, и т. д. Возможно даже осуществление всех фаз компиляции за один проход. Обсуждение вопроса об оптимальном числе проходов для данного компилятора выходит за рамки нашей книги.

9.1.2. Представления промежуточной программы

В этом разделе мы рассмотрим некоторые возможные представления промежуточной программы, вырабатываемой синтаксическим анализатором. Промежуточная программа должна отражать синтаксическую структуру исходной программы. В то же время каждый оператор промежуточной программы должен относительно просто транслироваться в машинный код.

Компиляторы, осуществляющие значительную работу по оптимизации кода, создают детальное представление промежуточной программы, точно отражающее порядок выполнения исходной программы. В других компиляторах представлением промежуточной программы служит простое представление синтаксического дерева, такое, как польская запись. Некоторые компиляторы, незначительно оптимизирующие код, генерируют объектный код по мере разбора. В этом случае «промежуточная» программа существует только условно в виде последовательности шагов алгоритма разбора.

Вот несколько общепринятых представлений промежуточной программы:

(1) постфиксная польская запись,
(2) префиксная польская запись,
(3) связные списочные структуры, представляющие деревья,
(4) многоадресный код с явно именуемыми результатами,
(5) многоадресный код с неявно именуемыми результатами.

Рассмотрим примеры таких представлений.

Польская запись для арифметических выражений была определена в разд. 3.1.1. Например, оператор присваивания

$$A=B+C*-D \tag{9.1.1}$$

с обычными приоритетами операций и знака присваивания (=) в постфиксной польской записи имеет вид[1])

$$ABCD-*+=$$

а в префиксной польской записи — вид

$$=A+B*C-D$$

В постфиксной польской записи операнды выписаны слева направо в порядке их использования. Знаки операций стоят справа от операндов в порядке выполнения операций. Постфиксная польская запись часто используется интерпретаторами в качестве промежуточного языка. Фаза выполнения программы интерпретатором может состоять в вычислении постфиксного выражения с помощью магазина при сумматоре (см. пример 9.4).

Оба типа польских выражений являются линейными представлениями синтаксического дерева выражения (9.1.1), показанного на рис. 9.1. Это дерево отражает синтаксическую структуру выражения (9.1.1). Можно также взять это дерево само в качестве промежуточной программы, закодировав его в виде связной списочной структуры.

Другой метод кодирования синтаксического дерева — применение многоадресного кода. Например, используя многоадресный код с явно именуемыми результатами, можно представить выражение (9.1.1) последовательностью операторов присван-

[1]) В этой записи операция — является унарной, а операции *, + и = бинарными.

вания

$$T_1 \leftarrow -D$$
$$T_2 \leftarrow *CT_1$$
$$T_3 \leftarrow +BT_2$$
$$A \leftarrow T_3$$ [1]

Оператор вида $A \leftarrow \theta B_1 \ldots B_r$ означает, что r-местную операцию θ надо применить к текущим значениям переменных $B_1, \ldots, B_r$, а полученное значение присвоить переменной A. Соответствующий язык будет формально определен в разд. 11.1.

Многоадресный код с явно именуемыми результатами при каждом присваивании требует имени временной переменной для того, чтобы связать с ним значение выражения, стоящего справа в операторе присваивания. Можно избежать необходимости употреблять временные переменные, если пользоваться многоадресным кодом с неявно именуемыми результатами. В такой записи каждый оператор присваивания помечается числом. Затем мы убираем левую часть оператора и обращаемся к промежуточным результатам при помощи числа, приписанного оператору, вырабатывающему промежуточный результат. Например, выражение (9.1.1) будет представлено последовательностью

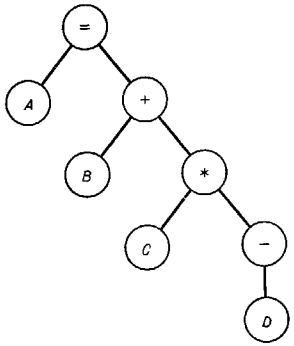


Рис. 9.1. Синтаксическое дерево.

1: $-D$
2: $*C(1)$
3: $+B(2)$
4: $=A(3)$

Здесь число в скобках адресует выражение, помеченное этим числом.

Представления в виде многоадресного кода удобны с вычислительной точки зрения, если фаза оптимизации кода следует за фазой синтаксического анализа. В течение фазы оптимизации кода представление программы может существенно меняться. Внести существенные изменения в польское представление промежуточной программы значительно труднее, чем в представление ее в виде связной списочной структуры.

[1]) Заметим, что операцию присваивания нужно рассматривать иначе, чем остальные. Простая „оптимизация" состоит в том, чтобы в третьей строке заменить T_3 на A, а четвертую строку выбросить.

В качестве второго примера рассмотрим представление оператора

(9.1.2) **if** $I = J$ **then** S_1 **else** S_2

где S_1 и S_2 обозначают произвольные операторы.

Возможным постфиксным польским представлением этого оператора может быть

$$I\ J\ \text{EQUAL?}\ L_2\ \text{JFALSE}\ S_1'\ L\ \text{JUMP}\ S_2'$$

Здесь S_1' и S_2' — постфиксные представления для S_1 и S_2 соответственно; EQUAL? — бинарная операция, дающая значение **истина**, если оба аргумента равны, и **ложь** в противном случае; L_2 — константа, именующая начало S_2'; JFALSE — бинарная операция, вызывающая переход на место, указываемое вторым аргументом, если значение первого аргумента **ложь**, и не вызывающая никаких действий, если значение первого аргумента **истина**; константа L именует первую следующую за S_2' команду; JUMP — унарная операция, вызывающая переход на место, указываемое аргументом.

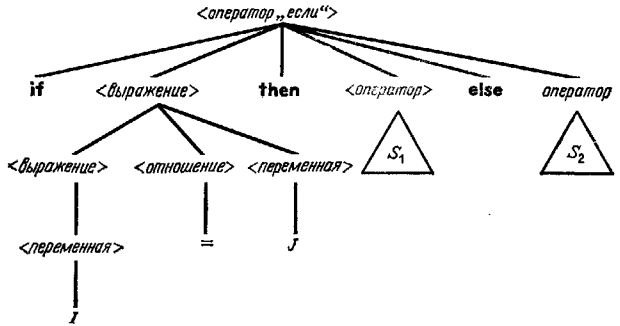


Рис. 9.2. Дерево вывода.

Дерево вывода оператора (9.1.2) приведено на рис. 9.2. Существенную информацию, содержащуюся в этом дереве вывода, можно представить синтаксическим деревом, приведенным на рис. 9.3, где S_1' и S_2' — синтаксические деревья S_1 и S_2.

Синтаксический анализатор, генерирующий промежуточную программу, разбирал бы выражение (9.1.2), просматривая дерево на рис. 9.2, но его выходом была бы последовательность команд построения синтаксического дерева рис. 9.3.

В синтаксическом дереве, вообще говоря, внутренняя вершина представляет операцию, операндами которой служат ее прямые потомки. Листья синтаксического дерева соответствуют идентификаторам. Фактически листья являются указателями на таблицу имен, в которой хранятся имена и атрибуты соответствующих идентификаторов.

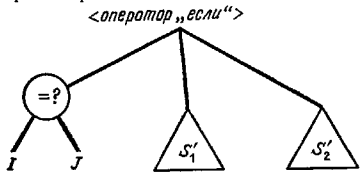


Рис. 9.3. Синтаксическое дерево.

Часть выхода синтаксического анализатора составляют команды занесения информации в таблицу имен. Например, конструкция исходной программы вида

INTEGER *I*

будет преобразована в команду, заносящую атрибут „целое“ в позицию таблицы имен, зарезервированную для идентификатора *I*. Явного представления этой конструкции в промежуточной программе не будет.

9.1.3. Модели процесса генерации кода

Генерация кода — это отображение промежуточной программы в цепочку. Мы будем считать его функцией, определенной на синтаксическом дереве и на информации, содержащейся в таблице имен. Характер отображения зависит от исходной программы, конкретной машины и качества желаемого объектного кода.

Одна из простейших схем генерации кода состоит в том, что каждый оператор многоадресного представления отображается в последовательность команд на выходном языке, независимо от контекста команд в многоадресном представлении. Например, оператор присваивания вида

$$A \leftarrow +BC$$

может отображаться в три машинные команды

```
LOAD   B
ADD    C
STORE  A
```


Здесь мы предполагаем, что используется однопроцессорная машина, в которой команда LOAD *B* помещает содержимое ячейки *B* в сумматор, ADD *C* складывает[1]) содержимое ячейки памяти *C* с содержимым сумматора, а STORE *A* помещает содержимое сумматора в ячейку *A*. Команда STORE содержимого сумматора не изменяет.

Однако если в сумматоре уже хранилось содержимое ячейки *B* (например, в результате выполнения предыдущей команды присваивания $B \leftarrow +DE$), то команда LOAD *B* становится лишней. Кроме того, если следующая команда имеет вид $F \leftarrow +AG$ и *A* нигде больше не используется, то не нужна и команда STORE *A*.

В разд. 11.1 и 11.2 мы изложим несколько приемов генерации кода по операторам многоадресного представления.

УПРАЖНЕНИЯ

9.1.1. Нарисуйте синтаксические деревья для следующих операторов входного языка:

(а) $A = (B - C)/(B + C)$ (в Фортране),
(б) $I = \text{LENGTH}\ (C1 \| C2)$ (в ПЛ/I),
(в) **if** $B > C$ **then**
  if $D > E$ **then** $A := B + C$ **else** $A := B - C$
  else $A := B * C$ (в Алголе).

9.1.2. Найдите постфиксные польские представления для программ из упр. 9.1.1.

9.1.3. Сгенерируйте многоадресный код с явно именуемыми результатами для операторов из упр. 9.1.1.

9.1.4. Постройте детерминированный МП-преобразователь, отображающий префиксную польскую запись в постфиксную польскую запись.

9.1.5. Покажите, что не существует детерминированного МП-преобразователя, отображающего постфиксную польскую запись в префиксную польскую запись. Существует ли недетерминированный МП-преобразователь, выполняющий такое отображение? *Указание:* См. теорему 3.15.

9.1.6. Разработайте алгоритм вычисления постфиксного польского выражения, использующий магазин.

9.1.7. Сконструируйте МП-преобразователь, который, получая на вход выражение w из $L(G_0)$, порождает на выходе по-

[1]) Для простоты допустим, что у нас только один тип арифметики. Если таких типов несколько, например арифметика с фиксированной и плавающей точкой, то перевод операции + будет зависеть от информации об атрибутах имен *B* и *C*, записанной в таблице имен.

следовательность команд построения синтаксического дерева (или многоадресного кода) для w.

9.1.8. Сгенерируйте программы на языке ассемблера наиболее знакомой Вам машины для программ из упр. 9.1.1.

***9.1.9.** Разработайте алгоритмы для генерации программ на языке ассемблера наиболее знакомой Вам машины, соответствующих промежуточным программам, представляющим арифметические действия и присваивания, когда промежуточная программа является

(а) программой в постфиксной польской записи,

(б) программой в многоадресном коде с явно именуемыми результатами,

(в) программой в многоадресном коде с неявно именуемыми результатами,

(г) некоторым представлением синтаксического дерева.

***9.1.10.** Сконструируйте язык, удобный для представления некоторого подмножества Фортрана (или ПЛ/I, или Алгола). Это подмножество должно включать операторы присваивания и некоторые операторы управления. В нем должны допускаться переменные с индексами.

****9.1.11.** Постройте генератор кода, отображающий промежуточную программу из упр. 9.1.10 в машинный код наиболее знакомой Вам машины.

Замечания по литературе

К сожалению, даже для простых конструкций входного языка нельзя точно определить наилучший объектный код. Тем не менее существует ряд книг и статей, посвященных переводу различных языков программирования. Бэкус и др. [1957] подробно описали ранний компилятор Фортрана. Ренделл и Рассел [1964], Грау и др. [1967] изучили задачу реализации Алгола 60. Некоторые детали реализации языка ПЛ/I приведены в ИБМ [1969].

Во многих работах излагаются методы, полезные при генерации кода. Кнут [1968а] исследовал различные методы распределения памяти. Элсон и Рейк [1970] рассмотрели вопрос о генерации кода для древовидных структур промежуточного языка. Уилкокс [1971] представил несколько общих моделей для процесса генерации кода.

9.2. СИНТАКСИЧЕСКИ УПРАВЛЯЕМЫЕ ПЕРЕВОДЫ

В настоящем разделе мы рассмотрим модель компилятора, в которой синтаксический анализ и генерация кода объединены в одну фазу. Такую модель можно представлять себе как компилятор, в котором операции генерации кода совмещены с операциями разбора. Для описания компиляции такого типа часто употребляется термин *синтаксически управляемая компиляция.*

Описываемые здесь методы можно применять также при генерации промежуточного, а не только объектного кода, или кода на языке ассемблера. Мы приводим один пример перевода в промежуточный код, остальные примеры относятся к коду на языке ассемблера.

Нашим отправным пунктом будет схема синтаксически управляемого перевода (СУ-схема), описанная в гл. 3. Мы покажем, как осуществить синтаксически управляемый перевод с помощью детерминированного МП-преобразователя, выполняющего нисходящий или восходящий анализ. В этом разделе будем считать, что ДМП-преобразователь использует специальный символ $ в качестве ограничителя правого конца входной цепочки. Кроме того, чтобы сделать СУ-схему более гибкой, мы наделим ее дополнительными свойствами. Прежде всего, разрешим существование семантических правил, в которых определяется более одного перевода в различных вершинах дерева разбора. В формулах для этих переводов разрешим повторения и условные выражения. Затем рассмотрим переводы, не являющиеся цепочками; полезные дополнительные типы переводов — целые и логические переменные. Наконец, разрешим, чтобы переводы определялись другими переводами, найденными не только в прямых потомках рассматриваемой вершины, но и в ее прямом предке.

Сначала покажем, что всякую простую СУ-схему, заданную на LL-грамматике, можно реализовать детерминированным преобразователем. Затем исследуем, какие простые СУ-схемы на LR-грамматике можно реализовать таким способом. Мы изучим расширение ДМП-автомата, называемое МП-процессором, которое служит для реализации всего класса СУ-переводов, входными грамматиками которых являются LL- или LR-грамматики. После этого вкратце рассмотрим синтаксически управляемые переводы в связи с алгоритмами разбора с возвратом.

9.2.1. Простые синтаксически управляемые переводы

В гл. 3 мы видели, что простую СУ-схему можно реализовать недетерминированным МП-преобразователем. В настоящем разделе мы исследуем детерминированные реализации некоторых простых СУ-схем.

Переводы, определяемые для языка $L(G)$, и эффективность, с которой их можно непосредственно реализовать, могут зависеть от входной грамматики G.

Пример 9.1. Пусть требуется отобразить выражения, порождаемые грамматикой G с правилами $E \to a+E \mid a*E \mid a$, в префиксные польские выражения, в предположении что операция *

имеет более высокий приоритет, чем $+$, т е. префиксным выражением для $a*a+a$ будет $+*aaa$, а не $*a+aa$.

Не существует СУ-схемы, для которой грамматика G была бы входной грамматикой и которая определяла бы такой перевод. Причина состоит в том, что выходная грамматика такой СУ-схемы должна быть линейной КС-грамматикой. В то же время нетрудно показать, что множество префиксных польских выражений в алфавите $\{+, *, a\}$, соответствующих инфиксным выражениям языка $L(G)$, не является линейным КС-языком. Тем не менее этот конкретный перевод можно определить, пользуясь простой СУ-схемой, у которой входной грамматикой служит G_0 (без правила $F \to (E)$). □

В теореме 3.14 было доказано, что если (x, y) — элемент простого синтаксически управляемого перевода, то выход y можно построить по левому разбору цепочки x с помощью детерминированного МП-преобразователя. Отсюда следует, что если грамматику G удается естественным образом разобрать детерминированно сверху вниз с помощью ДМП-преобразователя M, то M легко модифицировать так, чтобы реализовать любую простую СУ-схему, входной грамматикой которой служит G. Если G — LL(k)-грамматика, то любой простой СУ-перевод, определенный на G, реализуется ДМП-преобразователем следующим образом.

Теорема 9.1. *Пусть $T = (N, \Sigma, \Delta, R, S)$ — семантически однозначная*[1]) *простая СУ-схема, входной грамматикой которой служит LL(k)-грамматика. Тогда перевод $\{(x\$, y) \mid (x, y) \in \tau(T)\}$ можно осуществить детерминированным МП-преобразователем.*

Доказательство. Доказательство вытекает из доказательств теорем 3.14 и 5.4. Если G — входная грамматика схемы T, то по алгоритму 5.3 для G можно построить k-предсказывающий алгоритм разбора. Реализуем этот k-предсказывающий анализатор на ДМП-преобразователе M с концевым маркером и выполним перевод следующим образом.

Пусть $\Delta' = \{a' \mid a \in \Delta\}$, причем $\Delta' \cap \Sigma = \varnothing$, и $h(a) = a'$ для всех $a \in \Delta$. Анализатор, сконструированный по алгоритму 5.3, поочередно заменяет нетерминалы правыми частями правил, используя для этого LL(k)-таблицы, содержащие нетерминалы. Автомат M делает по существу то же самое. Предположим, что левый анализатор заменяет A на $w_0B_1w_1 \ldots B_mw_m$ (к нетерминалам добавляются некоторые LL(k)-таблицы, не показанные здесь). Пусть

$$A \to w_0B_1w_1 \ldots B_mw_m, \quad x_0B_1x_1 \ldots B_mx_m$$

— правило из R. Тогда M заменит A на $w_0h(x_0)B_1w_1h(x_1) \ldots \ldots B_mw_mh(x_m)$.

Как и в алгоритме 5.3, всякий раз, когда символ из Σ появляется в верхушке магазина, он сравнивается с текущим входным символом, и если они совпадают, то символ удаляется из магазина, а входная головка сдвигается на одну ячейку вправо. Когда в верхушке магазина оказывается символ a' из Δ', M порождает a и удаляет a' из верхушки магазина, не сдвигая при этом входной головки. M не порождает номеров правил.

Более формальное построение автомата M оставляем читателю. □

Пример 9.2. Пусть T — простая СУ-схема с правилами

$$S \to aSbSc, \quad 1S2S3$$
$$S \to d, \quad 4$$

Входная грамматика является простой LL(1)-грамматикой. Поэтому не обязательно запоминать в магазине LL-таблицы. Пусть M — детерминированный МП-преобразователь $(Q, \Sigma, \Delta, \Gamma, \delta, q, S, \{\text{допуск}\})$, где

$$Q = \{q, \text{допуск}, \text{ошибка}\}$$
$$\Sigma = \{a, b, c, d, \$\}$$
$$\Gamma = \{S, \$\} \cup \Sigma \cup \Delta$$
$$\Delta = \{1, 2, 3, 4\}$$

Так как $\Sigma \cap \Delta = \varnothing$, положим $\Delta' = \Delta$. Функцию δ зададим равенствами

$$\delta(q, a, S) = (q, Sb2Sc3, 1)\text{[1]}$$
$$\delta(q, d, S) = (q, e, 4)$$
$$\delta(q, b, b) = (q, e, e)$$
$$\delta(q, c, c) = (q, e, e)$$
$$\delta(q, e, 2) = (q, e, 2)$$
$$\delta(q, e, 3) = (q, e, 3)$$
$$\delta(q, \$, \$) = (\text{допуск}, e, e)$$

В остальных случаях

$$\delta(q, X, Y) = (\text{ошибка}, e, e)$$

[1]) Авторы используют понятие, определение которого дается несколько позже — в разд. 9.3. Однозначная СУ-схема — это обычная СУ-схема. — *Прим. ред.*

[1]) Здесь мы отказываемся от ограничений, сформулированных при доказательстве теоремы 9.1 и заключающихся в том, что операции порождения выходного символа и сдвига входной цепочки после распознавания правила грамматики должны выполняться в разных тактах.

Легко проверить, что $\tau(M) = \{(x\$, y) \mid (x, y) \in \tau(T)\}$. □

Рассмотрим теперь простую СУ-схему, входной грамматикой которой служит LR(k)-грамматика. Поскольку класс LR(k)-грамматик шире класса LL(k)-грамматик, интересно исследовать, какой класс простых СУ-схем, входными грамматиками которых служат LR(k)-грамматики, можно реализовать на ДМП-преобразователях. Оказывается, существуют семантически однозначные простые СУ-схемы, имеющие в качестве входных грамматик LR(k)-грамматики и не реализуемые ни на каком ДМП-преобразователе. Неформально это объясняется тем, что элемент перевода может требовать порождения выхода задолго до того, как выяснится, что правило, которому приписан этот элемент перевода, действительно было применено.

Пример 9.3. Рассмотрим простую СУ-схему T с правилами

$$S \to Sa,\ aSa$$
$$S \to Sb,\ bSb$$
$$S \to e,\ e$$

Входная грамматика является LR(1)-грамматикой, но по лемме 3.15 не существует ДМП-преобразователя, определяющего перевод $\{(x\$, y) \mid (x, y) \in \tau(T)\}$. Неформально это объясняется тем, что, согласно первому правилу этой СУ-схемы, символ a должен порождаться раньше, чем выяснится, что было применено правило $S \to Sa$. □

Однако, если простая СУ-схема, входная грамматика которой является LR(k)-грамматикой, не требует, чтобы выход порождался до того, как будет установлено, какое правило применялось, такой перевод можно реализовать на ДМП-преобразователе.

Определение. СУ-схему $T = (N, \Sigma, \Delta, R, S)$ будем называть *постфиксной*, если каждое правило из R имеет вид $A \to \alpha, \beta$, где $\beta \in N^*\Delta^*$. Иными словами, каждый элемент перевода представляет собой цепочку из нетерминалов, за которыми следует цепочка выходных символов.

Теорема 9.2. *Пусть $T = (N, \Sigma, \Delta, R, S)$ — семантически однозначная простая постфиксная СУ-схема, входной грамматикой которой служит LR(k)-грамматика. Тогда перевод $\{(x\$, y) \mid (x, y) \in \tau(T)\}$ можно осуществить детерминированным МП-преобразователем.*

Доказательство. Применяя алгоритм 5.11, можно для входной LR(k)-грамматики сконструировать детерминированный

правый анализатор M с концевым маркером. Однако, вместо того чтобы порождать номер правила, M порождает цепочку выходных символов элемента перевода, связанного с этим правилом. Иными словами, если $A \to \alpha, \beta x$ — правило из R, где $\beta \in N^*$ и $x \in \Delta^*$, то всякий раз, когда анализатор порождает номер правила $A \to \alpha$, автомат M порождает цепочку x. Работая таким образом, автомат M определяет перевод $\{(x\$, y) \mid (x, y) \in \tau(T)\}$. □

В качестве упражнения предлагаем доказать утверждение, обратное теореме 9.2, а именно: любое детерминированное МП-преобразование можно выразить в виде постфиксной простой СУ-схемы, заданной на LR(1)-грамматике.

Пример 9.4. Постфиксные переводы полезнее, чем это может показаться на первый взгляд. Мы рассмотрим расширенный ДМП-преобразователь, отображающий арифметические выражения языка $L(G_0)$ в машинный код специальным образом выбранной машины. Машина имеет магазин при сумматоре. Команда

LOAD X

помещает значение, записанное в ячейке X, в верхушку магазина; все остальные элементы магазина проталкиваются вниз. Команды ADD и MPY соответственно складывают и перемножают содержимое двух верхних ячеек, убирая эти ячейки и помещая результат в магазин. Для разделения команд служит точка с запятой.

Наша СУ-схема имеет вид

$E \to E + T,$	ET	'ADD;'
$E \to T,$	T	
$T \to T * F,$	TF	'MPY;'
$T \to F,$	F	
$F \to (E),$	E	
$F \to a,$	'LOAD a;'	

В этом и во всех последующих примерах мы будем пользоваться обозначениями Снобола, заключая цепочки букв и цифр в правилах перевода в кавычки. Кавычки не являются частью выходной цепочки.

Получив на вход цепочку $a + (a * a)\$$, ДМП-преобразователь проходит следующую последовательность конфигураций (здесь LR(k)-таблицы из магазина выброшены; в последовательности опущены, кроме того, некоторые очевидные конфигурации, а также

очевидные состояния и маркер, содержащийся в самой нижней ячейке магазина):

$$
\begin{aligned}
& [e,\ a+(a*a)\$,\ e] \\
\vdash\ & [a,\ +(a*a)\$,\ e] \\
\vdash\ & [F,\ +(a*a)\$,\ \text{LOAD } a;] \\
\vdash^2\ & [E,\ +(a*a)\$,\ \text{LOAD } a;] \\
\vdash^3\ & [E+(a,\ *a)\$,\ \text{LOAD } a;] \\
\vdash\ & [E+(F,\ *a)\$,\ \text{LOAD } a;\text{LOAD } a;] \\
\vdash\ & [E+(T,\ *a)\$,\ \text{LOAD } a;\text{LOAD } a;] \\
\vdash^2\ & [E+(T*a,\ )\$,\ \text{LOAD } a;\text{LOAD } a;] \\
\vdash\ & [E+(T*F,\ )\$,\ \text{LOAD } a;\text{LOAD } a;\text{LOAD } a;] \\
\vdash\ & [E+(T,\ )\$,\ \text{LOAD } a;\text{LOAD } a;\text{LOAD } a;\ \text{MPY};] \\
\vdash\ & [E+(E,\ )\$,\ \text{LOAD } a;\text{LOAD } a;\text{LOAD } a;\text{MPY};] \\
\vdash\ & [E+(E),\ \$,\ \text{LOAD } a;\text{LOAD } a;\text{LOAD } a;\text{MPY};] \\
\vdash^2\ & [E+T,\ \$,\ \text{LOAD } a;\text{LOAD } a;\text{LOAD } a;\text{MPY};] \\
\vdash\ & [E,\ \$,\ \text{LOAD } a;\text{LOAD } a;\text{LOAD } a;\text{MPY};\text{ADD};]
\end{aligned}
$$

Заметим, что если буквы a, представляющие идентификаторы, снабжены индексами так, что входное выражение имеет, например, вид $a_1+(a_2*a_3)$, то выходным кодом будет

$$
\begin{aligned}
&\text{LOAD } a_1 \\
&\text{LOAD } a_2 \\
&\text{LOAD } a_3 \\
&\text{MPY} \\
&\text{ADD}
\end{aligned}
$$

На нашей машине эта программа вычисляет исходное выражение. □

Хотя модель вычислительной машины, описанная в примере 9.4, предназначалась только для демонстрации синтаксически управляемого перевода весьма простых выражений, постфиксная схема способна определить полезные классы переводов. В оставшейся части главы мы покажем, как получается объектный код в других моделях вычислительных машин, работающих подобно МП-преобразователям над тем, что является по существу простой постфиксной СУ-схемой, у которой входной грамматикой служит LR-грамматика.

Предположим, что у нас есть простая, но не постфиксная СУ-схема, входная грамматика которой является LR(k)-грамматикой. Как выполнить такой перевод? Один из возможных методов состоит в использовании следующей многопроходной схемы перевода. Этот метод иллюстрирует каскадную связь

ДМП-преобразователей. Правда, на практике такой перевод можно было бы осуществить и за один проход, если методы следующего раздела применить к произвольным СУ-схемам.

Пусть $T=(N,\ \Sigma,\ \Delta,\ R,\ S)$ — семантически однозначная простая СУ-схема с входной LR(k)-грамматикой G. Для нахождения

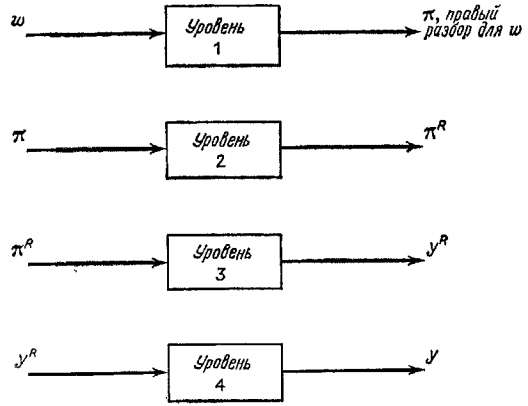


Рис. 9.4. Простой СУ-перевод над LR(k)-грамматикой.

$\tau(T)$ можно построить четырехуровневую схему перевода. Первый уровень занимает ДМП-преобразователь. Его входом служит входная цепочка $w\$$. Выходом является правый разбор π цепочки w, соответствующий входной грамматике G. На втором уровне π обращается — получается обратный правый разбор π^R [1]).

Входом для третьего уровня будет π^R. Выходом — перевод, определяемый простой СУ-схемой $T'=(N,\ \Sigma',\ \Delta,\ R',\ S)$, где R' содержит правило

$$A \to iB_mB_{m-1}\ldots B_1,\ y_mB_m\ldots y_1B_1y_0$$

тогда и только тогда, когда $A \to x_0B_1x_1\ldots B_mx_m,\ y_0B_1y_1\ldots B_my_m$ — правило из P, а $A \to x_0B_1x_1\ldots B_mx_m$ — правило с номером i входной LR(k)-грамматики. Легко доказать, что $(\pi^R,\ y^R)\in\tau(T')$ тогда и только тогда, когда $(S,\ S)\Rightarrow_T \pi^R(w,\ y)$.

[1]) Напомним, что правый разбор π — это последовательность правил в правом выводе, записанная в обратном порядке. Таким образом, π^R начинается с первого примененного правила и заканчивается последним. Для построения π^R достаточно прочитать буфер, в котором записана последовательность π, в обратном порядке.

T' — это простая СУ-схема, основанная на LL(1)-грамматике, и, следовательно, ее можно реализовать на ДМП-преобразователе. На четвертом уровне просто обращается выход третьего уровня. Если выход третьего уровня записывается в магазин, то на четвертом уровне символы выталкиваются из магазина и по мере выталкивания пишутся в выходную цепочку.

Эта процедура представлена на рис. 9.4.

Число основных операций, выполняемых на каждом из этих уровней, пропорционально длине цепочки w. Таким образом, получаем следующий результат:

Теорема 9.3. *Любую простую СУ-схему с входной* LR(k)-*грамматикой можно реализовать за время, пропорциональное длине входной цепочки.*

Доказательство. Доказательство представляет собой формализацию изложенного выше. □

9.2.2. Обобщенный преобразователь

С помощью МП-преобразователя хорошо описываются все простые СУ-схемы над LL-грамматиками и некоторые простые СУ-схемы над LR-грамматиками, но для реализации

(1) непростых СУ-схем,

(2) непостфиксных простых СУ-схем над LR-грамматиками,

(3) простых СУ-схем над грамматиками, не являющимися LR-грамматиками,

(4) синтаксически управляемого перевода в случае, когда разбор проводится недетерминированным образом и не за один проход (как в гл. 4 и 6),

нам потребуется более гибкая модель транслятора.

Определим теперь новое устройство, называемое МП-*процессором*. С его помощью можно будет определить синтаксически управляемые переводы, отображающие цепочки в графы. МП-процессор — это МП-преобразователь, выходом которого служит помеченный ориентированный граф, представляющий собой в общем случае дерево или часть дерева, которое строит процессор. Основная особенность МП-процессора состоит в том, что в его магазине, помимо магазинных символов, могут содержаться указатели на вершины выходного графа.

Подобно расширенному МП-автомату, МП-процессор может исследовать верхние k ячеек своего магазина для любого конечного k и произвольным образом обрабатывать их содержимое. Если эти k ячеек содержат указатели на выходной граф, то в отличие от МП-автомата МП-процессор может видоизменять выходной граф, добавляя или выбрасывая направленные дуги,

соединяющие вершины, на которые указывают эти указатели. МП-процессор может также создавать новые вершины, помечать их, создавать указатели на них и строить дуги, соединяющие эти вершины с теми вершинами, на которые указывают указатели, содержащиеся в верхних ячейках магазина.

Из-за трудностей, возникающих при выработке компактной и ясной формальной записи для описания таких операций, будем пользоваться словесным описанием работы МП-процессора. Так как на каждом шаге работы МП-процессора обрабатывается лишь конечное число указателей, вершин и дуг, такой формализм в принципе возможен, однако мы считаем, что он только заслонил бы идейную простоту рассматриваемых алгоритмов. Начнем сразу с примера.

Пример 9.5. Построим МП-процессор P, отображающий арифметические выражения языка $L(G_0)$ в синтаксические деревья. В этом случае синтаксическое дерево будет деревом, в котором внутренние вершины помечены знаками + или *, а листья — буквами a. На рис. 9.5 и 9.6 перечислены операции разбора и выдачи, выполняемые МП-процессором для всех возможных комбинаций текущего входного символа и символа, содержащегося в верхушке магазина. P сконструирован на основе SLR(1)-анализатора для G_0, показанного на рис. 7.37.

Правда, в данном случае таблица T_4 исключена из рис. 7.37 и правило $F \to a$ рассматривается как цепное; таблицы переименованы следующим образом:

Старое имя	T_0	T_1	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}
Новое имя	T_0	T_1	(	+	*	T_2	T_3	T_4	)

Кроме того, правым концевым маркером служит символ \$. Операции разбора и выдачи процессора P приведены на рис. 9.5 и 9.6. Для того чтобы определить нужное действие, P использует LR(1)-таблицы. Помимо этого, когда таблицы T_1, T_2, T_3 и T_4 помещаются в магазин, P прикрепляет к ним указатели [1]). Эти указатели относятся к порождаемому выходному графу. Однако указатели не влияют на процесс разбора. При разборе символы грамматики в магазин не записываются. Тем не менее имена таблиц указывают, что это должны быть за символы.

Последний столбец на рис. 9.5 содержит имя новой LR(1)-таблицы, которую нужно поместить в магазин после свертки. Пустые элементы обозначают ошибочные ситуации. Новый вход-

[1]) На практике эти указатели можно запоминать в магазине непосредственно под таблицами.

ной символ считывается только после переноса. Числа в таблице относятся к действиям, приведенным на рис. 9.6.

Таблица в верхушке магазина \ Входной символ	Действие						Переход
	a	+	*	(	)	$	
T_0	1			перенос (			T_1
T_1		перенос +	перенос *			допуск	
T_2		перенос +	перенос *		перенос)		
T_3		2	перенос *		2	2	
T_4		3	3		3	3	
+	4			перенос (			T_3
*	5			перенос (			T_4
(	6			перенос (			T_2
)	7	7			7	7	

Рис. 9.5. МП-процессор P.

(1) Построить новую вершину n с меткой a. Поместить в верхушку магазина символ $[T_1, p]$, где p — указатель на n. Прочитать новый входной символ.

(2) На этом этапе в верхушке магазина записана цепочка из четырех символов вида $X[T_i, p_1]+[T_j, p_2]$, где p_1 и p_2 — указатели на вершины n_1 и n_2 соответственно. Построить новую вершину n с меткой +. Сделать n_1 и n_2 левым и правым прямыми потомками вершины n. Заменить $[T_i, p_1]+[T_j, p_2]$ на $[T, p]$, где $T=$ **переход**(X), а p — указатель на вершину n.

(3) То же, что и (2), но вместо + стоит *.

(4) То же, что и (1), но вместо T_1 стоит T_3.

(5) То же, что и (1), но вместо T_1 стоит T_4.

(6) То же, что и (1), но вместо T_1 стоит T_2.

(7) Магазин теперь содержит $X([T, p])$, где p — указатель на некоторую вершину n. Заменить $([T, p])$ на $[T', p]$, где $T'=$ **переход**(X).

Рис. 9.6. Действия процессора по построению выхода.

Проследим за поведением процессора P, на вход которого поступает цепочка $a_1*(a_2+a_3)\$$. Для ясности буквы a снабжены

индексами. P выполняет такую последовательность тактов:

(1)	T_0	$a_1*(a_2+a_3)\$$
(2)	$T_0[T_1, p_1]$	$*(a_2+a_3)\$$
(3)	$T_0[T_1, p_1]*$	$(a_2+a_3)\$$
(4)	$T_0[T_1, p_1]*($	$a_2+a_3)\$$
(5)	$T_0[T_1, p_1]*([T_2, p_2]$	$+a_3)\$$
(6)	$T_0[T_1, p_1]*([T_2, p_2]+$	$a_3)\$$
(7)	$T_0[T_1, p_1]*([T_2, p_2]+[T_3, p_3]$	$)\$$
(8)	$T_0[T_1, p_1]*([T_2, p_4]$	$)\$$
(9)	$T_0[T_1, p_1]*([T_2, p_4])$	$\$$
(10)	$T_0[T_1, p_1]*[T_4, p_4]$	$\$$
(11)	$T_0[T_1, p_5]$	$\$$

Здесь левый столбец представляет содержимое магазина, правый — входную цепочку.

Исследуем некоторые интересные такты из этой последовательности. Переходя из конфигурации (1) в конфигурацию (2), P создает вершину n_1, помеченную символом a, и устанавливает указатель p_1 на эту вершину. Указатель p_1 запоминается в магазине вместе с LR(1)-таблицей T_1. Аналогично, переходя из конфигурации (4) в конфигурацию (5), P создает новую вершину n_2, помеченную символом a_2, и помещает в магазин указатель p_2 на эту вершину вместе с LR(1)-таблицей T_2. В конфигурации (7) образуется еще одна вершина n_3, помеченная символом a_3, и устанавливается указатель p_3 на эту вершину.

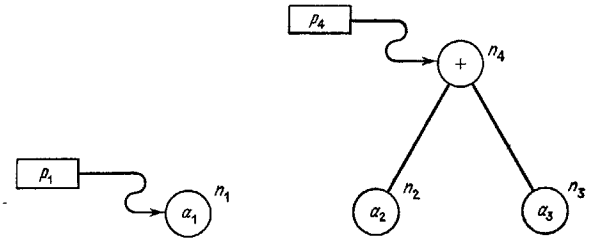


Рис. 9.7. Выход после достижения конфигурации (8).

Перейдя в конфигурацию (8), P создает выходной граф, изображенный на рис. 9.7. Здесь p_4 — указатель на вершину n_4. После того как процессор попадает в конфигурацию (11), выходной граф имеет вид, показанный на рис. 9.8. Здесь p_5 — указатель на вершину n_5. В конфигурации (11) действием таблицы T_1 на

входе $ служит **допуск**, и, следовательно, дерево, изображенное на рис. 9.8, является окончательным выходом. Это синтаксическое дерево для выражения $a_1 * (a_2 + a_3)$. □

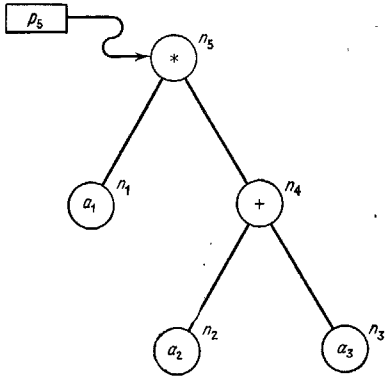


Рис. 9.8. Окончательный выход.

9.2.3. Детерминированный однопроходный восходящий перевод

Это первый из трех разделов, в которых показывается, как можно обобщить различные алгоритмы анализа для реализации СУ-схемы с помощью детерминированного МП-процессора вместо МП-преобразователя. Начнем с того, что дадим алгоритм реализации произвольной СУ-схемы с входной LR-грамматикой.

Алгоритм 9.1. Реализация СУ-схемы над LR-грамматикой.

Вход. Семантически однозначная СУ-схема $T = (N, \Sigma, \Delta, P, S)$ с входной LR(k)-грамматикой $G = (N, \Sigma, P, S)$ и входная цепочка $w \in \Sigma^*$.

Выход. Выходное дерево, крона которого служит переводом w.

Метод. Дерево строится МП-процессором M, моделирующим LR(k)-анализатор $\mathcal{A}$ для грамматики G. Как и $\mathcal{A}$, M запоминает в магазине (верхушка которого расположена справа) символы из $N \cup \Sigma$ и LR(k)-таблицы. Кроме того, M записывает под каждым нетерминалом указатель на порождаемый выходной граф. Действия **перенос**, **свертка** и **допуск** процессора M описываются так:

(1) Когда $\mathcal{A}$ переносит символ a в магазин, M делает то же самое.

(2) Если $\mathcal{A}$ свертывает $X_1 \ldots X_m$ в нетерминал A, то M делает следующее:

(а) Пусть $A \to X_1 \ldots X_m$, $w_0 B_1 w_1 \ldots B_r w_r$ — правило СУ-схемы, где символы B взаимно однозначно соответствуют тем символам X, которые являются нетерминалами.

(б) M удаляет из магазина $X_1 \ldots X_m$ вместе с промежуточными указателями LR(k)-таблиц и, если $X_1 \in N$, указатель, расположенный непосредственно под X_1.

(в) M создает новую вершину n, помечает ее символом A и помещает указатель на нее в магазин непосредственно под A.

(г) Метки прямых потомков вершины n читаются слева направо: $w_0 B_1 w_1 \ldots B_r w_r$. Для всех символов цепочек w создаются вершины. Вершина для B_i, $1 \leqslant i \leqslant r$, представляет собой вершину, на которую указывает указатель, расположенный в магазине процессора M непосредственно под X_j, где X_j — нетерминал, соответствующий B_i в данном конкретном правиле СУ-схемы.

(3) Если входная цепочка исчерпана (достигнут правый концевой маркер) и в магазине процессора M содержатся только pS и две LR(k)-таблицы, то M допускает тогда, когда допускает $\mathcal{A}$; p указывает на корень выходного дерева процессора M. □

Пример 9.6. Пусть алгоритм 9.1 применяется к СУ-схеме

$$S \to aSA,\ 0AS$$
$$S \to b,\ 1$$
$$A \to bAS,\ 1SA$$
$$A \to a,\ 0$$

а входной цепочкой служит *abbab*. Входная грамматика является SLR(1)-грамматикой. Мы не будем рассматривать LR(1)-таблицы, считая, что они правильно управляют разбором. Будем последовательно выписывать цепочки, содержащиеся в магазине МП-процессора, а затем приведем древовидную структуру, на которую указывает каждый из указателей. LR(1)-таблицы, содержащиеся в магазине, опускаем.

$$\begin{aligned}(e,\ abbab\$) &\vdash^2 (ab,\ bab\$)\\ &\vdash (ap_1S,\ bab\$)\\ &\vdash^2 (ap_1Sba,\ b\$)\\ &\vdash (ap_1Sbp_2A,\ b\$)\\ &\vdash (ap_1Sbp_2Ab,\ \$)\\ &\vdash (ap_1Sbp_2Ap_3S,\ \$)\\ &\vdash (ap_1Sp_4A,\ \$)\\ &\vdash (p_5S,\ \$)\end{aligned}$$

Деревья, построенные к моментам, когда процессор прошел третью, пятую, седьмую, восьмую и девятую конфигурации, изображены на рис. 9.9.

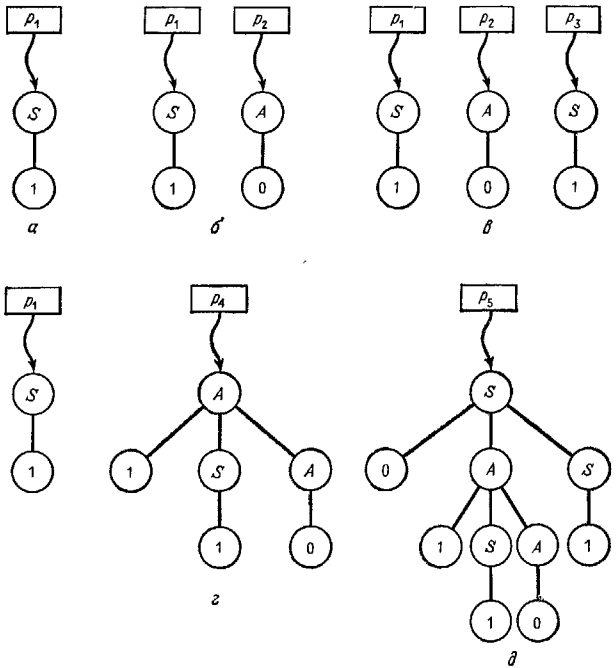


Рис. 9.9. Перевод с помощью МП-процессора.

Заметим, что когда bp_2Ap_3S свертывается в A, поддерево, на которое вначале указывал указатель p_3, появляется слева от поддерева, на которое указывает p_2, так как элемент перевода, связанный с $A \to bSA$, переставляет справа символы S и A. Аналогичная перестановка происходит при последней свертке. Можно видеть, что результатом работы процессора на рис. 9.9, *д* будет 01101. □

Теорема 9.4. *Алгоритм* 9.1 *правильно выполняет перевод входной цепочки в соответствии с данной* СУ-*схемой.*

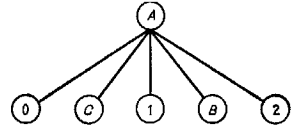
Доказательство. Элементарная индукция по порядку, в котором образуются указатели в алгоритме 9.1. □

Добавим, что если мы „реализуем" МП-процессор на обычной вычислительной машине с прямым доступом, то один такт МП-процессора осуществляется за конечное число шагов такой машины. Следовательно, время, затрачиваемое алгоритмом 9.1, является линейной функцией длины входной цепочки.

9.2.4. Детерминированный однопроходный нисходящий перевод

Предположим, что перед нами предсказывающий (нисходящий) анализатор. Превращение такого анализатора в транслятор требует подхода, несколько отличного от того, который применялся в случае восходящего анализатора. Допустим, что у нас есть СУ-схема с входной LL-грамматикой. В процессе разбора дерево строится сверху вниз, и на каждом этапе можно считать, что построено частичное дерево. Листья этого частичного дерева, помеченные нетерминалами, соответствуют нетерминалам, содержащимся в магазине предсказывающего анализатора, построенного алгоритмом 5.3. Развертка нетерминала эквивалентна созданию потомков для соответствующего листа дерева.

Стратегия при переводе состоит в том, чтобы хранить указатель на каждый лист „текущего" дерева с нетерминальной меткой. При обычном LL-разборе этот указатель хранится в магазине непосредственно под нетерминалом, соответствующим вершине, на которую указывает указатель. При развертке нетерминала по некоторому правилу для соответствующего элемента перевода создаются новые листья и на вершины с нетерминальными метками устанавливаются вновь созданные указатели, записанные в магазине. Указатель, записанный непосредственно под развертываемым нетерминалом, исчезает. Поэтому необходимо хранить где-то вне МП-процессора указатель на корень образующегося дерева. Например, если магазин содержит Ap и для развертки A применяется правило $A \to aBbCc$ с элементом перевода $0C1B2$, то МП-процессор заменит Ap на aBp_1bCp_2c. Если указатель p до развертки указывал на лист с меткой A, то после развертки A будет корнем поддерева



где p_1 указывает на вершину B, а p_2 — на вершину C.

Алгоритм 9.2. Нисходящая реализация СУ-схемы над LL-грамматикой.

Вход. Семантически однозначная СУ-схема $T = (N, \Sigma, \Delta, R, S)$ с входной LL(k)-грамматикой $G = (N, \Sigma, \Delta, P, S)$.

Выход. МП-процессор, порождающий дерево, крона которого служит переводом для w, где w — любая цепочка из $L(G)$.

Метод. Построим МП-процессор M, моделирующий LL(k)-анализатор $\mathcal{A}$ для грамматики G. M моделирует $\mathcal{A}$ следующим образом. Как и в алгоритме 9.1, мы опускаем операции работы с таблицами. Для M они те же, что и для $\mathcal{A}$.

(1) Первоначально в магазине процессора M (верхушка находится слева) записана цепочка Sp_r, где p_r — указатель на корневую вершину n_r.

(2) Если в верхушке магазина анализатора $\mathcal{A}$ содержится терминал и он сравнивается с текущим входным символом, после чего и тот и другой стираются, то в M происходит то же самое.

(3) Предположим, что $\mathcal{A}$ развертывает нетерминал A (возможно, вместе с соответствующей LL(k)-таблицей) по правилу $A \to X_1 \ldots X_m$ с элементом перевода $y_0B_1y_1 \ldots B_ry_r$ и указатель, расположенный непосредственно под A (легко показать, что он всегда существует), указывает на вершину n. Тогда M делает следующее:

(а) Для n он образует прямых потомков, помеченных слева направо символами цепочки $y_0B_1y_1 \ldots B_ry_r$.

(б) В магазине он заменяет A и расположенный под ним указатель на $X_1 \ldots X_m$, причем под теми из символов $X_1 \ldots X_m$, которые являются нетерминалами, записываются указатели. Указатель, расположенный под X_j, указывает на вершину, образованную для B_i, если X_j и B_i соответствуют друг другу в правиле

$$A \to X_1 \ldots X_m,\ y_0B_1y_1 \ldots B_ry_r$$

(4) Если магазин процессора M становится пустым к моменту достижения конца входной цепочки, то процессор допускает; выходом служит построенное к этому моменту дерево с корнем n_r. □

Пример 9.7. Рассмотрим пример из области перевода естественного языка. Известно, что СУ-схема дает точную модель перевода с английского языка на другой естественный язык — распространенный жаргон „пиг лэтин“ [1]). Перечислим правила,

[1]) В оригинале pig Latin.— *Прим. ред.*

которые неформально определяют перевод английского слова в соответствующее слово на „пиг лэтин“:

(1) Если слово начинается с гласной, добавляем к нему суффикс YAY.

(2) Если слово начинается с непустой цепочки согласных, помещаем все согласные до первой гласной в конец слова и добавляем суффикс AY.

(3) Однобуквенные слова не изменяем.

(4) Буква U, стоящая после Q, считается согласной.

(5) Буква Y в начале слова считается гласной, если за ней не следует гласная.

Приведем СУ-схему, включающую только правила (1) и (2). В качестве упражнения предлагаем учесть и остальные правила. Правила СУ-схемы таковы:

```
<слово>      → <согласные> <гласная> <буквы>,
               <гласная> <буквы> <согласные> 'AY'
<слово>      → <гласная> <буквы>,
               <гласная> <буквы> 'YAY'
<согласные>  → <согласная> <согласные>,
               <согласная> <согласные>
<согласные>  → <согласная>, <согласная>
<буквы>      → <буква> <буквы>, <буква> <буквы>
<буквы>      → e, e
<гласная>    → 'A', 'A'
<гласная>    → 'E', 'E'
               .
               .
               .
<гласная>    → 'U', 'U'
<согласная>  → 'B', 'B'
<согласная>  → 'C', 'C'
               .
               .
               .
<согласная>  → 'Z', 'Z'
<буква>      → <гласная>, <гласная>
<буква>      → <согласная>, <согласная>
```

Легко видеть, что входная грамматика является LL(2)-грамматикой. Найдем перевод, соответствующий входному слову "THE". Как и в предыдущих двух примерах, сначала выпишем конфигурации процессора, а затем покажем, как выглядит дерево на различных этапах его построения. Верхушка магазина все время находится слева.

(1)	THE$	⟨слово⟩p_1
(2)	THE$	⟨согласные⟩p_2⟨гласная⟩p_3⟨буквы⟩p_4
(3)	THE$	⟨согласная⟩p_5⟨согласные⟩p_6⟨гласная⟩p_3⟨буквы⟩p_4
(4)	THE$	T⟨согласные⟩p_6⟨гласная⟩p_3⟨буквы⟩p_4
(5)	HE$	⟨согласные⟩p_6⟨гласная⟩p_3⟨буквы⟩p_4
(6)	HE$	⟨согласная⟩p_7⟨гласная⟩p_3 ⟨буквы⟩p_4
(7)	HE$	H⟨гласная⟩p_3⟨буквы⟩p_4
(8)	E$	⟨гласная⟩p_3⟨буквы⟩p_4
(9)	E$	E⟨буквы⟩p_4
(10)	$	⟨буквы⟩p_4
(11)	$	e

Здесь левый столбец означает входную цепочку, правый — содержимое магазина.

Структура дерева после шагов 1, 2, 6 и 11 представлена соответственно на рис. 9.10, *а—г*. □

Как и в предыдущем разделе, легко доказать, что рассматриваемый алгоритм выполняет правильный перевод и что на обычной машине с прямым доступом его можно реализовать так, чтобы он работал за время, линейно зависящее от длины входной цепочки. Зафиксируем изложенное в следующей теореме.

Теорема 9.5. *Алгоритм* 9.2 *строит* МП-*процессор, порождающий в качестве выхода дерево, крона которого служит переводом входной цепочки.*

Доказательство. Индукцией можно доказать, что входная цепочка w вызывает стирание нетерминала A и указателя p в магазине тогда и только тогда, когда $(A, A) \Rightarrow_T^* (w, x)$, где x — крона поддерева, корнем которого служит вершина, на которую указывает p (после стирания символа A, указателя p и символов, в которые развертывается A). Детали доказательства оставляем в качестве упражнения. □

9.2.5. Перевод при наличии возвратов

Центральные идеи МП-процессора можно применить также и к алгоритмам разбора с возвратом. Для определенности рассмотрим, как можно обобщить анализирующую машину из разд. 6.1, чтобы она строила дерево. Основная новая идея заключается в том, что процессор должен обладать способностью „разрушать" поддеревья, которые он построил. Иными словами, некоторые поддеревья могут становиться недостижимыми (и, хотя здесь мы не будем разбирать детали реализации, на практике использованные для представления таких поддеревьев ячейки памяти возвращаются в общую доступную память).

Модифицируем анализирующую машину из разд. 6.1 так, чтобы предоставить ей возможность помещать в магазин указатели на вершины графа. (Напомним, что в анализирующей ма-

шине уже есть указатели на ее вход; они хранятся в одной ячейке памяти с информационным символом.) Правила работы

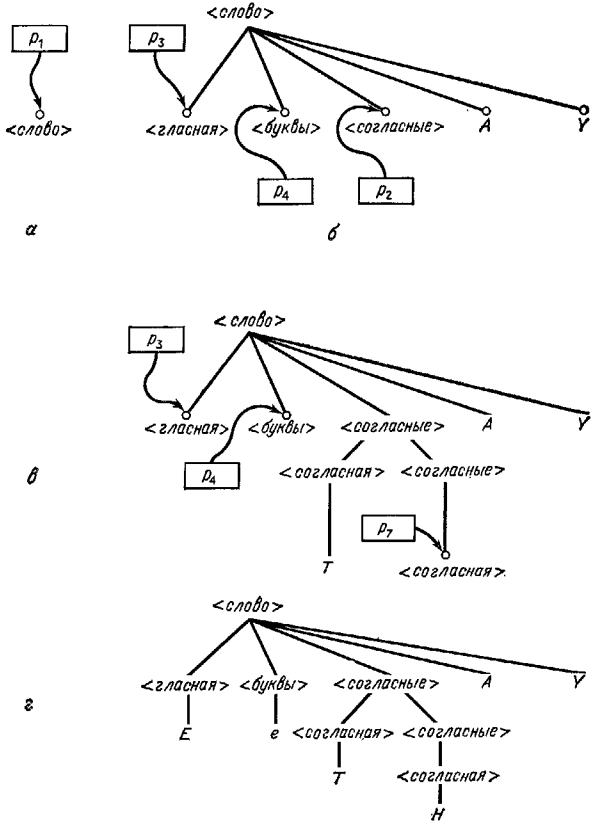


Рис 9.10. Перевод на «пиг лэтин».

с этими указателями те же, что и в случае МП-процессора, поэтому мы не будем здесь подробно на них останавливаться.

Прежде чем описать алгоритм перевода, связанный с анализирующей машиной, посмотрим, как переносится понятие СУ-

перевода на ОЯНРОВ-программы (программы, написанные на Обобщенном Языке Нисходящего Разбора с Ограниченными Возвратами) из разд. 6.1. Разумно предположить, что перевод связан с „вызовом" нетерминала тогда и только тогда, когда этот вызов успешен.

Пусть $P=(\mathrm{N}, \Sigma, R, S)$ — ОЯНРОВ-программа. Будем интерпретировать ОЯНРОВ-оператор $A \to a$, где $a \in \Sigma \cup \{e\}$, как попытку применить правило $A \to a$ в КС-грамматике. Тогда по аналогии с СУ-схемой можно было бы ожидать, что найдется связанная с этим правилом цепочка выходных символов. Иными словами, все правило будет иметь вид $A \to a, w$, где $w \in \Delta^*$; здесь Δ — „выходной алфавит".

Единственным другим ОЯНРОВ-оператором, способным породить перевод, будет $A \to B[C, D]$, где A, B, C и D принадлежат N. Можно считать, что здесь участвуют два правила КС-грамматики, а именно $A \to BC$ и $A \to D$. Если B и C приводят к успеху, то желательно, чтобы перевод для A включал переводы для B и C. Поэтому представляется естественным связать с правилом $A \to B[C, D]$ элемент перевода вида $wBxCy$ или $wCxBy$, где w, x и y принадлежат Δ^*. (Если $B=C$, то должно быть точно определено соответствие между нетерминалами правила и нетерминалами элемента перевода.)

Однако, если B приводит к неудаче, хотелось бы, чтобы перевод для A вызывал перевод для D. Поэтому второй элемент перевода, имеющий вид uDv, должен быть связан с правилом $A \to B[C, D]$. Дадим формальное определение такого метода описания перевода и посмотрим, как можно обобщить анализирующую машину, чтобы она выполняла такие переводы.

Определение. ОЯНРОВ-*программой с выходом* назовем пятерку $P=(\mathrm{N}, \Sigma, \Delta, R, S)$, где N, Σ и S те же, что для произвольной ОЯНРОВ-программы, Δ — конечное множество *выходных символов*, а R — множество *правил* вида:

(1) $A \to f, e$,

(2) $A \to a, y$, где $a \in \Sigma \cup \{e\}$ и $y \in \Delta^*$,

(3) (а) $A \to B[C, D], y_1By_2Cy_3, y_4Dy_5$,

(б) $A \to B[C, D], y_1Cy_2By_3, y_4Dy_5$,

где $y_i \in \Delta^*$.

Для каждого нетерминала существует не более одного из этих правил.

Определим отношения $\Rightarrow^n$ между нетерминалами и тройками вида $(u \uparrow v, y, r)$, где u и v принадлежат Σ^*, $y \in \Delta^*$ и r — это либо s, либо f. Здесь первая компонента — входная цепочка, в которой отмечено положение входной головки, вторая компонента — выходная цепочка, а третья — выход, означающий успех или неудачу.

(1) Если для A есть правило $A \to a, y$, то $A \Rightarrow^1 (a \uparrow u, y, s)$ для всех $u \in \Sigma^*$. Если v принадлежит Σ^*, но не начинается с a, то $A \Rightarrow^1 (\uparrow v, e, f)$.

(2) Если для A есть правило $A \to B[C, D], y_1Ey_2Fy_3, y_4Dy_5$, где $E=B$ и $F=C$ или наоборот, то справедливо следующее:

(а) Если $B \Rightarrow^{n_1} (u_1 \uparrow u_2u_3, x_1, s)$ и $C \Rightarrow^{n_2} (u_2 \uparrow u_3, x_2, s)$, то

$$A \Rightarrow^{n_1+n_2+1} (u_1u_2 \uparrow u_3, y_1x_1y_2x_2y_3, s)$$

при $E=B$, $F=C$ и

$$A \Rightarrow^{n_1+n_2+1} (u_1u_2 \uparrow u_3, y_1x_2y_2x_1y_3, s)$$

при $E=C$ и $F=B$. В случае $B=C$ будем предполагать, что соответствие между E и F, с одной стороны, и позициями, в которых стоят B и C, с другой стороны, указывается с помощью верхних индексов; например, $A \to B^{(1)}[B^{(2)}, D], y_1B^{(2)}y_2B^1y_3, y_4Dy_5$.

(б) Если $B \Rightarrow^{n_1} (u_1 \uparrow u_2, x, s)$ и $C \Rightarrow^{n_2} (\uparrow u_2, e, f)$, то

$$A \Rightarrow^{n_1+n_2+1} (\uparrow u_1u_2, e, f)$$

(в) Если $B \Rightarrow^{n_1} (\uparrow u_1u_2, e, f)$ и $D \Rightarrow^{n_2} (u_1 \uparrow u_2, x, s)$, то

$$A \Rightarrow^{n_1+n_2+1} (u_1 \uparrow u_2, y_4xy_5, s)$$

(г) Если $B \Rightarrow^{n_1} (\uparrow u, e, f)$ и $D \Rightarrow^{n_2} (\uparrow u, e, f)$, то

$$A \Rightarrow^{n_1+n_2+1} (\uparrow u, e, f)$$

Заметим, что если $A \Rightarrow^n (u \uparrow v, y, f)$, то $u=e$ и $y=e$. Иными словами, при неудаче входной указатель не сдвигается и не порождается никакого перевода. Следует также подчеркнуть, что в случае (2б) перевод символа B „аннулируется", если C приводит к неудаче.

Обозначим через $\Rightarrow^+$ объединение отношений $\Rightarrow^n$ для $n \geqslant 1$. *Переводом* $\tau(P)$, *определяемым программой* P, назовем множество $\{(w, x) \mid S \Rightarrow^+ (w \uparrow, x, s)\}$.

Пример 9.8. Напишем ОЯНРОВ-программу с выходом, осуществляющую перевод на „пиг лэтин", описанный в примере 9.7. Выход будем записывать строчными буквами. При переводе используются нетерминалы, связь которых с предыдущим примером показана в табл. 9.1. X и C_8 означают здесь цепочки нетерминалов.

Кроме того, мы будем пользоваться нетерминалами S и F с правилами $S \to e$ и $F \to f$. Наконец, правила для C_1, V и L должны быть такими, чтобы они соответствовали любой согласной, гласной или букве, определяя перевод, который является

Таблица 9.1

Пример 9.8	Пример 9.7
W	⟨слово⟩
C_1	⟨согласная⟩
C_2	⟨согласные⟩
C_3	⟨согласная⟩*
L_1	⟨буква⟩
L_2	⟨буквы⟩
V	⟨гласная⟩
X	⟨гласная⟩⟨буквы⟩

соответствующей буквой. В эти правила входят дополнительные нетерминалы, и мы их опускаем. Существенными правилами будут

$$W \to C_2[X, X],\ XC_2\ \text{'ay'},\ X\ \text{'yay'}$$
$$C_2 \to C_1[C_3, F],\ C_1C_3,\ F$$
$$C_3 \to C_1[C_3, S],\ C_1C_3,\ S$$
$$L_2 \to L_1[L_2, S],\ L_1L_2,\ S$$
$$X \to V[L_2, F],\ VL_2,\ F$$

Например, если рассматривается входная цепочка 'and', то легко видеть, что $V \Rightarrow^+ (a \uparrow nd, a, s)$ и $L_2 \Rightarrow^+ (nd \uparrow, nd, s)$. Поэтому $X \Rightarrow^+ (and \uparrow, and, s)$. Так как $C_1 \Rightarrow^+ (\uparrow and, e, f)$, то $C_2 \Rightarrow^+ (\uparrow and, e, f)$. Таким образом, $W \Rightarrow^+ (and \uparrow, andyay, s)$. □

Подобный перевод можно осуществить с помощью модифицированной анализирующей машины из разд. 6.1. Действие такой машины основано на следующем наблюдении. Если вызывается процедура A с правилом $A \to B[C, D]$, то A порождает перевод, только если B и C приводят к успеху или B приводит к неудаче, а D — к успеху. Опишем поведение модифицированной анализирующей машины в виде алгоритма.

Алгоритм 9.3. Реализация ОЯНРОВ-программы с выходом.

Вход. ОЯНРОВ-программа $P = (N, \Sigma, \Delta, R, S)$ с выходом.

Выход. Модифицированная анализирующая машина M, порождающая для любой входной цепочки w дерево с кроной x тогда и только тогда, когда (w, x) принадлежит $\tau(P)$.

Метод. Если не учитывать элементов перевода, отвечающих правилам из P, то можно считать, что у нас ОЯНРОВ-программа

P' в смысле разд. 6.1. Тогда можно по лемме 6.6 построить анализирующую машину M', распознающую $L(P')$.

Опишем неформально изменения, которые нужно внести в машину M' для получения модифицированной анализирующей машины M. Модифицированная машина M моделирует M' и тоже образует вершины в выходном дереве, помещая в магазин указатели на эти вершины.

Получив на вход цепочку w, M начинает работу в конфигурации (**начало**, $\uparrow w$, $(S, 0)p_r$). Здесь p_r — указатель на корневую вершину n_r.

(1) (а) Пусть $A \to B[C, D]$, $y_1Ey_2Fy_3$, y_4Dy_5 — правило для A. Предположим, что M выполняет следующую последовательность тактов, реализующую вызов процедуры A по правилу $A \to B[C, D]$:

$$(\textbf{начало},\ u_1 \uparrow u_2u_3,\ (A, i)\gamma \vdash (\textbf{начало},\ u_1 \uparrow u_2u_3,\ (B, j)(A, i)\gamma)$$
$$\vdash^* (\textbf{успех},\ u_1u_2 \uparrow u_3,\ (A, i)\gamma)$$
$$\vdash (\textbf{начало},\ u_1u_2 \uparrow u_3,\ (C, i)\gamma)$$

Затем M выполняет такую последовательность тактов, соответствующую последовательности тактов, выполненных ранее машиной M':

(9.2.1) $(\textbf{начало},\ u_1 \uparrow u_2u_3,\ (A, i)p_A\gamma')$

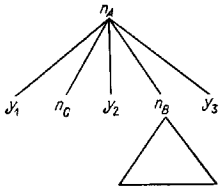
(9.2.2) $\vdash (\textbf{начало},\ u_1 \uparrow u_2u_3,\ (B, j)p_B(A, i)p_Bp_A\gamma')$

(9.2.3) $\vdash^* (\textbf{успех},\ u_1u_2 \uparrow u_3,\ (A, i)p_Bp_A\gamma')$

(9.2.4) $\vdash (\textbf{начало},\ u_1u_2 \uparrow u_3,\ (C, i)p_C\gamma')$

В конфигурации (9.2.1) указатель p_A на лист n_A расположен непосредственно под A. (Указатели на входную цепочку мы опускаем.) При переходе в конфигурацию (9.2.2) M образует новую вершину n_B, которая становится прямым потомком вершины n_A, и помещает указатель p_B на n_B непосредственно под A и над A. Затем M помещает B в верхушку магазина. В конфигурации (9.2.3) M возвращается к A в состоянии **успех**. При переходе в конфигурацию (9.2.4) M создает прямых потомков вершины n_A для всех символов цепочки $y_1Ey_2Fy_3$ и упорядочивает их слева направо. Вершина n_B становится вершиной для E или для F (для того символа, который соответствует B). Обозначим вершину для другого из символов E и F через n_C. В конфигурации (9.2.4) M заменяет Ap_Bp_A на Cp_C, где p_C — указатель на n_C. Таким образом, если $E = C$ и $F = B$, то

в конфигурации (9.2.4) вершина n_A является корнем поддерева



(б) Если, с другой стороны, B вырабатывает **неудачу** в правиле $A \to B[C, D]$, то M выполняет такую последовательность тактов:

(9.2.1) (**начало**, $u_1 \uparrow u_2u_3$, $(A, i)\, p_A\gamma'$)

(9.2.2) $\vdash$ (**начало**, $u_1 \uparrow u_2u_3$, $(B, j)\, p_B\, (A, i)\, p_Bp_A\gamma'$)

(9.2.5) $\vdash^*$ (**неудача**, $u_1 \uparrow u_2u_3$, $(A, i)\, p_Bp_A\gamma'$)

(9.2.6) $\vdash$ (**начало**, $u_1 \uparrow u_2u_3$, $(D, i)\, p_D\gamma'$)

При переходе из (9.2.5) в (9.2.6) M сначала удаляет из выходного дерева вершину n_B и всех ее прямых потомков, используя для запоминания n_B указатель p_B, записанный под A. Затем для всех символов цепочки y_4Dy_5 в порядке слева направо M образует прямых потомков вершины n_A и заменяет в верхушке магазина Ap_Bp_A на Dp_D, где p_D — указатель на вершину, построенную для D.

(2) Если $A \to a$, y — правило $a \in \Sigma \cup \{e\}$, то M выполняет один из следующих тактов:

(а) (**начало**, $u \uparrow av$, $(A, i)\, p_A\gamma$) $\vdash$ (**успех**, $ua \uparrow v$, γ). На этом шаге для каждого символа цепочки y создается прямой потомок вершины n_A (вершина, на которую указывает p_A). Если $y = e$, то прямым потомком станет одна вершина, помеченная символом e.

(б) (**начало**, $u \uparrow v$, $(A, i)\, p_A\gamma$) $\vdash$ (**неудача**, $u' \uparrow v'$, γ), где $|u'| = i$ и v не начинается с a.

(3) Если $A \to f$, e — правило, то M выполняет такт (2б).

(4) Если M' прочитывает всю входную цепочку w и стирает все символы в магазине, то переводом цепочки w будет построенное к этому моменту дерево с корнем n_r. □

Теорема 9.6. *Алгоритм 9.3 определяет модифицированную анализирующую машину, порождающую для входной цепочки w дерево, крона которого является переводом цепочки w.*

Доказательство. Доказательство представляет собой простую индукцию по числу тактов, совершаемых машиной M.

Предположение индукции состоит в том, что если M, начав работу в состоянии **начало** и имея в верхушке магазина символ A и указатель p на вершину n и цепочку uv справа от входной головки, исчерпывает вход u, в результате удаляя из магазина A и p, и заканчивает работу в состоянии **успех**, то вершина, на которую указывает p, будет корнем поддерева с такой кроной y, что $A \Rightarrow^* (u \uparrow v, y, s)$. □

Пример 9.9. Посмотрим, как анализирующая машина осуществляет перевод, описанный в примере 9.8. Мы примем следующую форму записи (табл. 9.2). За конфигурациями будет

Таблица 9.2

Состояние	Входная цепочка	Содержимое магазина
начало	$\uparrow$ and	Wp_1
начало	$\uparrow$ and	$C_2p_3Wp_2p_1$
начало	$\uparrow$ and	$C_1p_8C_2p_3p_2Wp_2p_1$
	.	
	.	
	.	
неудача	$\uparrow$ and	$C_2p_8p_2Wp_2p_1$
начало	$\uparrow$ and	$Fp_4Wp_2p_1$
неудача	$\uparrow$ and	Wp_2p_1
начало	$\uparrow$ and	Xp_5
начало	$\uparrow$ and	$Vp_6Xp_6p_5$
	.	
	.	
	.	
успех	$a \uparrow$ nd	Xp_6p_5
начало	$a \uparrow$ nd	L_2p_7
начало	$a \uparrow$ nd	$L_1p_8L_2p_8p_7$
	.	
	.	
	.	
успех	an $\uparrow$ d	$L_2p_8p_7$
начало	an $\uparrow$ d	L_2p_9
начало	an $\uparrow$ d	$L_1p_{10}L_2p_{10}p_9$
	.	
	.	
	.	
успех	and $\uparrow$	$L_2p_{10}p_9$
начало	and $\uparrow$	L_2p_{11}
начало	and $\uparrow$	$L_1p_{12}L_2p_{12}p_{11}$
	.	
	.	
	.	
неудача	and $\uparrow$	$L_2p_{12}p_{11}$
начало	and $\uparrow$	Sp_{13}
успех	and $\uparrow$	e

следовать построенное к этому моменту дерево. Такты, представляющие опознание согласных, гласных и букв (нетерминалы C_1, V и L_1), опускаем.

Выходное дерево после каждой четвертой конфигурации показано на рис. 9.11. □

Не вдаваясь в подробности, отметим только, что методы, представленные в алгоритме 9.3, применимы и к другим алгоритмам нисходящего разбора, таким, как алгоритм 4.1 и построение перевода в виде ОЯНРОВ-программы.

УПРАЖНЕНИЯ

9.2.1. Постройте МП-преобразователь, реализующий простую СУ-схему

$$
\begin{array}{lll}
E \to E+T, & ET & \text{'ADD'}; \\
E \to T, & T & \\
T \to T*F, & TF & \text{'MPY'}; \\
T \to F, & F & \\
F \to F \uparrow P, & FP & \text{'EXP'}; \\
F \to P, & P & \\
P \to (E), & E & \\
P \to a, & & \text{'LOAD } a\text{;'}
\end{array}
$$

Преобразователь должен опираться на SLR(1)-алгоритм разбора. Напишите последовательности выходных цепочек, возникающих при обработке выражений

(а) $a \uparrow a*(a+a)$,

(б) $a+a*a \uparrow a$.

9.2.2. Сделайте упр. 9.2.1 для случая МП-процессора, использующего LL(1)-алгоритм разбора. Если понадобится, видоизмените входную грамматику.

9.2.3. Покажите, каким образом МП-преобразователь, работающий как LL(1)-анализатор, будет выполнять перевод следующих цепочек в соответствии с простой СУ-схемой:

$$
\begin{array}{lll}
E \to TE', & TE' & \\
E' \to +TE', & T & \text{'ADD'};\ E' \\
E' \to e, & e & \\
T \to FT', & FT' & \\
T' \to *FT', & F & \text{'MPY;'}\ T' \\
T' \to e, & e & \\
F \to (E), & E & \\
F \to a, & & \text{'LOAD } a\text{;'}
\end{array}
$$

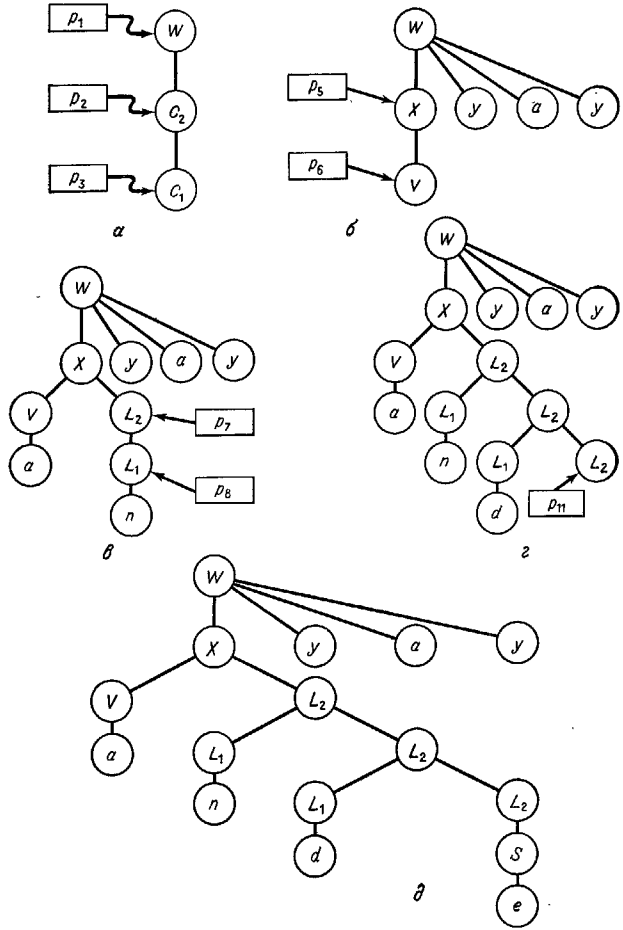


Рис. 9.11. Перевод с «пиг латин» с помощью анализирующей машины. (Здесь e — пустая цепочка, перевод символа S.)

(а) $a*(a+a)$,

(б) $((a+a)*a)+a$.

9.2.4. Покажите, как МП-процессор будет выполнять перевод цепочки *abbbaaaa* в соответствии с СУ схемой

$$\begin{array}{lll} S \to aA^{(1)}A^{(2)}, & 0A^{(2)}A^{(1)}1 \\ S \to b, & 2 \\ A \to bS^{(1)}S^{(2)}, & 1S^{(1)}S^{(2)}0 \\ A \to a, & e \end{array}$$

если разбор проводится с помощью

(а) LL(1)-алгоритма,

(б) LR(1)-алгоритма.

9.2.5. Покажите, что не существует СУ-схемы для перевода, описанного в примере 9.1, если язык порождается грамматикой $E \to a+E \mid a*E \mid a$.

9.2.6. Опишите формально построение МП-преобразователя M из теоремы 9.1.

***9.2.7.** Докажите, что каждый ДМП-преобразователь определяет постфиксный простой синтаксически управляемый перевод языка, порождаемого LR(1)-грамматикой. *Указание:* Покажите, что каждый ДМП-преобразователь можно преобразовать к нормальной форме, аналогичной той, которая была определена в разд. 8.2.1.

***9.2.8.** Покажите, что существуют такие переводы $T = \{(x\$, y)\}$, что T можно определить ДМП-преобразователем, но $\{(x, y) \mid (x\$, y) \in T\}$ нельзя определить никаким ДМП преобразователем. Сопоставьте этот результат с упр. 2.6.20(б).

9.2.9. Проведите формальное доказательство теоремы 9.3.

9.2.10. Докажите теорему 9.4.

9.2.11. Расширьте СУ-схему из примера 9.7 так, чтобы включить в нее правила (3)—(5) из определения перевода на „пиг лэтин".

9.2.12. Можно ли заменить СУ-схему из примера 9.7 простой СУ-схемой, если принять, что в английском языке нет слов, начинающихся более чем с четырех согласных? Изменится ли число правил СУ схемы?

9.2.13. Покажите, как анализирующая машина с указателями будет выполнять перевод цепочки *abb* согласно ОЯНРОВ-программе с выходом

$$\begin{array}{lll} S \to A[B, C], & 0BA, & 11C \\ A \to a, & 0 & \\ B \to S[C, A], & 0CS, & 1A \\ C \to b, & 1 & \end{array}$$

****9.2.14.** Напишите такую ОЯНРОВ-программу с выходом, что $\tau(P) = \{(x, y) \mid x$ -цепочка, содержащая n символов a и n символов b, $n \geqslant 0$ и $y = a^n b^n\}$. Постройте по P модифицированную анализирующую машину M, для которой $\tau(M) = \tau(P)$. Существует ли (а) ЯНРОВ-программа с выходом, (б) СУ-схема, определяющая тот же перевод?

9.2.15. Докажите теорему 9.5.

9.2.16. Докажите теорему 9.6.

***9.2.17.** Обобщите понятие процессора с указателями на граф и дайте алгоритмы перевода для СУ-схем, основанные на следующих алгоритмах:

(а) алгоритм 4.1,
(б) алгоритм 4.2,
(в) алгоритм Кока—Янгера—Касами,
(г) алгоритм Эрли,
(д) двухмагазинный анализатор (разд. 6.2),
(е) анализатор Флойда—Эванса.

Проблема для исследования

9.2.18. При реализации перевода мы часто заинтересованы в его эффективности. Разработайте методы оптимизации, аналогичные методам гл. 7, позволяющие находить эффективные трансляторы для полезных классов синтаксически управляемых переводов.

Упражнения на программирование

9.2.19. Напишите программу, которая получает на вход простую СУ-схему над LL(1)-грамматикой и выдает транслятор, реализующий эту СУ-схему.

9.2.20. Напишите программу, строящую транслятор, который реализует постфиксную простую СУ-схему над LALR(1)-грамматикой.

9.2.21. Напишите программу, которая строит транслятор, реализующий произвольную СУ-схему над LALR(1)-грамматикой.

Замечания по литературе

Впервые возможность реализации простой СУ-схемы с входной LL(k)-грамматикой на МП-преобразователе была доказана в работе Льюиса и Стирнза [1968]. Они также показали, что простую постфиксную СУ-схему над LR(k)-грамматикой можно реализовать на ДМП-преобразователе и любой перевод, выполняемый ДМП-преобразователем, можно эффективно описать простой постфиксной СУ-схемой над LR(k)-грамматикой (упр. 9.2.7). Понятие МП-процессора введено Ахо и Ульманом [1969а].

Во многих компиляторах компиляторов и системах построения компиляторов выходной компилятор описывается аналогично тому, как описывается схема синтаксически управляемого перевода. Синтаксис языка, для которого строится компилятор, определяется с помощью контекстно-свободной грамматики. Вводятся и семантические программы, связанные с каждым правилом. Полученный выходной компилятор можно промоделировать МП-процессором: входная цепочка анализируется компилятором, а для вычисления выхода вызываются семантические программы. ЯНРОВ с выходом представляет собой упрощенную модель системы построения компиляторов TMG [Мак-Клюр, 1965]. Мак-Илрой [1972] реализовал расширение TMG, в котором допускаются правила разбора типа ОЯНРОВ-правил и возможно моделирование поведения восходящих анализаторов. ОЯНРОВ с выходом—это упрощенная модель семейства компиляторов компиляторов META [Шорре, 1964].

Два класса простых СУ-схем, каждый из которых включает в себя простые СУ-схемы над LL-грамматиками и постфиксные простые СУ-схемы над LR-грамматиками, упоминаются Льюисом и Стирнзом [1968], а также Ахо и Ульманом [1972ж]. Каждый из этих классов можно реализовать на ДМП-преобразователе.

9.3. ОБОБЩЕННЫЕ СХЕМЫ ПЕРЕВОДОВ

В этом разделе мы рассмотрим, как можно естественным образом расширить обсуждаемые ранее схемы перевода с тем, чтобы получить возможность выполнять более широкий и более полезный класс переводов. Будем считать, что в общем случае элементом перевода, связанным с правилом, может быть любой тип функции. Основные расширения заключаются в следующем: допускаются несколько переводов в каждой вершине дерева разбора, используются переменные, принимающие в качестве значений не только цепочки, разрешается зависимость переводов в вершине от переводов не только в ее прямых потомках, но и в ее прямом предке.

Мы обсудим также важный вопрос о разбиении на фазы при выполнении переводов в различных вершинах.

9.3.1. Мультисхемы переводов

Наше первое расширение СУ-схемы будет допускать в каждой вершине дерева разбора несколько переводов, имеющих значениями цепочки. Как и в СУ-схеме, каждый перевод зависит от переводов прямых потомков рассматриваемой вершины. Однако элементами перевода могут быть произвольные цепочки выходных символов и символов, представляющих переводы в потомках. Таким образом, символы перевода могут повторяться.

Определение. *Обобщенной схемой синтаксически управляемого перевода* (ОСУ схемой) называется шестерка $T=(N, \Sigma, \Delta, \Gamma, R, S)$, где N, Σ и Δ—соответственно конечные множества *нетерминалов*, *входных символов* и *выходных символов*, а Γ—конечное множество *символов перевода* вида A_i; здесь $A \in N$ и i—целое число. Мы будем полагать, что $S_1 \in \Gamma$. S—*начальный символ*, элемент множества N. Наконец, R—множество *правил перевода* вида

$$A \to \alpha,\ A_1=\beta_1,\ A_2=\beta_2,\ \ldots,\ A_m=\beta_m,$$

удовлетворяющих следующим условиям:

(1) $A_j \in \Gamma$ для $1 \leqslant j \leqslant m$,

(2) каждый символ, входящий в $\beta_1, \ldots, \beta_m$, либо принадлежит Δ, либо является таким символом $B_k \in \Gamma$, что B—нетерминал, появляющийся в α,

(3) если α имеет более одного вхождения символа B, то каждый символ B_k во всех β соотнесен верхним индексом с конкретным вхождением B.

Мы будем называть $A \to \alpha$ *входным правилом вывода*, A_i—*переводом* нетерминала A и $A_i=\beta_i$—*элементом перевода*, связанным с этим правилом перевода. Если через P обозначить множество *входных правил вывода* всех правил перевода, то $G=(N, \Sigma, P, S)$ будет *входной грамматикой* для T. Если в ОСУ-схеме T нет двух правил перевода с одинаковым входным правилом вывода, то ее называют *семантически однозначной*.

Выход ОСУ-схемы определим снизу вверх. С каждой внутренней вершиной n дерева разбора (во входной грамматике), помеченной A, свяжем одну цепочку для каждого A_i из Γ. Эта цепочка называется *значением* (или *переводом*) символа A_i в вершине n. Каждое значение вычисляется подстановкой значений

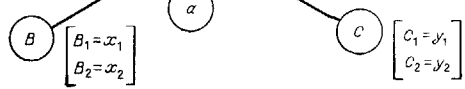


Рис. 9.12. Часть дерева разбора.

символов перевода данного элемента перевода $A_i=\beta_i$, определенных в прямых потомках вершины n. Например, пусть

$$A \to BaC,\quad A_1=bB_1C_2B_1,\quad A_2=C_1cB_2$$

—правило перевода в ОСУ-схеме и в вершине A дерева вывода используется входное правило вывода $A \to BaC$ (рис. 9.12).

Тогда, если значения символов B_1, B_2 в вершине B и C_1, C_2 в вершине C таковы, как указано на рис. 9.12, то значением символа A_1, определенным в вершине A, будет $bx_1y_2x_1$, а значением символа A_2 в этой же вершине будет y_1cx_2.

Переводом $\tau(T)$, *определяемым* ОСУ-схемой T, назовем множество $\{(x, y) \mid x$ имеет дерево разбора во входной грамматике для T и y—значение символа перевода S_1 в корне этого дерева$\}$.

Пример 9.10. Пусть $T = (\{S\}, \{a\}, \{a\}, \{S_1, S_2\}, R, S)$—ОСУ-схема, в которой R состоит из правил

$$S \to aS, \quad S_1 = S_1S_2S_2a, \quad S_2 = S_2a$$
$$S \to a, \quad S_1 = a, \quad S_2 = a$$

Тогда $\tau(T) = \{(a^n, a^{n^2}) \mid n \geqslant 1\}$. Например, a^4 имеет дерево разбора, изображенное на рис. 9.13, *а*. Значения двух переводов в каждой внутренней вершине показаны на рис. 9.13, *б*.

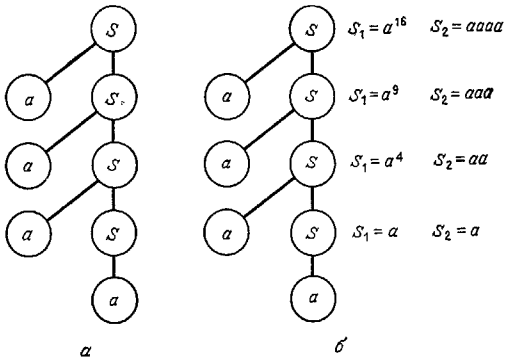


Рис. 9.13. Обобщенная схема синтаксически управляемого перевода.

Например, для того чтобы вычислить значение символа S_1 в корне, надо подставить в выражение $S_1S_2S_2a$ значения для S_1 и S_2 в вершине под корнем. Этими значениями являются соответственно a^9 и a^3. Для доказательства того, что $\tau(T)$ отображает a^n в a^{n^2}, достаточно заметить, что $(n+1)^2 = n^2 + 2n + 1$. □

Пример 9.11. Приведем пример формального дифференцирования выражений, включающих константы 0 и 1, переменную x и функции синус, косинус, $+$ и $*$. Такие выражения порождает

грамматика

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \sin(E) \mid \cos(E) \mid x \mid 0 \mid 1$$

Свяжем с каждым из E, T и F два перевода, обозначенные индексами 1 и 2. Индекс 1 указывает, что выражение не дифференцировано; 2 указывает, что выражение продифференцировано. Интересующий нас перевод—это E_2. Законы дифференцирования таковы:

$$d(f(x) + g(x)) = df(x) + dg(x)$$
$$d(f(x)\,g(x)) = f(x)\,dg(x) + g(x)\,df(x)$$
$$d\sin(f(x)) = \cos(f(x))\,df(x)$$
$$d\cos(f(x)) = -\sin(f(x))\,df(x)$$
$$dx = 1$$
$$d0 = 0$$
$$d1 = 0$$

Эти законы реализуются ОСУ-схемой

$E \to E + T$	$E_1 = E_1 + T_1$
	$E_2 = E_2 + T_2$
$E \to T$	$E_1 = T_1$
	$E_2 = T_2$
$T \to T * F$	$T_1 = T_1 * F_1$
	$T_2 = T_1 * F_2 + (T_2) * F_1$
$T \to F$	$T_1 = F_1$
	$T_2 = F_2$
$F \to E$	$F_1 = (E_1)$
	$F_2 = (E_2)$
$F \to \sin(E)$	$F_1 = \sin(E_1)$
	$F_2 = \cos(E_1) * (E_2)$
$F \to \cos(E)$	$F_1 = \cos(E_1)$
	$F_2 = -\sin(E_1) * (E_2)$
$F \to x$	$F_1 = x$
	$F_2 = 1$
$F_1 \to 0$	$F_1 = 0$
	$F_2 = 0$
$F \to 1$	$F_1 = 1$
	$F_2 = 0$

В качестве упражнения предлагаем доказать, что если $(\alpha, \beta) \in \tau(T)$, то β — производная от α. Цепочка β может содержать избыточные скобки.

Дерево вывода для $\sin(\cos(x)) + x$ изображено на рис. 9.14. Значения символов перевода в каждой внутренней вершине перечислены в табл. 9.3. □

Таблица 9.3

Вершины	E_1, T_1 или F_1	E_2, T_2 или F_2
n_3, n_2	x	1
n_{12}, n_{11}, n_{10}	x	1
n_9, n_8, n_7	$\cos(x)$	$-\sin(x) * (1)$
n_6, n_5, n_4	$\sin(\cos(x))$	$\cos(\cos(x)) * (-\sin(x) * (1))$
n_1	$\sin(\cos(x)) + x$	$\cos(\cos(x)) * (-\sin(x) * (1)) + 1$

Реализация ОСУ-схемы не сильно отличается от реализации СУ-схемы с помощью алгоритмов 9.1 и 9.2. Обобщим теперь алгоритм 9.1 так, чтобы на выходе иметь не дерево, а ориентированный ациклический граф. Тогда можно будет „проходить" по ориентированному ациклическому графу так, чтобы получался нужный перевод.

Алгоритм 9.4. Выполнение ОСУ-схемы снизу вверх.

Вход. Семантически однозначная ОСУ-схема $T = (N, \Sigma, \Delta, \Gamma, R, S)$ с входной LR(k)-грамматикой G и входная цепочка $x \in \Sigma^*$.

Выход. Ориентированный ациклический граф[1]), из которого можно получить такой выход y, что $(x, y) \in \tau(T)$.

Метод. Пусть $\mathcal{A}$ — LR(k)-анализатор для G. Построим МП-процессор M с концевым маркером, который будет моделировать $\mathcal{A}$ и строить граф. Если $\mathcal{A}$ содержит в своем магазине нетерминал A, то M поместит под A по одному указателю для каждого символа перевода $A_i \in \Gamma$. Таким образом, в графе будет столько вершин, соответствующих вершине A в дереве разбора, сколько существует переводов для A, т. е. символов A_i в Γ. Действия M таковы:

(1) Если $\mathcal{A}$ выполняет перенос, то M делает то же самое.

(2) Предположим, что $\mathcal{A}$ собирается выполнить свертку в соответствии с правилом $A \to \alpha$ с элементами перевода $A_1 = \beta_1, \ldots, A_m = \beta_m$. В этот момент в верхушке магазина процессора M находится α, а непосредственно под каждым нетерминалом в α

[1]) Для краткости мы до конца раздела будем писать просто „граф", опуская слова „ориентированный ациклический". — *Прим. перев.*

находятся указатели на каждый из переводов этого нетерминала. Когда M делает свертку, сначала порождаются m вершин, по одной на каждый символ перевода A_i. Прямые потомки этих вершин определяются символами из $\beta_1, \ldots, \beta_m$. Порождаются

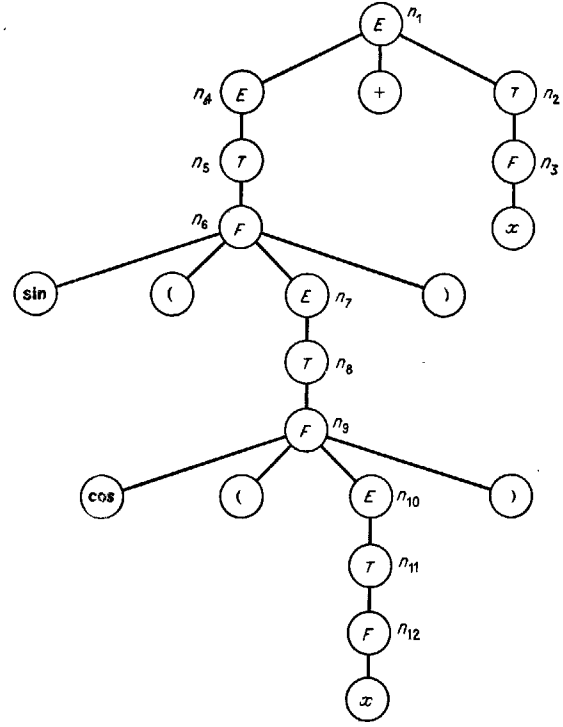


Рис. 9.14. Дерево вывода для $\sin(\cos(x)) + x$.

новые вершины для выходных символов. Вершиной для символа перевода $B_k \in \Gamma$ служит вершина, отмеченная тем указателем под нетерминалом B в α, который представляет k-й перевод для B. (Как обычно, если B входит в α более чем один раз, то каждое его конкретное появление будет помечено в элементе перевода дополнительным верхним индексом.) При свертке M

заменяет цепочку α и ее указатели символом A и m указателями на переводы для A. Например, допустим, что $\mathcal{A}$ выполняет свертку в соответствии с входным правилом вывода правила перевода

$$A \to BaC, \quad A_1 = bB_1C_2B_1, \quad A_2 = C_1cB_2$$

Тогда в верхушке своего магазина M имеет цепочку $p_{B_2}p_{B_1}Bap_{C_2}p_{C_1}C$ (C — самый верхний символ), где p — указатели на вершины, представляющие переводы. При свертке M заменяет эту цепочку цепочкой $p_{A_2}p_{A_1}A$, где p_{A_1} и p_{A_2} — указатели на первый и второй переводы для A. После свертки получается граф, изображенный

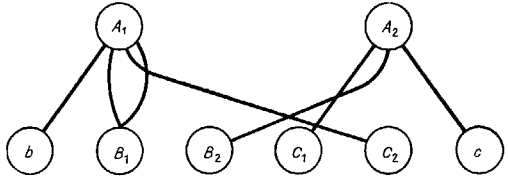


Рис 9.15. Часть выходного ориентированного ациклического графа.

на рис. 9.15. Предполагается, что px_i указывает на вершину для X_i.

(3) Если МП-процессор M достиг конца входной цепочки и его магазин содержит S и некоторые указатели, то указателем на вершину для S_1 является корень нужного выходного графа. □

Подробный пример мы рассмотрим после того, как обсудим интерпретацию графа. Очевидно, что каждая вершина графа „представляет" значение символа перевода в некоторой вершине дерева разбора. Но как это значение можно получить из графа? Из определения перевода, связанного с графом, должно быть ясно, что значение, представляемое вершиной n, должно быть конкатенацией слева направо значений, представляемых прямыми потомками вершины n. Отметим, что в действительности два или более прямых потомков могут быть одной и той же вершиной. В этом случае значение вершины должно повториться.

Учитывая все сказанное, можем заключить, что нужный выход порождается следующим методом прохождения по графу.

Алгоритм 9.5. Перевод графа.

Вход. Граф с помеченными листьями и единственным корнем.

Выход. Последовательность символов, помечающая листья.

Метод. Мы будем использовать рекурсивную процедуру $R(n)$, где n — вершина графа. Сначала вызывается $R(n_0)$, где n_0 — корень.

Процедура $R(n)$.

(1) Пусть $a_1, a_2, \ldots, a_m$ — дуги, исходящие из n в порядке записи. Совершаем шаг (2) для $a_1, a_2, \ldots, a_m$ по порядку.

(2) (а) Пусть a_i — текущая дуга. Если a_i входит в лист, выдаем метку этого листа.

(б) Если a_i входит во внутреннюю вершину n', выполняем $R(n')$. □

Теорема 9.7. *Если алгоритм* 9.5 *применяется к графу, построенному алгоритмом* 9.4, *то выходом алгоритма* 9.5 *является перевод цепочки* x, *подаваемой на вход алгоритма* 9.4.

Доказательство. Каждая вершина n, вырабатываемая алгоритмом 9.4, очевидным образом соответствует значению символа перевода в данной вершине дерева разбора для x. Индукцией по высоте вершины можно показать, что процедура $R(n)$ вырабатывает значение этого символа перевода. □

Пример 9.12. Применим алгоритмы 9.4 и 9.5 к ОСУ-схеме из примера 9.10, задав на входе цепочку *аааа*.

Последовательность конфигураций процессора приведена в табл. 9.4 (как обычно, LR(1)-таблицы опущены).

Таблица 9.4

	Содержимое магазина	Входная цепочка
(1)	e	$aaaa\$$
(2)	a	$aaa\$$
(3)	aa	$aa\$$
(4)	aaa	$a\$$
(5)	$aaaa$	$\$$
(6)	$aaap_1p_2S$	$\$$
(7)	aap_3p_4S	$\$$
(8)	ap_5p_6S	$\$$
(9)	p_7p_8S	$\$$

Графы, построенные после шагов 6, 7 и 9, приведены на рис. 9.16. Вершины слева соответствуют значениям для S_1, а справа — для S_2.

Применение алгоритма 9.5 к графу на рис. 9.16, *в* требует много вызовов процедуры $R(n)$. Работа алгоритма начинается с вершины n_1, поскольку она соответствует S_1. Выпишем последовательность вызовов $R(n)$ и генераций выходного символа a (обращение к $R(n_i)$ обозначено просто как n_i):

$$n_1n_3n_5n_7an_8an_8aan_6n_8aan_6n_8aaan_4n_6n_8aaan_4n_6n_8aaaa \quad \square$$

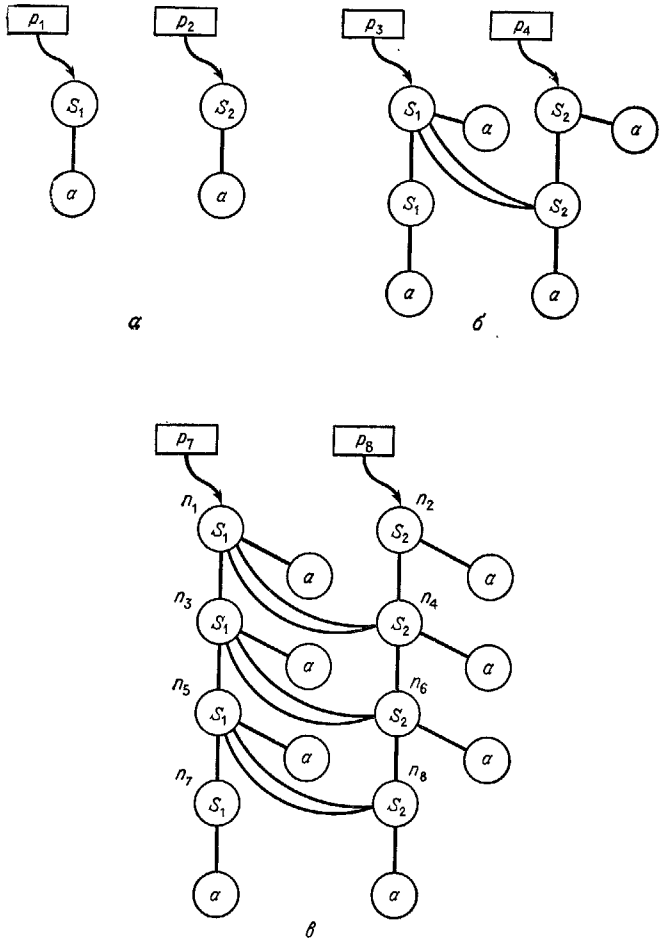


Рис. 9.16. Граф, построенный алгоритмом 9.4.

9.3.2. Разновидности переводов

До сих пор мы рассматривали в схеме перевода переменные, принимающие в качестве значений только цепочки. Те же принципы, что привели к определению СУ-схем и ОСУ-схем, позволят определить и реализовать схемы перевода, содержащие также логические и арифметические переменные.

Цепочки, вырабатываемые в процессе генерации кода, можно разбить на несколько различных классов:

(1) Машинные коды, или команды ассемблера, которые должны быть выходом компилятора.

(2) Сообщения об ошибках; это также выход компилятора, но не в том выходном потоке, что (1).

(3) Команды, указывающие, что некоторые операции должны производиться над данными, с которыми имеет дело сам компилятор.

К классу (3) относятся команды, осуществляющие операции по организации информации, арифметические операции над переменными, используемыми компилятором не в механизме синтаксического анализа, и операции по распределению памяти и генерации новых меток для выходных операторов. Обсуждение того, когда выполняются эти операции, мы отложим до следующего раздела.

Вот еще несколько примеров типа переводов, которые могут оказаться полезными при генерации кода:

(1) элементы конечного множества видов (вещественный, целый и т. д.), указывающие на вид арифметического выражения,

(2) цепочки, представляющие метки некоторых операторов при компиляции структур управления (например, **if — then — else**),

(3) целые, указывающие высоту вершины в дереве разбора.

Мы обобщим типы формул, которые можно применить в СУ-схеме для вычисления перевода. Разумеется, работая с числами или логическими переменными, мы будем употреблять для нахождения переводов логические и арифметические операторы. Удобно использовать также условные выражения вида **if** B **then** E_1 **else** E_2, где B — логическое выражение, а E_1 и E_2 — произвольные выражения, включая условные.

Например, в правиле вывода $A \to BC$ нетерминал B мог бы иметь логический перевод B_1, а C — два перевода C_1 и C_2 с цепочками в качестве значений. Формулой перевода для A_1 могла бы быть **if** B_1 **then** C_1 **else** C_2. Это означает, что если левый прямой потомок вершины, перевод A_1 которого вычисляется, имеет перевод $B_1 =$ **истина**, то в качестве A_1 надо взять первый перевод C_1 правого прямого потомка; иначе надо взять C_2. Другой случай: нетерминал B мог бы иметь перевод B_2, принимающий целое значение, и формулой для A_2 могла бы быть **if** $B_2 =$

$= 1$ **then** C_1 **else** C_2. Это приведет к тому, что значение A_2 будет то же, что и у C_2, если значение B_2 не 1.

Мы увидим, что не трудно обобщить алгоритм 9.4, работающий снизу вверх, так, чтобы в качестве переводов допускались числа, логические значения или элементы некоторого конечного множества. В схеме работы снизу вверх формулы для этих переменных (и для переменных, имеющих значениями цепочки) вычисляются только тогда, когда известны все аргументы.

В самом деле, часто все связанные с вершиной переводы, кроме одного, будут логическими, целыми или элементами конечного множества. Если этот один (имеющий значением цепочку) перевод можно получить с помощью правил перевода, являющихся по существу правилами простой постфиксной СУ-схемы (возможно, с условиями), то можно выполнить весь перевод на ДМП-преобразователе, хранящем в своем магазине переводы разного типа, в том числе и не цепочки. Эти переводы легко „связать" с ячейками магазина, хранящими нетерминалы; это будет очевидная связь между нетерминалами в магазине и внутренними вершинами дерева разбора. Если некоторый перевод имеет целое значение, мы выходим за пределы описания ДМП-преобразователя, что, впрочем, не может составить проблемы при реализации транслятора на вычислительной машине.

Однако обобщение алгоритмов 9.2 и 9.3, работающих сверху вниз, не столь просто. В случае когда должны быть сформированы только цепочки, мы допускали возможность неявного развертывания формул, т. е. формулы могли содержать указатели на вершины, которые в конечном счете представляют нужную цепочку. Но может оказаться, что трактовать таким же образом арифметические формулы или формулы с условиями невозможно.

Что касается алгоритма 9.2, можно сделать следующую модификацию. Если развертывается нетерминал A, находящийся в верхушке магазина, то указатель остается в магазине непосредственно под A. Он указывает на вершину n, представляющую это вхождение A. После того как A развернут, указатель снова будет в верхушке магазина и будут вычислены переводы для всех потомков вершины n. (Мы покажем это индуктивно.) Затем можно вычислять перевод вершины n точно так, как мы делали бы, если бы анализ шел снизу вверх. Детали этого обобщения предлагаем в качестве упражнения.

Закончим раздел несколькими примерами более общих схем перевода.

Пример 9.13. Опишем детально генерацию кода для арифметических выражений, о которой шла речь в разд. 1.2.5, для машины с одним сумматором и произвольным доступом к памяти.

Будем предполагать, что язык ассемблера содержит команды

```
ADD   α
MPY   α
LOAD  α
STORE α
```

имеющие очевидный смысл.

Перевод будет основан на грамматике G_0. Каждый из нетерминалов E, T и F содержит по два элемента перевода. Первый вырабатывает цепочку выходных символов, представляющую последовательность команд, после выполнения которой значение соответствующего входного выражения будет размещено на сумматоре. Второй перевод будет целым числом, представляющим высоту вершины в дереве разбора. Однако при определении высоты мы не будем считать правила $E \to T$, $T \to F$ и $F \to (E)$. Поскольку измерение высоты необходимо только для определения имени ячейки, хранящей промежуточную информацию, эти правила можно не учитывать.

Перечислим шесть правил и соответствующие им элементы перевода:

(1) $E \to E + T$	$E_1 = T_1$ [1]; STORE \$' E_2 [1];' E_1 '; ADD \$' E_2
	$E_2 = \mathbf{max}(E_2, T_2) + 1$
(2) $E \to T$	$E_1 = T_1$ [1])
	$E_2 = T_2$
(3) $T \to T * F$	$T_1 = F_1$ 'STORE \$' T_2 ';' T_1 '; MPY \$' T_2
	$T_2 = \mathbf{max}(T_2, F_2) + 1$
(4) $T \to F$	$T_1 = F_1$
	$T_2 = F_2$
(5) $F \to (E)$	$F_1 = E_1$
	$F_2 = E$
(6) $F \to a$	F_1 = 'LOAD' NAME(a)
	$F_2 = 1$

Для определения цепочек в элементах перевода мы вновь принимаем соглашения языка Снобол. Цепочки, заключенные в кавычки, представляют сами себя. Символы без кавычек обозначают текущее значение символа, которое надо вместо него подставить. Таким образом, в правиле (1) элемент перевода

[1]) При обработке синтаксического дерева, а не дерева разбора, нет сверток в соответствии с правилами вывода $E \to T$, $T \to F$ и $F \to (E)$. Поэтому в реализации нет необходимости отражать тривиальные правила перевода, соответствующие этим правилам вывода.

для E_1 построен так, что значением E_1 должна быть конкатенация

(1) значения нетерминала T_1, за которым следует точка с запятой;

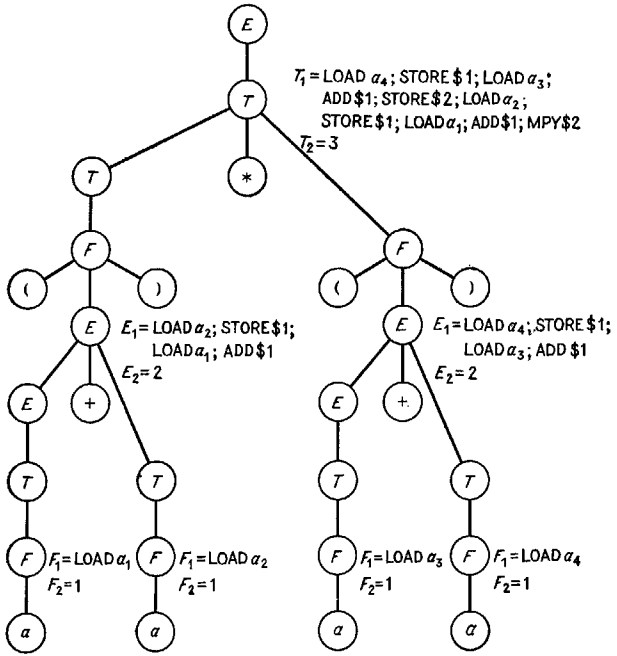


Рис. 9.17. Дерево разбора с переводами.

(2) команды STORE с адресом $\$n$, где n — высота вершины для E (в правой части правила вывода), за которой следует точка с запятой;

(3) значения нетерминала E_1 (левого аргумента операции $+$), за которым следует точка с запятой;

(4) команды ADD $\$n$, где n то же, что и в (2).

Здесь $\$n$ служит рабочей ячейкой, и перевод для E_1 (в правой части элемента перевода) эту ячейку использовать не будет из-за способа обработки E_2, T_2 и F_2.

В правиле (6) применяется NAME(a), где NAME — функция, участвующая в организации информации и дающая внутреннее (для компилятора) имя идентификатора, представленного лексемой a. Напомним, что терминальные символы всех наших грамматик — это лексемы. Если лексема представляет идентификатор, то она будет содержать указатель на позицию в таблице имен, в которой хранится информация об этом идентификаторе. Эта информация говорит как о том, какой именно идентификатор представляет данную лексему, так и о том, каковы атрибуты этого идентификатора.

На рис. 9.17 показано дерево разбора цепочки $(a+a)*(a+a)$ и даны значения некоторых символов перевода. Мы предполагаем, что NAME(a) для четырех идентификаторов, представленных символом a, дает значения a_1, a_2, a_3 и a_4. □

Пример 9.14. Код, вырабатываемый в примере 9.13, никоим образом не оптимален. Его можно значительно улучшить, если заметить, что когда правый операнд — просто идентификатор, нет необходимости читать его и запоминать в рабочей ячейке. Поэтому мы добавим третий перевод в правилах E, T и F, являющийся логической переменной и принимающий значение **истина** тогда и только тогда, когда выражением, соответствующим данной вершине, является просто идентификатор.

Первым переводом для E, T и F вновь будет код, вычисляющий выражение. Однако если выражение — идентификатор, то переводом будет только „NAME" этого идентификатора. Таким образом, схема перевода не „трудится" над одиночным идентификатором. Это приводит, впрочем, к некоторым неудобствам, поскольку можно считать, что грамматика для выражений составляет часть грамматики для оператора присваивания, а перевод для таких операторов присваивания, как $A \leftarrow B$, осуществляется на более высоком уровне.

Новые правила таковы:

(1) $E \rightarrow E+T$ $\quad E_1 =$ **if** T_3 **then**
if E_3 **then** 'LOAD' E_1 ';ADD' T_1
else E_1 ';ADD' T_1
else if E_3 **then** T_1 ';STORE $1; LOAD' E_1
'; ADD $1'
else T_1 ';STORE $' E_2 ';' E_1 '; ADD $' E_2
$E_2 = \max(E_2, T_2) + 1$
$E_3 =$ **ложь**

(2) $E \rightarrow T$ $\quad E_1 = T_1$
$E_2 = T_2$
$E_3 = T_3$

(3) $T \to T * F$ $\quad T_1 =$ **if** F_3 **then**
if T_3 **then** 'LOAD' T_1 ';MPY' F_1
else T_1 ';MPY' F_1
else if T_3 **then** F_1 ';STORE \$1;
LOAD' T_1 ';MPY \$1'
else F_1 ';STORE \$' T_2 ';' T_1 ';MPY \$' T_2
$T_2 =$ **max** $(T_2, F_2) + 1$
$T_3 =$ **ложь**

(4) $T \to F$ $\quad T_1 = F_1$
$T_2 = F_2$
$T_3 = F_3$

(5) $F \to (E)$ $\quad F_1 = E_1$
$F_2 = E_2$
$F_3 = E_3$

(6) $F \to a$ $\quad F_1 =$ NAME(a)
$F_2 = 1$
$F_3 =$ **истина**

В правиле (1) формула для E_1 проверяет, является ли один из аргументов или оба просто идентификаторами. Если правый аргумент — идентификатор, то генерируется код для вычисления левого аргумента и сложения правого аргумента с содержимым сумматора. Если левый аргумент — идентификатор, то генерируемый код будет именем идентификатора. В этом случае ему должно предшествовать 'LOAD'. Отметим, что тогда $E_2 = 1$, так что в качестве рабочей можно взять \$1, а не ячейку '\$' E_2 (которая в этом случае все равно была бы \$1).

Код, вырабатываемый для $(a + a) * (a + a)$, имеет вид

LOAD a_3; ADD a_4; STORE \$2; LOAD a_1; ADD a_2; MPY \$2

В этих правилах перевода не предполагается, что операции $+$ и $*$ коммутативны. □

Наш следующий пример снова связан с арифметическими выражениями. Он показывает, как с помощью ОСУ-схемы можно описать процесс, в основе которого лежит простая СУ-схема, но детерминированный МП-преобразователь, реализующий эту схему, способен хранить в ячейках своего магазина некоторую дополнительную информацию. В качестве промежуточного языка выбран трехадресный код.

Пример 9.15. Будем переводить $L(G_0)$ в последовательность трехадресных операторов вида $A \leftarrow + BC$ и $A \leftarrow * BC$, означающих, что переменной A надо присвоить сумму или соответственно

произведение B и C. В этом примере A будет цепочкой вида \$$i$, где i — целое. Основные переводы E_1, T_1 и F_1 будут последовательностями трехадресных операторов, вычисляющими значение выражения, соответствующее рассматриваемой вершине; E_2, T_2 и F_2 — целые числа, указывающие, как и в предыдущем примере, уровни; E_3, T_3 и F_3 — имя переменной, которой присваивается упомянутое выше значение выражения. Этим именем будет программная переменная, если выражение — просто идентификатор, и имя рабочей ячейки в противном случае. Схема перевода такова:

$E \to E + T$ $\quad E_1 = E_1 T_1$ '\$' **max** (E_2, T_2) '$\leftarrow +$' $E_3 T_3$ ';'
$E_2 =$ **max** $(E_2, T_2) + 1$
$E_3 =$ '\$' **max** (E_2, T_2)

$E \to T$ $\quad E_1 = T_1$
$E_2 = T_2$
$E_3 = T_3$

$T \to T * F$ $\quad T_1 = T_1 F_1$ '\$' **max** (T_2, F_2) '$\leftarrow *$' $T_3 F_3$ ';'
$T_2 =$ **max** $(T_2, F_2) + 1$
$T_3 =$ '\$' **max** (T_2, F_2)

$T \to F$ $\quad T_1 = F_1$
$T_2 = F_2$
$T_3 = F_3$

$F \to (E)$ $\quad F_1 = E_1$
$F_2 = E_2$
$F_3 = E_3$

$F \to a$ $\quad F_1 = e$
$F_2 = 1$
$F_3 =$ NAME(a)

Например, переводом для $a_1 * (a_2 + a_3)$ будет

\$1 $\leftarrow + a_2 a_3$;
\$2 $\leftarrow * a_1$ \$1;

Предоставляем читателю проследить, что правила перевода для E_1, T_1 и F_1 образуют простую постфиксную СУ-схему в предположении, что входными символами являются значения второго и третьего переводов для E, T и F. Практически перевод можно осуществить с помощью ДМП-преобразователя, хранящего в своем магазине значения второго и третьего переводов и выполняющего разбор грамматики G_0 методом LR(1). Таким образом, каждая ячейка магазина, содержащая E, будет также содержать значения E_2 и E_3 для соответствующих вершин дерева разбора (аналогично для ячеек, содержащих T и F).

Перевод осуществляется выдачей

$$\text{'\$'}\ \mathbf{max}\,(E_2, T_2)\ \text{'}\leftarrow\text{+'}\ E_3 T_3\ \text{';'}$$

при каждой свертке $E+T$ в E; здесь E_2, E_3, T_2 и T_3 — значения, приписанные ячейкам магазина, участвующим в свертке. При свертке $T \to T*F$ производятся аналогичные действия, а при остальных свертках не выдается ничего.

Следует отметить, что, поскольку второй и третий переводы для E, T и F могут принимать бесконечное число значений, устройство, осуществляющее перевод, не является, строго говоря, ДМП-преобразователем. Однако на практике это расширение легко реализуется на вычислительной машине с произвольным доступом. □

Пример 9.16. Будем генерировать код на языке ассемблера для условных операторов вида **if — then — else.** Предполагается, что нетерминал S представляет оператор и одно из правил вывода — это $S \to$ **if** B **then** S **else** S. Предполагается также, что S имеет перевод S_1, являющийся кодом для выполнения этого оператора. Если для S используется правило вывода, отличное от приведенного выше, будем считать, что перевод S_1 вычислен правильно.

Условимся также, что B представляет логическое выражение и переводы B_1 и B_2 для B вычисляются по другим правилам системы перевода. В частности, B_1 — код, вызывающий переход на ячейку B_2, если логическое выражение принимает значение **ложь.**

Для генерации требуемого кода необходимо, чтобы у нетерминала S был еще один перевод S_2, являющийся именем ячейки, на которую будет передано управление после того, как закончится выполнение оператора S. Будем предполагать также, что в вычислительной машине есть команда JUMP α, осуществляющая передачу управления на ячейку с именем α.

Для генерации имен этих ячеек применяется функция NEWLABEL, при обращении к которой генерируется имя для метки, никогда раньше не встречающееся. Например, компилятор может хранить глобальную переменную с целым значением i. Каждое обращение к NEWLABEL может приводить к тому, что i будет увеличиваться на 1, а в качестве значения будет выдаваться имя \$\$$i$. На самом деле функция NEWLABEL в указанном выше правиле вывода для S не вызывается, а вызывается в некоторых других правилах вывода для S и B.

Мы будем использовать также удобную псевдооперацию, подобную которой можно найти в большинстве языков ассемблера. Эта команда ассемблера имеет вид

EQUAL α, β

Она не генерирует никакого кода, а в результате ее выполнения ассемблер рассматривает α и β как имена одной и той же ячейки.

Команда EQUAL нужна потому, что каждое из вхождений S в правой части правила вывода $S \to$ **if** B **then** S **else** S содержит имя команды, которая должна быть выполнена следующей. Мы должны быть уверены, что ячейка, участвующая в одном из них, та же, что и в другом.

Элементы перевода для этого правила вывода таковы:

$$S \to \textbf{if}\ B\ \textbf{then}\ S^{(1)}\ \textbf{else}\ S^{(2)} \quad \begin{aligned} S_1 = {} & \text{'EQUAL'}\ S_2^{(1)}\ \text{','}\ S_2^{(2)}\ \text{';'} \\ & B_1\ \text{';'}\ S_1^{(1)}\ \text{'; JUMP'} \\ & S_2^{(1)}\ \text{';'}\ B_2\ \text{':'}\ S_1^{(2)} \\ S_2 = {} & S_2^{(1)} \end{aligned}$$

Таким образом, перевод S состоит из конкатенации

(1) команды, означающей, что $S_2^{(1)}$ и $S_2^{(2)}$ представляют одну и ту же ячейку;

(2) объектного кода для логического выражения (B_1), вызывающего переход на ячейку B_2, если его значением является **ложь;**

(3) объектного кода для первого оператора ($S_1^{(1)}$), за которым следует переход на ячейку, помеченную $S_2^{(1)}$ (ячейка с этой меткой находится вне компилируемого оператора);

(4) объектного кода для второго оператора ($S_1^{(2)}$). Первая ячейка этого кода помечена B_2.

Перевод S_2 для данного оператора тот же, что и перевод S_2 для первого подоператора. Таким образом, независимо от того, истинно или ложно B, управление на ячейку $S_2^{(1)}$ (совпадающую теперь с $S_2^{(2)}$) будет передано.

Рассмотрим сложный оператор

$$\textbf{if}\ B^{(1)}\ \textbf{then}\ \textbf{if}\ B^{(2)}\ \textbf{then}\ S^{(1)}\ \textbf{else}\ S^{(2)}\ \textbf{else}\ S^{(3)}$$

возникающий в результате двух применений рассматриваемого правила вывода (строго говоря, верхних индексов здесь не должно быть; они приведены для удобства ссылок). Объектным кодом, генерируемым для этого оператора, будет

$$\begin{array}{ll} & \text{EQUAL}\ S_2^{(1)},\ S_2^{(3)} \\ & \text{код для}\ B^{(1)} \\ & \text{EQUAL}\ S_2^{(1)},\ S_2^{(2)} \\ & \text{код для}\ B^{(2)} \\ & \text{код для}\ S^{(1)} \\ & \text{JUMP}\ S_2^{(1)} \\ B_2^{(2)}: & \text{код для}\ S^{(2)} \\ & \text{JUMP}\ S_2^{(1)} \\ B_2^{(1)}: & \text{код для}\ S^{(3)} \end{array}$$

(Здесь точки с запятой заменены переходами на новую строку.) □

Пример 9.17. В качестве последнего примера этого раздела рассмотрим генерацию кода для вызовов вида

$$\textbf{call}\ X(E^{(1)}, \ldots, E^{(n)})$$

Предположим, что в языке ассемблера есть команда CALL X, которая передает управление на ячейку X и запоминает в специальном регистре адрес текущей ячейки для последующего возврата из обращения к функции. Будем переводить **call** $X(E^{(1)}, \ldots, E^{(n)})$ в код, который вычисляет значения каждого из выражений $E^{(1)}, \ldots, E^{(n)}$ и запоминает их в рабочих ячейках $t_1, \ldots, t_n$. За этими вычислениями идет команда ассемблера CALL X и n команд ARG $t_1, \ldots,$ ARG t_n, осуществляющих „хранение значений" и использующихся как указатели на аргументы этого вызова X.

Основные правила вывода, описывающие оператор вызова, таковы:

$$S \to \textbf{call}\ aA$$
$$A \to e \mid (L)$$
$$L \to E,\ L \mid E$$

Таким образом, оператором является ключевое слово **call**, за которым следует идентификатор и список аргументов (A). Списком аргументов может быть либо пустая строка, либо список выражений, заключенный в скобки (L). Предполагается, что каждое выражение E имеет два перевода E_1 и E_2, причем E_1 — объектный код, помещающий в сумматор значение E, а E_2 — имя рабочей ячейки для запоминания значения E. Имена для рабочих ячеек порождаются функцией NEWLABEL. Требуемый перевод осуществляется следующими правилами:

$S \to \textbf{call}\ aA$	$S_1 = A_1$ ';CALL' NAME(a) A_2
$A \to (L)$	$A_1 = L_1$
	$A_2 =$ ';' L_2
$A \to e$	$A_1 = e$
	$A_2 = e$
$L \to E,\ L$	$L_1 = E_1$ ';STORE' E_2 ';' L_1
	$L_2 =$ 'ARG' E_2 ';' L_2
$L \to E$	$L_1 = E_1$ ';STORE' E_2
	$L_2 =$ 'ARG' E_2

Например, оператор **call** AB $(E^{(1)}, E^{(2)})$ переводится (в предположении, что рабочими ячейками для $E^{(1)}$ и $E^{(2)}$ являются

\$\$1 и \$\$2 соответственно) в

```
код для E(1)
STORE $$1
код для E(2)
STORE $$2
CALL AB
ARG $$1
ARG $$2
```

Для реализации вызова можно было бы запоминать значения выражений $E^{(1)}, \ldots, E^{(n)}$ непосредственно в ячейках, следующих за командой CALL(X). Построение схемы перевода, генерирующей такой код, оставляем читателю в качестве упражнения. □

9.3.3. Наследственный и синтезированный переводы

Существует другое обобщение синтаксически управляемого перевода, которое может оказаться полезным в некоторых приложениях. До сих пор мы рассматривали переводы нетерминалов как функции только переводов прямых потомков вершины. Однако можно допустить, чтобы значение перевода было функцией значений переводов не только прямых потомков вершины, но и ее предков. Дадим следующее определение.

Определение. Перевод нетерминала в схеме перевода будем называть *синтезированным*, если он зависит только от переводов вершины, в которой вычисляется, и переводов ее прямых потомков. Перевод будем называть *наследственным*, если он зависит только от переводов вершины, в которой вычисляется, и переводов ее прямого предка. Все переводы, рассмотренные до сих пор, были синтезированными и, более того, не содержали формул, включающих переводы в самой вершине.

Элементы перевода, связанные с правилом вывода, определим обычным образом, если определяемый перевод — синтезированный. Однако наследственные переводы, связанные с правилом вывода, могут использовать в качестве аргументов переводы левой части правила вывода, вычисляя при этом перевод для некоторого конкретного символа в правой части. Значит, необходимо отличать нетерминал слева от любого его вхождения справа. Как обычно, для этой цели мы будем употреблять верхние индексы.

Пример 9.18. Рассмотрим правила перевода

$$A^{(1)} \to A^{(2)} A^{(3)} \qquad A_1^{(1)} = aA_2^{(2)} A_2^{(3)} b$$
$$A_2^{(2)} = A_1^{(1)}$$
$$A_2^{(3)} = aA_1^{(1)}$$

Здесь A имеет два перевода: синтезированный перевод A_1 и наследственный перевод A_2. Правило для A_1 относится к знакомому нам типу. Правила для $A_2^{(2)}$ и $A_2^{(3)}$ служат примерами правил перевода для наследственных атрибутов. Исследуем часть дерева на рис. 9.18. Правило для $A_2^{(2)}$ говорит о том, что перевод A_2 в вершине n_2 должен совпадать с переводом A_1 в n_1. Поскольку A_1 в n_1 выражается через A_2 в n_2, это пример зацикленного определения, и в такой форме это правило недопустимо. □

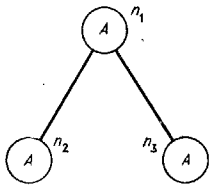


Рис. 9.18. Часть дерева.

Схема перевода, содержащая как наследственный, так и синтезированный переводы, реализуется не просто. В самом общем случае сначала надо полностью построить дерево разбора, а затем вычислить в каждой вершине все переводы, которые можно вычислить в терминах уже вычисленных переводов в потомках и предках. Прежде всего надо начать с синтезированных атрибутов в вершинах высоты 1. При перевычислении перевода все переводы, от него зависящие, как наследственные, так и синтезированные, также должны вычисляться снова. Тем самым может оказаться, что процесс перевычисления переводов никогда не закончится. Такую схему перевода называют *зацикленной*. Проблема распознавания, является ли схема перевода зацикленной, разрешима, хотя ее решение сложно, и мы предлагаем ее в качестве упражнения. Приведем пример, в котором для любого дерева разбора существует естественный порядок вычисления всех переводов.

Пример 9.19. Скомпилируем для вычисления арифметических выражений код, который должен выполняться на машине с двумя быстрыми регистрами, обозначенными через A и B. Будут использоваться команды

```
LOADA  α
LOADB  α
STOREA α
STOREB α
ADDA   α
ADDB   α
MPY    α
ATOB
```

Смысл первых шести команд очевиден: можно считывать, запоминать и складывать в любом регистре. Предполагается, что

команда MPY берет левый операнд из регистра B и заносит результат в регистр A. Последняя команда, ATOB, передает содержимое регистра A в регистр B.

Как и в примере 9.13, построим перевод на основе грамматики G_0. Переводы E_1, T_1 и F_1 будут представлять код для вычисления значения соответствующего входного выражения, заносящий результат либо в регистр A, либо в регистр B. Однако перевод E_1 для корня дерева разбора всегда будет помещать значение выражения в регистр A. Как и в примере 9.13, E_2, T_2 и F_2 будут целыми числами, равными высотам вершин. Переводы E_3, T_3 и F_3 будут логическими выражениями, принимающими значение **истина** тогда и только тогда, когда значение выражения должно остаться в регистре A. Эти переводы — наследственные, а первые шесть — синтезированные. Методы улучшения кода, данные в примере 9.14, здесь не применены. Правила перевода таковы:

$E^{(1)} \to E^{(2)} + T$ $\quad$ $E_1^{(1)} =$ **if** $E_3^{(1)}$ **then**
$\quad T_1$ ';STOREA \$' $E_2^{(2)}$ ';' $E_1^{(2)}$
$\quad$ ';ADDA \$' $E_2^{(2)}$
$\quad$ **else** T_1 ';STOREA \$' $E_2^{(2)}$ ';' $E_1^{(2)}$
$\quad$ ';ADDB \$' $E_2^{(2)}$
$E_2^{(1)} =$ **max** $(E_2^{(2)}, T_2) + 1$
$E_3^{(2)} = E_3^{(1)}$ [1])
$T_3 =$ **истина**

$E \to T$ $\quad$ $E_1 = T_1$
$E_2 = T_2$
$T_3 = E_3$

$T^{(1)} \to T^{(2)} * F$ $\quad$ $T_1^{(1)} =$ **if** $T_3^{(1)}$ **then**
$\quad F_1$ ';STOREA \$' $T_2^{(2)}$ ';' $T_1^{(2)}$
$\quad$ ';MPY \$' $T_2^{(2)}$
$\quad$ **else** F_1 ';STOREA \$' $T_2^{(2)}$ ';' $T_1^{(2)}$
$\quad$ ';MPY \$' $T_2^{(2)}$ ';ATOB'
$T_2^{(1)} =$ **max** $(T_2^{(2)}, F_2) + 1$
$T_3^{(2)} =$ **ложь**
$F_3 =$ **истина**

$T \to F$ $\quad$ $T_1 = F_1$
$T_2 = F_2$
$F_3 = T_3$

[1]) Предполагается, что вначале все логические переводы имеют значение **истина**. Таким образом, если $E_3^{(1)}$ ссылается на корень, считается, что перевод для него уже определен.

$F \to (E)$	$F_1 = E_1$
	$F_2 = E_2$
	$E_3 = F_3$
$F \to a$	$F_1 =$ if F_3 then 'LOADA' NAME(a) else 'LOADB' NAME(a)
	$F_2 = 1$

Стратегия заключается в том, чтобы вычислять все правые операнды в регистре A. Левый операнд для * всегда вычисляется в B, левый операнд для + вычисляется либо в A, либо в B в зависимости от того, где желательно его разместить. Таким образом, элемент перевода для $E_1^{(1)}$, связанный с правилом вывода $E \to E + T$, дает перевод, который вычисляет T в регистре A, запоминает его в рабочей ячейке, затем вычисляет E либо в A, либо в B в зависимости от того, где желательно, и выполняет сложение. Правило перевода для $T_1^{(1)}$, связанное с правилом вывода $T \to T * F$, вычисляет F в регистре A, запоминает его, вычисляет T в регистре B, умножает и, если надо, передает результат в регистр B.

Дерево разбора для $(a+a)*a$ показано на рис. 9.19, на котором некоторым внутренним вершинам присвоены имена. Наследственные и синтезированные атрибуты переносятся без изменений вверх и вниз по цепям $E \to T \to F$. В табл. 9.5 приведена последовательность вычисления переводов (перенос переводов из E в T, из T в F и обратно в таблице не указан). □

Таблица 9.5

Перевод	В вершине	Значение
T_3	n_5	истина
E_3	n_3	ложь
F_3	n_1	ложь
F_3	n_2	истина
F_1	n_2	LOADA a_2
F_2	n_2	1
F_1	n_1	LOADB a_1
F_2	n_1	1
E_1	n_3	LOADA a_2 ; STOREA \$1 ; LOADB a_1 ; ADDB \$1
E_2	n_3	2
F_3	n_4	истина
F_1	n_4	LOADA a_3
F_2	n_4	1
T_1	n_5	LOADA a_3 ; STOREA \$2 ; LOADA a_2 ; STOREA \$1 ; LOADB a_1 ; ADDB \$1 ; MPY\$2
T_2	n_5	3

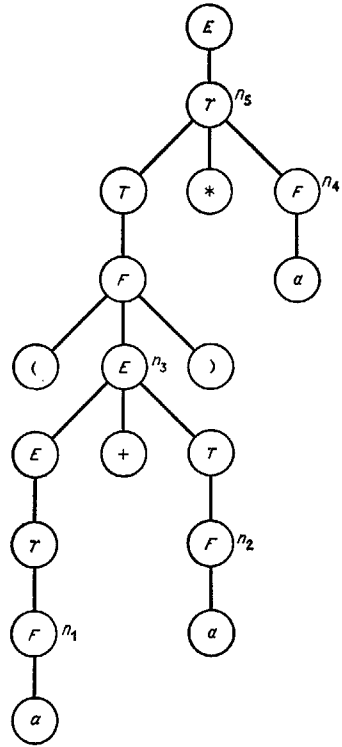


Рис. 9.19. Дерево разбора.

9.3.4. Замечание о разбиении на фазы

Мы обрисовали компилятор так, как будто каждый из его шагов — лексический анализ, синтаксический анализ и генерация кода — выполнялся отдельно в указанном порядке. Однако такое деление на последовательные шаги служит скорее методическим целям представления и на практике может и не соблюдаться.

Во-первых, фаза лексического анализа вырабатывает лексемы постольку, поскольку они необходимы для разбора. Реально же

входной цепочки лексем, которая была показана для демонстрации действий при разборе, может и не быть. Лексемы разыскиваются только по мере того, как они требуются при разборе. Конечно, эта разница никоим образом не влияет на процесс разбора.

Как мы уже отмечали выше, разбор и генерация кода могут осуществляться одновременно. Таким образом, три фазы могут работать комбинированно. Обращение к лексическому анализатору происходит тогда, когда процесс разбора не может идти дальше. После каждой свертки (при разборе снизу вверх) или развертки нетерминала (при разборе сверху вниз) фаза генерации кода обрабатывает вершину или вершины дерева разбора, сформированные только что.

Если бы вырабатываемым переводом была одна цепочка, то было бы не столь существенно, когда именно вырабатываются различные ее части. Однако напомним, что отдельные части вырабатываемого перевода могут быть различных типов: объектным кодом, диагностикой, командами, выполняемыми самим компилятором (такими, например, как команды по вводу информации в механизмы ее хранения).

Если перевод осуществляется МП-преобразователем или аналогичным устройством с одним выходным потоком, то проблем с разбиением перевода на фазы не возникает. Символы считаются выходными по мере того, как они вырабатываются устройством. Если предположить, что различные виды выходных потоков отделяются друг от друга метасимволами, то выходной поток можно разделять по мере его образования. Например, промежуточный код передается на фазу оптимизации, диагностика размещается в собственный список, чтобы затем быть напечатанной, а команды по организации информации выполняются немедленно.

Предположим, что для обобщенного синтаксически управляемого перевода применяется один из наиболее общих вариантов алгоритмов 9.1—9.4. Можно было бы дождаться, когда будет полностью сконструировано дерево или ориентированный ациклический граф, а затем сформировать единый выходной поток. Деление потока на объектный код и команды могло бы осуществляться точно так же, как и для МП-преобразователя. Это означает, что команды по организации информации не выполнялись бы до тех пор, пока не был бы полностью сформирован выходной поток, и что команды выполняются по мере обработки выходного потока. Такая организация требует дополнительного прохода и большой памяти с произвольным доступом, но может оказаться наиболее приемлемой, если требуется мощность самых общих схем синтаксически управляемого перевода.

С другой стороны, можно принять, что команды по организации информации и другие команды компилятора связаны с отдельными вершинами дерева разбора и выполняются сразу, как только вершина образована. Однако, если алгоритм разбора требует возвратов, надо быть осторожным, чтобы не выполнить команды, связанной с вершиной, не являющейся частью дерева разбора. В этой ситуации необходим механизм для отмены результата такой команды.

УПРАЖНЕНИЯ

9.3.1. Найдите ОСУ-схемы для следующих переводов:

(а) $\{(a^n,\ a^{n^3}) \mid n \geqslant 1\}$,

(б) $\{(a^n,\ a^{2^n}) \mid n \geqslant 1\}$,

(в) $\{(w,\ ww) \mid w \in (0+1)^*\}$.

9.3.2. Покажите, что существуют переводы, определяемые ОСУ-схемой, но не определяемые никакой СУ-схемой.

****9.3.3.** Пусть T — семантически однозначная ОСУ-схема с бесконечной областью определения, входная КС-грамматика которой является приведенной. Покажите, что в этом случае должно удовлетворяться одно из следующих условий:

(1) Существуют константы c_1 и c_2, большие 1 и такие, что если $(x,\ y) \in \tau(T)$, то $|y| \leqslant c_2^{|x|}$ и для бесконечного числа цепочек x найдутся такие пары $(x,\ y) \in \tau(T)$, что $|y| \geqslant c_1^{|x|}$.

(2) Существуют константы c_1 и c_2, большие 0, и целое число $i \geqslant 1$, такие, что если $(x,\ y) \in \tau(T)$ и $x \neq e$, то $|y| \leqslant c_2 |x|^i$ и для бесконечного числа цепочек x найдутся такие пары $(x,\ y) \in \tau(T)$, что $|y| \geqslant c_1 |x|^i$.

(3) Множество значений схемы T конечно.

9.3.4. Покажите, что перевод $\{(a^n,\ a^m) \mid m$ — целая часть числа $\sqrt{n}\}$ нельзя определить никакой ОСУ-схемой.

9.3.5. Для переводов из упр. 9.3.1 (а)—(в) постройте ориентированные ациклические графы, вырабатываемые алгоритмом 9.4 для входов a^3, a^4 и 011 соответственно.

9.3.6. Укажите последовательность вершин, по которой проходит алгоритм 9.5, примененный к ориентированным ациклическим графам из упр. 9.3.5.

***9.3.7.** Дополните пример 9.16, включив правило вывода $S \to$ **while** B **do** S, смысл которого должен заключаться в том, что проверяется выражение B, а затем выполняется оператор S до тех пор, пока значением B не станет **ложь**.

9.3.8. Следующая грамматика порождает объявления, подобные объявлениям в ПЛ/I:

$$D \to (L)\,M$$
$$L \to a,\ L \mid D,\ L \mid a \mid D$$
$$M \to m_1 \mid m_2 \mid \ldots \mid m_k$$

Терминал a изображает идентификатор; $m_1, \ldots, m_k$ — это k возможных атрибутов идентификатора в языке. Терминальными символами являются также запятая и скобки. L представляет список идентификаторов и объявлений; D — это объявление, состоящее из заключенного в скобки списка идентификаторов и объявлений. Атрибут, выведенный из M, должен применяться ко всем идентификаторам, выведенным из L в правиле вывода $D \to (L)\,M$, даже если идентификатор находится внутри сложного объявления. Например, объявление $(a_1, (a_2, a_3)\,m_1)\,m_2$ назначает идентификаторам a_2 и a_3 атрибут m_1, а идентификаторам a_1, a_2 и a_3 атрибут m_2. Используя любые типы перевода, постройте схему перевода, переводящую цепочки, выведенные из нетерминала D, в k списков, причем список i должен содержать в точности те идентификаторы, которым придан атрибут m_i.

9.3.9. Измените пример 9.17 так, чтобы в команду ARG заносились не указатели на значения выражений, а сами эти значения.

9.3.10. Рассмотрим схему перевода

$N \to L^{(1)} \cdot L^{(2)}$	$N_1 = L_1^{(1)} + L_1^{(2)}/2^{L_2^{(2)}}$
$N \to L$	$N_1 = L_1$
$L \to LB$	$L_1 = 2L_1 + B_1$
	$L_2 = L_2 + 1$
$L \to B$	$L_1 = B_1$
	$L_2 = 1$
$B \to 0$	$B_1 = 0$
$B \to 1$	$B_1 = 1$

Во входной грамматике из нетерминала N выводятся двоичные числа (возможно, с двоичной точкой). Нетерминал L представляет список битов, а нетерминал B — бит. Элементами перевода являются арифметические формулы. Элемент перевода N_1 представляет рациональное число — значение двоичного числа, выведенного из N. Элементы перевода L_1, L_2 и B_1 принимают целые значения. Например, 11.01 имеет перевод $3^1/_4$. Покажите, что $\tau(T) = \{(b, d) \mid b$ — двоичное целое, d — его значение$\}$.

***9.3.11.** Рассмотрим следующую схему перевода с входной грамматикой, как в упр. 9.3.10, но содержащую как синтезированные, так и наследственные атрибуты:

$N \to L^{(1)} \cdot L^{(2)}$	$N_1 = L_1^{(1)} + L_1^{(2)}$
	$L_2^{(1)} = 0$
	$L_2^{(2)} = -L_3^{(2)}$
$N \to L$	$N_1 = L_1$
	$L_2 = 0$
$L^{(1)} \to L^{(2)} B$	$L_1^{(1)} = L_1^{(2)} + B_1$
	$B_2 = L_2^{(1)}$
	$L_2^{(2)} = L_2^{(1)} + 1$
	$L_3^{(1)} = L_3^{(2)} + 1$
$L \to B$	$L_1 = B_1$
	$B_2 = L_2$
	$L_3 = 1$
$B \to 0$	$B_1 = 0$
$B \to 1$	$B_1 = 2^{B_2}$

Дерево разбора для 11.01 со значениями элементов перевода, соответствующих каждой вершине, изображено на рис. 9.20.

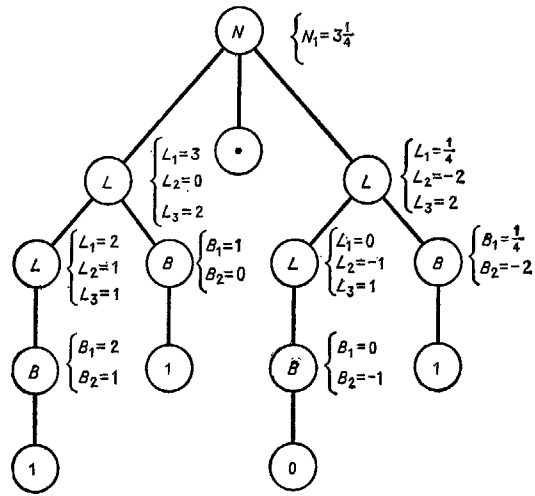


Рис. 9.20. Дерево разбора с переводами.

Отметим, что для того, чтобы вычислить элемент перевода N_1, надо сначала вычислить снизу вверх все L_3 справа от разделительной точки, затем сверху вниз все L_2 и, наконец, снизу вверх все L_1. Покажите, что эта схема определяет тот же перевод, что и схема из упр. 9.3.10.

***9.3.12.** Покажите, что всякий перевод, который можно выполнить с помощью наследственного и синтезированного переводов, можно выполнить с помощью только синтезированного перевода. *Указание:* На структуру перевода ограничений нет. Таким образом, перевод, определяемый в некоторой вершине, может быть целым поддеревом.

9.3.13. Можно ли любой перевод, использующий синтезированные переводы, выполнить только с помощью наследственного перевода?

****9.3.14.** Приведите алгоритм для проверки, является ли данная схема перевода, содержащая наследственные и синтезированные атрибуты, зацикленной.

9.3.15. Видоизмените пример 9.18 так, чтобы ввести возможности улучшения кода из примера 9.14.

***9.3.16.** Алгоритм дифференцирования примера 9.11 позволяет генерировать такие выражения, как $1*\cos(x)$ или $0*x+(1)*1$ (формальная производная от $1*x$). Можно выявить и исключить выражения, *явно* равные 0 или 1, где *явно* определяется следующим образом:

(1) 0 явно равен 0; 1 явно равна 1.
(2) Если E_1 явно равно 0, а E_2—некоторое выражение, то E_1*E_2 и E_2*E_1 явно равны 0.
(3) Если E_1 и E_2 явно равны 1, то E_1*E_2 явно равно 1.
(4) Если E_1 явно равно 0, а E_2 явно равно 1, то E_1+E_2 и E_2+E_1 явно равны 1.
(5) Если E_1 и E_2 явно равны 0, то E_1+E_2 явно равно 0.

Измените ОСУ-схему (в том числе, возможно, и входную грамматику) так, чтобы в переводе не появлялись выражения, явно равные 0, а в качестве множителей чтобы не появлялись выражения, явно равные 1.

9.3.17. Рассмотрим следующую грамматику для оператора присваивания, включающего переменные с индексом:

$$
\begin{aligned}
A &\to I := E \\
E &\to E\ \langle\text{знсл}\rangle\ T \mid T \\
T &\to T\ \langle\text{знумн}\rangle\ F \mid F \\
F &\to (E) \mid I \\
I &\to a \mid a\,(L) \\
L &\to a,\ L \mid a \\
\langle\text{знсл}\rangle &\to + \mid - \\
\langle\text{знумн}\rangle &\to * \mid /
\end{aligned}
$$

Примером оператора, генерируемого этой грамматикой, может служить

$$a\,(a,\ a) := a\,(a+a,\ a*(a+a))+a$$

Здесь a—лексема, представляющая идентификатор. Сконструируйте схему перевода, основанную на этой грамматике и генерирующую необходимые команды языка ассемблера или многоадресного кода для операторов присваивания.

9.3.18. Покажите, что ОСУ-схема из примера 9.11 правильно вырабатывает выражение для производной входного выражения.

****9.3.19.** Покажите, что перевод

$$\{(a_1\ldots a_n b_1\ldots b_n,\ a_1 b_1\ldots a_n b_n) \mid n \geqslant 1,\ a_i \in \{0, 1\},\ b_i \in \{2, 3\} \text{ для } 1 \leqslant i \leqslant n\}$$

не определяется никакой ОСУ-схемой.

Проблема для исследования

9.3.20. Перевод арифметических выражений становится довольно сложным, если допустить в качестве операндов элементы многих различных типов данных. Например, значениями могут быть логические переменные, цепочки, целые, вещественные или комплексные числа—в последних трех случаях они могут иметь одинарную, двойную или, возможно, более высокую точность. Кроме того, некоторые значения могут лежать в динамически распределяемой памяти, а некоторые размещаться статически. Число комбинаций вполне может быть настолько большим, что элементы перевода, связанные с таким правилом вывода, как $E \to E+T$, станут весьма громоздкими. Можете ли Вы по данному переводу, перечисляющему желаемые коды для каждого случая, придумать способ автоматического упрощения записи? Скажем, в примере 9.19 части перевода E_1^{U} для **then** и **else** отличаются только одним символом A или B в конце команды ADD. Таким образом, если использовать более изощренные механизмы, описание можно сократить почти вдвое.

Упражнение на программирование

***9.3.21.** Сконструируйте схему перевода, отображающую подмножество Фортрана в промежуточный язык, как в упр. 9.1.10. Напишите программу, реализующую эту схему перевода. Реализуйте генератор кодов, построенный в упр. 9.1.11. Скомбинируйте эти программы с лексическим анализатором, чтобы получить компилятор с этого подмножества Фортрана. Сконструируйте тестовые цепочки, на которых можно было бы проверить правильность работы каждой программы.

Замечания по литературе

Обобщенные схемы синтаксически управляемого перевода описаны в работе Ахо и Ульмана [1971]; там же можно найти решение упр. 9.3.3. В работе Кнута [1968 б] определяются схемы перевода с наследственными и синтезированными атрибутами; там же обсуждаются ОСУ-схемы из упр. 9.3.10 и 9.3.11.

Решение упр. 9.3.14 дали Бруно и Беркхард [1970] и Кнут [1968 б]; упр. 9.3.12 взято из работы Кнута [1968 б].

10 ОРГАНИЗАЦИЯ ИНФОРМАЦИИ

В этой главе рассматриваются методы, с помощью которых информацию можно быстро запомнить и найти в таблицах. Наиболее важное применение этих методов — хранение информации о лексемах, полученных в процессе лексического анализа, и нахождение ее при генерации кода. Мы разберем два подхода: простые методы нахождения информации и формализм грамматик свойств. Последний заключается в том, что атрибуты и идентификаторы связываются между собой по мере того, как появляется уверенность, что для каждого идентификатора доступна вся необходимая для целей перевода информация.

10.1. ТАБЛИЦЫ ИМЕН

Таблицы, хранящие имена и информацию, связанную с этими именами, называются *таблицами имен*. Таблицы имен используются практически во всех компиляторах. Таблица имен изо-

Имя	Данные
I	целое
LOOP	метка
X	вещественный массив

Рис. 10.1. Таблица имен.

бражена на рис. 10.1. Элементами полей имен являются обычно идентификаторы. Если имена могут быть различной длины, то удобнее, чтобы элементы полей имен были указателями на область памяти, фактически хранящую эти имена.

Элементы полей данных, называемые иногда *дескрипторами*, дают информацию, которую надо собрать о каждом имени. В некоторых ситуациях с данным именем можно связать дюжину или более элементов информации. Например, возможно, что надо знать тип данных (вещественный, целый, строковый и т. д.) идентификатора; был ли он меткой, именем процедуры или формальным параметром процедуры; должен ли он распределяться в статической или динамической области памяти; был ли он идентификатором, имеющим структуру (например, массивом), и если да, то какую структуру (например, размерность массива). Если число элементов информации, связанной с данным именем, переменно, то снова удобно хранить в поле данных указатель на эту информацию.

10.1.1. Хранение информации о лексемах

Компилятор использует таблицу имен для хранения информации о лексемах, в частности об идентификаторах. Эта информация используется затем для двух целей. Во-первых, для проверки семантической правильности исходной программы. Например, если в исходной программе есть оператор вида

GOTO LOOP

то компилятор должен проверить, что идентификатор LOOP встречается в программе в качестве метки соответствующего оператора. Такую информацию можно найти в таблице имен (хотя и не обязательно во время обработки оператора перехода). Во-вторых, информация в таблице имен используется при генерации кода. Например, если в исходной программе на Фортране есть оператор вида

$$A=B+C$$

то генерированный для операции + код зависит от атрибутов идентификаторов B и C (скажем, имеют ли они фиксированную или плавающую точку, находятся ли внутри или вне области COMMON и т. д.).

Лексический анализатор заносит имена и информацию в таблицу имен. Например, всякий раз, когда лексический анализатор обнаруживает идентификатор, он проверяет в таблице имен, встречалась ли ранее такая же лексема. Если нет, то лексический анализатор заносит в таблицу имя этого идентификатора вместе с соответствующей информацией. Если идентификатор уже есть в некоторой ячейке l таблицы имен, то лексический анализатор вырабатывает на выходе лексему (⟨идентификатор⟩, l).

Таким образом, лексический анализатор обращается в таблицу имен каждый раз, когда он находит лексему, и чтобы сконструировать эффективный компилятор, надо уметь по данному вхождению идентификатора быстро определить, отведена ли для этого идентификатора ячейка в таблице имен. Если этого элемента нет, то надо иметь возможность быстро вставить идентификатор в таблицу.

Пример 10.1. Предположим, что при компиляции языка типа Фортран для всех имен переменных используется единственная лексема типа ⟨идентификатор⟩. Когда (прямой) лексический анализатор впервые обнаруживает идентификатор, он может занести в таблицу имен информацию о том, имеет ли соответствующая переменная фиксированную или плавающую точку. Лексический анализатор получает эту информацию по первой букве идентификатора. Конечно, если в силу предыдущего объявления идентификатор является функцией либо подпрограммой или не подчиняется обычному соглашению о фиксированной или плавающей точке, то информация для него уже содержится в таблице имен, и это приводит к тому, что лексический анализатор не заносит информацию о нем в таблицу. □

Пример 10.2. Предположим, что разрабатывается компилятор с языка, в котором объявления массивов определяются правилами

⟨оператор массива⟩ → **массив** ⟨список массивов⟩
⟨список массивов⟩ → ⟨определение массива⟩, ⟨список массивов⟩ | ⟨определение массива⟩
⟨определение массива⟩ → ⟨идентификатор⟩ (⟨целое⟩)

Примером объявления массива в этом языке служит

(10.1.1) **массив** $AB(10)$, $CD(20)$

Для простоты мы здесь предполагаем, что массивы одномерны. Дерево разбора для оператора (10.1.1) изображено на рис. 10.2, на котором идентификаторами являются AB и CD, а целыми — 10 и 20.

На этом дереве разбора лексемы ⟨идентификатор⟩$_1$, ⟨идентификатор⟩$_2$, ⟨целое⟩$_1$ и ⟨целое⟩$_2$ представляют соответственно AB, CD, 10 и 20.

Естественно, оператор объявления массивов не выполняется; он не компилируется в машинный код. Однако имеет смысл рассматривать его перевод как последовательность шагов по организации информации, которые должны выполняться непосредственно компилятором. Таким образом, если переводом идентификатора является указатель на отведенное для него место в таблице имен, а перевод целого — оно само, то синтаксически управляемым переводом вершины, помеченной ⟨определение

массива>, могут быть команды для механизмов организации информации, предназначенные для того, чтобы запомнить, что идентификатор — это массив, а его размер — это соответствующее целое. □

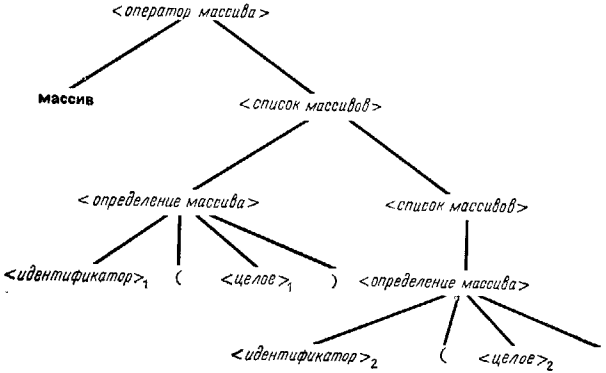


Рис. 10.2. Дерево разбора оператора массива.

Способы применения информации, запомненной в таблице имен, многочисленны. Простой пример: для каждого подвыражения в арифметическом выражении может требоваться информация о его виде, чтобы знать, над какими числами выполнять арифметические операции — над числами с фиксированной или плавающей точкой, комплексными числами и т. д. Эта информация выбирается из таблицы имен для листьев, помеченных как константы или идентификаторы, и передается вверх по дереву в соответствии с такими, например, соглашениями, что число с фиксированной точкой + число с плавающей точкой = число с плавающей точкой, а число с плавающей точкой + комплексное число = комплексное число. С другой стороны, язык, а следовательно и компилятор, может запрещать выражения со смешанными видами (как это делается, например, в некоторых версиях Фортрана).

10.1.2. Механизмы запоминания

Из сказанного можно заключить, что компилятору нужны методы организации информации, способные быстро запоминать и выдавать информацию о большом числе различных объектов

(например, идентификаторах). Кроме того, в то время как число объектов, реально появляющихся в программе, может быть и велико (скажем, для типичной программы порядка 100 или 1000), число всех возможных объектов имеет порядок несравненно более высокий; большинство из возможных идентификаторов в данной программе не появляется.

Изложим кратко возможные способы запоминания информации в таблицах, чтобы обосновать использование таблиц расстановки, или распределенной памяти, которые будут изучаться в следующем разделе.

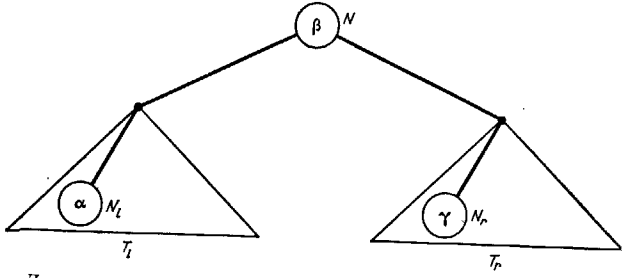
Основную проблему можно сформулировать так. Есть большой набор объектов, которые могут встретиться. Объект здесь может рассматриваться как имя, состоящее из цепочки символов. Объекты встречаются нам в непредсказуемом порядке, и точное число объектов, которые могут встретиться, заранее неизвестно. Как только объект встретился, надо проверить в таблице, появлялся ли он раньше, и если нет, внести его имя в таблицу. Помимо этого, будет еще и другая информация об объекте, которую нам надо будет запомнить в таблице.

Сначала разберем использование для запоминания информации об объектах таблиц с прямым доступом. В такой таблице для каждого возможного объекта резервируется отдельная ячейка. В ней запоминается информация относительно этого объекта, и нет необходимости запоминать в таблице его имя. Если число возможных объектов мало и для каждого объекта действительно можно выделить отдельную ячейку, такая таблица — таблица с прямым доступом — дает очень быстрый механизм запоминания и извлечения информации об объектах. Однако в большинстве случаев идею применения таблицы с прямым доступом приходится отвергнуть, поскольку размер таблицы был бы недопустим, а бо́льшая ее часть не использовалась бы. Например, число идентификаторов Фортрана (буква, за которой следует до пяти букв или цифр) равно примерно $1{,}33 \times 10^9$.

Другой возможный метод запоминания — использование магазина. Когда встречается новый объект, его имя и указатель на информацию, относящуюся к этому объекту, помещаются в верхушку магазина. В этом случае размер таблицы пропорционален числу действительно встретившихся объектов, а новые объекты можно вставить очень быстро. Но извлечение информации об объекте требует просмотра магазина до тех пор, пока объект не будет обнаружен. Таким образом, в среднем поиск требует времени, пропорционального числу объектов в магазине. Этот метод часто бывает удобен для небольшого числа объектов. Кроме того, он имеет преимущества при компиляции языков с блочной структурой, так как, помимо старого объявления переменной, в таблицу можно поместить и новое. Когда блок закончен, все

его объявления выталкиваются из магазина, а старые объявления переменных остаются.

Третий метод, более быстрый, чем использование магазина, заключается в применении *дерева двоичного поиска*. В дереве двоичного поиска у каждой вершины может быть *левый прямой потомок* и *правый прямой потомок*. Предполагается, что объекты данных можно линейно упорядочить некоторым отношением $<$, например отношением „предшествовать в алфавитном порядке". Объекты запоминаются как метки в вершинах дерева. Когда встречается первый объект, скажем α_1, порождается корень, который помечается α_1. Если α_2 — следующий объект и $\alpha_2 < \alpha_1$, то к дереву добавляется лист, помеченный α_2, и он становится левым прямым потомком корня. (Если $\alpha_1 < \alpha_2$, то этот лист становится правым прямым потомком.) Появление каждого нового объекта приводит к тому, что к дереву добавляется новый лист в такой позиции, что дерево всегда обладает следующим свойством. Пусь N — вершина дерева, помеченная β. Если N имеет левое поддерево, содержащее вершину, помеченную α, то $\alpha < \beta$; если N имеет правое поддерево с вершиной, помеченной γ, то $\beta < \gamma$.



Для внесения объектов в дерево двоичного поиска можно применить следующий алгоритм.

Алгоритм 10.1. Внесение объектов в дерево двоичного поиска.

Вход. Последовательность объектов $\alpha_1, \ldots, \alpha_n$ из множества объектов A с линейным порядком $<$ на A.

Выход. Двоичное дерево, каждая из вершин которого помечена одним из элементов $\alpha_1, \ldots, \alpha_n$ и такое, что если вершина N помечена α и некоторый ее потомок N' помечен β, то $\beta < \alpha$ тогда и только тогда, когда N' — левое поддерево вершины N.

Метод.

(1) Создать одиночную вершину (корень) и поместить ее α_1.

(2) Предположим, что $\alpha_1, \ldots, \alpha_{i-1}$ уже размещены в дереве, $i > 0$. Если $i = n+1$, остановиться. В противном случае внести α_i в дерево, выполняя шаг (3) начиная с корня.

(3) Пусть этот шаг выполняется в вершине N с меткой β.

(а) Если $\alpha_i < \beta$ и N имеет левого прямого потомка N_l, выполнить шаг (3) в вершине N_l. Если N не имеет левого прямого потомка, создать его и пометить α_i. Вернуться к шагу (2).

б) Если $\beta < \alpha_i$ и N имеет правого прямого потомка N_r, выполнить шаг (3) в вершине N_r. Если N не имеет правого прямого потомка, создать его и пометить α_i. Вернуться к шагу (2). □

Метод поиска объекта по существу совпадает с методом внесения, за исключением того, что на шаге (3) в каждой вершине надо проверять, является ли ее метка требуемым объектом.

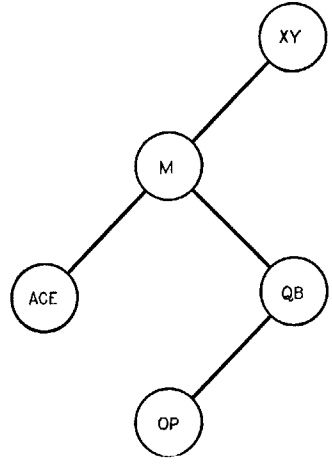


Рис. 10.3. Дерево двоичного поиска.

Пример 10.3. Пусть на вход алгоритма 10.1 подается последовательность объектов XY, M, QB, ACE и OP. Упорядочение предполагается алфавитным. Построенное дерево изображено на рис. 10.3. □

Можно показать, что, после того как в дерево двоичного поиска помещено n объектов, ожидаемое число вершин, которые надо просмотреть для поиска одного объекта, пропорционально $\log n$. Эта цена приемлема, хотя таблицы расстановки, которые мы теперь обсудим, дают меньшее ожидаемое время поиска.

10.1.3. Таблицы расстановки

Наиболее эффективный и широко применяемый в компиляторах метод при работе с таблицами имен дают *таблицы расстановки*. Таблица имен, организованная на принципе расстановки, схематически изображена на рис. 10.4.

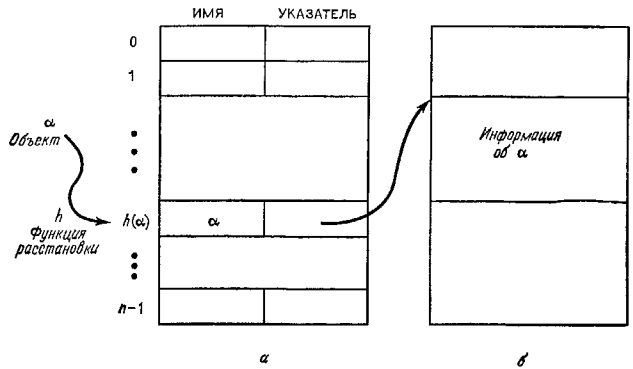


Рис. 10.4. Схема памяти, использующей расстановку: *а* — таблица расстановки, *б* — таблица данных.

Механизм расстановки состоит из *функции расстановки* h, *таблицы расстановки* и *таблицы данных*. Таблица расстановки содержит n элементов, где n фиксировано заранее. Каждый элемент в таблице расстановки имеет два поля: поле имени и поле указателя. Предполагается, что вначале все элементы в таблице расстановки пусты[1]). Если ранее встретился объект α, то в некоторой ячейке таблицы расстановки, обычно $h(\alpha)$, есть элемент, поле имени которого содержит α (или, возможно, указатель на ячейку в таблице имен, где хранится α) и поле указателя которого содержит указатель на часть таблицы данных, содержащую информацию, связанную с α.

[1]) Иногда вначале удобно помещать в таблицу расстановки зарезервированные слова и имена стандартных функций.

Таблица данных физически может совпадать с таблицей расстановки. Например, если для каждого объекта нужно k слов информации, то можно использовать таблицу расстановки размером kn. Каждый объект, запомненный в таблице расстановки, занимал бы часть из k последовательных слов памяти. Ячейку таблицы расстановки, соответствующую объекту α, легко найти, умножая адрес $h(\alpha)$ расстановки для α на k и беря полученный адрес в качестве первой ячейки набора слов для α.

Функция расстановки h — это фактически набор функций h_0, $h_1, \ldots, h_m$, каждая из которых отображает набор объектов в множество целых чисел $\{0, 1, \ldots, n-1\}$. Функцию h_0 будем называть *первичной функцией расстановки*. Когда встречается новый объект α, для вычисления $h(\alpha)$ можно воспользоваться следующим алгоритмом. Если объект уже встречался ранее, то $h(\alpha)$ — ячейка в таблице расстановки, в которой хранится α. Если объект α ранее не встречался, то $h(\alpha)$ — пустая ячейка, в которой можно запомнить α[1]).

Алгоритм 10.2. Вычисление адреса таблицы расстановки.

Вход. Объект α, функция расстановки h, состоящая из последовательности функций $h_0, h_1, \ldots, h_m$, каждая из которых отображает множество объектов в множество целых чисел $\{0, 1, \ldots, n-1\}$, и (не обязательно пустая) таблица расстановки с n ячейками.

Выход. Адрес расстановки $h(\alpha)$ и указание, встречался ли объект α ранее. Если α уже есть в таблице расстановки, то $h(\alpha)$ — ячейка, отведенная для α. Если объект α ранее не встречался, то $h(\alpha)$ — пустая ячейка, в которой α запоминается.

Метод.

(1) Вычисляем по порядку $h_0(\alpha), h_1(\alpha), \ldots, h_m(\alpha)$, выполняя шаг (2) до тех пор, пока не возникнет „конфликт". Если $h_m(\alpha)$ дает конфликт, останавливаемся и сообщаем о неудаче.

(2) Вычисляем $h_i(\alpha)$:

(а) Если ячейка $h_i(\alpha)$ в таблице расстановки пуста, полагаем $h(\alpha) = h_i(\alpha)$, сообщаем, что объект α ранее не встречался, и останавливаемся.

(б) Если ячейка $h_i(\alpha)$ не пуста, проверяем поле ее имени[2]). Если именем является α, полагаем $h(\alpha) = h_i(\alpha)$, со-

[1]) Строго говоря, при таком определении h не функция, поскольку один и тот же объект может отображаться в различные позиции таблицы. В действительности h — функция двух аргументов: α и самой таблицы. — *Прим. перев.*

[2]) Если поле имени содержит указатель на таблицу имен, надо проверить по этой таблице действительное имя.

общаем, что объект α уже встречался, и останавливаемся. Если именем является не α, то возник конфликт; повторяем шаг (2) для вычисления другого адреса. □

Каждый раз, когда проверяется ячейка $h_j(\alpha)$, мы говорим, что осуществляется так называемая *проба* таблицы.

Когда таблица расстановки слабо заполнена, конфликты бывают редко и для нового объекта α значение $h(\alpha)$ можно вычислить очень быстро. Обычно для этого достаточно вычислить значение первичной функции расстановки $h_0(\alpha)$. По мере заполнения таблицы растет вероятность того, что для каждого нового объекта α ячейка $h_0(\alpha)$ будет уже содержать другой объект. Таким образом, конфликты становятся тем более частыми, чем больше объектов внесено в таблицу, так что число проб, требуемых для определения $h(\alpha)$, увеличивается. Однако можно так конструировать таблицы расстановки, что они по всем характеристикам будут значительно превосходить деревья двоичного поиска.

В идеале хотелось бы иметь такую первичную функцию расстановки h_0, чтобы она давала различные положения в таблице расстановки для любых двух различных объектов. Разумеется это, вообще говоря, невозможно, поскольку общее число возможных объектов обычно значительно больше числа n позиций в таблице. На практике n делают несколько больше ожидаемого числа различных объектов. Однако необходимо предусматривать некоторые действия на тот случай, если таблица будет переполнена.

Для того чтобы запомнить информацию об α, сначала вычисляется $h(\alpha)$. Если объект α ранее не встречался, то его имя запоминается в поле имени позиции $h(\alpha)$. (Если используется отдельная таблица имен, то α запоминается в следующей ее пустой позиции, а указатель на эту позицию помещается в поле имени ячейки $h(\alpha)$.) Затем в таблице данных отводится очередной подходящий участок памяти, а указатель на него помещается в поле указателя ячейки $h(\alpha)$. Потом информация размещается в этом участке памяти таблицы данных.

Точно так же, чтобы найти информацию об α, можно вычислить $h(\alpha)$ с помощью алгоритма 10.2 (если α есть в таблице). Для локализации в памяти таблицы данных информации, относящейся к α, используется указатель в поле указателя.

Пример 10.4. Возьмем $n=10$, и пусть объектами будут произвольные цепочки прописных латинских букв. Определим CODE(α), где α — цепочка букв, как сумму „числовых значений" каждой буквы в α, считая, что A имеет числовое значение 1, B — числовое значение 2 и т. д. Определим $h_j(\alpha)$ для $0 \leqslant j \leqslant 9$

как $(\mathrm{CODE}(\alpha)+j) \bmod 10$[1]). Внесем в таблицу расстановки объекты A, W и EF.

Вычисляем $h_0(\mathrm{A})=(1 \bmod 10)=1$, так что A вносим в позицию 1. Объект W вносим в позицию $h_0(\mathrm{W})=(23 \bmod 10)=3$. Затем встречается EF. Вычисляем $h_0(\mathrm{EF})=(5+6) \bmod 10=1$. Поскольку позиция 1 уже занята, пробуем $h_1(\mathrm{EF})=(5+6+1) \bmod 10=2$. Таким образом, для EF отводится позиция 2.

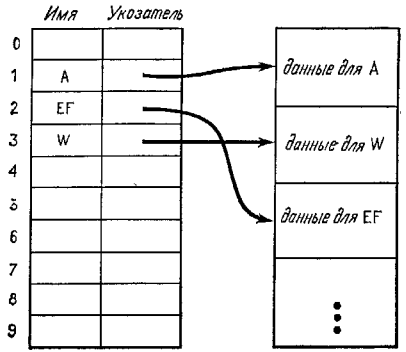


Рис. 10.5. Содержимое таблицы расстановки.

Содержимое таблицы расстановки к этому моменту приведено на рис. 10.5.

Предположим, что мы хотим выяснить, есть ли в таблице объект HX. Находим $h_0(\mathrm{HX})=2$. С помощью алгоритма 10.2 проверяем позицию 2 и видим, что возник конфликт, поскольку позиция 2 занята, но не объектом HX. Проверяем затем позицию $h_1(\mathrm{HX})=3$ и снова получаем конфликт. Наконец, вычисляем $h_2(\mathrm{HX})=4$ и находим, что позиция 4 пуста. Приходим, таким образом, к выводу, что объекта HX в таблице нет. □

10.1.4. Функции расстановки

Желательно пользоваться такой первичной функцией расстановки h_0, которая распределяла бы объекты равномерно по всей таблице расстановки. Надо избегать функций, которые отображают не на все множество позиций или выбирают одни позиции

[1]) $a \bmod b$ — это остаток от деления a на b.

чаще, чем другие, а также функций, требующих больших вычислений.

Перечислим наиболее широко используемые первичные функции:

(1) Если α занимает в машине несколько слов, можно вычислить арифметическую (или логическую) сумму этих слов, чтобы получить *представление* α в виде *единственного слова*. Это слово теперь можно трактовать как число, извлечь квадратный корень и взять в качестве $h_0(\alpha)$ средние $\log_2 n$ битов результата. Поскольку средние биты квадратного корня зависят от всех символов объекта, различные объекты в среднем будут давать различные адреса независимо от того, имеют ли объекты общие префиксы или суффиксы.

(2) Единственное слово, представляющее α, можно разбить на несколько частей фиксированной длины (обычно по $\log_2 n$ битов). Затем можно просуммировать эти части и взять в качестве $h_0(\alpha)$ младшие $\log_2 n$ битов суммы.

(3) Единственное слово, представляющее α, можно разделить на размер n таблицы и взять в качестве $h_0(\alpha)$ остаток от деления (в этом случае n должно быть простым числом).

Рассмотрим теперь методы разрешения конфликтов, т. е. построения дополнительных функций $h_1, h_2, \ldots, h_m$. Прежде всего отметим, что значение $h_i(\alpha)$ должно отличаться от $h_j(\alpha)$ для всех $i \neq j$. Когда $h_i(\alpha)$ дает конфликт, бессмысленно пробовать $h_{i+r}(\alpha)$, если $h_{i+r}(\alpha) = h_i(\alpha)$. В большинстве случаев хотелось бы, чтобы было $m = n-1$, поскольку всегда желательно найти в таблице пустую позицию, если она есть. В общем случае метод разрешения конфликтов оказывает сильное влияние на эффективность всей системы с распределенной памятью.

Простейшим, но, как мы увидим, одним из самых худших, набором функций $h_1, h_2, \ldots, h_{n-1}$ является

$$h_i(\alpha) = (h_0(\alpha) + i) \bmod n \qquad 1 \leqslant i \leqslant n-1$$

В этом случае мы движемся вперед относительно значения $h_0(\alpha)$ первичной позиции до тех пор, пока не исчезнет конфликт. Если достигнута позиция $n-1$, переходим на позицию 0. Метод прост для выполнения, однако если уже возник конфликт, то занятые позиции имеют тенденцию скапливаться. Например, если известно, что $h_0(\alpha)$ вызывает конфликт, то вероятность того, что $h_1(\alpha)$ также вызывает конфликт, выше средней.

Более эффективный метод получения вторичных адресов заключается в выборе

$$h_i(\alpha) = (h_0(\alpha) + r_i) \bmod n \qquad 1 \leqslant i \leqslant n-1$$

где r_i — псевдослучайное число. Часто для этой цели достаточен самый элементарный генератор случайных чисел, дающий

все числа в диапазоне от 1 до $n-1$ в точности по одному разу (см. упр. 10.1.8). Каждый раз, когда используются вторичные функции, генератор случайных чисел устанавливается в одно и то же состояние. Таким образом, каждый раз, когда происходит обращение к h, генерируется одна и та же последовательность $r_1, r_2, \ldots$, и, следовательно, поведение генератора „случайных" чисел вполне детерминировано.

В качестве вторичных функций можно взять также

$$h_i(\alpha) = [i(h_0(\alpha) + 1)] \bmod n$$

и

$$h_i(\alpha) = [h_0(\alpha) + ai^2 + bi] \bmod n$$

где a и b — подходящие константы.

Несколько отличный от этого метод разрешения конфликтов, называемый методом цепочек, разбирается в упражнениях.

10.1.5. Эффективность таблиц расстановки

Сколько требуется в среднем времени для внесения и поиска данных в таблице расстановки? С этим вопросом связан следующий: даны вероятности появления различных объектов; как выбрать функции $h_0, \ldots, h_{n-1}$, чтобы оптимизировать работу с таблицей расстановки? Интересно, что здесь есть ряд нерешенных вопросов.

Как мы уже отмечали, было бы неразумно иметь повторения в последовательности позиций $h_0(\alpha), \ldots, h_{n-1}(\alpha)$ для какого-нибудь объекта α. Будем считать, что хорошая система функций расстановки не допускает повторений. Например, последовательность $h_0, \ldots, h_9$ из примера 10.4 такова, что ни для какого объекта одна и та же позиция не будет проверяться дважды.

Если n — размер таблицы расстановки с позициями, занумерованными от 0 до $n-1$, и $h_0, \ldots, h_{n-1}$ — функции расстановки, то каждому объекту α можно поставить в соответствие перестановку Π_α множества $\{0, \ldots, n-1\}$, а именно $\Pi_\alpha = [h_0(\alpha), \ldots, h_{n-1}(\alpha)]$. Таким образом, первая компонента перестановки Π_α дает первичную позицию, отводимую для α, вторая компонента — следующую возможную позицию и т. д. Если для каждого объекта α известна вероятность p_α его появления, то можно определить *вероятность перестановки* Π как $\sum p_\alpha$, где сумма берется по всем таким объектам α, что $\Pi_\alpha = \Pi$[1]). Отныне будем полагать, что вероятности всех перестановок даны.

[1]) Иными словами, это сумма вероятностей появления объектов, которые образуют данную перестановку Π. — *Прим. перев.*

Ясно, что ожидаемое число позиций, которые надо проверить при внесении или поиске объекта, можно вычислить, зная только вероятности перестановок. Для вычисления эффективности конкретной функции расстановки не обязательно знать реальные функции $h_0, \ldots, h_{n-1}$. В этом разделе мы изучим свойства функций расстановки, определяемые вероятностями перестановок.

Основываясь на сказанном выше, можно дать следующее определение, которое сводит задачу построения таблицы расстановки к вопросу о том, каков желательный набор вероятностей перестановок.

Определение. *Системой расстановки* назовем пару, состоящую из размера таблицы n и функции вероятности p, определенной на перестановках целых чисел от 0 до $n-1$. Систему расстановки будем называть *случайной*, если $p(\Pi)=1/n!$ для каждой из перестановок Π.

Укажем важные функции, связанные с системой расстановки:

(1) $p(i_1 i_2 \ldots i_k)$ — вероятность того, что некоторый объект имеет i_1 в качестве первичной позиции, i_2 в качестве первой вторичной, i_3 в качестве следующей вторичной и т. д. (здесь каждое i_j — это целое между 0 и $n-1$),

(2) $p(\{i_1, i_2, \ldots, i_k\})$ — вероятность того, что последовательность k объектов заполнит в точности множество позиций $\{i_1, i_2, \ldots, i_k\}$.

Легко вывести следующие формулы:

$$p(i_1 \ldots i_k) = \sum_{i \notin \{i_1, \ldots, i_k\}} p(i_1 \ldots i_k i), \text{ если } k < n \tag{10.1.2}$$

$$p(i_1 \ldots i_n) = p(\Pi) \tag{10.1.3}$$

где Π — перестановка $[i_1, \ldots, i_n]$.

$$p(S) = \sum_{i \in S} p(S - \{i\}) \sum_{w} p(wi) \tag{10.1.4}$$

где S — произвольное подмножество множества $\{0, \ldots, n-1\}$, состоящее из k элементов, а правая сумма берется по всем цепочкам w из $k-1$ или менее различных позиций в $S-\{i\}$. Полагаем, что $p(\varnothing)=1$.

Формула (10.1.2) позволяет вычислять по вероятностям перестановок вероятность любой последовательности вторичных позиций. Формула (10.1.4) позволяет вычислять вероятность того, что позициями, отведенными для k объектов, являются в точности те, которые составляют множество S. Эта вероятность равна сумме, взятой по всем $i \in S$, вероятностей того, что первые $k-1$ объектов займут все позиции в S, кроме позиции i, а последний объект займет позицию i.

Пример 10.5. Пусть $n=3$, а вероятности шести перестановок приведены в табл. 10.1.

Таблица 10.1

Перестановка	Вероятность
[0, 1, 2]	0.1
[0, 2, 1]	0.2
[1, 0, 2]	0.1
[1, 2, 0]	0.3
[2, 0, 1]	0.2
[2, 1, 0]	0.1

По (10.1.3) вероятность того, что в результате применения функции расстановки к некоторому объекту будет выработана цепочка позиций 012, равна просто вероятности перестановки [0, 1, 2]. Таким образом, $p(012)=0.1$. По (10.1.2) вероятность того, что вырабатывается 01, равна $p(012)$. Аналогично $p(02)$ — вероятность перестановки [0, 2, 1], равная 0.2. Применяя формулу (10.1.2), получаем, что $p(0)=p(01)+p(02)=0.3$. Вероятность появления каждой цепочки длины 2 или 3 — это вероятность той единственной перестановки, префиксом которой служит данная цепочка. Вероятности появления цепочек 1 и 2 равны соответственно $p(10)+p(12)=0.4$ и $p(20)+p(21)=0.3$.

Вычислим теперь вероятности заполнения различных множеств позиций. Для множеств S, состоящих из одного элемента, (10.1.4) сводится к $p(\{i\})=p(i)$. Вычислим $p(\{0, 1\})$ по формуле (10.1.4). Прямая подстановка дает

$$\begin{aligned} p(\{0,1\}) &= p(\{0\})[p(1)+p(01)] + p(\{1\})[p(0)+p(10)] \\ &= 0.3[0.4+0.1] + 0.4[0.3+0.1] = 0.31 \end{aligned}$$

Аналогично находим

$$p(\{0,2\}) = 0.30 \quad \text{и} \quad p(\{1,2\}) = 0.39$$

Для вычисления $p(\{0,1,2\})$ по формуле (10.1.4) подсчитываем

$$\begin{aligned} &p(\{0,1\})[p(2)+p(02)+p(12)+p(012)+p(102)] \\ &+ p(\{0,2\})[p(1)+p(01)+p(21)+p(021)+p(201)] \\ &+ p(\{1,2\})[p(0)+p(10)+p(20)+p(120)+p(210)] \end{aligned}$$

что, конечно, равно 1. □

Основной величиной, которая нам понадобится для оценки системы расстановки, будет ожидаемое число проб, необходимых для того, чтобы внести объект α в таблицу, в которой k пози-

ций из n заполнены. Успех достигается при первой же пробе, если $h_0(\alpha)=i$ и i — одна из $n-k$ пустых позиций. Таким образом, вероятность того, что успех будет достигнут при первой пробе, равна

$$\sum_{i=0}^{n-1} p(i) \sum_S p(S)$$

где правая сумма берется по всем множествам S из k позиций, не содержащим i.

Если $h_0(\alpha) \in S$, но $h_1(\alpha) \notin S$, то успех будет достигнут при второй попытке. Поэтому вероятность того, что первая попытка будет неудачной, а вторая — удачной, равна

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} p(ij) \sum_S p(S)$$

где правая сумма берется по всем множествам S из k позиций, содержащим i и не содержащим j. Отметим, что $p(ij)=0$ при $i=j$.

Продолжая в том же духе, получаем формулу *ожидаемого числа $E(k, n)$ проб, требуемых для внесения объекта в таблицу, в которой k из n позиций заняты, $k<n$*:

$$E(k, n)=\sum_{m=1}^{k+1} m \sum_w p(w) \sum_S p(S) \tag{10.1.5}$$

где

(1) средняя сумма берется по всем цепочкам w различных позиций длины m,

(2) правая сумма берется по всем таким множествам S, состоящим из k позиций, что в w все символы, кроме последнего, принадлежат S (последний символ цепочки w не принадлежит S).

Предполагается, что в первой сумме для вычисления первичной позиции и первых $m-1$ вторичных позиций требуется m шагов. Отметим, что при $k<n$ пустая позиция всегда отыскивается не более чем за $k+1$ попыток.

Пример 10.6. Используем вероятности примера 10.5 для вычисления $E(2, 3)$. По формуле (10.1.5)

$$\begin{aligned}E(2, 3)&=\sum_{m=1}^{3} m \sum_{|w|=m} p(w) \sum p(S)\\
&=p(0)\,p(\{1, 2\})+p(1)\,p(\{0, 2\})+p(2)\,p(\{0, 1\})\\
&+2[p(01)\,p(\{0, 2\})+p(10)\,p(\{1, 2\})+p(02)\,p(\{0, 1\})\\
&+p(20)\,p(\{1, 2\})+p(12)\,p(\{0, 1\})+p(21)\,p(\{0, 2\})\\
&+3[p(012)\,p(\{0, 1\})+p(021)\,p(\{0, 2\})+p(102)\,p(\{0, 1\})\\
&+p(120)\,p(\{1, 2\})+p(201)\,p(\{0, 2\})+p(210)\,p(\{1, 2\})=2.008\end{aligned}$$

Правая сумма берется по всем множествам S из двух элементов, содержащим все символы цепочки w, кроме последнего. □

Для оценки систем расстановки нам пригодится также *ожидаемое число $R(k, n)$ проб, требуемых для поиска объекта в таблице, в которой k из n позиций заняты.* Однако этот показатель легко выразить через $E(k, n)$. Можно предположить, что каждый из k находящихся в таблице объектов отыскивается с равной вероятностью. Следовательно, ожидаемое время поиска равно среднему числу проб, которые раньше потребовались на то, чтобы внести эти k объектов в таблицу. Таким образом,

$$R(k, n)=\frac{1}{k}\sum_{i=0}^{k-1} E(i, n)$$

Поэтому основным показателем мы будем считать $E(k, n)$.

Естественно думать, что случайная система расстановки лучше всех остальных, поскольку любая неслучайность может привести только к тому, что вероятность заполнения для одних позиций будет больше, чем для других, и при попытке внести новые объекты эти позиции будут чаще проверяться. Однако мы покажем, что это не совсем так, и точный оптимум не известен. Мы полагаем, что случайная расстановка оптимальна в смысле минимального времени поиска, а другие используемые системы расстановки существенно ей уступают. Поэтому вычислим $E(k, n)$ для случайной системы расстановки.

Лемма 10.1. *Если система расстановки случайна, то*

(1) *для всех последовательностей w позиций, для которых $1 \leqslant |w| \leqslant n$,*

$$p(w)=(n-|w|)!/n!$$

(2) *для всех подмножеств S множества $\{0, 1, \ldots, n-1\}$*

$$p(s)=\frac{1}{\binom{n}{\#S}}$$

Доказательство. (1) Это доказывается элементарной индукцией по $(n-|w|)$, начиная $|w|=n$ и кончая $|w|=1$, с учетом формулы (10.1.2).

(2) Простые рассуждения с использованием симметрии приводят к выводу, что значение $p(S)$ одинаково для всех подмножеств S из k элементов. Поскольку число множеств мощности k равно $\binom{n}{k}$, утверждение (2) доказано. □

Лемма 10.2. *Если $n \geqslant k$, то* $\sum_{l=0}^{k} \binom{n-l}{k-l}=\binom{n+1}{k}$.

Доказательство. Упражнение. □

Теорема 10.1. *Если система расстановки случайна, то* $E(k, n) = (n+1)/(n+1-k)$.

Доказательство. Предположим, что в таблице расстановки k из n позиций заполнены. Требуется внести $(k+1)$-й объект α. Из леммы 10.1(2) следует, что для любого множества из k позиций вероятность быть заполненным одинакова. Таким образом, $E(k, n)$ не зависит от того, какие именно k позиций заполнены. Поэтому без потери общности можно считать, что заполнены позиции $0, 1, 2, \ldots, k-1$.

При вычислении ожидаемого числа проб, требуемых для внесения α, нас интересует последовательность адресов, получаемых при применении h к α. Пусть этой последовательностью будет $h_0(\alpha), h_1(\alpha), \ldots, h_{n-1}(\alpha)$. По определению все такие последовательности из n позиций равновероятны.

Пусть q_j — вероятность того, что первые $j-1$ позиций в этой последовательности принадлежат $\{0, 1, \ldots, k-1\}$, а позиция j не принадлежит. Ясно, что ожидаемое число $E(k, n)$ проб, необходимых для внесения α, равно $\sum_{j=1}^{k+1} jq_j$. Заметим, что

$$\sum_{j=1}^{k+1} jq_j = \sum_{m=1}^{k+1} \sum_{j=1}^{m} q_m = \sum_{j=1}^{k+1} \sum_{m=j}^{k+1} q_m \tag{10.1.6}$$

Но $\sum_{m=j}^{k+1} q_m$ — это как раз вероятность того, что первые $j-1$ позиций в последовательности $h_0(\alpha), h_1(\alpha), \ldots, h_{n-1}(\alpha)$ лежат между 0 и $k-1$, т. е. что для внесения $(k+1)$-го объекта требуется по крайней мере j проб. По лемме 10.1 (1) эта величина равна

$$\left(\frac{k}{n}\right)\left(\frac{k-1}{n-1}\right)\cdots\left(\frac{k-j+2}{n-j+2}\right) = \frac{k!}{(k-j+1)!}\,\frac{(n-j+1)!}{n!} = \frac{\binom{n-j+1}{k-j+1}}{\binom{n}{k}}$$

Тогда, применяя лемму 10.2, получаем

$$E(k, n) = \sum_{j=1}^{k+1} \frac{\binom{n-j+1}{k-j+1}}{\binom{n}{k}} = \frac{\binom{n+1}{k}}{\binom{n}{k}} = \frac{n+1}{n-k+1} \qquad \square$$

Из теоремы 10.1 видно, что при больших n и k функция $E(k, n)$ зависит только от отношения k/n и приблизительно равна $1/(1-\rho)$, где $\rho = k/n$. График этой функции приведен на рис. 10.6.

Отношение k/n называется *коэффициентом загрузки*. Когда коэффициент загрузки мал, время внесения как функция числа k заполненных позиций растет медленнее, чем $\log k$, и расстановка предпочтительнее двоичного поиска. Конечно, при приближении к n, т. е. по мере заполнения таблицы, внесение становится дорогим, а при $k = n$ вообще невозможным, если нет специальных механизмов обработки переполнений. Один метод, связанный с переполнением, предлагается в упр. 10.1.11 и 10.1.12.

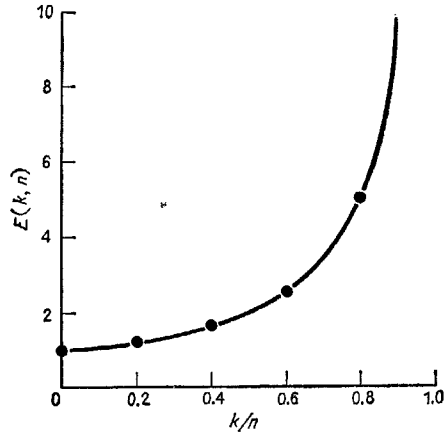


Рис. 10.6. $E(k, n)$ как функция коэффициента загрузки для случайной расстановки.

Ожидаемое число попыток при внесении объекта является не единственным критерием качества схемы расстановки. Желательно также, чтобы вычисление функций расстановки было простым. В схемах расстановки, которые мы до сих пор рассматривали, вторичные функции $h_1(\alpha), \ldots, h_{n-1}(\alpha)$ вычислялись не от α, а от $h_0(\alpha)$, и это характерно для большинства схем расстановки. Такая организация эффективна, поскольку $h_0(\alpha)$ — целое число известной длины, в то время как α может иметь произвольную длину. Этот метод мы будем называть *расстановкой по позициям*. Частный случай этого метода, еще более легкий с точки зрения выполнения, — *линейная расстановка*, где $h_i(\alpha)$ вычисляется как $(h_0(\alpha) + i) \bmod n$. В этом случае перебираются последовательные ячейки, пока не отыскивается пустая; если достигнут конец таблицы, переходим на ее начало. Линейная расстановка используется в примере 10.4. Приведем теперь пример, показывающий, что по ожидаемому числу попыток внесения линейная расстановка может быть хуже случайной. Затем

рассмотрим расстановку по позициям и покажем, что по крайней мере в одном случае она также хуже случайной расстановки.

Пример 10.7. Рассмотрим систему линейной расстановки с $n=4$ и вероятностью $^1/_4$ для каждой из четырех перестановок [0123], [1230], [2301] и [3012]. Читатель может убедиться в том, что если вероятности сделать неравными, то система станет только хуже. Для случайной расстановки теорема 10.1 дает $E(2,4)=^5/_3$.

По формуле (10.1.4) можно вычислить вероятности того, что заполнено множество из двух позиций. Эти вероятности таковы:

$$p(\{0,\ 1\})=p(\{1,\ 2\})=p(\{2,\ 3\})=p(\{s,\ 0\})={}^3/_{16}$$
$$p(\{0,\ 2\})=p(\{1,\ 3\})={}^1/_8$$

Затем по формуле (10.1.5) находим, что для линейной расстановкн $E(2,4)=27/16$, а это больше $^5/_3$ — стоимости случайной расстановки. □

Сравним теперь расстановку по позициям со случайной расстановкой в частном случае, когда в таблицу вносится третий объект. Отметим, что при расстановке по позициям для каждой позиции i точно одна перестановка Π_i начинается с i и имеет ненулевую вероятность. Вероятность перестановки Π_i обозначим через p_i. Второй элемент в Π_i, т. е. первую вторичную позицию для i, обозначим через a_i. Если $p_i=1/n$ для всех i, то систему будем называть *случайной расстановкой по позициям.*

Теорема 10.2. *При всех $n>3$ $E(2,\ n)$ для случайной расстановки меньше, чем для случайной расстановки по позициям.*

Доказательство. По теореме 10.1 для случайной расстановки $E(2,\ n)=(n+1)/(n-1)$. Найдем нижнюю границу функции $E(2,\ n)$ для расстановки по позициям. Предположим, что первые три объекта, вносимые в таблицу, имеют перестановки Π_i, Π_j и Π_k соответственно. Рассмотрим отдельно случаи $i=j$ и $i\neq j$.

Случай 1: $i\neq j$. Это происходит с вероятностью $(n-1)/n$. Ожидаемое число попыток при внесении третьего объекта оказывается равным

$$1+(2/n)+(2/n)[1/(n-1)]=(n+1)/(n-1)$$

Случай 2: $i=j$. Это происходит с вероятностью $1/n$. Тогда третий объект вносится с первой попытки с вероятностью $(n-2)/n$; при этом $k\neq i$ и $k\neq a_i$ (a_i — вторая заполненная позиция). Вероятность того, что $k=a_i$, равна $1/n$; нужны по крайней мере две попытки. Вероятность того, что $k=i$, также равна $1/n$, и надо сделать три попытки. (Это вытекает из того, что

вторая попытка делается для позиции a_i, которую занимает j.) Таким образом, в этом случае ожидаемое число проб не менее $[(n-2)/n]+(2/n)+(3/n)=(n+3)/n$.

Взвешивая эти два случая в соответствии с их вероятностями, получаем для случайной расстановки по позициям

$$E(2,\ n)\geqslant\left(\frac{n+1}{n-1}\right)\left(\frac{n-1}{n}\right)+\left(\frac{n+3}{n}\right)\left(\frac{1}{n}\right)=\frac{n^2+2n+3}{n^2}$$

Последнее выражение превосходит $(n+1)/(n-1)$ для $n>3$. □

Основное в предыдущем примере и теореме заключается в том, что многие простые схемы расстановки не удовлетворяют требованиям определения случайной расстановки. Интуитивно это объясняется тем, что при использовании неслучайных схем стремятся пробовать одну и ту же позицию снова и снова. Даже если коэффициент загрузки мал, некоторые позиции с большой вероятностью испытываются многократно. При использовании такой схемы, как расстановка по позициям, каждый раз, когда заполняется первичная позиция $h_0(\alpha)$, все вторичные позиции для $h_0(\alpha)$, которые уже испытывались, будут испытываться снова, а это ведет к плохим характеристикам.

Из сказанного вовсе не следует, что не стóит пользоваться такими схемами, как расстановка по позициям, особенно если затраты во времени на одну попытку занесения компенсируются за счет простоты реалнзации.

В действительности весьма вероятно, что случайная расстановка — не самое лучшее из всего, что можно применить. Следующий пример показывает, что при применении случайной расстановки $E(k,\ n)$ не всегда достигает минимума. Мы, однако, думаем, что случайная расстановка дает минимум $R(k,\ n)$ — ожидаемого времени поиска объекта.

Пример 10.8. Пусть перестановки [0123] и [1032] имеют вероятности 0.2, [2013], [2103], [3012] и [3102] имеют вероятности 0.15, а все остальные имеют нулевые вероятности. Можно вычислить $E(2,4)$ непосредственно по формуле (10.1.5) и получить 1.665. Это значение меньше, чем $^5/_3$ для случайной расстановки. □

УПРАЖНЕНИЯ

10.1.1. С помощью алгоритма 10.1 внесите в дерево двоичного поиска последовательность объектов T, D, H, F, A, P, O, Q, W, TO, TH. Считайте, что объекты упорядочены по алфавиту.

10.1.2. Разработайте алгоритм, на вход которого подается дерево двоичного поиска и который перечисляет по порядку все

элементы, запомненные в дереве. Примените этот алгоритм к дереву, построенному в упр. 10.1.1.

***10.1.3.** Покажите, что ожидаемое время внесения (или поиска) одного объекта в дерево двоичного поиска составляет $0(\log n)$, где n — число вершин в дереве. Каково максимальное значение времени, требуемого для внесения одного произвольного объекта?

***10.1.4.** Какая информация о переменных и константах Фортрана необходима в таблице имен для генерации кода?

10.1.5. Опишите механизм хранения таблицы имен для языка с блочной структурой, такого, как Алгол, в котором область действия переменной X ограничена данным блоком и всеми теми блоками, содержащимися в данном, в которых переменная X не переобъявлена.

10.1.6. Выберите размер таблицы и первичную функцию расстановки h_0. Вычислите $h_0(\alpha)$, где α берется из множества (а) ключевых слов Фортрана, (б) ключевых слов Алгола и (в) ключевых слов ПЛ/I. Каково максимальное число объектов с одинаковым значением первичного адреса расстановки? Можете провести эти вычисления на ЭВМ. Саммет [1969] дает необходимые наборы ключевых слов.

***10.1.7.** Покажите, что функция $R(k, n)$ для случайной расстановки приблизительно равна $(-1/\rho)\log(1-\rho)$, где $\rho = k/n$. Начертите график этой функции. *Указание*: Аппроксимируйте $(n/k)\sum_{i=0}^{k-1}(n+1)/(n-i+1)$ интегралом.

***10.1.8.** Рассмотрим следующий генератор псевдослучайных чисел. Этот генератор вырабатывает последовательность $r_1, r_2, \ldots, r_{n-1}$ чисел, которые можно использовать для вычисления $h_i(\alpha) = [h_0(\alpha) + r_i] \bmod n$ при $1 \leqslant i \leqslant n-1$. Каждый раз, когда должна генерироваться последовательность чисел, целому R присваивается значение 1. Предполагается, что $n = 2^p$. Для того чтобы получить очередное число, надо выполнить такие шаги:

(1) $R = 5 * R$,
(2) $R = R \bmod 4n$,
(3) выдать в качестве значения $r = [R/4]$.

Покажите, что для любого i все разности $r_{i+k} - r_i$ для $k \geqslant 1$ и $i + k \leqslant n-1$ различны.

****10.1.9.** Покажите, что если $h_i = [h_0 + ai^2 + bi] \bmod n$ для $1 \leqslant i \leqslant n-1$, то различны не более половины позиций в последовательности $h_0, h_1, h_2, \ldots, h_{n-1}$. При каком условии в этой последовательности будет в точности половина различных позиций?

****10.1.10.** Каково число различных позиций в последовательности $h_0, h_1, \ldots, h_{p-1}$, если для $1 \leqslant i < p/2$

$$h_{2i-1} = [h_0 + i^2] \bmod p$$
$$h_{2i} = h_0 - \left[\left(\frac{p-1}{2}\right) + i\right]^2 \bmod p$$

и p — простое число?

Определение. Методом разрешения конфликтов, более эффективным (чем описанный в разд. 10.1.4) с точки зрения внесения и поиска, является *метод цепочек*. В этом методе каждый элемент таблицы расстановки имеет дополнительное поле, содержащее указатель на дополнительные элементы с тем же, как у него самого, первичным адресом. Все элементы с одинаковым первичным адресом соединены в последовательный список, начиная с этой первичной позиции.

Существует несколько способов реализации метода цепочек. В одном из них, называемом *прямым методом цепочек*, для хранения всех объектов используется сама таблица расстановки. Для того чтобы внести объект α, проверяется позиция $h_0(\alpha)$.

(1) Если эта позиция пуста, α помещается в ней. Если она занята и с нее начинается цепочка, находим некоторым подходящим способом пустой элемент таблицы расстановки и помещаем его в цепочку, начинающуюся с $h_0(\alpha)$.

(2) Если позиция $h_0(\alpha)$ занята, но не началом цепочки, то перемещаем текущий элемент β, находящийся в $h_0(\alpha)$, в пустую позицию таблицы расстановки и вносим α в $h_0(\alpha)$. (При этом надо заново вычислить $h(\beta)$, чтобы хранить β в соответствующей цепочке.)

Это перемещение элементов составляет главный недостаток прямого метода цепочек. Тем не менее метод довольно быстр. Другое его достоинство в том, что в переполненную таблицу с помощью той же стратегии внесения и поиска можно поместить дополнительные объекты.

10.1.11. Покажите, что если вторичные позиции выбираются случайно, то ожидаемое время поиска $R(k, n)$ для прямого метода цепочек равно $1 + \rho/2$, где $\rho = k/n$. Сравните эту функцию с $R(k, n)$ в упр. 10.1.7.

В другом методе цепочек, не требующем перемещения объектов, перед таблицей расстановки применяется таблица индексов. Первичная функция расстановки h_0 вычисляет адрес в таблице индексов. Элементами таблицы индексов служат указатели на таблицу расстановки, элементы которой заполняются по порядку. Для того чтобы при этой схеме внести объект α, вычисляется $h_0(\alpha)$, т. е. адрес в таблице индексов. Если позиция $h_0(\alpha)$ пуста,

занимаем очередиую доступную позицию в таблице расстановки и вносим в иее α. Указатель на эту позицию размещаем затем в $h_0(\alpha)$.

Если $h_0(\alpha)$ уже содержит указатель на позицию в таблице расстановки, обращаемся в эту позицию. Затем просматриваем иачинающуюся с нее цепочку. Достигнув конца цепочки, выбираем очередную доступную позицию в таблице расстановки, вносим в нее α и присоединяем эту позицию к концу цепочки.

Если таблица расстановки заполняется сверху вниз, то очередную доступиую позицию можно иайти очень быстро. В этой схеме нет необходимости перемещать объекты, поскольку каждый элемент таблицы иидексов всегда указывает на начало цепочки.

Кроме того, легко обрабатывать переполнения, добавляя место к коицу таблицы расстановки.

***10.1.12.** Каково ожидаемое время поиска объекта для метода цепочек с таблицей иидексов? Считайте, что первичные позиции распределены равиомерно.

10.1.13. Рассмотрим систему случайной расстановки с n позициями, как в разд. 10.1.5. Покажите, что если множество S состоит из k позиций и $i \notin S$, то $\sum_w p(wi) = 1/(n-k)$, где сумма берется по всем цепочкам w из k или менее различных позиций в S.

***10.1.14.** Докажите тождества

(а) $\sum_{i=0}^{k} \binom{n+i}{i} = \binom{n+k+1}{k}$,

(б) $\sum_{i=0}^{k} \binom{n-i}{k-i} = \binom{n+1}{k}$,

(в) $\sum_{i=0}^{k} i \binom{n+i}{i} = k\binom{n+k+1}{k} - \binom{n+k+1}{k-1}$.

***10.1.15.** Предположим, что объектами являются цепочки длины от одиой до шести латинских букв. Пусть CODE — функция, определениая в примере 10.4, а объект α имеет вероятность $(1/6)\,26^{-|\alpha|}$. Вычислите вероятности перестановок множества $\{0, 1, \ldots, n-1\}$, если

(а) $h_i(\alpha) = (\mathrm{CODE}(\alpha) + i) \bmod n$, $0 \leqslant i \leqslant n-1$,

(б) $h_i(\alpha) = (i(h_0(\alpha)+1)) \bmod n$, $1 \leqslant i \leqslant n-1$, где
$h_0(\alpha) = \mathrm{CODE}(\alpha) \bmod n$.

****10.1.16.** Покажите, что для линейной расстановки со случайно распределенной первичной позицией $h_0(\alpha)$ предел функции $E(k, n)$ при k и n, стремящихся к ∞, и $k/n = \rho$ равен $1 + [\rho(1-\rho/2)/(1-\rho)^2]$. Как это соотносится с соответствующей функцией от ρ для случайной расстановки?

Определение. Систему расстановки назовем *k-равномерной*, если для любого множества $S \subseteq \{0, 1, 2, \ldots, n-1\}$, состоящего из k элемеитов, вероятность того, что последовательность из k различиых позиций образована в точности позициями из S, равна $1\Big/\binom{n}{k}$ (иаиболее существенио здесь то, что вероятность не зависит от S).

***10.1.17.** Покажите, что если система расстановки k-равномерна, то, как и для случайной расстановки,

$$E(k, n) = (n+1)/(n+1-k)$$

***10.1.18.** Покажите, что для любой системы расстановки, для которой существует такое число k, что $E(k, n) < (n+1)/(n+1-k)$, существует такое $k' < k$, что

$$E(k', n) > (n+1)/(n+1-k')$$

Таким образом, если для некоторого k данная система расстановки лучше случайной, то существует меньшее число k', для которого эта система хуже случайной. *Указание*: Покажите, что если система расстановки $(k-1)$-равномерна, но не k-равномерна, то $E(k, n) > (n+1)/(n-k+1)$.

***10.1.19.** Приведите пример системы расстановки, k-равномерной для любого k, но не случайиой.

***10.1.20.** Обобщите пример 10.7 иа случай неравных вероятностей для циклических перестаиовок.

***10.1.21.** Усильте теорему 10.2 с тем, чтобы включить системы, осуществляющие расстановку по позициям, но имеющие неравиые вероятиости p_i.

Открытые проблемы

10.1.22. Оптимальна ли в смысле ожидаемого времени поиска объекта случайная расстановка. Другими словами, верно ли, что функция $R(k, n)$ всегда ограиичена снизу величиной

$$(1/k) \sum_{i=0}^{k-1} (n+1)/(n+1-i)$$

Мы думаем, что случайиая расстановка оптимальна.

10.1.23. Найдите наибольшую нижнюю границу функции $E(k, n)$ для систем, осуществляющих расстановку по позициям.

Любая нижняя граница, превышающая $(n+1)/(n+1-k)$, представляет интерес.

10.1.24. Найдите наибольшую нижнюю границу функции $E(k,n)$ для произвольной системы расстановки. В примере 10.8 мы видели, что $(n+1)/(n+1-k)$ такой границей не является.

Проблема для исследования

10.1.25. В некоторых применениях таблиц расстановки вводимые объекты известны заранее. Примерами служат библиотеки стандартных подпрограмм или таблицы кодов операций языка ассемблера. Если известно, что именно размещается в таблице расстановки, то можно выбрать систему расстановки так, чтобы минимизировать ожидаемое время просмотра. Можете ли Вы привести алгоритм, который принимает на вход список объектов, подлежащих запоминанию, и выдает систему расстановки, эффективно реализуемую и требующую мало времени при поиске для этой конкретной загрузки таблицы расстановки?

Упражнения на программирование

10.1.26. Запрограммируйте систему расстановки, осуществляющую расстановку по позициям. Проверьте поведение системы на ключевых словах Фортрана и общеупотребительных именах функций.

10.1.27. Запрограммируйте систему расстановки, использующую для разрешения конфликтов метод цепочек. Сравните поведение этой системы и системы из упр. 10.1.26.

Замечания по литературе

Таблицы расстановки известны также как таблицы с распределенной памятью, ключевые трансформационные таблицы, случайные таблицы и таблицы с вычисляемыми элементами. Таблицы расстановки применялись программистами с начала 50-х годов. Самая ранняя работа по адресам расстановки написана Петерсоном [1957]. Хороший обзор методов расстановки приведен Моррисом [1968]. У него же можно найти ответ на упр. 10.1.7.

Методы вычисления вторичных функций для уменьшения ожидаемого числа конфликтов обсуждаются Маурером [1968], Радке [1970] и Беллом [1970]. Ответ на упр. 10.1.10 можно найти у Радке [1970]. В работе Ульмана [1972] изучаются k-равномерные системы расстановки.

Хорошим справочником по деревьям двоичного поиска и таблицам расстановки является работа Кнута [1973].

10.2. ГРАММАТИКИ СВОЙСТВ

Интересный конструктивный метод присваивания свойств идентификаторам в языке программирования основан на формализме „грамматик свойств". Это контекстно-свободные грамматики с дополнительными механизмами передачи информации

об идентификаторах и обработки некоторых не контекстно-свободных аспектов синтаксиса языков программирования (таких, как требование, чтобы идентификаторы были описаны раньше их использования). В этом разделе мы дадим введение в теорию грамматик свойств и покажем, как с помощью грамматики свойств смоделировать синтаксический анализатор, соединяющий в себе разбор с некоторыми аспектами семантического анализа.

10.2.1. Мотивировка

Попробуем понять сначала, почему не всегда бывает достаточно объявления идентификатора для того, чтобы установить все его свойства и поместить их в ячейки таблицы имен, отведенные для этого идентификатора. Если мы рассмотрим язык с блочной структурой (например, Алгол или ПЛ/1), то увидим, что свойства идентификатора могут многократно меняться,

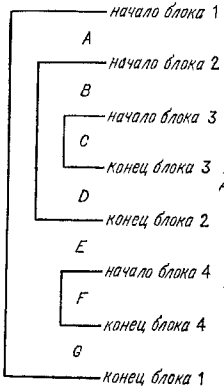


Рис. 10.7. Блочная структура программы.

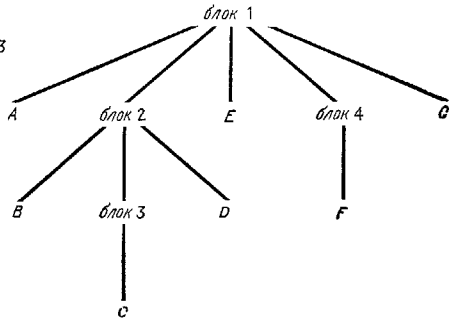


Рис. 10.8. Представление блочной структуры в виде дерева.

поскольку его можно определить в охватывающем блоке и переопределить во внутреннем. Когда внутренний блок заканчивается, свойства становятся снова такими, какими они были в охватывающем блоке.

Рассмотрим, например, диаграмму на рис. 10.7, изображающую блочную структуру программы. Буквы обозначают области действия в программе. Эту блочную структуру можно представить деревом на рис. 10.8.

Предположим, что идентификатор I определен в блоке 1 этой программы и помещен в таблицу имен сразу, как только встретился. Предположим, что после этого I переопределен

в блоке 2. В областях B, C и D этой программы для I действует новое определение. Таким образом, встретив определение I в блоке 2, надо внести это новое определение в таблицу имен. Однако нельзя просто заменить определение, которое было у I в блоке 1, новым определением в блоке 2, поскольку в области E надо будет вернуться к первоначальному определению I.

Один из путей решения этой проблемы заключается в том, что с каждым определением идентификатора связываются два числа: *номер уровня* и *индекс*. Номер уровня — это глубина вложенности, и всем блокам с одним и тем же номером уровня даются различные индексы. Например, идентификаторы в областях B и D блока 2 имеют номер уровня 2 и индекс 1. Идентификаторы в области F блока 4 имеют номер уровня 2 и индекс 2.

Если ссылка на идентификатор осуществляется в блоке с уровнем i и индексом j, то в таблице имен разыскивается определение этого идентификатора с тем же номером уровня и тем же индексом. Если, однако, в блоке с номером уровня i идентификатор не определен, то его определение надо искать в блоке с номером уровня $i-1$, содержащем блок с номером уровня i и индексом j, и т. д. Если определение появлялось на нужном уровне, но индекс слишком мал, можно отбросить определение, так как оно уже больше неприменимо. Таким образом, для запоминания определений каждого идентификатора по мере их появлений полезен магазин. Описанный поиск облегчается, если индекс текущего активного блока доступен на каждом уровне.

Например, если идентификатор K встречается в области C, но его определения там нет, надо взять его определение в B (или D, если допускаются объявления после использования). Если же определения K нет ни в B, ни в D, надо взять его в областях A, E или G. Но нельзя искать определение K в области F.

Метод „уровень — индекс" работы с определениями годится для языков с принципом строгой вложенности областей действия определений (например, для Алгола и ПЛ/I). В этом разделе, однако, мы изучим более общий формализм, называемый грамматиками свойств, допускающий произвольные соглашения об области действия определений. Грамматикам свойств присущи общность и элегантность, вытекающие из единообразной трактовки идентификаторов и их свойств. Кроме того, обработку с их помощью можно осуществить за время, линейное относительно длины входного потока. Константа пропорциональности может быть велика, но мы излагаем эту концепцию в надежде на то, что будущие исследования сделают использование грамматик свойств практически приемлемым.

10.2.2. Определение грамматики свойств

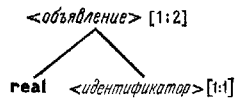
Неформально грамматику свойств можно определить как входную КС-грамматику, нетерминальным и терминальным символам которой приписываются „таблицы свойств". Таблицу свойств можно рассматривать как абстракцию таблицы имен. При разборе снизу вверх в соответствии с входной грамматикой таблицы свойств, приписанные нетерминалам, представляют в данной точке разбора информацию, хранящуюся обычно в таблице имен.

Таблица свойств T — это отображение множества индексов I в множество свойств V. Множеством индексов у нас будет множество неотрицательных целых чисел. Каждое целое число в множестве индексов можно интерпретировать как указатель на таблицу имен. Таким образом, если указываемый элемент таблицы имен представляет идентификатор, то целое можно трактовать как имя этого идентификатора.

Множество V — это множество „свойств", или „значений", и им у нас будет конечное множество целых чисел. Одно из них (обычно 0) выделяется как „нейтральное" свойство. Остальные можно связать с различными свойствами, представляющими интерес. Например, число 1 можно связать со свойством „на этот идентификатор была ссылка", 2 — со свойством „объявленное вещественное" и т. д.

В таблицах, связанных с терминалами, все индексы, кроме одного, отображаются в нейтральное свойство. Этот один индекс может отображаться в любое свойство (в том числе и нейтральное). Однако если терминалом является лексема ⟨идентификатор⟩, то индекс, представляющий имя конкретной лексемы (а именно, компонента данных этой лексемы), должен отображаться на свойство типа „это идентификатор, упоминаемый здесь".

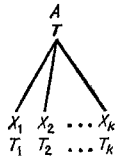
Например, если в программе встретилось объявление **real** B, то эту цепочку можно разобрать так:



где компонента данных лексемы ⟨идентификатор⟩ ссылается на цепочку B. С терминалом ⟨идентификатор⟩ связана таблица [1:1], отображающая индекс 1 (который соответствует теперь цепочке B) в свойство 1 (возможно, „упоминаемый"), а остальные индексы — в нейтральное свойство. Терминальная лексема **real** связана с таблицей, отображающей все индексы в нейтральное свойство. Обычно такая таблица явно не представлена.

С нетерминалом ‹объявление› в дереве разброса связывается таблица, получаемая слиянием по некоторому правилу таблиц прямых потомков этого нетерминала. В приведенном примере это таблица [1:2], отображающая индекс 1 в свойство 2, а все другие индексы — в нейтральное свойство. Тогда ее можно трактовать так, что идентификатор, связанный с индексом 1 (а именно, B), обладает свойством „объявленное вещественное".

Вообще говоря, если в дереве разбора есть структура



то свойство индекса i в таблице T является функцией только свойства индекса i в таблицах $T_1, T_2, \ldots, T_k$. Иными словами, каждый индекс трактуется независимо от всех остальных.

Дадим теперь формальное определение грамматики свойств.

Определение. *Грамматикой свойств* называется восьмерка $G = (N, \Sigma, P, S, V, v_0, F, \mu)$, где

(1) (N, Σ, P, S) — *входная* КС-грамматика,
(2) V — конечное множество *свойств*,
(3) $v_0 \in V$ — *нейтральное свойство*,
(4) $F \subseteq V$ — множество *допустимых свойств*, v_0 всегда принадлежит F,
(5) μ — такое отображение из $P \times V^*$ в V, что
(а) если значение $\mu(p, s)$ определено, то длина правой части правила вывода p та же, что и у s, где s — цепочка свойств;
(б) значение $\mu(p, v_0 v_0 \ldots v_0)$ равно v_0, если длина цепочки из нейтральных свойств v_0 та же, что и у правой части правила вывода p, и не определено в противном случае.

Функция μ говорит о том, как выбирать свойства в таблицах, связанных с внутренними вершинами дерева разбора. В зависимости от правила вывода, использованного в этой вершине, свойство, связанное с каждым целым числом, вычисляется независимо от свойств всех других целых чисел. Условие (5б) констатирует, что таблица в некоторой внутренней вершине может обладать не нейтральным свойством для некоторого целого числа, только если один из ее прямых потомков обладает не нейтральным свойством для этого числа.

Выводимой цепочкой в G называется цепочка $X_1T_1X_2T_2 \ldots X_nT_n$, где символы X принадлежат $N \cup \Sigma$, а T — таблицы свойств. Предполагается, что каждая таблица связана с символом непосредственно слева от нее и представляет собой такое отображение множества индексов в V, что все индексы, кроме конечного числа, отображаются в v_0.

Определим на выводимых цепочках отношение $\Rightarrow_G$ (или $\Rightarrow$, если ясно, о какой грамматике идеть речь). Пусть $\alpha AT\beta$ — выводимая цепочка в G и $A \to X_1 \ldots X_k$ — правило вывода $p \in P$. Тогда $\alpha AT\beta \Rightarrow \alpha X_1T_1 \ldots X_kT_k\beta$, если для всех индексов i

$$\mu(p, T_1(i)\, T_2(i) \ldots T_k(i)) = T(i)$$

Обозначим через $\Rightarrow_G^*$ (или $\Rightarrow^*$, если G подразумевается) рефлексивное и транзитивное замыкание отношения $\Rightarrow_G$. Языком $L(G)$, определяемым грамматикой G, называется множество всех таких цепочек $a_1T_1a_2T_2 \ldots a_nT_n$, что для некоторой таблицы T

(1) $ST \Rightarrow^* a_1T_1a_2T_2 \ldots a_nT_n$,
(2) $a_j \in \Sigma$ для всех j,
(3) $T(i) \in F$ для всех i,
(4) T_j для каждого j отображает все индексы, кроме, быть может, одного, в v_0.

Следует заметить, что хотя определение вывода дается сверху вниз, определение грамматики свойств скорее основано на разборе снизу вверх. Если можно определить таблицы, связанные с терминалами, то можно детерминированно построить таблицы, связанные с каждой вершиной дерева разбора, так как μ — функция таблиц, связанных с прямыми потомками вершины.

Должно быть ясно, что если G — грамматика свойств, то множество $\{a_1a_2 \ldots a_n \mid a_1T_1a_2T_2 \ldots a_nT_n \in L(G)$ для некоторой последовательности таблиц $T_1, T_2, \ldots, T_n\}$ является контекстно-свободным языком, поскольку любую цепочку, выводимую во входной КС-грамматике, можно вывести и в грамматике свойств, считая при этом все таблицы нейтральными.

Соглашение. Мы и дальше будем придерживаться принятого соглашения относительно обозначений для контекстно-свободных грамматик, по которому $a, b, c, \ldots$ принадлежат Σ и т. д. Кроме того, будем считать, что символ v обозначает свойство, а символ s — цепочку свойств. Если T отображает все целые числа в нейтральное свойство, то вместо XT будем писать X.

Пример 10.9. Приведем довольно длинный пример применения грамматик свойств для обработки объявлений в языке с блочной структурой. Покажем, как в ходе разбора можно детер-

минимально строить таблицы, если входная КС-грамматика допускает детерминированный разбор снизу вверх [1]).

Пусть $G=(N, \Sigma, P, S, V, 0, \{0\}, \mu)$ — грамматика свойств, в которой

(i) $N=\{$⟨блок⟩, ⟨оператор⟩, ⟨список объявлений⟩, ⟨список операторов⟩, ⟨список переменных⟩$\}$. Нетерминал ⟨список переменных⟩ вырабатывает список переменных, используемых в операторе. Оператор мы представим только переменными, действительно используемыми в операторе, а не задаиием его полиой структуры. Эта абстракция позволяет выделять основные особенности грамматик свойств, ие загромождая пример подробностями.

(ii) $\Sigma=\{$**begin, end, declare, label, goto,** $a\}$. Терминал **declare** служит для объявления одиого идеитификатора. Объявленные идентификаторы ие изображаются, поскольку эта информация будет в таблице свойств, связанной с **declare**. Аналогично **label** устаиавливает, что идентификатор употребляется как метка оператора. Сам идентификатор явно не изображается. Терминал **goto** заменяет **goto** ⟨метка⟩, но метка снова явно не изображается, поскольку она будет в таблице свойств, связанной с **goto**. Терминал a представляет переменную.

(iii) P состоит из следующих правил вывода:

(1) ⟨блок⟩ → **begin** ⟨список объявлеиий⟩ ⟨список операторов⟩ **end**
(2) ⟨список операторов⟩ → ⟨список операторов⟩ ⟨оператор⟩
(3) ⟨список операторов⟩ → ⟨оператор⟩
(4) ⟨оператор⟩ → ⟨блок⟩
(5) ⟨оператор⟩ → **label** ⟨список переменных⟩
(6) ⟨оператор⟩ → ⟨список переменных⟩
(7) ⟨оператор⟩ → **goto**
(8) ⟨список переменных⟩ → ⟨список переменных⟩ a
(9) ⟨список перемеиных⟩ → e
(10) ⟨список объявлений⟩ → **declare** ⟨список объявлений⟩
(11) ⟨список объявлеиий⟩ → e

Неформально правило (1) говорит, что блок — это список объявлений и список операторов, охваченные **begin** и **end**. Правило (4) говорит, что оператором может быть блок, (5) и (6) — что оператор — это список использованных внутри него переменных, перед которым, возможно, стоит метка. Правило (7) говорит, что оператором может быть оператор перехода, (8) и (9) — что список перемеиных — это цепочка из 0 или более терминалов a, а (10) и (11) — что список объявлений — это цепочка из 0 или более терминалов **declare**.

[1]) Грамматика этого примера неоднозначна, но это не мешает иллюстрировать основные понятия.

(iv) $V=\{0, 1, 2, 3, 4\}$ — миожество свойств, имеющих такой смысл:

0 — идеитификатор не появляется в цепочке, выводимой из данной вершиы (нейтральное свойство),
1 — идентификатор объявлен как переменная,
2 — идентификатор является меткой оператора,
3 — идентификатор используется как перемениая, но (поскольку, поскольку речь идет о потомках рассматриваемой вершины) еще не объявлен,
4 — идентификатор используется как метка в операторе перехода, но еще не было объявления этой метки.

(v) Определим на свойствах функцию μ, преследуя такие цели:

(а) если обнаружеио иедопустимое использование переменной или метки, то нельзя строить таблицы дальше, так что процесс вычислеиия таблиц „зайдет в тупик" (можно было бы для этой цели ввести свойство „ошибка"),
(б) идентификатор, использованный в операторе **goto**, должен быть меткой некоторого оператора того блока, в котором ои используется [1]),
(в) идентификатор, использованный как переменная, должен быть объявлен внутри этого же блока или блока, в который данный блок вложен.

Зададим значеиие $\mu(p, w)$ для каждого правила вывода по порядку и прокомментируем наши действия.

(1) ⟨блок⟩ → **begin** ⟨список объявлений⟩ ⟨список операторов⟩ **end**

s	$\mu(1, s)$
0 0 0 0	0
0 1 0 0	0
0 1 3 0	0
0 0 3 0	3
0 0 2 0	0

Единственным возможным свойством для всех целых чисел, связанных с **begin** и **end**, является 0; этим и объясняются два столбца нулей. В списке объявлений каждый идентификатор будет иметь свойство 0 (не объявлен) или 1 (объявлен). Если

[1]) Это отличается от соглашений Алгола. Например, Алгол допускает переходы в блок, охватывающий данный. Мы принимаем наше соглашение потому, что хотим, чтобы обработка меток отличалась от обработки идентификаторов.

идентификатор объявлен, то внутри тела блока, т. е. в списке операторов, он либо может использоваться только как переменная (3), либо не использоваться (0). В любом случае идентификатор не объявлен, коль скоро это касается программы вне данного блока, так что идентификатору придается свойство 0. Так появляются строки вторая и третья.

Если идентификатор не объявлен в данном блоке, он все-таки может использоваться либо как метка, либо как переменная. Если он используется как переменная (свойство 3), то это надо передать за пределы блока, чтобы можно было проверить, объявлен ли он в соответствующем месте (строка 4). Если идентификатор определен как метка внутри данного блока, то это не передается за пределы данного блока (строка 5), поскольку на метку внутри данного блока нельзя перейти извне его.

Поскольку функция μ не определена для остальных значений s, грамматика свойств вылавливает использование меток, не найденных внутри данного блока, а также и использование объявленных переменных в качестве меток внутри данного блока. Метка, использованная как переменная, будет выловлена в другом месте.

(2) ⟨список операторов⟩ → ⟨список операторов⟩ ⟨оператор⟩

s		μ (2, s)
0	0	0
3	3	3
0	3	3
3	0	3
4	4	4
0	4	4
4	0	4
4	2	2
2	4	2
0	2	2
2	0	2

Строки 2—4 говорят, что идентификатор, использованный как переменная в ⟨списке операторов⟩ или в ⟨операторе⟩ в правой части правила вывода, используется как переменная, коль скоро это касается ⟨списка операторов⟩ слева. Строки 5—7 говорят то же самое о метках. Строки 8—11 говорят, что метка, определенная либо в ⟨списке операторов⟩, либо в ⟨операторе⟩ в правой части, определена для ⟨списка операторов⟩ слева, независимо от того, использовалась метка или нет.

В этом месте вылавливается использование идентификатора в качестве переменной и в качестве метки внутри одного блока

(3) ⟨список операторов⟩ → ⟨оператор⟩

s	μ (3, s)
0	0
2	2
3	3
4	4

Свойства передаются естественным образом. Свойство 1 для оператора невозможно.

(4) ⟨оператор⟩ → ⟨блок⟩

s	μ (4, s)
0	0
3	3

Все рассуждения, относящиеся к правилу (3), применимы также и к правилу (4).

(5) ⟨оператор⟩ → **label** ⟨список переменных⟩

s		μ (5, s)
0	0	0
0	3	3
2	0	2

Использование идентификатора в качестве метки в **label** или переменной в ⟨списке переменных⟩ передается в ⟨оператор⟩ в левой части.

(6) ⟨оператор⟩ → ⟨список переменных⟩

s	μ (6, s)
0	0
3	3

Здесь использование переменной в ⟨списке переменных⟩ передается в ⟨оператор⟩.

(7) ⟨оператор⟩ → **goto**

s	μ (7, s)
0	0
4	4

Использование метки в **goto** передается в ⟨оператор⟩.

(8) ⟨список переменных⟩ ⟶ ⟨список переменных⟩ a

s		$\mu(8, s)$
0	0	0
3	0	3
0	3	3
3	3	3

Любое использование переменной передается в ⟨список переменных⟩.

(9) ⟨список переменных⟩ ⟶ e

s	$\mu(9, s)$
e	0

Единственным значением, для которого функция μ определена, является $s=e$. Свойством должно быть число 0 по определению нейтрального свойства.

(10) ⟨список объявлений⟩ ⟶ **declare** ⟨список объявлений⟩

s		$\mu(10, s)$
0	0	0
0	1	1
1	0	1
1	1	1

Объявленные идентификаторы передаются в ⟨список объявлений⟩.

(11) ⟨список объявлений⟩ ⟶ e

s	$\mu(11, s)$
e	0

Здесь та же ситуация, что и в правиле (9).

Теперь приведем пример построения таблиц снизу вверх на дереве разбора. Таблицу, которая индексу i_j присваивает свойство v_j для $1 \leqslant j \leqslant n$, а всем остальным индексам — нейтральное свойство, будем обозначать $[i_1{:}v_1, i_2{:}v_2, \ldots, i_n{:}v_n]$.

Рассмотрим входную цепочку

```
begin
    declare [1:1]
    declare [2:1]
        begin
            label [1:2] a[2:3]
            goto [1:4]
        end
    a[1:3]
end
```

Таким образом, во внешнем блоке идентификаторы 1 и 2 объявлены посредством символов **declare** [1:1] и **declare** [2:1] как переменные. Затем во внутреннем блоке идентификатор 1 посредством символов **label** [1:2] и **goto** [1:4] соответственно объявлен и используется как метка (что законно), а идентификатор 2 посредством символа a[2:3] используется как переменная.

На рис. 10.9 приведено дерево разбора с таблицами, связанными с каждой вершиной. □

Понятие грамматики свойств можно обобщить. Например,

(1) отображение μ можно сделать недетерминированным, т. е. допустить, что $\mu(p, w)$ — подмножество множества V,

(2) можно не предполагать наличия нейтрального свойства,

(3) можно наложить ограничения на класс таблиц, связанных с символами; например, можно не допускать наличия полностью нейтральных таблиц.

Некоторые теоремы о грамматиках свойств в этой более общей формулировке изложены в упражнениях.

10.2.3. Реализация грамматик свойств

Рассмотрим реализацию грамматики свойств для случая, когда входная КС-грамматика допускает детерминированный восходящий разбор с помощью алгоритма типа „перенос — свертка". Нисходящий разбор или применение других алгоритмов разбора, изложенных в этой книге, приводит к тем же проблемам согласования при конструировании таблиц, что и при построении цепочек переводов в СУ-переводе. Поскольку их решение в основном то же, что и в гл. 9, мы рассмотрим только восходящий разбор.

В нашей модели компилятора вход для лексического анализатора не имеет таблиц, связанных с его терминалами. Будем предполагать, что лексический анализатор сигнализирует о том, является ли выбранная лексема идентификатором или

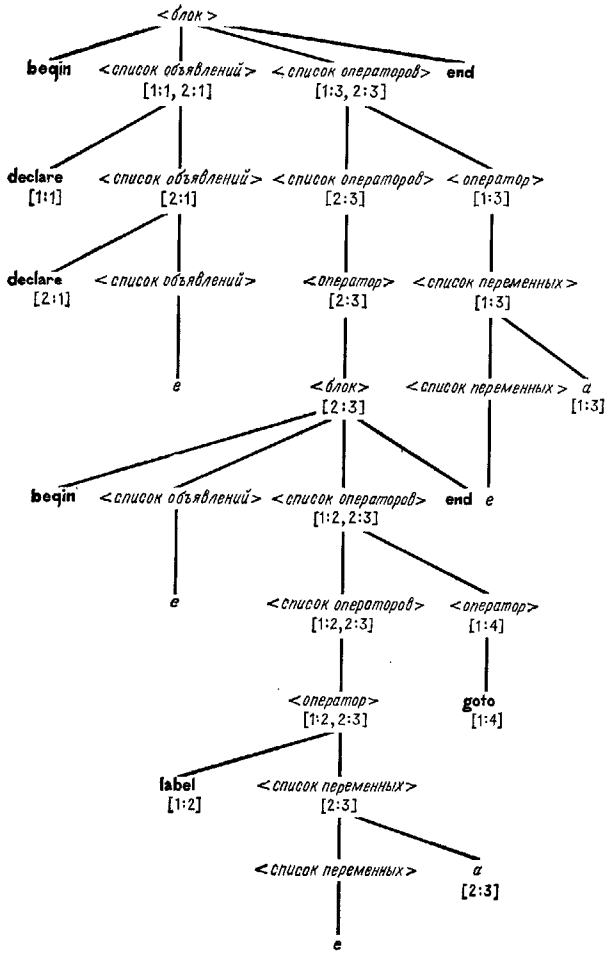


Рис. 10.9. Дерево разбора с таблицами.

нет. Таким образом, каждая лексема, будучи входом при разборе, имеет одну из двух таблиц — либо полностью нейтральную, либо таблицу, в которой один индекс обладает не нейтральным свойством. Поэтому мы будем считать, что еще необработанный вход вообще не имеет таблиц, и они строятся, когда символ передается в магазин.

Будем также предполагать, что таблицы не влияют на разбор, не считая случая ошибки, когда разбор надо прервать. Таким образом, механизмом разбора будет нормальный механизм типа „перенос — свертка", причем к каждому символу магазина будет добавляться указатель на таблицу свойств этого элемента.

Пусть в верхушке магазина находится $[B, T_1][C, T_2]$ и выполняется свертка в соответствии с правилом $A \to BC$. Наша задача заключается в том, чтобы быстро и просто по T_1 и T_2 построить таблицу, связанную с A. Поскольку большинство индексов в таблице отображается в нейтральное свойство, желательно помещать в нее только те индексы, которые обладают не нейтральным свойством.

Схема обработки таблиц будет реализована так, чтобы все операции по обработке таблиц и запросы о свойстве данного индекса можно было выполнить за время, фактически пропорциональное числу операций и запросов. Естественно предположить, что число запросов в таблицы пропорционально длине входа. Предположение корректно, если разбор детерминированный, поскольку число выполняемых сверток (а следовательно, и число слияний таблиц) в этом случае пропорционально длине входа. В то же время разумный алгоритм перевода не должен осуществлять запросы свойств более чем фиксированное число раз на одну свертку.

Далее мы будем считать, что в нашей грамматике свойств входная КС-грамматика имеет нормальную форму Хомского. Обобщения алгоритма вынесены в упражнения. Будем предполагать, что в любой таблице свойств каждый индекс (идентификатор), обладающий не нейтральным свойством, занимает позицию в таблице расстановки, и что можно создавать структуры данных из элементарных единиц, называемых *ячейками*. Каждая ячейка имеет форму

ДАННОЕ1	...	ДАННОЕm	...	УКАЗАТЕЛЬ1	...	УКАЗАТЕЛЬn

т. е. состоит из одного или более полей, каждое из которых может содержать некоторые данные или указатель на другую ячейку. Ячейки понадобятся при построении связных списков.

Предположим, что V состоит из k свойств. Таблица свойств, связанная с символом грамматики, расположенным в магазине, представляется структурой данных, состоящей не более чем из k *списков свойств* и из *списка пересечений*. Каждый список свойств начинается с *заголовка списка свойств*. Список пересечений начинается с *заголовка списка пересечений*. Эти заголовки связаны так, как это показано на рис. 10.10.

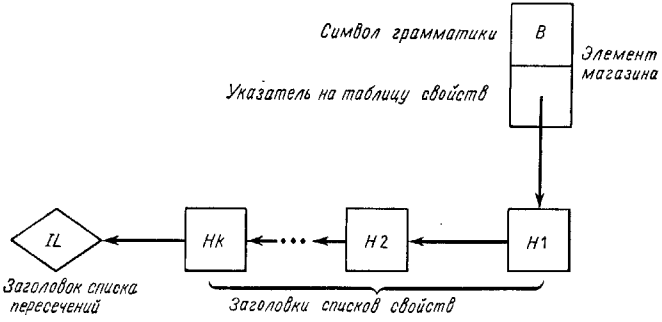


Рис. 10.10. Элемент магазина со структурой, представляющей таблицу свойств.

Заголовок списка свойств имеет три поля:

СВОЙСТВО	РАЗМЕР	СЛЕДУЮЩИЙ ЗАГОЛОВОК

Заголовок списка пересечений содержит только указатель на первую ячейку списка пересечений. Каждая ячейка списков свойств и списков пересечений называется *индексной ячейкой*. Индексная ячейка состоит из четырех полей с указателями:

НА СВОЙСТВО	НА СПИСОК ПЕРЕСЕЧЕНИЙ	К ТАБЛИЦЕ РАССТАНОВКИ	ОТ ТАБЛИЦЫ РАССТАНОВКИ

Предположим, что T — таблица свойств, связанная с символом в магазине. Тогда T будет представляться p списками свойств, где p — число различных свойств в T, и одним списком пересечений. Для каждого индекса в T, обладающего не нейтральным свойством j, есть одна индексная ячейка в списке свойств, начинающемся с заголовка списка свойств для свойства j.

Индексные ячейки, обладающие одинаковым свойством, объединены в дерево [1]), корнем которого служит заголовок для этого свойства. Первый указатель в индексной ячейке — это указатель на ее прямого предка в этом дереве.

Если индексная ячейка принадлежит списку пересечений, то вторым указателем в ней является указатель на следующую ячейку списка пересечений. Если ячейка не принадлежит списку пересечений, то этот указатель пуст.

Два остальных указателя связывают индексные ячейки, представляющие один и тот же индекс, но в различных таблицах свойств. Предположим, что $\delta T_3 \gamma T_2 \beta T_1 \alpha$ представляет такую цепочку таблиц в магазине, что каждая из таблиц T_1, T_2 и T_3 содержит индекс i с не нейтральным свойством и все таблицы в α, β, γ и δ дают индексу i нейтральное свойство.

Если ближе всех к верхушке магазина расположена таблица T_1, то индексная ячейка C_1, представляющая i в T_1, будет содержать в третьем поле указатель на позицию таблицы имен для i. Четвертое поле в C_1 содержит указатель на ячейку C_2 в T_2, представляющую i. В C_2 третье поле содержит указатель на C_1, а четвертое поле содержит указатель на ячейку в T_3, представляющую i.

Таким образом, на индексные ячейки налагается дополнительная структура: все индексные ячейки, представляющие один и тот же индекс во всех таблицах в магазине, помещаются в двунаправленный список, начинающийся с элемента таблицы расстановки для этого индекса.

Ячейка размещается в списке пересечений таблиц тогда и только тогда, когда какая-нибудь таблица, расположенная над ней в магазине, имеет индексную ячейку, представляющую тот же индекс. В приведенном выше примере ячейка C_2 будет в списке пересечений для T_2.

Пример 10.10. Предположим, что в магазине находятся символы грамматики B и C, причем C — в верхушке магазина. Пусть с этими элементамм связаны соответственно таблицы

$$T_1 = [1:v_1,\ 2:v_3,\ 5:v_2,\ 6:v_1,\ 8:v_2]$$
$$T_2 = [2:v_1,\ 3:v_1,\ 4:v_1,\ 5:v_1,\ 7:v_2,\ 8:v_3]$$

Возможная реализация этих таблиц показана на рис. 10.11. Кружочками указаны индексные ячейки. Число внутри кружочка — это индекс, представляемый ячейкой. Связи списка пересечений указаны пунктиром. Отметим, что список пересечений самой верхней таблицы пуст по определению, а список пересечений таблицы T_1 состоит из индексных ячеек для индексов 2,

[1]) Структура этого дерева будет ясна из алгоритма 10.3. — *Прим. перев.*

5 и 8. Ссылки на таблицу расстановки и на ячейки, представляющие один и тот же индекс в других таблицах, указаны штриховыми линиями. Во избежание путаницы они показаны только для индексов 2 и 3. □

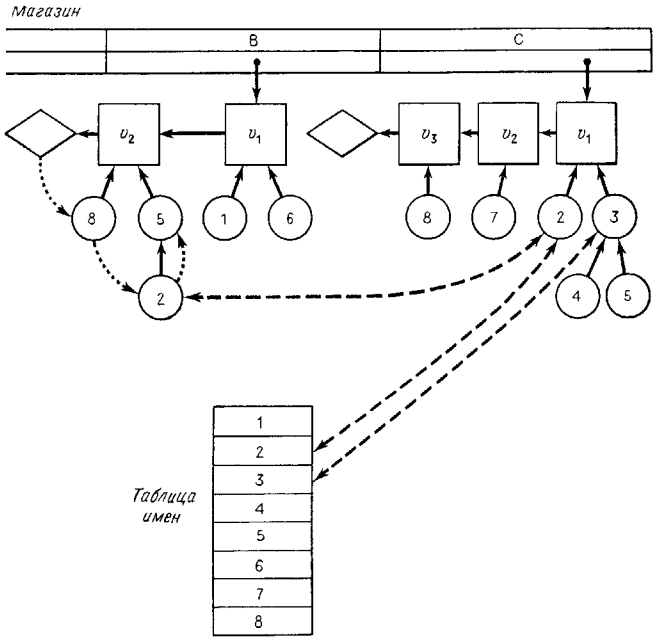


Рис. 10.11. Реализация таблиц свойств.

Предположим, что в процессе разбора надо в магазине свернуть BC в A. Надо вычислить таблицу T для A по таблицам T_1 и T_3 для B и C соответственно. Поскольку C — верхний символ магазина, список пересечений таблицы T_1 содержит в точности те индексы, которые обладают не нейтральным свойством в обеих таблицах T_1 и T_2 (отсюда и название „список пересечений"). Эти индексы мы рассмотрим позднее.

Индексы, не принадлежащие списку пересечений T_1, обладают нейтральным свойством по крайней мере в одной из таблиц T_1 или T_2. Таким образом, их свойство в T есть функция только

их одного не нейтрального свойства. Если отвлечься от индексов из списка пересечений, то структуру данных, представляющую таблицу T, можно построить, комбинируя различные деревья из T_1 и T_2. После того как это сделано, каждый элемент списка пересечений для T_1 рассматривается отдельно, и из него делается ссылка на соответствующую ячейку таблицы T.

Прежде чем формализовать эти идеи, отметим, что хотелось бы, чтобы на практике свойства можно было разбить на непересекающиеся подмножества так, чтобы V можно было представить в виде $V_1 \times V_2 \times \ldots \times V_m$ для некоторого относительно большого m. Различные компоненты $V_1, V_2, \ldots$ были бы, вообще говоря, небольшими. Например, множество V_1 могло бы содержать два элемента, обозначающие „вещественное" и „целое"; V_2 — два элемента, обозначающие „одинарная точность" и „двойная точность"; V_3 — „динамически размещаемые" и „статически размещаемые" и т. д. В каждом из множеств $V_1, V_2, \ldots$ один элемент можно рассматривать как условие отсутствия остальных, а произведение таких элементов отсутствия — как нейтральное свойство. Наконец, можно ожидать, что различные компоненты свойств идентификатора могут определяться независимо от остальных.

Если происходит именно так, то можно создавать один заголовок свойств для каждого элемента из V_1, кроме элемента отсутствия, один для каждого элемента из V_2, кроме элемента отсутствия, и т. д. Каждая индексная ячейка связана с несколькими заголовками свойств, но не более чем с одним из каждого множества V_i. Если ссылки на заголовки для V_i отличаются от ссылок на заголовки для V_j при $i \neq j$, то идеи этого раздела можно с тем же успехом применить и к этой ситуации; общее число заголовков свойств будет близко к сумме мощностей множеств V_i, а не к их произведению.

Дадим формальный алгоритм реализации грамматики свойств. Для простоты изложения ограничимся грамматиками свойств с входной КС-грамматикой в нормальной форме Хомского.

Алгоритм 10.3. Обработка таблиц при реализации грамматик свойств.

Вход. Грамматика свойств $G = (N, \Sigma, P, S, V, v_0, F, \mu)$ с входной КС-грамматикой в нормальной форме Хомского. Будем предполагать, что не нейтральное свойство никогда не отображается в нейтральное, т. е. если $\mu(p, v_1v_2) = v_0$, то $v_1 = v_2 = v_0$. (Это условие приемлемо, поскольку если $\mu(i, v_1v_2) = v_0$, но v_1 или v_2 не равно v_0, то в правой части можно заменить v_0 на новое не нейтральное свойство v_0' и ввести правила, которые делали бы v_0' „похожим" на v_0.) Входом для данного алгоритма служит

также алгоритм разбора типа „перенос—свертка“ для входной КС-грамматики.

Выход. Модифицированный алгоритм разбора типа „перенос — свертка“ для входной КС-грамматики, вычисляющий в процессе разбора таблицы, связанные с теми вершинами дерева разбора, которые соответствуют символам магазина.

Метод. Предположим, что каждая таблица имеет формат, как на рис. 10.10, т. е. не более одного списка пересечений и списка заголовков для каждого свойства, а в ее дереве индексов с каждым заголовком связано данное свойство. Описание работы механизма построения таблиц мы разобьем на две части в зависимости от того, свертывается один терминал или два нетерминала. (Напомним, что входная грамматика задана в нормальной форме Хомского.)

Часть 1: Предположим, что терминальный символ a переносится в магазин и свертывается в нетерминал A. Пусть таблицей, требуемой для A, будет $[i:v]$. Для того чтобы выполнить эту операцию, поместим A непосредственно в магазин и следующим образом выработаем таблицу $[i:v]$ для A.

(1) При элементе A в верхушке магазина поместим указатель на единственный заголовок свойства, обладающий свойством v и размером 1. Этот заголовок свойства указывает на заголовок списка пересечений с пустым списком пересечений.

(2) Создадим индексную ячейку C для i.

(3) В первом поле ячейки C поместим указатель на заголовок свойства.

(4) Второе поле ячейки C будет пустым.

(5) В третье поле ячейки C поместим указатель на элемент таблицы расстановки для i и в этот элемент таблицы расстановки поместим указатель на C.

(6) Если другая ячейка C' уже была связана с элементом таблицы расстановки для i, поместим C' в список пересечений таблицы, в которую она входит. (Для этого в заголовок списка пересечений поместим указатель на C', а во второе поле ячейки C' поместим указатель на прежнюю первую ячейку списка пересечений, если она была.)

(7) В четвертое поле ячейки C поместим указатель на C'.

(8) В третье поле ячейки C' поместим указатель на C.

Часть 2: Предположим теперь, что два нетерминала свертываются в один, скажем, по правилу $A \to BD$. Пусть таблицами, связанными с B и D, будут T_1 и T_2 соответственно. Тогда для вычисления таблицы T для A сделаем следующее.

(1) Рассмотрим каждую индексную ячейку в списке пересечений для T_1. (Напомним, что у T_2 нет списка пересечений.) Каждая такая ячейка дает элемент для некоторого индекса i, который есть как в T_1, так и в T_2. Свойства данного индекса

в этих таблицах можно найти с помощью алгоритма 10.4[1]). Пусть этими свойствами будут v_1 и v_2. Вычисляем $v = \mu(p, v_1v_2)$, где p — правило вывода $A \to BD$. Строим список из всех индексных ячеек списка пересечений вместе с их новыми свойствами и старым содержимым ячеек таблиц T_1 и T_2.

(2) Рассмотрим заголовки свойств таблицы T_1. Изменим свойство заголовка со свойством v на свойство $\mu(p, vv_0)$. Таким образом, мы предполагаем, что все индексы со свойством v в T_1 обладают нейтральным свойством в T_2.

(3) Рассмотрим заголовки свойств в T_2. Изменим свойство заголовка со свойством v на $\mu(p, v_0v)$. Таким образом, мы предполагаем, что все индексы со свойством v в T_2 обладают нейтральным свойством в T_1.

(4) Теперь некоторые из заголовков свойств, ранее принадлежащие T_1 и T_2, могут обладать одинаковым свойством. Их слияние осуществляется в результате выполнения следующих шагов, объединяющих два дерева в одно:

(а) заменяем заголовок свойства с меньшим размером (произвольно нарушая связи) на фиктивную индексную ячейку, не соответствующую никакому индексу,

(б) помещаем в эту новую индексную ячейку указатель на заголовок свойства с большим размером,

(в) полагаем размер оставшегося заголовка равным сумме размеров этих двух заголовков плюс 1, так что оно отражает число индексных ячеек в дереве, включая фиктивные ячейки.

(5) Рассмотрим теперь список индексов, созданный на шаге (1). Для каждого такого индекса

(а) создаем новую индексную ячейку C,

(б) в первое поле ячейки C помещаем указатель на заголовок свойства с исправленным свойством, а скорректированное значение размера помещаем в этот заголовок,

(в) в третье поле ячейки C помещаем указатель на позицию таблицы расстановки для этого индекса, а из этой позиции таблицы расстановки — на C,

(г) в четвертое поле ячейки C помещаем указатель на первую индексную ячейку, лежащую под ней (в магазине) и имеющую тот же индекс,

(д) рассмотрим теперь две исходные ячейки C_1 и C_2, представляющие этот индекс в T_1 и T_2. Делаем C_1 и C_2 „фиктивными“, сохраняя указатели в их первых полях (ссылки

[1]) Очевидно, что свойство данного индекса можно найти, идя из ячеек, соответствующих ему в двух таблицах, по направлению к корням деревьев, на которых эти ячейки находятся. Однако, чтобы обработка таблиц в целом была линейной во времени, необходимо, чтобы движение к корню осуществлялось специальным образом. Этот метод мы опишем ниже в алгоритме 10.4.

на их предков в деревьях), но удаляя указатели в третьих и четвертых полях (их ссылки на таблицу расстановки и на ячейки в других таблицах, имеющие тот же индекс). Таким образом, вновь созданная индексная ячейка C играет роль двух ячеек C_1 и C_2, сделанных теперь фиктивными.

(б) Оставшиеся в результате фиктивные ячейки можно вернуть в свободную память. □

Пример 10.11. Исследуем две таблицы свойств из примера 10.10 (рис. 10.11). Пусть $\mu(p, st)$ задается табл. 10.2.

Таблица 10.2

s	t	$\mu(p, st)$
v_0	v_1	v_1
v_0	v_2	v_2
v_0	v_3	v_3
v_1	v_0	v_1
v_2	v_0	v_2
v_2	v_1	v_3
v_2	v_3	v_2

Согласно части 2 алгоритма 10.3, сначала надо рассмотреть список пересечений для T_1, состоящий из индексных ячеек 2, 5 и 8 (рис. 10.11). Из табл. 10.2 видно, что в новой таблице свойств T эти индексы будут обладать свойствами v_3, v_3 и v_2 соответственно.

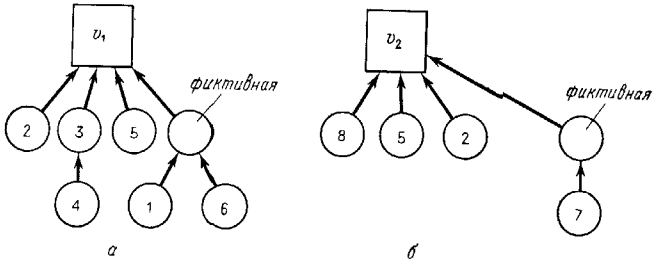


Рис. 10.12. Слившиеся деревья.

Затем рассмотрим заголовки свойств для T_1 и T_2. Свойства в заголовках остаются теми же, поскольку $\mu(p, v_0v_i) = v_i$ и $\mu(p, v_iv_0) = v_i$. Теперь дерево для v_1 в T_1 „вливаем" в дерево для v_1 в T_2, так как последнее больше. Полученное дерево изображено на рис. 10.12, *а*. Затем дерево для v_2 в T_2 „вливаем" в

дерево для v_2 в T_1 (рис. 10.12, *б*). Заметим, что на рис. 10.12, *а* вершина 5 стала прямым потомком заголовка, в то время как на рис. 10.11 она была прямым потомком вершины с номером 3. Это происходит при анализе списка пересечений для T_1 в алгоритме 10.4. Вершина 2 на рис. 10.12, *б* переместилась по той же причине.

В качестве последнего шага исследуем индексы в списке пересечений для T_1. Для этих индексов создаются новые индексные ячейки; новые ячейки указывают прямо на соответствующий

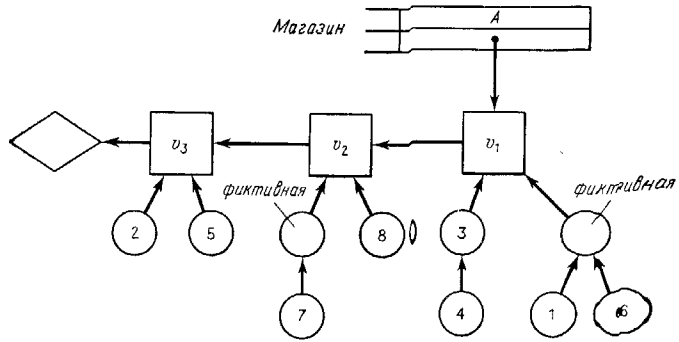


Рис. 10.13. Новая таблица T.

заголовок. Все остальные ячейки для этого индекса в таблице T делаются фиктивными. Затем удаляются фиктивные ячейки без потомков. Полученная таблица T показана на рис. 10.13. Таблица имен не приведена. Отметим, что список пересечений для T пуст.

Предположим теперь, что входной символ переносится в магазин и свертывается в $D[2:v_1]$. Тогда индексная ячейка для 2, которая на рис. 10.13 указывает на v_3, включается в список пересечений своей таблицы. Изменения отражены на рис. 10.14. □

Дадим теперь алгоритм, с помощью которого из таблицы T выбирается свойство индекса i. Этот алгоритм используется в алгоритме 10.3 для нахождения свойств индексов в списке пересечений.

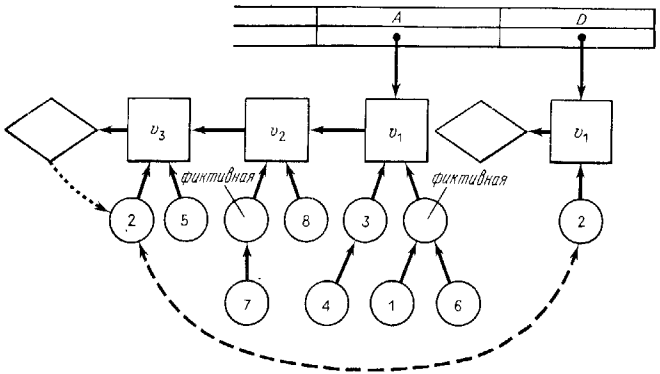


Рис 10.14. Таблицы после переноса.

Алгоритм 10.4. Нахождение свойства индекса.

Вход. Индексная ячейка некоторой таблицы T. Предполагаем, что таблицы имеют ту же структуру, что и в алгоритме 10.3.

Выход. Свойство этого индекса в таблице T.

Метод.

(1) Проходим по указателям из данной индексной ячейки до корня дерева, в которое она входит. Формируем список всех ячеек, встретившихся на этом пути.

(2) Помещаем в каждую ячейку на этом пути, кроме корня, ссылку прямо на этот корень. (Конечно, ячейка на этом пути, непосредственно предшествующая корню, уже ссылается на него). □

Отметим, что наиболее важен шаг 2 алгоритма 10.4, являющийся по существу „побочным эффектом" алгоритма. Именно шаг (2) гарантирует, что общее время, потраченное на обработку таблицы, в целом будет пропорционально длине входа.

10.2.4. Анализ алгоритма обработки таблиц

Оставшаяся часть этой главы посвящена анализу временнóй сложности алгоритмов 10.3 и 10.4. Определим сначала две функции F и G, которые нам понадобятся в этой главе.

Определение. $F(n)$ определим рекуррентно:

$$F(1) = 1$$
$$F(n) = 2^{F(n-1)}$$

Некоторые значения $F(n)$ приведены в табл. 10.3.

Таблица 10.3

n	$F(n)$
1	1
2	2
3	4
4	16
5	65536

Определим $G(n)$ как наименьшее из целых чисел i, для которых $F(i) \geqslant n$. Функция G растет так медленно, что реально $G(n) \leqslant 6$ для всех n, представимых одним машинным словом, даже с плавающей точкой. С другой стороны, $G(n)$ можно определить как количество применений $\log_2$ к n для того, чтобы получить число, меньшее или равное 0.

Оставшаяся часть главы посвящена доказательству того, что алгоритм 10.3 требует $O(nG(n))$ шагов вычислительной машины с произвольным доступом при длине входной цепочки, равной n. Покажем сначала, что, если не считать время, затрачиваемое на алгоритм 10.4, временнáя сложность алгоритма 10.3 составляет $O(n)$.

Лемма 10.3. *Если не считать обращений к алгоритму* 10.4, *то алгоритм* 10.3 *можно реализовать на вычислительной машине с произвольным доступом за время* $O(n)$, *где* n — *длина разбираемой входной цепочки.*

Доказательство. Каждое выполнение части 1 требует фиксированного количества времени, а таких выполнений в точности n.

Отметим, что шаги (1) и (5) в части 2 требуют времени, пропорционального длине списка пересечений. (Напомним еще раз, что время работы алгоритма 10.4 не учитывается.) Но единственный способ поиска по индексной ячейке пути в списке пересечений заключается в выполнении части 1. Каждое выполнение части 1 размещает не более одной индексной ячейки в списке пересечений. Таким образом, во все списки пересечений всех таблиц помещается не более n индексов. После выполнения части 2 все индексные ячейки удаляются из списка пересечений,

так что общее время, затрачиваемое на шаги (1) и (5) части 2, составляет $0(n)$.

Легко видеть, что остальные шаги части 2 требуют постоянного количества времени на одно выполнение части 2. Поскольку часть 2 выполняется в точности $n-1$ раз, можно заключить, что общее время, затрачиваемое алгоритмом 10.3, не считая вызовов алгоритма 10.4, составляет $0(n)$. □

Поставим теперь некоторую абстрактную задачу и дадим ее решение, отражающее идеи алгоритмов 10.3 и 10.4, в терминах, более абстрактных и легче поддающихся анализу.

Определение. Для оставшейся части главы определим *задачу слияния множеств* как

(1) набор *объектов* $a_1, \ldots, a_n$,

(2) набор *имен* множеств, включающий $A_1, A_2, \ldots, A_n$,

(3) последовательность команд $I_1, I_2, \ldots, I_m$, причем каждая команда I_i имеет форму

(а) **слить** (A, B, C) или

(б) **найти** (a),

где A, B и C — имена множеств, а a — объект.

(Имена множеств можно мыслить как пары, состоящие из таблицы и свойства, а объекты — как индексные ячейки.)

Команда **слить** (A, B, C) формирует объединение множеств A и B и получающемуся множеству дает имя C. При этом не генерируется никакого выхода.

Команда **найти** (a) печатает имя множества, элементом которого является a.

Вначале предполагается, что каждый объект a_i принадлежит множеству A_i, т. е. $A_i = \{a_i\}$. *Ответом последовательности команд* $I_1, I_2, \ldots, I_m$ называется последовательность выходов, генерируемая при последовательном выполнении команд.

Пример 10.12. Пусть заданы объекты $a_1, a_2, \ldots, a_6$ и последовательность команд

слить (A_1, A_2, A_2)
слить (A_3, A_4, A_4)
слить (A_5, A_6, A_6)
слить (A_2, A_4, A_4)
слить (A_4, A_6, A_6)
найти (a_3)

После выполнения первой команды множеством A_2 будет $\{a_1, a_2\}$. После второй команды $A_4 = \{a_3, a_4\}$. После третьей множеством A_6 станет $\{a_5, a_6\}$. Затем после команды **слить** (A_2, A_4, A_4) множеством A_4 станет $\{a_1, a_2, a_3, a_4\}$. После последней команды слияния

$A_6 = \{a_1, a_2, \ldots, a_6\}$. Затем команда **найти** (a_3) печатает имя A_6, являющееся ответом на эту последовательность команд. □

Теперь дадим алгоритм выполнения любой последовательности команд длины $0(n)$ за время $0(nG(n))$, где n — число объектов. Относительно способа доступа к объектам и множествам будут сделаны некоторые допущения. Сразу не очевидно, что реализация грамматики свойств алгоритмами 10.3 и 10.4 этим условиям удовлетворяет. Однако достаточно простых рассуждений, чтобы увидеть, что в действительности при необходимости определить свойства объектов (индексных ячеек) или слить множества (заголовки свойств) они (объекты и множества) будут доступны.

Алгоритм 10.5. Вычисление ответа на последовательность команд.

Вход. Набор объектов $\{a_1, \ldots, a_n\}$ и последовательность команд $I_1, \ldots, I_m$ описанного выше типа.

Выход. Ответ на последовательность $I_1, \ldots, I_m$.

Метод. Множество запоминается в виде дерева, каждая вершина которого представляет один элемент множества. Корень дерева имеет метку, дающую

(1) имя множества, представляемого деревом, и

(2) число вершин этого дерева (*размер*).

Предполагается, что за конечное число шагов можно найти вершину, представляющую объект, или корень дерева, представляющего множество. Это можно сделать, например, используя два таких вектора OBJECT и SET, что OBJECT (a) — указатель на вершину, представляющую a, а SET (A) — указатель на корень дерева, представляющего множество A.

Сначала строим n вершин, по одной для каждого объекта a_i. Вершина для a_i — это корень дерева с одной вершиной; корень помечен A_i и имеет размер 1.

(1) Для выполнения команды **слить** (A, B, C) определяем местоположение корней деревьев для A и B (с помощью SET (A) и SET (B)). Сравниваем размеры деревьев, указываемых с помощью A и B. Корень меньшего дерева становится прямым потомком корня большего дерева (связи нарушаются произвольно). Большему корню дается имя C, а его размером становится сумма размеров A и B[1]). Указатель на корень C помещается в ячейку SET (C).

[1]) Аналогия между этим шагом и процедурой слияния алгоритма 10.3 очевидна. Различие в способе подсчета заключается только в том, считается ли корень или нет. В данном случае считается, в алгоритме 10.3 нет.

(2) Для выполнения команды **найти** (a) определяем с помощью OBJECT (a) вершину, представляющую a. Затем проходим из нее в корень r этого дерева. Печатаем имя, найденное в r. Все вершины на этом пути, кроме r, делаем прямыми потомками r [1]). □

Пример 10.13. Рассмотрим последовательность команд из примера 10.12. После выполнения первых трех команд слияния

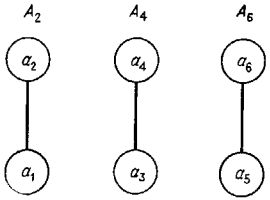


Рис. 10.15. Деревья после трех команд **слить**.

получаются три дерева, изображенные на рис. 10.15. Корни помечены именем множества и размером. (Размер не показан.) Затем, выполнив команду **слить** (A_2, A_4, A_4), приходим к структуре на рис. 10.16. После последней команды **слить** (A_4, A_6, A_6)

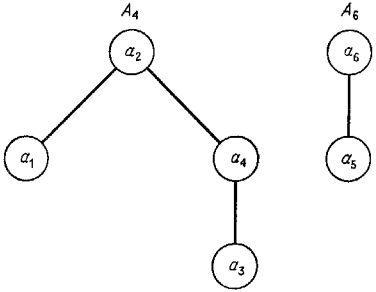


Рис. 10.16. Деревья после очередной команды **слить**.

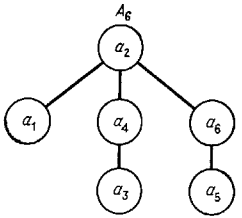


Рис. 10.17. Дерево после последней команды **слить**.

получаем рис. 10.17. Выполняя далее команду **найти** (a_3), печатаем имя A_6, а вершины a_3 и a_4 делаем прямыми потомками корня (a_4 уже является прямым потомком). Окончательная структура приведена на рис. 10.18. □

[1]) Аналогия с алгоритмом 10.4 должна быть очевидна.

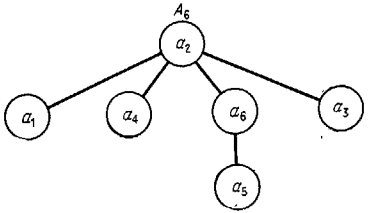


Рис. 10.18. Дерево после команды **найти**.

Покажем теперь, что алгоритм 10.5 может работать за время $O(nG(n))$ при разумном предположении, что выполнение команды **слить** требует одной единицы времени, а команды вида **найти** (a) — времени, пропорционального длине пути от вершины, представляющей a, до корня. Все последующие результаты выведены при этом предположении. Начиная отсюда, будем считать, что число объектов n фиксировано и последовательность команд имеет длину $O(n)$.

Определение. Определим *ранг* вершины в структуре, вырабатываемой алгоритмом 10.5:

(1) лист имеет ранг 0,

(2) если у вершины N когда-либо появляется прямой потомок ранга i, то N имеет ранг по крайней мере $i+1$,

(3) ранг вершины — это наименьшее целое число, удовлетворяющее условию (2).

Сразу не ясно, что это определение корректно. Однако если вершина M становится прямым потомком вершины N в алгоритме 10.5, то у M уже никогда больше не будет прямых потомков. Таким образом, в этот момент можно зафиксировать ранг M. Например, на рис. 10.17 ранг вершины a_6 можно зафиксировать равным 2, поскольку a_6 имеет одного прямого потомка ранга 1, а новых потомков уже больше иметь не будет.

Три следующие леммы устанавливают некоторые свойства ранга вершины.

Лемма 10.4. *Пусть N — корень ранга i, созданный алгоритмом 10.5. Тогда N имеет по крайней мере 2^i потомков.*

Доказательство. Базис, $i=0$, очевиден, поскольку вершина тривиально является собственным потомком. Для доказательства шага индукции предположим, что N — корень ранга i. Тогда в некоторый момент у вершины N должен появиться прямой потомок M ранга $i-1$. Кроме того, вершина M должна была стать прямым потомком вершины N на шаге (1) алгоритма 10.5, поскольку иначе ранг вершины N был бы не менее $i+1$. Отсюда

следует, что вершина M была корнем и по предположению индукции имела в тот момент по крайней мере 2^{i-1} потомков, а на шаге (1) алгоритма 10.5 в тот момент вершина N имела по крайней мере 2^{i-1} потомков. Таким образом, после слияния N имеет по крайней мере 2^i потомков. До тех пор пока N остается корнем, она не может потерять потомков. □

Лемма 10.5. *Если вершина N в какой-то момент работы алгоритма* 10.5 *имеет прямого потомка M, то ранг N больше ранга M.*

Доказательство. Прямая индукция по числу выполняемых команд. □

Следующая лемма дает границу числа вершин ранга i.

Лемма 10.6. *Существует не более $n2^{-i}$ вершин ранга i.*

Доказательство. Структура, создаваемая алгоритмом 10.5, представляет собой набор деревьев. Поэтому никакая вершина не может быть потомком двух различных вершин ранга i. Поскольку в структуре n вершин, результат вытекает непосредственно из леммы 10.4. □

Следствие. *Никакая вершина не имеет ранга, большего* $\log_2 n$. □

Определение. При фиксированном числе n объектов определим *группы* рангов. Будем говорить, что *целое i принадлежит группе j*, тогда и только тогда, когда

$$\log_2^{(j)}(n) \geqslant i > \log_2^{(j+1)}(n)$$

где $\log_2^{(1)}(n) = \log_2 n$ и $\log_2^{(k+1)}(n) = \log_2^{(k)}(\log_2(n))$. Таким образом, $\log_2^{(k)}$ — это функция, представляющая собой последовательное применение k раз функции $\log_2$. Например,

$$\log_2^{(3)}(65\,536) = \log_2^{(2)}(16) = \log_2(4) = 2$$

Говорят, что ранг r принадлежит *группе рангов* j, если r принадлежит группе j. Поскольку $\log_2^{(k)}(F(k)) = 1$ и никакая вершина не имеет ранга, большего $\log_2 n$, заключаем, что никакая вершина не принадлежит группе рангов, более высокой, чем $G(n)$. Группы рангов при $n = 65\,536$ приведены в табл. 10.4.

Таблица 10.4

Ранг вершины	Группа рангов
0	5
1	4
2	3
3, 4	2
5, 6, ..., 16	1

Теперь мы готовы доказать, что временная сложность алгоритма 10.5 равна $O(nG(n))$.

Теорема 10.3. *Любая последовательность σ, состоящая из $O(n)$ команд, выполняется за время $O(nG(n))$.*

Доказательство. Ясно, что общее время выполнения команд **слить** в σ равно $O(n)$. Длины путей, проходимых командами **найти** в σ, подсчитаем в два этапа. Предположим, что при выполнении команды **найти** мы поднимаемся вверх от вершины M к вершине N. Если M и N принадлежат разным группам рангов, то увеличим цену команды **найти** на одну единицу времени. Если N — корень, увеличим цену еще на одну единицу. Поскольку вдоль любого пути не более $G(n)$ различных групп рангов, никакой из команд **найти** не будет назначена цена, большая, чем $G(n)$ единиц.

Если, с другой стороны, M и N принадлежат одной группе рангов и N не корень, то вершине M набавим цену на 1 единицу времени. Отметим, что в этом случае вершина M должна быть перемещена. По лемме 10.5 ее новый прямой предок имеет более высокий ранг, чем предыдущий прямой предок. Таким образом, если вершина M принадлежит группе рангов j, то ей можно набавлять цену не более чем $\log_2^{(j)}(n)$ раз, прежде чем ее прямой предок попадет в более низкую группу рангов. Начиная с этого момента, цена для M уже больше не будет меняться; цена перемещения M будет порождаться выполненной командой **найти**, как это описано выше.

Ясно, что цена для всех команд **найти** составляет $O(nG(n))$. Чтобы найти верхнюю границу общей цены для всех объектов, просуммируем по всем группам рангов максимальную цену для каждой вершины в группе, умноженную на максимальное число вершин в группе. Пусть g_j — максимальное число вершин в группе рангов j, а c_j — цена для всех вершин в группе j. Тогда по лемме 10.6

$$g_j \leqslant \sum_{k=\log_2^{(j+1)}(n)}^{\log_2^{(j)}(n)} n2^{-k} \tag{10.2.1}$$

Слагаемые в формуле (10.2.1) образуют геометрическую прогрессию со знаменателем $1/2$, так что их сумма не превосходит удвоенного первого члена. Таким образом, $g_j \leqslant 2n\,2^{-\log_2^{(j+1)}(n)} = 2n/\log_2^{(j)}(n)$. Так как цена c_j ограничена сверху величиной $g_j \log_2^{(j)}(n)$, то $c_j \leqslant 2n$. Поскольку j может меняться только от 1 до $G(n)$, видим, что вершинам назначена цена в $O(nG(n))$ единиц времени. Следовательно, общая стоимость работы алгоритма 10.5 равна $O(nG(n))$. □

Применим теперь абстрактные результаты к грамматикам свойств.

Теорема 10.4. *Предположим, что механизм разбора и обработки таблиц, разработанный в алгоритмах* 10.3 *и* 10.4, *применяется ко входной цепочке длины* n. *Предположим также, что число запросов относительно свойств индекса в таблице составляет величину* $0(n)$ *и эти запросы относятся только к верхней таблице магазина, для которой индекс обладает не нейтральным свойством*[1]. *Тогда общее время, затраченное на обработку таблицы на машине с произвольным доступом, равно* $0(nG(n))$.

Доказательство. Заметим прежде всего, что предположения о нашей модели выполняются, а именно, что время, требуемое в алгоритме 10.3 для достижения любой вершины, которую нужно обработать, фиксировано и не зависит от n. Заголовков свойств (корней деревьев) можно достичь за фиксированное время, поскольку их конечное число на одну таблицу и они связаны. Индексные ячейки (вершины для объектов) непосредственно достижимы либо из таблицы расстановки, когда нам надо узнать их свойства (вот почему мы предполагали, что запросы осуществляются только для индексов верхней таблицы), либо в процессе спуска по списку пересечений.

Теперь достаточно показать, что каждый индекс и ячейку заголовка, которые порождаются, можно смоделировать как объектные вершины, что их всего $0(n)$ и что все манипуляции можно выразить в точности как некоторую последовательность команд **слить** и **найти**. Приведем полный список всех возможных порождаемых ячеек.

(1) $2n$ „объектов" соответствуют n ячейкам заголовков и n индексным ячейкам, создаваемым в процессе операции переноса (часть 1 алгоритма 10.3). С помощью операции **слить** можно сделать так, чтобы индексная ячейка указывала на ячейку заголовка.

(2) Новым индексным ячейкам, порожденным на шаге (5) части 2 алгоритма 10.3, соответствует самое большее n объектов. С помощью подходящей операции **слить** можно сделать так, чтобы эти ячейки указывали на правильный корень.

Таким образом, существует не более $3n$ объектов (n в алгоритме 10.5 означает $3n$ здесь). Кроме того, число команд, необходимых для обработки множеств и объектов при „моделировании" алгоритмов 10.3 и 10.4 алгоритмом 10.5, равно $0(n)$. Уже

[1]) Если речь идет о типичном использовании этих свойств (например, когда при свертке a в F в грамматике G_0 желательно знать свойства конкретного идентификатора a), то это предположение кажется вполне правдоподобным.

было отмечено, что для инициализации таблиц после переноса [(1) выше] и для сопоставления новых индексных ячеек с заголовками [(2) выше] достаточно не более $3n$ команд **слить**. Кроме того, на шаге (4) части 2 алгоритма 10.3, когда два множества индексов, обладающих одним и тем же свойством, сливаются, достаточно $0(n)$ команд **слить**. Это следует из того, что число различных свойств фиксировано и что можно сделать только $n-1$ сверток.

Из леммы 10.3 вытекает, что для проверки свойств индексов в списке пересечений [шаг (1) части 2 алгоритма 10.3] достаточно $0(n)$ команд **найти**. Наконец, по условию теоремы для определения свойств индексов (подразумевается использование в процессе перевода) требуется $0(n)$ дополнительных команд **найти**. Если разместить все эти команды в порядке, определяемом анализатором и алгоритмом 10.3, получим последовательность из $0(n)$ команд. Настоящая теорема следует, таким образом, из леммы 10.3 и теоремы 10.3. □

УПРАЖНЕНИЯ

10.2.1. Пусть G — КС-грамматика с правилами

$$E \to E+T \mid E \oplus T \mid T$$
$$T \to (E) \mid a$$

где a — идентификатор, $+$ представляет сложение с фиксированной точкой, $\oplus$ — сложение с плавающей точкой. Образуйте из G грамматику свойств, в которой

(1) каждый символ a имеет таблицу с одним не нейтральным элементом,

(2) если в вершине n дерева разбора применяется правило $E \to E+T$, то все a, вершины которых находятся ниже n, обладают свойством „используются в сложении с фиксированной точкой",

(3) если, как и в (2), применяется правило $E \to E \oplus T$, то все a, находящиеся ниже n, обладают свойством „используются в сложении с плавающей точкой",

(4) осуществляется разбор в соответствии с грамматикой G, при этом ведется контроль за тем, чтобы идентификатор, использованный в сложении с плавающей точкой, не использовался затем (выше в дереве разбора) в сложении с фиксированной точкой.

10.2.2. С помощью грамматики свойств из примера 10.9 постройте дерево разбора, если оно существует, для каждой из следующих входных цепочек:

```
(а) begin
      declare [1:1]
      declare [1:1]
      begin
          a[1:3]
      end
      label
      begin
        declare [3:1]
        a[3:3]
      end
      goto [2:4]
    end
(б) begin
      declare [1:1]
      a [1:3]
      begin
        declare [2:1]
        goto [1:4]
      end
    end
```

Определение. *Недетерминированная грамматика свойств* определяется так же, как грамматика свойств, за исключением того, что

(1) значениями функции μ являются подмножества множества V, и

(2) не требуется наличия нейтрального свойства.

В соответствии с соглашениями этого раздела мы пишем $\alpha A T \beta \Rightarrow \alpha X_1 T_1 \ldots X_n T_n \beta$, если $A \to X_1 \ldots X_n$ — некоторое правило вывода, скажем p, и для каждого i таблица $T(i)$ принадлежит множеству $\mu(p, T_1(i) \ldots T_n(i))$. Отношение $\Rightarrow^*$ и порождаемый язык определяются в точности так же, как и в детерминированном случае.

***10.2.3.** Докажите, что для каждой недетерминированной грамматики свойств G найдется грамматика свойств G' (возможно, без нейтрального свойства), порождающая тот же язык, в которой μ является функцией.

10.2.4. Покажите, что если грамматика G из упр. 10.2.3 обладает нейтральным свойством (т. е. все, кроме конечного числа, целые числа обладают этим свойством в каждой таблице каждого вывода), то G' — грамматика свойств с нейтральным свойством.

10.2.5. Обобщите алгоритм 10.3 для грамматик не в нормальной форме Хомского. Обобщенный алгоритм должен иметь ту же временну́ю сложность, что и исходный.

***10.2.6.** Изменим определение грамматики свойств, потребовав, чтобы ни у какого терминального символа не было полностью нейтральной таблицы. Если G — такая грамматика свойств, то пусть $L'(G) = \{a_1 \ldots a_n \mid a_1 T_1 \ldots a_n T_n \in L(G)$ для некоторых (не полностью нейтральных) таблиц $T_1, \ldots, T_n\}$. Покажите, что $L'(G)$ может не быть КС-языком. *Указание*: Покажите, что таким способом порождается язык $\{a^i b^j c^k \mid i \leqslant j \leqslant k\}$.

****10.2.7.** Покажите, что для „грамматики свойств" G, модифицированной в упр. 10.2.6, неразрешима проблема пустоты множества $L'(G)$, даже если входная КС-грамматика для G праволинейная.

10.2.8. Пусть G — входная КС-грамматика из примера 10.9. Предположим, что терминал **declare** связывает с индексом одно из двух свойств: либо „объявлен как вещественное", либо „объявлен как целое". Определите на G такую грамматику свойств, что если реализуется алгоритм 10.3, то самая верхняя таблица в магазине, имеющая конкретный индекс i с не нейтральным свойством (т. е. таблица с ячейкой для i, на которую указывает элемент таблицы расстановки для i), будет иметь в качестве свойства для i текущее объявление идентификатора i. Таким образом, решить, вещественный ли или целый идентификатор, можно сразу же, как только он обнаружен во входном потоке.

10.2.9. Найдите выход алгоритма 10.5 и окончательное дерево при данном множестве объектов $\{a_1, \ldots, a_{12}\}$ и следующей последовательности команд. Предположите, что в случае равенства размеров корень для A_i становится потомком корня для A_j при $i < j$.

слить (A_1, A_2, A_1)
слить (A_3, A_1, A_1)
слить (A_4, A_5, A_4)
слить (A_6, A_4, A_4)
слить (A_7, A_8, A_7)
слить (A_9, A_7, A_7)
слить (A_{10}, A_{11}, A_{10})
слить (A_{12}, A_{10}, A_{10})
найти (a_1)
слить (A_1, A_4, A_1)
слить (A_7, A_{10}, A_2)
найти (a_7)
слить (A_1, A_2, A_1)
найти (a_3)
найти (a_4)

****10.2.10.** Предположим, что алгоритм 10.5 модифицирован так, что при выполнении слияний каждый корень может стать потомком другого. Покажите, что временна́я сложность такого алгоритма не лучше $0(n \log n)$.

Открытые проблемы

10.2.11. Действительно ли алгоритм 10.5, как это утверждается в нашей книге, имеет сложность $0(nG(n))$, или его сложность равна $0(n)$, или, возможно, лежит где-то между этими величинами?

10.2.12. Составляет ли временна́я сложность модифицированного алгоритма из упр. 10.2.10 величину $0(n \log n)$?

Проблема для исследования

10.2.13. Исследуйте или охарактеризуйте разновидности свойств идентификаторов, которые можно правильно обработать грамматиками свойств.

Замечания по литературе

Грамматики свойств впервые были определены Стирнзом и Льюисом [1969]. В их работе можно найти ответы к упр. 10.2.6 и 10.2.7. Метод реализации грамматик свойств за $n \log \log n$ обсуждался Стирнзом и Розенкранцем [1969].

Насколько нам известно, впервые алгоритм 10.5 предложили Моррис и Мак-Илрой, но они не опубликовали его. Анализ алгоритма проведен Хопкрофтом и Ульманом [1972]. Упр. 10.2.10 взято у Фишера [1972].

11 ОПТИМИЗАЦИЯ КОДА

Одной из самых трудных и непонятных проблем, возникающих при построении компиляторов, является генерация „хорошего" объектного кода. Два наиболее широко распространенных критерия, по которым определяют качество программы, заключаются в оценке времени ее выполнения и размера. К сожалению, для данной конкретной программы, вообще говоря, невозможно установить время выполнения самой быстрой эквивалентной программы или длину самой короткой эквивалентной программы. Как уже указывалось в гл. 1, если программа имеет циклы, мы вынуждены довольствоваться только улучшением кода, а не его истинной оптимизацией.

Большинство алгоритмов улучшения кода можно рассматривать как приложение различных преобразований к некоторому промежуточному представлению исходной программы с целью привести промежуточную программу к форме, из которой можно получить более эффективный объектный код. Эти преобразования по улучшению кода могут применяться в любой точке процесса компиляции. Один из широко распространенных методов заключается в применении этих преобразований к программе на промежуточном языке, появляющейся после синтаксического анализа, но перед генерацией кода.

Преобразования по улучшению кода можно разделить на машинно-независимые и машинно-зависимые. Примером машинно-независимой оптимизации может служить удаление из программы бесполезных операторов, т. е. таких, которые никаким образом не влияют на выход программы[1]). Такие машинно-независимые преобразования были бы полезны во всех компиляторах.

С помощью машинно-зависимых преобразований можно попытаться привести программу к форме, в которой могут ска-

[1]) Поскольку обычно такие операторы не должны появляться в программе, весьма вероятно, что в программе ошибка, и потому компилятору следовало бы информировать пользователя о бесполезности оператора.

заться преимущества, связанные с машинными командами специального вида. Как следствие этого машинно-зависимые преобразования трудно охарактеризовать в общем виде, и потому мы их рассматривать не будем.

В настоящей главе мы изучим различные машинно-независимые преобразования, применимые к промежуточной программе, появляющейся внутри компилятора после синтаксического анализа, но перед генерацией кода. Начнем с того, что покажем, как генерировать оптимальный код для простого, но важного класса линейных программ. Затем расширим этот класс, включив в программы циклы, и исследуем некоторые методы улучшения кода, которые можно применить к этим программам.

11.1. ОПТИМИЗАЦИЯ ЛИНЕЙНОГО УЧАСТКА

Рассмотрим сначала схему программы, представляющей собой последовательность операторов присваивания, каждый из которых имеет вид $A \leftarrow f(B_1, \ldots, B_r)$, где A и $B_1, \ldots, B_r$ — скалярные переменные, а f — функция от r переменных для некоторого r. Для этого ограниченного класса программ мы разработаем множество преобразований и покажем, как использовать их для нахождения программы, оптимальной относительно некоторой оценки. Поскольку действительная оценка программы зависит от природы машинного кода, который в конечном итоге будет выработан, мы рассмотрим, какие части процедуры оптимизации машинно-независимы и как зависят оставшиеся от реально выбранной машины.

11.1.1. Модель линейного участка

Начнем с определения *блока*. Блок моделирует часть программы на промежуточном языке, которая содержит только операторы присваивания.

Определение. Пусть Σ — счетное множество *имен переменных*, а Θ — конечное множество *операций*. Предположим, что каждая операция $\theta \in \Theta$ имеет известное фиксированное количество операндов. Предположим также, что Θ и Σ не пересекаются.

Оператором называется цепочка вида

$$A \leftarrow \theta B_1 \ldots B_r$$

где $A, B_1, \ldots, B_r$ — переменные из Σ, а θ — это r-местная операция из Θ. Будем говорить, что оператор осуществляет *присваивание* переменной A и *ссылается на* $B_1, \ldots, B_r$.

Блок $\mathcal{B}$ — это тройка (P, I, U), где

(1) P — список операторов $S_1; S_2; \ldots; S_n$ $(n \geqslant 0)$,

(2) I — множество *входных* переменных,
(3) U — множество *выходных* переменных.

Будем предполагать, что если оператор S_j ссылается на A, то либо A — входная переменная, либо осуществлено присваивание ей значения некоторым оператором до S_j (т. е. некоторым оператором S_i, $i < j$). Таким образом, внутри блока все переменные, на которые ссылаются, к этому моменту определены либо внутренним образом как переменные, которым присвоены значения, либо внешним как входные переменные. Аналогично будем предполагать, что каждая выходная переменная либо является входной переменной, либо ей присвоено значение некоторым оператором.

Типичным можно считать оператор $A \leftarrow +BC$, который есть не что иное, как префиксная запись более привычного оператора присваивания $A \leftarrow B+C$. Если оператор осуществляет присваивание переменной A, то с этим присваиванием можно связать некоторое „значение". Последнее представляет собой формулу для A в терминах (неизвестных) первоначальных значений входных переменных. Эту формулу можно записать как префиксное выражение, включающее входные переменные и операции.

Начиная отсюда, будем предполагать, что входные переменные имеют неизвестные значения и их надо рассматривать как алгебраические неизвестные. Кроме того, значения операций и множества, на которых они определены, не конкретизированы, так что в качестве значения мы можем рассматривать лишь формулу, а не некоторую величину.

Определение. Пусть (P, I, U) — блок, $P = S_1; \ldots; S_n$. Определим *значение* $v_t(A)$ переменной A непосредственно после момента времени t, $0 \leqslant t \leqslant n$, как следующее префиксное выражение:

(1) Если $A \in I$, то $v_0(A) = A$.
(2) Если оператором S_t является $A \leftarrow \theta B_1 \ldots B_r$, то
  (а) $v_t(A) = \theta v_{t-1}(B_1) \ldots v_{t-1}(B_r)$,
  (б) $v_t(C) = v_{t-1}(C)$ для всех $C \neq A$, при условии что $v_{t-1}(C)$ определено.
(3) Для всех $A \in \Sigma$, если $v_t(A)$ не определено по (1) или (2), то оно не определено.

Отметим, что поскольку у каждой операции известное число операндов, каждое выражение, имеющее значение, либо является одиночным символом из Σ, либо допускает единственное представление в виде $\theta E_1 \ldots E_r$, где θ — это r-местная операция, а $E_1, \ldots, E_r$ — выражения, имеющие значения. (См. упр. 3.1.17.)

Значением $v(\mathcal{B})$ блока $\mathcal{B}=(P, I, V)$ называется множество

$$\{v_n(A) \mid A \in V,\ n\text{—число операторов в } P\}$$

Два блока считаются (*топологически*) *эквивалентными* ($\equiv$), если они имеют одно и то же значение. Отметим, что цепочки, образующие префиксные выражения, равны тогда и только тогда, когда они тождественны. Таким образом, пока мы не будем предполагать никаких алгебраических тождеств. В разд. 11.1.6 мы изучим эквивалентность блоков относительно некоторых алгебраических законов.

Пример 11.1. Пусть $I=\{A, B\}$, $U=\{F, G\}$ и P состоит из операторов [1])

$$\begin{aligned} T &\leftarrow A+B \\ S &\leftarrow A-B \\ T &\leftarrow T*T \\ S &\leftarrow S*S \\ F &\leftarrow T+S \\ G &\leftarrow T-S \end{aligned}$$

Сначала $v_0(A)=A$ и $v_0(B)=B$. После первого оператора $v_1(T)=A+B$ (в префиксной записи $v_1(T)=+AB$), $v_1(A)=A$ и $v_1(B)=B$. После второго оператора $v_2(S)=A-B$, а значения остальных переменных прежние. После третьего оператора $v_3(T)=(A+B)*(A+B)$. Последние три оператора приводят к следующим значениям (остальные значения переносятся с предыдущего шага):

$$\begin{aligned} v_4(S) &= (A-B)*(A-B) \\ v_5(F) &= (A+B)*(A+B)+(A-B)*(A-B) \\ v_6(G) &= (A+B)*(A+B)-(A-B)*(A-B) \end{aligned}$$

Поскольку $v_6(F)=v_5(F)$, значением блока будет

$$\{(A+B)*(A+B)+(A-B)*(A-B),\ (A+B)*(A+B)-(A-B)*(A-B)\}\ ^{2)}$$

Напомним, что мы не учитываем никаких алгебраических законов. Если применить обычные законы алгебры, то можно было бы записать $F=2(A^2+B^2)$ и $G=4AB$. □

[1]) При изображении списка операторов мы часто пользуемся переходом на новую строку вместо точки с запятой в качестве разделителя между операторами. В примерах для бинарных операций мы будем употреблять инфиксную запись.

[2]) В префиксной записи значение блока таково:

$$\{+*+AB+AB*-AB,\ -AB,\ -*+AB+AB*-AB-AB\}$$

Во избежание излишней сложности мы не будем на этот раз рассматривать операторы, включающие структурированные переменные (массивы и т. д.). Один из путей обработки массивов заключается в следующем. Если A — массив, то рассматриваем оператор присваивания $A(I)=J$ так, как если бы A была скалярной переменной, которой было присвоено значение некоторой функции от I, J и предыдущего значения. Другими словами, можно написать $A \leftarrow \theta AIJ$, где θ — операция, символизирующая присваивание массиву. Аналогично такой оператор, как $J=A(I)$, можно выразить в виде $J \leftarrow \psi AI$.

Сделаем, кроме того, еще несколько предположений, в результате чего наши построения потеряют некоторую общность. Например, будем пренебрегать операторами проверки, константами, операторами присваивания вида $A \leftarrow B$. Однако снятие этих ограничений приведет к аналогичной теории, и мы делаем эти предположения прежде всего для удобства — чтобы привести пример теории такого типа.

Наши основные предположения:

(1) Важным фактором, касающимся блока, является набор функций входных переменных (переменных, определенных вне блока), вычисляемых внутри блока. Сколько раз вычисляется конкретная функция, несущественно. Такой подход основан на том, что различные блоки программы передают друг другу значения переменных и что нет необходимости передавать две копии одного и того же значения из одного блока в другой.

(2) Имена переменных, участвующих в вычислениях, несущественны. Это предположение неудовлетворительно, если блок составляет часть цикла и функция вычисляется повторно. Например, если вычисляется $I \leftarrow I+1$, то недопустимо заменять это вычисление на $J \leftarrow I+1$, а затем повторять блок, ожидая, что результаты будут такими же. Тем не менее мы принимаем это предположение, поскольку оно ведет к некоторой симметрии решения и часто оказывается справедливым. В упражнениях предлагается сделать модификации, необходимые для того, чтобы допустить, что некоторым выходным значениям даны фиксированные имена.

(3) Мы не включаем операторы вида $X \leftarrow Y$. Если такой оператор встречается, то при условии, что предположение (2) выполнено, можно вместо X подставить Y, исключив всюду X. И вновь это предположение вносит симметрию в модель, и читателю предлагается модифицировать теорию так, чтобы такие операторы можно было включать.

11.1.2. Преобразования блоков

Заметим, что если даны два блока $\mathscr{B}_1$ и $\mathscr{B}_2$, то можно установить, эквивалентны ли они, вычислив их значения $v(\mathscr{B}_1)$ и $v(\mathscr{B}_2)$ и выяснив, выполняется ли равенство $v(\mathscr{B}_1)=v(\mathscr{B}_2)$. Однако можно указать бесконечное число блоков, эквивалентных любому данному.

Например, если $\mathscr{B}=(P,\ I,\ U)$ — блок, X — переменная, на которую нет ссылки в $\mathscr{B}$, A — входная переменная, а θ — операция, то к P можно любое число раз присоединять оператор $X \leftarrow 0A\ldots A$, не меняя значения $\mathscr{B}$.

Если выбрать подходящую оценку, то не все эквивалентные блоки будут одинаково эффективны. При данном блоке $\mathscr{B}$ существуют различные преобразования, применимые для отображения его в эквивалентный и, возможно, более желательный блок $\mathscr{B}'$. Пусть $\mathscr{T}$ — множество всех преобразований, сохраняющих эквивалентность блоков. Мы покажем, что любое преобразование из $\mathscr{T}$ можно осуществить с помощью конечной последовательности четырех простейших преобразований блоков. Затем опишем последовательности преобразований, которые приводят к блоку, оптимальному относительно некоторой разумной оценки.

Определение. Пусть $\mathscr{B}=(P,\ I,\ U)$ — блок, $P=S_1;S_2;\ldots;S_n$. Для единообразия обозначений примем, что все элементы входного множества I приписаны к некоторому нулевому оператору S_0, а все элементы выходного множества — к некоторому $(n+1)$-у оператору S_{n+1}.

Переменная A называется *активной* после момента времени t, если

(1) ей присвоено значение некоторым оператором S_i,
(2) ей не присвоены значения операторами $S_{i+1}, S_{i+2}, \ldots, S_j$,
(3) на нее ссылается оператор S_{j+1},
(4) $0 \leqslant i \leqslant t \leqslant j \leqslant n$.

Если число j, о котором идет речь выше, имеет максимально возможное значение, то последовательность операторов S_{i+1}, $S_{i+2}, \ldots, S_{j+1}$ называют *областью действия* оператора S_i и областью действия этого присваивания переменной A. Если A — выходная переменная и после S_i ей не присваивается значение, то $j=n+1$, и говорят, что U лежит в области действия оператора S_i (это следует из принятого выше соглашения; здесь мы лишь подчеркиваем это).

Если блок содержит такой оператор S, что переменная, которой присваивается значение в S, не является активной после этого оператора, то область действия оператора S пуста, и говорят, что S — *бесполезный* оператор. Другими словами, S — бесполезный оператор, если он присваивает значение переменной, которая не является выходной и на которую нет ссылки в последующих операторах.

Пример 11.2. Рассмотрим блок, в котором α, β и γ — списки из нуля или более операторов:

$$\begin{array}{l} \alpha \\ A \leftarrow B+C \\ \beta \\ D \leftarrow A*E \\ \gamma \end{array}$$

Если переменной A не присваивается значение в последовательности операторов β и на нее нет ссылки из γ, то область действия оператора $A \leftarrow B+C$ включает полностью β и оператор $D \leftarrow A*E$. Если в γ нет операторов, ссылающихся на D, и D не является выходной переменной, то оператор $D \leftarrow A*E$ бесполезен. □

Определим теперь четыре простейших преобразования блоков, сохраняющие эквивалентность. Пусть $\mathscr{B}=(P,\ I,\ U)$ — блок и $P=S_1;S_2;\ldots;S_n$. Как и выше, предположим, что всем входным переменным присваиваются значения в операторе S_0 и все выходные переменные входят в оператор S_{n+1}. Преобразования определим в терминах их воздействия на данный блок $\mathscr{B}$. Первое из преобразований интуитивно желательно, а именно из блока удаляются все те входные переменные или операторы, которые не оказывают влияния на выходные переменные.

T_1: Удаление бесполезных присваиваний

Если оператор S_i, $0 \leqslant i \leqslant n$, присваивает значение переменной A и она не активна после момента i, то

(1) при $i>0$ можно удалить S_i из P,
(2) при $i=0$ можно удалить A из I.

Пример 11.3. Пусть $\mathscr{B}=(P,\ I,\ U)$, где $I=\{A,\ B,\ C\}$, $U=\{F,\ G\}$ и P состоит из

$$\begin{array}{l} F \leftarrow A+A \\ G \leftarrow F*C \\ F \leftarrow A+B \\ G \leftarrow A*B \end{array}$$

Второй оператор бесполезен, так как его область действия пуста. Таким образом, одно применение преобразования T_1

отображает $\mathcal{B}$ в $\mathcal{B}_1=(P_1, I, U)$, где P_1 состоит из

$$F \leftarrow A+A$$
$$F \leftarrow A+B$$
$$G \leftarrow A*B$$

Теперь в $\mathcal{B}_1$ бесполезны входная переменная C и первый оператор. Поэтому T_1 можно применить дважды и получить $\mathcal{B}_2 = (P_2, \{A, B\}, U)$, где P_2 состоит из

$$F \leftarrow A+B$$
$$G \leftarrow A*B$$

Отметим, что $\mathcal{B}_2$ получается независимо от того, удаляется сначала переменная C или первый оператор из P_1. □

Для того чтобы можно было систематически удалять из блока $\mathcal{B}=(P, I, U)$ все бесполезные операторы, надо определить множество полезных переменных (тех, которые явно или неявно используются в вычислении выхода) после каждого оператора блока, начиная с последнего оператора в P и поднимаясь затем вверх. Ясно, что $U_n=U$ — множество переменных, полезных после последнего оператора S_n.

Предположим, что оператором S_i является $A \leftarrow \theta B_1 \ldots B_r$ и U_i — множество переменных, полезных после S_i.

(1) Если $A \in U_i$, то S_i — полезный оператор, так как переменная A используется для вычисления выходной переменной. Тогда множество U_{i-1} полезных переменных после S_{i-1} находится заменой A в U_i на переменные $B_1, \ldots, B_r$ (т. е. $U_{i-1} = (U_i - \{A\}) \cup \{B_1, \ldots, B_r\}$).

(2) Если $A \notin U_i$, то оператор S_i бесполезен, и его можно удалить. В этом случае $U_{i-1}=U_i$.

(3) После вычисления U_0 можно удалить из I все входные переменные, которых нет в U_0.

Второе преобразование блоков сливает общие выражения.

T_2: *Исключение избыточных вычислений*

Предположим теперь, что $\mathcal{B}=(P, I, U)$ — блок, где P имеет вид

$$\alpha$$
$$A \leftarrow \theta C_1 \ldots C_r$$
$$\beta$$
$$B \leftarrow \theta C_1 \ldots C_r$$
$$\gamma$$

причем ни одна из переменных $C_1, \ldots, C_r$ не есть A и ни одной из них не присваивается значение ни в каком операторе в β.

Преобразование T_2 отображает $\mathcal{B}$ в $\mathcal{B}'=(P', I, U')$, где P' есть

$$\alpha$$
$$D \leftarrow \theta C_1 \ldots C_r$$
$$\beta'$$
$$\gamma'$$

и

(1) β' — это список β, в котором все ссылки на переменную A в области действия данного изображения этой переменной заменены ссылками на D,

(2) γ' — это список γ, в котором все ссылки на A и B в областях действия данных изображений A и B заменены ссылками на D.

Если область действия переменной A или B распространяется на S_{n+1}, то U' — это множество U, в котором A или B заменена на D. В противном случае $U'=U$.

D может быть любым именем, не меняющим значения блока. Подходит любой символ, на который нет ссылки в P; кроме того, могут использоваться и некоторые символы, на которые есть ссылки в P.

Пример 11.4. Пусть $\mathcal{B}=(P, \{A, B\}, \{F, G\})$, где P состоит из

$$S \leftarrow A+B$$
$$F \leftarrow A*S$$
$$R \leftarrow B+B$$
$$T \leftarrow A*S$$
$$G \leftarrow T*R$$

Второй и четвертый операторы дают избыточные вычисления, так что к $\mathcal{B}$ можно применить преобразование T_2, в результате чего получится $\mathcal{B}'=(P', \{A, B\}, \{D, G\})$, где P' состоит из

$$S \leftarrow A+B$$
$$D \leftarrow A*S$$
$$R \leftarrow B+B$$
$$G \leftarrow D*R$$

Выходным множеством становится $\{D, G\}$. D может быть любым новым символом или одной из переменных F, A, S или T. Легко проверить, что если в качестве D взять B, R или G, то изменится значение программы. □

T_3: *Переименование*

Ясно, что поскольку речь идет о значении блока $\mathcal{B}=(P, I, U)$, имена переменных, которым присваиваются значения, несущественны. Предположим, что оператором S_i в P является $A \leftarrow$

$\theta B_1 \ldots B_r$ и переменная C не активна в области действия оператора S_i. Тогда можно положить $\mathscr{B}' = (P', I, U')$, где P' — это P, в котором оператор S_i заменен на $C \leftarrow \theta B_1 \ldots B_r$, а все вхождения A в области действия оператора S_i заменены на C. Если U лежит в области действия оператора S_i, то U' — это U, в котором переменная A заменена на C. В противном случае $U' = U$. Преобразование T_3 отображает $\mathscr{B}$ в $\mathscr{B}'$.

Пример 11.5. Пусть $\mathscr{B} = (P, \{A, B\}, \{F\})$, где P состоит из

$$T \leftarrow A * B$$
$$T \leftarrow T + A$$
$$F \leftarrow T * T$$

Одно применение T_3 позволяет изменить имя переменной T, которой присваивается значение первым оператором, на S. Таким образом, T_3 отображает $\mathscr{B}$ в $\mathscr{B}' = \{P', \{A, B\}, \{F\})$, где P' состоит из

$$S \leftarrow A * B$$
$$T \leftarrow S + A$$
$$F \leftarrow T * T$$

Отметим, что переменная T заменена на S только в первом операторе присваивания. □

T_4: *Перестановка*

Пусть $\mathscr{B} = (P, I, U)$ — блок, в котором оператором S_i является $A \leftarrow \theta B_1 \ldots B_r$, оператором S_{i+1} является $C \leftarrow \psi D_1 \ldots D_s$, A не совпадает ни с одной из переменных $C, D_1, \ldots, D_s$ и C не совпадает ни с одной из переменных $A, B_1, \ldots, B_r$. Тогда преобразование T_4 отображает блок $\mathscr{B}$ в $\mathscr{B}' = (P', I, U)$, где P' — это P, в котором S_i и S_{i+1} переставлены.

Пример 11.6. Пусть $\mathscr{B} = (P, \{A, B\}, \{F, G\})$, где P состоит из

$$F \leftarrow A + B$$
$$G \leftarrow A * B$$

Можно применить T_4 и отобразить $\mathscr{B}$ в $(P', \{A, B\}, \{F, G\})$, где P' состоит из

$$G \leftarrow A * B$$
$$F \leftarrow A + B$$

Однако с помощью T_4 нельзя отобразить блок $\mathscr{B}_1 = (P_1, \{A, B\}, \{F, G\})$, где P_1 состоит из

$$F \leftarrow A + B$$
$$G \leftarrow F * A$$

в блок $\mathscr{B}_2 = (P_2, \{A, B\}, \{F, G\})$, где P_2 состоит из

$$G \leftarrow F * A$$
$$F \leftarrow A + B$$

В самом деле, в этом случае $\mathscr{B}_2$ даже не блок, потому что переменная F используется, не будучи предварительно определенной. □

Определим теперь некоторые отношения эквивалентности, связанные с действиями четырех определенных выше преобразований.

Определение. Пусть S — подмножество множества $\{1, 2, 3, 4\}$. Будем писать $\mathscr{B}_1 \Rightarrow_S \mathscr{B}_2$, если одно применение преобразования T_i переводит $\mathscr{B}_1$ в $\mathscr{B}_2$, причем $i \in S$. Будем писать $\mathscr{B}_1 \overset{*}{\underset{S}{\Leftrightarrow}} \mathscr{B}_2$, если существует такая последовательность блоков $\mathscr{C}_0, \ldots, \mathscr{C}_n$, что

(1) $\mathscr{C}_0 = \mathscr{B}_1$,
(2) $\mathscr{C}_n = \mathscr{B}_2$,
(3) для каждого i, $0 \leqslant i \leqslant n$, либо $\mathscr{C}_i \Rightarrow_S \mathscr{C}_{i+1}$, либо $\mathscr{C}_{i+1} \Rightarrow_S \mathscr{C}_i$.

Таким образом, $\overset{*}{\underset{S}{\Leftrightarrow}}$ — это наименьшее отношение эквивалентности, содержащее $\Rightarrow_S$ и отражающее идею о том, что преобразования могут применяться в любом направлении.

Соглашение. Подмножества $\{1, 2, 3, 4\}$ будем изображать без скобок, так что, например, $\Rightarrow_{\{1,2\}}$ будем записывать $\Rightarrow_{1,2}$.

Теперь мы хотим показать, что блоки $\mathscr{B}_1$ и $\mathscr{B}_2$ эквивалентны тогда и только тогда, когда существует последовательность преобразований, включающая в себя только преобразования $T_1 - T_4$, которая отображает $\mathscr{B}_1$ в $\mathscr{B}_2$. Иными словами, $\mathscr{B}_1 \equiv \mathscr{B}_2$ тогда и только тогда, когда $\mathscr{B}_1 \overset{*}{\underset{1,2,3,4}{\Leftrightarrow}} \mathscr{B}_2$. Достаточность условия проверяется легко: надо лишь показать, что каждое отдельное преобразование сохраняет значение блока.

Теорема 11.1. *Если* $\mathscr{B}_1 \Rightarrow_{1,2,3,4} \mathscr{B}_2$, *то* $\mathscr{B}_1 \equiv \mathscr{B}_2$.

Доказательство. Упражнение. Читатель должен также показать, что для D в преобразовании T_2 можно выбрать любое новое имя, как это утверждалось в описании преобразования. □

Следствие. *Если* $\mathscr{B}_1 \overset{*}{\underset{1,2,3,4}{\Leftrightarrow}} \mathscr{B}_2$, *то* $\mathscr{B}_1 \equiv \mathscr{B}_2$. □

В разд. 11.1.4 будет доказано обращение этого утверждения.

11.1.3. Графическое представление блоков

В настоящем разделе мы покажем, что для каждого блока $\mathcal{B}=(P, I, U)$ можно найти ориентированный ациклический граф D, естественным образом представляющий $\mathcal{B}$. Каждый лист графа D соответствует одной входной переменной в I, а каждая его внутренняя вершина—оператору из P. К ориентированным ациклическим графам легко применяются преобразования блоков, рассмотренные в предыдущем разделе.

Определение. Пусть $\mathcal{B}=(P, I, U)$—блок. Построим из $\mathcal{B}$ помеченный упорядоченный граф[1]) $D(\mathcal{B})$:

(1) Пусть $P=S_1; \ldots ; S_n$.

(2) Для каждой переменной $A \in I$ образуем вершину с меткой A и будем называть ее *последним определением* для A.

(3) Для $i=1, 2, \ldots, n$ делаем следующее. Пусть оператором S_i является $A \leftarrow \theta B_1 \ldots B_r$. Образуем новую вершину, помеченную θ, из которой выходят r ориентированных дуг. Пусть j-я дуга (при упорядочении дуг слева направо) указывает на последнее определение для B_j, $1 \leqslant j \leqslant r$. Новая вершина, помеченная θ, становится последним определением для A. Эта вершина *соответствует* оператору S_i в P.

(4) После шага (3) вершины, являющиеся последними определениями выходных переменных, в дальнейшем помечаются как „выделенные". Выделенные вершины будем изображать двойными кружками.

Пример 11.7. Пусть $\mathcal{B}=(P, \{A, B\}, \{F, G\})$—блок, в котором P состоит из операторов

$$T \leftarrow A+B$$
$$F \leftarrow A*T$$
$$T \leftarrow B+F$$
$$G \leftarrow B*T$$

Граф $D(\mathcal{B})$ изображен на рис. 11.1.

Отметим, что на рис. 11.1 четыре оператора из $\mathcal{B}$ соответствуют по порядку вершинам n_1, n_2, n_3, n_4. Отметим также, что прямым потомком вершины n_4 является n_3, а не n_1, поскольку при создании вершины n_4 вершина n_3 была последним определением для T. □

Каждый граф естественным образом представляет класс эквивалентности отношения $\overset{*}{\underset{3,4}{\Leftrightarrow}}$. Иными словами, если блок $\mathcal{B}_1$ с помощью некоторой последовательности преобразований T_3 и T_4

[1]) До конца этого и в следующем разделе под словом „граф" будем подразумевать „ориентированный ациклический граф".— *Прим. ред.*

можно отобразить в блок $\mathcal{B}_2$, то блоки $\mathcal{B}_1$ и $\mathcal{B}_2$ имеют один и тот же граф, и обратно. Одна часть этого утверждения составляет приведенную ниже лемму, доказательство которой заключается в использовании определений и предоставляется читателю.

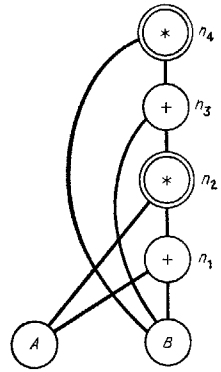


Рис. 11.1. Пример ориентированного ациклического графа.

Лемма 11.1. *Если* $\mathcal{B}_1 \Rightarrow_{3,4} \mathcal{B}_2$, *то* $D(\mathcal{B}_1)=D(\mathcal{B}_2)$.

Доказательство. Упражнение. □

Следствие. *Если* $\mathcal{B}_1 \overset{*}{\underset{3,4}{\Leftrightarrow}} \mathcal{B}_2$, *то* $D(\mathcal{B}_1)=D(\mathcal{B}_2)$. □

Доказательство обратного утверждения более трудно. Для него понадобятся еще одно определение и лемма.

Определение. Блок $\mathcal{B}=(P, I, U)$ называют *открытым*, если

(1) ни один из операторов в P не имеет вида $A \leftarrow \alpha$, где $A \in I$,

(2) в P нет двух операторов, присваивающих значение одной и той же переменной.

В открытом блоке $\mathcal{B}=(P, I, U)$ все операторы S_i из P присваивают значения различным переменным X_i, не входящим в I. В лемме 11.2 утверждается, что открытый блок всегда можно получить переименованием переменных с помощью только преобразования T_3.

Лемма 11.2. *Пусть* $\mathcal{B}=(P, I, U)$—*блок. Тогда существует такой эквивалентный открытый блок* $\mathcal{B}'=(P', I, U')$, *что* $\mathcal{B} \overset{*}{\underset{3}{\Leftrightarrow}} \mathcal{B}'$.

Доказательство. Упражнение. □

Следующая теорема показывает, что два блока имеют один и тот же граф тогда и только тогда, когда один из них можно преобразовать в другой переименованием и перестановкой.

Теорема 11.2. $D(\mathcal{B}_1)=D(\mathcal{B}_2)$ *тогда и только тогда, когда* $\mathcal{B}_1 \overset{*}{\underset{3,4}{\Leftrightarrow}} \mathcal{B}_2$.

Доказательство. Достаточность следует из леммы 11.1. Для доказательства необходимости рассмотрим два блока $\mathcal{B}_1=(P_1, I_1, U_1)$ и $\mathcal{B}_2=(P_2, I_2, U_2)$, для которых $D(\mathcal{B}_1)=D(\mathcal{B}_2)=D$. Поскольку графы одинаковы, входные множества должны совпа-

дать, так что можно положить $I_1 = I_2 = I$. Кроме того, число операторов в P_1 и P_2 должно быть одинаковым, так что можно положить $P_1 = S_1; \ldots; S_n$ и $P_2 = R_1; \ldots; R_n$.

С помощью переименования T_3 можно построить два открытых блока $\mathscr{B}_1' = (P_1', I_1', U_1')$ и $\mathscr{B}_2' = (P_2', I_2', U_2')$ с одним и тем же множеством переменных, которым присваиваются значения, причем

(1) $\mathscr{B}_1' \underset{3}{\overset{*}{\Leftrightarrow}} \mathscr{B}_1$,

(2) $\mathscr{B}_2' \underset{3}{\overset{*}{\Leftrightarrow}} \mathscr{B}_2$,

(3) если $P_1' = S_1'; \ldots; S_n'$ и $P_2' = R_1'; \ldots; R_n'$, то S_i' и R_j' осуществляют присваивание одной и той же переменной тогда и только тогда, когда им соответствует одна и та же вершина в D. (Из следствия леммы 11.1 видно, что $D(\mathscr{B}_1') = D(\mathscr{B}_2') = D$.)

При создании открытых блоков сначала всем переменным блоков $\mathscr{B}_1$ и $\mathscr{B}_2$ даем новые имена. Затем можно снова воспользоваться переименованием, чтобы удовлетворить условию (3).

Будем теперь строить такую последовательность блоков $\mathscr{C}_0, \ldots, \mathscr{C}_n$, что

(4) $\mathscr{C}_0 = \mathscr{B}_1'$,

(5) $\mathscr{C}_n = \mathscr{B}_2'$,

(6) $\mathscr{C}_i \underset{4}{\overset{*}{\Leftrightarrow}} \mathscr{C}_{i+1}$ для $0 \leqslant i \leqslant n$,

(7) операторами в $\mathscr{C}_i$ являются $R_1'; \ldots; R_i'$, за которыми следуют операторы из $S_1'; \ldots; S_n'$, не осуществляющие присваиваний тем переменным, которым уже присвоены значения одним из операторов $R_1'; \ldots; R_i'$. Ясно, что $D(\mathscr{C}_i) = D$ и операторы, определяющие одни и те же переменные в $\mathscr{C}_i$ и $\mathscr{B}_2'$, порождают одинаковые вершины в D.

Начнем с $\mathscr{C}_0 = \mathscr{B}_1'$. Условие (7) удовлетворяется тривиально. Предположим, что мы уже построили $\mathscr{C}_i$, $i \geqslant 0$. Список операторов из $\mathscr{C}_i$ можно записать как $R_1'; \ldots; R_i'; S_{j_1}'; \ldots; S_{j_{n-i}}'$, и в нем операторы $S_{j_1}', \ldots, S_{j_{n-i}}'$ удовлетворяют условию (7). По определению P_1' и P_2' найдется оператор S_{j_k}', осуществляющий присваивание той же переменной, что и R_{i+1}'. Поскольку S_{j_k}' и R_{i+1}' соответствуют одной и той же вершине в D, то S_{j_k}' ссылается только на переменные из I или на те, которым присваиваются значения операторами $R_1'; \ldots; R_i'$, и в действительности $R_{i+1} = S_{j_k}'$. Таким образом, повторяя T_4, можно поставить S_{j_k}' перед всеми $S_{j_1}'; \ldots; S_{j_{n-i}}'$. В результате будет блок $\mathscr{C}_{i+1}$; условия (6) и (7) легко проверяются.

Когда в условии (7) $i = n$, получаем условие (5). Итак,

$$\mathscr{B}_1 \underset{3}{\overset{*}{\Leftrightarrow}} \mathscr{B}_1' \underset{4}{\overset{*}{\Leftrightarrow}} \mathscr{B}_2' \underset{3}{\overset{*}{\Leftrightarrow}} \mathscr{B}_2$$

откуда $\mathscr{B}_1 \underset{3,4}{\overset{*}{\Leftrightarrow}} \mathscr{B}_2$. □

Следствие. *Если* $D(\mathscr{B}_1) = D(\mathscr{B}_2)$, *то* $\mathscr{B}_1 \equiv \mathscr{B}_2$.

Доказательство. Следует непосредственно из теорем 11.1 и 11.2. □

В силу этого следствия граф можно снабдить значением, а именно значением любого блока, имеющего данный граф.

Пример 11.8. Рассмотрим два блока $\mathscr{B}_1 = (P_1, \{A, B\}, \{F\})$ и $\mathscr{B}_2 = (P_2, \{A, B\}, \{F\})$, множества P_1 и P_2 для них приведены

Таблица 11.1

P_1	P_2
$C \leftarrow A * A$ $D \leftarrow B * B$ $E \leftarrow C - D$ $F \leftarrow C + D$ $F \leftarrow E / F$	$C \leftarrow B * B$ $D \leftarrow A * A$ $E \leftarrow D + C$ $C \leftarrow D - C$ $F \leftarrow C / E$

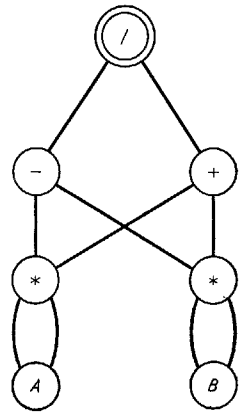


Рис. 11.2. Граф для $\mathscr{B}_1$ и $\mathscr{B}_2$.

в табл. 11.1. Блоки $\mathscr{B}_1$ и $\mathscr{B}_2$ имеют один и тот же граф, изображенный на рис. 11.2. С помощью T_3 можно отобразить $\mathscr{B}_1$ и $\mathscr{B}_2$ в открытые блоки $\mathscr{B}_1' = (P_1', \{A, B\}, \{X_5\})$ и $\mathscr{B}_2' = (P_2', \{A, B\}, \{X_5\})$, при этом будет удовлетворяться условие (3) из доказательства теоремы 11.2. P_1' и P_2' приведены в табл. 11.2. Затем, начиная с блока $\mathscr{C}_0$, у которого P_1' — список операторов, легко построить блоки $\mathscr{C}_1$, $\mathscr{C}_2$, $\mathscr{C}_3$, $\mathscr{C}_4$ и $\mathscr{C}_5$, упомянутые в доказательстве теоремы 11.2. Блок $\mathscr{C}_1$ получается применением преобразования T_4, перемещающего второй оператор на первое место (см. табл. 11.3). Далее, $\mathscr{C}_2 = \mathscr{C}_1$. Блок $\mathscr{C}_3$ строится из $\mathscr{C}_2$ с помощью преобразования T_4, ставящего четвертый оператор перед третьим (табл. 11.3); блоки $\mathscr{C}_4$ и $\mathscr{C}_5$ совпадают с $\mathscr{C}_3$. □

Таблица 11.2

P_1'	P_2'
$X_1 \leftarrow A * A$ $X_2 \leftarrow B * B$ $X_3 \leftarrow X_1 - X_2$ $X_4 \leftarrow X_1 + X_2$ $X_5 \leftarrow X_3 / X_4$	$X_2 \leftarrow B * B$ $X_1 \leftarrow A * A$ $X_4 \leftarrow X_1 + X_2$ $X_3 \leftarrow X_1 - X_2$ $X_5 \leftarrow X_3 / X_4$

Таблица 11.3

$\mathscr{C}_1$	$\mathscr{C}_3$
$X_2 \leftarrow B * B$ $X_1 \leftarrow A * A$ $X_3 \leftarrow X_1 - X_2$ $X_4 \leftarrow X_1 \mid X_2$ $X_5 \leftarrow X_3 / X_4$	$X_2 \leftarrow B * B$ $X_1 \leftarrow A * A$ $X_4 \leftarrow X_1 + X_2$ $X_3 \leftarrow X_1 - X_2$ $X_5 \leftarrow X_3 / X_4$

11.1.4. Критерий эквивалентности блоков

Покажем теперь, что $\mathscr{B}_1 \equiv \mathscr{B}_2$ тогда и только тогда, когда $\mathscr{B}_1 \overset{*}{\underset{1,2,3,4}{\Longleftrightarrow}} \mathscr{B}_2$. На самом деле справедливо более сильное утверждение, а именно: $\mathscr{B}_1 \equiv \mathscr{B}_2$ тогда и только тогда, когда $\mathscr{B}_1 \overset{*}{\underset{1,2}{\Leftrightarrow}} \mathscr{B}_2$. Иными словами, преобразований T_1 и T_2 достаточно, чтобы отобразить любой блок в любой эквивалентный ему. Доказательство этого более сильного утверждения предлагаем в качестве упражнения (упр. 11.9 и 11.10).

Определение. Блок $\mathscr{B}$ называется *приведенным*, если не существует такого блока $\mathscr{B}'$, что $\mathscr{B} \Rightarrow_{1,2} \mathscr{B}'$.

Приведенный блок не содержит ни бесполезных операторов, ни избыточных вычислений. Если дан блок $\mathscr{B}$, то можно найти эквивалентный ему приведенный блок, повторно применяя T_1 и T_2. Поскольку каждое применение T_1 или T_2 сокращает длину блока, в конце концов мы должны прийти к приведенному блоку. Наша цель заключается в том, чтобы показать, что для приведенных блоков $\mathscr{B}_1 \equiv \mathscr{B}_2$ тогда и только тогда, когда $D(\mathscr{B}_1) = D(\mathscr{B}_2)$. Таким образом, если дан блок $\mathscr{B}$, то можно найти единственный граф, соответствующий всем приведенным блокам, получаемым из $\mathscr{B}$, независимо от той конкретной последовательности преобразований T_1 и T_2, в результате которой был получен приведенный блок. Какова бы ни была машинная модель, нахождение этого графа — важный шаг „оптимизации" блока.

Определение. Пусть $P = S_1; \ldots; S_n$ — список операторов блока. Обозначим через $E(P)$ множество выражений, вычисляемых в P. Формально $E(P) = \{v_t(A) \mid S_t$ осуществляет присваивание переменной A, $1 \leqslant t \leqslant n\}$.

Выражение η вычисляется в P k раз, если существуют ровно k таких различных значений t, что $v_t(A) = \eta$ и S_t присваивает значение A.

Лемма 11.3. *Если $\mathscr{B} = (P, I, U)$ — приведенный блок, то P не вычисляет никакого выражения более одного раза.*

Доказательство. Если два оператора вычисляют одно и то же выражение, находим „первое" вхождение выражения, вычисляемого дважды. Иными словами, если S_i и S_j, $i < j$, вычисляют одно и то же выражение η, будем называть (i, j) *первым* вхождением, если для всех пар S_k и S_l, $k < l$, вычисляющих то же самое выражение, либо $i < k$, либо одновременно $i = k$ и $j < l$. В качестве упражнения предлагаем показать, что к S_i и S_j можно применить в прямом направлении преобразование T_2, а это противоречит предположению о приведенности P. □

Лемма 11.4. *Если $\mathscr{B}_1 = (P_1, I_1, U_1)$ и $\mathscr{B}_2 = (P_2, I_2, U_2)$ — эквивалентные приведенные блоки, то $E(P_1) = E(P_2)$.*

Доказательство. Если $E(P_1) \neq E(P_2)$, то без потери общности можно считать, что η — последнее вычисляемое выражение в $E(P_1) - E(P_2)$. Поскольку $v(\mathscr{B}_1) = v(\mathscr{B}_2)$ и каждое выражение единственным образом можно разбить на подвыражения [1]), то η не является подвыражением никакого выражения из $v(\mathscr{B}_1)$. Таким образом, оператор, вычисляющий η в P_1, бесполезен, и его можно удалить с помощью преобразования T_1, что противоречит предположению о приведенности $\mathscr{B}_1$. Подробное доказательство оставляем в качестве упражнения. □

Теорема 11.3. *Если $\mathscr{B}_1$ и $\mathscr{B}_2$ — два приведенных блока, то $\mathscr{B}_1 \equiv \mathscr{B}_2$ тогда и только тогда, когда $D(\mathscr{B}_1) = D(\mathscr{B}_2)$.*

Доказательство. Достаточность условия — частный случай следствия теоремы 11.2. Докажем необходимость. Пусть $\mathscr{B}_1 \equiv \mathscr{B}_2$. Из предыдущих двух лемм вытекает, что существует

[1]) Подвыражение — это префиксное выражение, являющееся операндом другого префиксного выражения. — *Прим. перев.*

взаимно однозначное соответствие между операторами из $\mathcal{B}_1$ и $\mathcal{B}_2$, вычисляющими одни и те же выражения.

Предположим, что $D(\mathcal{B}_1) \neq D(\mathcal{B}_2)$. Попытаемся „сопоставить" вершины в $D(\mathcal{B}_1)$ и в $D(\mathcal{B}_2)$ настолько „высоко" по графу, насколько это возможно. Ясно, что листья двух графов можно сопоставить — в противном случае входные множества блоков $\mathcal{B}_1$ и $\mathcal{B}_2$ были бы различны. Затем можно найти входную переменную одного из блоков, на которую не было ссылки, и, применяя T_1, удалить ее. Поскольку $\mathcal{B}_1$ и $\mathcal{B}_2$ приведены, приходим к противоречию.

Продолжаем сопоставлять вершины; если некоторая вершина в $D(\mathcal{B}_1)$ имеет ту же метку (операцию), что и какая-то вершина в $D(\mathcal{B}_2)$, если из них выходит одинаковое количество дуг и соответствующие дуги (начиная слева) указывают на сопоставленные вершины, то рассматриваемые две вершины можно сопоставить. Если, продолжая в том же духе, мы сопоставим все вершины из $D(\mathcal{B}_1)$ и $D(\mathcal{B}_2)$, то эти графы совпадут.

В противном случае мы придем к некоторой вершине в $D(\mathcal{B}_1)$ или $D(\mathcal{B}_2)$, которой не соответствует никакая вершина в другом графе. Без потери общности можно считать, что эта вершина принадлежит $D(\mathcal{B}_1)$ и что обнаружена самая „нижняя" такая вершина, т. е. вершина, каждая выходящая дуга которой указывает на сопоставленную вершину. Пусть этой вершиной будет n_1. Можно видеть, что сопоставленные вершины в $D(\mathcal{B}_1)$ и $D(\mathcal{B}_2)$ порождаются операторами в $\mathcal{B}_1$ и $\mathcal{B}_2$, вычисляющими одни и те же выражения. Это легко показать простой индукцией по порядку сопоставления.

Однако по лемме 11.3 никакую вершину нельзя сопоставить с двумя вершинами другого графа. По лемме 11.4 в $D(\mathcal{B}_2)$ найдется вершина n_2, порождаемая оператором из $\mathcal{B}_2$, вычисляющим то же самое выражение, что и оператор из $\mathcal{B}_1$, порождающий n_1. Поскольку „разбор" выражений определен однозначно, прямые потомки вершин n_1 и n_2 сопоставлены. Это следует из предположения, что вершина n_1 находится в $D(\mathcal{B}_1)$ настолько „низко", насколько возможно. Таким образом, вершины n_1 и n_2 можно сопоставить вопреки предположению. Следовательно, $D(\mathcal{B}_1) = D(\mathcal{B}_2)$. □

Следствие. *Все приведенные блоки, эквивалентные данному, имеют один и тот же граф.* □

Соберем теперь вместе доказанные выше результаты и покажем, что четырех описанных преобразований достаточно для того, чтобы превратить блок в любой ему эквивалентный.

Теорема 11.4. $\mathcal{B}_1 \equiv \mathcal{B}_2$ *тогда и только тогда, когда* $\mathcal{B}_1 \overset{*}{\underset{1,2,3,4}{\Longleftrightarrow}} \mathcal{B}_2$.

Доказательство. Достаточность условия — это следствие теоремы 11.1. Обратно, предположим, что $\mathcal{B}_1 \equiv \mathcal{B}_2$. Тогда существуют такие приведенные блоки $\mathcal{B}_1'$ и $\mathcal{B}_2'$, что $\mathcal{B}_1 \overset{*}{\underset{1,2}{\Leftrightarrow}} \mathcal{B}_1'$ и $\mathcal{B}_2 \overset{*}{\underset{1,2}{\Leftrightarrow}} \mathcal{B}_2'$. Согласно следствию теоремы 11.1, $\mathcal{B}_1 \equiv \mathcal{B}_1'$ и $\mathcal{B}_2 \equiv \mathcal{B}_2'$. Таким образом, $\mathcal{B}_1' \equiv \mathcal{B}_2'$. По теореме 11.3 $D(\mathcal{B}_1') = D(\mathcal{B}_2')$. По теореме 11.2 $\mathcal{B}_1' \overset{*}{\underset{3,4}{\Leftrightarrow}} \mathcal{B}_2'$. Следовательно, $\mathcal{B}_1 \overset{*}{\underset{1,2,3,4}{\Longleftrightarrow}} \mathcal{B}_2$. □

11.1.5. Оптимизация блоков

Займемся теперь преобразованием блока $\mathcal{B}$ в блок $\mathcal{B}'$, оптимальный относительно некоторой оценки блока. На практике часто встречается ситуация, изображенная на рис. 11.3. По данному блоку $\mathcal{B}$ мы хотим в конце концов получить программу

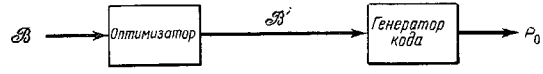


Рис. 11.3. Схема оптимизации.

на объектном языке, оптимальную относительно некоторой оценки объектных программ, такой, например, как размер программы или скорость ее выполнения. Оптимизатор применяет к блоку $\mathcal{B}$ последовательность преобразований, чтобы построить блок $\mathcal{B}'$, эквивалентный $\mathcal{B}$, из которого можно получить оптимальную программу на объектном языке. Таким образом, одна из задач заключается в нахождении оценки блоков, отражающей оценку вырабатываемой в конце концов объектной программы.

Существует несколько оценок блоков, для которых идея оптимизации даже не имеет смысла. Например, если мы будем говорить, что блок тем лучше, чем он длиннее, то вообще нет оптимального блока, эквивалентного данному. Здесь мы ограничимся оценками блоков, отражающими наиболее широко применяемые критерии качества программ на объектном языке, такие, например, как скорость выполнения или объем используемой памяти.

Определение. *Оценка* блоков — это функция, отображающая блоки в вещественные числа. Блок $\mathcal{B}$ называется *оптимальным* относительно оценки C, если $C(\mathcal{B}) \leqslant C(\mathcal{B}')$ для всех $\mathcal{B}'$, эквивалентных $\mathcal{B}$. Оценка называется *приемлемой*, если $\mathcal{B}_1 \Rightarrow_{1,2} \mathcal{B}_2$ влечет $C(\mathcal{B}_2) \leqslant C(\mathcal{B}_1)$ и любой блок имеет эквивалентный блок, оптимальный относительно C. Иными словами, оценка приемлема, если преобразования T_1 и T_2, применяемые в прямом направлении, не увеличивают оценки блока.

Лемма 11.5. *Если C — приемлемая оценка, то любой блок имеет эквивалентный приведенный блок, оптимальный относительно C.*

Доказательство. Следует непосредственно из определений. □

Лемма 11.5 устанавливает, что если дан блок $\mathcal{B}$, то поиски эквивалентного оптимального блока можно ограничить множеством приведенных блоков, эквивалентных $\mathcal{B}$. В лемме 11.6 ниже утверждается, что в результате применения к данному приведенному блоку $\mathcal{B}$ последовательности преобразований T_3 и T_4 получаются только приведенные блоки, эквивалентные $\mathcal{B}$.

Лемма 11.6. *Если $\mathcal{B}_1$ — приведенный блок и $\mathcal{B}_1 \overset{*}{\underset{3,4}{\Leftrightarrow}} \mathcal{B}_2$, то $\mathcal{B}_2$ — также приведенный блок.*

Доказательство. Упражнение. □

Наш следующий результат показывает, что если дан открытый блок, то последовательность переименований, за которой идет перестановка, можно заменить на перестановку, за которой следуют переименования.

Лемма 11.7. *Пусть $\mathcal{B}_1$ — открытый блок и $\mathcal{B}_1 \overset{*}{\underset{3}{\Leftrightarrow}} \mathcal{B}_2 \Rightarrow_4 \mathcal{B}_3$. Тогда существует такой блок $\mathcal{B}_2'$, что $\mathcal{B}_1 \Rightarrow_4 \mathcal{B}_2' \overset{*}{\underset{3}{\Leftrightarrow}} \mathcal{B}_3$.*

Доказательство. Упражнение. □

Теперь мы готовы к тому, чтобы дать общую схему оптимизации блоков в соответствии с любой приемлемой оценкой. Теорема 11.5 подводит базис для этой оптимизации.

Теорема 11.5. *Пусть $\mathcal{B}$ — любой блок. Существует блок $\mathcal{B}'$, эквивалентный $\mathcal{B}$ и такой, что если C — приемлемая оценка, то найдутся такие блоки $\mathcal{B}_1$ и $\mathcal{B}_2$, что*

(1) $\mathcal{B}' \overset{*}{\underset{4}{\Leftrightarrow}} \mathcal{B}_1$,

(2) $\mathcal{B}_1 \overset{*}{\underset{3}{\Leftrightarrow}} \mathcal{B}_2$,

(3) *$\mathcal{B}_2$ оптимален относительно C.*

Доказательство. Пусть $\mathcal{B}''$ — любой приведенный блок, эквивалентный $\mathcal{B}$. Можно преобразовать $\mathcal{B}''$ в открытый блок $\mathcal{B}'$, эквивалентный $\mathcal{B}''$, с помощью только T_3. По лемме 11.6 блок $\mathcal{B}'$ приведен и открыт.

Пусть $\mathcal{B}_2$ — оптимальный приведенный блок, эквивалентный $\mathcal{B}$. По лемме 11.5 $\mathcal{B}_2$ существует. Таким образом, $D(\mathcal{B}_2) = D(\mathcal{B}')$ в силу следствия теоремы 11.3. По теореме 11.2 $\mathcal{B}' \overset{*}{\underset{3,4}{\Leftrightarrow}} \mathcal{B}_2$. Заметим, что T_3 и T_4 взаимно „обратимы", т. е. $\mathcal{C} \Rightarrow_{3,4} \mathcal{C}'$ тогда и только тогда, когда $\mathcal{C}' \Rightarrow_{3,4} \mathcal{C}$. Следовательно, найдется такая последовательность блоков $\mathcal{C}_1, \ldots, \mathcal{C}_n$, что $\mathcal{B}' = \mathcal{C}_1$, $\mathcal{B}_2 = \mathcal{C}_n$ и $\mathcal{C}_i \Leftrightarrow_{3,4} \mathcal{C}_{i+1}$ для $1 \leqslant i < n$. Многократно применяя лемму 11.7, можно перенести все T_4 перед всеми T_3 и найти такой блок $\mathcal{B}_1$, что $\mathcal{B}' \overset{*}{\underset{4}{\Leftrightarrow}} \mathcal{B}_1 \overset{*}{\underset{3}{\Leftrightarrow}} \mathcal{B}_2$. □

Если внимательно присмотреться к теореме 11.5, можно заметить, что она разбивает весь процесс оптимизации на три стадии. Допустим, мы хотим оптимизировать данный блок $\mathcal{B}$:

(1) сначала можно исключить из $\mathcal{B}$ лишние и бесполезные вычисления и переименовать переменные с тем, чтобы получить приведенный открытый блок $\mathcal{B}'$,

(2) затем в $\mathcal{B}'$ можно переупорядочить операторы с помощью перестановки и делать это до тех пор, пока не сформируется блок $\mathcal{B}_1$, в котором операторы расположены в наилучшем порядке,

(3) наконец, в $\mathcal{B}_1$ можно переименовывать переменные до тех пор, пока не будет найден оптимальный блок $\mathcal{B}_2$.

Отметим, что шаг (1) можно выполнить эффективно (как функцию длины блока). Читателю предлагаем разработать алгоритм для шага (1), обрабатывающий блок из n операторов за время $O(n \log n)$.

Один из шагов (2) или (3) часто оказывается тривиальным. Приведем пример, показывающий, как преобразовать операторы промежуточного языка в язык ассемблера, чтобы число выполняемых команд языка ассемблера было минимальным. Мы увидим, что этот алгоритм оптимизации не нуждается в шаге (3). Дальнейшее переименование переменных не будет влиять на оценку.

Пример 11.9. Наш пример содержит несколько интересных идей, которые не затрагиваются ни в какой другой части книги. Читателю настоятельно рекомендуем подробно разобраться в этом примере. Рассмотрим машинный код, генерируемый для блоков. Вычислительная машина имеет один сумматор и следующий набор команд языка ассемблера, смысл которых мы кратко расшифруем:

(1) LOAD M. Содержимое ячейки памяти M помещается на сумматор.

(2) STORE M. Содержимое сумматора запоминается в ячейке памяти M.

(3) $\theta M_2 M_3 \ldots M_r$. В этой команде θ — имя r-местной операции. Ее первый аргумент — сумматор, второй — ячейка памяти M_2,

третий — ячейка памяти M_3 и т. д. Результат применения операции θ к своим аргументам размещается на сумматоре.

Генератор кодов должен переводить оператор вида $A \leftarrow \theta B_1 \ldots B_r$ в последовательность машинных команд

```
LOAD   B1
θ      B2, ..., Br
STORE  A
```

Однако если значение B_1 уже находится на сумматоре (т. е. перед этим был оператор, присваивающий значение B_1), то первую команду LOAD генерировать не надо. Аналогично, если значение A не требуется нигде, кроме первого аргумента следующего оператора, то заключительная команда STORE не нужна.

Оценивать оператор $A \leftarrow \theta B_1 \ldots B_n$ можно числами 1, 2 и 3. Оценка равна 3, если B_1 нет на сумматоре и в дальнейшем есть ссылка на новое значение A, отличная от первого аргумента следующего оператора (т. е. A надо запомнить). Оценка равна 1, если B_1 уже на сумматоре и нет ссылки на A, отличной от первого аргумента следующего оператора. В остальных случаях оценка равна 2.

Заметим, что такой способ оценки полностью исчерпывает все случаи, заслуживающие рассмотрения. Чтобы показать, что он правильно отражает число команд, требующихся для выполнения блока на нашей машине, надо сначала точно определить эффект последовательности команд языка ассемблера. Если это сделано должным образом, то каждую программу на языке ассемблера можно сопоставить с блоком на промежуточном языке путем отождествления команд языка ассемблера типа (3), т. е. операций, с операторами блока. Все эти детали мы предлагаем в качестве упражнения. В настоящем примере мы будем пользоваться оценкой блоков в том виде, как мы ее определили.

Рассмотрим блок $\mathcal{B}_1 = (P, \{A, B, C\}, \{F, G\})$, который мог бы получиться, скажем, из операторов Фортрана

$$F = (A + B) * (A - B)$$
$$G = (A - B) * (A - C) * (B - C)$$

Список операторов в P_1 таков:

$$T \leftarrow A + B$$
$$S \leftarrow A - B$$
$$F \leftarrow T * S$$
$$T \leftarrow A - B$$
$$S \leftarrow A - C$$
$$R \leftarrow B - C$$
$$T \leftarrow T * S$$
$$G \leftarrow T * R$$

Бесполезных операторов здесь нет. Есть, однако, повторения — второй и четвертый операторы. Эту избыточность можно удалить, а затем переименовать во всех операторах переменные, которым присваиваются значения. В результате получится приведенный открытый блок $\mathcal{B}_2 = (P_2, \{A, B, C\}, \{X_3, X_7\})$. Он играет здесь роль блока $\mathcal{B}'$ в теореме 11.5. Операторами в P_2 будут

$$X_1 \leftarrow A + B$$
$$X_2 \leftarrow A - B$$
$$X_3 \leftarrow X_1 * X_2$$
$$X_4 \leftarrow A - C$$
$$X_5 \leftarrow B - C$$
$$X_6 \leftarrow X_2 * X_4$$
$$X_7 \leftarrow X_6 * X_5$$

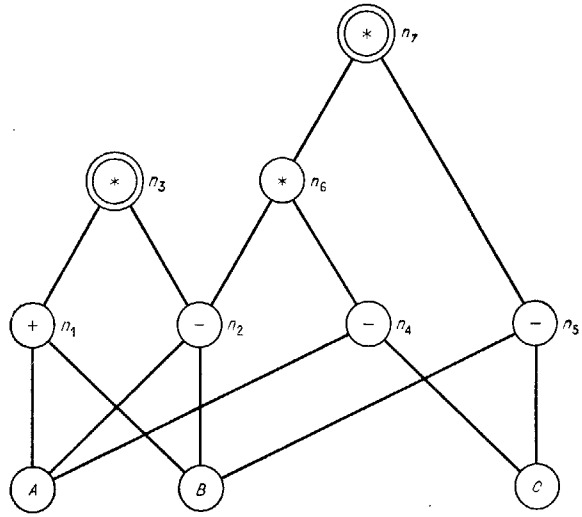


Рис. 11.4. Граф для $\mathcal{B}_2$.

Граф для $\mathcal{B}_2$ изображен на рис. 11.4. Вершина n_i соответствует оператору списка P_2, присваивающему значение переменной X_i.

Мы видим, что существует много программ, в которые можно преобразовать $\mathcal{B}_2$ с помощью только T_4. Предлагаем в качестве упражнения доказать, что их столько, сколько существует

линейных порядков, в которых выполняется частичный порядок, представленный на рис. 11.4.

Верхняя граница этого числа равна 7!, т. е. число перестановок из семи операторов. Но в действительности это число будет меньше, так как операторы из P_2 не могут располагаться в любом порядке. Например, третий оператор в P_2 всегда должен следовать за вторым, поскольку третий ссылается на переменную X_2, а второй определяет ее. Отметим, что применение T_3 может изменить имя X_2, но то же отношение будет выполняться и с новым именем.

Другая интерпретация ограничений на возможности T_4 переупорядочивать блок заключается в том, что при любом таком переупорядочении каждая вершина в $D(\mathscr{B}_2)$ будет соответствовать некоторому оператору. Оператор, соответствующий внутренней вершине n, не может предшествовать никакому оператору, соответствующему внутренней вершине, являющейся потомком вершины n.

Этот пример достаточно прост для того, чтобы просмотреть все линейные порядки в P_2, но для произвольного блока этого сделать нельзя, поскольку это связано с большими затратами времени. Для того чтобы быстро вырабатывать хорошее, но не обязательно оптимальное упорядочение, нужна некоторая эвристика Здесь мы предлагаем один такой способ. Следующий алгоритм дает линейное упорядочение вершин графа. В требуемом блоке операторы, соответствующие этим вершинам, расположены в обратном порядке.

(1) Строим список L. Вначале он пуст.

(2) Выбираем вершину n графа так, что $n \notin L$ и, если существуют входящие в n дуги, они должны выходить из вершин, уже принадлежащих L. Добавляем n к L. Если такой вершины нет, то останавливаемся.

(3) Если n_1 — последняя вершина, добавленная к L, самая левая дуга, выходящая из n_1, указывает на внутреннюю вершину n, не принадлежащую L, и все прямые предки n уже принадлежат L, то добавляем n к L и повторяем шаг (3). В противном случае переходим к шагу (2).

Например, используя граф рис. 11.4, можно начать с $L = n_3$. Согласно шагу (3), к L добавляем n_1. Затем выбираем и добавляем к L вершину n_7, а потом n_6 и n_2. В результате еще двух применений правила (2) добавляются n_4 и n_5, так что L имеет вид n_3, n_1, n_7, n_6, n_2, n_4, n_5. Вспоминая, что оператор, присваивающий значение X_i, порождает вершину n_i и что список L соответствует операторам в обратном порядке, получаем блок $\mathscr{B}_3 = (P_3, \{A, B, C\}, \{X_3, X_7\})$. Легко проверить, что $\mathscr{B}_2 \overset{*}{\underset{4}{\Leftrightarrow}} \mathscr{B}_3$.

Список операторов в P_3 таков:

$$
\begin{aligned}
X_5 &\leftarrow B - C \\
X_4 &\leftarrow A - C \\
X_2 &\leftarrow A - B \\
X_6 &\leftarrow X_2 * X_4 \\
X_7 &\leftarrow X_6 * X_5 \\
X_1 &\leftarrow A + B \\
X_3 &\leftarrow X_1 * X_2
\end{aligned}
$$

Программы на языке ассемблера, созданные из блоков $\mathscr{B}_2$ и $\mathscr{B}_3$, приведены на рис. 11.5. □

LOAD	A	LOAD	B
ADD	B	SUBTR	C
STORE	X	STORE	X_5
LOAD	A	LOAD	A
SUBTR	B	SUBTR	C
STORE	X_2	STORE	X_4
LOAD	X_1	LOAD	A
MULT	X_2	SUBTR	B
STORE	X_3	STORE	X_2
LOAD	A	MULT	X_4
SUBTR	C	MULT	X_5
STORE	X_4	STORE	X_7
LOAD	B	LOAD	A
SUBTR	C	ADD	B
STORE	X_5	MULT	X_2
LOAD	X_2	STORE	X_3
MULT	X_4		
MULT	X_5		
STORE	X_7		

а — Из $\mathscr{B}_2$ *б* — Из $\mathscr{B}_3$

Рис. 11.5. Программы на языке ассемблера.

11.1.6. Алгебраические преобразования

Известно, что во многих языках программирования некоторые операции и операнды подчиняются определенным алгебраическим законам. Учитывая эти законы, можно проводить такие улучшения программы, какие невозможно сделать с использованием лишь четырех рассмотренных выше типов преобразований.

Перечислим несколько полезных и распространенных алгебраических законов:

(1) Бинарная операция θ называется *коммутативной*, если $\alpha\theta\beta = \beta\theta\alpha$ для всех выражений α и β. Примерами коммутативных операций служат сложение и умножение целых чисел [1]).

(2) Бинарная операция θ называется *ассоциативной*, если $\alpha\theta(\beta\theta\gamma) = (\alpha\theta\beta)\theta\gamma$ для всех α, β и γ. Например, сложение ассоциативно, так как

$$\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma \text{ [2]}$$

(3) Бинарная операция θ_1 называется *дистрибутивной* относительно бинарной операции θ_2, если $\alpha\theta_1(\beta\theta_2\gamma) = (\alpha\theta_1\beta)\theta_2(\alpha\theta_1\gamma)$. Например, умножение дистрибутивно относительно сложения, так как $\alpha * (\beta + \gamma) = \alpha * \beta + \alpha * \gamma$. Предосторожности здесь те же, что и в (1) и (2).

(4) Унарная операция θ называется *идемпотентной*, если $\theta\theta\alpha = \alpha$ для всех α. Например, логическое *не* и унарный минус идемпотентны.

(5) Выражение ε называется *нейтральным* относительно (бинарной) операции θ, если $\varepsilon\theta\alpha = \alpha\theta\varepsilon = \alpha$ для всех α. Вот примеры нейтральных выражений:

(а) Константа 0 нейтральна относительно сложения. Нейтрально и любое выражение, имеющее значение 0, такое, например, как $\alpha - \alpha$, $\alpha * 0$, $(-\alpha) + \alpha$ и т. д.

(б) Константа 1 нейтральна относительно умножения.

(в) Логическая константа **истина** нейтральна относительно конъюнкции (т. е. $\alpha \wedge$ **истина** $= \alpha$ для всех α).

(г) Логическая константа **ложь** нейтральна относительно дизъюнкции (т. е. $\alpha \vee$ **ложь** $= \alpha$ для всех α).

Если $\mathcal{A}$ — множество алгебраических законов, будем говорить, что *выражение* α *эквивалентно выражению* β *относительно* $\mathcal{A}$ (и писать $\alpha \equiv_{\mathcal{A}} \beta$), если α можно преобразовать в β, применяя только алгебраические законы из $\mathcal{A}$.

Пример 11.10. Рассмотрим выражение

$$A * (B * C) + (B * A) * D + A * E$$

С помощью ассоциативного закона умножения можно записать $A * (B * C)$ в виде $(A * B) * C$. С помощью коммутативного закона

[1]) Однако необходимо соблюдать осторожность, если операнды коммутативной операции — функции с побочным эффектом. Например, $f(x) + g(y)$ может отличаться от $g(y) + f(x)$, если функция f меняет значение y.

[2]) Это преобразование также надо применять с осторожностью. Например, пусть x значительно больше y и $z = -x$, а вычисления осуществляются с плавающей точкой. Тогда $(y + x) + z$ может в результате дать 0, в то время как $y + (x + z)$ дает y.

умножения можно записать $B * A$ в виде $A * B$. Применяя дистрибутивный закон, все выражение можно записать как

$$(A * B) * (C + D) + A * E$$

Наконец, применяя ассоциативный закон к первому слагаемому, затем дистрибутивный закон, можно записать выражение как

$$A * (B * (C + D) + E)$$

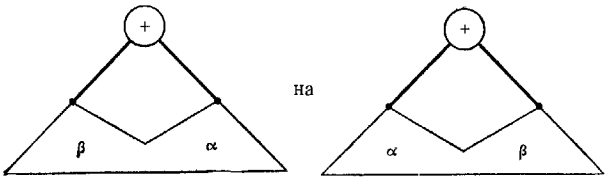
Таким образом, это выражение эквивалентно исходному относительно ассоциативного, коммутативного и дистрибутивного законов умножения и сложения. В то же время последнее выражение вычисляется только двумя сложениями и двумя умножениями, в то время как для исходного требовалось пять умножений и два сложения. □

Определение эквивалентности относительно множества алгебраических законов $\mathcal{A}$ можно распространить и на блоки. Будем говорить, что *блоки* $\mathcal{B}_1$ *и* $\mathcal{B}_2$ *эквивалентны относительно* $\mathcal{A}$ (и писать $\mathcal{B}_1 \equiv_{\mathcal{A}} \mathcal{B}_2$), если для любого выражения $\alpha \in v(\mathcal{B}_1)$ существует такое выражение $\beta \in v(\mathcal{B}_2)$, что $\alpha \equiv_{\mathcal{A}} \beta$, и обратно.

Каждый алгебраический закон приводит к соответствующим преобразованиям блоков (и графов).

Пример 11.11. Если сложение коммутативно, то преобразование блоков, соответствующее этому алгебраическому закону, позволяет заменить в блоке оператор вида $X \leftarrow A + B$ на оператор $X \leftarrow B + A$.

Соответствующее преобразование графа позволяет заменить всюду в графе структуру



□

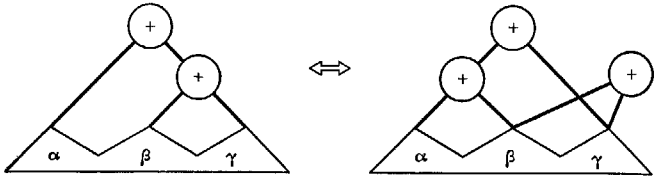
Пример 11.12. Рассмотрим преобразование блоков, соответствующее ассоциативному закону сложения. Последовательность из двух операторов вида

$$X \leftarrow B + C$$
$$Y \leftarrow A + X$$

можно заменить на три оператора

$$X \leftarrow B + C$$
$$X' \leftarrow A + B$$
$$Y \leftarrow X' + C$$

где X' — новая переменная. Это преобразование имеет следующий аналог на графах:



Отметим, что мы сохранили оператор $X \leftarrow B+C$, поскольку на переменную X может быть ссылка из последующего оператора. Однако, если оператор $X \leftarrow B+C$ после этого преобразования становится бесполезным, его можно удалить, применяя T_1. Если, кроме того, можно удалить $X' \leftarrow A+B$ с помощью T_2, то использование ассоциативного закона принесло выгоду. □

Если дан конечный набор алгебраических законов и соответствующие преобразования блоков, то для нахождения оптимального блока, эквивалентного данному, желательно было бы применить их вместе с четырьмя топологическими преобразованиями, описанными в разд. 11.1.2. К сожалению, для конкретного набора алгебраических законов может не быть эффективного способа применения этих преобразований для нахождения оптимального блока.

Обычный подход к решению этого вопроса заключается в том, что алгебраические преобразования применяют в ограниченном виде, надеясь сделать возможно больше „упрощений" выражений и выработать возможно большее число общих подвыражений. В типичной схеме $\beta\theta\alpha$ заменяется на $\alpha\theta\beta$, если θ — коммутативная бинарная операция, а α предшествует β при некотором лексикографическом упорядочении имен переменных. Если θ — ассоциативная и коммутативная бинарная операция, то $\alpha_1\theta\alpha_2\theta \ldots \theta\alpha_n$ можно преобразовать, располагая имена $\alpha_1, \ldots, \alpha_n$ в лексикографическом порядке и группируя их слева направо.

Закончим этот раздел примером, иллюстрирующим возможный эффект алгебраических преобразований блоков.

Пример 11.13. Рассмотрим блок $\mathcal{B}=(P, I, \{Y\})$, где $I = \{A, B, C, D, F\}$ и P — последовательность операторов

$$X_1 \leftarrow B-C$$
$$X_2 \leftarrow A*X_1$$
$$X_3 \leftarrow E*F$$
$$X_4 \leftarrow D*X_3$$
$$Y \leftarrow X_2*X_4$$

Блок $\mathcal{B}$ вычисляет выражение

$$Y=(A*(B-C))*(D*(E*F))$$

Граф для $\mathcal{B}$ приведен на рис. 11.6.

Предположим, что для $\mathcal{B}$ генерируется программа на языке ассемблера, в которой участвуют команды языка ассемблера и

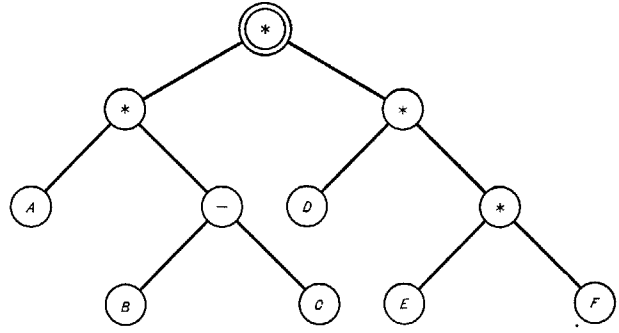


Рис. 11.6. Граф для $\mathcal{B}$.

функция оценки, описанные в примере 11.9. Если генерировать программу на языке ассемблера прямо из $\mathcal{B}$, то результирующая программа будет иметь оценку 15.

Предположим теперь, что умножение коммутативно и ассоциативно и мы хотим найти оптимальный блок для $\mathcal{B}$, эквивалентный $\mathcal{B}$ относительно ассоциативного и коммутативного законов умножения. Применим к $\mathcal{B}$ алгебраические преобразования, соответствующие этим двум законам. Мы хотим получить последовательность операторов, в которой промежуточные результаты можно без запоминания немедленно использовать последующими командами.

Предполагая, что умножение ассоциативно, можно заменить в $\mathcal{B}$ два оператора

$$X_3 \leftarrow E*F$$
$$X_4 \leftarrow D*X_3$$

на три оператора

$$X_3 \leftarrow E*F$$
$$X_3' \leftarrow D*E$$
$$X_4 \leftarrow X_3'*F$$

Теперь оператор $X_3 \leftarrow E*F$ бесполезен, и его можно удалить

преобразованием T_1. Затем с помощью ассоциативного преобразования можно заменить операторы

$$X_4 \leftarrow X_3' * F$$
$$Y \leftarrow X_2 * X_4$$

на

$$X_4 \leftarrow X_3' * F$$
$$X_4' \leftarrow X_2 * X_3'$$
$$Y \leftarrow X_4' * F$$

Оператор $X_4 \leftarrow X_3' * F$ теперь бесполезен, и его можно удалить. К этому моменту у нас пять операторов

$$X_1 \leftarrow B - C$$
$$X_2 \leftarrow A * X_1$$
$$X_3' \leftarrow D * E$$
$$X_4' \leftarrow X_2 * X_3'$$
$$Y \leftarrow X_4' * F$$

Если применить ассоциативное преобразование еще раз к третьему и четвертому операторам, получим (после исключения бесполезных операторов) блок

$$X_1 \leftarrow B - C$$
$$X_2 \leftarrow A * X_1$$
$$X_3'' \leftarrow X_2 * D$$
$$X_4' \leftarrow X_3'' * E$$
$$Y \leftarrow X_4' * F$$

Наконец, если предположить, что умножение коммутативно, можно переставить операнды второго оператора и получить блок $\mathscr{B}'$:

$$X_1 \leftarrow B - C$$
$$X_2 \leftarrow X_1 * A$$
$$X_3'' \leftarrow X_2 * D$$
$$X_4' \leftarrow X_3'' * E$$
$$Y \leftarrow X_4' * F$$

Граф для $\mathscr{B}'$ приведен на рис. 11.7. Блок $\mathscr{B}'$ имеет оценку 7, самую нижнюю возможную оценку для блока, эквивалентного $\mathscr{B}'$. В следующем разделе мы изложим систематический способ оптимизации арифметических выражений с использованием ассоциативного и коммутативного алгебраических законов. □

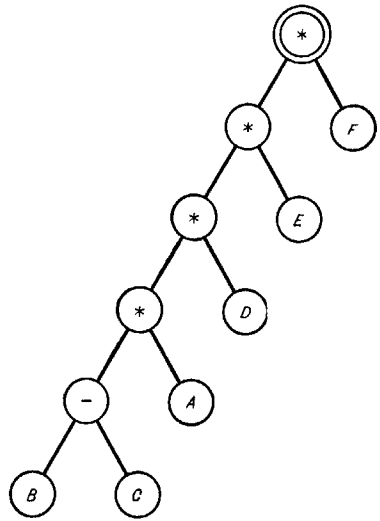


Рис. 11.7. Граф для $\mathscr{B}'$.

УПРАЖНЕНИЯ

11.1.1. Пусть $\mathscr{B} = (P, \{A, B, C\}, \{F, G\})$ — блок, в котором P состоит из

$$T \leftarrow A + B$$
$$R \leftarrow A * T$$
$$S \leftarrow B + C$$
$$F \leftarrow R * S$$
$$T \leftarrow A * A$$
$$R \leftarrow A + B$$
$$S \leftarrow A * R$$
$$G \leftarrow S + T$$

(а) Что представляет собой $v(\mathscr{B})$?

(б) Укажите область действия каждого оператора из P.

(в) Имеет ли P бесполезные операторы?

(г) Преобразование T_2 применимо к первому и шестому операторам. Какие значения может принимать D (см. определение D в разд. 11.1.2) при этом применении T_2?

(д) Изобразите граф для $\mathcal{B}$.

(е) Найдите для $\mathcal{B}$ эквивалентный приведенный блок.

(ж) Сколько существует эквивалентных приведенных блоков для $\mathcal{B}$, не считая тех, которые получаются переименованием? (Точнее, пусть $\mathcal{B}'$ — открытый приведенный блок, эквивалентный $\mathcal{B}$. Какова мощность множества $\left\{\mathcal{B}'' \mid \mathcal{B}'' \overset{*}{\underset{4}{\Leftrightarrow}} \mathcal{B}'\right\}$?)

(з) Найдите для $\mathcal{B}$ эквивалентный блок, оптимальный относительно оценки из примера 11.9.

11.1.2. Докажите, что преобразования T_1, T_3 и T_4 сохраняют эквивалентность блоков (т. е. $\mathcal{B} \Rightarrow_i \mathcal{B}'$ влечет $v(\mathcal{B}) = v(\mathcal{B}')$ для $i = 1, 3$ и 4).

11.1.3. Докажите, что если в преобразовании T_2 (разд. 11.1.2) D — любой символ, не употребляемый в P, то $v(\mathcal{B}) = v(\mathcal{B}')$.

***11.1.4.** Дайте алгоритм определения множества допустимых имен для D в преобразовании T_2.

11.1.5. Докажите, что алгоритм, изложенный после примера 11.3, удаляет из блока все бесполезные операторы и входные переменные. Покажите, что число шагов, требуемых для выполнения алгоритма, линейно зависит от числа операторов блока.

***11.1.6.** Приведите алгоритм удаления из блока всех лишних вычислений (преобразование T_2) за время $0(n \log n)$, где n — число операторов блока[1]). (Это аналогично минимизации автоматов с конечным числом состояний с помощью алгоритма 2.6.)

11.1.7. Приведите алгоритм определения области действия оператора в блоке.

11.1.8. Определите преобразования графов, аналогичные преобразованиям блоков $T_1 - T_4$.

Упр. 11.1.9 и 11.1.10 показывают, что $\mathcal{B}_1 \overset{*}{\underset{1,2,3,4}{\Longleftrightarrow}} \mathcal{B}_2$ влечет $\mathcal{B}_1 \overset{*}{\underset{1,2}{\Longleftrightarrow}} \mathcal{B}_2$.

***11.1.9.** Покажите, что если $\mathcal{B}_1 \Rightarrow_3 \mathcal{B}_2$, то существует такой блок $\mathcal{B}_3$, что $\mathcal{B}_3 \Rightarrow_2 \mathcal{B}_2$ и $\mathcal{B}_3 \Rightarrow_1 \mathcal{B}_1$. Таким образом, преобразование T_3 можно выполнить одним применением T_1 в обратном порядке с последующим применением T_2.

[1]) Не надо забывать, что число возможных имен переменных бесконечно. Поэтому могут пригодиться методы организации информации, аналогичные упомянутым в разд. 10.1.

***11.1.10.** Покажите, что если $\mathcal{B}_1 \Rightarrow_4 \mathcal{B}_2$, то существует такой блок $\mathcal{B}_3$, что $\mathcal{B}_3 \Rightarrow_2 \mathcal{B}_2$ и $\mathcal{B}_3 \Rightarrow_1 \mathcal{B}_1$.

Определение. Множество S преобразований блоков называется *полным*, если $v(\mathcal{B}_1) = v(\mathcal{B}_2)$ влечет $\mathcal{B}_1 \overset{*}{\underset{S}{\Leftrightarrow}} \mathcal{B}_2$. Множество S называется *минимально полным*, если никакое его собственное подмножество не полно.

Упр. 11.1.9 и 11.1.10 показывают, что множество $\{T_1, T_2\}$ полно. Следующие два упражнения показывают, что $\{T_1, T_2\}$ минимально полно.

***11.1.11.** Покажите, что с помощью только преобразований T_1, T_3 и T_4 нельзя преобразовать блок $\mathcal{B} = (P, \{A, B\}, \{C, D\})$ в $\mathcal{B}' = (P', \{A, B\}, \{C, D\})$, где P и P' таковы:

P	P'
$E \leftarrow A + B$ $D \leftarrow E * E$ $C \leftarrow A + B$	$C \leftarrow A + B$ $D \leftarrow C * C$

Следовательно, множество $\{T_1, T_3, T_4\}$ не полно, и потому $\{T_1\}$ не полно.

***11.1.12.** Покажите, что с помощью только преобразований T_2, T_3 и T_4 нельзя преобразовать блок $\mathcal{B} = (P, \{A, B\}, \{C\})$ в $\mathcal{B}' = (P', \{A, B\}, \{C\})$, где P и P' таковы:

P	P'
$C \leftarrow A * B$ $C \leftarrow A + B$	$C \leftarrow A + B$

11.1.13. Приведите алгоритм, выясняющий, эквивалентны ли два данных блока.

11.1.14. Пусть $P = S_1; S_r; \ldots; S_n$ — последовательность операторов присваивания, а I — множество входных переменных. Дайте алгоритм нахождения всех неопределенных (т. е. таких, на которые есть ссылка до присваивания) переменных из P.

***11.1.15.** Пусть в качестве операторов блока, помимо данных выше в определении блока, допускаются также операторы вида

$A \leftarrow B$, имеющие очевидный смысл. Найдите для таких блоков полное множество преобразований.

**11.1.16. Предположим, что сложение коммутативно. Пусть T_5—преобразование, заменяющее оператор $A \leftarrow B+C$ на $A \leftarrow C+B$. Покажите, что T_5 вместе с T_1 и T_2 преобразуют два блока друг в друга тогда и только тогда, когда они эквивалентны относительно закона коммутативности сложения.

**11.1.17. Предположим, что сложение ассоциативно. Пусть T_6—преобразование, заменяющее операторы $X \leftarrow A+B; Y \leftarrow X+C$ на $X \leftarrow A+B; X' \leftarrow B+C; Y \leftarrow A+X'$ или операторы $X \leftarrow B+C; Y \leftarrow A+X$ на $X \leftarrow B+C; X' \leftarrow A+B; Y \leftarrow X'+C$, где X'—новая переменная. Покажите, что T_6, T_1 и T_2 преобразуют два блока друг в друга тогда и только тогда, когда они эквивалентны относительно закона ассоциативности сложения.

11.1.18. Что представляет собой преобразование блоков, соответствующее закону дистрибутивности умножения относительно сложения? Что представляет собой соответствующее преобразование графов?

**11.1.19. Покажите, что существуют алгебраические законы, для которых рекурсивно неразрешима проблема эквивалентности двух выражений.

Определение. Говорят, что алгебраический закон *сохраняет операнды*, если при одном его применении операнды не порождаются и не исчезают. Например, коммутативный и ассоциативный законы сохраняют операнды, а дистрибутивный нет.

Говорят, что алгебраический закон *сохраняет операции*, если одно его применение не влияет на число операций. Алгебраический закон $\theta\theta\alpha=\alpha$ (идемпотентность) не сохраняет операции, а закон $(\alpha-\beta)-\gamma=\alpha-(\beta+\gamma)$ сохраняет.

Числа внутренних вершин и листьев в графе, связанном с блоком, сохраняются при применении к блоку преобразований, соответствующих алгебраическим законам, сохраняющим операции и операнды.

*11.1.20. Покажите, что проблема эквивалентности двух блоков относительно алгебраических законов, сохраняющих операции и операнды, разрешима.

*11.1.21. Обобщите теорему 11.5 так, чтобы можно было оптимизировать блоки, применяя как топологические преобразования из разд. 11.1.2, так и произвольный набор алгебраических преобразований, сохраняющих операции и операнды.

*11.1.22. Рассмотрим блоки, в которых переменные могут представлять одномерные массивы. Рассмотрим операторы присваивания вида

(1) $A(X) \leftarrow B$,
(2) $B \leftarrow A(X)$,

где A—одномерный массив, а B и X—скаляры. Используя тот факт, что A—массив, найдите преобразования, применимые к блокам, в которых каждый оператор имеет вид (1), (2) или $B \leftarrow \theta C_1 \ldots C_r$, где $B, C_1, \ldots, C_r$—скаляры.

11.1.23. Докажите лемму 11.1.

11.1.24. Докажите лемму 11.2.

11.1.25. Закончите доказательство леммы 11.3.

11.1.26. Закончите доказательство леммы 11.4.

*11.1.27. Дайте пример такой оценки C, что $\mathcal{B}_1 \Rightarrow_{1,2} \mathcal{B}_2$ влечет $C(\mathcal{B}_2) \leqslant C(\mathcal{B}_1)$, но не для каждого блока существует блок, оптимальный относительно C.

11.1.28. Докажите лемму 11.6.

11.1.29. Покажите, что если $\mathcal{B}_1$—открытый блок и $\mathcal{B}_1 \Rightarrow_3 \mathcal{B}_2 \Rightarrow_4 \mathcal{B}_3$, то существует такой блок $\mathcal{B}$, что $\mathcal{B}_1 \Rightarrow_4 \mathcal{B} \Rightarrow_3 \mathcal{B}_3$.

11.1.30. Докажите лемму 11.7. *Указание*: Воспользуйтесь упр. 11.1.29.

*11.1.31. Предположим, что у нас есть машина с N регистрами, причем в этих регистрах можно разместить некоторые или все аргументы операции, а также ее результат. Покажите, что выходные значения блока можно вычислять на такой машине, не прибегая к командам запоминания (результаты помещаются в регистры), тогда и только тогда, когда для этого блока существует эквивалентный блок, в котором в командах встречается не более N разных имен переменных.

*11.1.32. Покажите, что если к данному блоку $\mathcal{B}$ в любом порядке применять преобразования T_1 и T_2, пока не получится приведенный блок, то результатом будет всегда один и тот же блок (с точностью до переименования).

Проблемы для исследования

11.1.33. Используя оценку из примера 11.9 или какую-нибудь другую интересную оценку, постройте быстрый алгоритм нахождения оптимального блока, эквивалентного данному.

11.1.34. Укажите набор алгебраических преобразований, полезных при оптимизации широкого класса программ. Разработайте эффективные методы применения этих преобразований.

Упражнения на программирование

11.1.35. Используя подходящее представление графов, реализуйте преобразования T_1 и T_2, описанные в настоящем разделе.

11.1.36. Реализуйте эвристику, предложенную в примере 11.9, для „оптимизации" кода машины с одним сумматором.

Замечания по литературе

Материал этого раздела излагается в соответствии с работой Ахо и Ульмана [1972е]. Игараси [1968] дает преобразования аналогичных блоков, в которых допускаются операторы $A \leftarrow B$ и считаются существенными имена выходных переменных. Де Бэккер [1971] рассматривает блоки, в которых все операторы имеют вид $A \leftarrow B$. Браха [1972] изучает линейные блоки с переходами вперед.

Ричардсон [1968] доказал, что не существует алгоритма „упрощения" выражений, когда последние состоят из довольно простых операторов. В его работе можно найти ответ на упр. 11.1.19. Кавнесс [1970] также рассматривает классы алгебраических законов, для которых проблема эквивалентности блоков неразрешима.

Флойд [1961а] и Брюер [1969] представили алгоритмы нахождения общих подвыражений в линейных участках при выполнении определенных алгебраических законов. Ахо и Ульман [1972ж] исследовали эквивалентность блоков со структурированными переменными, как и в упр. 11.1.22. Некоторые методы, полезные для упр. 11.1.32, содержатся в работе Ахо и др. [1972] [1]).

11.2. АРИФМЕТИЧЕСКИЕ ВЫРАЖЕНИЯ

Займемся теперь построением генератора кода, вырабатывающего для блоков код на языке ассемблера. Входом для генератора кода служит блок, состоящий из последовательности операторов присваивания. Выходом служит эквивалентная программа на языке ассемблера.

Нам бы хотелось, чтобы результирующая программа на языке ассемблера была хорошей относительно некоторой оценки, такой, например, как число команд языка ассемблера или число обращений к памяти. К сожалению, как отмечено в предыдущем разделе, эффективный алгоритм, который давал бы оптимальный код на языке ассемблера, не известен даже для простой машины с одним сумматором, такой, как в упр. 11.9.

[1]) Авторы не упоминают работ советских авторов по методам оптимизации программ, хотя советскими учеными внесен значительный вклад в эту область автоматизации программирования. В работах Лаврова [1961] и Ершова [1962] применяются операторные схемы для решения задачи глобальной экономии памяти. В работе Ершова и др. [1965] эта теория используется для экономии памяти в АЛЬФА-трансляторе. Кроме того, в этой же работе вводится понятие линейного участка, употребляемого для построения операторной схемы.

См. также примечания переводчика к замечаниям по литературе в разд. 11.2—11.4. — *Прим. перев.*

Здесь мы приведем эффективный алгоритм генерации кода на языке ассемблера для ограниченного класса блоков, а именно блоков, представляющих собой одно выражение без одинаковых операндов. Для этого класса блоков алгоритм будет генерировать код на языке ассемблера, оптимальный относительно различных оценок, включая такие, как длина программы и число используемых сумматоров.

Предположение об отсутствии одинаковых операндов, разумеется, не реально, но часто оно бывает хорошим первым приближением. Кроме того, это предположение весьма удобно, если мы собираемся генерировать код, применяя синтаксически управляемый перевод только с синтезированными атрибутами. Наконец, проблема генерации оптимального кода для выражений с одной только парой одинаковых операндов уже оказывается значительно более сложной.

Блок, представляющий одно выражение, имеет только одну выходную переменную. Например, оператор присваивания $F = Z * (X + Y)$ можно представить блоком $\mathcal{B} = (P, \{X, Y, Z\}, \{F\})$, где P состоит из

$$R \leftarrow X + Y$$
$$F \leftarrow Z * R$$

Ограничение, состоящее в том, что выражение не имеет одинаковых операндов, эквивалентно требованию, чтобы граф для этого выражения был деревом.

Для удобства будем предполагать, что все операции бинарные. Это ограничение несущественно, поскольку результаты этого раздела легко обобщаются на выражения с произвольными операциями.

Код на языке ассемблера будем генерировать для машины с N сумматорами, где $N \geqslant 1$. Оценкой будет длина программы на языке ассемблера (т. е. число команд). Алгоритм впоследствии будет расширен, и во внимание будут приняты коммутативность и ассоциативность операций.

11.2.1. Модель машины

Рассмотрим вычислительную машину с $N \geqslant 1$ универсальными сумматорами и командами четырех типов.

Определение. *Командой языка ассемблера* называется цепочка символов одного из четырех типов:

```
LOAD   M, A
STORE  A, M
OP θ   A, M, B
OP θ   A, B, C
```

В этих командах M — ячейка памяти, а A, B и C — имена сумматоров (возможно, одинаковые). OP θ — это код бинарной операции θ. Предполагается, что каждой операции θ соответствует машинная команда типа (3) или (4).

Эти команды выполняют следующие действия:

(1) LOAD M, A помещает содержимое ячейки памяти M на сумматор A.

(2) STORE A, M помещает содержимое сумматора A в ячейку памяти M.

(3) OP θ A, M, B применяет бинарную операцию θ к содержимому сумматора A и ячейки памяти M, а результат помещает на сумматор B.

(4) OP θ A, B, C применяет бинарную операцию θ к содержимому сумматоров A и B, а результат помещает на сумматор C.

Если сумматор только один, то множество команд сводится к множеству, приведенному в примере 11.9, но только теперь команды типа (4) превращаются в OP θ A, A, A. Возможности применения этой команды не используются, так что с этой точки зрения множество команд можно считать обобщением команд для одноадресной машины с одним сумматором.

Программой на языке ассемблера (или для краткости просто *программой*) будем называть последовательность команд языка ассемблера.

Если $P = I_1; I_2; \ldots; I_n$ — программа, то можно определить *значение* $v_t(R)$ *регистра* R *после команды* t (здесь под *регистром* понимается сумматор или ячейка памяти):

(1) $v_0(R)$ равно R, если R — ячейка памяти, и не определено, если R — сумматор,

(2) если I_t — это LOAD M, A, то $v_t(A) = v_{t-1}(M)$,

(3) если I_t — это STORE A, M, то $v_t(M) = v_{t-1}(A)$,

(4) если I_t — это OP θ A, R, C, то $v_t(C) = \theta v_{t-1}(A)\, v_{t-1}(R)$ (напомним, что R может быть сумматором или ячейкой памяти),

(5) если $v_t(R)$ не определено по (2)—(4), а $v_{t-1}(R)$ определено, то $v_t(R) = v_{t-1}(R)$; в противном случае $v_t(R)$ не определено.

Таким образом, значения вычисляются именно так, как и следовало ожидать. Команды LOAD и STORE передают значения с одного регистра на другой, оставляя их в исходном регистре. Операции помещают вычисленное значение на сумматор, определяемый третьим аргументом, не меняя значений остальных регистров. Будем говорить, что программа P *вычисляет* выражение α, помещая результат на сумматор A, если после последнего оператора из P сумматор A имеет значение α.

Пример 11.14. Рассмотрим программу на языке ассемблера с двумя сумматорами A и B, значения которых после каждой

Таблица 11.4

	$v(A)$	$v(B)$
LOAD X, A	X	не определено
ADD A, Y, A	$X+Y$	не определено
LOAD Z, B	$X+Y$	Z
MULT B, A, A	$Z*(X+Y)$	Z

команды приведены в табл. 11.4 в инфиксной записи, как обычно. Значение сумматора A в конце программы соответствует (инфиксному) выражению $Z*(X+Y)$. Таким образом, эта программа вычисляет $Z*(X+Y)$, помещая результат на сумматор A. Формально вычисляется также и значение выражения Z. □

В настоящем разделе мы формально определим (арифметическое) *синтаксическое дерево* как помеченное двоичное дерево T, имеющее одну или более таких вершин, что

(1) каждая внутренняя вершина помечена бинарной операцией $\theta \in \Theta$,

(2) все листья помечены различными именами переменных $X \in \Sigma$.

Для удобства будем предполагать, что множества Θ и Σ не пересекаются. На рис. 11.8 изображено дерево для $Z*(X+Y)$.

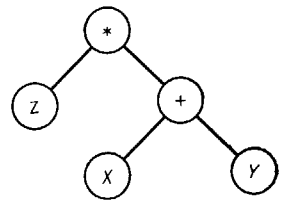


Рис. 11.8. Дерево для $Z*(X+Y)$.

Вершинам дерева, начиная снизу, можно следующим образом приписать *значения*:

(1) если n — лист, помеченный X, то n имеет значение X,

(2) если n — внутренняя вершина, помеченная θ, и ее прямыми потомками являются n_1 и n_2 со значениями v_1 и v_2, то n имеет значение $\theta v_1 v_2$.

Значением дерева будем считать значение его корня. Например, значением дерева на рис. 11.8 будет в инфиксной записи $Z*(X+Y)$.

Рассмотрим кратко связь между блоками на промежуточном языке (разд. 11.1) и программами на языке ассемблера, которые мы только что определили. Прежде всего если дан приведенный блок, в котором

(1) все операции бинарные,
(2) на каждую входную переменную делается одна ссылка,
(3) есть в точности одна выходная переменная,

то граф, связанный с этим блоком, является деревом. По нашей терминологии это дерево синтаксическое. Значение дерева — это в то же время и значение блока.

Можно естественным образом, оператор за оператором, преобразовать блок на промежуточном языке в программу на языке ассемблера. Оказывается, что если при этом преобразовании учесть возможность оставлять вычисленные значения на сумматорах, то оптимальную программу на языке ассемблера можно получить из данного приведенного открытого блока, произведя сначала только преобразование T_4, как это предлагалось в теореме 11.5, и выполнив затем преобразование в язык ассемблера.

Однако все это, быть может, не вполне очевидно; читатель должен сам проверить все эти утверждения. Именно в результате существенной переделки многих определений из разд. 11.1 для случая программ на языке ассемблера мы добиваемся возможности показать, что нет такой оптимальной программы на языке ассемблера, которая не была бы связана с блоком на промежуточном языке, получаемым из приведенного открытого блока, некоторым естественным пооператорным преобразованием и преобразованием T_4.

11.2.2. Разметка деревьев

Основное в алгоритме генерации кода для выражений — это метод назначения дополнительных меток вершинам синтаксического дерева. Такими метками будут целые числа, и в дальнейшем мы будем ссылаться на них как на метки вершин, несмотря на то, что каждая вершина помечена также операцией или переменной. Целочисленные метки определяют номера сумматоров, нужных для оптимального вычисления выражения.

Алгоритм 11.1. Разметка синтаксического дерева.

Вход. Синтаксическое дерево T.

Выход. Помеченное синтаксическое дерево.

Метод. Вершинами дерева T рекурсивно, начиная снизу, назначаем целочисленные метки:

(1) Если вершина есть лист, являющийся левым прямым потомком своего прямого предка, или корень (т. е. дерево состоит из одной этой вершины), помечаем эту вершину 1; если вершина есть лист, являющийся правым прямым потомком, помечаем ее 0.

(2) Пусть вершина n имеет прямых потомков n_1 и n_2, помеченных l_1 и l_2. Если $l_1 \neq l_2$, берем в качестве ее метки большее из чисел l_1 и l_2. Если $l_1 = l_2$, берем в качестве ее метки число $l_1 + 1$. □

Пример 11.15. Арифметическое выражение

$$A * (B - C)/(D * (E - F))$$

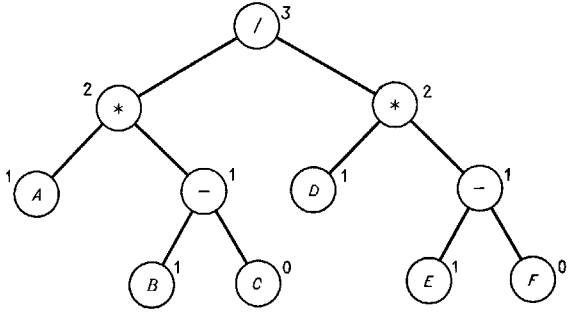


Рис. 11.9. Помеченное синтаксическое дерево.

изображено в виде дерева на рис. 11.9, на котором проставлены целочисленные метки. □

Дадим теперь алгоритм, преобразующий помеченное синтаксическое дерево в программу на языке ассемблера для машины с N сумматорами. Мы покажем, что для любого N получаемая программа оптимальна относительно различных оценок, включая такие, как длина программы.

Алгоритм 11.2. Построение кода на языке ассемблера для выражений.

Вход. Помеченное синтаксическое дерево T и N сумматоров $A_1, A_2, \ldots, A_N$, где $N \geqslant 1$.

Выход. Программа P на языке ассемблера, после последней команды которой значением $v(A_1)$ становится $v(T)$; т. е. P, вычисляет выражение, представленное деревом T, и помещает результат на сумматор A_1.

Метод. Предполагается, что дерево T помечено в соответствии с алгоритмом 11.1. Затем рекурсивно выполняется процедура **code**(n, i). Входом для **code** служит вершина n дерева T и целое число i от 1 до N. Число i означает, что в данный момент для вычисления выражения в вершине n доступны сумматоры $A_i, A_{i+1}, \ldots, A_N$. Выходом для **code**$(n, i)$ служит последовательность команд языка ассемблера, которая вычисляет значение $v(n)$ и помещает результат на сумматор A_i.

Сначала выполняется **code**$(n_0, 1)$, где n_0 — корень дерева T. Последовательность команд, генерированных этим вызовом процедуры **code**, и будет нужной программой на языке ассемблера.

Процедура **code**(n, i). Предполагается, что n — вершина дерева T, а i — целое число между 1 и N.

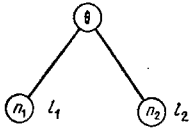
(1) Если n — лист, выполняем шаг (2). В противном случае выполняем шаг (3).

(2) Если вызвана процедура **code**(n, i), а n — лист, то n всегда будет левым прямым потомком (или корнем, если n — единственная вершина дерева). Если с листом n связано имя переменной X, то

$$\textbf{code}(n, i) = \text{'LOAD } X, A_i\text{'}$$

(это означает, что выходом процедуры **code**(n, i) служит команда LOAD X, A_i).

(3) В это место мы приходим, только если n — внутренняя вершина. Пусть с вершиной n связана операция θ и ее прямыми



потомками являются n_1 и n_2 с метками l_1 и l_2 (см. рисунок). Следующий шаг определяется значениями меток l_1 и l_2:

(а) если $l_2 = 0$ (n_2 — правый лист), выполняем шаг (4),
(б) если $1 \leqslant l_1 < l_2$ и $l_1 < N$, выполняем шаг (5),
(в) если $1 \leqslant l_2 \leqslant l_1$ и $l_2 < N$, выполняем шаг (6),
(г) если $N \leqslant l_1$ и $N \leqslant l_2$, выполняем шаг (7).

(4) **code**$(n, i) =$ **code**(n_1, i)
'OP θ A_i, X, A_i'

Здесь X — переменная, связанная с листом n_2, а OP θ — код операции θ. Выходом для **code**(n, i) служит выход для **code**(n_1, i), сопровождаемый командой OP θ A_i, X, A_i.

(5) **code**$(n, i) =$ **code**(n_2, i)
code$(n_1, i+1)$
'OP θ A_{i+1}, A_i, A_i'

(6) **code**$(n, i) =$ **code**(n_1, i)
code$(n_2, i+1)$
'OP θ A_i, A_{i+1}, A_i'

(7) **code**$(n, i) =$ **code**(n_2, i)
$T \leftarrow$ **newtemp**
'STORE A_i, T'
code(n_1, i)
'OP θ A_i, T, A_i'

Здесь **newtemp** — функция, которая при обращении к ней вырабатывает новую временную ячейку памяти для запоминания промежуточных результатов. □

Позже мы увидим, что при выполнении шагов (5)—(7) алгоритма 11.2 справедливы соотношения между l_1, l_2 и i, приведенные в табл. 11.5.

Таблица 11.5

Шаг	Соотношение
(5)	$i \leqslant N - l_1$
(6)	$i \leqslant N - l_2$
(7)	$i = 1$

Несколько усложнив процедуру **code** на шаге (5), можно избежать использования команд вида

OP θ A, B, A
OP θ A, B, B

Несколько усложнив процедуру **code** на шаге (5), можно избежать использования команд вида

OP θ A, B, B

не входящих в определенный нами выше набор команд машин с многими регистрами (см. упр. 11.2.11).

Можно рассматривать **code**(n, i) как функцию, вычисляющую перевод в каждой вершине выражения в терминах переводов и меток ее прямых потомков. Чтобы лучше понять алгоритм 11.2, разберем несколько примеров.

Пример 11.16. Пусть T — синтаксическое дерево, состоящее лишь из вершины X (помеченной 1). В соответствии с шагом (2) **code** — это единственная команда LOAD X, A_1. □

Пример 11.17. Пусть T — помеченное синтаксическое дерево, изображенное на рис. 11.10. В табл. 11.6 показано, как вырабатывается алгоритмом 11.2 программа на языке ассемблера для T при $N=2$. Приведены вызовы процедуры **code**(n, i) и со-

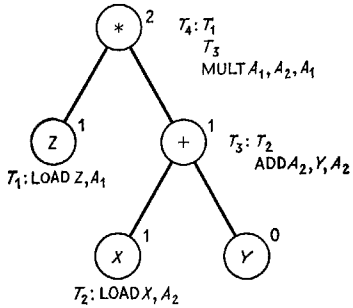


Рис. 11.10. Помеченное синтаксическое дерево с переводами.

ответствующие им шаги алгоритма 11.2. Вершина указывается при помощи связанной с ней переменной или операции.

Таблица 11.6

Вызов	Шаг алгоритма 11.2
code(*, 1)	(3в)
code(Z, 1)	(2)
code(+, 2)	(3а)
code(X, 2)	(2)

Вызов **code**(X, 2) генерирует команду LOAD X, A_2, являющуюся переводом, связанным с вершиной X. Вызов **code**(+, 2) генерирует последовательность команд

```
LOAD  X, A2
ADD   A2, Y, A2
```

являющуюся переводом для вершины +.

Вызов **code**(Z, 1) генерирует команду LOAD Z, A_1, являющуюся переводом для вершины Z. Вызов **code**(*, 1) генерирует

окончательную программу — перевод корня:

```
LOAD  Z, A1
LOAD  X, A2
ADD   A2, Y, A2
MULT  A1, A2, A1
```

Эта программа аналогична (но не идентична) программе из примера 11.14. Очевидно, что в конце программы на сумматоре A_1 будет значение $Z*(X+Y)$. □

Пример 11.18. Применим алгоритм 11.2 при $N=2$ к синтаксическому дереву рис. 11.9. Последовательность вызовов **code**(n, i) приведена в табл. 11.7. Здесь $*_L$ — ссылка на левый потомок

Таблица 11.7

Вызов	Шаг
code(/, 1)	(3г)
code($*_R$, 1)	(3в)
code(D, 1)	(2)
code($-_R$, 2)	(3а)
code(E, 2)	(2)
code($*_L$, 1)	(3в)
code(A, 1)	(2)
code($-_L$, 2)	(3а)
code(B, 2)	(2)

вершины /, $*_R$ — на правый потомок вершины /, $-_L$ — на правый потомок вершины $*_L$, а $-_R$ — на правый потомок вершины $*_R$. Указаны также шаги алгоритма 11.2, выполняемые при каждом вызове.

Процедура **code**(/, 1) генерирует программу

```
LOAD   D, A1
LOAD   E, A2
SUBTR  A2, F, A2
MULT   A1, A2, A1
STORE  A1, TEMP1
LOAD   A, A1
LOAD   B, A2
SUBTR  A2, C, A2
MULT   A1, A2, A1
DIV    A1, TEMP1, A1
```

Здесь TEMP1 — ячейка памяти, генерированная функцией **newtemp**. □

Докажем, что значение метки корня помеченного синтаксического дерева, построенного алгоритмом 11.1, равно наименьшему числу сумматоров, необходимому для того, чтобы вычислить выражение, не прибегая к командам STORE.

Начнем с нескольких замечаний по поводу алгоритма 11.2.

Лемма 11.18. *Программа, выработанная процедурой* **code**(n, i) *в алгоритме* 11.2, *правильно вычисляет значение вершины* n, *помещая его на* i*-й сумматор.*

Доказательство. Элементарная индукция по высоте вершины. □

Лемма 11.9. *Если алгоритм* 11.2 *при* N *доступных сумматорах применяется к корню синтаксического дерева, то при вызове процедуры* **code**(n, i) *в вершине* n *с меткой* l *либо*

(1) $l \geqslant N$ *и для этого вызова доступны* N *сумматоров* (*т. е.* $i=1$), *либо*

(2) $l < N$ *и для этого вызова доступны по крайней мере* l *сумматоров* (*т. е.* $i \leqslant N-l+1$).

Доказательство. Снова элементарная индукция, на этот раз по числу вызовов **code** (n, i), сделанных до рассматриваемого вызова. □

Теорема 11.6. *Пусть* T — *синтаксическое дерево,* l — *метка его корня,* N — *число доступных сумматоров. Программа, вычисляющая* T *без использования команд* STORE, *существует тогда и только тогда, когда* $l \leqslant N$.

Доказательство. *Достаточность.* Если $l \leqslant N$, то шаг (7) процедуры **code**(n, i) никогда не выполняется. Действительно, вершина, прямые потомки которой помечены числом не менее N, сама помечена числом не менее $N+1$. Команда STORE генерируется только на шаге (7). Поэтому при $l \leqslant N$ программа, вырабатываемая алгоритмом 11.2, не содержит команд STORE.

Необходимость. Пусть $l > N$. Поскольку $N \geqslant 1$, должно быть $l \geqslant 2$. Предположим, что утверждение неверно. Тогда без потери общности можно считать, что дерево T имеет программу P, вычисляющую его с помощью N сумматоров, P не содержит команд STORE и не существует синтаксического дерева T', имеющего меньше вершин, чем T, и также не удовлетворяющего утверждению. Поскольку метка корня дерева T превосходит 1, T не может состоять из единственного листа. Пусть n — корень, а n_1 и n_2 — его прямые потомки с метками l_1 и l_2 соответственно.

Случай 1: $l_1 = l$. Значение n можно вычислить, только если значение n_1 появится в некоторый момент на сумматоре, поскольку n_1 не может быть листом. Формируем из P новую программу P', удаляя операторы, следующие за вычислением значения n_1. Тогда P' вычисляет поддерево с корнем n_1 и не содержит команд STORE. Таким образом, утверждение неверно при числе вершин, меньшем, чем в T, вопреки предположению относительно T.

Случай 2: $l_2 = l$. Этот случай аналогичен случаю 1.

Случай 3: $l_1 = l_2 = l-1$. Мы уже предполагали, что никакие два листа не имеют одинаковых имен связанных с ними переменных. Без потери общности можно считать, что программа P „коротка, насколько возможно" в том смысле, что если удалить любой из операторов, то в конце программы значения n в том же сумматоре уже не будет. Таким образом, первым оператором программы должен быть LOAD X, A, где X — имя переменной, связанной с некоторым листом дерева T, так как иначе первый оператор можно было бы удалить.

Пусть X — значение листа, являющегося потомком вершины n_1 (возможно, самой вершины n_1). Случай, когда X — значение потомка вершины n_2, аналогичен; мы его рассматривать не будем. До тех пор пока не вычислится значение n_1, всегда по крайней мере один сумматор будет хранить значение, содержащее X. Этим значением нельзя воспользоваться в правильном вычислении значения n_2. Поэтому из P можно получить программу P', вычисляющую значение n_2, помеченную $l-1$, не использующую команд STORE и требующую одновременно не более $N-1$ сумматоров. Читателю предлагаем показать, что для P' можно построить эквивалентную программу P'', для которой никогда не понадобится более $N-1$ различных сумматоров. (Отметим, что в P' могут употребляться все N сумматоров, даже если одновременно их используется не более $N-1$.) Таким образом, утверждение теоремы неверно для поддерева с корнем n_2, а это противоречит предположению о минимальности дерева T. Отсюда следует, что теорема справедлива. □

11.2.3. Программы с командами STORE

Выясним теперь, сколько команд LOAD и STORE требуется для вычисления синтаксического дерева, если пользоваться N сумматорами, когда корень помечен числом, превышающим N. Нам пригодятся следующие определения.

Определение. Пусть T — синтаксическое дерево, а N — число доступных сумматоров. Вершина из T называется *старшей*, если каждый ее прямой потомок помечен числом, не меньшим чем N. Вершина называется *младшей*, если она является листом и левым

прямым потомком своего прямого предка (т. е. это лист с меткой 1).

Пример 11.19. Рассмотрим синтаксическое дерево, изображенное на рис. 11.9, при $N=2$. Здесь одна старшая вершина — корень и четыре младших вершины — листья со значениями A, B, D и E. □

Лемма 11.10. *Пусть T — синтаксическое дерево. Программа P, вычисляющая T и использующая m команд* LOAD, *существует тогда и только тогда, когда T имеет не более m младших вершин.*

Доказательство. Если присмотреться к процедуре **code**(n, i) из алгоритма 11.2, то можно заметить, что оператор LOAD вставляется только на шаге (2). Поскольку шаг (2) применяется только к младшим вершинам, достаточность доказана.

Необходимость доказывается, как в теореме 11.6, с учетом того, что значение на сумматоре может появиться только в результате команды LOAD, а левый аргумент любой операции должен быть на сумматоре. □

Лемма 11.11. *Пусть T — синтаксическое дерево. Программа P, вычисляющая T и использующая M команд* STORE, *существует тогда и только тогда, когда T имеет не более M старших вершин.*

Доказательство. *Достаточность.* Исследуя снова процедуру **code**(n, i), видим, что команда STORE вставляется только на шаге (7), а он применяется только к старшим вершинам.

Необходимость. Эта часть доказывается индукцией по числу вершин в T. Базис индукции, т. е. случай, когда дерево имеет вершину, тривиален, так как вершина помечена 1 и, значит, старших вершин нет. Предположим, что утверждение верно для синтаксических деревьев с числом вершин вплоть до $k-1$, и пусть T имеет k вершин.

Рассмотрим программу P, вычисляющую T, и обозначим через M число старших вершин в T. Без потери общности можно считать, что P содержит не больше команд STORE, чем любая другая программа, вычисляющая T. При $M=0$ нужный результат очевиден, поэтому будем считать, что $M \geqslant 1$. Тогда P содержит хотя бы одну команду STORE, поскольку метка старшей вершины не меньше $N+1$, и если бы в P не было команд STORE, теорема 11.6 была бы неверна.

Значение, запомненное первой командой STORE, должно быть значением некоторой вершины n из T, а иначе было бы легко найти вычисляющую T программу, в которой команд STORE меньше, чем в P. Кроме того, по той же самой причине можно считать, что n не лист. Пусть T' — синтаксическое дерево, получающееся из T, если сделать вершину n листом и дать ей в качестве значения некоторое новое имя X. Тогда у T' будет меньше вершин, чем у T, так что к нему можно применить предположение индукции. Можно найти программу P', вычисляющую T' и использующую ровно на одну команду STORE меньше, чем P. Программа P' получается из P, если удалить в точности те операторы, которые нужны для вычисления первого запомненного значения, и затем заменить в P ссылки на ячейку, участвующую в этой команде STORE, именем X, пока в этой ячейке не будет запомнено новое значение.

Если мы сможем показать, что T' имеет не менее $M-1$ старших вершин, то лемма будет доказана, поскольку по предположению индукции P' имеет не менее $M-1$ команд STORE и, значит, P имеет не менее M команд STORE.

Ясно, что ни один из потомков вершины n в T не может быть старшим, поскольку в этом случае будет неверна теорема 11.6. Рассмотрим старшую вершину n' в T. Если n не является ее потомком, то n' — старшая вершина в T'. Таким образом, достаточно рассмотреть только старшие вершины $n_1, n_2, \ldots$ на пути от n к корню дерева T. В соответствии с рассуждениями, проведенными в случае 3 теоремы 11.6, сама вершина n не может быть старшей. Первая вершина n_1, если она существует, не может тогда быть старшей в T'. Однако метка вершины n_1 в T' не меньше чем N, потому что ее прямой потомок, не являющийся предком вершины n, должен иметь метку не менее N и в T, и в T'. Следовательно, $n_2, n_3, \ldots$ остаются старшими вершинами в T'. Отсюда можно заключить, что T' имеет не менее $M-1$ старших вершин. Индукция проведена полностью. □

Теорема 11.7. *Алгоритм* 11.2 *всегда вырабатывает кратчайшую программу для вычисления данного выражения.*

Доказательство. В соответствии с леммами 11.10 и 11.11 алгоритм 11.2 генерирует программу с минимально возможным числом команд LOAD и STORE. Поскольку ясно, что минимальное число команд операций равно числу внутренних вершин дерева, а алгоритм 11.2 дает по одной такой команде для каждой внутренней вершины, то утверждение теоремы очевидно. □

Пример 11.20. Как указывалось в примере 11.19, арифметическое выражение на рис. 11.9 имеет одну старшую и четыре младших вершины (в предположении, что $N=2$). Оно имеет также пять внутренних вершин. Таким образом, для его вычисления требуется не менее десяти операторов. Программа из

примера 11.18 содержит как раз десять операторов. Отметим, что среди них одна команда STORE, четыре LOAD, а остальные — команды операций. □

11.2.4. Влияние некоторых алгебраических законов

Определим *оценку синтаксического дерева* как сумму

(1) числа внутренних вершин,
(2) числа старших вершин,
(3) числа младших вершин.

Результаты предыдущего раздела показывают, что эта оценка служит разумной мерой „сложности" синтаксического дерева, поскольку число команд, необходимых для вычисления синтаксического дерева, равно его оценке.

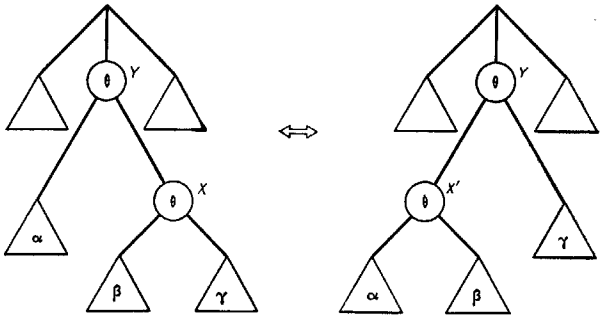


Рис. 11.11. Ассоциативное преобразование синтаксических деревьев.

Часто к некоторым операциям можно применить алгебраические законы и с их помощью уменьшить оценку данного синтаксического дерева. Из разд. 11.1.6 известно, что каждый алгебраический закон индуцирует соответствующее преобразование синтаксического дерева. Например, если n — внутренняя вершина синтаксического дерева, связанная с коммутативной операцией, то коммутативное преобразование меняет порядок прямых потомков вершины n.

Аналогично, если θ — ассоциативная операция (т. е. $\alpha\theta(\beta\theta\gamma) = (\alpha\theta\beta)\theta\gamma$), то, применяя соответствующее преобразование деревьев, можно перевести друг в друга два дерева, изображенные на рис. 11.11. Ассоциативное преобразование, приведенное

на рис. 11.11, соответствует преобразованию блоков

$$\begin{matrix} X \leftarrow B\theta C \\ Y \leftarrow A\theta X \end{matrix} \Leftrightarrow \begin{matrix} X' \leftarrow A\theta B \\ Y \leftarrow X'\theta C \end{matrix}$$

В разд. 11.1.6 после применения преобразования слева направо оператор $X \leftarrow B\theta C$ сохранялся. Однако теперь после преобразования этот оператор становится бесполезным, так что его можно удалить, не меняя значения блока.

Определение. Если дано множество $\mathcal{A}$ алгебраических законов, то два синтаксических дерева T_1 и T_2 будем называть *эквивалентными относительно* $\mathcal{A}$ и писать $T_1 \equiv_{\mathcal{A}} T_2$, если существует последовательность преобразований, выводимая из этих законов, переводящая T_1 в T_2. Через $[T]_{\mathcal{A}}$ будем обозначать класс эквивалентности деревьев $\{T' \mid T \equiv_{\mathcal{A}} T'\}$.

Таким образом, если дано синтаксическое дерево T и известно, что выполняются алгебраические законы из некоторого множества законов $\mathcal{A}$, то для нахождения оптимальной программы для T надо искать в $[T]_{\mathcal{A}}$ дерево выражения с минимальной оценкой. Если дерево с минимальной оценкой найдено, то для нахождения оптимальной программы можно применить алгоритм 11.2. Теорема 11.7 гарантирует, что получаемая программа будет оптимальной.

Если каждый закон сохраняет число операций, как в случае коммутативного и ассоциативного законов, достаточно минимизировать сумму чисел старших и младших вершин. В качестве примера приведем алгоритм этой минимизации сначала для случая, когда некоторые операции коммутативны, а затем для случая, когда некоторые коммутативные операции также и ассоциативны.

По данному синтаксическому дереву T и множеству $\mathcal{A}$ алгебраических законов приведенный ниже алгоритм будет строить синтаксическое дерево $T' \in [T]_{\mathcal{A}}$ с минимальной оценкой при условии, что $\mathcal{A}$ содержит только коммутативные законы, применимые к определенным операциям. Затем к T' можно будет применить алгоритм 11.2 и найти оптимальную программу для исходного дерева T.

Алгоритм 11.3. Построение дерева с минимальной оценкой, некоторые операции которого коммутативны.

Вход. Синтаксическое дерево T (с тремя или более вершинами) и множество $\mathcal{A}$ коммутативных законов.

Выход. Синтаксическое дерево из $[T]_{\mathcal{A}}$ с минимальной оценкой.

Метод. Ядро алгоритма составляет рекурсивная процедура **commute**(n), аргументом которой служит вершина n синтаксического дерева, а результатом — модифицированное поддерево с вершиной n в качестве корня. Вначале вызывается **commute**(n_0), где n_0 — корень данного дерева T.

Процедура **commute**(n).

(1) Если n — лист, полагаем **commute**(n) $= n$.

(2) Если n — внутренняя вершина, рассмотрим два случая:

а) Пусть вершина n имеет два прямых потомка n_1 и n_2 (в указанном порядке) и операция, связанная с n, коммутативна. Если n_1 — лист, а n_2 нет, то выход **commute**(n) — дерево на рис. 11.12, *а*.

б) Во всех других случаях выход **commute**(n) — дерево на рис. 11.12, *б*. □

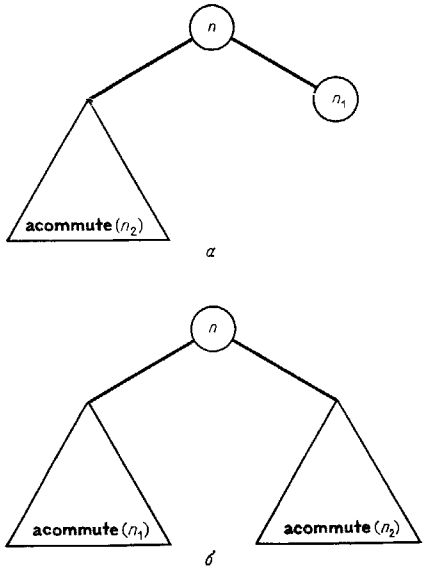


Рис. 11.12. Результат процедуры commute.

Пример 11.21. Рассмотрим рис. 11.9 и предположим, что коммутативно только умножение. Результат применения алгоритма 11.3 к этому дереву показан на рис. 11.13. Отметим, что корень на рис. 11.13 помечен 2 и что здесь две младшие вершины. Таким образом, если у нас два сумматора, то для

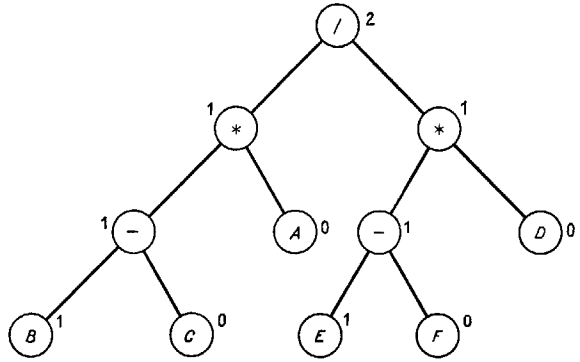


Рис. 11.13. Преобразованное арифметическое выражение.

вычисления этого дерева требуется только семь команд, а для дерева на рис. 11.9 — десять. □

Теорема 11.8. *Если допустимо применение одного только коммутативного для некоторых операций закона, то алгоритм* 11.3 *вырабатывает такое синтаксическое дерево из класса эквивалентности для данного дерева, которое имеет минимальную оценку.*

Доказательство. Легко видеть, что коммутативный закон не может изменить числа внутренних вершин. Простая индукция по высоте вершины показывает, что алгоритм 11.3 минимизирует число младших вершин и метки, которые должны быть у вершин после применения алгоритма 11.1. Следовательно, минимизируется также число старших вершин. □

Ситуация усложняется, когда некоторые операции одновременно коммутативны и ассоциативны. В этом случае уменьшить число старших вершин часто можно за счет существенного преобразования дерева.

Определение. Пусть T — синтаксическое дерево. Множество S из двух или более его вершин называется *кистью*, если

(1) все вершины в S внутренние и представляют одну и ту же ассоциативную и коммутативную операцию,

(2) вершины из S вместе с соединяющими их дугами образуют дерево,

(3) ни одно из собственных иадмножеств множества S не обладает свойствами (1) и (2).

Корнем кисти называется корень дерева, о котором идет речь в пункте (2). *Прямыми потомками* кисти S называются те вершины из T, которые не принадлежат S, но являются прямыми потомками вершин из S.

Пример 11.22. Рассмотрим синтаксическое дерево на рис. 11.14, в котором сложение и умножение коммутативны, а иикакие другие алгебраические законы неприменимы.

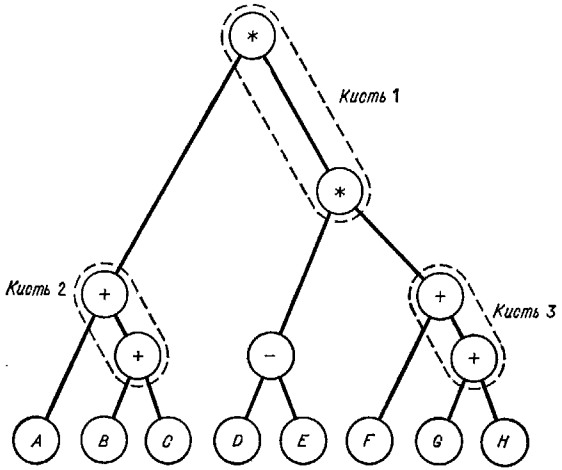


Рис. 11.14. Синтаксическое дерево.

Три кисти обведены штриховыми линиями. Кисть, включающая корень дерева, имеет в качестве прямых потомков в порядке слева направо корень кисти 2, вершину, с которой сопоставлена операция вычитания, и корень кисти 3. □

Заметим, что кисти синтаксического дерева T определяются однозначно и не пересекаются. Для нахождения в $[T]_{\mathcal{A}}$ дерева с минимальной оценкой, когда $\mathcal{A}$ содержит законы, отражающие тот факт, что одни операции коммутативны и ассоциативны, а другие только коммутативны, вводится понятие ассоциативного дерева, в котором кисти представляются одной вершиной.

Определение. Пусть T — синтаксическое дерево. *Ассоциативным деревом* T' для T назовем дерево, получающееся заменой каждой кисти S в T на единственную вершину n с той же ассоциативной и коммутативной операцией, что и у вершины кисти S. Прямые потомки кисти в T становятся прямыми потомками вершины n в T'.

Пример 11.23. Рассмотрим синтаксическое дерево на рис. 11.15. Предполагая, что сложение и умножение ассоциативны и коммутативны, получаем кисти, обведенные на рис. 11.15

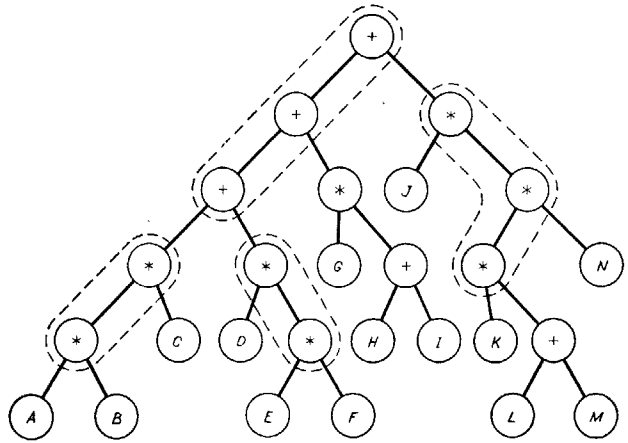


Рис. 11.15. Синтаксическое дерево с кистями.

штриховыми линиями. Ассоциативное дерево для T изображено иа рис. 11.16. Отметим, что ассоциативное дерево не обязано быть двоичным. □

Вершины ассоциативного дерева можно *пометить* целыми числами, начииая синзу, следующим образом:

(1) Лист, являющийся самым левым прямым потомком своего предка, помечается 1. Все остальные листья помечаются 0.

(2) Пусть n — внутренняя вершина, прямые потомки которой $n_1, n_2, \ldots, n_m$ помечены $l_1, l_2, \ldots, l_m$, $m \geqslant 2$.

(а) Если одно из чисел $l_1, l_2, \ldots, l_m$ превосходит остальные, берем его в качестве метки вершины n.

(б) Если вершина n имеет коммутативную операцию и n_i — внутренняя вершина с $l_i = 1$, а остальные вершины

$n_1, \ldots, n_{i-1}, n_{i+1}, \ldots, n_m$ — листья, то в качестве метки вершины n берем 1.

(в) Если условие (б) не выполняется, $l_i = l_j$ для некоторых $i \neq j$ и l_i не меньше остальных l_k, в качестве метки вершины n берем $l_i + 1$.

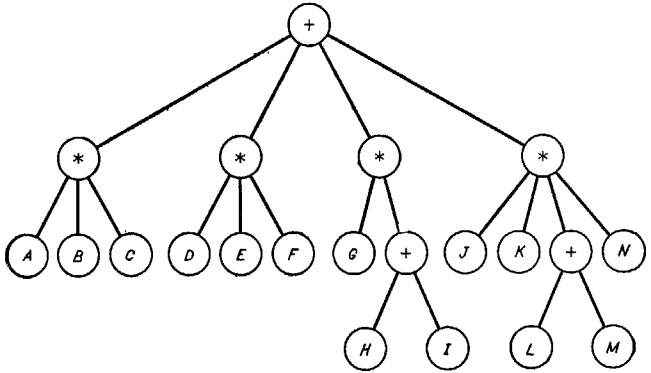


Рис. 11.16. Ассоциативное дерево.

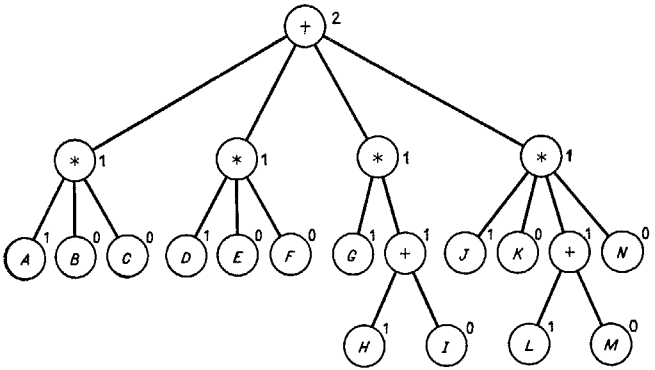


Рис. 11.17. Помеченное ассоциативное дерево.

Пример 11.24. Рассмотрим ассоциативное дерево на рис. 11.16. Помеченное ассоциативное дерево изображено на рис. 11.17.

Отметим, что к третьему и четвертому прямым потомкам корня применяется условие (2б) процедуры разметки, поскольку умножение коммутативно. □

Приведем теперь алгоритм, принимающий на вход данное синтаксическое дерево и вырабатывающий такое дерево из класса эквивалентности для данного дерева, которое имеет минимальную оценку.

Алгоритм 11.4. Построение синтаксического дерева с минимальной оценкой в предположении, что одни операции коммутативны, другие коммутативны и ассоциативны и больше никакие алгебраические законы не учитываются.

Вход. Синтаксическое дерево T и множество $\mathcal{A}$ коммутативных и коммутативно-ассоциативных законов.

Выход. Синтаксическое дерево из $[T]_{\mathcal{A}}$ с минимальной оценкой.

Метод. Строим сначала помеченное ассоциативное дерево T' для T. Затем вычисляем **acommute**(n_0), где **acommute** — процедура, определенная ниже, а n_0 — корень дерева T'. Выходом **acommute**(n_0) служит синтаксическое дерево из $[T]_{\mathcal{A}}$ с минимальной оценкой.

Процедура **acommute**(n).

Аргументом n служит вершина помеченного ассоциативного дерева. Если n — лист, полагаем **acommute**$(n) = n$. Если n — внутренняя вершина, рассмотрим три случая:

(1) Пусть вершина имеет два прямых потомка n_1 и n_2 (в указанном порядке) и операция, связанная с n, коммутативна (и, возможно, ассоциативна).

(а) Если n_1 — лист, а n_2 нет, то выход **acommute**(n) — дерево на рис. 11.18, *а*.

(б) В противном случае **acommute**(n) — дерево на рис. 11.18, *б*.

(2) Предположим, что операция θ, связанная с n, коммутативна и ассоциативна и вершина n имеет прямых потомков $n_1, n_2, \ldots, n_m$, $m \geqslant 3$ (в порядке слева направо).

Пусть $n_{\max}$ — вершина из $n_1, \ldots, n_m$ с наибольшей меткой. Если одной и той же наибольшей меткой помечены две или более вершин, то вершину $n_{\max}$ выбираем так, чтобы она была внутренней. Обозначим через $p_1, p_2, \ldots, p_{m-1}$ вершины в $\{n_1, \ldots, n_m\} - \{n_{\max}\}$ в любом порядке.

Тогда выходом **acommute**(n) служит двоичное дерево на рис. 11.19, где r_i $(1 \leqslant i \leqslant m-1)$ — новые вершины, связанные с коммутативной и ассоциативной операцией θ, соответствующей вершине n.

(3) Если операция, связанная с n, не коммутативна и не ассоциативна, то выходом **acommute**(n) служит дерево на рис. 11.18, *б*[1]). □

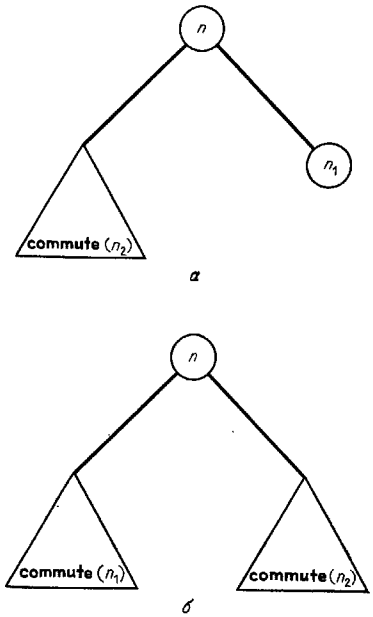


Рис. 11.18. Результат процедуры **acommute**.

Пример 11.25. Применим алгоритм 11.4 к помеченному ассоциативному дереву, изображенному на рис. 11.17. Применяя **acommute**, а точнее случай (2) к корню, берем в качестве n_{max} первого слева прямого потомка. Двоичное дерево, являющееся выходом алгоритма 11.4, показано на рис. 11.20. □

Закончим этот раздел доказательством того, что алгоритм 11.4 находит в $[T]_{\mathcal{A}}$ дерево с минимальной оценкой. Центральной в нашем доказательстве будет следующая лемма.

[1]) В этом случае n имеет ровно два прямых потомка.— *Прим. перев.*

Лемма 11.12. *Пусть T — помеченное синтаксическое дерево, а S — кисть в T*[1]). *Предположим, что метки r прямых потомков вершин из S больше или равны N, где N — число сумматоров. Тогда по крайней мере $r-1$ вершин из S являются старшими.*

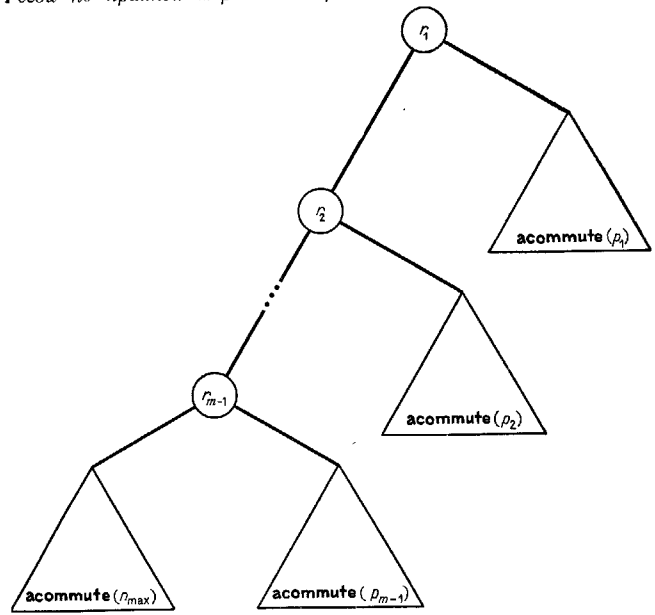


Рис. 11.19. Результат процедуры **acommute**.

Доказательство. Будем доказывать лемму индукцией по числу вершин в T. Базис, случай одной вершины, тривиален. Предположим поэтому, что утверждение верно для всех деревьев с числом вершин, меньшим, чем в T. Пусть n — корень кисти S, и пусть n имеет прямых потомков n_1 и n_2 с метками l_1 и l_2. Пусть T_1 и T_2 — имена поддеревьев с корнями n_1 и n_2 соответственно.

Случай 1: ни n_1, ни n_2 не принадлежат S. Очевидно, что в этом случае утверждение верно.

[1]) Доказательство будет согласовано с формулировкой, если S — либо кисть, либо одиночная вершина, не входящая в кисть.— *Прим. перев.*

Случай 2: n_1 принадлежит S, а n_2 нет. Поскольку число вершин в дереве T_1 меньше, чем в T, к нему применимо предположение индукции. Таким образом, в T_1 множество $S-\{n\}$ содержит по крайней мере $r-2$ старших вершин, если $l_2 \geqslant N$, и по крайней мере $r-1$ старших вершин, если $l_2 < N$. В последнем случае утверждение тривиально. В обоих случаях результат очевиден, если $r \leqslant 1$. Поэтому рассмотрим случай $r > 1$ и $l_2 \geqslant N$.

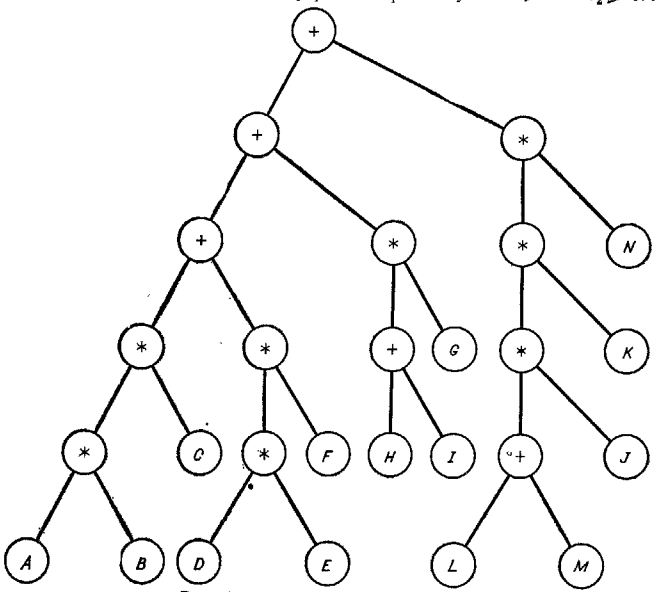


Рис. 11.20. Выход алгоритма 11.4.

Тогда $S-\{n\}$ имеет по крайней мере одного прямого потомка, метка которого больше или равна N, так что $l_1 \geqslant N$. Таким образом, n — старшая вершина и S содержит не менее $r-1$ старших вершин.

Случай 3: n_2 принадлежит S, а n_1 нет. Этот случай аналогичен случаю 2.

Случай 4: n_1 и n_2 принадлежат S. Пусть r_1 прямых потомков вершин из S с метками, не меньшими N, являются потомками вершины n_1, а r_2 из них являются потомками вершины n_2. Тогда $r_1+r_2=r$. По предположению индукции части кисти S

в T_1 и T_2 имеют соответственно не менее r_1-1 и не менее r_2-1 старших вершин. Если ни r_1, ни r_2 не равны нулю, то $l_1 \geqslant N$ и $l_2 \geqslant N$, так что n — старшая вершина. Таким образом, S имеет по крайней мере $(r_1-1)+(r_2-1)+1=r-1$ старших вершин. Если $r_1=0$, то $r_2=r$, так что часть кисти S в T_2 имеет не менее $r-1$ старших вершин. Случай $r_2=0$ аналогичен. □

Теорема 11.9. *Алгоритм* 11.4 *вырабатывает дерево из* $[T]_{\mathcal{A}}$ *с минимальной оценкой.*

Доказательство. Прямая индукция по числу вершин в ассоциативном дереве A показывает, что в результате применения процедуры **acommute** к его корню будет построено синтаксическое дерево T, корень которого после разметки, описанной в алгоритме 11.1, помечен так же, как корень дерева A. Никакое дерево из $[T]_{\mathcal{A}}$ не имеет корня с меткой, меньшей, чем у корня дерева A, и никакое дерево из $[T]_{\mathcal{A}}$ не имеет меньше, чем в A, старших и младших вершин.

Предположим, что это не так. Тогда пусть T — наименьшее[1]) дерево, для которого одно из этих условий нарушается. Пусть θ — операция в корне дерева T.

Случай 1: операция θ не ассоциативна и не коммутативна. Любое ассоциативное или коммутативное преобразование дерева T должно совершаться целиком внутри поддерева, корнем которого служит один из двух прямых потомков корня дерева T. Таким образом, касается ли нарушение метки, числа старших или числа младших вершин, оно должно проявиться в одном из этих поддеревьев, что противоречит минимальности дерева T.

Случай 2: операция θ коммутативна, но не ассоциативна. Этот случай аналогичен случаю 1, но теперь коммутативное преобразование можно применить к корню. Но на шаге (1) процедуры **acommute** уже учитывалась возможность применения этого преобразования, так что любое нарушение в дереве T вновь приведет к нарушению в одном из его поддеревьев.

[1]) Здесь «наименьшее» понимается в следующем смысле. По каждому данному синтаксическому дереву T_1 можно построить ассоциативное дерево A. В результате применения к A процедуры **acommute** получается синтаксическое дерево T, которому соответствует множество $[T]_{\mathcal{A}}$. Выберем теперь из всех множеств $[T]_{\mathcal{A}}$ для всевозможных деревьев T_1 все деревья, для которых при сопоставлении с соответствующим деревом A нарушается одно из перечисленных выше условий.

Размер (число вершин) этих деревьев ограничен снизу, и нижняя граница достигается. Выберем среди этих деревьев дерево с наименьшим числом вершин. — *Прим. перев.*

Случай 3: операция θ коммутативна и ассоциативна. Пусть S—кисть, содержащая корень. Можно считать, что ни в одном из поддеревьев, корнями которых служат прямые потомки вершин из S, не нарушается ни одно из указанных выше условий. Любое ассоциативное или коммутативное преобразование должно совершаться целиком внутри одного из этих поддеревьев или целиком внутри S. Если внимательно присмотреться к шагу (2) процедуры **acommute**, станет ясно, что число младших вершин, возникающих из S, минимально. По лемме 11.12 число старших вершин, возникающих из S, минимально (для того чтобы это увидеть, надо вновь исследовать результат процедуры **acommute**), и значит, метка корня минимальна.

Наконец, видно, что изменения, вносимые алгоритмом 11.4, всегда можно совершить с помощью ассоциативных и коммутативных преобразований. □

УПРАЖНЕНИЯ

11.2.1. Предполагая, что не выполняются никакие алгебраические законы, найдите оптимальную программу на языке ассемблера при числе сумматоров $N=1$, 2 и 3 для каждого из следующих выражений:

(а) $A-B*C-D*(E+F)$,
(б) $A+(B+(C*(D+E/F+G)*H))+(I+J)$,
(в) $(A*(B-C))*(D*(E*F))+(G+(H+I))+(J+(K+L)))$.

Вычислите оценку каждой из найденных программ.

11.2.2. Повторите упр. 11.2.1 в предположении, что сложение и умножение коммутативны.

11.2.3. Повторите упр. 11.2.1 в предположении, что сложение и умножение коммутативны и ассоциативны.

11.2.4. Пусть E—бинарное выражение с k операциями. Какое максимальное число скобок требуется для того, чтобы выразить E без лишних скобок?

***11.2.5.** Пусть T—двоичное синтаксическое дерево, корень которого после применения алгоритма 11.1 имеет метку $N \geqslant 2$. Покажите, что T содержит не менее $3 \times 2^{N-2}-1$ внутренних вершин.

11.2.6. Пусть T—дерево бинарного выражения с k внутренними вершинами. Покажите, что T может содержать не более k младших вершин.

***11.2.7.** Покажите, что при данном $N \geqslant 2$ дерево с M старшими вершинами содержит не менее $3(M+1)2^{N-2}-1$ внутренних вершин.

***11.2.8.** Какова максимальная оценка двоичного синтаксического дерева с k вершинами?

***11.2.9.** Каков максимальный выигрыш в оценке двоичного синтаксического дерева с k вершинами при переходе от машины с N сумматорами к машине с $N+1$ сумматорами?

****11.2.10.** Пусть $\mathcal{A}$—произвольное множество алгебраических законов. Разрешима ли проблема эквивалентности относительно $\mathcal{A}$ двух двоичных синтаксических деревьев?

11.2.11. Для алгоритма 11.2 определите процедуру **code**$(n,[i_1, i_2, \ldots, i_k])$, которая вычисляет значение вершины n, пользуясь сумматорами $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$, и помещает результат на сумматоре A_{i_1}. Покажите, что если взять в качестве шага (5)

$$\begin{aligned}&\mathbf{code}(n,[i_1, i_2, \ldots, i_k]) = \\ &\quad \mathbf{code}(n_2,([i_2, i_1, i_3, \ldots, i_k]) \\ &\quad \mathbf{code}(n_1[i_1, i_3, i_4, \ldots, i_k]) \\ &\quad \text{'OP } \theta \ A_{i_1}, A_{i_2}, A_{i_1}\text{'}\end{aligned}$$

то алгоритм 11.2 можно модифицировать так, что все команды языка ассемблера типов (3) и (4) примут вид

OP θ A, B, A

11.2.12. Пусть

$$\mathbf{sign}(a, b) = \begin{cases} |a| & \text{при } b>0 \\ 0 & \text{при } b=0 \\ -|a| & \text{при } b<0 \end{cases}$$

Покажите, что операция **sign** ассоциативна, но не коммутативна. Дайте примеры других операций, ассоциативных, но не коммутативных.

***11.2.13.** Каждая из команд

LOAD M, A
STORE A, M
OP θ A, M, B

использует одно обращение к памяти. Если B—сумматор, то OP θ A, B, C вообще не содержит обращений к памяти. Найдите минимальное число обращений к памяти, генерируемое программой, вычисляющей двоичное синтаксическое дерево с k вершинами.

***11.2.14.** Покажите, что алгоритм 11.2 вырабатывает программу, требующую при вычислении данного выражения минимального числа обращений к памяти.

***11.2.15.** Взяв в качестве входной грамматики G_0, постройте схему синтаксически управляемого перевода, переводящего ин-

фиксные выражения в оптимальную программу на языке ассемблера. Считайте, что имеется N сумматоров и

(а) не выполняются никакие алгебраические законы,

(б) сложение и умножение коммутативны,

(в) сложение и умножение ассоциативны и коммутативны.

11.2.16. Приведите алгоритмы реализации процедур **code**, **commute** и **acommute** за линейное время.

***11.2.17.** Некоторые операции могут потребовать дополнительных сумматоров (например, вызов подпрограмм или арифметика с кратной точностью). Модифицируйте алгоритмы 11.1 и 11.2 так, чтобы учитывать эту возможную потребность в дополнительных сумматорах со стороны операций.

11.2.18. Алгоритмы 11.1—11.4 также можно применять к произвольному двоичному графу, если сначала с помощью расщепления вершин преобразовать его в дерево. Покажите, что алгоритм 11.2 в этом случае не всегда будет генерировать оптимальную программу. Оцените, насколько плохим он может оказаться при этих условиях.

11.2.19. Обобщите алгоритмы 11.1—11.4 так, чтобы они могли обрабатывать выражения, включающие операции с произвольным числом аргументов.

***11.2.20.** Арифметические выражения могут содержать также и унарные операции + и —. Постройте алгоритм генерации оптимального кода для этого случая. (Предполагается, что все операнды различны и что четыре бинарные арифметические операции и унарные + и — связаны обычными алгебраическими законами.)

***11.2.21.** Постройте алгоритм генерации оптимального кода для одного арифметического выражения, в котором каждый операнд является либо отличной от других переменной, либо целой константой. Считайте, что для сложения и умножения выполняются ассоциативный и коммутативный законы и справедливы следующие тождества:

(1) $\alpha+0=0+\alpha=\alpha$,

(2) $\alpha*1=1*\alpha=\alpha$,

(3) $c_1\theta c_2=c_3$, где c_3—целое число, результат применения операции θ к целым числам c_1 и c_2.

11.2.22. Найдите алгоритм генерации оптимального кода для одного логического выражения, в котором каждый операнд является либо отличной от других переменной, либо логической константой (0 или 1). Считайте, что логическое выражение включает операции **и**, **или**, **не** и они удовлетворяют законам булевой алгебры (см. том 1).

****11.2.23.** В некоторых ситуациях две или более операции могут выполняться параллельно. Алгоритмы из разд. 11.1 и 11.2 предполагают последовательное выполнение. Однако, если машина способна выполнять параллельные операции, можно попытаться так упорядочить выполнение, чтобы порождалось как можно больше одновременных параллельных вычислений. Предположим, например, что у нас есть машина с четырьмя регистрами, в которой одновременно могут выполняться четыре операции. Тогда выражение $A_1+A_2+A_3+A_4+A_5+A_6+A_7+A_8$

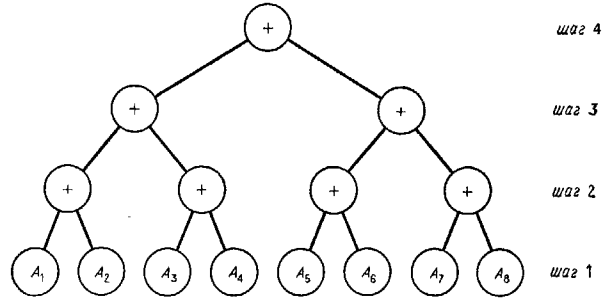


Рис. 11.21. Дерево для параллельных вычислений.

можно представить так, как показано на рис. 11.21. На первом шаге нужно поместить A_1 в первый регистр, A_3—во второй, A_5—в третий, A_7—в четвертый. На втором шаге нужно прибавить A_2 к содержимому регистра 1, A_4—к содержимому регистра 2, A_6—к содержимому регистра 3, а A_8—к содержимому регистра 4. После этого шага регистр 1 будет содержать A_1+A_2, регистр 2 будет содержать A_3+A_4 и т. д. На третьем шаге прибавляем содержимое регистра 2 к содержимому регистра 1, а содержимое регистра 4 к содержимому регистра 3. На четвертом шаге прибавляем содержимое регистра 3 к содержимому регистра 1. Постройте машину с N регистрами, в которой на одном шаге может выполняться до N параллельных операций. Модифицируйте алгоритм 11.1 так, чтобы он генерировал оптимальный код (в смысле минимального числа шагов) для одного арифметического выражения с различными операндами для такой машины.

Проблема для исследования

11.2.24. Найдите эффективный алгоритм, который будет генерировать оптимальный код того же типа, что и рассматриваемый в этом разделе, но для произвольного блока.

Упражнение на программирование

11.2.25. Напишите программы, реализующие алгоритмы 11.1—11.4.

Замечания по литературе

Генерации хорошего кода для арифметических выражений для конкретных машин или классов машин посвящено много статей. Флойд [1961a] описывает ряд методов оптимизации арифметических выражений, включая определение общих подвыражений. Он предлагает также, чтобы второй операнд некоммутативной бинарной операции вычислялся первым. Андерсон [1964] дает алгоритм генерации кода для машины с одним регистром, по существу совпадающий с кодом, вырабатываемым алгоритмом 11.1 при $N=1$. Наката [1967] и Мейерс [1965] приводят аналогичные результаты.

Число регистров, требуемых для вычисления дерева выражения, исследовали Наката [1967], Редзиевский [1969], Сети и Ульман [1970]. Алгоритмы 11.1—11.4 в том виде, в каком они представлены здесь, разработаны Сети и Ульманом [1970]. Упр. 11.2.11 предложил Стокхаузен. Бити [1972] и Фрейли [1970] исследовали расширения, включающие операцию унарного минуса. Чен [1972] обобщил алгоритм 11.2 на некоторые ориентированные ациклические графы.

Эффективный алгоритм генерации оптимального кода для произвольных выражений не известен. Один эвристический метод использования регистров в процессе вычисления выражения основан на следующем алгоритме.

Предположим, что должно вычисляться выражение α, а его значение должно запоминаться в быстром регистре (сумматоре).

(1) Если значение α уже запомнено в некотором регистре i, то не перевычисляем α. Регистр i находится теперь „в употреблении".

(2) Если значение α не запомнено ни в каком регистре, запоминаем его в очередном свободном регистре, скажем j. Регистр j находится теперь в употреблении. Если нет свободных доступных регистров, то содержимое некоторого регистра k запоминаем в основной памяти, а значение α запоминаем в регистре k. Регистр k выбираем так, чтобы к его значению как можно дольше не было обращения.

Бледи [1966] показал, что в некоторых ситуациях этот алгоритм оптимален. Однако этот алгоритм (разработанный для листования) предполагает модель, не совпадающую в точности с моделью линейного кода. В частности, порядок вычислений считается фиксированным, в то время как в разд. 11.1 и 11.2 было показано, что, переупорядочивая вычисления, можно добиться значительного выигрыша.

Аналогичная задача распределения регистров исследуется в работе Хорвица и др. [1966]. Там предполагается, что дана последовательность операций, обращающаяся к значениям и изменяющая их. Задача заключается в том, чтобы занести эти значения в быстрые регистры так, чтобы число записей и считываний из быстрых регистров в основную память было минимальным. Их решение сводится к выбору пути с минимальной оценкой на ориентированном ациклическом графе возможных решений. Даются методы уменьшения размера графа. Дальнейшее исследование распределения регистров, когда порядок вычислений не фиксирован, проведено Кеннеди [1972] и Сети [1972].

Перевод арифметических выражений в код для параллельных машин излагается Аллардом и др. [1964], Хеллерманом [1966], Стоуном [1967], Бэром и Бове [1968]. Общая задача оптимального распределения задач между параллельными процессорами очень трудна. Некоторые ее интересные аспекты обсуждаются Грэхемом [1972] [1]).

11.3. ПРОГРАММЫ С ЦИКЛАМИ

При рассмотрении программ, содержащих циклы, мы не можем осуществлять автоматическую оптимизацию. Основные трудности связаны с проблемами неразрешимости. Для двух произвольных программ не существует алгоритма, выясняющего, эквивалентны ли они в каком-нибудь смысле. Как следствие этого не существует алгоритма, который находил бы по данной программе эквивалентную ей и оптимальную относительно какой-нибудь оценки.

Эти результаты становятся понятными, если вспомнить, что существует, вообще говоря, сколько угодно способов вычисления одной и той же функции. Таким образом, существует бесконечное множество алгоритмов, которые можно использовать для реализации функции, определяемой исходной программой. Если мы хотим получить полную оптимизацию, то компилятор должен быть способен найти наиболее эффективный алгоритм для функции, вычисляемой исходной программой, а затем сгенерировать для него наиболее эффективный код. Излишне говорить, что как с теоретической, так и с практической точек зрения таких оптимизирующих компиляторов не может быть.

Тем не менее во многих ситуациях можно указать набор преобразований, применимых к исходной программе для уменьшения размера и/или увеличения скорости результирующей объектной программы. В этом разделе мы исследуем несколько таких преобразований. Благодаря их широкому распространению, они стали известны как „оптимизирующие" преобразования. Точнее было бы называть их преобразованиями „улучшения кода". Однако мы будем следовать традиции и пользоваться более популярным, хотя и менее точным термином „оптимизирующее преобразование". Главной нашей целью будет уменьшение времени выполнения объектной программы.

Начнем с определения промежуточных программ с циклами. Эти программы будут весьма примитивными, так что основные концепции мы изложим, не вдаваясь в кучу мелких подробностей. Затем определим граф управления программы. Граф управле-

[1]) Оптимизация арифметических выражений, включающая выделение общих подвыражений и такое преобразование программы, что эти общие подвыражения вычисляются только один раз, описана в работах: [Ершов, 1958б], [Камынин, Любимский, Шура-Бура, 1958], [Ершов, 1958а], [Китов, Криницкий, 1959], [Ершова, Мостинская, Руднева, 1961], [Поттосин, 1965]. При этом полностью учитывалась коммутативность сложения и умножения. В работах Ершова [1958б] и Поттосина [1965] приведены алгоритмы экономии выражений, работающие за линейное время. — *Прим. перев.*

ния — это двумерное представление программы, отражающее порядок выполнения операторов программы. Обычно двумерная структура дает более наглядное представление, нежели линейная последовательность операторов.

В оставшейся части этого раздела мы опишем ряд важных преобразований, применимых к программе с целью уменьшить время выполнения объектной программы. В следующем разделе рассмотрим, как собрать вместе всю информацию, необходимую для применения некоторых из преобразований, описанных в этом разделе.

11.3.1. Модель программы

Мы будем использовать для программ представление, промежуточное между исходной программой и языком ассемблера. Программа состоит из последовательности операторов. Каждый оператор можно пометить идентификатором, за которым следует двоеточие. Существует пять основных типов операторов: присваивание, переход, условный, ввод-вывод и останов.

(1) *Оператор присваивания* — это цепочка вида $A \leftarrow \theta B_1 \ldots B_r$, где A — переменная, $B_1, \ldots, B_r$ — переменные или константы, θ — r-местная операция. Как и в предыдущих разделах, для бинарных операций обычно будем пользоваться инфиксной записью. Оператор вида $A \leftarrow B$ также будем относить к этой категории.

(2) *Оператор перехода* — это цепочка вида

goto ⟨метка⟩

где ⟨метка⟩ — цепочка букв. Будем предполагать, что если в программе употребляется оператор перехода, то метка, следующая за **goto**, появляется в качестве метки только одного оператора программы.

(3) *Условный оператор* имеет вид

if A ⟨отношение⟩ B **goto** ⟨метка⟩

где A и B — переменные или константы, а ⟨отношение⟩ — бинарное отношение, такое, как $<$, $\leqslant$, $=$, $\neq$.

(4) *Оператор ввода-вывода* — это либо *оператор чтения* вида

read A

где A — переменная, либо *оператор записи* вида

write B

где B — переменная или константа. Для удобства последовательность операторов

read A_1
read A_2
.
.
.
read A_n

будем изображать в виде

read $A_1, A_2, \ldots, A_n$

Аналогичное соглашение у нас будет и для операторов записи.

(5) Наконец, *оператор останова* — это команда **halt**.

Интуитивный смысл операторов каждого типа должен быть ясен. Например, условный оператор вида

if $A\,r\,B$ **goto** L

означает, что если между текущими значениями A и B выполняется отношение r, то управление должно передаваться на оператор, помеченный L. В противном случае управление переходит к следующему оператору.

Оператор определения (или просто *определение*) — это оператор вида **read** A или $A \leftarrow \theta B_1 \ldots B_r$. Про оба эти оператора говорят, что они *определяют* переменную A.

Сделаем еще несколько предположений относительно программ. Переменные — это либо простые переменные, например $A, B, C, \ldots$, либо переменные, индексированные одной простой переменной или константой, например $A(1)$, $A(2)$, $A(I)$ или $A(J)$. Далее будем считать, что все переменные, на которые в программе есть ссылки, либо должны быть входными переменными (т. е. появляться в предыдущем операторе чтения), либо должны ранее определяться с помощью оператора присваивания. Наконец, будем предполагать, что каждая программа содержит хотя бы один оператор останова и, если программа заканчивается, последний выполненный оператор — это оператор останова.

Выполнение программы начинается с первого оператора программы и продолжается, пока не встретится оператор останова. Предполагается, что каждая переменная имеет известный тип (например, целое, вещественное) и ее *значение* в любой момент в процессе выполнения либо не определено, либо представляет собой некоторую величину подходящего типа. (Будем считать, что все используемые операторы соответствуют типам переменных, к которым они применяются, и, когда надо, происходит преобразование типов.)

Вообще говоря, входные переменные программы — это те, которые связаны с операторами чтения, а выходные — с операторами записи. Присваивание значения каждой входной переменной при каждом чтении называется *входным присваиванием*. *Значение программы* при некотором входном присваивании — это последовательность значений, записанных выходными переменными в процессе выполнения программы. Две программы будем считать *эквивалентными*, если для каждого входного присваивания они имеют одинаковые значения [1]).

Это определение эквивалентности обобщает определение эквивалентности блоков (разд. 11.1). Чтобы убедиться в этом, предположим, что два блока $\mathcal{B}_1=(P_1, I_1, U_1)$ и $\mathcal{B}_2=(P_2, I_2, U_2)$ эквивалентны в смысле разд. 11.1. Преобразуем очевидным образом блоки $\mathcal{B}_1$ и $\mathcal{B}_2$ в программы $\mathcal{P}_1$ и $\mathcal{P}_2$. Иными словами, поставим операторы чтения для переменных в I_1 и I_2 перед P_1 и P_2 соответственно, а операторы записи для переменных в U_1 и U_2 — после P_1 и P_2. Затем к каждой программе присоединим оператор останова. Операторы записи к P_1 и P_2 нужно добавлять так, чтобы каждая выходная переменная печаталась хотя бы один раз и последовательности напечатанных значений для $\mathcal{P}_1$ и $\mathcal{P}_2$ были одинаковыми. Поскольку блоки $\mathcal{B}_1$ и $\mathcal{B}_2$ эквивалентны, это всегда можно сделать.

Легко видеть, что программы $\mathcal{P}_1$ и $\mathcal{P}_2$ эквивалентны независимо от того, что именно берется в качестве множества входных присваиваний и как интерпретируются функции, представленные операциями, входящими в $\mathcal{P}_1$ и $\mathcal{P}_2$. Например, можно взять в качестве входного множество префиксных выражений и считать, что применение операции θ к выражениям $\varepsilon_1, \ldots, \varepsilon_r$ дает $\theta\varepsilon_1\ldots\varepsilon_r$.

Однако, если $\mathcal{B}_1$ и $\mathcal{B}_2$ не эквивалентны, то всегда найдется множество типов данных для переменных и интерпретации для операций, приводящие к тому, что соответствующие блокам программы $\mathcal{P}_1$ и $\mathcal{P}_2$ будут вырабатывать различные выходные последовательности. В частности, пусть „типом" переменных будут префиксные выражения, а результатом применения операции θ к префиксным выражениям $\varepsilon_1, \ldots, \varepsilon_k$ будет префиксное выражение $\theta\varepsilon_1\ldots\varepsilon_k$.

Ясно, что относительно типов данных и алгебры, связанной с функциями и знаками отношений, можно сделать такие предположения, что программы $\mathcal{P}_1$ и $\mathcal{P}_2$ окажутся эквивалентными.

[1]) Предполагается, что смысл каждой операции и знака отношения, а также тип данных каждой переменной зафиксирован. Таким образом, это понятие эквивалентности отличается от понятия, введенного ранее, например Патерсоном [1968] или Лакхэмом и др. [1970], тем, что они требуют, чтобы две программы давали одинаковые значения не только для каждого входного присваивания, но и для каждого типа данных для переменных и каждого множества функций и отношений, которые мы подставляем вместо операций и знаков отношений.

В таком случае блоки $\mathcal{B}_1$ и $\mathcal{B}_2$ будут эквивалентными относительно соответствующего множества алгебраических законов.

Пример 11.26. Рассмотрим следующую программу для алгоритма Евклида, описанного в гл. 0 (том 1). Выходом должен быть наибольший общий делитель двух положительных целых чисел p и q.

```
        read p
        read q
цикл:   r ← remainder (p, q)
        if r = 0 goto выход
        p ← q
        q ← r
        goto цикл
выход:  write q
        halt
```

Если, например, входным переменным p и q присвоить значения 72 и 56 соответственно, то при обычной интерпретации операций выходная переменная q к моменту выполнения оператора записи будет иметь значение 8. Таким образом, значением этой программы для входного присваивания $p \leftarrow 72$, $q \leftarrow 56$ служит „последовательность" 8, вырабатываемая выходной переменной q.

Если заменить оператор **goto** *цикл* на **if** $q \neq 0$ **goto** *цикл*, то получим эквивалентную программу. Это следует из того, что оператора **goto** *цикл* нельзя достичь, если в четвертом операторе не выполняется условие $r \neq 0$. Поскольку в шестом операторе q принимает значение r, то при выполнении седьмого оператора равенство $q=0$ невозможно. □

Следует отметить, что преобразования, которые мы можем применять, в значительной степени определяются теми алгебраическими законами, которые мы считаем справедливыми.

Пример 11.27. Для некоторых типов данных можно считать, что $a*a=0$ тогда и только тогда, когда $a=0$. Если принять такой закон, то программе из примера 11.26 эквивалентна программа

```
        read p
        read q
цикл:   r ← remainder (p, q)
        t ← r * r
        if t = 0 goto выход
        p ← q
        q ← r
        goto цикл
выход:  write q
        halt
```

Конечно, при любых мыслимых обстоятельствах эта программа ничем не лучше, но если сформулированный выше закон неверен, то эта программа и программа из примера 11.26 могут оказаться и не эквивалентными. □

Если дана программа P, то наша цель заключается в нахождении эквивалентной программы P', для которой ожидаемое время выполнения в машинном языке меньше, чем для P. Разумным приближением нашей задачи будет нахождение такой эквивалентной программы P'', что ожидаемое число команд машинного языка, которые предстоит выполнить в P'', меньше числа команд, выполняемых в P. Эта задача является приближением, поскольку различные машинные команды могут выполняться за различное машинное время. Например, такая операция, как умножение или деление, обычно занимают больше времени, чем сложение или вычитание. Тем не менее мы сначала займемся уменьшением только числа выполняемых команд машинного языка.

В большинстве программ одни последовательности операторов исполняются значительно чаще, чем другие. Кнут [1971] на большом числе программ Фортрана обнаружил, что в типичной программе около половины времени тратится менее чем на 4% программы. Таким образом, на практике часто достаточно применять оптимизирующие процедуры только к этим многократно проходимым участкам программы. В частности, оптимизация может состоять в том, что операторы перемещаются из многократно проходимых областей в области, редко проходимые, а число операторов в самой программе не меняется или даже увеличивается.

Во многих случаях можно определить, какой кусок исходной программы будет выполняться чаще других, и вместе с исходной программой передать эту информацию оптимизирующему компилятору. В других случаях довольно просто написать подпрограмму, подсчитывающую, сколько раз исполняется данный оператор. С помощью такого счетчика можно получить „частотный профиль" программы и выявить те ее куски, на которые надо направить основные усилия по оптимизации.

11.3.2. Анализ потока управления

Первый шаг на пути оптимизации программ заключается в определении потока управления внутри программы. Для того чтобы сделать это, разобьем программу на группы операторов так, чтобы внутри группы управление передавалось только на первый оператор, и если уж он выполнился, все остальные операторы группы выполняются последовательно. Такую группу операторов будем называть *линейным участком* или просто *участком*.

Определение. Оператор S в программе P называется *входом в линейный участок*, если он

(1) первый оператор в P или

(2) помечен идентификатором, появляющимся после **goto** в операторе перехода либо в условном операторе, или

(3) непосредственно следует за условным оператором.

Линейный участок, относящийся к входу в участок S, состоит из S и всех операторов, следующих за S,

(1) вплоть до оператора останова и включая его или

(2) вплоть до входа в следующий участок, но не включая его.

Отметим, что программа, построенная из блока в смысле разд. 11.1, будет линейным участком в смысле настоящего раздела.

Пример 11.28. Рассмотрим программу из примера 11.26. В ней четыре входа в участок, а именно первый оператор программы, оператор, помеченный *цикл*, оператор присваивания $p \leftarrow q$ и оператор, помеченный *выход*.

Таким образом, в этой программе четыре линейных участка:

```
Участок 1                read p
                         read q
Участок 2   цикл:        r ← remainder (p, q)
                         if r = 0 goto выход
Участок 3                p ← q
                         q ← r
                         goto цикл
Участок 4   выход:       write q
                         halt   □
```

Из участков программы можно сконструировать граф, весьма похожий на блок-схему программы.

Определение. *Графом управления* назовем помеченный ориентированный граф G, содержащий выделенную вершину n, из которой достижима каждая вершина в G. Вершину n назовем *начальной*.

Граф управления программы—это граф управления, в котором каждая вершина соответствует какому-нибудь участку программы. Предположим, что вершины i и j графа управления соответствуют участкам i и j программы. Тогда дуга идет из вершины i в вершину j, если

(1) последний оператор участка i не является ни оператором перехода, ни оператором останова, а участок j следует в программе за участком i, или

(2) последний оператор участка i является оператором **goto** L либо оператором **if** ... **goto** L, где L — метка первого оператора участка j.

Участок, содержащий первый оператор программы, назовем *начальной вершиной*.

Ясно, что любой участок, недостижимый из начальной вершины, можно удалить из данной программы, не меняя ее значения. Отныне будем считать, что все такие участки рассматриваемой программы уже удалены.

Пример 11.29. Граф управления программы[1]) примера 11.26 изображен на рис. 11.22. Начальной вершиной является участок 1. □

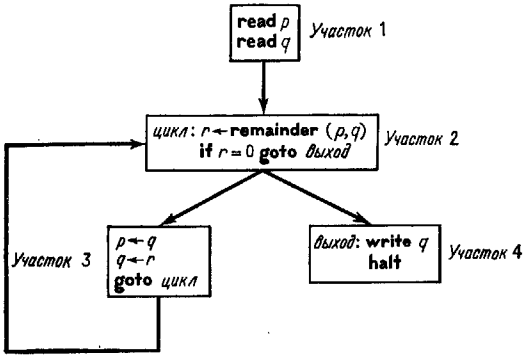


Рис. 11.22. Граф управления.

Многие оптимизирующие преобразования программ требуют знания тех мест в программе, где переменная определяется, и тех, где на ее определение есть ссылка. Эти связи между определением и ссылкой зависят от последовательностей выполняемых участков. Первым участком в такой последовательности будет начальная вершина, а в каждый последующий участок должна вести дуга из предыдущего. Иногда предикаты, используемые в условных операторах, могут запрещать проход по некоторым путям в графе управления. Однако алгоритма для выявления всех таких ситуаций нет, и мы будем предполагать, что нет „запрещенных" путей.

Удобно также знать, существует ли для участка $\mathcal{B}$ такой участок $\mathcal{B}'$, что всякий раз, когда выполняется $\mathcal{B}$, перед ним

[1]) Везде в дальнейшем граф управления программы будем называть для краткости просто графом управления.— *Прим. перев.*

выполняется $\mathcal{B}'$. В частности, если мы знаем это и если в обоих участках $\mathcal{B}$ и $\mathcal{B}'$ вычисляется одно и то же значение, можно запомнить его после вычисления в $\mathcal{B}'$ и избежать тем самым перевычисления его в $\mathcal{B}$. Формализуем теперь эти идеи.

Определение. Пусть F — граф управления, имена участков которого выбираются из некоторого множества Δ. Последовательность участков $\mathcal{B}_1 \ldots \mathcal{B}_n$ из Δ^* назовем *путем вычислений* (*участков*) в F, если

(1) $\mathcal{B}_1$ — начальная вершина в F,

(2) для $1 < i \leqslant n$ существует дуга, ведущая из $\mathcal{B}_{i-1}$ в $\mathcal{B}_i$.

Другими словами, путь вычислений $\mathcal{B}_1 \ldots \mathcal{B}_n$ — это путь в F из $\mathcal{B}_1$ в $\mathcal{B}_n$, в котором $\mathcal{B}_1$ — начальная вершина.

Будем говорить, что участок $\mathcal{B}'$ *доминирует* над участком $\mathcal{B}$, если $\mathcal{B}' \neq \mathcal{B}$ и каждый путь, ведущий из начальной вершины в $\mathcal{B}$, содержит $\mathcal{B}'$. Будем говорить, что $\mathcal{B}'$ *прямо доминирует* над $\mathcal{B}$, если

(1) $\mathcal{B}'$ доминирует над $\mathcal{B}$ и

(2) если $\mathcal{B}''$ доминирует над $\mathcal{B}$ и $\mathcal{B}'' \neq \mathcal{B}'$, то $\mathcal{B}''$ доминирует над $\mathcal{B}'$.

Таким образом, участок $\mathcal{B}'$ прямо доминирует над $\mathcal{B}$, если $\mathcal{B}'$ — „ближайший" к $\mathcal{B}$ участок, доминирующий над $\mathcal{B}$.

Пример 11.30. На рис. 11.22 последовательность 1232324 — это путь вычислений. Участок 1 прямо доминирует над участком 2 и доминирует над участками 3 и 4. Участок 2 прямо доминирует над участками 3 и 4. □

Приведем некоторые алгебраические свойства отношения доминирования.

Лемма 11.13. *Если $\mathcal{B}_1$ доминирует над $\mathcal{B}_2$, а $\mathcal{B}_2$ доминирует над $\mathcal{B}_3$, то $\mathcal{B}_1$ доминирует над $\mathcal{B}_3$* (*транзитивность*).

(2) *Если $\mathcal{B}_1$ доминирует над $\mathcal{B}_2$, то $\mathcal{B}_2$ не доминирует над $\mathcal{B}_1$* (*асимметричность*).

(3) *Если $\mathcal{B}_1$ и $\mathcal{B}_2$ доминируют над $\mathcal{B}_3$, то либо $\mathcal{B}_1$ доминирует над $\mathcal{B}_2$, либо $\mathcal{B}_2$ доминирует над $\mathcal{B}_1$.*

Доказательство. Утверждения (1) и (2) оставляем в качестве упражнений. Докажем (3). Пусть $\mathcal{C}_1 \ldots \mathcal{C}_n \mathcal{B}_3$ — любой путь вычислений без циклов (т. е. $\mathcal{C}_i \neq \mathcal{B}_3$ и $\mathcal{C}_i \neq \mathcal{C}_j$ при $i \neq j$). Один такой путь существует, так как мы предполагали, что все вершины достижимы из начальной вершины. По условию $\mathcal{C}_i = \mathcal{B}_1$ и $\mathcal{C}_j = \mathcal{B}_2$ для некоторых i и j. Без потери общности можно считать, что $i < j$. Мы утверждаем, что $\mathcal{B}_1$ доминирует над $\mathcal{B}_2$.

Предположим противное, т. е. что $\mathcal{B}_1$ не доминирует над $\mathcal{B}_2$. Тогда найдется путь вычислений $\mathcal{D}_1 \ldots \mathcal{D}_m \mathcal{B}_2$, в котором среди $\mathcal{D}_1, \ldots, \mathcal{D}_m$ нет $\mathcal{B}_1$. Отсюда следует, что $\mathcal{D}_1 \ldots \mathcal{D}_m \mathcal{B}_2 \mathcal{C}_{j+1} \ldots \mathcal{C}_n \mathcal{B}_3$ — также путь вычислений. Но среди символов, предшествующих $\mathcal{B}_3$, нет $\mathcal{B}_1$, а это противоречит предположению о том, что $\mathcal{B}_1$ доминирует над $\mathcal{B}_3$. □

Лемма 11.14. *Каждый участок, кроме начальной вершины (не имеющей доминаторов), имеет единственный прямой доминатор.*

Доказательство. Пусть $\mathcal{S}$ — множество участков, доминирующих над некоторым участком $\mathcal{B}$. По лемме 11.13 отношение доминирования устанавливает (строгий) линейный порядок на $\mathcal{S}$. Таким образом, $\mathcal{S}$ обладает наименьшим элементом, который должен быть прямым доминатором участка $\mathcal{B}$ (см. упр. 0.1.23). □

Изложим теперь алгоритм, вычисляющий отношение прямого доминирования для графов управления.

Алгоритм 11.5. Вычисление прямого доминирования.

Вход. Граф управления F и множество $\Delta = \{\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n\}$ его участков. Предполагается, что $\mathcal{B}_1$ — начальная вершина.

Выход. Прямой доминатор $\mathrm{DOM}(\mathcal{B})$ участка $\mathcal{B}$ для каждого $\mathcal{B} \in \Delta$, кроме начальной вершины.

Метод. $\mathrm{DOM}(\mathcal{B})$ вычисляется рекурсивно для каждого $\mathcal{B}$ из $\Delta - \{\mathcal{B}_1\}$. В любой момент $\mathrm{DOM}(\mathcal{B})$ будет участком, ближайшим к $\mathcal{B}$ среди всех участков, для которых уже известно, что они доминируют над $\mathcal{B}$. В конечном итоге $\mathrm{DOM}(\mathcal{B})$ будет прямым доминатором участка $\mathcal{B}$. Вначале $\mathrm{DOM}(\mathcal{B})$ — это $\mathcal{B}_1$ для всех $\mathcal{B}$ из $\Delta - \{\mathcal{B}_1\}$. Для $i = 2, 3, \ldots, n$ выполняем следующие два шага:

(1) Исключаем участок $\mathcal{B}_i$ из F. С помощью алгоритма 0.3 находим все участки $\mathcal{B}$, ставшие теперь недостижимыми из начальной вершины в F. Участок $\mathcal{B}_i$ доминирует над $\mathcal{B}$ тогда и только тогда, когда $\mathcal{B}$ становится недостижимым из начальной вершины после исключения $\mathcal{B}_i$ из F. Снова заносим $\mathcal{B}_i$ в F.

(2) Предположим, что на шаге (1) обнаружено, что $\mathcal{B}_i$ доминирует над $\mathcal{B}$. Если $\mathrm{DOM}(\mathcal{B}) = \mathrm{DOM}(\mathcal{B}_i)$, берем $\mathcal{B}_i$ в качестве $\mathrm{DOM}(\mathcal{B})$. В противном случае $\mathrm{DOM}(\mathcal{B})$ не меняем. □

Пример 11.31. С помощью алгоритма 11.5 вычислим прямые доминаторы для графа управления рис. 11.22. Здесь $\Delta = \{\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4\}$. Последовательные значения $\mathrm{DOM}(\mathcal{B})$ после исследования участков $\mathcal{B}_i$, $2 \leqslant i \leqslant 4$, приведены в табл. 11.8. Вычислим строку 2. После исключения участка $\mathcal{B}_2$ участки $\mathcal{B}_3$ и $\mathcal{B}_4$ становятся недостижимыми. Таким образом, $\mathcal{B}_2$ доминирует над $\mathcal{B}_3$ и $\mathcal{B}_4$. Перед этим было $\mathrm{DOM}(\mathcal{B}_2) = \mathrm{DOM}(\mathcal{B}_3) = \mathcal{B}_1$, так что в

Таблица 11.8

i	$\mathrm{DOM}(\mathcal{B}_2)$	$\mathrm{DOM}(\mathcal{B}_3)$	$\mathrm{DOM}(\mathcal{B}_4)$
Вначале	$\mathcal{B}_1$	$\mathcal{B}_1$	$\mathcal{B}_1$
2	$\mathcal{B}_1$	$\mathcal{B}_2$	$\mathcal{B}_2$
3	$\mathcal{B}_1$	$\mathcal{B}_2$	$\mathcal{B}_2$
4	$\mathcal{B}_1$	$\mathcal{B}_2$	$\mathcal{B}_2$

соответствии с шагом (2) алгоритма 11.5 берем $\mathcal{B}_2$ в качестве $\mathrm{DOM}(\mathcal{B}_3)$. Аналогично полагаем $\mathcal{B}_2 = \mathrm{DOM}(\mathcal{B}_4)$. Исключение участков $\mathcal{B}_3$ и $\mathcal{B}_4$ не делает никакой участок недостижимым, так что никаких изменений больше не требуется. □

Теорема 11.10. *По окончании работы алгоритма 11.5* $\mathrm{DOM}(\mathcal{B})$ — *прямой доминатор участка* $\mathcal{B}$.

Доказательство. Заметим сначала, что на шаге (1) правильно определяются те участки $\mathcal{B}$, над которыми доминирует $\mathcal{B}_i$, поскольку $\mathcal{B}_i$ доминирует над $\mathcal{B}$ тогда и только тогда, когда любой путь в $\mathcal{B}$ из начальной вершины F проходит через $\mathcal{B}_i$.

Индукцией по i покажем, что после шага (2) $\mathrm{DOM}(\mathcal{B})$ — это участок $\mathcal{B}_h$, $1 \leqslant h \leqslant i$, который доминирует над $\mathcal{B}$ и над которым в свою очередь доминируют все $\mathcal{B}_j$, $1 \leqslant j \leqslant i$, также доминирующие над $\mathcal{B}$. Существование такого участка $\mathcal{B}_h$ вытекает непосредственно из леммы 11.13. Базис, $i = 2$, тривиален.

Перейдем к доказательству шага индукции. Если $\mathcal{B}_{i+1}$ не доминирует над $\mathcal{B}$, то заключение вытекает непосредственно из предположения индукции. Если $\mathcal{B}_{i+1}$ доминирует над $\mathcal{B}$, но существует такой участок $\mathcal{B}_j$, $1 \leqslant j \leqslant i$, что $\mathcal{B}_j$ доминирует над $\mathcal{B}$, а $\mathcal{B}_{i+1}$ доминирует над $\mathcal{B}_j$, то $\mathrm{DOM}(\mathcal{B}) \neq \mathrm{DOM}(\mathcal{B}_{i+1})$. Таким образом, $\mathrm{DOM}(\mathcal{B})$ не меняется, что вполне соответствует предположению индукции. Если $\mathcal{B}_{i+1}$ доминирует над $\mathcal{B}$ и доминируется всеми $\mathcal{B}_j$, доминирующими над $\mathcal{B}$, $1 \leqslant j \leqslant i$, то перед этим шагом $\mathrm{DOM}(\mathcal{B}) = \mathrm{DOM}(\mathcal{B}_{i+1})$. В самом деле, если бы это было не так, то нашелся бы участок $\mathcal{B}_k$, $1 \leqslant k \leqslant i$, доминирующий над $\mathcal{B}_{i+1}$, но не над $\mathcal{B}$, что невозможно по лемме 11.13(1). Следовательно, $\mathrm{DOM}(\mathcal{B})$ принимает именно значение $\mathcal{B}_{i+1}$. Теорема доказана. □

Отметим, что если F строится из программы, то число дуг не более чем вдвое превосходит число участков. Поэтому шаг (1) алгоритма 11.5 выполняется за время, пропорциональное квадрату числа участков. Требуемая емкость пропорциональна числу участков.

Если $\mathscr{B}_1, \mathscr{B}_2, \ldots, \mathscr{B}_n$ — участки программы (кроме начального), то их доминаторы можно запоминать как последовательность $\mathscr{C}_1, \mathscr{C}_2, \ldots, \mathscr{C}_n$, где $\mathscr{C}_i$ — прямой доминатор для $\mathscr{B}_i$, $1 \leqslant i \leqslant n$. Все доминаторы для $\mathscr{B}_i$ легко выбрать из этой последовательности, вычисляя DOM($\mathscr{B}_i$), DOM(DOM($\mathscr{B}_i$)) и т. д. вплоть до начального участка.

11.3.3. Примеры преобразований программ

Займемся теперь преобразованиями, которые можно применить к программе или ее графу управления, чтобы уменьшить время выполнения получающейся в конце концов объектной программы. В этом разделе мы рассмотрим примеры таких преобразований. Хотя полного каталога оптимизирующих преобразований программ с циклами не существует, рассматриваемые здесь преобразования полезны для широкого класса программ.

1. Удаление бесполезных операторов

Это — обобщение преобразования T_1 из разд. 11.1. Без оператора, не влияющего на значение программы, можно обойтись, так что его можно удалить. Линейные участки, недостижимые из начальной вершины, очевидно, бесполезны, и их можно удалить. Операторы, вычисляющие значения, не используемые в конечном итоге при вычислении выходной переменной, также попадают в эту категорию. В разд. 11.4 мы опишем технику применения этих преобразований к программам с циклами.

2. Исключение избыточных вычислений

Это преобразование обобщает преобразование T_2 из разд. 11.1. Предположим, что у нас есть программа, в которой участок $\mathscr{B}$ доминирует над $\mathscr{B}'$, и что $\mathscr{B}$ и $\mathscr{B}'$ содержат операторы $A \leftarrow B+C$ и $A' \leftarrow B+C$ соответственно. Если ни B, ни C не переопределены на каком-нибудь (не обязательно без циклов) пути из $\mathscr{B}$ в $\mathscr{B}'$ (выяснить это нетрудно; см. упр. 11.3.5), то значения, вычисляемые этими двумя выражениями, совпадают. Тогда в $\mathscr{B}$ после $A \leftarrow B+C$ можно вставить оператор $X \leftarrow A$, где X — новая переменная. Затем $A' \leftarrow B+C$ можно заменить на $A' \leftarrow X$. Кроме того, если A нигде на пути из $\mathscr{B}$ в $\mathscr{B}'$ не переопределяется, то оператор $X \leftarrow A$ не нужен, а $A' \leftarrow B+C$ можно заменить на $A' \leftarrow A$.

Здесь предполагается, что выгоднее сделать два присваивания $X \leftarrow A$ и $A' \leftarrow X$, чем вычислять $A' \leftarrow B+C$; это предположение справедливо для многих моделей машин.

Пример 11.32. Рассмотрим граф управления, изображенный на рис. 11.23. В этом графе участок $\mathscr{B}_1$ доминирует над $\mathscr{B}_2$, $\mathscr{B}_3$ и $\mathscr{B}_4$. Предположим, что все операторы присваивания, в которые входят переменные A, B, C и D, таковы, как показано на рис. 11.23. Тогда $B+C$ принимает одно и то же значение

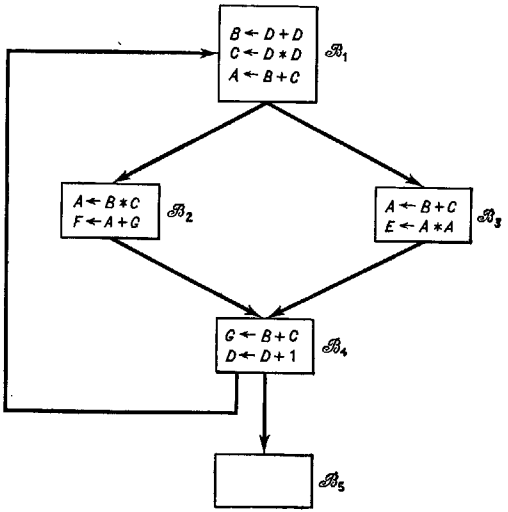


Рис. 11.23. Граф управления.

при вычислениях в участках $\mathscr{B}_1$, $\mathscr{B}_3$ и $\mathscr{B}_4$. Поэтому в $\mathscr{B}_3$ и $\mathscr{B}_4$ не обязательно перевычислять выражение $B+C$. В $\mathscr{B}_1$ после оператора $A \leftarrow B+C$ можно поместить оператор присваивания $X \leftarrow A$. Здесь X — новое имя переменной. Тогда в $\mathscr{B}_3$ и $\mathscr{B}_4$ операторы $A \leftarrow B+C$ и $G \leftarrow B+C$ можно заменить на простые операторы присваивания $A \leftarrow X$ и $G \leftarrow X$ соответственно, и это не повлияет на значение программы. Отметим, что, поскольку A вычисляется в $\mathscr{B}_2$, вместо X нельзя использовать A. Преобразованный таким образом граф управления изображен на рис. 11.24.

Теперь в $\mathscr{B}_3$ присваивание $A \leftarrow X$ становится избыточным и его можно исключить. Отметим также, что если в $\mathscr{B}_2$ оператор $F \leftarrow A+G$ заменить на $B \leftarrow A+G$, то в $\mathscr{B}_4$ нельзя уже заменять $G \leftarrow B+C$ на $G \leftarrow X$. □

Для исключения из программы избыточных вычислений (общих подвыражений) надо выявить вычисления, общие для двух

или более участков программы. Избыточные вычисления, общие для участка и какого-нибудь из его доминаторов, мы уже рассмотрели. Выражения типа $A+B$ могут вычисляться в нескольких участках, ни один из которых не доминирует над данным участком $\mathscr{B}$ (где также требуется выражение $A+B$). Вообще

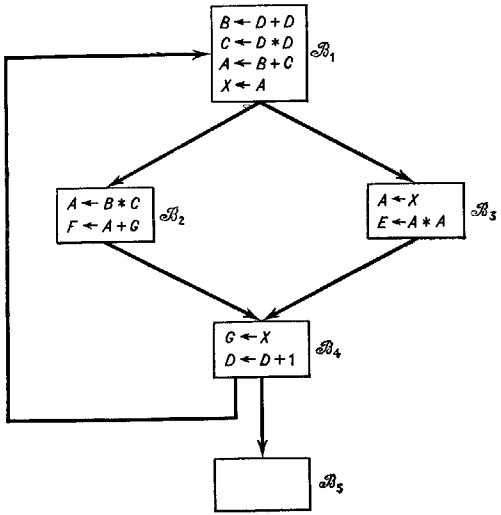


Рис. 11.24. Преобразованный граф управления.

вычисление выражения $A+B$ считается *избыточным* в участке $\mathscr{B}$, если

(1) любой путь из начального участка в $\mathscr{B}$ (в том числе и проходящий через $\mathscr{B}$ несколько раз) проходит через вычисление $A+B$,

(2) вдоль любого такого пути между последним вычислением $A+B$ и использованием $A+B$ в $\mathscr{B}$ не встречается определения ни A, ни B.

В разд. 11.4 мы изложим некоторые методы определения этой наиболее общей ситуации.

Отметим, что как и в линейном случае, применение алгебраических законов может увеличить число общих подвыражений.

3. Замена вычислений периода выполнения вычислениями периода компиляции

Если это возможно, то имеет смысл выполнить вычисление один раз при компиляции, а не повторять его многократно при исполнении объектной программы. Простой пример — *размножение констант*, т. е. замена переменной на константу, когда значение переменной постоянно и известно.

Пример 11.33. Рассмотрим участок

```
read R
PI ← 3.14159
A ← 4/3
B ← A * PI
C ← R↑3
V ← B*C
write V
```

В четвером операторе вместо PI можно подставить значение 3.14159 и получить оператор $B \leftarrow A*3.14159$. Можно также вычислить 4/3 и, подставив найденное значение в $B \leftarrow A*3.14159$, получить $B \leftarrow 1.33333*3.14159$. Можно вычислить $1.33333*3.14159 = 4.18878$ и, подставив 4.18878 в оператор $V \leftarrow B*C$, получить $V \leftarrow 4.18878*C$. Наконец, можно удалить получившиеся бесполезные операторы. В результате у нас будет более короткая эквивалентная программа

```
read R
C ← R↑3
V ← 4.18878 * C
write V    □
```

4. Замена сложных операций

Замена сложных операций представляет собой замещение одной операции, занимающей довольно много машинного времени, более быстрой последовательностью. Например, пусть исходная программа на ПЛ/I содержит оператор

$$I = \text{LENGTH}(S1 \| S2)$$

где $S1$ и $S2$ — цепочки переменной длины. $\|$ означает конкатенацию цепочек. Реализовать конкатенацию цепочек довольно сложно. Предположим, однако, что мы заменяем этот оператор эквивалентным оператором

$$I = \text{LENGTH}(S1) + \text{LENGTH}(S2)$$

Теперь мы должны дважды выполнить операцию определения длины и один раз сложение. Но эти операции занимают существенно меньше времени, чем конкатенация цепочек.

Другие примеры оптимизации такого типа: замена некоторых умножений сложениями и замена некоторых возведений в степень повторными умножениями. Например, оператор $C \leftarrow R\uparrow 3$ можно заменить последовательностью

$$C \leftarrow R * R$$
$$C \leftarrow C * R$$

при условии, что дешевле вычислить $R * R * R$, чем вызывать подпрограммы для вычисления R^3 как ANTILOG(3 * LOG(R)).

В следующем разделе мы изучим более интересную форму замены сложных операций в циклах, когда есть возможность заменять некоторые умножения сложениями.

11.3.4. Оптимизация циклов

Грубо говоря, цикл в программе — это последовательность участков, которая может исполняться повторно. Циклы присущи большинству программ, а многие программы имеют циклы, исполняемые много раз. Во многих языках программирования есть конструкции, предназначенные специально для организации циклов. Часто можно добиться существенных улучшений в смысле времени исполнения программы, применяя преобразования, уменьшающие только оценку циклов. Универсальные преобразования, которые мы только что рассматривали, а именно удаление бесполезных операторов, исключение избыточных вычислений, размножение констант, замена сложных операций, полезны, в частности, и в применении к циклам. Существуют, однако, и другие преобразования, ориентированные специально на циклы. Это вынесение вычислений из циклов, замена дорогих операций в циклах более дешевыми и развертывание циклов.

Для того чтобы применить эти преобразования, цикл сначала надо выделить из данной программы. В случае циклов DO в Фортране или промежуточного кода, образуемого из цикла DO, найти цикл просто. Однако понятие цикла в графе управления более общо, чем понятие цикла, возникающего из операторов DO Фортрана. Эти обобщенные циклы графов управления называют „сильно связными областями". Любой цикл графа управления [1]) с единственной точкой входа служит примером сильно связной области. Однако и более общие структуры циклов также служат примерами сильно связных областей. Дадим определение сильно связной области.

[1]) Подчеркиваем: „цикл графа", т. е. замкнутый путь в графе. — *Прим. ред.*

Определение. Пусть F — граф управления, а $\mathscr{S}$ — подмножество его участков. Будем называть $\mathscr{S}$ *сильно связной областью* (или, короче, *областью*) в F, если

(1) в $\mathscr{S}$ существует такой единственный участок $\mathscr{B}$ (*вход*), что найдется путь из начальной вершины графа F в $\mathscr{B}$, не проходящий ни через какой другой участок из $\mathscr{S}$,

(2) существует путь (ненулевой длины), лежащий целиком внутри $\mathscr{S}$ и ведущий из любого участка в $\mathscr{S}$ в любой другой участок в $\mathscr{S}$.

Пример 11.34. Рассмотрим абстрактный граф управления (рис. 11.25). {2, 3, 4, 5} — сильно связная область с входом 2.

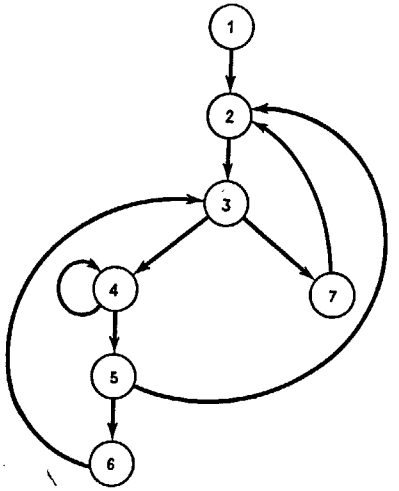


Рис. 11.25. Граф управления.

{4} — сильно связная область с входом 4. {3, 4, 5, 6} — область с входом 3. {2, 3, 7} — область с входом 2. Еще одна область с входом 2 — {2, 3, 4, 5, 6, 7}. Последняя *максимальна* в том смысле, что любая другая область с входом 2 содержится внутри этой области. □

Важной особенностью сильно связной области, благодаря которой она и помогает улучшать код, является однозначность

определения входного участка. Например, одна из оптимизаций, выполняемых на графах управления, заключается в перемещении инвариантного в области вычисления в участки, предшествующие входному для данной области (можно также построить новый участок, содержащий инвариантные вычисления и предшествующий входу).

Входные участки графов управления можно охарактеризовать в терминах отношения доминирования.

Теорема 11.11. *Пусть F — граф управления. Участок $\mathcal{B}$ в F является входным участком области тогда и только тогда, когда существует такой участок $\mathcal{B}'$, что из него есть дуга в $\mathcal{B}$ и $\mathcal{B}$ либо доминирует над $\mathcal{B}'$, либо совпадает с $\mathcal{B}'$.*

Доказательство. *Необходимость.* Предположим, что $\mathcal{B}$ — входной участок области $\mathcal{S}$. Если $\mathcal{S}=\{\mathcal{B}\}$, то результат тривиален. Если нет, пусть $\mathcal{B}'\in\mathcal{S}$, $\mathcal{B}'\neq\mathcal{B}$. Тогда $\mathcal{B}$ доминирует над $\mathcal{B}'$, поскольку в противном случае найдется путь из начальной вершины в $\mathcal{B}'$, не проходящий через $\mathcal{B}$, вопреки предположению о том, что $\mathcal{B}$ — единственный входной участок. Таким образом, входной участок области доминирует над любым другим участком. Поскольку в $\mathcal{B}$ ведет путь из любого элемента множества $\mathcal{S}$, в $\mathcal{S}-\{\mathcal{B}\}$ должен быть хотя бы один участок $\mathcal{B}'$, связанный непосредственно с $\mathcal{B}$.

Достаточность. Случай $\mathcal{B}=\mathcal{B}'$ тривиален, так что будем считать, что $\mathcal{B}\neq\mathcal{B}'$. Определим $\mathcal{S}$ как объединение участка $\mathcal{B}$ с такими участками $\mathcal{B}''$, что $\mathcal{B}$ доминирует над $\mathcal{B}''$ и существует путь из $\mathcal{B}''$ в $\mathcal{B}$, проходящий только через вершины, над которыми доминирует $\mathcal{B}$. По предположению $\mathcal{B}$ и $\mathcal{B}'$ оба принадлежат $\mathcal{S}$. Надо показать, что $\mathcal{S}$ — область с входом $\mathcal{B}$. Ясно, что условие (2) определения области удовлетворяется, поэтому мы должны показать, что существует путь из начальной вершины в $\mathcal{B}$, не проходящий ни через какой другой участок в $\mathcal{S}$. Пусть $\mathcal{C}_1\ldots\mathcal{C}_n\mathcal{B}$ — кратчайший путь вычисления, ведущий в $\mathcal{B}$. Если $\mathcal{C}_j\in\mathcal{S}$, то найдется такое число i, $1\leqslant i\leqslant j$, что $\mathcal{C}_i=\mathcal{B}$, поскольку $\mathcal{B}$ доминирует над $\mathcal{C}_j$. Пришли к противоречию, так как тогда $\mathcal{C}_1\ldots\mathcal{C}_n\mathcal{B}$ не будет кратчайшим путем вычислений, ведущим в $\mathcal{B}$. Итак, условие (1) определения сильно связной области удовлетворяется. □

Очевидно, что множество $\mathcal{S}$, построенное в ходе доказательства достаточности условия в теореме 11.11, является максимальной областью с входом $\mathcal{B}$. Было бы очень хорошо, если бы область с входом $\mathcal{B}$ была единственна, но, к сожалению, это не всегда так. В примере 11.34 три области с входом 2. Тем не менее теорема 11.11 полезна при построении эффективного алгоритма, вычисляющего максимальную область, которая единственна.

Если область не максимальна, входной участок может доминировать над участками, не принадлежащими области, а из них могут быть достижимы участки области. В примере 11.34, скажем, область $\{2, 3, 7\}$ может быть достижима через участок 6. Поэтому говорят, что область имеет *один вход*, если каждая дуга, входящая в ее участок, отличный от входного, выходит из участка внутри области. В примере 11.34 область $\{2, 3, 4, 5, 6, 7\}$ имеет один вход. В дальнейшем будем предполагать, что области имеют один вход, хотя не трудно дать обобщение на любые области.

1. Перемещение кода

Существует несколько преобразований, в которых для улучшения кода можно воспользоваться знанием областей. Одно из важнейших — *перемещение кода*. Вычисления, не зависящие от области, можно вынести за ее пределы. Пусть, скажем, внутри некоторой области с одним входом переменные Y и Z не меняются, но есть оператор $X\leftarrow Y+Z$. Вычисление $Y+Z$ можно переместить в заново образованный участок, связанный только с входным участком области[1]). Все связи вне области, ранее шедшие во входной участок, теперь идут в новый участок.

Пример 11.35. Может показаться, что вычисления, инвариантные относительно области, появляются только в неаккуратно написанной программе. Рассмотрим, однако, следующий внутренний цикл DO исходной программы на Фортране, где переменная определена вне цикла:

```
      K=0
      DO 3 I=1,1000
3     K=J+1+I+K
```

Промежуточная программа для этого фрагмента исходной программы может быть такой:

```
        K ← 0
        I ← 1
цикл    T ← J+1
        S ← T+I
        K ← S+K
        if I=1000 goto выход
        I=I+1
        goto цикл
выход   halt
```

Соответствующий граф управления изображен на рис. 11.26.

[1]) Добавление такого участка может привести к графу управления, не получающемуся ни из какой программы. Но здесь нигде не предполагается, что граф должен обязательно получаться из программы.

Из рис. 11.26 видно, что $\{\mathscr{B}_2, \mathscr{B}_3\}$ — область с входом $\mathscr{B}_2$. Оператор $T \leftarrow J+1$ инвариантен в области, так что его можно переместить в новый участок, как показано на рис. 11.27.

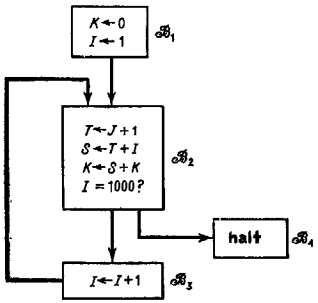


Рис. 11.26. Граф управления.

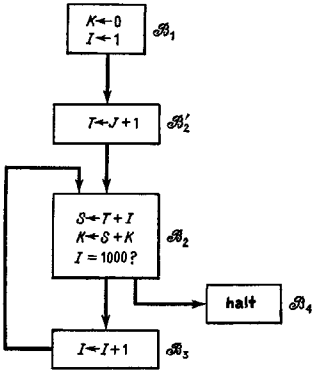


Рис. 11.27. Преобразованный граф управления.

Несмотря на то, что на рис. 11.26 операторов столько же, сколько на рис. 11.27, операторы в области будут, по-видимому, выполняться часто, так что ожидаемое время выполнения уменьшится. □

2. *Индуктивные переменные*

Другое полезное преобразование связано с удалением переменных, которые мы будем называть индуктивными.

Определение. Пусть $\mathscr{S}$ — область с одним входом $\mathscr{B}$, а X — переменная, появляющаяся в некотором операторе, входящем в один из участков в $\mathscr{S}$. Пусть $\mathscr{B}_1 \ldots \mathscr{B}_n \mathscr{B}\mathscr{C}_1 \ldots \mathscr{C}_m$ — такой путь вычислений, что $\mathscr{C}_i$ принадлежит $\mathscr{S}$, $1 \leqslant i \leqslant m$, а $\mathscr{B}_n$, если существует, не принадлежит $\mathscr{S}$. Обозначим через $X_1, X_2, \ldots$ значения, присваиваемые X в последовательности $\mathscr{B}\mathscr{C}_1 \ldots \mathscr{C}_m$. Если $X_1, X_2, \ldots$ образуют арифметическую прогрессию (с положительной или отрицательной разностью) для любого пути вычислений типа указанного выше, то будем называть X *индуктивной переменной* в $\mathscr{S}$.

Будем также называть X индуктивной переменной, если в первый раз она в $\mathscr{B}$ не определена и ее значения образуют арифметическую прогрессию. Тогда может понадобиться подходящим образом инициализировать ее при входе в участок извне, чтобы выполнить разбираемые здесь оптимизации.

Отметим, что задача нахождения всех индуктивных переменных в области нетривиальна. На самом деле можно доказать, что алгоритма для этого вообще нет. Тем не менее в общих ситуациях можно выявить достаточно много индуктивных переменных, так что это понятие заслуживает рассмотрения.

Пример 11.36. На рис. 11.27 область $\{\mathscr{B}_2, \mathscr{B}_3\}$ имеет вход $\mathscr{B}_2$. Если вход в $\mathscr{B}_2$ осуществляется из $\mathscr{B}_2'$ и управление повторно передается от $\mathscr{B}_2$ к $\mathscr{B}_3$ и снова к $\mathscr{B}_2$, то переменная I принимает значения 1, 2, 3, ... Таким образом, I — индуктивная переменная. Менее очевидно, но S — также индуктивная переменная, поскольку она принимает значения $T+1$, $T+2$, $T+3$, ... А вот переменная K не индуктивная, так как она принимает значения $T+1$, $2T+3$, $3T+6$, ... □

Важная особенность индуктивных переменных — их линейная связь друг с другом при передаче управления внутри области, которой они принадлежат. Например, на рис. 11.27 каждый раз при выходе из $\mathscr{B}_2$ справедливы соотношения $S=T+I$ и $I=S-T$.

Если, как на рис. 11.27, какая-то индуктивная переменная используется только для управления в области (на это указывает тот факт, что ее значение не требуется за пределами области и что непосредственно перед входом в область ей присваивается всегда одна и та же константа), то ее можно исключить. Даже если за пределами области требуются все индуктивные переменные, внутри области можно использовать только одну, а остальные вычислять при выходе из области.

Пример 11.37. Рассмотрим рис. 11.27. Исключим индуктивную переменную I, удовлетворяющую перечисленным выше критериям. Ее роль будет играть S. Заметим, что после участка $\mathscr{B}_2$ переменная S принимает значение $T+I$, так что, когда управление возвращается от $\mathscr{B}_3$ к $\mathscr{B}_2$, должно выполняться соотношение $S=T+I-1$. Таким образом, оператор $S \leftarrow T+I$ можно заменить на $S \leftarrow S+1$. Но затем в $\mathscr{B}_2'$ надо правильно инициировать S, так что, когда управление перейдет из $\mathscr{B}_2'$ в $\mathscr{B}_2$, значением S после оператора $S \leftarrow S+1$ будет $T+I$. Ясно, что в $\mathscr{B}_2'$ после оператора $T \leftarrow J+1$ надо ввести новый оператор $S \leftarrow T$.

Затем мы должны исправить проверку $I=1000$? так, чтобы получить эквивалентную проверку относительно S. При выполнении этой проверки S имеет значение $T+I$. Следовательно, эквивалентной проверкой будет

$$R \leftarrow T+1000$$
$$S = R?$$

Поскольку R не зависит от области, вычисление $R \leftarrow T+1000$ можно вынести в участок $\mathscr{B}_2'$. Тогда можно полностью избавиться от I. Новый граф управления изображен на рис. 11.28.

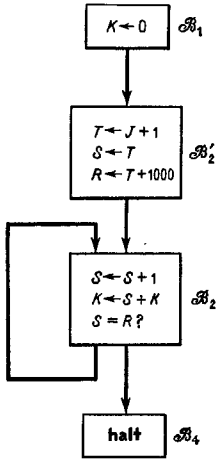


Рис. 11.28. Дальнейшее преобразование графа управления.

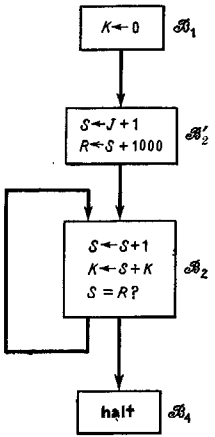


Рис. 11.29. Окончательный граф управления.

Из рис. 11.28 видно, что участок $\mathscr{B}_3$ исключен полностью, а область укорочена на один оператор. Конечно, размер участка $\mathscr{B}_2'$ увеличился, но, по-видимому, области исполняются значительно чаще, чем участки вне области. Таким образом, рис. 11.28 представляет ускоренный вариант рис. 11.27.

Шаг $S \leftarrow T$ в $\mathscr{B}_2'$ можно исключить, если отождествить S и T. Это возможно только потому, что значения переменных S и T никогда не будут различными, но несмотря на это обе они будут „активными" в том смысле, что обе будут использоваться в дальнейших вычислениях. Иными словами, в $\mathscr{B}_2$ активна только переменная S, ни одна из них не активна в $\mathscr{B}_1$, а в $\mathscr{B}_2'$ обе активны между операторами $S \leftarrow T$ и $R \leftarrow T+1000$. В этот момент они, разумеется, принимают одно и то же значение. Если T заменить на S, получим граф управления на рис. 11.29.

Для того чтобы понять, чем граф на рис. 11.29 лучше графа на рис. 11.26, превратим каждый из них в программу на языке ассемблера для абстрактной машины с одним сумматором. Коды операций должны быть понятны (JZERO означает „переход

в случае нулевого сумматора", а JNZ — „переход в случае ненулевого сумматора"). Две эти программы приведены на рис. 11.30.

```
        LOAD   =0
        STORE  K
        LOAD   =1
цикл:   STORE  I
        LOAD   J
        ADD    =1
        ADD    I
        ADD    K
        STORE  K
        LOAD   I
        SUBTR  =1000
        JZERO  выход
        LOAD   I
        ADD    =1
        JUMP   цикл
выход:  END
```

а

```
        LOAD   =0
        STORE  K
        LOAD   J
        ADD    =1
        STORE  S
        ADD    =1000
        STORE  R
цикл:   LOAD   S
        ADD    =1
        STORE  S
        ADD    K
        STORE  K
        LOAD   S
        SUBTR  R
        JNZ    цикл
        END
```

б

Рис. 11.30. Эквивалентные программы: *а* — программа для графа управления на рис. 11.26; *б* — программа для графа управления на рис. 11.29.

Заметим, что длина программы на рис. 11.30,*а* такая же, как и на рис. 11.30,*б*. Однако цикл на рис. 11.30,*б* короче, чем на рис. 11.30,*а* (8 команд вместо 12), а это важно с точки зрения времени. □

3. *Замена сложных операций*

Внутри областей возможна интересная форма замены сложных операций. Если внутри области есть оператор вида $A \leftarrow B*I$, в котором значение B не зависит от области, а I — индуктивная переменная, то можно заменить умножение сложением или вычитанием величины, равной произведению значения, не зависящего от области, и разности арифметической прогрессии, порождаемой индуктивной переменной. При этом надо соответствующим образом инициировать величину, вычисляемую в предыдущем операторе умножения.

Пример 11.38. Рассмотрим фрагмент исходной программы

```
      DO 5 J=1, N
      DO 5 I=1, M
5     A(I, J)=B(I, J)
```

делающий массив A равным массиву B в предположении, что A и B имеют одинаковый размер $M \times N$. Пусть $A(I, J)$ запоминается в ячейке $A+M*(J-1)+I$ для $1 \leqslant I \leqslant M$, $1 \leqslant J \leqslant N$;

аналогичное предположение относится и к $B(I, J)$. Для удобства обозначим ячейку $A+L$ через $A(L)$. Тогда из этой исходной

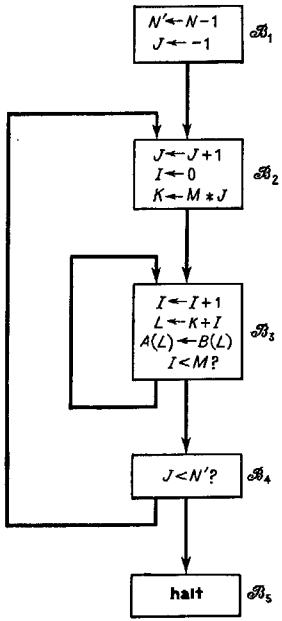


Рис. 11.31. Граф управления.

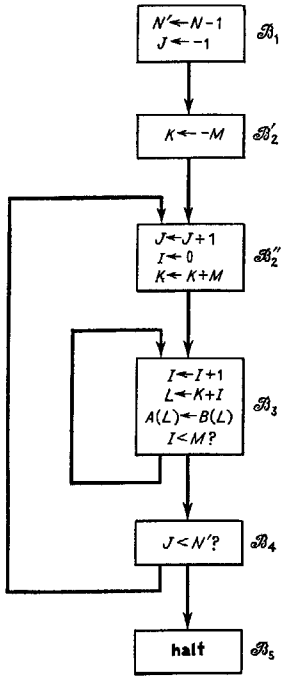


Рис. 11.32. Новый граф управления.

программы можно получить частично оптимизированную программу

$$
\begin{array}{ll}
 & N' \leftarrow N-1 \\
 & J \leftarrow -1 \\
\textit{внеш:} & J \leftarrow J+1 \\
 & I \leftarrow 0 \\
 & K \leftarrow M * J \\
\textit{цикл:} & I \leftarrow I+1 \\
 & L \leftarrow K+I \\
 & A(L) \leftarrow B(L) \\
 & \textbf{if } I < M \textbf{ goto } \textit{цикл} \\
 & \textbf{if } J < N' \textbf{ goto } \textit{внеш} \\
 & \textbf{halt}
\end{array}
$$

Граф управления новой программы изображен на рис. 11.31. В этом графе $\{\mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4\}$ — область, в которой переменная M инвариантна, а J — индуктивная переменная, возрастающая на 1. Поэтому оператор $K \leftarrow M * J$ можно заменить на $K \leftarrow K+M$, предварительно присвоив K значение $-M$ вне области. Новый граф управления показан на рис. 11.32. Программа, представляемая этим новым графом, длиннее прежней, но области, соответствующие участкам $\mathcal{B}_2''$, $\mathcal{B}_3$ и $\mathcal{B}_4$, могут исполняться быстрее, поскольку умножение заменено сложением.

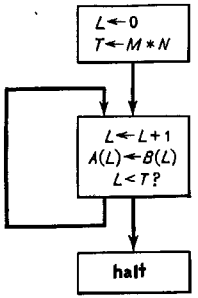


Рис. 11.33. Окончательный граф управления.

Интересно отметить, что можно получить еще более экономную программу, заменив всю область $\{\mathcal{B}_2'', \mathcal{B}_3, \mathcal{B}_4\}$ одним участком, в котором $A(L)$ принимает значение $B(L)$ для $1 \leqslant K \leqslant M * N$. Окончательный граф управления изображен на рис. 11.33. □

4. *Развертывание циклов*

Последнее преобразование по улучшению кода, которое мы изучим, чрезвычайно просто, но часто остается незамеченным. Это *развертывание циклов*. Рассмотрим граф управления на рис. 11.34. Участки $\mathcal{B}_2$ и $\mathcal{B}_3$ исполняются по 100 раз. Таким образом, выполняется 100 команд проверки. От всех этих 100 команд можно избавиться, „развернув" цикл. Иными словами, цикл можно превратить в линейный участок, состоящий из 100 операторов присваивания

$$
\begin{array}{c}
A(1) \leftarrow B(1) \\
A(2) \leftarrow B(2) \\
\vdots \\
A(100) \leftarrow B(100)
\end{array}
$$

Не допуская таких вольностей, можно было бы развернуть цикл только „на один шаг" и получить граф управления, изображенный на рис. 11.35. Программа, представленная рис. 11.35, длиннее, но в ней исполняется меньше команд. На рис. 11.35 достаточно 50 команд проверки (вместо 100 команд на рис. 11.34).

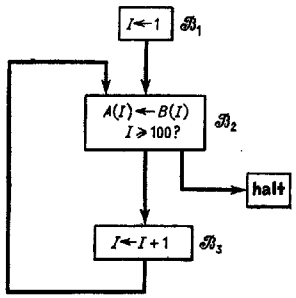


Рис. 11.34. Граф управления.

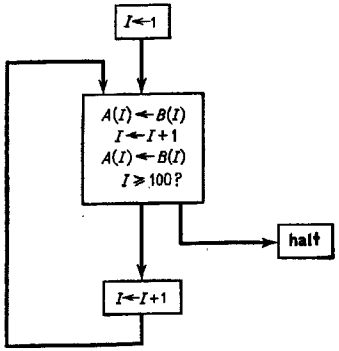


Рис. 11.35. Граф управления после развертывания цикла.

УПРАЖНЕНИЯ

11.3.1. Постройте промежуточные программы, эквивалентные следующим исходным программам:

(а) $S=(A+B+C)*.5$
$D=S*(S-A)*(S-B)*(S-C)$
AREA = SQRT (D)

(б)
```
for I := 1 step 1 until N do
    begin
        A[I] := B[I]+C[I*2];
        if (A[I]=0) then halt
          else A[I] := I
    end
```

(в)
```
    DO 5 I=1, N
5   A(I, I)=C*A(I, I)
```


11.3.2. Какие функции вычисляются следующими двумя программами на промежуточном языке?

(а)
```
        read N
        S ← 0
        I ← 1
цикл:   S ← S+I
        if I ≥ N goto выход
        I = I+1
        goto цикл
выход:  write S
        halt
```

(б)
```
        read N
        T ← N+1
        T ← T*N
        T ← T*.5
        write T
        halt
```

Эквивалентны ли эти две программы, если N и I представляют целые, а S и I — вещественные числа?

11.3.3. Рассмотрим программу P:

```
        read A, B
        R ← 1
        C ← A*A
        D ← B*B
        if C < D goto X
        E ← A*A
        R ← R+1
        E ← E+R
        write E
        halt
X:      E ← B*B
        R ← R+2
        E ← E+R
        write E
        if E > 100 goto Y
        halt
Y:      R ← R−1
        goto X
```

Постройте для нее граф управления.

11.3.4. Найдите доминаторы и прямые доминаторы каждой вершины $\mathcal{B}$ в графе управления на рис. 11.36.

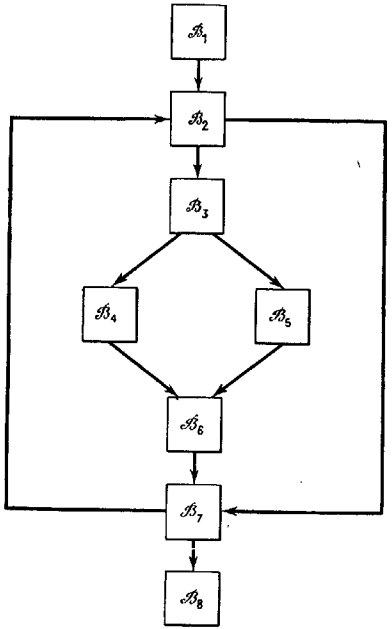


Рис. 11.36. Граф управления.

11.3.5. Пусть ON $(\mathcal{B}_1, \mathcal{B}_2)$ — множество участков, которые могут появиться в графе управления на пути из $\mathcal{B}_1$ в $\mathcal{B}_2$ (через $\mathcal{B}_1$ нельзя проходить повторно, а через $\mathcal{B}_2$ можно). Покажите, что если $\mathcal{B}_1$ доминирует над $\mathcal{B}_2$, то ON $(\mathcal{B}_1, \mathcal{B}_2) = \{\mathcal{B} \mid$ существует путь из $\mathcal{B}$ в $\mathcal{B}_2$, когда участок $\mathcal{B}_1$ удален из графа управления$\}$. Сколько времени займет вычисление ON $(\mathcal{B}_1, \mathcal{B}_2)$?

11.3.6. Докажите утверждения (1) и (2) из леммы 11.13.

11.3.7. Можно следующим образом определить отношение постдоминирования. Пусть F — граф управления и $\mathcal{B}$ — вершина в F. Вершину $\mathcal{B}'$ назовем *постдоминатором* для $\mathcal{B}$, если любой путь из $\mathcal{B}$ в оператор **halt** проходит через $\mathcal{B}'$. *Прямой постдоминатор* для $\mathcal{B}$ постдоминируется любым другим постдоминатором для $\mathcal{B}$. Покажите, что если вершина графа управления имеет постдоминатор, то она имеет и прямой постдоминатор.

11.3.8. Разработайте алгоритм построения прямых постдоминаторов всех вершин графа управления.

11.3.9. Найдите все сильно связные области на рис. 11.36. Какие области максимальны?

11.3.10. Пусть P — программа из упр. 11.3.3.
(а) Исключите из P все общие подвыражения.
(б) Исключите из P все ненужные вычисления констант.
(в) Удалите из цикла в P все инвариантные вычисления.

11.3.11. Найдите индуктивные переменные в следующей программе:

```
      I ← 1
      read J, K
X:    A ← K * I
      B ← J * I
      C ← A + B
      write C
      I ← I + 1
      if I < 100 goto X
      halt
```

Исключите их столько, сколько сможете, и замените насколько возможно умножения сложениями.

11.3.12. Приведите алгоритм для нахождения в графе управления всех (а) областей, (б) областей с одним входом и (в) максимальных областей.

***11.3.13.** Приведите алгоритм, выявляющий некоторые индуктивные переменные в области с одним входом.

***11.3.14.** Обобщите алгоритм из упр. 11.3.13, чтобы можно было обрабатывать не только области с одним входом.

11.3.15. Приведите алгоритм для перемещения вычислений, не зависящих от области, за пределы области (не обязательно с одним входом). *Указание*: В участках вне области, из которых можно достичь области не через вход в область, допускается менять переменные, участвующие в вычислениях, инвариантных относительно области. Возможно, между участками вне области и участками внутри области потребуется поместить новые участки.

****11.3.16.** Покажите, что проблема эквивалентности двух программ неразрешима. *Указание*: Выберите подходящие типы данных и интерпретации для операций.

**11.3.17. Покажите, что проблема, индуктивна ли данная переменная, неразрешима.

11.3.18. Обобщите понятие области действия переменных и операторов на программы с циклами. Приведите алгоритм, вычисляющий область действия переменной в программе с циклами.

*11.3.19. Расширьте преобразования T_1—T_4 из разд. 11.1 с тем, чтобы их можно было применить к программам без циклов назад (программам с операторами присваивания и условными операторами вида **if** yRy **goto** L, где L ссылается на оператор, расположенный после условного оператора).

*11.3.20. Покажите, что проблема, окончится ли когда-нибудь программа, неразрешима.

Проблемы для исследования

11.3.21. Охарактеризуйте модели машин, для которых описанные преобразования будут приводить к более быстро выполнимым программам.

11.3.22. Разработайте алгоритмы, которые будут определять широкий класс явлений, исследованных в настоящем разделе (например, инвариантные в цикле вычисления или индуктивные переменные). Отметим, что для большинства этих явлений нет алгоритма, определяющего все их вхождения.

Открытая проблема

11.3.23. Можно ли вычислить прямые доминаторы графа управления с n вершинами менее чем за $O(n^2)$ шагов? Разумно предположить, что $O(n^2)$ — лучшее, чего можно достичь для получения всего отношения доминирования, поскольку это как раз столько, сколько требуется для печати результата в матричной форме.

Замечания по литературе

В ряде работ предполагаются различные оптимизирующие преобразования программ. Нивергельт [1965], Марилл [1962], Мак-Киман [1965] и Кларк [1967] перечисляют некоторые машинно-независимые преобразования. Гир [1965] предлагает оптимизатор, способный к исключению некоторых общих подвыражений, размножению констант и к некоторым оптимизациям циклов, таким, как замена сложных операций и удаление инвариантных вычислений. Бузам и Энглунд [1969] описывают аналогичные преобразования в рамках Фортрана. Аллен и Кок [1972] дают хороший обзор этих методов. Аллен [1969] приводит схему глобальной оптимизации, основанной на нахождении сильно связных областей программы.

Подход к оптимизации с точки зрения доминаторов впервые предложили Лоури и Медлок [1969], хотя идея отношения доминирования пришла от Проссера [1959].

Много теоретических работ посвящено схемам программ, аналогичным графам управления, но с неспецифицированными множествами значений переменных и неспецифицированными функциями для знаков операций. Две фундаментальные работы, в которых рассматривается эквивалентность между такими схемами независимо от реальных множеств и функций, принадлежат Янову [1958] и Лаххэму и др. [1970]. Обзор этих исследований содержится в работах Каплана [1970] и Манны [1973] [1]).

11.4. АНАЛИЗ ПОТОКА ДАННЫХ

В предыдущем разделе мы использовали информацию о вычислениях в участках программ, не описывая, как ее можно эффективно получить. В частности, мы использовали:

(1) „Доступные" при входе в участок выражения. Выражение $A+B$ называют *доступным* при входе в участок, если $A+B$ всегда вычисляется до достижения участка, но не ранее, чем определяются A и B.

(2) Множество участков, в которых переменная могла определяться в последний раз перед тем, как поток управления достиг текущего участка. Эта информация полезна для размножения констант и выявления бесполезных вычислений. Она используется также для выявления возможных ошибок программиста, заключающихся в том, что на переменную делается ссылка до того, как она определена.

Информация третьего типа, для вычисления которой можно применить методы настоящего раздела, связана с выявлением активных переменных, т. е. переменных, значения которых должны сохраняться при выходе из участка. Эта информация полезна, когда участки преобразуются в машинный код, поскольку она указывает переменные, которые при выходе из участка должны либо запоминаться, либо сохраняться в быстром регистре. В терминах разд. 11.1 эта информация нужна для выявления выходных переменных. Отметим, что переменная может вычисляться не в рассматриваемом участке, а в каком-нибудь предыдущем, но быть тем не менее входной и выходной переменной участка.

Из трех этих проблем мы исследуем только вторую — установление участка, где могла определяться переменная перед тем, как был достигнут данный участок. Предлагаемый метод, назван-

[1]) В работах советских авторов задача оптимизации циклов рассматривалась на ранних этапах автоматизации программирования. Задача развертывания циклов была впервые поставлена в работе Ершова и Курочкина [1961]. Развертывание циклов, основанное на анализе и преобразовании графа управления, реализовано в АЛЬФА-трансляторе [Бежанова, Поттосин, 1965]. Анализ зависимости индексного выражения от параметра цикла, основанный по существу на понятии индуктивной переменной, проводился в работах Великановой и др. [1961], Китова и Криницкого [1959], Камынина, Любимского и Шура-Буры [1958].— *Прим. перев.*

ный „анализом интервалов“, заключается в разбиении графа управления на все бо́льшие и бо́льшие множества вершин; тем самым с графом связывается иерархическая структура. С помощью этой структуры можно будет дать эффективный алгоритм для класса графов управления, называемых „сводимыми“; такие графы очень часто встречаются в качестве графов управления, возникающих из реальных программ. Затем мы укажем расширения, необходимые для обработки несводимых графов. В упражнениях рассмотрим изменения, которые надо сделать, чтобы при анализе интервалов учитывалась информация двух других типов.

11.4.1. Интервалы

Начнем с определения типа подграфа, полезного при анализе потока данных.

Определение. Если h — вершина графа управления F, определим *интервал* $I(h)$ *с заголовком* h как такое множество вершин графа F, что

(1) h принадлежит $I(h)$,

(2) если вершина n, еще не включенная в $I(h)$, не является начальной и все дуги, входящие в n, выходят из вершин, принадлежащих $I(h)$, добавляем n к $I(h)$,

(3) повторяем шаг (2) до тех пор, пока не останется вершин, которые можно добавить к $I(h)$.

Пример 11.39. Рассмотрим граф управления, изображенный на рис. 11.37.

Рассмотрим интервал с начальной вершиной n_1 в качестве заголовка. Согласно шагу (1), $I(n_1)$ включает n_1. Поскольку единственная дуга, входящая в n_2, выходит из n_1, добавляем n_2 к $I(n_1)$. Вершину n_3 нельзя добавить к $I(n_1)$, так как в нее можно попасть не только из n_2, но и из n_5. Никаких других вершин добавить к $I(n_1)$ нельзя. Таким образом, $I(n_1)=\{n_1, n_2\}$.

Рассмотрим теперь $I(n_3)$. Согласно шагу (1), n_3 принадлежит $I(n_3)$. Однако n_4 нельзя добавить к $I(n_3)$, так как в n_4 можно попасть через n_6 (как и через n_3), а n_6 не принадлежит $I(n_3)$. Никаких других вершин добавить к $I(n_3)$ нельзя, так что $I(n_3)=\{n_3\}$.

Продолжая в том же духе, разбиваем наш граф управления на интервалы

$$I(n_1)=\{n_1, n_2\}$$
$$I(n_3)=\{n_3\}$$
$$I(n_4)=\{n_4, n_5, n_6\}$$
$$I(n_7)=\{n_7, n_8, n_9\} \quad \square$$

Приведем алгоритм выделения заголовков интервалов и построения соответствующих интервалов; алгоритм разбивает граф управления на непересекающиеся интервалы. Сначала сделаем три замечания относительно интервалов.

Теорема 11.12. (1) *Заголовок h доминирует над всеми остальными вершинами в $I(h)$ (хотя не обязательно все вершины, над которыми доминирует h, принадлежат $I(h)$).*

(2) *Для каждой вершины h графа управления F интервал $I(h)$ определяется однозначно и не зависит от порядка, в котором на шаге (2) определения интервала выбираются кандидаты для n.*

(3) *Каждый цикл в интервале $I(h)$ включает заголовок интервала h.*

Доказательство. Утверждения (1) и (2) оставляем в качестве упражнений; докажем (3). Предположим, что $I(h)$ имеет цикл $n_1, \ldots, n_k$, не включающий h. Это означает, что существуют дуга из n_i в n_{i+1}, $1 \leqslant i < k$, и дуга из n_k в n_1. Пусть вершина n_i первая среди $n_1, \ldots, n_k$ добавляется к $I(h)$. Тогда вершина n_{i-1} (или n_k, если $i=1$) должна к этому времени уже быть в $I(h)$ вопреки предположению. $\square$

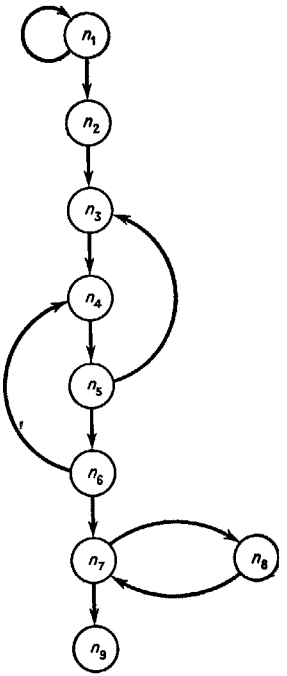


Рис. 11.37. Граф управления.

Один из интересных аспектов анализа интервалов заключается в том, что графы управления можно единственным образом разбить на интервалы, а интервалы одного графа управления можно рассматривать как вершины другого графа управления, в котором из интервала I_1 ведет дуга в другой интервал I_2, если какая-нибудь дуга ведет из вершины интервала I_1 в заголовок интервала I_2. (Ясно, что никакая дуга не может вести из I_1 в вершину интервала I_2, отличную от заголовка.) Новый граф можно точно таким же образом снова разбить на интервалы, и этот процесс можно продолжить. Поэтому в дальнейшем мы будем считать, что граф управления состоит не из участков, а из вер-

шии, тип которых не специфицирован. Итак, вершины могут представлять структуры произвольной сложности.

Дадим теперь алгоритм разбиения графа управления на непересекающиеся интервалы.

Алгоритм 11.6. Разбиение графа управления на непересекающиеся интервалы.

Вход. Граф управления F.

Выход. Множество непересекающихся интервалов, объединение которых содержит все вершины графа F.

Метод.

(1) С каждой вершиной в F свяжем два параметра: *счетчик* и *достижимость*. Счетчик для n вначале равен числу дуг, входящих в n. В ходе выполнения алгоритма счетчик для n равен числу еще не пройденных дуг, входящих в n. Достижимость для n либо не определена, либо является некоторой вершиной из F. Вначале достижимость не определена для всех вершин, кроме начальной, достижимость которой есть она сама. В конечном итоге достижимостью для n станет первый найденный заголовок интервала h, такой, что из некоторой вершины интервала $I(h)$ ведет дуга в n.

(2) Образуем список вершин, называемый *списком заголовков*. Вначале список заголовков содержит только начальную вершину графа F.

(3) Если список заголовков пуст, остановиться. В противном случае пусть n—следующая вершина списка заголовков. Удаляем n из списка заголовков.

(4) Затем применяем шаги (5)—(7) для построения интервала $I(n)$. На этих шагах к списку заголовков добавляются прямые потомки вершин из $I(n)$.

(5) $I(n)$ строится как список вершин. Вначале $I(n)$ содержит только вершину n и она „не помечена".

(6) Выбираем в $I(n)$ непомеченную вершину n', помечаем ее и для каждой вершины n'', в которую ведет дуга из n', выполняем такие операции:

(а) Уменьшаем на 1 счетчик для n''.

(б) (i) Если достижимость для n'' не определена, полагаем ее равной n и делаем следующее. Если счетчик для n'' равен теперь 0 (перед этим был равен 1), то добавляем вершину n'' к $I(n)$ и переходим к шагу (7); иначе добавляем вершину n' к списку заголовков, если ее там не было, и переходим к шагу (7).

(ii) Если достижимость для n'' равна n, а счетчик вершины n'' равен 0, добавляем вершину n'' к $I(n)$ и удаляем ее из списка заголовков, если она там есть. Переходим к шагу (7).

Если ни (i), ни (ii) не применимы, в (б) не делается ничего.

(7) Если в $I(n)$ остается непомеченная вершина, возвращаемся к шагу (6). Иначе список $I(n)$ заполнен; возвращаемся к шагу (3). □

Определение. Из интервалов графа управления F можно построить другой граф управления $I(F)$, который будем называть *производным графом* от F. Производный граф определяется так:

(1) $I(F)$ имеет по одной вершине для каждого интервала, построенного алгоритмом 11.6.

(2) Начальной вершиной для $I(F)$ служит интервал, содержащий начальную вершину для F.

(3) Из интервала I в интервал J ведет дуга тогда и только тогда, когда $I \neq J$ и из вершины в I ведет дуга в заголовок интервала J.

Производный граф $I(F)$ графа управления F показывает поток управления между интервалами в F. Поскольку граф $I(F)$ сам является графом управления, можно также построить граф $I(I(F))$, производный от $I(F)$. Таким образом, если дан граф управления F_0, можно построить последовательность графов управления $F_0, F_1, \ldots, F_n$, называемую *производной последовательностью* от F, в которой F_{i+1}—производный граф от F_i, $0 \leqslant i < n$, а F_n—граф, производный от самого себя (т. е. $I(F_n) = F_n$). Граф F_i называют *i-м производным графом* от F_0. Граф F_n называется *пределом* графа F_0. Нетрудно показать, что F_n всегда существует и единствен.

Если F_n состоит из одной вершины, то граф F называют *сводимым*.

Интересно отметить, что если граф F_0 строится по реальной программе, то с большой вероятностью он будет сводимым. В разд. 11.4.3 мы изложим метод расщепления вершин, с помощью которого любой несводимый граф управления можно превратить в сводимый.

Пример 11.40. Воспользуемся алгоритмом 11.6 для построения интервалов графа управления, изображенного на рис. 11.38.

Начальной вершиной служит n_1. Вначале список заголовков содержит только n_1. Для построения $I(n_1)$ включаем n_1 в $I(n_1)$ как непомеченную вершину. Помечаем вершину n_1 ее прямым потомком n_2. Для этого уменьшаем счетчик n_2 с 2 до 1, полагаем достижимость для нее равной n_1 и добавляем ее к списку заголовков. К этому моменту в $I(n_1)$ не остается непомеченных вершин, так что список $I(n_1) = \{n_1\}$ заполнен.

Список заголовков содержит потомка n_2 вершины из $I(n_1)$. Для вычисления $I(n_2)$ включаем n_2 в $I(n_2)$ и рассматриваем вершину n_3, счетчик для которой равен 2. Уменьшаем счетчик на 1, полагаем достижимость для нее равной n_2 и добавляем ее к списку заголовков. Находим, таким образом, что $I(n_2)=\{n_2\}$.

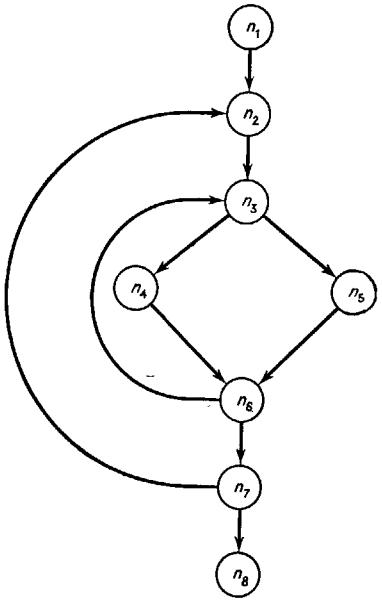


Рис. 11.38. Граф управления.

Список заголовков содержит теперь потомка n_3 вершины из $I(n_2)$. Вычисление $I(n_3)$ начинаем с занесения n_3 в $I(n_3)$. Рассматриваем затем вершины n_4 и n_5, уменьшая счетчики для них с 1 до 0, полагая достижимости для них равными n_3 и добавляя их к $I(n_3)$ как непомеченные вершины. Помечаем n_4, уменьшая счетчик для n_6 с 2 до 1, полагая достижимость для n_6, равной n_3 и добавляя n_6 к списку заголовков. Помечая n_5 в $I(n_3)$, уменьшаем счетчик для n_6 с 1 до 0, удаляем ее из списка заголовков и добавляем к $I(n_3)$.

Чтобы пометить n_6 в $I(n_3)$, делаем счетчик для n_7 равным 0, полагаем достижимость для нее равной n_3 и добавляем ее к $I(n_3)$.

Следующей рассматривается вершина n_3, поскольку есть дуга из n_6 в n_3. Так как достижимость для n_3 равна n_2, вершина n_3 в данный момент не изменяет $I(n_3)$ и списка заголовков. Чтобы пометить n_7, делаем счетчик для n_8 равным 0, полагаем достижимость для нее равной n_3 и добавляем ее к $I(n_3)$. Вершина n_3 также является потомком вершины n_7, но так как достижимость для n_2 равна n_1, то n_2 не добавляется ни к $I(n_3)$, ни к списку заголовков. Наконец, чтобы пометить n_8, не надо производить никаких операций, поскольку n_8 не имеет потомков. К этому моменту в $I(n_3)$ не остается непомеченных вершин, так что $I(n_3)=\{n_3, n_4, n_5, n_6, n_7, n_8\}$.

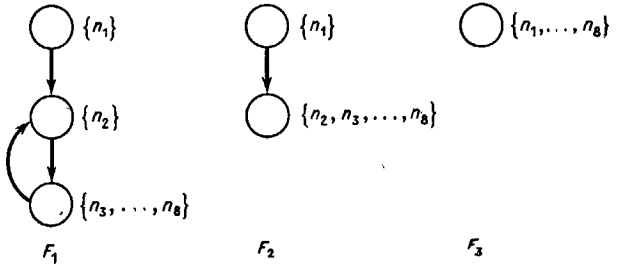


Рис. 11.39. Последовательность графов управления.

Список заголовков теперь пуст, так что алгоритм заканчивается. В результате граф управления оказался разбитым на следующие три непересекающихся интервала:

$$I(n_1)=\{n_1\}$$
$$I(n_2)=\{n_2\}$$
$$I(n_3)=\{n_3, n_4, n_5, n_6, n_7, n_8\}$$

Из этих интервалов можно построить первый производный граф F_1. Затем к F_1 можно применить алгоритм 11.6 и получить его интервалы. Повторяя весь этот процесс, строим последовательность производных графов, изображенную на рис. 11.39. □

Пример 11.41. Рассмотрим граф управления F на рис. 11.40. Интервалы для него таковы:

$$I(n_1)=\{n_1\}$$
$$I(n_2)=\{n_2\}$$
$$I(n_3)=\{n_3\}$$

Находим, что $I(F)=F$, так что граф F несводим. □

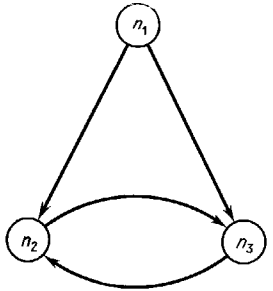


Рис. 11.40. Граф управления F.

Теорема 11.13. *Алгоритм* 11.6 *строит множество непересекающихся интервалов, объединение которых совпадает со всем графом.*

Доказательство. Ясно, что интервалы не пересекаются Если вершина добавлена к интервалу на шаге (6бi) алгоритма 11.6, то она не будет включена в список заголовков. Если вершина добавлена к интервалу на шаге (6бii), то она удаляется из списка заголовков. Аналогично, легко показать, что объединение всех построенных списков $I(n)$ — это множество вершин графа F. Предполагая, что F — граф управления, можно считать, что каждая вершина достижима из начальной вершины в F и потому попадает либо в список заголовков, либо в некоторый интервал. Если вершина не добавляется к интервалу, она становится заголовком своего собственного интервала.

Наконец, надо показать, что каждый построенный список $I(n)$ есть интервал. На шаге (6) вершина n'' добавляется к $I(n)$ тогда и только тогда, когда достижимость для нее равна n, а счетчик уменьшен до 0. Таким образом, каждая дуга, входящая в n'', идет из вершины, уже содержащейся в $I(n)$, и в соответствии с определением интервала вершину n'' можно добавить к $I(n)$. □

Заметим, что на машине с произвольным доступом алгоритм 11.6 может выполняться за время, пропорциональное числу дуг графа управления. Поскольку в графе управления, вершины которого являются участками программы, ни из какой вершины не выходит более двух дуг, это равносильно тому, что алгоритм 11.6 линейно зависит от числа участков программы. В качестве упражнения предлагаем показать, что каждый производный граф, построенный повторным применением алгоритма 11.6 из программы, состоящей из n участков, имеет не более $2n$ дуг.

11.4.2. Анализ потока данных с помощью интервалов

Покажем, как анализ интервалов может использоваться для определения потока данных внутри сводимого графа. Проблема, которую мы будем исследовать, заключается в том, что для каждого участка $\mathcal{B}$ и для каждой переменной A сводимого графа управления выясняется, в каких операторах программы могла определяться переменная A в последний раз перед тем, как управление достигло участка $\mathcal{B}$. Затем основной алгоритм анализа интервалов мы распространим на несводимые графы управления.

Важно отметить, что отчасти преимущество подхода к анализу потока данных с помощью интервалов связано с трактовкой множеств как упакованных векторов битов. Для вычисления пересечения, объединения и дополнения множеств используются логические операции AND, OR и NOT на векторах битов, обычно весьма эффективно выполняющиеся на большинстве машин.

Построим таблицы, дающие для каждого участка $\mathcal{B}$ программы все позиции l, где определяется данная переменная A и откуда существует путь в $\mathcal{B}$, вдоль которого A не переопределяется. Эта информация может пригодиться для выявления возможных значений A при входе в участок $\mathcal{B}$.

Начнем с определения четырех функций, отображающих участки в множества.

Определение. *Путем вычислений из оператора* s_1 *в оператор* s_2 назовем последовательность операторов, начинающуюся с s_1 и заканчивающуюся в s_2, которая в данном порядке может выполняться в процессе исполнения программы.

Пусть $\mathcal{B}$ — участок программы P. Определим четыре множества, связанные с операторами определения:

(1) $\mathrm{IN}(\mathcal{B}) = \{d \in P \mid$ существует такой путь вычислений из оператора определения d в первый оператор в $\mathcal{B}$, что никакой из операторов на этом пути, кроме, быть может, первого оператора в $\mathcal{B}$, не переопределяет переменную, определяемую оператором $d\}$.

(2) $\mathrm{OUT}(\mathcal{B}) = \{d \in P \mid$ существует такой путь вычислений из d в последний оператор в $\mathcal{B}$, что никакой из операторов на нем не переопределяет переменную, определяемую $d\}$.

(3) $\mathrm{TRANS}(\mathcal{B}) = \{d \in P \mid$ переменная, определяемая d, не определяется никаким оператором в $\mathcal{B}\}$.

(4) $\mathrm{GEN}(\mathcal{B}) = \{d \in \mathcal{B} \mid$ переменная, определяемая d, не определяется впоследствии нигде в $\mathcal{B}\}$.

Говоря неформально, $\mathrm{IN}(\mathcal{B})$ содержит определения, которые могут быть активными при входе в $\mathcal{B}$, $\mathrm{OUT}(\mathcal{B})$ содержит опре-

деления, которые могут быть активными при выходе из $\mathcal{B}$, TRANS($\mathcal{B}$) содержит определения, передаваемые через $\mathcal{B}$ без определения в $\mathcal{B}$, GEN($\mathcal{B}$) содержит определения, создаваемые в $\mathcal{B}$, которые остаются активными при выходе из $\mathcal{B}$. Легко показать, что

$$\mathrm{OUT}(\mathcal{B}) = (\mathrm{IN}(\mathcal{B}) \cap \mathrm{TRANS}(\mathcal{B})) \cup \mathrm{GEN}(\mathcal{B})$$

Пример 11.42. Рассмотрим программу

```
S1:  I ← 1
S2:  J ← 0
S3:  J ← J + 1
S4:  read I
S5:  if I < 100 goto 38
S6:  write J
S7:  halt
S8:  I ← I * I
S9:  goto S3
```

Для удобства все операторы помечены. Граф управления этой программы изображен на рис. 11.41. Каждый участок помечен явно. Определим для $\mathcal{B}_2$ множества IN, OUT, TRANS и GEN.

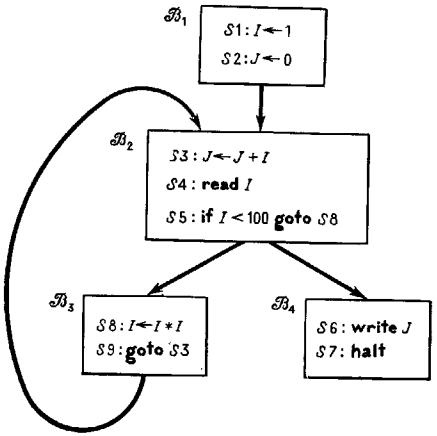


Рис. 11.41. Граф управления.

Оператор $S1$ определяет I, а $S1$, $S2$, $S3$ — путь вычислений, на котором I не определяется (кроме как в $S1$). Поскольку этот путь ведет из $S1$ в первый оператор участка $\mathcal{B}_2$, ясно,

что $S1 \in \mathrm{IN}(\mathcal{B}_2)$. Аналогично можно показать, что

$$\mathrm{IN}(\mathcal{B}_2) = \{S1,\ S2,\ S3,\ S8\}$$

Отметим, что $S4$ не принадлежит IN($\mathcal{B}_2$), поскольку нет пути вычислений из $S4$ в $S3$, не переопределяющего I после $S4$.

OUT($\mathcal{B}_2$) не содержит $S1$, поскольку все пути вычислений из $S1$ в $S5$ переопределяют I. Предоставляем читателю проверить, что

$$\mathrm{OUT}(\mathcal{B}_2) = \{S3,\ S4\}$$
$$\mathrm{TRANS}(\mathcal{B}_2) = \varnothing$$
$$\mathrm{GEN}(\mathcal{B}_2) = \{S3,\ S4\} \qquad \square$$

Остаток этого раздела посвящен разработке алгоритма вычисления IN($\mathcal{B}$) для всех участков программы. Предположим, что $\mathcal{B}_1, \ldots, \mathcal{B}_k$ — все прямые предки участка $\mathcal{B}$ в P (одним из них может быть сам участок $\mathcal{B}$). Ясно, что

$$\mathrm{IN}(\mathcal{B}) = \bigcup_{i=1}^{k} \mathrm{OUT}(\mathcal{B}_i)$$
$$= \bigcup_{i=1}^{k} [(\mathrm{IN}(\mathcal{B}_i) \cap \mathrm{TRANS}(\mathcal{B}_i)) \cup \mathrm{GEN}(\mathcal{B}_i)]$$

Для вычисления IN($\mathcal{B}$) можно было бы выписать это уравнение для каждого участка программы вместе с уравнением $\mathrm{IN}(\mathcal{B}_0) = \varnothing$, где $\mathcal{B}_0$ — начальный участок, а затем попытаться разрешить систему уравнений[1]). Однако мы дадим другой метод решения, учитывающий преимущества представления графов управления в виде интервалов. Определим сначала, что мы понимаем под входом и выходом из интервала.

Определение. Пусть P — программа, а F_0 — ее граф управления. Пусть $F_0, F_1, \ldots, F_n$ — производная последовательность от F_0. Каждая вершина в F_i, $i \geqslant 1$, является интервалом в F_{i-1} и называется *интервалом порядка* i.

Входом интервала порядка 1 служит заголовок интервала (отметим, что этот заголовок — участок программы). *Вход* интервала порядка $i > 1$ — это вход в заголовок этого интервала. Таким образом, вход любого интервала — это линейный участок исходной программы P.

Выходом интервала $I(n)$ порядка 1 служит такой последний оператор участка $\mathcal{B}$ в $I(n)$, что $\mathcal{B}$ имеет прямого потомка, который является либо заголовком интервала n, либо участком вне $I(n)$. *Выход* интервала $I(n)$ порядка $i > 1$ — это последний

[1]) Как и в случае уравнений над регулярными выражениями разд. 2.2, решение может быть не единственным. Здесь нам хотелось бы иметь наименьшее решение.

оператор участка $\mathcal{B}$, содержащегося в $I(n)$[1]) и такого, что в F_0 есть дуга, ведущая из $\mathcal{B}$ либо в заголовок интервала n, либо в участок вне $I(n)$.

Отметим, что каждый интервал имеет один вход и нуль или более выходов.

Пример 11.43. Пусть F_0—граф управления на рис. 11.41. С помощью алгоритма 11.6 построим его разбиение на интервалы:

$$I_1 = I(\mathcal{B}_1) = \{\mathcal{B}_1\}$$
$$I_2 = I(\mathcal{B}_2) = \{\mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4\}$$

Из этих интервалов можно построить первый производный граф F_1, показанный на рис. 11.42. Из F_1 можно построить его интервалы (в данном случае только один):

$$I_3 = I(I_1) = \{I_1, I_2\} = \{\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4\}$$

и получить предельный граф управления, также показанный на рис. 11.42.

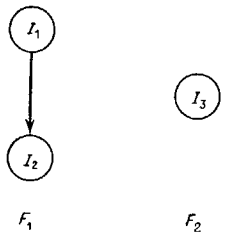


Рис. 11.42. Производная последовательность от F_0.

Интервалами порядка 1 являются I_1 и I_2. Вход для I_2—это $\mathcal{B}_2$. Вход для I_3—это $\mathcal{B}_1$. Единственный выход для I_1—оператор $S2$. Единственный выход для I_2—оператор $S9$. Интервалом порядка 2 является I_3 с входом $\mathcal{B}_1$. Интервал I_3 не имеет выходов. □

Продолжим теперь функции IN, OUT, TRANS и GEN на интервалы. Пусть $F_0, F_1, \ldots, F_n$—производная последовательность от F_0, где F—граф управления для P. Пусть $\mathcal{B}$—участок в P, а I—интервал некоторого графа F_i, $i \geqslant 1$. Введем следующие рекурсивные определения:

$$(1)\quad \mathrm{IN}(I) = \begin{cases} \mathrm{IN}(\mathcal{B}), & \text{если } I \text{ имеет порядок } 1, \\ & \text{а } \mathcal{B}\text{—заголовок для } I, \\ \mathrm{IN}(I'), & \text{если } I \text{ имеет порядок } i > 1, \\ & \text{а } I'\text{—заголовок для } I. \end{cases}$$

В (2)—(4) ниже s—выход для I.

(2) $\mathrm{OUT}(I, s) = \mathrm{OUT}(\mathcal{B})$, где участок $\mathcal{B} \in I$ таков, что s—его последний оператор.

(3) (а) $\mathrm{TRANS}(\mathcal{B}, s) = \mathrm{TRANS}(\mathcal{B})$, если s—последний оператор в $\mathcal{B}$.

(б) $\mathrm{TRANS}(I, s)$—множество таких операторов $d \in P$, что существуют путь без циклов $I_1, I_2, \ldots, I_k$, состоящий исключительно из вершин в I, и такая последовательность выходов $s_1, \ldots, s_k$ для $I_1, \ldots, I_k$ соответственно, что

(i) I_1—заголовок для I,

(ii) в F_0 участок s_j является прямым предком входа для I_{j+1} при $1 \leqslant j < k$,

(iii) $d \in \mathrm{TRANS}(I_j, s_j)$ для $1 \leqslant j \leqslant k$,

(iv) $s_k = s$.

Эти условия проиллюстрированы на рис. 11.43.

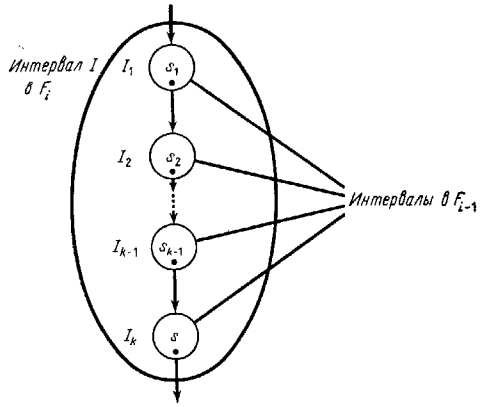


Рис. 11.43. TRANS(I, s).

(4) (а) $\mathrm{GEN}(\mathcal{B}, s) = \mathrm{GEN}(\mathcal{B})$, если s—последний оператор в $\mathcal{B}$.

(б) $\mathrm{GEN}(I, s)$—множество таких операторов $d \in P$, что существуют путь без циклов $I_1, \ldots, I_k$, состоящий

[1]) Строго говоря, интервал I порядка $i > 1$ состоит из интервалов порядка $i-1$. Будем неформально говорить, что участок $\mathcal{B}$ содержится в I, если он принадлежит одному из интервалов, входящих в I. Таким образом, множество участков, представляющих интервал произвольного порядка, определено так, как этого и следовало ожидать.

исключительно из вершин в I, и такая последовательность выходов $s_1, \ldots, s_k$ для $I_1, \ldots, I_k$ соответственно, что

(i) $d \in \mathrm{GEN}(I_1, s_1)$,

(ii) в F_0 участок s_j является прямым предком входа для I_{j+1}, $1 \leqslant j < k$,

(iii) $d \in \mathrm{TRANS}(I_j, s_j)$ при $2 \leqslant j \leqslant k$,

(iv) $s_k = s$.

Отметим, что здесь интервал I_1 не обязан быть заголовком для I.

Таким образом, $\mathrm{TRANS}(I, s)$ — это множество определений, которые можно передать со входа в I на выход s без переопределения в I, $\mathrm{GEN}(I, s)$ — это множество определений в I, которые без переопределений могут достичь s.

Пример 11.44. Рассмотрим F_0 на рис. 11.41 и F_1 и F_2 на рис. 11.42. В F_1 интервал I_2 — это $\{\mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4\}$, и он имеет выход $S9$. Таким образом, $\mathrm{IN}(I_2) = \mathrm{IN}(\mathcal{B}_2) = \{S1, S2, S3, S8\}$ и $\mathrm{OUT}(I_2, S9) = \mathrm{OUT}(\mathcal{B}_3) = \{S3, S8\}$.

$\mathrm{TRANS}(I_2, S9) = \varnothing$, поскольку $\mathrm{TRANS}(\mathcal{B}_2) = \varnothing$.

$\mathrm{GEN}(I_2, S9)$ содержит $S8$, поскольку существует последовательность участков, состоящая только из $\mathcal{B}_3$, в которой $S8 \in \mathrm{GEN}(\mathcal{B}_3, S9)$. Кроме того, $S3 \in \mathrm{GEN}(I_2, S9)$, поскольку существует последовательность участков $\mathcal{B}_2$, $\mathcal{B}_3$ с выходами $S5$ и $S9$. Это означает, что $S3 \in \mathrm{GEN}(\mathcal{B}_2, S5)$, $\mathcal{B}_2$ — прямой предок участка $\mathcal{B}_3$, $S3 \in \mathrm{TRANS}(\mathcal{B}_3, S9)$. □

Дадим теперь алгоритм вычисления $\mathrm{IN}(\mathcal{B})$ для всех участков программы P. Он обрабатывает только программы со сводимым графом управления. Модификации, необходимые для обработки программ с несводимыми графами, приведены в следующем разделе.

Алгоритм 11.7. Вычисление функции IN.

Вход. Сводимый граф управления F_0 для программы P.

Выход. $\mathrm{IN}(\mathcal{B})$ для каждого участка $\mathcal{B} \in P$.

Метод.

(1) Пусть $F_0, F_1, \ldots, F_k$ — производная последовательность от F_0. Вычисляем $\mathrm{TRANS}(\mathcal{B})$ и $\mathrm{GEN}(\mathcal{B})$ для всех участков $\mathcal{B}$ из F_0.

(2) Для $i = 1, \ldots, k$ последовательно вычисляем $\mathrm{TRANS}(I, s)$ и $\mathrm{GEN}(I, s)$ для всех интервалов порядка i и выходов s для I. Рекурсивное определение этих функций гарантирует, что это можно сделать.

(3) Полагаем $\mathrm{IN}(I) = \varnothing$, где I — одиночный интервал порядка k. Устанавливаем $i = k$.

(4) Для всех интервалов порядка i выполняем следующее. Пусть $I = \{I_1, \ldots, I_n\}$ — интервал порядка i ($I_1, \ldots, I_n$ — интервалы порядка $i-1$ или участки, если $i = 1$). Можно считать, что эти подынтервалы перечислены в том порядке, в котором из них составлялся интервал I в алгоритме 11.6. Иными словами, I_1 — это заголовок и для каждого $j > 1$ множество $\{I_1, \ldots, I_{j-1}\}$ содержит все вершины из F_{i-1}, являющиеся прямыми предками интервала I_j.

(а) Пусть $s_1, s_2, \ldots, s_r$ — выходы интервала I, каждый из которых принадлежит участку в F_0, являющемуся прямым предком входа для I. Полагаем

$$\mathrm{IN}(I_1) = \mathrm{IN}(I) \cup \bigcup_{i=1}^{r} \mathrm{GEN}(I, s_i)$$

(б) Для всех выходов $s \in I_1$[1]) полагаем

$$\mathrm{OUT}(I_1, s) = (\mathrm{IN}(I_1) \cap \mathrm{TRANS}(I_1, s)) \cup \mathrm{GEN}(I_1, s)$$

(в) Для $j = 2, \ldots, n$ пусть $s_{r1}, s_{r2}, \ldots, s_{rk_r}$ — выходы интервала I_r, $1 \leqslant r < j$, каждый из которых принадлежит участку в F_0, являющемуся прямым предком входа для I_j. Полагаем

$$\mathrm{IN}(I_j) = \bigcup_{r,\, l} \mathrm{OUT}(I_r, s_{rl})$$

$$\mathrm{OUT}(I_j, s) = (\mathrm{IN}(I_j) \cap \mathrm{TRANS}(I_j, s)) \cup \mathrm{GEN}(I_j, s)$$

для всех выходов s интервала I_j.

(5) Если $i = 1$, остановиться. В противном случае уменьшить i на 1 и вернуться к шагу (4). □

Пример 11.45. Применим алгоритм 11.7 к графу управления на рис. 11.41. Для четырех участков в F_0 GEN и TRANS вычисляются просто. Результаты вычислений приведены в табл. 11.9.

Таблица 11.9

Участок	GEN	TRANS
$\mathcal{B}_1$	$\{S1, S2\}$	$\varnothing$
$\mathcal{B}_2$	$\{S3, S4\}$	$\varnothing$
$\mathcal{B}_3$	$\{S8\}$	$\{S2, S3\}$
$\mathcal{B}_4$	$\varnothing$	$\{S1, S2, S3, S4, S8\}$

[1]) Если интервал имеет два выхода, ведущих к одному и тому же следующему интервалу, то их для эффективности выполнения можно „слить". „Слияние" состоит в объединении функций GEN и TRANS.

Например, поскольку $\mathcal{B}_3$ определяет только переменную I, $\mathcal{B}_3$ „убивает" предыдущие ее определения, но передает определения переменной J, а именно $S2$ и $S3$. Поскольку никакой из участков не определяет переменную дважды, все операторы определения внутри участка принадлежат множеству GEN для данного участка.

Заметим, что интервал I_1, состоящий из одного участка $\mathcal{B}_1$, имеет один выход — оператор $S2$. Так как пути в I_1 тривиальны, то $\mathrm{GEN}(I_1, S2) = \{S1, S2\}$ и $\mathrm{TRANS}(I_1, S2) = \varnothing$.

Интервал I_2 имеет выход $S9$. В примере 11.44 мы видели, что $\mathrm{GEN}(I_2, S9) = \{S3, S8\}$ и $\mathrm{TRANS}(I_2, S9) = \varnothing$.

Теперь можно начать вычисление функции IN. Как и требуется, $\mathrm{IN}(I_3) = \varnothing$. Затем к двум подынтервалам в I_3 можно применить шаг (4) алгоритма 11.7. Это можно сделать только в порядке I_1, I_2. На шаге (4а) вычисляем $\mathrm{IN}(I_1) = \mathrm{IN}(I_3) = \varnothing$, а на шаге (4б) —

$$\mathrm{OUT}(I_1, S2) = (\mathrm{IN}(I_1) \cap \mathrm{TRANS}(I_1, S2)) \cup \mathrm{GEN}(I_1, S2) = \{S1, S2\}$$

Далее, на шаге (4в)

$$\mathrm{IN}(I_2) = \mathrm{OUT}(I_1, S2) = \{S1, S2\}$$

Проходя по интервалам порядка 1, мы должны рассмотреть составляющие для I_1 и I_2. Интервал I_1 состоит только из $\mathcal{B}_1$, так что вычисляем $\mathrm{IN}(\mathcal{B}_1) = \varnothing$. Интервал I_2 состоит из участков $\mathcal{B}_2$, $\mathcal{B}_3$ и $\mathcal{B}_4$, которые в таком порядке и можно рассматривать. На шаге (4а)

$$\mathrm{IN}(\mathcal{B}_2) = \mathrm{IN}(I_2) \cup \mathrm{GEN}(I_2, S9) = \{S1, S2, S3, S8\}$$

На шаге (4б)

$$\mathrm{OUT}(\mathcal{B}_2, S5) = (\mathrm{IN}(\mathcal{B}_2) \cap \mathrm{TRANS}(\mathcal{B}_2, S5)) \cup \mathrm{GEN}(\mathcal{B}_2, S5) = \{S3, S4\}$$

Поскольку $S5$ ведет к $\mathcal{B}_3$, находим

$$\mathrm{IN}(\mathcal{B}_3) = \mathrm{OUT}(\mathcal{B}_2, S5) = \{S3, S4\}$$

а так как $S5$ также ведет к $\mathcal{B}_4$, то

$$\mathrm{IN}(\mathcal{B}_4) = \mathrm{OUT}(\mathcal{B}_2, S5) = \{S3, S4\}$$

Итак,

$$\begin{aligned}
&\mathrm{IN}(\mathcal{B}_1) = \varnothing \\
&\mathrm{IN}(\mathcal{B}_2) = \{S1, S2, S3, S8\} \\
&\mathrm{IN}(\mathcal{B}_3) = \{S3, S4\} \\
&\mathrm{IN}(\mathcal{B}_4) = \{S3, S4\} \quad \square
\end{aligned}$$

Индукцией по порядку интервала I можно доказать, что

(1) $\mathrm{TRANS}(I, s)$ — множество таких операторов определений $d \in P$, что существует путь из первого оператора заголовка для I вплоть до s, вдоль которого ни один из операторов не переопределяет переменную, определенную в d,

(2) $\mathrm{GEN}(I, s)$ — множество таких определений d, что существует путь из d в s, вдоль которого ни один из операторов не переопределяет переменную, определенную в d.

Индукцией по числу применений шага (4) алгоритма 11.7 можно доказать, что

(11.4.1) если для вычисления $\mathrm{IN}(I_j)$ применяется шаг (4), то $\mathrm{IN}(I_j)$ — множество таких определений d, что существует путь из d во вход для I_j, вдоль которого ни один из операторов не переопределяет переменную, определенную в d, а $\mathrm{OUT}(I_j, s)$ — множество таких d, что существует путь из d в s, вдоль которого ни один из операторов не переопределяет переменную, определенную в d.

Частным случаем утверждения (11.4.1), когда I_j — участок, является следующая теорема.

Теорема 11.14. *Для всех линейных участков $\mathcal{B} \in P$ множество $\mathrm{IN}(\mathcal{B})$ в алгоритме 11.7 — это множество таких определений d, что в F_0 существует путь из d в первый оператор участка $\mathcal{B}$, вдоль которого ни один из операторов не переопределяет переменную, определенную в d.* $\square$

11.4.3. Несводимые графы управления

Поскольку не каждый граф управления сводим, введем еще одно понятие, называемое расщеплением вершин, позволяющее обобщить алгоритм 11.7 на все графы управления. Вершина, в которую входит более одной дуги, „расщепляется" на несколько одинаковых копий, по одной на каждую входящую дугу. Таким образом, каждая копия имеет единственную входящую дугу и становится частью интервала для вершины, из которой идет эта дуга. Поэтому расщепление вершины с последующим построением интервала уменьшает число вершин графа по крайней мере на 1. Повторяя, если необходимо, этот процесс, можно превратить любой несводимый граф управления в сводимый.

Пример 11.46. Рассмотрим несводимый граф управления на рис. 11.40. Вершину n_3 можно расщепить на две копии n_3' и n_3'', получив граф F', изображенный на рис. 11.44. Интервалами для F' будут

$$\begin{aligned}
I_1 &= I(n_1) = \{n_1, n_3'\} \\
I_2 &= I(n_2) = \{n_2, n_3''\}
\end{aligned}$$

Первый производный от F' граф F_1' имеет две вершины (рис. 11.45). Второй производный граф состоит из единственной вершины. Таким образом, с помощью расщепления вершин мы превратили F в сводимый граф управления F'. □

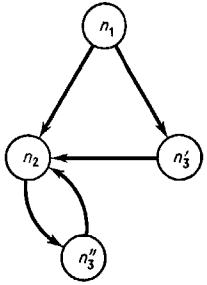


Рис. 11.44. Расщепленный граф управления.

Рис. 11.45. Первый производный граф.

Дадим теперь модифицированный вариант алгоритма 11.7, учитывающий этот новый метод. Начнем с простого полезного замечания.

Лемма 11.15. *Если G — граф управления и $I(G)=G$, то любая вершина n, отличная от начальной, имеет по крайней мере две входящие дуги; ни одна из них не выходит из n.*

Доказательство. Все дуги, выходящие из некоторой вершины и входящие в нее же, исчезают при построении интервалов. Поэтому предположим, что вершина n имеет только одну входящую дугу, выходящую из другой вершины m. Тогда n принадлежит $I(m)$. Если $I(G)=G$, то вершина m в конце концов появляется в списке заголовков алгоритма 11.6. Но тогда n заносится в $I(m)$, так что $I(G)$ не может совпасть с G. □

Алгоритм 11.8. Общее вычисление функции IN.

Вход. Произвольный граф управления F программы P.
Выход. IN($\mathcal{B}$) для каждого участка $\mathcal{B}$ из P.
Метод.

(1) Для каждого участка $\mathcal{B} \in F$ вычисляем GEN($\mathcal{B}$) и TRANS($\mathcal{B}$). Затем к F рекурсивно применяем шаг (2). Входом для шага (2) служит граф управления G вместе с GEN(I, S) и TRANS(I, s), известными для каждой вершины $I \in G$ и каждого выхода s интервала I. Выходом шага (2) служит IN(I) для каждой вершины $I \in G$.

(2) (а) Пусть G — вход для этого шага и $G, G_1, \ldots, G_k$ — его производная последовательность. Если G_k — одиночная вершина, продолжаем в точности так же, как в алгоритме 11.7. Если G_k — не одиночная вершина, то для всех вершин в $G_1, \ldots, G_k$ можно вычислить GEN и TRANS, как в алгоритме 11.7. Тогда по лемме 11.15 G_k содержит некоторую вершину, отличную от начальной, в которую входит более одной дуги. Выберем одну из таких вершин I. Если в I входит j дуг, заменяем I новыми вершинами $I_1, \ldots, I_j$. В каждую из $I_1, \ldots, I_j$ входит по одной дуге, все они выходят из различных вершин, из которых ранее шли дуги в I.

(б) Для каждого выхода s интервала I порождаем выход s_i для $1 \leqslant i \leqslant j$ и считаем, что в F есть дуга, ведущая из каждого s_i во вход каждой вершины, с которой s связан в G_k. Определяем GEN(I_i, s_i) = GEN(I, s) и TRANS(I_i, s_i) = = TRANS(I, s) для $1 \leqslant i \leqslant j$. Результирующий граф обозначим G'.

(в) Применим шаг (2) к G'. Функция IN будет рекурсивно вычисляться для G'. Затем, полагая $\mathrm{IN}(I) = \bigcup_{i=1}^{j} \mathrm{IN}(I_i)$, вычисляем IN для вершин из G_k. Никакие другие изменения для IN не требуются.

(г) Как и в алгоритме 11.7, вычисляем IN для G по IN для G_k.

(3) После завершения шага (2) функция IN будет вычислена для каждого участка из F. Эта информация и образует выход алгоритма. □

Пример 11.47. Рассмотрим граф управления F_0 на рис. 11.46,*а*. Можно вычислить граф $F_1 = I(F_0)$, изображенный на рис. 11.46,*б*. Однако $I(F_1) = F_1$, так что надо применять процедуру расщепления вершин шага (2). Пусть $\{n_3, n_5\}$ — вершина I, которая расщепляется на I_1 и I_2. Результат показан на рис. 11.47. Свяжем n_1 с I_1, а $\{n_3, n_4\}$ с I_2. На рисунке изображены дуги из I_1 и I_2 в $\{n_3, n_4\}$. В действительности каждый выход для I дублируется — один для I_1 и один для I_2. С входом для $\{n_3, n_4\}$ связаны именно продублированные выходы, и этот факт на рис. 11.47 представлен двумя дугами. Отметим, что граф на рис. 11.47 — сводимый. □

Теорема 11.15. *Алгоритм* 11.8 *всегда заканчивается.*

Доказательство. По лемме 11.15 каждое обращение к шагу (2) осуществляется либо на сводимом графе, и в этом случае можно быть уверенным, что вычисление закончится; либо на вершине I, которую можно расщепить. Заметим, что каждый из интервалов $I_1, \ldots, I_j$, порождаемых на шаге (2), имеет един-

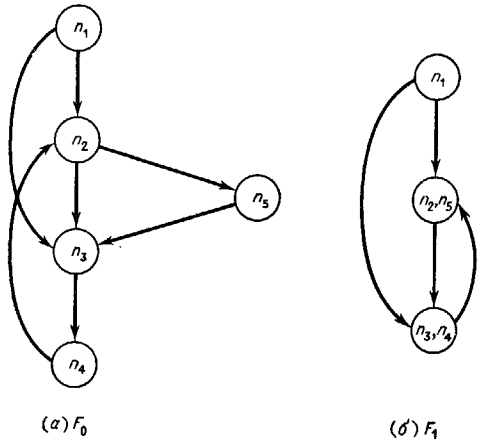


Рис. 11.46. Несводимые графы управления.

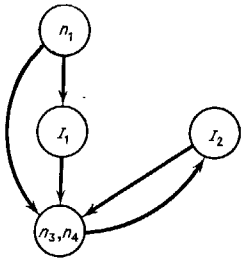


Рис. 11.47. Граф управления.

ственную входящую дугу. Таким образом, при построении интервалов каждый из них окажется в интервале с другой вершиной в качестве заголовка. Из этого можно заключить, что следующее обращение к шагу (2) будет на графе, имеющем по крайней мере на одну вершину меньше, так что алгоритм 11.8 должен закончиться. □

Теорема 11.16. *Алгоритм* 11.8 *правильно вычисляет функцию* IN.

Доказательство. Достаточно заметить, что GEN и TRANS для $I_1, \ldots, I_j$ на шаге (2) те же, что и для I. Кроме того, ясно,

что $\mathrm{IN}(I) = \bigcup_{i=1}^{j} \mathrm{IN}(I_i)$ и $\mathrm{OUT}\ (I) = \bigcup_{i=1}^{j} \mathrm{OUT}$. Поскольку каждый интервал I_i связывает те же вершины, что и I, функция IN для вершин в G_k, отличных от I, та же, что и в G'. Таким образом, простая индукция по числу обращений к шагу (2) показывает, что IN вычисляется правильно. □

11.4.4. Обзор содержания главы

При построении оптимизирующего компилятора сначала надо решить, какие лучше всего применить оптимизации. Решение должно базироваться на характеристиках того класса программ, которые, как предполагается, должны компилироваться. К сожалению, часто эти характеристики трудно определить, а работ по этому вопросу опубликовано мало.

В этой главе мы рассматривали оптимизацию кода с довольно общей точки зрения, и разумно было бы спросить, как связаны друг с другом различные обсуждаемые нами оптимизации кода.

Методами разд. 11.2, касающимися арифметических выражений, можно пользоваться при построении окончательной объектной программы. Однако некоторые аспекты алгоритмов разд. 11.2 можно внести также и в генерацию промежуточного кода. Иными словами, отдельные части алгоритмов разд. 11.2 можно встроить в выход синтаксического анализатора. Это приведет к тому, что на линейных участках будут эффективно использоваться регистры.

На фазе компилятора, генерирующей код, у нас есть промежуточная программа, которую можно рассматривать как нечто аналогичное „программам" из разд. 11.3. Наша первая задача — построить граф управления способом, описанным в этом разделе. Следующий возможный шаг — оптимизировать циклы, как описано в разд. 11.3, сначала внутренние, а затем и внешние.

Когда это сделано, можно вычислить глобальную информацию относительно потока данных, например, как в алгоритме 11.8 и/или в упр. 11.4.19 и 11.4.20. Зная эту информацию, можно выполнить „глобальные" оптимизации из разд. 11.3, например размножение констант и исключение общих подвыражений. На этой стадии, однако, надо проявлять осторожность, чтобы не увеличить размера внутренних циклов. Для этого можно поместить участки во внутренних циклах и избегать в таких участках дополнительных действий, связанных с запоминанием значения выражения, даже если это выражение используется позднее в участке вне цикла. Если машина, для которой генериру-

ется код, имеет более одного регистра, можно выявить активные переменные (упр. 11.4.20), чтобы определить, какие переменные должны занимать регистры при выходе из участков.

Наконец, линейные участки можно преобразовать методами разд. 11.1 или аналогичными методами в зависимости от конкретного промежуточного языка. На этой стадии осуществляется также распределение регистров внутри участка, удовлетворяющее упомянутым выше ограничениям на глобальное распределение регистров. Здесь обычно требуются эвристические методы.

УПРАЖНЕНИЯ

11.4.1. Постройте производную последовательность графов управления, изображенных на рис. 11.32 и 11.36. Сводимы ли эти графы?

11.4.2. Приведите дополнительные примеры несводимых графов управления.

11.4.3. Докажите утверждения (1) и (2) теоремы 11.12.

***11.4.4.** Покажите, что алгоритм 11.6 можно выполнить так, что он будет тратить время, пропорциональное числу дуг графа управления F.

11.4.5. Докажите теорему 11.14.

11.4.6. Закончите доказательство теоремы 11.16.

11.4.7. Используйте анализ интервалов (алгоритм 11.7) как основу алгоритма, выясняющего по данному оператору, ссылающемуся на переменную A, определена ли она явно так, что будет принимать в качестве значения одну и ту же константу при каждом выполнении оператора. *Указание*: Необходимо найти, какие из операторов, определяющих A, могут быть предыдущими определениями A до очередного выполнения рассматриваемого оператора. Это легко выяснить, если предыдущий оператор, определяющий A, встречается в том же участке, что и рассматриваемый оператор. Если нет, нужно вычислить $\mathrm{IN}(\mathcal{B})$ для участка, в котором встречается оператор. В последнем случае говорят, что A имеет значением константу тогда и только тогда, когда все операторы определения в $\mathrm{IN}(\mathcal{B})$, определяющие A, присваивают A одну и ту же константу.

11.4.8. Приведите алгоритм, использующий анализ интервалов как основу для выяснения, бесполезен ли оператор S, т. е. существует ли оператор, употребляющий значение, определяемое S.

11.4.9. Пусть $\mathcal{B}$ — участок графа управления с дугами в $\mathcal{B}_1$ и $\mathcal{B}_2$, а d — оператор управления в $\mathcal{B}$, значение которого в $\mathcal{B}$

не употребляется. Если никакой из участков, достижимых из $\mathcal{B}_2$, не употребляет значения, определяемого оператором d, то d можно переместить в $\mathcal{B}_1$. Используйте анализ интервалов как основу для алгоритма, выявляющего такие ситуации.

11.4.10. Вычислите IN для каждого участка программы

```
        N ← 2
Y:      I ← 2
W:      if I < N goto X
        write N
Z:      N ← N + 1
        goto Y
X:      J ← remainder (N, I)
        if J = 0 goto Z
        I ← I + 1
        goto W
```

11.4.11. Вычислите IN для каждого участка программы

```
        read I
        if I = 1 goto X
Z:      if I > 10 goto Y
X:      J ← I + 3
        write J
W:      I ← I + 1
        goto Z
Y:      I ← I − 1
        if I > 15 goto W
        halt
```

***11.4.12.** Пусть T_1 и T_2 — преобразования, следующим образом определенные на графах управления:

T_1: Если вершина n имеет дугу в себя, удаляем эту дугу.

T_2: Если в вершину n входит единственная дуга, выходящая из вершины m, и n не является начальной вершиной, сливаем m и n, заменяя дугу из m в n дугами из m в каждую вершину n', в которую раньше шла дуга из n. Затем исключаем n.

Покажите, что если T_1 и T_2 применяются к графу управления F до тех пор, пока они применимы, то результатом будет предел графа F.

***11.4.13.** Примените преобразования T_1 и T_2 для вычисления функции IN другим способом, не использующим анализа интервалов.

****11.4.14.** Пусть G — граф управления с начальной вершиной n_0. Покажите, что G несводим тогда и только тогда, когда он имеет такие вершины n_1, n_2 и n_3, что существуют пути из n_0 в n_1, из n_1 в n_2 и в n_3, из n_2 в n_3 и из n_3 в n_2 (рис. 11.48), совпадающие только в их конечных точках. Все n_0, n_1, n_2 и n_3 должны быть различны, только n_1 может совпадать с n_0.

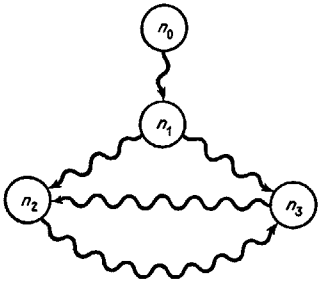


Рис. 11.48. Структура любого несводимого графа.

11.4.15. Покажите, что любая d-схема (см. разд. 1.3.2) сводима. *Указание*: Воспользуйтесь упр. 11.4.14.

11.4.16. Покажите, что любая программа на Фортране, в которой каждый переход на предыдущий оператор вызывается циклом DO, имеет сводимый граф управления.

****11.4.17.** Покажите, что можно за время $0(n \log n)$ выяснить, сводим ли данный граф управления. *Указание*: Воспользуйтесь упр. 11.4.12.

***11.4.18.** Какая существует связь между понятиями интервала и области с одним входом?

***11.4.19.** Приведите основанный на понятии интервала алгоритм, выясняющий для каждого выражения (скажем, $A+B$) и каждого участка $\mathscr{B}$, при каждом ли выполнении программы достигается оператор, вычисляющий $A+B$ (т. е. существует ли оператор вида $C \leftarrow A+B$) и не переопределяющий впоследствии ни A, ни B. *Указание:* Если $\mathscr{B}$ не является начальным участком, то положим $\mathrm{IN}(\mathscr{B}) = \bigcap_i \mathrm{OUT}(\mathscr{B}_i)$, где все $\mathscr{B}_i$ — прямые предки участка $\mathscr{B}$. Пусть

$$\mathrm{OUT}(\mathscr{B}) = (\mathrm{IN}(\mathscr{B}) \cap X) \cup Y$$

где X — множество выражений, „убиваемых" в $\mathscr{B}$ (мы „убиваем" $A+B$, переопределяя A или B), а Y — множество выражений, вычисляемых участком и неубиваемых. Для каждого интервала I различных производных графов и каждого его выхода s вычисляем $\mathrm{GEN}'(I, S)$ как множество выражений, вычисляемых и впоследствии не убиваемых ни на каком пути из входа в интервал I в его выход. Вычисляем также $\mathrm{TRANS}'(I, s)$ как множество выражений, которые, будучи убитыми, генерируются затем на любом таком пути. Отметим, что $\mathrm{GEN}'(I, s) \subseteq \mathrm{TRANS}'(I, s)$.

***11.4.20.** Приведите алгоритм, основанный на анализе интервалов, который для каждой переменной A и каждого участка $\mathscr{B}$ выясняет, есть ли выполнение, которое после прохождения через $\mathscr{B}$ будет ссылаться на переменную A перед ее переопределением.

11.4.21. Пусть F — граф с n вершинами и e дугами. Покажите, что i-й производный граф от F имеет не более чем $e-i$ дуг.

***11.4.22.** Приведите пример графа управления с n вершинами и $2n$ дугами, производная последовательность которого имеет длину n.

***11.4.23.** Покажите, что алгоритм 11.7 и алгоритмы из упр. 11.4.19 и 11.4.20 тратят не более $0(n^2)$ элементарных векторных шагов на графах управления с n вершинами и не более чем $2n$ дугами.

Существует другой подход к анализу потока данных, табличный по своей природе. Например, по аналогии с алгоритмом 11.8 можно построить таблицу значений $\mathrm{IN}(d, \mathscr{B})$, равных 1, если в $\mathrm{IN}(\mathscr{B})$ есть определение d, и 0 в противном случае. Вначале пусть $\mathrm{IN}(d, \mathscr{B}) = 1$ тогда и только тогда, когда из некоторой вершины $\mathscr{B}'$ ведет дуга в $\mathscr{B}$ и $d \in \mathrm{GEN}(\mathscr{B}')$. Для каждой единицы, добавленной в таблицу, скажем на элемент $(d, \mathscr{B})$, помещаем 1 в элемент $(d, \mathscr{B}'')$, если есть дуга из $\mathscr{B}$ в $\mathscr{B}''$ и $\mathscr{B}$ не убивает d.

***11.4.24.** Покажите, что приведенный выше алгоритм правильно вычисляет $\mathrm{IN}(\mathscr{B})$ и занимает $0(mn)$ времени на графе управления с n вершинами, не более чем $2n$ дугами и m определениями.

***11.4.25.** Дайте алгоритмы, аналогичные приведенному выше и решающие те же задачи, что и в упр. 11.4.19 и 11.4.20. Как быстро они работают?

****11.4.26.** Приведите алгоритмы, тратящие $0(n \log n)$ элементарных векторных шагов на вычисление функций IN алгоритма 11.7 или упр. 11.4.19 для графов управления с n вершинами.

****11.4.27.** Покажите, что граф управления сводим тогда и только тогда, когда множество его дуг можно разбить на два таких множества E_1 и E_2, что 1) E_1 образует ориентированный ацик-

лический граф и 2) если (m, n) принадлежит E_2, то $m=n$ или n доминирует над m.

**11.4.28. Приведите алгоритм, вычисляющий прямые доминаторы сводимого графа с n вершинами за время $O(n \log n)$.

Проблемы для исследования

11.4.29. Предложите дополнительную информацию относительно потока данных (не упоминаемую в алгоритме 11.7 и упр. 11.4.19 и 11.4.20), которая пригодилась бы для целей оптимизации кода. Дайте алгоритмы, вычисляющие эту информацию как для сводимых, так и для несводимых графов управления.

11.4.30. Существуют ли методы вычисления функций IN алгоритма 11.8 или других функций, связанных с потоком данных, которые были бы предпочтительнее для расщепления вершин несводимых графов? Под „предпочтительностью" мы понимаем, что допустимы операции над элементарными векторами и что алгоритмы упр. 11.4.24 и 11.4.25 становятся явно оптимальными.

Замечания по литературе

Подход к оптимизации с точки зрения анализа интервалов был развит Коком [1970] и затем разработан Коком и Шварцем [1970] и Алленом [1970]. Кеннеди [1971] рассматривает глобальный алгоритм, использующий анализ интервалов для распознавания в программе активных переменных (упр. 11.4.20).

Решения упр. 11.4.12—11.4.16 можно найти у Хекта и Ульмана [1972а]. Упр. 11.4.17 взято из работы Хопкрофта и Ульмана [1972б]. Ответ на упр. 11.4.19 можно найти у Кока [1970] или Шефера [1973]. Упр. 11.4.24 и 11.4.26 взяты у Ульмана [1972б], упр. 11.4.27 — у Хекта и Ульмана [1972б], упр. 11.4.28 — у Ахо и др. [1972]. В ряде статей обсуждается реализация оптимизирующих компиляторов. Лоури и Медлок [1969] приводят некоторые оптимизации, применяемые в компиляторе OS/360 FORTRAN Н. Бузам и Энглунд [1969] описывают методы для распознавания общих подвыражений, удаления инвариантных вычислений из циклов и распределения регистров для другого компилятора с Фортрана.

Кнут [1971] собрал много программ на Фортране и проанализировал некоторые их характеристики [1]).

---

[1]) Анализ потока данных, использующий представление программы в виде операторной схемы, применяется в АЛЬФА-трансляторе (Ершов и др. [1965]). О применении графов управления программ для целей оптимизации см. также [Касьянов, Трахтенброт, 1975], [Поттосин, 1973, 1975], [Касьянов, 1975]. — *Прим. перев.*

СПИСОК ЛИТЕРАТУРЫ К I и 2 ТОМАМ [1])

Айронс [1961] (Irons E. T.), A syntax directed compiler for ALGOL 60, *Comm. ACM*, **4**:1, 51—55.

Айронс [1963а] (Irons E. T.), An error correcting parse algorithm, *Comm. ACM*, **6**:11, 669—673.

Айронс [1963б] (Irons E. T.) The structure and use of the syntax directed compiler, *Ann. Rev. Auton. Program.*, **3**, 207—227.

Айронс [1964] (Irons E. T.), Structural connections in formal languages, *Comm. ACM*, **7**:2, 62—67.

Аллард, Вольф, Землин [1964] (Allard R. W., Wolf K. A., Zemlin R. A.), Some effects of the 6600 computer on language structures, *Comm. ACM*, **7**:2, 112—127.

Аллен [1969] (Allen F. E.), Program optimization, *Ann. Rev. Autom. Program.*, **5**.

Аллен [1970] (Allen F. E.), Control flow analysis, *ACM SIGPLAN Notices*, **5**: 7, 1—19.

Аллен, Кок [1972] (Allen F. E., Cocke J.), A catalogue of optimizing transformations, сб. Design and Optimization of Compilers, под ред. Rustin R., Prentice-Hall, Englewood Cliffs, N. J., pp. 1—30.

Ангер [1968] (Unger S. H.), A global parser for context-free phrase structure grammars, *Comm. ACM*, **11**:4, 240—246; **11**:6, 427.

Андерсон [1964] (Anderson J. P.), A note on some compiling algorithms, *Comm. ACM*, **7**:3, 149—150.

АНС [1966] (Ans X3.9), American National Standards FORTRAN, American National Standards Institute, New York.

АНС Подкомитет [1971] (Ansi Subcommittee X3J3), Clarification of FORTRAN Standards—Second Report, *Comm. ACM*, **14**:10, 628—642.

Арбиб [1970] (Arbib M. A.), Theories of abstract automata, Prentice-Hall, Inc., Englewood Cliffs, N. J.

Ахо [1968] (Aho A. V.), Indexed grammars—an extention of context-free grammars, *J. ACM*, **15**:4, 647—671. (Русский перевод: Ахо А., Индексные грамматики—расширение контекстно-свободных грамматик, в сб. „Языки и автоматы", изд-во „Мир", М., 1975, стр. 130—165.)

Ахо [1973] (Aho A. V. (ed.)), Currents in the theory of computing, Prentice-Hall, Englewood Cliffs, N. J.

° Ахо, Ульман [1968] (Aho A. V., Ullman J. D.), The theory of languages, *Math. Syst. Theory*, **2**:2, 97—125. (Русский перевод: Ахоу А., Улман Дж., Теория языков, Кибернетический сборник, новая серия, вып. 6, изд-во „Мир", М., 1969, стр. 145—183.)

Ахо, Ульман [1969а] (Aho A. V., Ullman J. D.), Syntax directed translations and the pushdown assembler, *J. Comput. Syst. Sci.*, **3**:1, 37—56.

---

[1]) Кружочком ° отмечена литература, добавленная при переводе тома 2. — *Прим. перев.*

Ахо, Ульман [1969б] (Aho A. V., Ullman J. D.), Properties of syntax directed translations, *J. Comput. Syst. Sci.*, **3**:3, 319—334.

Ахо, Ульман [1971а] (Aho A. V., Ullman J. D.), The care and feeding of LR(*k*) grammars, Proc. of 3rd Annual ACM Symposium on Theory of Computing, 159—170.

Ахо, Ульман [1971б] (Aho A. V., Ullman J. D.), Translations on a context-free grammar, *Inform. and Control*, **19**:5, 439—475.

Ахо, Ульман [1972а] (Aho A. V., Ullman J. D.), Linear precedence functions for weak precedence grammars, *Intern. J. Comput. Math.*, в печати.

Ахо, Ульман [1972б] (Aho A. V., Ullman J. D.), Error detection in precedence parsers, *Math. Syst. Theory*, в печати.

Ахо, Ульман [1972в] (Aho A. V., Ullman J. D.), Optimization of LR(*k*) parsers, *J. Comput. Syst. Sci.*, в печати.

Ахо, Ульман [1972г] (Aho A. V., Ullman J. D.), A technique for speeding up LR(*k*) parsers, Proc. Fourth Annual ACM Symposium on Theory of Computing, pp. 251—263.

Ахо, Ульман [1972д] (Aho A. V., Ullman J. D.), Optimization of straight line code, *SIAM J. on Computing*, **1**:1, 1—19.

Ахо, Ульман [1972е] (Aho A. V., Ullman J. D.), Equivalence of programs with structured variables, *J. Comput. Syst. Sci.*, **6**:2, 125—137.

Ахо, Ульман [1972ж] (Aho A. V., Ullman J. D.), LR(*k*) syntax directed translation, неопубликованная рукопись, Bell Laboratories, Murray Hill, N. J.

Ахо и др. [1968] (Aho A. V., Hopcroft J. E., Ullman J. D.), Time and tape complexity of pushdown automaton languages, *Inform. and Control*, **13**:3, 186—206. (Русский перевод: Ахо А., Хопкрофт Дж., Ульман Дж., Временная и ленточная сложности языков, допускаемых магазинными автоматами, в сб. „Языки и автоматы", изд-во „Мир", М., 1975, стр. 181—197.)

Ахо и др. [1972а] (Aho A. V., Denning P. J., Ullman J. D.), Weak and mixed strategy precedence parsing, *J. ACM*, **19**:2, 225—243.

Ахо и др. [1972б] (Aho A. V., Hopcroft J. E., Ullman J. D.), On finding lowest common ancestors in trees, неопубликованная рукопись, Bell Laboratories, Murray Hill, N. J.

Ахо и др. [1972в] (Aho A. V., Sethi R., Ullman J. D.), Code optimization and finite Church-Rosser systems, в сб. Design and Optimization of Compilers, под ред. Rustin R., Prentice-Hall, Englewood Cliffs, N. J., pp. 89—106.

Ахо и др. [1975] (Aho A. V., Johnson S. C., Ullman J. D.), Deterministic parsing of ambiguous grammars, *Comm. ACM*, **18**:8, 441—453.

Багвелл [1970] (Bagwell J. T.), Local optimizations, *ACM SIGPLAN Notices*, 5:7, 52—66.

Барнет, Футрель [1962] (Barnett M. P., Futrelle R. P.), Syntactic analysis by digital computer, *Comm. ACM*, **5**:10, 515—526.

Бар-Хиллел [1964] (Bar-Hillel Y.), Language and information, Addison-Wesley, Reading, Mass.

Бар-Хиллел и др. [1961] (Bar-Hillel Y., Perles M., Shamir E.), On formal properties of simple phrase structure grammars. *Z. Phonetik, Sprachwissenschaft und Kommunikationsforschung*, **14**, 143—172. См. также Бар-Хиллел [1964], pp. 116—150.

Бауэр и др. [1968] (Bauer H., Becker S., Graham S.), ALGOL W Implementation, CS98, Computer Science Dept. Stanford University, Stanford, Calif.

*Бежанова М. М., Поттосин И. В. [1965], Программирование циклов и индексных выражений в Альфа-трансляторе, в сб. „Альфа-система автоматизации программирования" под ред. А. П. Ершова; ВЦ СО АН СССР, Новосибирск (2-е издание, „Наука", М., 1967).

Белл [1969] (Bell J. R.), A new method for determining linear precedence functions for precedence grammars, *Comm. ACM*, **12**:10, 316—333.

Белл [1970] (Bell J. R.), The quadratic quotient method: a hash code eliminating secondary clustering, *Comm. ACM*, **13**:2, 107—109.

Берж [1958] (Berge C.), The theory of graphs and its applications, Wiley, New York. (Русский перевод: Берж К., Теория графов и ее применение, ИЛ, М., 1962.)

Бжозовский [1962] (Brzozowski J. A.), A survey of regular expressions and their applications, *IRE Trans. on Electr. Comput.*, **11**:3, 324—335

Бжозовский [1964] (Brzozowski J. A.), Derivatives of regular expressions, *J. ACM*, **11**:4, 481—494.

Биледи [1966] (Belady L. A.), A study of replacement algorithms for a virtual storage computer, *IBM Systems J.*, **5**, 78—82.

Билз [1969] (Beals A. J.), The generation of a deterministic parsing algorithm, Report № 304, Department of Computer Science, University of Illinois, Urbana.

Билз и др. [1969] (Beals A. J., LaFrance J. E., Northcote R. S.), The automatic generation of Floyd production syntactic analyzers, Report № 350, Department of Computer Science, University of Illinois, Urbana.

Бити [1972] (Beatty J. C.), An axiomatic approach to code optimization for expressions, *J. ACM*, **19**:4.

Бирман, Ульман [1970] (Birman A., Ullman J. D.), Parsing algorithms with backtrack, IEEE Conf. Record of 11th Annual Symposium on Switching and Automata Theory, 153—174. (Расширенный вариант этой работы см. в *Inform. and Control*, **23**:1 (1973), 1—34.)

Блатнер [1972] (Blattner M.), The unsolvability of the equality problem for sentential forms of context-free languages, неопубликованное сообщение, UCLA, Los Angeles, Calif.

Бобров [1963] (Bobrow D. G.), Syntactic analysis of English by computer — a survey, *Proc. AFIPS Fall Joint Computer Conference*, **24**, 365—387.

Бородин [1970] (Borodin A.), Computational complexity — a survey, Proc. 4th Annual Princeton Conference on Information Sciences and Systems, 257—262. См. также Ахо [1973].

*Бошман [1976] (Bochmann G. V.), Semantic evaluation from left to right, *Comm. ACM*, **19**:2, 55—62.

*Братчиков И. Л. [1975], Синтаксис языков программирования, изд-во „Наука", М.

Браффорт, Хиршберг [1963] (Braffort P., Hirschberg D. (eds.)), Computer programming and formal systems, North-Holland, Amsterdam.

Браха [1972] (Bracha N.), Transformations on loop-free program schemata, Report № UIUCDCS-R-72-516, Department of Computer Science, University of Illinois, Urbana.

Брукер, Моррис [1963] (Brooker R. A., Morris D.), The compiler-compiler, *Ann Rev. Autom. Program.*, **3**, 229—275.

Бруно, Беркхард [1970] (Bruno J. L., Burkhard W. A.), A circularity test for interpreted grammars. Technical Report 88. Computer Sciences Laboratory, Department of Electrical Engineering, Princeton University, Princeton, N. J.

Брюер [1969] (Breuer M. A.), Generation of optimal code for expressions via factorization, *Comm. ACM*, **12**:6, 333—340.

Бузам, Энглунд [1969] (Busam V. A., Englund D. E.), Optimization of expressions in Fortran, *Comm. ACM*, **12**:12, 666—674.

Бук [1970] (Book R. V.), Problems in formal language theory, Proc. 4th Annual Princeton Conference on Information Sciences and Systems, 253—256. См. также Ахо [1973].

Бут [1967] (Booth T. L.), Sequential machines and automata theory, Wiley, New York.

Бэкус и др. [1957] (Backus J. W. et al.), The FORTRAN automatic coding system, *Proc. Western Joint Computer Conference*, **11**, 188—198.

Бэр, Бове [1968] (Baer J. L., Bovet D. P.), Compilation of arithmetic expressions for parallel computations, Proc. IFIP Congress 68, B4—B10.

Вайз [1971] (Wise D. S.), Domölki's algorithm applied to generalized overlap resolvable grammars, Proc. 3d Annual Symposium on Theory of Computing, 171—184.

Ван Вейнгаарден [1969] (Van Wijngaarden A. (ed.)), Report on the algorithmic language ALGOL 68, *Numer. Math.*, **14**, 79—218. (Русский перевод: Алгоритмический язык АЛГОЛ 68, *Кибернетика*, **6** (1969); **1** (1970).)

°Ван Вейнгаарден и др. [1975] (Van Wijngaarden et al. (eds.)), Revised report on the algorithmic language ALGOL 68, *Acta Informatica*, **5**, 1—236. (Русский перевод готовится к печати в изд-ве „Мир", М.)

Вегбрейт [1970] (Wegbreit B.), Studies in extensible programming languages, Ph. D. Thesis, Harvard Univ., Cambridge, Mass.

°Великанова Т. М., Ершов А. П., Ким К. В., Курочкин В. М., Олейник-Овод Ю. Н., Поддерюгин В. Д. [1961], Программируемая программа для машины (ППС), Тр. Всесоюзн. совещ. по вычисл. математике и применению средств вычисл. техники, Изд-во АН АзССР, Баку.

Виноград [1965] (Winograd S.), On the time required to perform addition, *J. ACM*, **12**:2, 277—285. (Русский перевод: Виноград С., О времени, требующемся для выполнения сложения, Кибернетический сборник, новая серия, вып. 6, изд-во „Мир", М., 1969, стр. 41—54.)

Виноград [1967] (Winograd S.), On the time required to perform multiplication, *J. ACM*, **14**:4, 793—802. (Русский перевод: Виноград С., О времени, требующемся для выполнения умножения, Кибернетический сборник, новая серия, вып. 6, изд-во „Мир", М., 1969, стр. 55—71.)

Вирт [1965] (Wirth N.), Algorithm 265: Find precedence functions, *Comm. ACM*, **8**:10, 604—605.

Вирт [1968] (Wirth N.), PL 360 — a programming language for the 360 computers, *J. ACM*, **15**:1, 37—54.

Вирт, Вебер [1966] (Wirth N., Weber H.), EULER — a generalization of ALGOL and its formal definition, Parts 1 and 2, *Comm. ACM*, **9**:1, 13—23; **9**:2, 89—99.

Возенкрафт, Эванс [1969] (Wozencraft J. M., Evans A., Jr.), Notes on programming languages, Dept. of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Mass.

Вуд [1969a] (Wood D.), The theory of left factored languages, *Comput. J.*, **12**:4, 349—356; **13**:1, 55—62.

Вуд [1969б] (Wood D.), A note on top on top-down deterministic languages, *BIT*, **9**:4, 387—399.

Вуд [1970] (Wood D.), Bibliography 23: Formal language theory and automata theory, *Comput. Rev.*, **11**:7, 417—430.

Галлер, Перлис [1967] (Galler B. A., Perlis A. J.), A proposal for definitions in ALGOL, *Comm. ACM*, **10**:4, 204—219.

Гарвик [1964] (Garwick J. V.), GARGOYLE, a language for compiler writing, *Comm. ACM*, **7**:1, 16—20.

Гарвик [1968] (Garwick J. V.), GPL, a truly general purpose language, *Comm. ACM*, **11**:9, 634—638.

Гилл [1962] (Gill A.), Introduction to the theory of finite state machines, McGraw-Hill, New York. (Русский перевод: Гилл А., Введение в теорию конечных автоматов, изд-во „Наука", М., 1966.)

Гинзбург [1968] (Ginzburg A.), Algebraic theory of automata, Academic Press, New York.

Гинзбург [1962] (Ginsburg S.), An Introduction to mathematical machine theory, Addison-Wesley, Reading, Mass.

Гинзбург [1966] (Ginsburg S.), The mathematical theory of context-free languages, McGraw-Hill, New York. (Русский перевод: Гинзбург С., Математическая теория контекстно-свободных языков, изд-во „Мир", М., 1970.)

Гинзбург, Грейбах [1966] (Ginsburg S., Greibach S. A.), Deterministic context-free languages, *Inform. and Control*, **9**:6, 620—648.

Гинзбург, Грейбах [1969] (Ginsburg S., Greibach S. A.), Abstract families of languages, Memoir Amer. Math. Soc., 87, 1—32. (Русский перевод: Гинзбург С., Грейбах Ш., Абстрактные семейства языков, в сб. „Языки и автоматы", изд-во „Мир", М., 1975, стр. 233—281.)

Гинзбург, Райс [1962] (Ginsburg S., Rice H. G.), Two families of languages related to ALGOL, *J. ACM*, **9**:3, 350—371. (Русский перевод: Гинзбург С., Райс Х., Два класса языков типа АЛГОЛ, Кибернетический сборник, новая серия, вып. 6, изд-во „Мир", М., 1969, стр. 184—216.)

Гир [1965] (Gear C. W.), High speed compilation of efficient object code, *Comm. ACM*, **8**:8, 483—487.

Глении [1960] (Glennie A.), On the syntax machine and the construction of a universal compiler, Technical Report № 2, Computation Center, Carnegie-Mellon University, Pittsburg, Pa.

Грау и др. [1967] (Grau A. A., Hill U., Langmaack H.), Translation of ALGOL 60, Springer, New York.

Грейбах [1965] (Greibach S. A.), A new normal form theorem for context-free phrase structure grammars, *J. ACM*, **12**:1, 42—52.

Грейбах, Хопкрофт [1969] (Greibach S. A., Hopcroft J.), Scattered context grammars, *J. Comput. Syst. Sci.*, **3**:3, 233—247. (Русский перевод: Грейбах Ш., Хопкрофт Дж., Грамматики с рассеянным контекстом, в сб. „Языки и автоматы", изд-во „Мир", М., 1975, стр. 166—184.)

Грис [1971] (Gries D.), Compiler construction for digital computers, Wiley, New York. (Русский перевод: Грис Д., Конструирование компиляторов для цифровых вычислительных машин, изд-во „Мир", М., 1975.)

Гриswold [1971] (Griswold R. E., Poage J. F., Polonsky I. P.), The SNOBOL 4 programming language (2nd ed.), Prentice-Hall, Englewood Cliffs, N. J.

Гриффитс [1968] (Griffiths T. V.), The unsolvability of the equivalence problem for Λ-free nondeterministic generalized machines, *J. ACM*, **15**:3, 409—413. (Русский перевод: Гриффитс Т. В., Неразрешимость проблемы эквивалентности для Λ-свободных обобщенных машин, Кибернетический сборник, новая серия, вып. 8, изд-во „Мир", М., 1971.)

Гриффитс, Петрик [1965] (Griffiths T. V., Petrick S. R.), On the relative efficiencies of context-free grammar recognizers, *Comm. ACM*, **8**:5, 289—300.

Гросс, Лантен [1970] (Gross M., Lentin A.), Introduction to formal grammars, Springer, New York. (Русский перевод: Гросс М., Лантен А., Теория формальных грамматик, изд-во „Мир", М., 1971.)

Грэй [1969] (Gray J. N.), Precedence parsers for programming languages, Ph. D. Thesis, Univ. of California, Berkeley.

Грэй, Харрисон [1969] (Gray J. N., Harrison M. A.), Single pass precedence analysis, IEEE Conf. Record of 10th Annual Symposium on Switching and Automata Theory, 106—117.

Грэй и др. [1967] (Gray J. N., Harrison M. A., Ibarra O.), Two way pushdown automata, *Inform. and Control*, **11**:1, 30—70.

Грэхем [1972] (Graham R. L.), Bounds on multiprocessing anomalies and related packing algorithms, *Proc. AFIPS Spring Joint Computer Conference*, **40**, AFIPS Press, Montvale, N. J., 205—217.

Грэхем [1964] (Graham R. M.), Bounded context translation, *Proc. AFIPS Spring Joint Computer Conference*, **25**, 17—29.

Грэхем [1970] (Graham S. L.), Extended precedence languages, bounded right context languages and deterministic languages, IEEE Conf. Record of 11th Annual Symposium on Switching and Automata Theory, 175—180.

Де Бэккер [1971] (De Bakker J. W.), Axiom systems for simple assignment statements, см. Энгелер [1971, стр. 1—22].

Де Ремер [1968] (DeRemer F. L.), On the generation of parsers for BNF grammars: an algorithm, Report № 276, Dept. of Computer Science, University of Illinois, Urbana.

Де Ремер [1969] (DeRemer F. L.), Practical translators for LR(*k*) languages, Ph. D. Thesis, Massachusetts Institute of Technology, Cambridge, Mass.

Де Ремер [1971] (DeRemer F. L.), Simple LR(*k*) grammars, *Comm. ACM*, **14**:7, 453—460.

Джентльмен [1971] (Gentleman W. M.), A portable coroutine system, Information processing—71 (IFIP Congress), TA—3, 94—98.

Джонсон и др. [1968] (Johnson W. L., Porter J. H., Ackley S. I., Ross D. T.), Automatic generation of efficient lexical processors using finite state techniques, *Comm. ACM*, **11**:12, 805—813.

Дьюар и др. [1969] (Dewar R. B. K., Hochsprung R. R., Worley W. S.), The IITRAN programming language, *Comm. ACM*, **12**:10, 569—575.

Дэвис [1958] (Davis M.), Computability and unsolvability, McGraw-Hill, New York.

Дэвис [1965] (Davis M. (ed.)), The undecidable, Basic papers in undecidable propositions, unsolvable problems and computable functions, Raven Press, New York.

°Ершов А. П. [1958а], Программируемая программа для быстродействующей электронной счетной машины, Изд-во АН СССР, М.

°Ершов А. П. [1958б], О программировании арифметических операторов, *ДАН СССР*, **118**: 3.

°Ершов А. П. [1962], Сведение задачи распределения памяти при составлении программ к задаче раскраски вершин графов, *ДАН СССР*, **142**: 4.

Ершов А. П. [1966], ALPHA—an automatic programming system of high efficiency, *J. ACM*, **13**:1, 17—24.

°Ершов А. П., Змиевская Л. Л., Мишкович Р. Д., Трохан Л. К. [1965], Экономия и распределение памяти в Альфа-трансляторе, в сб. „Альфа-система автоматизации программирования", ВЦ СОАН СССР, Новосибирск.

°Ершов А. П., Курочкин В. М. [1961], О некоторых проблемах автоматического программирования, Тр. Всесоюзн. совещ. по вычисл. матем. и применению средств вычисл. техники, Изд-во АН АзССР, Баку.

°Ершова Н. М., Мостинская С. В., Рудиева Т. Л. [1961], Арифметический блок, в сб. „Система автоматизации программирования" под ред. Н. П. Трифонова и М. Р. Шура-Буры, изд-во „Наука", М.

Замельзон, Бауэр [1960] (Samelson K., Bauer F. L.), Sequential formula translation, *Comm. ACM*, **3**:2, 76—83.

ИБМ [1969] (IBM), System 360 Operating System PL/I(F) Compiler Program Logic Manual, Publ. № Y286800, IBM, Hursley, Winchester, England.

Игараси [1968] (Igarashi S.), On the equivalence of programs represented by Algol-like statements, Report of the Computer Centre, University of Tokyo, **1**, 103—118.

Ингерман [1966] (Ingerman P. Z.), A syntax oriented translator, Academic Press, New York. (Русский перевод: Ингерман П., Синтаксически ориентированный транслятор, изд-во „Мир", М., 1969.)

Ихбиа, Морэс [1970] (Ichbiah J. D., Morse S. P.), A technique for generating almost optimal Floyd-Evans productions for precedence grammars, *Comm. ACM*, **13**:8, 501—508.

Кавинесс [1970] (Caviness B. F.), On canonical forms and simplification, *J. ACM*, **17**:2, 385—396.

Камеда, Вайнер [1968] (Kameda T., Weiner P.), On the reduction of nondeterministic automata, Proc. 2nd Annual Princeton Conference on Information Sciences and Systems, 348—352.

Кантор [1962] (Cantor D. G.), On the ambiguity problem of Backus systems, *J. ACM*, **9**:4, 477—479.

°Камынин С. С., Любимский Э. З., Шура-Бура М. Р. [1958], Об автоматизации программирования, *Проблемы кибернетики*, вып. 1.

Каплан [1970] (Kaplan D. M.), Proving things about programs, Proc. 4th Annual Princeton Conference on Information Sciences and Systems, 244—251.

Касами [1965] (Kasami T.), An efficient recognition and syntax analysis algorithm for context-free languages, Sci. Rep. AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass.

Касами, Торин [1969] (Kasami T., Torii K.), A syntax analysis procedure for unambiguous context-free grammars, *J. ACM*, **16**:3, 423—431.

°Касьянов В. Н. [1975], Выделение гамаков в ориентированном графе, *ДАН СССР*, **221**:5, 1020—1022.

°Касьянов В. Н., Трахтенброт М. Б. [1975], Анализ структур программ в глобальной оптимизации, Труды Всесоюзн. симп. по методам реализации новых алгоритм. языков, Новосибирск.

Кеннеди [1971] (Kennedy K.), A global flow analysis algorithm, *Intern. J. Comput. Math.*, **3**:1, 5—16.

Кеннеди [1972] (Kennedy K.), Index register allocation in straight line code and simple loops, в сб. „Design and Optimization of Compilers" под ред. Rustin R., Prentice-Hall, Englewood Cliffs, N. J., 51—64.

°Китов А. И., Криницкий Н. А. [1959], Электронные цифровые машины и программирование, Физматгиз, М.

Кларк [1967] (Clark E. R.), On the automatic simplification of source language programs, *Comm. ACM*, **10**:3, 160—164.

Клини [1952] (Kleene S. C.), Introduction to metamathematics, Van Nostrand Reinhold, New York. (Русский перевод: Клини С. К., Введение в метаматематику, ИЛ, М., 1957.)

Клини [1956] (Kleene S. C.), Representation of events in nerve nets, в сб. „Automata Studies" под ред. Shannon C. E., McCarthy J., Princeton University Press, Princeton, N. J. (Русский перевод: Клини С. К., Представление событий в нервных сетях, в сб. „Автоматы", ИЛ, М., 1956, стр. 15—67.)

Кнут [1965] (Knuth D. E.), On the translation of languages from left to right, *Inform. Control*, **8**:6, 607—639. (Русский перевод: Кнут Д. Е., О переводе (трансляции) языков слева направо, в сб. „Языки и автоматы", изд-во „Мир", М., 1975, стр. 9—42.)

Кнут [1967] (Knuth D. E.), Top-down syntax analysis, Lecture Notes, International Summer School on Computer Programming, Copenhagen. См. также *Acta Informatica*, **1**:2 (1971), 79—110.

Кнут [1968а] (Knuth D. E.), The art of computer programming, Vol. 1: Fundamental algorithms, Addison-Wesley, Reading, Mass. (Русский перевод: Кнут Д., Искусство программирования для ЭВМ, т. 1, изд-во „Мир", М., 1975.)

Кнут [1968б] (Knuth D. E.), Semantics of context-free languages, *Math. Syst. Theory*, **2**:2, 127—146. См. также *Math. Syst. Theory*, **5**:1, 95—96.

Кнут [1971] (Knuth D. E.), An empirical study of FORTRAN programs, *Software—Practice and Experience*, **1**:2, 105—134.

Кнут [1973] (Knuth D. E.), The art of computer programming, Vol. 3: Sorting and searching, Addison-Wesley, Reading, Mass. (Русский перевод готовится к печати в изд-ве „Мир", М.)

Кок [1970] (Cocke J.), Global common subexpression elimination, *ACM SIGPLAN Notices*, **5**:7, 20—24.

Кок, Шварц [1970] (Cocke J., Schwartz J. T.), Programming languages and their compilers, Courant Institute of Mathematical Sciences, New York University, Nem York.

Колмерауэр [1970] (Colmerauer A.), Total precedence relations, *J. ACM*, **17**:1, 14—30.

Конвэй [1963] (Conway M. E.), Design of a separable transition-daigram compiler, *Comm. ACM*, **6**:7, 396—408. (Русский перевод: Конвей М. Е., Проект делимого компилятора, основанного на диаграммах перехода, в сб. „Современное программирование", изд-во „Сов. радио", М., 1967, стр. 206—246.)

Конвэй, Максвелл [1963] (Conway R. W., Maxwell W. L.), CORC: The Cornell computing language, *Comm. ACM*, **6**:6, 317—321.
Конвэй, Максвелл [1968] (Conway R. W., Maxwell W. L.), CUPL—an approach to introductory computing instruction, TR № 68—4, Dept. of Computer Science, Cornell Univ., Ithaca, N. Y.
Конвэй и др. [1970] (Conway R. W. et al.), PL/C. A high performance subset of PL/I, TR 70—55, Dept. of Computer Science, Cornell Univ., Ithaca, N. Y.
Кореньяк [1969] (Korenjak A. J.), A practical method for constructing LR (*k*) processors. *Comm. ACM*, **12**:11, 613—623.
Кореньяк, Хопкрофт [1966] (Korenjak A. J., Hopcroft J. E.), Simple deterministic languages, IEEE Conf. Record of 7th Annual Symposium on Switching and Automata Theory, 36—46. (Русский перевод: Кореньяк А., Хопкрофт Дж., Простые детерминированные языки, в сб. „Языки и автоматы", изд-во „Мир", М., 1975, стр. 71—96.)
Косараю [1970] (Kosaraju S. R.), Finite state automata with markers, Proc. 4th Annual Princeton Conference on Information Sciences and Systems, 380.
Коэн, Готлиб [1970] (Cohen D. J., Gotlieb C. C.), A list structure form of grammars for syntactic analysis, *Comput. Surveys*, **2**:1, 65—82.
Коэн, Чулик [1971] (Cohen R. S., Culik K., II), LR-regular grammars—an extension of LR(*k*) grammars, IEEE Conf. Record of 12th Annual Symposium on Switching and Automata Theory, 153—165.
Кристенсен, Шоу [1969] (Christensen C., Shaw J. C. (eds)), Proc. of the extensible languages symposium, *ACM SIGPLAN Notices*, **4**:8.
Кук [1971] (Cook S. A.), Linear time simulation of deterministic two-way pushdown automata, Information Processing—71 (IFIP Congress), TA—2, 174—179.
Кук, Аандераа [1969] (Cook S. A., Aanderaa S. D.), On the minimum computation time of functions, Trans. Amer. Math. Soc. 142, 291—314. (Русский перевод: Кук С. А. Аандераа С. Д., О минимальном времени вычисления функций, Кибернетический сборник, новая серия, вып. 8, изд-во „Мир", М., 1971, стр. 168—200.)
Куно, Эттингер [1962] (Kuno S., Oettinger A. G.), Multiple-path syntactic analyzer, Information Processing—62 (IFIP Congress), 306—311.
Курки-Суонио [1969] (Kurki-Suonio R.), Notes on top-down languages, *BIT*, **9**, 225—238.
*Курочкин В. М. [1975а], Обобщенная схема синтаксически управляемой трансляции, Труды Всесоюзного симпозиума по методам реализации новых алгоритмических языков, ч. I. Новосибирск.
*Курочкин В. М. [1975б], Об одном алгоритме преобразования грамматик, Труды Всесоюзного симпозиума по методам реализации новых алгоритмических языков, ч. I, Новосибирск.
*Курочкин В. М., Столяров Л. Н., Сушков Б. Г., Флеров Ю. А. [1973], Теория и реализация языков программирования, МФТИ, М.
*Лавров С. С. [1961], Об экономии памяти в замкнутых операторных схемах, *Журнал вычисл. матем. и матем. физики*, **1**:4.
*Лавров С. С., Гончарова Л. И. [1971], Автоматическая обработка данных. Хранение информации в памяти ЭВМ, изд-во „Наука", М.
Лакхэм и др. [1970] (Luckham D. C., Park D. M. R., Paterson M. S.), On formalized computer programs, *J. Comput. Syst. Sci.*, **4**:3, 220—249.
Лалонд и др. [1971] (LaLonde W. R., Lee E. S., Horning J. J.), An LALR (*k*) parser generator, Information Processing—71. (IFIP Congress), TA—3, 153—157.
Лафранс [1970] (LaFrance J.), Optimization of error recovery in syntax directed parsing algorithms, *ACM SIGPLAN Notices*, **5**:12, 2—17.
Ледли, Уилсон [1960] (Ledley R. S., Wilson J. B.), Automatic programming language translation through syntactical analysis, *Comm. ACM*, **3**, 213—214.

Лейниус [1970] (Leinius R. P.), Error detection and recovery for syntax directed compiler systems. Ph. D. Thesis, Univ. of Wisconsin, Madison.
Ли [1967] (Lee J. A. N.), Anatomy of a compiler, Van Nostrand Reinhold, New York.
Ливенворт [1966] (Leavenworth B. M.), Syntax macros and extended translation, *Comm. ACM*, **9**:11, 790—793.
Локс [1970] (Loeckx J.), An algorithm for the construction of bounded-context parsers, *Comm. ACM*, **13**:5, 297—307.
Лоури, Медлок [1969] (Lowry E. S., Medlock C. W.), Object code optimization, *Comm. ACM*, **12**:1, 13—22.
Лукаш, Вальк [1969] (Lucas P., Walk K.), On the formal description of PL/I, *Ann. Rev. Autom. Program.*, **6**:3, 105—182.
Льюис, Розенкранц [1971] (Lewis P. M., II, Rosenkrantz D. J.), An ALGOL compiler designed using automata theory, Proc. Polytechnic Institute of Brooklyn Symposium on Computers and Automata, 75—88.
Льюис, Стирнз [1968] (Lewis P. M., II, Stearns R. E.), Syntax directed transduction, *J. ACM*, **15**:3, 464—488.
*Льюис и др. [1974] (Lewis P M., II, Rosenkrantz D. J., Stearns R. E.), Attributed translations, *J. Comput. Syst. Sci.*, **9**:3, 279—307.
Мак-Илрой [1960] (McIlroy M. D.), Macro instruction extentions of compiler languages, *Comm. ACM*, **3**:4, 414—220.
Мак-Илрой [1968] (McIlroy M. D.), Coroutines, неопубликованная рукопись, Bell Laboratories, Murray Hill, N. J.
Мак-Илрой [1972] (McIlroy M. D.), A manual for the TMG compiler writing language, неопубликованное сообщение, Bell Laboratories, Murray Hill, N. J.
Мак-Каллок, Питс [1943] (McCulloch W. S., Pitts E.), A logical calculus of the ideas immanent in nervous activity, *Bull. Math. Biophys.*, **5**, 115—133. (Русский перевод: Маккалок У. С., Питтс Э., Логическое исчисление идей, относящихся к нервной активности, в сб. „Автоматы", ИЛ, М., 1956, стр. 362—384.)
Мак-Карти [1963] (McCarthy J.), A basis for the mathematical theory of computation, в сб. Computer Programming and Formal Systems, под ред. Braffort P., Hirschberg D., 33—71.
Мак-Карти, Пейнтер [1967] (McCarthy J., Painter J. A.), Correctness of a compiler for arithmetic expressions, в сб. под ред. Шварца [1967], 33—41.
Мак-Киман [1965] (McKeeman W. M.), Peephole optimization, *Comm. ACM*, **8**:7, 443—444.
Мак-Киман [1976] (McKeeman W. M.), An approach to computer language design, CS48, Computer Science Department, Stanford Univ., Stanford, Calif.
Мак-Киман и др. [1970] (McKeeman W. M., Horning J. J., Wortman D. B.), A compiler generator, Prentice-Hall, Inc., Englewood Cliffs, N. J.
Мак-Клюр [1965] (McClure R. M.), TMG—a syntax directed compiler, *Proc. ACM National Conference*, **20**, 262—274.
Мак-Нотон [1967] (McNaughton R.), Parenthesis grammars, *J. ACM*, **14**:3, 490—500.
Мак-Нотон, Ямада [1960] (McNaughton R., Yamada H.), Regular expressions and state graphs for automata, *IRE Trans. on Electr. Comput.*, **9**:1, 39—47.
Манна [1973] (Manna Z.), Program schemas, в сб. Ахо [1973].
Марилл [1962] (Marill M.), Computational chains and the size of computer programs, *IRE Trans. on Electr. Comput.*, **EC-11**:2, 173—180.
Марков А. А. [1951], Теория алгорифмов, *Труды Математического института им. В. А. Стеклова*, **38**.
Мартин [1972] (Martin D. F.), A Boolean matrix method for the computation of linear precedence functions, *Comm. ACM*, **15**:6, 448—454.

Маурер [1968] (Maurer W. D.), An improved hash code for scatter storage, *Comm. ACM*, **11**:1, 35—38.
Мейерс [1965] (Meyers W. J.), Optimization of computer code, неопубликованное сообщение, G. E. Research Center, Schenectady, N. Y.
Мендельсон [1968] (Mendelson E.), Introduction to mathematical logic, Van Nostrand Reinhold, New York. (Русский перевод: Мендельсон Э., Введение в математическую логику, изд-во „Наука", М., 1971.)
Миллер, Шоу [1968] (Miller W. F., Shaw A. C.), Linguistic methods in picture processing—a survey, *Proc. AFIPS Fall Joint Computer Conference*, **33**, 279—290.
Минский [1967] (Minsky M.), Computation: finite and infinite machines, Prentice-Hall, Inc., Englewood Cliffs, N. J. (Русский перевод: Минский М., Вычисления и автоматы, изд-во „Мир", М., 1971.)
Монтанари [1970] (Montanari U. G.), Separable graphs, planar graphs, and web grammars, *Inform. and Control*, **16**:3, 243—267.
Морган [1970] (Morgan H. L.), Spelling correction in systems programs, *Comm. ACM*, **13**:2, 90—93.
Моррис [1968] (Morris R.), Scatter storage techniques, *Comm. ACM*, **11**:1, 35—44.
Моултон, Мюллер [1967] (Moulton P. G., Muller M. E.), A compiler emphasizing diagnostics, *Comm. ACM*, **10**:1, 45—52.
Мунро [1971] (Munro I.), Efficient determination of the transitive closure of a directed graph, *Inform. Processing Letters*, **1**:2, 56—58.
Мур [1956] (Moore E. F.), Gedanken experiments on sequential machines, в сб. под ред. Шеннона и Мак-Карти [1956], 129—153. (Русский перевод: Мур Э. Ф., Умозрительные эксперименты с последовательностными машинами, в сб. „Автоматы", ИЛ, М., 1956, стр. 179—210.)
Мур [1964] (Moore E. F.), Sequential machines: Selected Papers, Addison-Wesley, Reading, Mass.
Наката [1967] (Nakata I.), On compiling algorithms for arithmetic expressions, *Comm. ACM*, **12**:2, 81—84.
Наур [1963] (Naur P. (ed.)), Revised report on the algorithmic language ALGOL 60, *Comm. ACM*, **6**:1, 1—17. (Русский перевод: Алгоритмический язык АЛГОЛ 60, изд-во „Мир", М., 1965.)
Нивергельт [1965] (Nievergelt J.), On the automatic simplification of computer programs, *Comm. ACM*, **8**:6, 366—370.
Огден [1968] (Ogden W.), A helpful result for proving inherent ambiguity, *Math. Syst. Theory*, **2**:3, 191—194. (Русский перевод: Огден У., Результат, полезный для доказательства существенной неоднозначности, в сб. „Языки и автоматы", изд-во „Мир", М., 1975, стр. 109—113.)
Оре [1962] (Ore O.), Theory of graphs. *Amer. Math. Soc. Colloq. Publ.*, **38**. (Русский перевод: Оре О., Теория графов, изд-во „Наука", М., 1968.)
Павлидис [1972] (Pavlidis T.), Linear and context-free graph grammars, *J. ACM*, **19**:1, 11—23.
Парик [1966] (Parikh R.J.), On context-free languages, *J. ACM*, **13**:4, 570—581.
Патерсон [1968] (Paterson M. S.), Program schemata, в сб. Machine Intelligence, v. 3, под ред. Michie, Edinburg University Press, Edinburg, 19—31.
Паул, Ангер [1968a] (Paul M. C., Unger S. H.), Structural equivalence of context-free grammars, *J. Comp. Syst. Sci.*, **2**:1, 427—463.
Паул, Ангер [1968б] (Paull M. C., Unger S. H.), Structural equivalence and LL(k) grammars.
Пейджер [1970] (Pager D.), A solution to an open problem by Knuth, *Inform. and Control*, **17**:5, 462—473.
Пейнтер [1970] (Painter J. A.), Effectiveness of an optimizing compiler for arithmetic expressions, *ACM SIGPLAN Notes*, **5**:7, 101—126.
Петерсон [1957] (Peterson W. W.), Adressing for random access storage, *IBM J. Research and Development*, **1**:2, 130—146.

Петроне [1968] (Petrone L.), Syntax directed mapping of context-free languages, IEEE Conference Record of 9th Annual Symposium on Switching and Automata Theory, 160—175.
Пол [1962] (Paul M.), A general processor for certain formal languages, Proc. ICC Symposium of Symb. Lang. Data Processing, 65—74.
Пост [1943] (Post E. L.), Formal reductions of the general combinatorial decision problem, *Amer. J. Math.*, **65**, 197—215.
Пост [1947] (Post E. L.), Recursive unsolvability of a problem of Thue, *J. Symb. Logic*, **12**, 1—11. См. также Дэвис [1965], 292—303.
Пост [1965] (Post E. L.), Absolutely unsolvable problems and relatively undecidable propositions—account of an anticipation, в сб. под ред. Дэвиса [1965], 338—433.
°Поттосин И. В. [1973], Оптимизирующие преобразования и их последовательность, в сб. „Системное программирование", ч. 2 (материалы Всесоюзного симпозиума, март 1973), Новосибирск, стр. 128—137.
°Поттосин И. В. [1975], Глобальная оптимизация: практический подход, Труды Всесоюзного симпоз. по методам реализации новых алгоритм. языков, Новосибирск.
Прайс [1971] (Price C. E.), Table lookup techniques, *ACM Comput. Surveys*, **3**:2, 49—66.
Пратер [1969] (Prather R. E.), Minimal solutions of Paull-Unger problems, *Math. Syst. Theory*, **3**:1, 76—85.
Проссер [1959] (Prosser R. T.), Applications of Boolean matrices to the analysis of flow diagrams, Proc. Eastern J. Computer Conf., 133—138.
Пфальц, Розенфельд [1969] (Pfaltz J. L., Rosenfeld A.), Web grammars, Proc. International Joint Conf. on Artificial Intelligence, Washington, 609—619.
Пэр [1964] (Pair C.), Trees, pushdown stores and compilation, *RFTI—Chiffres*, **7**:3, 199—216.
Рабин [1967] (Rabin M. O.), Mathematical theory of automata, в сб. под ред. Шварца [1967], 173—175.
Рабин, Скотт [1959] (Rabin M. O., Scott D.), Finite automata and their decision problems, *IBM J. Res. Devel.*, **3**, 114—125. См. также Мур [1964], 63—91. (Русский перевод: Рабин М. О., Скотт Д., Конечные автоматы и задачи их разрешения, Кибернетический сборник, вып. **4**, ИЛ, М., 1962, стр. 56—91.)
Радке [1970] (Radke C. E.), The use of quadratic residue search, *Comm. ACM*, **13**:2, 103—109.
Редзиевский [1969] (Redziejowski R. R.), On arithmetic expressions and trees, *Comm. ACM*, **12**:2, 81—84.
Рейнольдс [1965] (Reynolds J. C.), An introduction to the COGENT programming system, *Proc. ACM National Conference*, **20**, 422.
Рейнольдс, Хаскел [1970] (Reynolds J. C., Haskell R.), Grammatical coverings, неопубликованное сообщение, Syracuse Univ.
Ренделл, Рассел [1964] (Randell B., Russell L. J.), ALGOL 60 implementation, Academic Press, New York. (Русский перевод: Ренделл Б., Рассел Л., Реализация АЛГОЛа 60, изд-во „Мир", М., 1967.)
Ричардсон [1968] (Richardson D.), Some unsolvable problems involving elementary functions of a real variable, *J. Symb. Logic*, **33**, 514—520.
Роджерс [1967] (Rogers H., Jr.), Theory of recursive functions and effective computability, McGraw-Hill, New York. (Русский перевод: Роджерс Х., Теория рекурсивных функций и эффективная вычислимость, изд-во „Мир", М., 1972.)
Розен [1967a] (Rosen S, (ed.)), Programming systems and languages, McGraw-Hill, New York.
Розен [1967б] (Rosen S.), A compiler-building system developed by Brooker and Morris, в сб. под ред. Розена [1967a], 306—331.

Розенкранц [1967] (Rosenkrantz D. J.), Matrix equations and normal forms for context-free grammars, *J. ACM*, **14**:3, 501—507.
Розенкранц [1968] (Rosenkrantz D. J.). Programmed grammars and classes of formal languages, *J. ACM*, **16**:1, 107—131. (Русский перевод: Розенкранц Д., Программные грамматики и классы формальных языков, в сб. „Сборник переводов по вопросам информационной теории и практики", ВИНИТИ, № 16, М., 1970, стр. 117—146.)
Розенкранц, Льюис [1970] (Rosenkrantz D. J., Lewis P. M., II), Deterministic left corner parsing, IEEE Conf. Record of 11th Annual Symposium on Switching and Automata Theory, 139—152.
Розенкранц, Стирнз [1970] (Rosenkrantz D. J., Stearns R. E.), Properties of deterministic top-down grammars, *Inform. and Control*, **17**:3, 226—256.
Саломаа [1966] (Salomaa A.), Two complete axiom systems for the algebra of regular events, *J. ACM*, **13**:1, 158—169.
Саломаа [1969a] (Salomaa A.), Theory of automata, Pergamon, Elmsford, N. Y.
Саломаа [1969б] (Salomaa A.), On the index of a context-free grammar and language, *Inform. and Control*, **14**:5, 474—477.
Саммет [1969] (Sammet J. E.), Programming languages: history and fundamentals, Prentice-Hall, Englewood Cliffs, N. J.
Сети [1972] (Sethi R.), Validating register allocations for straight line programs, Ph. D. Thesis, Department of Electrical Engineering, Princeton University.
Сети, Ульман [1970] (Sethi R., Ullman J. D.), The generation of optimal code for arithmetic expressions, *J. ACM*, **17**:4, 715—728.
Скотт, Стрэчи [1971] (Scott D., Strachey C.), Towards a mathematical semantics for computer languages, Proc. Symposium on Computers and Automata, Microwave Research Institute Symposia Series, Vol. 21, Polytechnic Institute of Brooklyn, New York, 19—46.
Стил [1966] (Steel T. B. (ed.)), Formal language description languages for computer programming, North-Holland, Amsterdam.
Стирнз [1967] (Stearns R. E.), A regularity test for pushdown machines, *Inform. and Control*, **11**:3, 323—340. (Русский перевод: Стирнз Р., Проверка регулярности для магазинных автоматов, Кибернетический сборник, новая серия, вып. 8, изд-во „Мир", М., 1971, 117—139.)
Стирнз [1971] (Stearns R. E.), Deterministic top-down parsing, Proc. 5th Annual Princeton Conference on information Sciences and Systems, 182—188.
Стирнз, Льюис [1969] (Stearns R. E., Lewis P. M., II), Property grammars and table machines, *Inform. and Control*, **14**:6, 524—549.
Стирнз, Розенкранц [1969] (Stearns R. E., Rosenkrantz D. J.), Table machine simulation, IEEE Conf. Record of 10th Annual Symposium on Switching and Automata Theory, 118—128.
Стоун [1967] (Stone H. S.), One-pass compilation of arithmetic expressions for a parallel processor, *Comm. ACM*, **10**:4, 220—223.
Суппес [1960] (Suppes P.), Axiomatic set theory, Van Nostrand Reinhold, New York.
Тарьян [1972] (Tarjan R.), Depth first search and linear graph algorithms, *SIAM J. Comput.*, **1**:2, 146—160.
Томпсон [1968] (Thompson K.), Regular expression search algorithm, *Comm. ACM*, **11**:6, 419—422.
Тьюринг [1936] (Turing A. M.), On computable numbers with an application to the Entscheidungsproblem, *Proc. London Math. Soc.*, ser. 2, **42**, 230—265; Corrections, *там же*, **43** (1937), 544—546.
Уилкокс [1971] (Wilcox T. R.), Generating machine code for high-level programming languages, TR 71—103, Dept. of computer Science, Cornell University, Ithaca, N. Y.
Ульман [1972a] (Ullman J. D.), A note on hashing functions, *J. ACM*, **19**:3, 569—575.

Ульман [1972б] (Ullman J. D.), Fast algorithms for the elimination of common subexpressions, TR-106, Dept. of Electrical Engineering, Princeton University, Princeton, N. J.
Уолтерс [1970] (Walters D. A.), Deterministic context-sensitive languages, *Inform. and Control*, **17**:1, 14—61.
Уоршолл [1962] (Warshall S.), A theorem on Boolean matrices, *J. ACM*, **9**:1, 11—12.
Уоршолл, Шапиро [1964] (Warshall S., Shapiro R. M.), A general purpose table driven compiler, *Proc. AFIPS Spring Joint Computer Conference*, **25**, 59—65.
Фельдман [1966] (Feldman J. A.), A formal semantics for computer languages and its application in a compiler-compiler, *Comm. ACM*, **9**:1, 3—9.
Фельдман, Грис [1968] (Feldman J. A., Gries D.), Translator writing systems, *Comm. ACM*, **11**:2, 77—113. (Русский перевод: Фельдман Дж., Грис Д., Системы построения трансляторов, в сб. „Алгоритмы и алгоритмические языки", вып. 5, ВЦ АН СССР, М., 1971.)
Фишер [1968] (Fisher M. J.), Grammars with macro-like productions, IEEE Conf. Record of 9th Annual Symposium on Switching and Automata Theory, 131—142.
Фишер [1969] (Fischer M. J.), Some properties of precedence languages, Proc. of 1st Annual ACM Symposium on Theory of Computing, 181—190.
Фишер [1972] (Fischer M. J.), Efficiency of equivalence algorithms, Memo № 256, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass.
Флойд [1961a] (Floyd R. W.), An algorithm for coding efficient arithmetic operations, *Comm. ACM*, **4**:1, 42—51.
Флойд [1961б] (Floyd R. W.), A descriptive language for symbol manipulation, *J. ACM*, **8**:4, 579—584.
Флойд [1962a] (Floyd R. W.), Algorithm 97: shortest path, *Comm. ACM*, **5**:6, 345.
Флойд [1962б] (Floyd R. W.), On ambiguity in phrase structure languages, *Comm. ACM*, **5**:10, 526—534.
Флойд [1963] (Floyd R. W.), Syntactic analysis and operator precedence, *J. ACM*, **10**:3, 316—333.
Флойд [1964a] (Floyd R. W.), Bounded context syntactic analysis, *Comm. ACM*, **7**:2, 62—67.
Флойд [1964б] (Floyd R. W.), The syntax of programming languages—a survey, *IEEE Trans. Electr. Comput.*, **EC-13**:4, 346—353.
Флойд [1967a] (Floyd R. W.), Assigning meanings to programs, в сб. под ред. Шварца [1967], 19—32.
Флойд [1967б] (Floyd R. W.), Nondeterministic algorithms, *J. ACM*, **14**:4, 636—644.
Фрейли [1970] (Frailey D. J.), Expression optimization using unary complement operators, *ACM SIGPLAN Notices*, 5:7, 67—85.
Фримэн [1964] (Freeman D. N.), Error correction in CORC, the Cornell computing language, *Proc. AFIPS Fall Joint Computer Conference*, **26**, 15—34.
Хакстэбл [1964] (Huxtable D. H. R.), On writing an optimizing translator for ALGOL 60, в сб. „Introduction to System Programming", Academic Press, New York.
Халмош [1960] (Halmos P. R.), Naive set theory, Van Nostrand Reinhold, New York.
Халмош [1963] (Halmos P. R.), Lectures on Boolean algebras, Van Nostrand Reinhold, New York.
Харари [1969] (Harary F.), Graph theory, Addison-Wesley, Reading, Mass. (Русский перевод: Харари Ф., Теория графов, изд-во „Мир", М., 1973.)
Харрисон [1965] (Harrison M. A.), Introduction to switching and automata theory, McGraw-Hill, New York.

ртманис, Хопкрофт [1970] (Hartmanis J., Hopcroft J. E.), An overview of the theory of computational complexity, *J. ACM*, **18**:3, 444—475. (Русский перевод: Хартманис Ю., Хопкрофт Дж., Обзор теории сложности вычислений, Кибернетический сборник, новая серия, вып. 11, изд-во „Мир", М., 1974, стр. 131—176.)

ртманис и др. [1965] (Hartmanis J., Lewis P. M., II, Stearns R. E.), Classifications of computations by time and memory requirements, *Proc. IFIP Congress*, **65**, 31—35.

фмен [1954] (Huffman D. A.), The synthesis of sequential switching circuits, *J. Franklin Inst.*, **257**, 3—4, 161, 190, 275—303.

йнс [1970] (Haines L. H.), Representation theorems for context-sensitive languages, Dept. of Electrical Engineering and Computer Sciences, Univ. of California, Berkeley.

йнс, Шютте [1970] (Haynes H. R., Schutte L. J.), Compilation of optimized syntactic recognizers from Floyd-Evans productions, *ACM SIGPLAN Notices*, **5**:7, 38—51.

йс [1967] (Hays D. G.), Introduction to computational linguistics, American Elsevier, New York.

кст, Робертс [1970] (Hext J. B., Roberts P. S), Syntax analysis by Domolki's algorithm, *Computer J.*, **13**:3, 263—271.

кт, Ульман [1972a] (Hecht M. S., Ullman J. D.), Flow graph reducibility, *SIAM J. on Computing*, **1**:2, 188—202.

кт, Ульман [1972б] (Hecht M. S., Ullman J. D.), неопубликованное сообщение, Dept. of Electrical Engeneering, Princeton University.

ллерман [1966] (Hellerman H.), Parallel processing of algebraic expressions, *IEEE Trans. Elect. Comput.*, **EC-15**:1, 82—91.

мский [1956] (Chomsky N.), Three models for the description of language, *IEEE Trans. Inform. Theory*, **2**:3, 113—124. (Русский перевод: Хомский Н., Три модели описания языка, Кибернетический сборник, вып. 2, ИЛ, М., 1961, стр. 237—266.)

мский [1957] (Chomsky N.), Syntactic structures, Mouton and Co., The Hague. (Русский перевод: Хомский Н., Синтаксические структуры, в сб. „Новое в лингвистике", вып. 11, ИЛ, М., 1962, стр. 412—527.)

мский [1959a] (Chomsky N.), On certain formal properties of grammars, *Inform. and Control*, **2**:2, 137—167. (Русский перевод: Хомский Н., О некоторых формальных свойствах грамматик, Кибернетический сборник, вып. 5, ИЛ, М., 1962, стр. 279—311.)

мский [1959б] (Chomsky N.), A note on phrase structure grammars, *Inform. and Control*, **2**:4, 393—395. (Русский перевод: Хомский Н., Заметка о грамматиках непосредственно составляющих, Кибернетический сборник, вып. 5, ИЛ, М., 1962.)

мский [1962] (Chomsky N.), Context-free grammars and pushdown storage, Quarterly Progress Report, № 65, Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Mass.

мский [1963] (Chomsky N.), Formal properties of grammars, в сб. „Handbook of Mathematical Psychology", 2, под ред. Luce R. D., Bush R. R., Galanter E., Wiley, New York, 323—418. (Русский перевод: Хомский Н., Формальные свойства грамматик, Кибернетический сборник, новая серия, вып. 2, изд-во „Мир", М., 1966, стр. 121—230.)

мский [1965] (Chomsky N.), Aspects of the theory of syntax, M.I.T. Press, Cambridge, Mass. (Русский перевод: Хомский Н., Аспекты теории синтаксиса, Изд-во МГУ, М., 1972.)

мский, Миллер [1958] (Chomsky N., Miller G. A.), Finite state languages, *Inform. and Control*, **1**:2, 91—112. (Русский перевод: Хомский Н., Миллер Дж., Языки с конечным числом состояний, Кибернетический сборник, вып. 4, ИЛ, М., 1962, стр. 233—255.)

мский, Шютценберже [1963] (Chomsky N., Schutzenberger M. P.), The algebraic theory of context-free languages, в сб. под ред. Браффорта и Хиршберга [1963], 118—161. (Русский перевод: Хомский Н., Шютценберже М. П., Алгебраическая теория контекстно-свободных языков, Кибернетический сборник, новая серия, вып. 3, изд-во „Мир", М., 1966, стр. 195—242.)

Хопгуд [1969] (Hopgood F.R.A.), Compiling techniques, American Elsevier, New York. (Русский перевод: Хопгуд Ф., Методы компиляции, изд-во „Мир", 1972.)

Хопкинс [1971] (Hopkins M. E.), An optimizing compiler design. Proc. IFIP Congress 71, TA-3, North-Holland, Amsterdam, 69—73.

Хопкрофт [1971] (Hopcroft J. E.), An $n \log n$ algorithm for minimizing states in a finite automaton, CS71—190, Computer Science Dept., Stanford Univ., Stanford, Calif. (Русский перевод: Хопкрофт Дж., Алгоритм для минимизации конечного автомата, Кибернетический сборник, новая серия, вып. 11, изд-во „Мир", М., 1974, стр. 177—184.)

Хопкрофт, Ульман [1967] (Hopcroft J. E., Ullman J. D.), An approach to a unified theory of automata, *Bell System Tech. J.*, **46**:8, 1763—1829.

Хопкрофт, Ульман [1969] (Hopcroft J. E., Ullman J. D.), Formal languages and their relation to automata, Addison-Wesley, Reading, Mass.

Хопкрофт, Ульман [1972a] (Hopcroft J. E., Ullman J. D.), Set merginig algorithms, неопубликованное сообщение, Dept. of Computer Science, Cornell University, Ithaca, N.Y.

Хопкрофт, Ульман [1972б] (Hopcroft J. E., Ullman J. D.) An $n \log n$ algorithm to detect reducible graphs, Proc. 6th Annual Princeton Conference on Information Sciences and Systems, 119—122.

Хорвиц и др. [1966] (Horwitz L. P., Karp R. M., Miller R. E., Winograd S.), Index register allocation, *J. ACM*, **13**:1, 43—61.

Чен [1972] (Chen S.), On the Sethi—Ullman algorithm, неопубликованное сообщение, Bell Laboratories, Holmdel, N. J.

Чёрч [1941] (Church A.) The calculi of lambda-conversion, *Ann. Math. Stud.*, **6**.

Чёрч [1956] (Church A.), Introduction to mathematical logic, Princeton University Press, Princeton, N. J. (Русский перевод: Чёрч А., Введение в математическую логику, ИЛ, М., 1961.)

Читэм [1965] (Cheatham T. E.), The TOTE TGS-II translator-generator system, *Proc. IFIP Congress*, **65**, 592—593.

Читэм [1966] (Cheatham T. E.), The introduction of definitional facilities into higher level programming languages, *Proc. AFIPS Fall Joint Computer Conference*, **30**, 623—637.

Читэм [1967] (Cheatham T. E.), The theory and construction of compilers (2nd ed.), Computer Associates, Inc., Wakefield, Mass.

Читэм, Стэндиш [1970] (Cheatham T. E., Standish T.) Optimization aspects of compiler-compilers, *ACM SIGPLAN Notices*, **5**:10, 10—17.

Читэм, Сэттли [1964] (Cheatham T. E., Sattley K.), Syntax directed compiling, *Proc. AFIPS Spring Joint Computer Conference*, **25**, 31—57.

Чулик [1968] (Čulik K., II), Contribution to deterministic top-down analysis of context-free languages, *Kybernetika*, **4**:5, 422—431.

Чулик [1970] (Čulik K., II), n-ary grammars and the description of mapping of languages, *Kybernetika*, **6**, 99—117.

Шварц [1967] (Schwartz J. T. (ed.)), Mathematical aspects of computer science, *Proc. Symp. Appl. Math.*, **19**.

Шеннон, Мак-Карти [1956] (Shannon C. E., McCarthy J. (eds)), Automata studies, Princeton University Press, Princeton, N. J. (Русский перевод: Автоматы (сб. статей), ИЛ, М., 1956.)

Шепердсон [1959] (Shepherdson J. C.), The reduction of two-way automata to one-way automata, *IBM J. Res.*, **3**, 198—200. См. также Мур [1964], 92—97. (Русский перевод: Шепердсон Дж., Сведение двухсторонних авто-

матов к односторонним автоматам, Кибернетический сборник, вып. 4, ИЛ, М., 1962, стр. 92—98.)
Шефер [1973] (Schaefer M.), A mathematical theory of global flow analysis, Prentice-Hall, Englewood Cliffs, N. J., в печати.
Шорре [1964] (Schorre D. V.), META II, a syntax oriented compiler writing language, *Proc. ACM National Conference*, **19**, D1. 3-1—D1. 3-11.
Шоу [1970] (Shaw A. C.), Parsing of graph-representable pictures, *J. ACM*, **17**:3, 453—481.
Штрассен [1969] (Strassen V.), Gaussian elimination is not optimal, *Numer. Math.*, **13**, 354—356. (Русский перевод: Штрассен Ф., Алгоритм Гаусса не оптимален, Кибернетический сборник, новая серия, вып. 7, изд-во „Мир", М., 1970, стр. 67—70.)
Шютценберже [1963] (Schutzenberger M. P.), On context-free languages and pushdown automata, *Inform. and Control*, **6**:3, 246—264.
Эванс [1964] (Evans A., Jr.), An ALGOL 60 compiler, *Ann. Rev. Autom. Program.* **4**, 87—124.
Эви [1963] (Evey R. J.), Applications of pushdown-store machines, *Proc. AFIPS Fall Joint Computer Conference*, **24**, 215—227.
Эйкель и др. [1963] (Eickel J., Paul M., Bauer F.L., Samelson K.), A syntax-controlled generator of formal language processors, *Comm. ACM*, **6**:8, 451—455.
Элсон, Рейк [1970] (Elson M., Rake S. T.), Code-generation technique for large-language compilers, *IBM Systems J.*, **9**:3, 166—188.
Элспас и др. [1971] (Elspas B., Green M. W., Levitt K. N.), Software reliability, *Computer*, **1**, 21—27.
Энгелер [1971] (Engeler E.(ed.)), Symposium on semantics of algorithmic languages, Lecture Notes in Mathematics, Springer, Berlin.
Эрланд, Фишер [1970] (Irland M. I., Fisher P. C.), A bibliography on computational complexity, CSRR 2028, Der. of Applied Analysis and Computer Science, Univ. of Waterloo, Ontario.
Эрли [1966] (Earley J.), Generating a recognizer for a BNF grammar, Computation Center Report, Carnegie-Mellon University, Pitsburgh.
Эрли [1968] (Earley J.), An efficient context-free parsing algorithm, Ph. D. Thesis, Carnegie-Mellon Univ., Pittsburgh, Pa. См. также *Comm. ACM*. **13**:2, (1970), 94—102. (Русский перевод: Эрли Дж., Эффективный алгоритм анализа контекстно-свободных языков, в сб. „Языки и автоматы", изд-во „Мир", М., 1975, стр. 47—70.)
Эттингер [1961] (Oettinger A.), Automatic syntactic analysis and the pushdown store, в сб. Structure of Language and its Mathematical Concepts, Proc. 12th Symposium on Applied Mathematics, American Mathematical Society, Providence, 104—129.
°Языки и автоматы, Сборник переводов, изд-во „Мир", М., 1975.
Янгер [1967] (Younger D. H.), Recognition and parsing of context-free languages in time n^3, *Inform. and Control*, **10**:2, 189—208. (Русский перевод: Янгер Д. Х., Распознавание и анализ контекстно-свободных языков за время n^3, в сб. „Проблемы математической логики", изд-во „Мир", М., 1970, стр. 344—362.)
Янов Ю. И. [1958], О логических схемах алгоритмов, сб. „Проблемы кибернетики", вып. 1, Физматгиз, М., 1958.

УКАЗАТЕЛЬ ЛЕММ, ТЕОРЕМ И АЛГОРИТМОВ К ТОМУ 2

Номер леммы	Стр.	Номер леммы	Стр.	Номер леммы	Стр.	Номер леммы	Стр.
7.1	83	8.7	160	10.2	283	11.6	346
7.2	83	8.8	169	10.3	315	11.7	346
7.3	109	8.9	170	10.4	319	11.8	372
7.4	127	8.10	179	10.5	320	11.9	372
8.1	142	8.11	180	10.6	320	11.10	374
8.2	150	8.12	180	11.1	339	11.11	374
8.3	154	8.13	188	11.2	339	11.12	385
8.4	156	8.14	189	11.3	343	11.13	401
8.5	156	8.15	190	11.4	343	11.14	402
8.6	157	10.1	283	11.5	346	11.15	440

Номер теоремы	Стр.	Номер теоремы	Стр.	Номер теоремы	Стр.	Номер теоремы	Стр.
7.1	13	8.4	153	8.20	187	11.2	339
7.2	21	8.5	156	8.21	189	11.3	343
7.3	39	8.6	157	8.22	192	11.4	344
7.4	40	8.7	161	9.1	208	11.5	346
7.5	58	8.8	161	9.2	210	11.6	372
7.6	61	8.9	165	9.3	214	11.7	375
7.7	71	8.10	168	9.4	220	11.8	379
7.8	84	8.11	171	9.5	224	11.9	387
7.9	95	8.12	172	9.6	230	11.10	403
7.10	110	8.13	173	9.7	243	11.11	410
7.11	111	8.14	174	10.1	284	11.12	425
7.12	125	8.15	176	10.2	286	11.13	430
7.13	129	8.16	176	10.3	321	11.14	439
8.1	142	8.17	180	10.4	322	11.15	441
8.2	146	8.18	182	11.1	337	11.16	442
8.3	149	8.19	185				

Номер алгоритма	Стр.	Номер алгоритма	Стр.	Номер алгоритма	Стр.	Номер алгоритма	Стр.
7.1	11	7.11	102	8.7	190	10.4	314
7.2	33	7.12	104	9.1	218	10.5	317
7.3	38	7.13	106	9.2	220	11.1	366
7.4	40	8.1	147	9.3	228	11.2	367
7.5	51	8.2	151	9.4	240	11.3	377
7.6	56	8.3	155	9.5	242	11.4	383
7.7	60	8.4	172	10.1	272	11.5	402
7.8	69	8.5	177	10.2	275	11.6	426
7.9	82	8.6	188	10.3	309	11.7	436
7.10	94						

ИМЕННОЙ УКАЗАТЕЛЬ К 1 И 2 ТОМАМ [1])

Аандераа (Aanderaa S. D.) 48
Абрахам (Abraham S.) 123
Абэ (Abe N.) 102
Агафонов В. Н. 408
Айронс (Irons E. T.) 96, 268, 351, 510
Аллард (Allard R. W.) *392*
Аллен (Allen F. E.) *422*, *448*
Ангер (Unger S. H.) 351; *163*
Андерсон (Anderson J. P.) *392*
Анисимов А. В. 408
Арбиб (Arbib M. A.) 163
Ахо (Aho A. V.) 123, 220, 283, 408, 409, 452, 480; *29*, *91*, *117*, *138*, *185*, *236*, *266*, *362*, *448*

Барздинь Я. М. 163
Барнет (Barnet M. P.) 268
Бар-Хиллел (Bar-Hillel Y.) 102, 123, 241
Бауэр Ф. (Bauer F. L.) 510
Бауэр Х. (Bauer H.) 480
Бежанова М. М. *423*
Беккер (Becker S.) 480
Белл (Bell J. R.) *29*, *292*
Берж (Berge C.) 68
Беркхард (Berkhard W. A.) *266*
Бжозовский (Brzozowsky J. A.) 147, 163
Биледи (Belady L. A.) *392*
Билз (Beals A. J.) *45*
Бирман (Birman A.) 542
Блатнер (Blattner M.) 241
Бобров (Bobrow D. G.) 102
Бове (Bovet D. P.) *392*
Бородин (Borodin A.) 52
Браха (Bracha N.) *362*
Брукер (Brooker R. A.) 96, 351
Бруно (Bruno J. L.) *266*
Брюер (Breuer M. A.) *362*
Бузам (Busam V. A.) *422*, *448*
Бук (Book R. V.) 123, 241
Бут (Booth T. L.) 163
Бэкус (Backus J. W.) 95; *206*
Бэр (Baer J. L.) *392*

Вайз (Wise D. S.) 510
Вайнер (Weiner P.) 163
Валлант (Valiant L. G.) 372
Вальк (Walk K.) 74
Вебер (Weber H.) 480; *29*
ван Вейнгаарден (van Wijngaarden A.) 75, 559
Вегбрейт (Wegbreit B.) 75
Великанова Т. М. *423*
Вельбицкий И. В. 408, 411
Виноград С. (Winograd S.) 48
Виноград Т. (Winograd T.) 102
Вирт (Wirth N.) 480, 565; *29*
Возенкрафт (Wozencraft J. M.) 569
Вуд (Wood D.) 408
Вудс (Woods W. A.) 102

Галлер (Galler B. A.) 74
Гилл (Gill A.) 163
Гинзбург А. (Ginzburg A.) 163
Гинзбург С. (Ginsburg S.) 123, 163, 192, 241, 268, 305
Гир (Gear C. W.) *422*
Гладкий А. В. 123, 241
Гончарова Л. И. 452
Глушков В. М. 163
Готлиб (Gotlieb C. C.) 352
Грау (Grau A. A.) *206*
Грин (Green M. W.) 96
Грейбах (Greibach S.) 123, 192, 241, 305
Грис (Gries D.) 95, 96
Гриswолд (Griswold R. E.) 562

[1]) Курсивом выделены номера страниц тома 2.— *Прим. перев.*

Гриффитс (Griffiths T. V.) 268, 351
Гросс (Gross M.) 241
Грэй (Gray J. N.) 220, 315, 480, 510, 558; *185*
Грэхем Р. (Graham R. M.) 510; *393*
Грэхем С. (Graham S. L.) 372, 480; *185*

Деннинг (Denning P. J.) 480
Де Ремер (De Remer F. L.) 452, 569; *45*, *117*, *138*
Джентльмен (Gentleman W. M.) 77
Джонсон В. (Johnson W. L.) 295
Джонсон С. (Johnson S. C.) 408, 409
Дейкстра (Dijkstra E. W.) 98
Домёлки (Domölki B.) 352
Дьюар (Dewar R. B. K.) 96
Дэвис (Davis M.) 51, 52

Ершов А. П. *393*, *423*, *448*
Ершова Н. М. *393*

Замельсон (Samelson K.) 510
Змиевская Л. Л. *362*
Зонис В. С. 452
Зыков А. А. 68

Ибарра (Ibarra O.) 220
Игараси (Igarashi S.) *362*
Ингерман (Ingerman P. Z.) 96
Ихбиа (Ichbia J. D.) 480; *45*

Кавинесс (Caviness B. F.) *362*
Камеда (Kameda T.) 163
Камынин С. С. *393*, *423*
Кантор (Cantor D. G.) 241
Каплан (Kaplan D. M.) *422*
Касами (Kasami T.) 372
Касьянов В. Н. *448*
Кауфман В. Ш. 408, 409
Кеннеди (Kennedy K.) *392*, *448*
Ким К. В. *423*
Китов А. И. *393*, *423*
Кларк (Clark E. R.) *422*
Клини (Kleene S. C.) 38, 52, 147
Кнут (Knuth D. E.) 52, 68, 74, 408, 452, 542; *163*, *185*, *206*, *266*, *292*, *448*
Кок (Cocke J.) 95, 96, 372; *422*, *448*
Колмерауэр (Colmerauer A.) 558
Комор (Komor T.) 408, 409
Конвэй М. (Conway M. E.) 408, 414
Конвэй Р. (Conway R. W.) 96
Кореньяк (Korenjak A. J.) 408, 452; *117*, *163*
Косараю (Kosaraju S. R.) 163
Коэн Д. (Cohen D. J.) 352
Коэн Р. (Cohen R. S.) 558
Кравчик (Krawczyk T.) 408
Криницкий Н. А. *393*, *423*
Кристенсен (Christensen C.) 75
Кук (Cook S. A.) 48, 220
Куно (Kuno S.) 351
Курки-Суонио (Kurki-Suonio R.) 408; *163*
Курочкин В. М. *423*

Лавров С. С. 558
Лакхэм (Luckham D. C.) *422*
Лалонд (Lalonde W. R.) 504
Лантен (Lentin A.) 241
Лафранс (LaFrance J.) *45*
Левит (Levitt K. N.) 96
Лейниус (Leinius R. P.) 452, 480; *117*
Ли Дж. (Lee J. A. N.) 96
Ли Э. (Lee E. S.) 504
Ливенворт (Leavenvorth B. M.) 74, 559
Локс (Loecks J.) 510
Ломет (Lomet D.) 417
Лоури (Lowry E. S.) *422*, *448*
Лукасевич (Lukasiewicz J.) 244
Лукаш (Lucas P.) 74
Льюис (Lewis P. M., II) 220, 268, 408; *91*, *236*, *326*
Любимский Э. З. *393*, *423*

Мак-Илрой ((McIlroy M. D.) 74, 77; *236*, *326*
Мак-Каллок (McCulloch W. S.) 123
Мак-Карти (McCarthy J.) 96
Мак-Кимаи (McKeeman W. M.) 96, 480, 510; *422*
Мак-Клюр (McClure R. M.) 96, 542; *236*
Мак-Нотон (McNaughton R.) 147; *163*
Максвелл (Maxwell W. L.) 96
Мальцев А. И. 43, 46, 52
Манна (Manna Z.) *422*
Марилл (Marill M.) *422*
Марков А. А. 42
Мартин (Martin D. F.) *29*
Маурер (Maurer W. D.) *292*
Медлок (Medlock C. W.) *422*, *448*
Мейерс (Meyers W. J.) *392*
Миллер В. (Miller W. F.) 102
Миллер Г. (Miller G. A.) 147
Минский (Minsky M.) 43, 52, 122, 163
Мицумото (Mizumoto M.) 102
Мишкович Р. Д. *362*

Монтанари (Montanary U. G.) 102
Морган (Morgan H. L.) 96, 296
Морзе (Morse S. P.) 480; *45*
Моррис (Morris R.) 96, 351; *292*, *326*
Мостинская С. В. *393*
Моултон (Moulton P. G.) 96
Мунро (Munro I.) 68
Мюллер (Muller M. E.) 96

Наката (Nakata I.) *392*
Наур (Naur P.) 74, 559
Непомнящая А. Ш. 408
Нивергельт (Nievergelt J.) *422*
Никитченко Н. С. 408
Нийхольт (Nijholt A.) 408

Огден (Ogden W.) 241
Олейник-Овод Ю. И. *423*
Ордян А. А. 558
Оре (Ore O.) 68

Павлидис (Pavlidis T.) 102
Парик (Parikh R. J.) 241
Паул (Paul M. C.) 192; *163*
Пейджер (Pager D.) *91*
Пейнтер (Painter J. A.) 96
Перлз (Perles M.) 241
Перлис (Perlis A. J.) 74
Петерсон (Peterson W. W.) *292*
Петрик (Petrick S. R.) 351
Петроне (Petrone L.) 268
Питтс (Pitts E.) 123
Поддерюгин В. Д. *423*
Пол (Paul M.) 510
Полонский (Polonsky I. P.) 562
Портер (Porter J. H.) 295
Пост (Post E. L.) 42, 52
Поттосин И. В. *393*, *423*, *448*
Поудж (Poage J. F.) 562
Пратер (Prather R. E.) 163
Проссер (Prosser R. T.) *422*
Пфальц (Pfaltz J. L.) 98, 102
Пэр (Pair C.) 480

Рабин (Rabin M. O.) 123, 147, 162
Радке (Radke C. E.) *292*
Райс (Rice H. G.) 192
Рассел (Russel L. J.) 96; *206*
Редзиевский (Redziejowski R. R.) *392*
Редько В. Н. 147, 408
Рейк (Rake S. T.) *206*
Рейнольдс (Reynolds J. C.) 96, 315, 351
Ренделл (Randell B.) 96; *206*
Робертс (Roberts P. S.) 352
Роджерс (Rogers H.) 46
Розен (Rosen S.) 95, 351
Розенкранц (Rosenkrantz D. J.) 123, 192, 408; *91*, *163*, *236*, *326*
Розенфельд (Rosenfeld A.) 98, 102
Росс (Ross D. T.) 295
Роудз (Rhodes E. M.) 408
Руднева Т. Л. *393*
Руццо (Ruzzo W. L.) 372

Саломаа (Salomaa A.) 147, 163, 241
Саммет (Sammet J. E.) 42, 74
Сети (Sethi R.) *392*
Скотт (Scott D.) 123, 147, 162
Стил (Steel T. B.) 74
Стирнз (Stearns R. E.) 220, 241, 268, 408; *91*, *163*, *326*
Стокхаузен (Stokhausen) *392*
Стоун (Stone H. S.) *392*
Стэндиш (Standish T.) 96
Суппес (Suppes P.) 13
Сэттли (Sattley K.) 351

Танака (Tanaka K.) 102
Тоёда (Tojoda J.) 102
Томпсон (Thompson K.) 136, 296
Торин (Torrii K.) 372
Трахтенброт Б. А. 163
Трахтенброт М. Б. *448*
Трахтенгерц Э. А. 510
Трохан Л. К. *362*
Тьюринг (Turing A. M.) 42, 43, 51, 123

Уилкокс (Wilcox T. R.) *206*
Уллиан (Ullian J. S.) 241
Ульман (Ullman J. D.) 52, 123, 200, 241, 283, 408, 409, 452, 480, 542; *29*, *91*, *117*, *236*, *266*, *292*, *326*, *362*, *392*, *448*
Уолтерс (Walters D. A.) 452
Уорли (Worley W. S.) 96
Уортмэн (Wortman D. B.) 96, 510
Уоршолл (Warshall S.) 68, 96

Фельдман (Feldman J. A.) 95, 96, 499, 510
Фишер М. (Fischer M. J.) 123, 480; *195*, *326*
Фишер П. (Fischer P. S.) 52
Флойд (Floyd R. W.) 68, 96, 192, 241, 351, 480, 510; *29*, *362*, *392*
Фостер (Foster J. M.) 408
Фрейли (Frailey D. J.) *392*

Фримэн (Freeman D. N.) 96, 296
Фу (Fu K. S.) 102
Фуксман А. Л. 408, 409
Футрель (Futrelle R. P.) 268

Хавел (Havel I. M.) 510
Халмош (Halmos P. R.) 13, 38
Хант (Hunt H. B.) 452
Харари (Harary F.) 68
Харрисон (Harrison M. A.) 163, 220, 315, 372, 510, 558; *185*
Хартманис (Hartmanis J.) 52, 147, 220
Хаскел (Haskell R.) 315
Хафмен (Huffman D. A.) 162
Хейнс Л. (Haines L. H.) 123
Хейнс Х. (Haynes H. R.) *45*
Хейс (Hays D. G.) 372
Хекст (Hexst J. B.) 352
Хект (Hecht M. S.) *448*
Хеллерман (Hellerman H.) *392*
Хомский (Chomsky N.) 42, 74, 102, 123, 147, 192, 220, 241
Хопгуд (Hopgood F. R. A.) 96, 510
Хопкрофт (Hopcroft J. E.) 52, 123, 163, 220, 241, 408, 452; *163, 326, 448*
Хорвиц (Horwitz L. P.) *392*
Хорнинг (Horning J. J.) 96, 504, 510
Хохшпрунг (Hochsprung R. R.) 96

Цейтин Г. С. 102

Чен (Chen S.) *392*
Чёрч (Church A.) 38, 42, 43
Читэм (Cheatham T. E.) 74, 96, 315, 351; *45*
Чулик I (Čulik K., I) 268
Чулик II (Čulik K., II) 408, 558

Шамир (Shamir E.) 241
Шапиро (Shapiro R. M.) 96
Шварц (Schwartz J. T.) 95, 96; *448*
Шевченко В. В. 408
Шепердсон (Sheperdson J. C.) 147
Шефер (Schaefer M.) *448*
Шиманский (Szymansky T. G.) 452
Шкильняк С. С. 408
Шорре (Schorre D. U.) 96, 351, 542; *236*
Шоу (Shaw A. S.) 98, 102
Штрассен (Strassen V.) 48, 68
Шумей А. С. 452, 510
Шура-Бура М. Р. *393, 423*
Шютте (Schutter L. J.) *45*
Шютценберже (Schutzenberger M. P.) 192, 220, 241

Эванс (Evans A., Jr.) 510, 569
Эви (Evey R. J.) 192, 220
Эйкель (Eickel J.) 510
Эклн (Ackley S. I.) 295
Элсон (Elson M.) *206*
Элспас (Elspas B.) 96
Энгелер (Engeler E.) 74
Энглунд (Englund D. E.) *442, 448*
Эрлаид (Irland M. I.) 52
Эрлн (Earley J.) 372; *45, 117*
Эттингер (Oettinger A.) 220, 351

Ющенко Е. Л. 408, 411

Яжабек (Jarzabek S.) 408
Ямада (Jamada H.) 147
Янгер (Younger D. H.) 372
Янов Ю. И. *422*

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ К I И 2 ТОМАМ [1])

Аванцепочка (lookahead string) 378, 427 *см. также* Заглядывание вперед
Автомат (automaton) *см. также* Распознаватель, Преобразователь
— анализирующий (parsing) *117—138*
— — каноническнй (canonical) *119, 120, 129*
— — полуприведенный (semireduced) *125, 134*
— — расщепленный (split) *123—125, 130, 131*
— конечный (finite) 138, 147—151, 286—293, 449
— — двусторонний (two-way) 146
— — детерминированный (deterministic) 138, 287
— — недетерминированный (nondeterministic) 135, 287—293
— — полностью определенный (completely specified) 135
— — приведенный (reduced) 148—151
— линейно ограниченный (linear bounded) 120
— с магазнннной памятью (pushdown) 114, 193—220, 318
— — двусторонний (two-way) 219, 220
— — детерминированный (deterministic) 221—220, 228, 229, 237—240, 283, 384, 450, 501, 503, 522—525; *157—159, 169, 176, 182—184, 164—185*
— — — в нормальной форме (normal form DPDA) *164—169*
— — — дочитывающнй (continuing) 215, 216
— — — незацикливающийся (halting) 318
— — расширенный (extended) 199—201, 212
— — с одним поворотом (one-turn) 237
Алгебра булева (Boolean algebra) 36, 153
Алгол (ALGOL) 74, 75, 226, 227, 264, 287, 347, 408, 559
Алгоритм (algorithm) 38—51
— всюду определенный 40
— Домёлки (Domölki's) 351, 352, 507, 510
— Евклида *397*
— Кока — Янгера — Касами (Cocke — Younger — Kasami) 352—358
— Маркова 42
— недетерминированный (nondeterministic) 320, 346, 347, 351
— разбора, предсказывающнй (predictive parsing) 205, 378—381, 391—395, 408
— — корректный (valid) 380
— Уоршолла (Warshall's) 63, 68
— частичный (procedure) 38—51
— Эрли (Earley's) 358—372, 450
Алфавит (alphabet) 27
— входной (input) *см.* Символ входной

[1]) См. примечанне на стр. 467.— *Прим. перев.*

Алфавит выходной (output) *см.* Символ выходной
— состояний (of states) *см.* Состояние
Альтернатива (нетерминала) (alternate) 321
Анализ (analysis)
— интервалов (interval) *424—448*
— лексический (lexical) 76—79, 283—296; *198, 259, 268, 269, 304*
— — непрямой (indirect) 78, 286—290
— — прямой (direct) 78, 290—293
— семантический (semantic) *199*
— синтаксический (syntactic) *см.* Разбор
Анализатор (parser) *см. также* Анализ
— двухмагазинный (two-stack) 544—547, 556—558
— канонический LR(*k*) (canonical LR(*k*)) 444—447
— левый (left) 299—301
— по левому участку (left-corner) 348—350
— правый (right) 302—304, 338
— предсказывающий (predictive) *см.* Алгоритм разбора, предсказывающий
A-правило (*A*-production) 175

Блок (block)
— открытый (open) *339—342*
— приведенный (reduced) *342—345*
Блок-схема (flow chart, d-chart) 98—101

Веер (в дереве) *см.* Куст (в дереве)
Вершина (графа) (node, vertex) 52
— концевая *см.* Лист
— младшая (minor) *373, 388*
— начальная (begin) *400, 424*
— старшая (major) *373, 388*
Вершины независимые (independent nodes) *17—19*
Вес LR(*k*)-таблицы (height of LR(*k*)) *66*
Включение (множеств) (inclusion) 13, 238
Вход (input) 38 *см. также* Лента входная
Вывод (derivation) 107, 118
— левый (leftmost) 167, 168, 232, 233, 297, 356, 357
— правый (rightmost) 167, 168, 297
Выражение (expression)
— арифметическое (arithmetic) 72, 73, 108, 245, 253; *246—252, 256—258, 362—393*
— доступное (available) *423*
— инфиксное (infix) 244
— постфиксное (postfix) 244, 245, 247, 248, 259, 264, 267, 529; *201, 211, 212*
— префиксное (prefix) 244, 245, 259, 264, 267; *201, 207*
— расширенное регулярное (extended regular) 284—290
— регулярное (regular) 124—131, 145, 147
Высота (вершины дерева) (height) 58, 84
Выход (output) 38, 243, 255, 258
Вычисления (computations)
— избыточные (redundant) *334, 335, 338—342, 345—347, 404—407*
— периода компиляции (compile time) *407*

Генерация кода (code generation) 75, 82—88, 90, 92, 93; *200, 204, 245, 259—261, 347—351*
Глубина (вершины дерева) (depth) 58

66.2-9

Головка входная (распознавателя) (input head) 113—115
Гомоморфизм (homomorphism) 29, 225, 236, 238, 239, 243, 244; *163*
Грамматика (grammar) 105
— автоматная *см.* Грамматика регулярная
— без *e*-правил (*e*-free) 172, 173, 314, 340, 343, 346, 376, 402, 450, 478, 492
— без ограничений (unrestricted) *см.* Грамматика общего вида
— без циклов (cycle-free) 175, 312, 340, 343, 346, 367
— бесконтекстная *см.* Грамматика контекстно-свободная
— входная (СУ-схемы) (input, underlying) 250; *237, 296*
— выходная (output) 250
— индексная (indexed) 120, 121
— каноническая (canonical) *166—172, 174, 175*
— Колмерауэра (Colmerauer) 549, 554—558
— контекстно-зависимая (context-sensitive) 111, 112, 117, 119, 121, 237, 452
— контекстно-свободная (context-free) 111, 112, 117, 119, 121, 237
— левоанализируемая (left parsable) 304—306, 381; *145, 146*
— леволинейная (left linear) 145
— леворекурсивная (left recursive) 178—181, 324, 325, 331, 332, 385, 400; *154*
— линейная (linear) 191, 237, 268
— LC(*k*) 402—408
— LL 376
— LL(*k*) 301, 373—408, 449, 450; *46, 114, 115, 137, 139—164, 185, 187, 192—194, 207—210, 221—224*
— LL(0) 162
— LL(1) 373, 382—387, 408, 411, 593; *113, 134—136, 148, 149, 161, 162, 192—194, 209, 214*
— LR(*k*) 304, 373, 421, 423—452, 478, 482, 484, 503; *45—138, 139—146, 207, 209—215, 218—221*
— — простая (simple) *75, 92—102, 103, 111—116, 135*
— — с заглядыванием (LALR(*k*)) *100, 101, 114—117, 135*
— LR(0) *57, 113, 117—129, 145, 164, 174, 176, 183*
— LR(1) 424, 463, 503, 504; *164, 176*
— неоднозначная (ambiguous) 168, 189, 231—236, 239, 317, 546; *151, 188 см. также* Грамматика однозначная
— неукорачивающая *см.* Грамматика контекстно-зависимая *и* Грамматика без *e*-правил
— общего вида (unrestricted) 105—112, 118—120, 122
— обратимая (unique invertible) 422, 450, 457, 503—506, 547, 557; *188—192, 195*
— ограниченного контекста (bounded context) 505, 507
— ограниченного правого контекста (bounded right context) 481—488, 503—507
— однозначная (unambiguous) 119, 168, 231—233, 240, 364—366, 384, 447, 449, 460, 476, 485, 549; *136*
— операторная (operator) 190, 492
— операторного предшествования (operator precedence) 493—497, 503, 504, 507; *14—16, 188—195*
— (1,1)-ОПК ((1,1)-bounded-right-context) 484, 503; *164—176*
— ОПК (BRG, bounded right context) *139, 174—176, 184, 193*
— (1,0)-ОПК *164, 174—176, 184*
— остовная (skeletal) 496, 507; *80*
— (2,1)-предшествования (2,2)-precedence 480, 503; *139, 164, 176—182*
— правоанализируемая (right parsable) 304—306; 450; *146*
— праволинейная (right linear) 111, 119, 131—133, 143—145, 230, 237
— праворекурсивная (right recursive) 178
— порождающая графы (graph, web) 98—102
— пополненная (augmented) 424, 481; *104*
— предшествования (precedence) 456, 457, 463, 480; *7—29, 46*
— приведенная (proper) 175; *169*

Грамматика простая LL(1) (simple LL(1)) 376, 408, 409
— простая смешанной стратегии предшествования (simple-mixed-strategy precedence) 491, 503, 507
— простого предшествования (simple precedence) 455—463, 474—479, 549, 565; *8, 9, 19—25, 85, 139, 185—189, 192—194*
— псевдоразделенная 409, 410, 414, 420
— разделенная *см.* Грамматика простая LL(1)
— расширенного предшествования (extended precedence) 463—469, 478—480, 484, 505; *194*
— регулярная (regular) 145, 557
— рекурсивная (recursive) 178
— свойств (property) *199, 267, 292—326*
— — недетерминированная (nondeterministic) *324*
— сильно LL(*k*) (strong LL(*k*)) 384, 388
— скобочная (parenthesis) *163*
— слабого предшествования (weak precedence) 469—477, 479, 491, 492, 503—506; *16, 24, 28, 29—45, 140, 189—192*
— слаборазделенная 410, 411, 414
— смешанной стратегии предшествования (mixed-strategy precedence) 488—492, 503, 507; *16*
— простая (simple, SMSP) *113, 139, 164, 169—174*
— с индикаторами (tagged) *87, 89, 90, 135*
— с самовставлением (self-embedding) 240
— *T*-канонического предшествования (*T*-canonical precedence) 507—510
— *T*-остовная (*T*-skeletal) 509
— Хомского (Chomsky) *см.* Грамматика Граф (graph) 52—68
— ациклический (ориентированный) (directed acyclic, dag) 54, 138; *240—244, 333—345, 349*
— линеаризации (linearization) *11—14*
— нагруженный *см.* Граф помеченный
— неориентированный (indirected) 66
— переходов (автомата) (transition) *см.* Диаграмма конечного автомата
— помеченный (labelled) 53, 57
— производный (derived) *427*
— сверток (reduce) *40*
— связный (неориентированный) (connected) 66
— сдвигов (shift) *36—40*
— сильно связный (strongly connected) 54
— упорядоченный (ordered) 56
— — ациклический (dag) 57
— управления (flow) *393, 399, 400*
— — несводимый (irreducible) *439—443*
— — предельный (limit) *427*
— — сводимый (reducible) *424, 427, 431—439, 446*
— GOTO *67*

Дерево (tree) 55, 61, 62, 71—73, 81—87, 100, 101, 319, 487—490
— ассоциативное (associative) *381—388*
— вывода (derivation) *см.* Дерево разбора
— двоичного поиска (binary search) *272*
— неориентированное (undirected) 66, 67
— остовное (spanning) *36—40*
— разбора (parse) 164—168, 205—207, 250—252, 307, 431, 432, 519—521; *198—204, 365*
— — помеченное (labelled) *366, 367, 381—388*
— синтаксическое (syntax) *см.* Дерево разбора
— упорядоченное (ordered) 57, 58

Диаграмма
— конечного автомата (transition graph) 138, 255, 256
— синтаксическая (syntactical diagram) 414, 415
Диаграммер (diagrammer) 415—418
Дифференцирование (differentiation) *238—240*
Длина (length)
— вывода (of a derivation) 107
— цепочки (of a string) 28
ДМП-автомат *см.* Автомат с магазинной памятью детерминированный
Доминатор (dominator) *401—406, 410, 420, 425, 448*
Дополнение (множества) (complementation) 14, 216, 226, 237, 541
Допускать (цепочку, язык) (accept) 115, 136, 195, 201
Дуга (в графе) (arc, edge) 53

e-правило (грамматики) (*e*-production) 111, 177, 178, 340, 402; *147—153, 159—162, 163*
e-такт (распознавателя) (*e*-move) 194, 218

Заглядывание вперед (lookahead) 337, 344, 370, 371, 373—375, 378, 402, 421, 424, 450
Заголовок интервала (header of an interval) *424—427*
Загрузчик (loader) *197*
Задача слияния множеств (set merging problem) *316—323, 326*
Закон (law)
— ассоциативности (associative) *352—356, 360, 376, 379—388*
— дистрибутивности (distributive) *352*
— коммутативности (commutative) *352—356, 360, 376—388*
Законы де Моргана (De Morgan's laws) 23
Замена сложных операций (reduction in strength) *408, 415—417*
Замкнутость (относительно операций) (closure) 152, 224—226, 257, 266
Замыкание (отношения) (closure)
— множества допустимых ситуаций (of a set of valid items) *104*
— рефлексивное и транзитивное (reflexive and transitive) 19
— транзитивное (transitive) 18, 62—65, 68
Запись польская (Polish notation) *см.* Выражение префиксное
Значение блока (value of a block) *330, 341*

Идентификатор (identifier) 76—80, 116, 286, 287, 289—293
Иерархия Хомского (Chomsky hierarchy) 112
Индекс
— грамматики, языка 239, 240
— отношения эквивалентности (index) 17
Интерпретатор (interpreter) *197, 201*
Исправление ошибок (error correction) 75, 90—93, 96, 337, 338, 407, 446, 451, 452, 480; *9, 10, 53*
Исчисление высказываний (propositional calculus) 35, 50
Итерация (языка) (closure) 29, 225
— маркированная (marked) 240
— позитивная (positive) 29

Кисть (cluster)
— графа линеаризации (of a linearization graph) *18*
— синтаксического дерева (of a syntax tree) *379, 380*
КЗ-грамматика *см.* Грамматика контекстно-зависимая

Код (code)
— машинный перемещаемый (relocatable machine) *197*
— многоадресный (multiple address) *201—204*
— объектный (object) 75, 82, 242; *196*
— промежуточный (intermediate) 75, 82—87; *199—204*, *327*, *328*, *394*
Компилятор (compiler) 75—96, 351, 408
— компиляторов (compiler-compiler) 96, 351
Компиляция синтаксически управляемая (syntax directed compiling) *206*
Композиция (отношений) (composition) 25, 281
Конкатенация (concatenation) 27, 29, 225, 238; *163*
— маркированная (marked) 240
Конфигурация (configuration) 49, 115, 135, 194, 254, 258, 326, 340, 378, 379, **404**, 411, 412, 415, 418, 422, 423, 453, 534, 545; *50*
— допускающая (accepting) *см.* Конфигурация заключительная
— достижимая (accessible) *51*
— заключительная (final) 115, 135, 195, 201, 255, 258, 259, 379; *51*, *120*
— зацикливающая (looping) 213—216
— начальная (initial) 115, 135, 194, 326, 340; 379; *50*, *120*
Конфликт в таблице расстановки (collision in a hash table) *275—279*
— отношений предшествования (precedence conflict) 472, 473
— правил 389, 393
— „перенос — свертка" (shift — reduce) *114*
Крона (дерева разбора) (frontier) 165—168
КС-грамматика *см.* Грамматика контекстно-свободная
Куст (в дереве) 206

Лексема (token) 76—79, 283—296
Лемма Огдена (Ogden's lemma) 220—223
— о разрастании (pumping lemma)
— (для КС-языков) 223, 224
— (для регулярных множеств) 152
Лента входная (распознавателя) (input tape) 113—115
Лист (в графе) (leaf) 54
ЛО-автомат *см.* Автомат линейно ограниченный

Магазин (pushdown list) 114, 192—194, 378
Макрос синтаксический (syntax macro) 265, 266, 559—562
Маркер концевой (endmarker) 113, 304, 326, 378, 381, 409, 412, 415, 418, 457, 522, 525, 541; *173*, *174*, *175*, *182*, *189—192*
Матрица (matrix)
— предшествования (precedence) 457—459
— — приведенная (reduced) *11*
— сверток (reduce) *40*
— смежностей (adjacency) 62, 63
Машина (machine)
— анализирующая (parsing) 533—538, 540; *224—232*
— Тьюринга (Turing) 42, 49—51, 120, 123
— — универсальная (universal) 50
— с произвольным доступом к памяти (random access) 154, 189, 354, 355, 372, 528, 530, 531 *см. также* Операция элементарная (алгоритма)
Метод цепочек (chaining) *289*
— прямой (direct) *289*
Множество (set) 11—30
— бесконечное (infinite) 22, 26
— вполне упорядоченное (well ordered) 24, 30
— знаменательных символов (token) 508

Множество конечное (finite) 12, 222
— линейное (linear) 239
— непротиворечивое LR(k)-ситуаций (consistent set of LR(k) items) 442, 443
— отсрочки (postponement) *67—76*, *96*
— полулинейное (semi-linear) 239
— пустое (empty) 12
— регулярное (regular) 124—163, 218, 219, 225, 236—238, 257, 266, 269, **270**, 400, 478; *163*
— рекурсивное (recursive) 42, 48, 112, 119, 120
— рекурсивно перечислимое (recursively enumerable) 42, 48, 111, 112, 118, **268**, 558
— счетное (countable) 22, 26
— универсальное (universal) 14
— упорядоченное (ordered) 20
— LR(k)-таблиц каноническое (canonical set of LR(k)) *45*, *95*
— — — корректное (valid) *51*
— — — φ-недостижимое (φ-inaccesible) *55—65*, *69—72*, *82*, *85*, *86*
Модуль загрузки (load module) *197*
Мощность цепочки (thickness of a string) *156*, *157*, *163*
МП-автомат *см.* Автомат с магазинной памятью

Неоднозначность (ambiguity) 231—236 *см. также* Грамматика неоднозначная *и* Язык неоднозначный
— конечной степени (finite) 371
— семантическая (semantic) 307, 308
— существенная (inherent) *см.* Язык неоднозначный
Нетерминал (nonterminal) 105, 120, 248, 512—514

Область (region) *409—411*
— с одним входом (with a single entry) *411*
Область действия (scope) *332*
Обратимость (unique invertibility) *см.* Грамматика обратимая
Обращение
— гомоморфизма (inverse homomorphism) 30
— цепочки или языка (reversal) 27, 144, 153
Обращение к памяти (memory reference) *389*
Объединение (union) 14, 225, 229—231, 238, 541
— маркированное (marked) 240
Однозначность семантическая (semantic unambiguity) *237*
ОК-грамматика *см.* Грамматика ограниченного контекста
Оператор
— бесполезный (useless statement) *327*, *332—334*, *338—342*, *404*, *423*
— определения (definition statement) *395*
— перехода вычисляемый (computed goto) *30*
Операция элементарная (алгоритма) (elementary operation) 355, 357, 364—**366**, 447
— идемпотентная (self-inverse) *352*
— коммутативная (commutative) *352*, *353—357*, *360*, *376—389*
ОПК-грамматика *см.* Грамматика ограниченного правого контекста
Определение регулярное (regular definition) 285, 286
Оптимизация кода (code optimization) 75, 88—90; *200*, *249*, *250*, *327—448*
Организация информации (bookkeeping) *198—200*, *259—261*, *267—326*
— для языка с блочной структурой (for block-structured language) *271*, *293*, *294*, *297—304*
Основа (правовыводимой цепочки) (handle of a right sentential form) 205, 206, 429, 431—434, 455—457, 543

Остов (графа) (spanning tree) 67
Отображение (mapping) *см.* Функция
Отношение (relation) 16—26
— антисимметричное (antisymmetric) 21
— асимметричное (asymmetric) 20
— иррефлексивное (irreflexive) 20
— конгруэнтности (congruence) 158
— обратное (inverse) 16, 22
— операторного предшествования (operator precedence) 493
— предшествования Вирта — Вебера (Wirth-Weber precedence) 456, 457
— — Колмерауэра (Colmerauer precedence) 547—549
— рефлексивное (reflexive) 17
— симметричное (symmetric) 17
— транзитивное (transitive) 17
— эквивалентности (equivalence) 17, 18, 24, 149—151, 157, 158
— — правоинвариантное (right invariant) 157, 158
Оценка блоков приемлемая (cost criterion on blocks) *345*, *346*

Память (распознавателя) (memory) 113—115
Пара цепочек выводимая (translation form) 246—249
Перевод (translation) 71, 242—245, 258, 379, 380, 545; *196—266*
— регулярный (regular) *см.* Преобразование конечное
— синтаксически управляемый (syntax directed) 74, 83—88, 246—253, 260—282, 499, 569—574; *206—266*, *363*
— — простой (simple) 253, 260—263, 271—274, 282, 298, 569—574; *207*, *214*
— наследственный (inherited) *255—259*, *263*
— синтезированный (synthesized) *255—259*, *263*
Переменная (variable)
— активная (active) *332*, *423*
— входная (input) *329*
— выходная (output) *329*
— индуктивная (inductive) *412—415*
Перемещение кода (code motion) *411*, *412*
Пересечение (множеств) (intersection) 14, 225, 226, 230, 237, 541; *163*
Переименование переменных (renaming of variables) *335*, *336*, *338—342*, *345—347*
Перестановка операторов (flipping of statements) *336—342*, *345—347*
Пиг лэтин (pig Latin) *222—224*, *234*
ПЛ/1 (PL/I) 74, 76, 78, 559
ПЛ 360 (PL 360) 565—569
Подстановка (языков) (substitution) 224, 225
Позиция (в цепочке) (position) 220
Покрытие (грамматики) (cover) 309—311, 314, 315
— левое (left) 310, 314, 315, 345; *163*
— правое (right) 310, 314, 315, 345; *184*, *195*
Порядок (как отношение) (order) 20, 21
— лексикографический (lexicografic) 25, 331, 343
— линейный (linear) 21, 25, 59, 60; *350*
— обратный (вершин дерева) (postorder) 59
— полный (well) 24, 30
— прямой (вершин дерева) (preorder) 58; *144*
— частичный (partial) 20, 21, 25, 26, 59, 60; *350*
Порядок (схемы СУ-перевода) (order) 274—281
Последовательность пишущая (write sequence) *168*
Постдоминатор (postdominator) *420*, *421*
Поток данных (data flow) *423—448*
Потомок (в графе) (descendant) 55
Правило (грамматики) (production) 106, 120

Правило цепное (single) 173, 174, 507; *76—84*, *86*, *188*, *189*
Правило вывода (в формальной системе) (rule of inference) 31
Предикат (predicate) 12
Предложение (sentence) *см.* Цепочка
Предок (в графе) (ancestor) 55
Представление (дерева) (representation)
— левое скобочное (left-bracketed) 61, 245
— правое скобочное (right-bracketed) 61, 245
Преобразование (transduction) *см.* Преобразователь
— конечное обратное (inverse finite) 257, 266
Преобразователь (transducer)
— конечный (finite) 254—258, 266, 268—270, 273, 281, 282, 284, 291; *198*
— — детерминированный (deterministic) 256, 257, 268
— с магазинной памятью (pushdown) 258—263, 267, 268, 298—301, 317—321, 379, 381, 402
— — — — детерминированный (deterministic) 259, 260, 283, 304—309, 379, 381, 402, 446, 498, 501; *206—214*, *234*
— — — — расширенный (extended) 302—304, 421—423
Префикс (цепочки) (prefix) 28
— активный (правовыводимой цепочки) (viable) 432, 444; *86*
Приоритет (операций) (precedence) 82, 168, 169, 264; *76*, *87*
Проблема (алгоритмическая) (problem) 43—52
— неразрешимая (undecidable) 44
— остановки (halting) 50
— принадлежности (membership) 154—156, 161, 162, 257
— пустоты (emptiness) 154—156, 161, 162, 169, 170, 539
— разрешимая (decidable) 44
— соответствий Поста (Post's correspondence problem) 47, 228, 229
— эквивалентности (equivalence) 154—156, 161, 162, 228—230, 268, 401; *156—159*, *184*, *421*
Программа (program)
— абсолютная (absolute machine code) *196*
— исходная (source) 75, 93, 242; *196*
— объектная (object) *см.* Код объектный
Продукция (production) *см.* Правило (грамматики)
Произведение декартово (Cartesian product) 16
Производная (регулярного выражения) (derivative) 160, 161
Профиль частотный (frequency profile) *398*
Проход компилятора (pass of a compiler) *200*, *260*
Процессор с магазинной памятью (pushdown processor) *214—224*, *240—242*
Путь (в графе) (path) 54, 66
— вычислений (computation) *401*, *431*

Разбиение совместимое (compatible partition) *59—64*, *69*, *97*
Разбор (как синтаксический анализ) (parsing) 72, 75, 80—82, 90—93, 296—309
— восходящий (bottom-up) 205—209, 301—309, 338—344, 542—558; *134—136*, *218—221*, *246*, *297 см. также* Грамматика предшествования, LR(k), ОПК
— нисходящий (top-down) 205, 297—301, 304—309, 321—338, 499, 511—542; *134—136*, *221—224*, *246*, *297 см. также* Грамматика LL(k)
— — по текущему символу 408—419
— по левому участку (left corner) 313, 314, 348—350, 403—406
— сверху вниз *см.* Разбор нисходящий
— снизу вверх *см.* Разбор восходящий
— с возвратами (backtrack) 317—352, 511—558; *224—232*
— типа „перенос — свертка" (shift—reduce) 303, 338, 350, 351, 420—423, 426, 427; *8*, *304*, *305*
Разбор (как результат синтаксического анализа) (parse) 297, 379, 544; *198*, *260*

Разбор левый (left) *см.* Вывод левый
— по левому участку (left corner) 313, 314, 404
— правый (right) 297, 367 *см. также* Вывод правый
— частичный левый (partial left) 329, 330
— — правый (partial right) 343
Развертывание циклов (loop unrolling) *417, 418*
Разметка (графа) (labeling) 53, 57
Разность (множеств) (difference) 14
Распознавание образов (pattern recognition) 98—102
Распознаватель (recognizer) 113—116, 123 *см. также* Автомат
— адекватный 410, 413, 417, 419, 420, 501, 513
— односторонний (one-way) 113
— простой МП 409
— синхронный 418—420
— СМР 412—416
Распределение памяти (storage allocation) *200*
Расстановка (hashing)
— линейная (linear) *285, 286*
— по позициям (on locations) *285—287*
Расщепление (splitting)
— грамматики (grammar splitting) *102—117*
— множества LR(*k*)-ситуаций (of a set of LR(*k*) items) *100*
— состояний анализирующего автомата (of states of parsing automaton) *122—125, 129—131*
Редактор связей (link editor) *197*
Рубеж (в выводимой цепочке) (border) 374, 421

Свертка (цепочки) (reduction) 205, 302, 338, 339, 344 *см. также* Разбор типа „перенос — свертка"
Свойство (property)
— допустимое (acceptable) *296*
— нейтральное (neutral) *295—297, 304, 309—313, 323—325*
— префиксное (prefix) 29, 31, 239, 289; *164, 183, 184*
— суффиксное (suffix) 29, 31
Связка логическая (logical connective) 33—35
Семантика (semantics) 71—74, 243—246
Сечение (дерева разбора) (cut) 165
Символ (symbol)
— бесполезный (в грамматике) (useless) 169, 171, 172, 275, 282, 315
— вспомогательный *см.* Нетерминал
— входной (input) 135, 193, 194, 248, 254, 412, 415
— выходной (output) 248, 258
— исчезающий (nullable) *147—153*
— магазинный (pushdown) 193
— начальный (initial, start) 106, 120, 194, 249, 514
— недостижимый (в КС-грамматике) (inaccessible) 170, 171
— нетерминальный (nonterminal) *см.* Нетерминал
— терминальный (terminal) *см.* Терминал
Синтаксис (syntax) 71—74
Система (system)
— каноническая LR(*k*)-таблиц (canonical set of LR(*k*) tables) 444, 445
— каноническая множеств допустимых ситуаций (canonical collection of sets of valid items) *85, 91*
— Поста каноническая (Post's canonical) 42, 123
— расстановки (hashing system)
— — *k*-равномерная (*k*-uniform) *291*
— случайная (random) *280, 283—287, 289—291*

Система стандартная уравнений с регулярными коэффициентами (set of regular expression equations in standard form) 127—131
— формальная (formal) 31
Ситуация (item)
— в алгоритме Эрли 359, 371, 450
— — LR(*k*)-алгоритме 433
— — допустимая (valid) 433, 435—441, 446
— — квазидопустимая (quasivalid) *108*
Слияние столбцов (в LR(*k*)-анализаторе) (column merger) *80*
Слово (word) *см.* Цепочка
Сложность вычисления (временная и емкостная) (computational complexity; time complexity, space complexity), 41, 158, 159, 162, 189, 333—337, 343, 355—357, 364—366, 368—372, 395, 447, 460, 530—532, 557; *214, 322, 323, 347, 358, 430, 446, 447*
Снобол (SNOBOL) 70, 562—565
Сортировка топологическая (topological sort) 59, 60
Составляющая (phrase) 543
Состояние (распознавателя, преобразователя) (state) 135, 193, 194, 254, 326, 411, 412, 415
— выталкивания (pop) *122, 123, 130*
— достижимое (accessible) 140, 148
— заключительное (final) 135, 194, 254, 326, 412
— записи (write) *164, 167, 183*
— заталкивания (push) *130*
— начальное (initial) 135, 194, 254, 412
— опроса (interrogation) *122, 130*
— переноса (shift) *118*
— свертки (reduce) *118*
— стирания (erase) *164, 167, 183*
— чтения (read, scan) *130, 164, 167, 183*
Состояния неразличимые (конечного автомата) (indistinguishable states) 148; ***60***
— — анализирующего автомата (indistinguishable states of parsing automaton) *126*
— различимые анализирующего автомата (distinguishable states of parsing automaton) *126*
Список (list)
— магазинный (pushdown) *см.* Магазин
— пересечений (intersection list) *306—316, 322, 323*
— разбора (parse) 359
— свойств (property) *306—314, 322, 323*
ССП-грамматика *см.* Грамматика смешанной стратегии предшествования
Степень (вершины графа) (degree)
— по входу (in-) 54
— — выходу (out-) 54
СУ-перевод *см.* Перевод синтаксически управляемый
Суффикс (цепочки) (suffix) 28
Схема перевода (translation scheme) *см.* Перевод
— — зацикленная (circular) *256*
— — обобщенная (generalized) *236—244, 261—266*
— — простая постфиксная (simple postfix) *210, 235, 236*
Сцепление (concatenation) *см.* Конкатенация

Таблица (table)
— идентификаторов (symbol) *см.* Таблица имен
— имен (symbol) 75, 79, 80, 92, 93, 287
— разбора (parse) 352
— расстановки (hash table) *274—287*

Таблица с прямым доступом (direct access table) *270—274*
— управляющая разбором (parsing) 379, 385—387, 391—395, 404, 505; *48 см. также* LR(*k*)-таблица
— LL(*k*) 389—391, 394, 395
— LR(*k*) 426, 427, 444—446, 451; *45—138, 215—218*
ТАГ-система (Tag system) 42, 122
Такт (распознавателя) (move) 155, 135, 194
Тезис Чёрча — Тьюринга (Church-Turing thesis) 43
Теорема (theorem) 32
— Парика (Parikh's) 239, 241
Терминал (в грамматике) (terminal) 106, 120, 514
Точка наименьшая неподвижная (minimal fixed point) 127, 129—131, 144—146, 185, 186
Траверс (МП-автомата) (traverse (of a PDA)) *165, 167*
Транслятор (translator) 77, 246, 247 *см. также* Преобразователь
Трансляция (translation) *см.* Перевод

Узел (графа) (node) *см.* Вершина (графа)
Упорядочение (ordering) *см.* Порядок (как отношение)
Уравнения (equations)
— определяющие (для КС-языков) (defining) 185, 186
— с регулярными коэффициентами (regular expression) 126—133, 144, 146
Уровень (вершины дерева) (level) *см.* Высота (вершины дерева)
Устройство управляющее с конечной памятью (finite control) 114, 498 *см. также* Состояние (распознавателя)
Участок линейный (straight-line code) *328—362*

Фаза компиляции (phase of compilation) *198—200, 259—261*
Факторизация левая (left factoring) 385
Форма (form)
— Бэкуса — Наура (Backus-Naur) 74
— нормальная Грейбах (Greibach normal) 182—188, 190, 274, 315, 402, 406; *141, 154, 163, 184*
— — слабая 409
— нормальная Хомского (Chomsky normal) 176, 177, 190, 273, 274, 310, 311, 314, 352, 401; *163, 184, 305*
Фортран (FORTRAN) 70, 74, 77, 79, 236, 284, 286, 346, 347, 559; *206, 266, 288, 398, 408, 422, 446, 448*
Функция (function) 21, 22, 25
— биективная (bijective) 22
— взаимно однозначная *см.* Функция инъективная
— всюду определенная (total) 21, 25
— действия (parsing action) 426—428, 444—446
— доступа к памяти (распознавателя) (store) 114
— инъективная (injective) 22
— общерекурсивная (total recursive) 41
— переходов (goto) 426—428, 444—446
— переходов (состояний автомата) (state transition) 135
— предшествования (precedence) *7—29*
— преобразования памяти (распознавателя) (fetch) 114
— расстановки (hashing) *274, 275, 277—279*
— рекурсивная (recursive) 41
— слабого предшествования (weak precedence) *16—19, 26, 27*
— характеристическая (characteristic) 48
— частичная (partial) 22
— частично рекурсивная (partial recursive) 41

Функция EFF 433, 444, 450
— FIRST 337, 375, 376, 396—399
— FOLLOW 383, 479; *85, 111*
— GEN 431—443
— GOTO 438—441, 444; *66, 87, 86*
— IN *433—443*
— KGOTO *107, 108*
— NAME *247*
— NEWLABEL *252*
— NEXT *64—76, 78, 82, 86*
— OUT *431—446*
— TRANS *431—443*

Цепочка (string) 27, 106
— выводимая (sentential form) 106; *297*
— левовыводимая (left sentential form) 167
— правовыводимая (right sentential form) 167, 459, 460, 469—471
— пустая (empty) 27
— характеристическая (characteristic) *136*
Цикл (в графе) (cycle, circuit) 54
— в грамматике *см.* Грамматика без циклов
— — программе (loop in program) *393—448*

Часть (portion)
— (левовыводимой цепочки) законченная (closed) 374
— незаконченная (open) 374
— (правовыводимой цепочки) замкнутая (closed) 421
— открытая (open) 421
Число псевдослучайное (pseudorandom number) *278, 288*

Эквивалентность (equivalence) 71, 147—150, 154—156, 162
— автоматов анализирующих (of parsing automata) *125*
— анализаторов (of analyzers) *24, 27, 28, 48*
— — строгая (exact) *19—25, 26, 52—55*
— выражений относительно алгебраических законов (of expressions under algebraic laws) *352*
— деревьев относительно алгебраических законов (of trees under algebraic laws) *377*
— линейных участков (of straight-line blocks) *332*
— структурная (structure) *164*
Элемент несущественный (don't care) *48, 114*
— перевода (translation element) 246—248; *237*

Ядро множества LR(*k*)-ситуаций (core of a set of LR(*k*) items) *95*
Язык (language) 28, 103, 104, 116, 136, 195 *см. также* Множество
— ассемблера (assembly) 82—88; *197, 347—351, 363, 364*
— бесконтекстный (context-free) *см.* Язык контекстно-свободный
— детерминированный (deterministic) 211, 216, 229—231, 238, 241, 283, 384, 450, 501, 503, 522—525
— Дика (Dyck) 239
— естественный (natural) 72, 97, 98, 102, 317, 351
— контекстно-зависимый (context-sensitive) 111, 112, 117, 119—121, 237, 452
— контекстно-свободный (context-free) 111, 112, 117, 119, 121, 237
— левых разборов (left parse) 307, 311

Язык линейный (linear) 191, 237, 268
— LL 376
— LC(k) 402—408
— LL(k) 373—408, 449, 450
— LR(k) *см.* Язык детерминированный *и* LR(k)-грамматика
— неоднозначный (ambiguous) 234—236, 238, 239, 241
— нисходящего разбора с ограниченными возвратами (ЯНРОВ) (top-down parsing language, TDPL) 512—525, 528—530, 539—542
— — обобщенный (ОЯНРОВ) (generalized, GTDPL) 525—542; *224—232*
— ограниченного контекста (bounded context) 505, 507
— — правого контекста (bounded right context) 481—488, 503—507
— однозначный (unambiguous) 234, 240 *см. также* Грамматика однозначная
— операторного предшествования (operator precedence) 493—497, 503, 504, 507
— (1,1)-ОПК ((1,1)-bounded-right-context) 484, 503
— правых разборов (right parse) 307, 311
— программирования (programming) 42, 69—74, 373 *см. также* Алгол, ПЛ/I, ПЛ 360, Фортран, Снобол
— простого предшествования (simple precedence) 455—463, 474—479, 549, 565; *8, 9, 19—25, 85, 139, 185—189, 192—194*
— расширяемый (extensible) 74, 75, 559—562
— регулярный (regular) *см.* Множество регулярное
— существенно неоднозначный (inherent ambiguous) *29—45 см. также* Язык неоднозначный
— Флойда — Эванса (Floyd-Evans) 411, 497—502, 507, 510
— характеризующий (characterizing) 269—273, 282
ЯНРОВ *см.* Язык нисходящего разбора с ограниченными возвратами

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ 5

7 МЕТОДЫ ОПТИМИЗАЦИИ СИНТАКСИЧЕСКИХ АНАЛИЗАТОРОВ 7

7.1. Функции предшествования 7
7.1.1. Теорема о представлении матрицы 8
7.1.2. Применения к разбору, основанному на операторном предшествовании 14
7.1.3. Функции слабого предшествования 16
7.1.4. Преобразование матриц предшествования 19
Упражнения 25
Замечания по литературе 29

7.2. Оптимизация анализаторов Флойда—Эванса 29
7.2.1. Механическое построение анализаторов Флойда—Эванса для грамматик слабого предшествования 30
7.2.2. Усовершенствование анализаторов Флойда—Эванса 35
Упражнения 42
Замечания по литературе 45

7.3. Преобразования, определенные на множествах LR(k)-таблиц 45
7.3.1. Общее понятие LR(k)-таблицы 47
7.3.2. Эквивалентность множеств таблиц 52
7.3.3. φ-недостижимые множества таблиц 55
7.3.4. Слияние таблиц с помощью совместимых разбиений 59
7.3.5. Отсрочка в обнаружении ошибок 64
7.3.6. Исключение сверток по цепным правилам 76
Упражнения 84
Замечания по литературе 91

7.4. Методы построения LR(k)-анализаторов 91
7.4.1. Простые LR-грамматики 92
7.4.2. Распространение SLR-подхода на грамматики, не являющиеся SLR-грамматиками 97
7.4.3. Расщепление грамматики 102
Упражнения 113
Замечания по литературе 117

7.5. Анализирующие автоматы 117
7.5.1. Канонический анализирующий автомат 117
7.5.2. Расщепление функций состояний 122
7.5.3. Обобщение на LR(k)-анализаторы 129
7.5.4. Обзор содержания главы 134
Упражнения 136
Замечания по литературе 138

8 ТЕОРИЯ ДЕТЕРМИНИРОВАННОГО РАЗБОРА 139

8.1. Теория LL-языков 141
8.1.1. LL- и LR-грамматики 141
8.1.2. LL-грамматики в нормальной форме Грейбах 147
8.1.3. Проблема эквивалентности для LL-грамматик 156
8.1.4. Иерархия LL-языков 159
Упражнения 162
Замечания по литературе 163

8.2. Классы грамматик, порождающие детерминированные языки 164
8.2.1. ДМП-автоматы в нормальной форме и канонические грамматики 164
8.2.2. Простые ССП-грамматики и детерминированные языки 169
8.2.3. ОПК-грамматики, LR-грамматики и детерминированные языки 174
8.2.4. Грамматики расширенного предшествования и детерминированные языки 176
Упражнения 182
Замечания по литературе 185

8.3. Теория языков простого предшествования 185
8.3.1. Класс языков простого предшествования 185
8.3.2. Языки операторного предшествования 188
8.3.3. Обзор содержания главы 192
Упражнения 194
Замечания по литературе 195

9 ПЕРЕВОД И ГЕНЕРАЦИЯ КОДА 196

9.1. Роль перевода в процессе компиляции 196
9.1.1. Фазы компиляции 198
9.1.2. Представления промежуточной программы 200
9.1.3. Модели процесса генерации кода 204
Упражнения 205
Замечания по литературе 206

9.2. Синтаксически управляемые переводы 206
9.2.1. Простые синтаксически управляемые переводы 206
9.2.2. Обобщенный преобразователь 214
9.2.3. Детерминированный однопроходной восходящий перевод 218
9.2.4. Детерминированный однопроходной нисходящий перевод 219
9.2.5. Перевод при наличии возвратов 224
Упражнения 232
Замечания по литературе 236

9.3. Обобщенные схемы переводов 236
9.3.1. Мультисхемы переводов 236
9.3.2. Разновидности переводов 245
9.3.3. Наследственный и синтезированный переводы 253
9.3.4. Замечание о разбиении на фазы 259
Упражнения 261
Замечания по литературе 266

10 ОРГАНИЗАЦИЯ ИНФОРМАЦИИ 267

10.1. Таблицы имен 267
10.1.1. Хранение информации о лексемах 268
10.1.2. Механизмы запоминания 270
10.1.3. Таблицы расстановки 274
10.1.4. Функции расстановки 277
10.1.5. Эффективность таблиц расстановки 279
Упражнения 287
Замечания по литературе 292

10.2. Грамматики свойств 292
10.2.1. Мотивировка 293
10.2.2. Определение грамматики свойств 395
10.2.3. Реализация грамматик свойств 304
10.2.4. Анализ алгоритма обработки таблиц 314
Упражнения 323
Замечания по литературе 326

11 ОПТИМИЗАЦИЯ КОДА 327

11.1. Оптимизации линейного участка 328
11.1.1. Модель линейного участка 328
11.1.2. Преобразования блоков 332
11.1.3. Графическое представление блоков 338
11.1.4. Критерий эквивалентности блоков 342
11.1.5. Оптимизация блоков 345
11.1.6. Алгебраические преобразования 351
Упражнения 357
Замечания по литературе 362

11.2. Арифметические выражения 362
11.2.1. Модель машины 363
11.2.2. Разметка деревьев 366
11.2.3. Программы с командами STORE 373
11.2.4. Влияние некоторых алгебраических законов 376
Упражнения 388
Замечания по литературе 392

11.3. Программы с циклами 393
11.3.1. Модель программы 394
11.3.2. Анализ потока управления 398
11.3.3. Примеры преобразований программ 404
11.3.4. Оптимизация циклов 408
Упражнения 418
Замечания по литературе 422

11.4. Анализ потока данных 423
11.4.1. Интервалы 424
11.4.2. Анализ потока данных с помощью интервалов 431
11.4.3. Несводимые графы управления 439
11.4.4. Обзор содержания главы 443
Упражнения 444
Замечания по литературе 448

Список литературы 449
Указатель лемм, теорем и алгоритмов к тому 2 465
Именной указатель к 1 и 2 томам 467
Предметный указатель к 1 и 2 томам 471