

А. АХО, ДЖ. УЛЬМАН

THE THEORY OF PARSING, TRANSLATION AND COMPIILING

volume 1: PARSING

ALFRED V. AHO

Bell Telephone Laboratories, Inc.
Murray Hill, N. J.

JEFFREY D. ULLMAN

Department of Electrical Engineering
Princeton University

ТЕОРИЯ СИНТАКСИЧЕСКОГО АНАЛИЗА, ПЕРЕВОДА И КОМПИЛЯЦИИ

Том 1
Синтаксический анализ

Перевод с английского
В. Н. Агафонова

Под редакцией
В. М. Курочкина

Prentice-Hall, Inc.
Englewood Cliffs, N. J.
1972

ИЗДАТЕЛЬСТВО «МИР»
МОСКВА 1978

ОТ РЕДАКТОРА ПЕРЕВОДА

Первый том фундаментальной монографии известных американских ученых содержит основной математический аппарат (в частности, теорию грамматик и автоматов), краткий обзор процесса компиляции, начала теории синтаксического управляемого перевода и обстоятельное изложение методов синтаксического анализа. Рассмотрены и систематизированы почти все известные алгоритмы разбора. Для некоторых из них впервыедается полное и корректное описание, для большинства доказывается корректность и оценивается сложность. Приведено большое количество упражнений.

Особенность книги в том, что она трактует теоретические вопросы в связи с потребностями реализации языков программирования, и этим она отличается от книг по системному программированию.

Книга предназначена тем, кто работает в области системного и теоретического программирования, преподает или изучает эти дисциплины, а также математикам, интересующимся приложениями теории грамматик и автоматов.

Перед читателями—перевод фундаментального труда американских ученых А. Ахо и Дж. Ульмана, трактующего широкий круг проблем, связанных с языками программирования и методами их реализации на вычислительных машинах. Основой для него до известной степени послужила большая серия статей по теории языков и перевода, опубликованная авторами в различных научных журналах в течение 4—5 предшествующих лет.

Первый том посвящен теории языков, общим вопросам теории перевода и (главным образом) методам синтаксического анализа контекстно-свободных языков. Во втором томе рассматриваются вопросы, более тесно связанные с реализацией языков, в частности вопросы оптимизации анализаторов и объектного кода. На русском языке уже имеется ряд изданий, касающихся этих тем или даже целиком посвященных им, но книга А. Ахо и Дж. Ульмана сравнительно мало пересекается с ними. Причин этому две. Во-первых, здесь собран и систематизирован очень большой по объему материал — по широте охвата вопросов синтаксического анализа контекстно-свободных языков книга, по-видимому, не имеет себе равных ни в нашей, ни в зарубежной литературе. Во-вторых, авторы сосредоточили основное внимание лишь на тех вопросах, которые уже нашли или могут найти приложения на практике при создании компиляторов и других связанных с языками частей математического обеспечения.

Приятен стиль авторов, значительно облегчающий чтение книги. Введение новых более или менее сложных понятий и концепций начинается, как правило, с неформального их изложения и примеров. При этом делаются намеки, которые позволяют лучше понять следующее затем строгое, аккуратное и формальное построение. Описание многочисленных алгоритмов построено по аналогичной схеме: сначала дается неформальное разъяснение сути их работы, затем четкая формулировка алгоритма, иллюстрируемая часто на примере, и, наконец, доказательство правильности алгоритма. Каждый раздел заканчивается

Редакция литературы по математическим наукам

солидным списком задач различной трудности— от чисто технических и учебных упражнений до нерешенных научных проблем. Благодаря такому характеру изложения эту книгу, содержащую большой и трудный материал, легко читать на любом уровне: поверхностного чтения (только по неформальным и общим описаниям), знакомства с фактическим материалом (определения, теоремы, алгоритмы), глубокого изучения (разбор или воспроизведение доказательств, решение трудных задач). Для свободного владения излагаемым материалом полезен разбор примеров и решение большинства приводимых задач.

В целом книга будет интересной для широкого круга читателей: от студентов, изучающих основы теории языков программирования, до специалистов— разработчиков систем математического обеспечения и научных работников. Вместе с авторами можно надеяться, „что алгоритмы и понятия, изложенные в этой книге, переживут следующее поколение вычислительных машин и языков программирования и что хотя бы некоторые из них найдут применение не только при построении компиляторов, но и в других областях“.

B. M. Курочкин

Посвящается Адриенне и Холли

ПРЕДИСЛОВИЕ

Эта книга задумана как пособие для одно- или двухсеместрового курса лекций по теории компиляции для студентов старших курсов. В ней дается теоретическая трактовка предмета, имеющего практическое значение. При ее написании мы исходили из следующих трех соображений.

(1) В преподавании такой быстро развивающейся области, какой является наука о вычислительных процессах, правильный педагогический принцип состоит в том, чтобы больше внимания уделять идеям, а не техническим подробностям реализации. Мы надеемся, что алгоритмы и понятия, изложенные в этой книге, переживут следующее поколение вычислительных машин и языков программирования и что хотя бы некоторые из них найдут применение не только при построении компиляторов, но и в других областях.

(2) Прогресс в построении компиляторов достиг того этапа, когда компилятор можно расчленить на много составных частей и каждую часть подвергнуть запланированной оптимизации. Поэтому важно снабдить людей, предпринимающих попытки такой оптимизации, подходящими математическими средствами.

(3) Для полного понимания некоторых из самых полезных и самых эффективных алгоритмов компиляции (например, алгоритма LR (k)-анализа) требуется основательная математическая подготовка. Мы полагаем, что хорошая теоретическая подготовка становится все более существенной для тех, кто строит компиляторы.

Включая в книгу трудные теоремы, имеющие отношение к компиляции, мы старались сделать изложение как можно более доступным. Для этого приводится много примеров, причем в каждом из них используется какая-нибудь малая грамматика, а не те большие грамматики, что встречаются на практике. Мы надеемся, что в тех случаях, когда трудно следить за теоретическими построениями, для иллюстрации основных идей этих примеров будет достаточно.

О пользовании книгой

Книга возникла из записей лекций, прочитанных на старших курсах Принстонского университета и Стивенсовского технологического института. По ним читались как односеместровый курс, так и двухсеместровый. В первом случае курсу по теории компиляции предшествовал курс теории конечных автоматов и контексто-свободных языков, поэтому не было необходимости излагать материал гл. 0, 2 и 8. Остальные же главы излагались подробно.

В случае двухсеместрового курса большая часть материала первого тома излагалась в первом семестре, а большая часть второго, исключая гл. 8,— во втором. При этом доказательствам и технике доказательств уделялось больше внимания, чем в односеместровом курсе.

Ясно, что одни разделы книги более важны, чем другие. Поэтому нам хочется кратко пояснить читателю, как мы оцениваем относительную важность различных частей первого тома. Общее замечание состоит в том, что большинство доказательств, по-видимому, можно пропустить. Мы включили доказательства всех главных результатов потому, что считаем их необходимыми для глубокого понимания предмета. Однако в курсах, посвященных компиляции, обычно предпочитают не особению углубляться во многие вопросы, причем разумный уровень понимания достигается при довольно поверхностном знакомстве с доказательствами.

В гл. 0 (математические основы) и 1 (обзор компиляции) почти весь материал существен, за исключением, быть может, разд. 1.3, в котором рассматриваются приложения синтаксического анализа, не связанные с компиляцией.

Мы считаем, что каждое понятие и теорема, введенные в гл. 2 (теория языков), найдут применение где-нибудь в остальных девяти главах. Однако в курсе лекций по компиляторам некоторый материал следует опустить. Подходящим кандидатом для этого служит довольно трудный материал об уравнениях с регулярными коэффициентами из разд. 2.2.1. Придется опустить тогда часть материала из разд. 2.2.2, касающегося праволинейных грамматик (а результат об эквивалентности между ними и конечными автоматами вывести другим способом), и материал из разд. 2.4.5 о преобразовании грамматики в грамматику в нормальной форме Грейбах методом Розенкранца.

Понятия, излагаемые в гл. 3 (перевод), очень важны для остальной части книги. Однако разд. 3.2.3 об иерархии синтаксически управляемых переводов довольно труден и его можно опустить.

Мы думаем, что разд. 4.1 о методах разбора с возвратами

менее важен, чем разд. 4.2, в котором рассматриваются табличные методы.

Глава 5 (однопроходный синтаксический анализ) большей частью очень важна. Максимальное предпочтение мы предлагаем отдать LL-грамматикам (разд. 5.1), LR-грамматикам (разд. 5.2), грамматикам предшествования (разд. 5.3.2 и 5.3.4) и грамматикам операторного предшествования (разд. 5.4.3). Другие разделы при необходимости можно опустить.

Глава 6 (алгоритмы с возвратами) менее важна, чем большая часть гл. 5 или разд. 4.2. Если надо выбирать, то мы предпочли бы изложить разд. 6.1, а не 6.2.

Организация книги

Вся книга состоит из двух томов:

I. Синтаксический анализ (гл. 0—6) и

II. Компиляция (гл. 7—11). (Во втором томе рассматриваются оптимизация анализаторов, теория детерминированного разбора, перевод, работа с таблицами и оптимизация кода.)

В конце каждого раздела (с номером i, j) приводятся упражнения, проблемы и замечания по литературе. Проблемы делятся на открытые и предлагаемые для дальнейшего исследования, а в упражнениях звездочками указывается степень трудности. Для решения упражнения, помеченного одной звездочкой, требуется одна существенная догадка, а для упражнения с двумя звездочками — более чем одна.

Чтение курса по этой книге рекомендуется сопровождать лабораторными работами по программированию, в ходе которых должны быть спроектированы и реализованы какие-то части компилятора. В конце некоторых разделов книги приведены упражнения на программирование, которые можно использовать в этих лабораторных работах.

Благодарности

Многие люди внимательно прочли различные части рукописи и серьезно помогли нам при ее подготовке к печати. Мы особенно хотим поблагодарить Джона Бруно, Стефена Чена, Джеймса Гимпеля, Жана Ихбия, Брайана Кернигана, Дугласа Мак-Илроя, Роберта Мартина и Роберта Морриса, а также рецензентов Томаса Читэма, Майкла Фишера и Уильяма Мак-Кимана. Важные замечания сделали многие студенты, пользовавшиеся нашими записями лекций, среди них Алан Демерс, Нахед Эль Джабри, Мэттью Хехт, Петер Хендerson, Петер Майка, Томас Петерсон, Рави Сети, Кеннет Силлз и Стивен Сквайрз.

Мы благодарны также Ханне Крессе и Дороти Лючиани за то, что они великолепно напечатали рукопись. Кроме того, мы выражаем признательность лабораториям компании „Белл телефон“ за содействие при подготовке рукописи. Она была ускорена с помощью UNIX, операционной системы для вычислительной машины PDP-11, разработанной Деннисом Ричи и Кеннетом Томпсоном.

*Альфред В. Ахо
Джефри Д. Ульман*

0 ПРЕДВАРИТЕЛЬНЫЕ МАТЕМАТИЧЕСКИЕ СВЕДЕНИЯ

Чтобы говорить ясно и точно, нам нужен точный и правильный язык. В этой главе описывается язык, которым мы будем пользоваться, обсуждая вопросы синтаксического анализа, трансляции и другие предметы, содержащиеся в нашей книге. Этот язык является главным образом языком элементарной теории множеств, к которому добавлены некоторые первоначальные понятия теории графов и математической логики. Читатели, знакомые с основами этих областей математики, могут только бегло просмотреть главу и использовать ее как справочник обозначений и определений.

0.1. ОСНОВНЫЕ ПОНЯТИЯ ТЕОРИИ МНОЖЕСТВ

В этом разделе будет сделан краткий обзор некоторых из самых основных понятий теории множеств, таких, как отношения, функции, упорядочения, а также обычные операции над множествами.

0.1.1. Множества

В дальнейшем мы будем предполагать, что существуют объекты, называемые *атомами*. Этим словом обозначается первоначальное понятие,— иначе говоря, термин „атом“ остается не определенным. Что называть атомом, зависит от рассматриваемой области. Часто бывает удобно считать атомами целые числа или буквы некоторого алфавита.

Мы будем также постулировать абстрактное понятие *принадлежности*. Если a принадлежит A , то пишут $a \in A$. Отрицание этого утверждения записывается так: $a \notin A$. Предполагается, что если a —атом, то ему ничто не принадлежит, т. е. $x \notin a$ для всех x из рассматриваемой области.

Будут также использоваться некоторые примитивные объекты, называемые *множествами*, которые не являются атомами. Если

A —множество, то его элементы—это те объекты a (не обязательно атомы), для которых $a \in A$. Каждый элемент множества представляет собой либо атом, либо другое множество. Предполагается, что каждый элемент множества появляется в нем точно один раз. Если A содержит конечное число элементов, то A называется *конечным множеством*, и часто пишут $A = \{a_1, a_2, \dots, a_n\}$, если a_1, \dots, a_n —все элементы множества A и $a_i \neq a_j$ для $i \neq j$. Заметим, что порядок элементов не играет роли. Можно было бы, например, написать $A = \{a_n, \dots, a_1\}$. Мы резервируем символ \emptyset для обозначения *пустого множества*, т. е. множества, в котором нет элементов. Заметим, что атом тоже не имеет элементов, но пустое множество не атом и атом не является пустым множеством.

Утверждение $\#A = n$ означает, что множество A имеет n элементов.

Пример 0.1. Пусть атомами будут неотрицательные целые числа. Тогда $A = \{1, \{2, 3\}, 4\}$ —множество. Элементами A служат 1, {2, 3} и 4. Элемент {2, 3} множества A сам является множеством, состоящим из атомов 2 и 3. Однако атомы 2 и 3 не принадлежат множеству A . Можно писать $A = \{4, 1, \{3, 2\}\}$. Заметим, что $\#A = 3$. \square

Один из полезных способов определения множества—определение с помощью *предиката*, т. е. утверждения, содержащего одно или несколько неизвестных и принимающего в зависимости от значений неизвестных одно из двух значений—*истина* или *ложь*. Множество, определяемое с помощью предиката, состоит в точности из тех элементов, для которых предикат истинен. Однако надо быть осторожным при выборе предиката для определения множества, иначе может оказаться, что мы пытаемся определить множество, которое, возможно, и не существует.

Пример 0.2. Только что отмеченное явление называется *парадоксом Рассела*. Пусть $P(X)$ —предикат „ X не является элементом самого себя“, т. е. $X \notin X$. Тогда мы могли бы подумать, что можно определить множество Y всех тех X , для которых $P(X)$ истинно, т. е. Y состоит в точности из тех множеств, которые не являются элементами самих себя. Так как большинство обычных множеств не являются элементами самих себя, возникает искушение допустить, что множество Y существует.

Но если Y существует, мы должны суметь ответить на вопрос: „Является ли Y элементом самого себя?“ А это приводит к невозможной ситуации. Если $Y \in Y$, то $P(Y)$ ложно, и Y не является элементом самого себя по определению Y . Отсюда невозможно, чтобы $Y \in Y$. Допустим наоборот, что $Y \notin Y$. Тогда по определению Y снова $Y \in Y$. Мы видим, что $Y \notin Y$ влечет $Y \in Y$, а $Y \in Y$

влечет $Y \notin Y$. Так как либо $Y \in Y$, либо $Y \notin Y$ истинно, то оба эти утверждения истинны—ситуация, которую мы считаем невозможной. Единственный выход из положения состоит в том, чтобы предположить, что Y не существует. \square

Обычный способ избежать парадокса Рассела заключается в том, чтобы определять множества только с помощью предикатов $P(X)$ вида „ X принадлежит A и $P_1(X)$ “, где A —известное множество, а P_1 —произвольный предикат. Если множество A подразумевается, то мы будем вместо „ X принадлежит A и $P_1(X)$ “ писать просто $P_1(X)$.

Если $P(X)$ —предикат, будем обозначать множество объектов X , для которых $P(X)$ истинно, через $\{X | P(X)\}$.

Пример 0.3. Пусть $P(X)$ —предикат „ X —неотрицательное четное число“, т. е. $P(X)$ имеет вид „ X принадлежит множеству неотрицательных целых чисел и $P_1(X)$ “, где $P_1(X)$ —предикат „ X четно“. Тогда $A = \{X | P(X)\}$ будет множеством, которое часто записывают так: $\{0, 2, 4, \dots, 2n, \dots\}$. Если по ходу дела ясно, что речь идет о множестве неотрицательных целых чисел, то можно писать $A = \{X | X \text{ четно}\}$. \square

Мы не останавливаемся здесь подробно на аксиоматической теории множеств. Интересующемуся читателю рекомендуем книги Халмоса [1960] и Суллеса [1960] (см. список литературы).

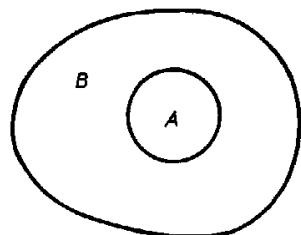
Определение. Говорят, что множество A *содержится* в множестве B , и пишут $A \subseteq B$, если каждый элемент из A является элементом из B . Иногда в этом случае говорят, что B *содержит* (или *включает*) A , и пишут $B \supseteq A$. Говорят также, что A —*подмножество* B , а B —*надмножество* A .

Если B содержит¹⁾ элемент, не принадлежащий A , и $A \subseteq B$, то говорят, что A *собственно содержится* в B , и пишут $A \subset B$ (или что B *собственно включает* A , и пишут $B \supset A$). Можно также сказать, что A —*собственное подмножество* B или что B —*собственное надмножество* A .

Два множества A и B называются *равными*, если $A \subseteq B$ и $B \subseteq A$.

Для того чтобы графически изобразить включение множеств, часто пользуются так называемыми *диаграммами Венна*. На рис. 0.1 показана диаграмма Венна для отношения $A \subseteq B$.

¹⁾ Русский термин „содержит“ (и его производные) обозначает в силу традиции два разных понятия: множество B *содержит* множество A , т. е. $B \supseteq A$, или $A \subseteq B$, и множество B *содержит элемент* b , т. е. $b \in B$. Из контекста каждый раз ясно, о чем идет речь, и можно надеяться, что у читателя трудностей по этой причине не возникнет.—Прим. ред.

Рис. 0.1. Диаграмма Венна для включения множеств: $A \subseteq B$.

0.1.2. Операции над множествами

Существует несколько основных операций над множествами, с помощью которых можно строить новые множества.

Определение. Пусть A и B —множества. Объединением множеств A и B (записывается $A \cup B$) называется множество, содержащее все элементы из A вместе со всеми элементами из B . Формально $A \cup B = \{x \mid x \in A \text{ или } x \in B\}$ ¹⁾.

Пересечением множеств A и B (записывается $A \cap B$) называется множество, состоящее из всех тех элементов, которые принадлежат обоим множествам A и B . Формально $A \cap B = \{x \mid x \in A \text{ и } x \in B\}$.

Разностью множеств A и B (записывается $A - B$) называется множество тех элементов из A , которые не принадлежат B .

Если A —множество всех элементов, рассматриваемых в данной ситуации (иногда его называют *универсальным* и обозначают через U), то разность $U - B$ часто обозначается \bar{B} и называется дополнением множества B .

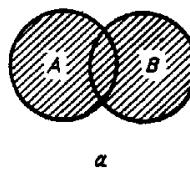
Заметим, что мы говорим об универсальном множестве как о множестве всех элементов, „рассматриваемых в данной ситуации“. При этом мы должны быть уверены в том, что U существует. Например, если взять в качестве U „множество всех множеств“, то снова получится парадокс Рассела. Заметим также, что \bar{B} не определено, если не ясно, по отношению к какому универсальному множеству рассматривается операция дополнения.

1) Заметим, что существование множества $A \cup B$ не гарантировано, так что возможность определения с помощью предиката вызывает сомнение. В аксиоматической теории множеств существование множества $A \cup B$ принимается за аксиому.

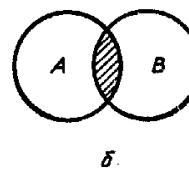
Вообще $A - B = A \cap \bar{B}$. Диаграммы Венна для этих операций над множествами показаны на рис. 0.2.

Если $A \cap B = \emptyset$, то говорят, что A и B не пересекаются.

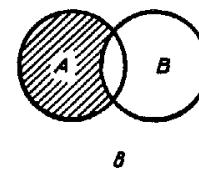
Определение. Пусть I —некоторое множество, элементы которого используются как индексы, и для каждого $i \in I$ множество A_i уже известно. Через $\bigcup_{i \in I} A_i$ обозначим множество $\{X \mid \text{существует такое } i \in I, \text{ что } X \in A_i\}$. Так как I может не быть конечным, то это определение обобщает определение объединения двух



a



б



в

 $A \cup B$ $A \cap B$ $A - B$

Рис. 0.2. Диаграммы Венна для операций над множествами.

множеств. Если множество I определено с помощью предиката $P(i)$, то иногда пишут $\bigcup_{P(i)} A_i$ вместо $\bigcup_{i \in I} A_i$. Например, $\bigcup_{i > 2} A_i$ означает $A_3 \cup A_4 \cup A_5 \cup \dots$.

Определение. Пусть A —множество. Множество всех подмножеств множества A будем обозначать через $\mathcal{P}(A)$ или 2^A , т. е. $\mathcal{P}(A) = \{B \mid B \subseteq A\}$ ¹⁾.

Пример 0.4. Пусть $A = \{1, 2\}$. Тогда $\mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$. Другой пример: $\mathcal{P}(\emptyset) = \{\emptyset\}$. \square

Вообще, если A —конечное множество, состоящее из m элементов, то $\mathcal{P}(A)$ состоит из 2^m элементов. Пустое множество является элементом множества $\mathcal{P}(A)$ для любого A .

Мы уже отмечали, что элементы множества считаются неупорядоченными. При некоторых обстоятельствах удобно рассматривать упорядоченные пары объектов. Поэтому дадим следующее определение.

1) Существование множества всех подмножеств для любого множества — аксиома теории множеств. Другими аксиомами теории множеств, в дополнение к этой и к ранее упомянутой аксиоме об объединении, являются следующие:

- (1) Если A —множество и P —предикат, то $\{X \mid P(X) \text{ и } X \in A\}$ —множество.
- (2) Если X —атом или множество, то $\{X\}$ —множество.
- (3) Если A —множество, то $\{X \mid \text{существует } Y, \text{ для которого } X \in Y \text{ и } Y \in A\}$ —множество.

Определение. Пусть a и b —объекты. Через (a, b) обозначим упорядоченную пару, состоящую из объектов a и b , взятых в этом порядке. Упорядоченные пары (a, b) и (c, d) называются равными, если $a=c$ и $b=d$. В противоположность этому $\{a, b\}=\{b, a\}$.

Упорядоченные пары можно рассматривать как множества, если определить (a, b) как множество $\{a, \{a, b\}\}$. Мы оставляем в качестве упражнения доказательство того, что $\{a, \{a, b\}\}=\{c, \{c, d\}\}$ тогда и только тогда, когда $a=b$ и $c=d$. Таким образом, это определение согласуется с тем, что можно считать фундаментальным свойством упорядоченных пар.

Определение. Декартовым произведением множеств A и B , обозначаемым через $A \times B$, называют множество $\{(a, b) \mid a \in A$ и $b \in B\}$.

Пример 0.5. Пусть $A=\{1, 2\}$ и $B=\{2, 3, 4\}$. Тогда

$$A \times B = \{(1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4)\} \quad \square$$

0.1.3. Отношения

Многие распространенные математические понятия, такие, как принадлежность, включение множеств, числовое неравенство, являются отношениями. Мы дадим формальное определение понятия отношения и посмотрим, как под это определение подходят известные примеры отношений.

Определение. Пусть A и B —множества. Отношением из A в B называется любое подмножество множества $A \times B$. Если $A=B$, то говорят, что отношение задано, или определено, на A (или просто, что это—отношение на множестве A). Если R —отношение из A в B и $(a, b) \in R$, то пишут aRb . Множество A называют областью определения отношения R , а множество B —множеством его значений.

Пример 0.6. Пусть A —множество целых чисел. Отношение $<$ представляет собой множество $\{(a, b) \mid a$ меньше $b\}$. Как и следовало ожидать, для таких пар (a, b) мы будем писать $a < b$. \square

Определение. Отношение $\{(b, a) \mid (a, b) \in R\}$ называется обратным к отношению R и часто обозначается через R^{-1} .

Понятие отношения очень общее. Часто отношение обладает рядом свойств, для которых установлены специальные названия.

Определение. Пусть A —множество и R —отношение на A . Отношение R называется

- (1) рефлексивным, если aRa для всех a из A ,
- (2) симметричным, если aRb влечет bRa для a и b из A ,
- (3) транзитивным, если aRb и bRc влечут aRc для a , b и c из A . Элементы a , b и c не обязаны быть различными.

Рефлексивное, симметричное и транзитивное отношение называется *отношением эквивалентности*.

Важное свойство любого отношения эквивалентности R , определенного на множестве A , заключается в том, что оно разбивает множество A на непересекающиеся подмножества, называемые *классами эквивалентности*. Для каждого элемента $a \in A$ обозначим через $[a]$ класс эквивалентности, содержащий a , т. е. множество $\{b \mid aRb\}$.

Пример 0.7. Рассмотрим отношение сравнения по модулю N , определенное на множестве неотрицательных целых чисел. Говорят, что a сравнимо с b по модулю N , и пишут $a \equiv b \pmod N$, если существует такое целое число k , что $a-b=kN$. Пусть, например, $N=3$. Тогда множество $\{0, 3, 6, \dots, 3n, \dots\}$ будет одним из классов эквивалентности, поскольку $3n \equiv 3m \pmod 3$ для любых целых чисел m и n . Этот класс обозначим через $[0]$; можно взять также $[3]$, или $[6]$, или $[3n]$, поскольку любой элемент класса эквивалентности является представителем этого класса.

Другие два класса эквивалентности отношения сравнения по модулю 3 таковы:

$$\begin{aligned}[1] &= \{1, 4, 7, \dots, 3n+1, \dots\} \\ [2] &= \{2, 5, 8, \dots, 3n+2, \dots\}\end{aligned}$$

Объединение трех множеств $[0]$, $[1]$ и $[2]$ совпадает с множеством всех неотрицательных целых чисел. Таким образом, это отношение эквивалентности разбивает множество всех неотрицательных целых чисел на три непересекающиеся класса эквивалентности $[0]$, $[1]$ и $[2]$ (рис. 0.3). \square

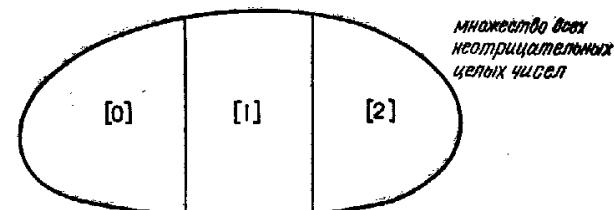


Рис. 0.3. Классы эквивалентности отношения сравнения по модулю 3.

Индексом отношения эквивалентности, определенного на множестве A , называется число классов эквивалентности, на которые разбивается множество A этим отношением.

В качестве упражнения предлагаем доказать следующую теорему об отношениях эквивалентности:

Теорема 0.1. Пусть R — отношение эквивалентности на A . Тогда для всех a и b из A либо $[a]=[b]$, либо $[a]$ и $[b]$ не пересекаются.

Доказательство. Упражнение. \square

0.1.4. Замыкание отношений

Для данного отношения R часто бывает нужно найти другое отношение R' , включающее R и обладающее некоторыми дополнительными свойствами, например транзитивностью. Более того, обычно хочется, чтобы R' было как можно „меньше“, т. е. чтобы оно было подмножеством любого другого отношения, включающего R и обладающего желаемыми свойствами. Конечно, это „наименьшее“ отношение может определяться неоднозначно, если дополнительные свойства какие-нибудь странные. Однако для тех распространенных свойств, которые упоминались в предыдущем разделе, часто можно однозначно найти наименьшее надмножество данного отношения, обладающее этими свойствами. Далее мы рассмотрим некоторые частные случаи.

Определение. k -я степень отношения R на множестве A (обозначаемая R^k), определяется следующим образом:

- (1) aR^4b тогда и только тогда, когда aRb ;
- (2) $aR^i b$ для $i > 1$ тогда и только тогда, когда существует такое $c \in A$, что aRc и $cR^{i-1}b$.

Это пример рекурсивного определения; таким методом определения мы будем пользоваться много раз. Чтобы уяснить себе рекурсивный аспект этого определения, допустим, что aR^4b . Тогда, по (2), существует такое c_1 , что aRc_1 и c_1R^3b . Снова применяя (2), видим, что существует такое c_2 , что c_1Rc_2 и c_2R^2b . Еще одно применение (2) говорит о том, что существует такое c_3 , что c_2Rc_3 и c_3R^1b . Теперь можно применить (1) и убедиться, что c_3Rb .

Таким образом, если aR^4b , то существует такая последовательность элементов c_1, c_2, c_3 из A , что aRc_1, c_1Rc_2, c_2Rc_3 и c_3Rb .

Транзитивное замыкание отношения R на множестве A обозначается через R^+ и определяется так: aR^+b тогда и только тогда, когда $aR^i b$ для некоторого $i \geq 1$. Мы увидим, что R^+ — наименьшее транзитивное отношение, включающее R .

Можно было бы иначе определить R^+ , сказав, что aR^+b , если существует такая последовательность c_1, c_2, \dots, c_n , состоящая из

нуля или более элементов, принадлежащих A , что $aRc_1, c_1Rc_2, \dots, c_{n-1}Rc_n, c_nRb$. Если $n=0$, то полагаем aRb .

Рефлексивное и транзитивное замыкание отношения R на множестве A обозначается через R^* и определяется следующим образом:

- (1) aR^*a для всех $a \in A$;
- (2) aR^*b , если aR^+b ;
- (3) в R^* нет ничего другого, кроме того, что содержится там в силу (1) или (2).

Если определить R^0 , сказав, что aR^0b тогда и только тогда, когда $a=b$, то aR^0b тогда и только тогда, когда aR^ib для некоторого $i \geq 0$.

Единственное различие между R^+ и R^* состоит в том, что aR^*a истинно для всех $a \in A$, но aR^+a может быть, а может и не быть истинным. R^* — это наименьшее рефлексивное и транзитивное отношение, включающее R .

В разд. 0.5.8 мы изучим методы эффективного построения рефлексивного и транзитивного замыкания отношения. А здесь докажем, что R^+ и R^* — наименьшие надмножества отношения R , обладающие нужными свойствами.

Теорема 0.2. Если R^+ и R^* — соответственно транзитивное и рефлексивно-транзитивное замыкание отношения R , то

- (a) R^+ транзитивно и если R' — любое транзитивное отношение, для которого $R \subseteq R'$, то $R^+ \subseteq R'$;
- (б) R^* рефлексивно и транзитивно и если R' — любое рефлексивное и транзитивное отношение, для которого $R \subseteq R'$, то $R^* \subseteq R'$.

Доказательство. Докажем только (а); утверждение (б) оставим в качестве упражнения. Во-первых, чтобы показать, что R^+ транзитивно, нужно показать, что если aR^+b и bR^+c , то aR^+c . Так как aR^+b , существует такая последовательность элементов d_1, \dots, d_n , что $d_1Rd_2, \dots, d_{n-1}Rd_n$, где $d_1=a$ и $d_n=b$. Так как bR^+c , то можно найти такие e_1, \dots, e_m , что $e_1Re_2, \dots, e_{m-1}Re_m$, где $e_1=b=d_n$ и $e_m=c$. Отсюда, используя определение R^+ , заключаем, что aR^+c .

Теперь покажем, что R^* — наименьшее транзитивное отношение, включающее R . Пусть R' — любое транзитивное отношение, для которого $R \subseteq R'$. Надо показать, что $R^* \subseteq R'$. Пусть $(a, b) \in R^*$, т. е. aR^+b . Тогда существует такая последовательность c_1, \dots, c_n , что $a=c_1, b=c_n$ и c_iRc_{i+1} для $1 \leq i < n$. Так как $R \subseteq R'$, получаем $c_iR'c_{i+1}$ для $1 \leq i < n$. Так как R' транзитивно, повторное применение определения транзитивности дает

$c_1 R' c_n$, т. е. $aR'b$. Поскольку (a, b) — произвольный элемент из R^+ , мы показали, что каждый элемент из R^+ принадлежит R' . Таким образом, $R^+ \subseteq R'$, что и требовалось доказать. \square

0.1.5. Отношения порядка

Важный класс отношений образуют отношения порядка. Вообще говоря, порядок на множестве A — это любое транзитивное отношение на A . При изучении алгоритмов важную роль играет специальный тип порядка, называемый частичным. Множество, на котором задан какой-нибудь порядок, называется *упорядоченным*.

Определение. Частичным порядком на множестве A называется отношение R , определенное на A и такое, что

- (1) R транзитивно,
- (2) для всех $a \in A$ утверждение aRa ложно¹⁾, т. е. отношение R *иррефлексивно*.

Из свойств (1) и (2) следует, что если aRb истинно, то bRa ложно. Это свойство называется *асимметричностью*.

Пример 0.8. Примером частичного порядка служит собственное включение множеств. Например, пусть $S = \{e_1, \dots, e_n\}$ — множество, состоящее из n элементов, и пусть $A = \mathcal{P}(S)$. Положим aRb для любых a, b из A тогда и только тогда, когда $a \subseteq b$. Отношение R является частичным порядком.

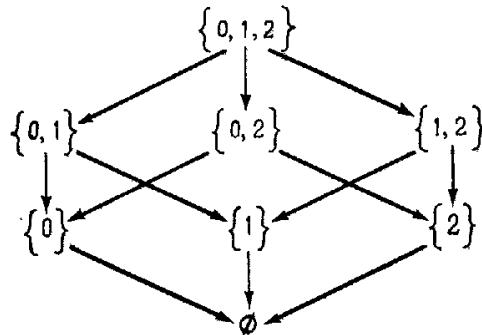


Рис. 0.4. Частичный порядок.

На рис. 0.4 графически изображен этот частичный порядок для случая $S = \{0, 1, 2\}$. Множество S_1 собственно включает S_2 , тогда и только тогда, когда существует направленный вниз путь, ведущий из S_1 в S_2 . \square

¹⁾ Под „ aRb ложно“ понимается, что $(a, b) \notin R$.

В литературе частичным порядком иногда называют то, что мы называем рефлексивным частичным порядком.

Определение. Рефлексивным частичным порядком на A называется отношение R , обладающее следующими свойствами:

- (1) R транзитивно,
- (2) R рефлексивно,
- (3) если aRb и bRa , то $a = b$.

Последнее свойство называют *антисимметричностью*.

Примером рефлексивного частичного порядка служит (не обязательно собственное) включение множеств.

В разд. 0.5 мы покажем, что каждый частичный порядок можно представить графически в виде структуры, называемой ориентированным ациклическим графом.

Важный частный случай частичного порядка — линейный порядок.

Определение. Линейный порядок R на множестве A — это такой частичный порядок, что если a и b принадлежат A , то либо aRb , либо bRa , либо $a = b$. Если A — конечное множество, то линейный порядок R удобно представлять себе, считая все элементы множества A расположеными в виде последовательности a_1, a_2, \dots, a_n , для которой a_iRa_j тогда и только тогда, когда $i < j$.

Аналогично можно определить рефлексивный линейный порядок, а именно R — рефлексивный линейный порядок на A , если R — такой рефлексивный частичный порядок, что aRb или bRa для всех a и b из A .

Например, отношение $<$ (меньше), определенное на множестве неотрицательных целых чисел, является линейным порядком. Примером рефлексивного линейного порядка может служить отношение \leqslant .

0.1.6. Отображения

Важный класс отношений, которым мы будем пользоваться, образуют отображения.

Определение. Отображением (а также функцией или преобразованием) M множества A в множество B называется такое отношение из A в B , что если (a, b) и (a, c) принадлежат M , то $b = c$.

Если $(a, b) \in M$, то обычно пишут $M(a) = b$. Говорят, что $M(a)$ определено, если существует такое $b \in B$, что $(a, b) \in M$. Если $M(a)$ определено для всех $a \in A$, то говорят, что M *всюду определено*. Если хотят подчеркнуть, что M может быть опреде-

лено не для всех $a \in A$, то говорят, что M —*частичное отображение* (функция) множества A в множество B . В любом случае пишут $M: A \rightarrow B$. Множества A и B называются соответственно *областью определения* и *множеством значений* M .

Если отображение $M: A \rightarrow B$ таково, что для каждого $b \in B$ существует не более одного такого $a \in A$, что $M(a) = b$, то M называется *инъективным* (взаимно однозначным). Если отображение M всюду определено на A и для каждого $b \in B$ существует точно одно такое $a \in A$, что $M(a) = b$, то M называется *биективным*.

Если отображение $M: A \rightarrow B$ инъективное, то можно найти *обратное отображение* $M^{-1}: B \rightarrow A$, для которого $M^{-1}(b) = a$ тогда и только тогда, когда $M(a) = b$. Если существует $b \in B$, для которого в A не найдется такого a , что $M(a) = b$, то M^{-1} будет частичной функцией.

Понятие биективного отображения используется для определения *мощности* множества, которая, говоря неформально, означает число элементов этого множества.

Определение. Два множества A и B называются *равномощными*, если существует биективное отображение из A в B .

Пример 0.9. Множества $\{0, 1, 2\}$ и $\{a, b, c\}$ равномощны. Для доказательства этого утверждения можно взять, например, биективное отображение $M = \{(0, a), (1, b), (2, c)\}$. Множество целых чисел имеет ту же мощность, что и множество четных чисел, хотя последнее является собственным подмножеством первого. Это легко доказать с помощью биективного отображения $\{(i, 2i) | i \text{ — целое число}\}$. \square

Теперь можно точно определить, что мы понимаем под конечным множеством и бесконечным множеством¹⁾.

Определение. Множество S *конечно*, если оно равномощно множеству $\{1, 2, \dots, n\}$ для некоторого целого числа n . Множество *бесконечно*, если оно равномощно некоторому своему собственному подмножеству. Множество *счетно*, если оно равномощно множеству положительных целых чисел. (Из примера 0.9 следует, что каждое счетное множество бесконечно.) Бесконечное множество, которое не является счетным, называется *несчетным*.

Примеры счетных множеств:

(1) Множество всех положительных и отрицательных целых чисел.

¹⁾ Ранее мы употребляли эти термины, предполагая, разумеется, что их интуитивный смысл ясен. Однако целесообразно дать формальные определения.

- (2) Множество четных чисел.
(3) $\{(a, b) | a \text{ и } b \text{ целые}\}$.

Примеры несчетных множеств:

- (1) Множество вещественных чисел.
(2) Множество всех отображений целых чисел в целые.
(3) Множество всех подмножеств множества положительных целых чисел.

УПРАЖНЕНИЯ

0.1.1. Пусть $A = \{0, 1, 2, 3, 4, 5, 6\}$. Запишите множества, определяемые следующими предикатами:

- (a) $\{X | X \text{ принадлежит } A \text{ и } X \text{ четно}\}$,
(b) $\{X | X \text{ принадлежит } A \text{ и является квадратом}\}$,
(в) $\{X | X \text{ принадлежит } A \text{ и } X \geqslant X^2 + 1\}$.

0.1.2. Пусть $A = \{0, 1, 2\}$ и $B = \{0, 3, 4\}$. Запишите множества

- (a) $A \cup B$,
(б) $A \cap B$,
(в) $A - B$,
(г) $\mathcal{P}(A)$,
(д) $A \times B$.

0.1.3. Покажите, что если A —множество, состоящее из n элементов, то $\mathcal{P}(A)$ состоит из 2^n элементов.

0.1.4. Пусть A и B —множества, а U —некоторое универсальное множество, по отношению к которому берутся дополнения. Покажите, что

- (а) $\overline{A \cup B} = \overline{A} \cap \overline{B}$,
(б) $\overline{A \cap B} = \overline{A} \cup \overline{B}$.

Эти тождества называются *законами де Моргана*.

0.1.5. Покажите, что не существует такого множества U , что $A \in U$ для всех множеств A . Указание: Рассмотрите парадокс Рассела.

0.1.6. Дайте пример отношения, которое

- (а) рефлексивно, но не симметрично и не транзитивно;
(б) симметрично, но не рефлексивно и не транзитивно;
(в) транзитивно, но не рефлексивно и не симметрично.

В каждом случае укажите множество, на котором определено отношение.

0.1.7. Приведите пример отношения на множестве, которое

- (а) рефлексивно и симметрично, но не транзитивно;
(б) рефлексивно и транзитивно, но не симметрично;
(в) симметрично и транзитивно, но не рефлексивно.

Предостережение: Вы заблуждаетесь, если считаете, что симметричное и транзитивное отношение обязано быть рефлексивным (так как aRb и bRa влечет aRa).

0.1.8. Покажите, что следующие отношения являются отношениями эквивалентности:

$$(a) \{(a, a) | a \in A\},$$

(б) отношение конгруэнтности, заданное на множестве треугольников.

0.1.9. Пусть R — отношение эквивалентности на множестве A .

Пусть a и b — элементы из A . Покажите, что

$$(a) [a] = [b] \text{ тогда и только тогда, когда } aRb,$$

$$(b) [a] \cap [b] = \emptyset \text{ тогда и только тогда, когда } aRb \text{ ложно.}$$

0.1.10. Пусть A — конечное множество. Какие отношения эквивалентности порождают наибольшее и наименьшее число классов эквивалентности?

0.1.11. Пусть $\bar{A} = \{0, 1, 2\}$ и $R = \{(0, 1), (1, 2)\}$. Найдите R^* и R^+ .

0.1.12. Докажите теорему 0.2 (б).

0.1.13. Пусть R — отношение на A . Покажите, что существует такое единственное отношение R_e , что

$$(1) R \subseteq R_e,$$

(2) R_e — отношение эквивалентности на A ,

(3) если R' — какое-то отношение эквивалентности на A и $R \subseteq R'$, то $R_e \subseteq R'$.

R_e называется *наименьшим отношением эквивалентности*, содержащим R .

Определение. Полным порядком на A называется такой рефлексивный частичный порядок на A , что для каждого непустого подмножества $B \subseteq A$ существует такое $b \in B$, что bRa для всех $a \in B$ (т. е. каждое непустое подмножество содержит наименьший элемент). Множество A называется тогда *вполне упорядоченным*.

0.1.14. Покажите, что отношение \leqslant (меньше или равно) является полным порядком на множестве положительных целых чисел.

Определение. Пусть A — множество. Обозначим

$$(1) A^1 = A,$$

$$(2) A^i = A^{i-1} \times A \text{ для } i > 1.$$

Пусть A^+ обозначает множество $\bigcup_{i \geq 1} A^i$.

0.1.15. Пусть R — полный порядок на A . Определим отношение \hat{R} на A^+ , положив $(a_1, \dots, a_m) \hat{R} (b_1, \dots, b_n)$ тогда и только тогда, когда выполняется одно из двух условий¹⁾:

¹⁾ Строго говоря, (a_1, \dots, a_m) означает $((\dots((a_1, a_2), a_3), \dots), a_m)$ в соответствии с определением A^n .

- (1) $a_i R b_i$ для некоторого $i \leq m$ и $a_j = b_j$ для всех $1 \leq j < i$,
(2) $m \leq n$ и $a_i = b_i$ для всех $1 \leq i \leq m$.

Покажите, что \hat{R} — полный порядок на A^+ . Он называется *лексикографическим порядком* на A^+ . (Примером лексикографического порядка может служить упорядочение слов в словаре.)

0.1.16. Для каждого из следующих отношений установить, является ли оно частичным порядком, рефлексивным частичным порядком, линейным порядком или рефлексивным линейным порядком:

$$(a) \subseteq \text{ на } \mathcal{P}(A),$$

$$(b) \subsetneq \text{ на } \mathcal{P}(A),$$

(в) отношение R_1 на множестве людей H , определенное так: aR_1b тогда и только тогда, когда a — отец b ,

(г) отношение R_2 на H , определенное так: aR_2b тогда и только тогда, когда a — предок b ,

(д) отношение R_3 на H , определенное так: aR_3b тогда и только тогда, когда a старше b .

0.1.17. Пусть R_1 и R_2 — отношения. Композицией отношений R_1 и R_2 , обозначаемой $R_1 \circ R_2$, называется отношение $\{(a, b) |$ существует такое c , что aR_1c и $cR_2b\}$. Покажите, что если R_1 и R_2 — отображения, то $R_1 \circ R_2$ — отображение. При каких условиях $R_1 \circ R_2$ будет всюду определенным отображением? Инъективным отображением? Биективным отображением?

0.1.18. Пусть A — конечное множество и $B \subseteq A$. Покажите, что если $M: A \rightarrow B$ — биективное отображение, то $A = B$.

0.1.19. Пусть A и B состоят соответственно из t и n элементов. Покажите, что существует n^t всюду определенных на A функций, отображающих A в B . Сколько существует всех не обязательно всюду определенных отображений из A в B ?

***0.1.20.** Пусть A — произвольное не обязательно конечное множество. Покажите, что множества $\mathcal{P}(A)$ и $\{M | M \text{ — всюду определенная функция, отображающая } A \text{ в } \{0, 1\}\}$ равномощны.

0.1.21. Покажите, что множество всех целых чисел равнощожно

(а) множеству простых чисел,

(б) множеству пар целых чисел.

Указание: Определите линейный порядок R на множестве пар целых чисел, положив $(i_1, j_1) R (i_2, j_2)$ тогда и только тогда, когда $i_1 + j_1 < i_2 + j_2$ или $i_1 + j_1 = i_2 + j_2$ и $i_1 < i_2$.

0.1.22. Множество A „больше“ множества B , если A и B имеют разную мощность и B равномощно подмножеству множества A . Покажите, что множество вещественных чисел, лежащих строго между

0 и 1, больше множества целых чисел. **Указание:** Возьмите однозначное десятичное представление вещественных чисел. Предположите от противного, что указанные множества равномощны. Тогда можно найти последовательность вещественных чисел r_1, r_2, \dots , которая содержит все вещественные числа $0 < r < 1$. Можете ли Вы найти вещественное число r между 0 и 1, которое отличается в i -м знаке от r_i , каково бы ни было i ?

***0.1.23.** Пусть R — линейный порядок на конечном множестве A . Покажите, что существует такой единственный элемент $a \in A$, что aRb для всех $b \in A - \{a\}$. Этот элемент a называется *наименьшим*. Если A бесконечно, всегда ли существует наименьший элемент?

***0.1.24.** Покажите, что $\{a, \{a, b\}\} = \{c, \{c, d\}\}$ тогда и только тогда, когда $a=c$ и $b=d$.

0.1.25. Пусть R — частичный порядок на множестве A . Покажите, что если aRb , то bRa ложно.

***0.1.26.** Используйте аксиомы об объединении множеств и о множестве всех подмножеств для того, чтобы показать, что если A и B — множества, то $A \times B$ — множество.

****0.1.27.** Покажите, что каждое множество либо конечно, либо бесконечно, но не то и другое вместе.

***0.1.28.** Покажите, что каждое счетное множество бесконечно.

***0.1.29.** Покажите, что следующие множества равномощны:

- (1) множество вещественных чисел между 0 и 1,
- (2) множество всех вещественных чисел,
- (3) множество всех отображений целых чисел в целые числа,
- (4) множество всех подмножеств положительных целых чисел.

****0.1.30.** Покажите, что $\mathcal{P}(A)$ всегда больше A для любого множества A .

0.1.31. Покажите, что если R — частичный порядок на множестве A , то отношение $R' = R \cup \{(a, a) | a \in A\}$ является рефлексивным частичным порядком на A .

0.1.32. Покажите, что если R — рефлексивный частичный порядок на множестве A , то отношение $R' = R - \{(a, a) | a \in A\}$ является частичным порядком на A .

0.2. МНОЖЕСТВА ЦЕПОЧЕК

В этой книге мы будем заниматься главным образом такими множествами, элементами которых служат цепочки символов. В настоящем разделе мы определим термины, связанные с цепочками.

0.2.1. Цепочки

Прежде всего нам необходимо понятие алфавита. Алфавитом мы будем называть любое множество символов. Предполагается, что термин „символ“ имеет достаточно ясный интуитивный смысл и не нуждается в дальнейшем пояснении.

Алфавит не обязан быть конечным и даже счетным, но во всех практических приложениях он будет конечным. Два примера алфавитов: множество, состоящее из 26 прописных и 26 строчных латинских букв (*латинский алфавит*), и множество $\{0, 1\}$, часто называемое *бинарным* или *двоичным алфавитом*.

Термины *буква* и *знак* будут использоваться как синонимы термина *символ* для обозначения элемента алфавита. Если написать последовательность символов, располагая их один за другим, то получится *цепочка символов*. Например, 01011 — цепочка в бинарном алфавите $\{0, 1\}$. Термины *слово* и *строка* (а иногда, особенно при лингвистических интерпретациях, *предложение*) часто используются как синонимы термина *цепочка*.

Существует одна цепочка, которая часто встречается и потому имеет специальное обозначение. Это *пустая цепочка*, обозначаемая символом e . Пустая цепочка не содержит ни одного символа.

Соглашение. Обычно мы будем прописными греческими буквами обозначать алфавиты. Буквы a, b, c и d будут обозначать отдельные символы, а буквы t, u, v, x, y и z , вообще говоря, будут обозначать цепочки символов. Цепочку, состоящую из i символов a , будем обозначать a^i . Например, $a^1 = a$, $a^2 = aa$, $a^3 = aaa$ и т. д. Тогда a^0 — пустая цепочка e .

Определение. Формально цепочки в алфавите Σ определяются следующим образом:

- (1) e — цепочка в Σ ,
- (2) если x — цепочка в Σ и $a \in \Sigma$, то xa — цепочка в Σ ,
- (3) y — цепочка в Σ тогда и только тогда, когда она является таковой в силу (1) или (2).

Определим операции над цепочками, которые понадобятся нам в дальнейшем. Если x и y — цепочки, то цепочка xy называется *конкатенацией* (или *сцеплением*) x и y . Например, если $x = ab$ и $y = cd$, то $xy = abcd$. Для любой цепочки x всегда $xe = ex = x$.

Обращением цепочки x (обозначается x^R) называется цепочка x , записанная в обратном порядке, т. е. если $x = a_1 \dots a_n$, где все a_i — символы, то $x^R = a_n \dots a_1$. Кроме того, $e^R = e$.

¹⁾ Мы отождествляем, таким образом, символ a с цепочкой, состоящей из одного символа a .

Пусть x, y и z —произвольные цепочки в некотором алфавите Σ . Назовем x префиксом цепочки xy , а y —суффиксом цепочки xy . Цепочку y назовем подцепочкой цепочки xz . Префикс и суффикс цепочки являются ее подцепочками. Например, ba —префикс и подцепочка цепочки bas . Заметим, что пустая цепочка является префиксом, суффиксом и подцепочкой любой цепочки.

Если $x \neq y$ и x —префикс (суффикс) цепочки y , то x называется *собственным* префиксом (суффиксом) цепочки y .

Длина цепочки—это число символов в ней, т. е. если $x = a_1 \dots a_n$, где все a_i —символы, то длина цепочки x равна n . Длину цепочки x будем обозначать $|x|$. Например, $|aab|=3$ и $|e|=0$. Все цепочки, с которыми нам придется встречаться, будут конечной длины.

0.2.2. Языки

Определение. Языком в алфавите Σ называется множество цепочек в Σ . Под это определение, конечно, подходит почти любое понятие языка. Фортран, Алгол, ПЛ/I и даже английский язык охватываются этим определением.

Пример 0.10. Рассмотрим простые примеры языков в алфавите Σ . Пустое множество \emptyset —это язык. Множество $\{e\}$, содержащее только пустую цепочку, тоже является языком. Заметим, что \emptyset и $\{e\}$ —два различных языка. \square

Определение. Обозначим через Σ^* множество, содержащее все цепочки в алфавите Σ , включая e . Например, если Σ —бинарный алфавит $\{0, 1\}$, то

$$\Sigma^* = \{e, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$$

Каждый язык в алфавите Σ является подмножеством множества Σ^* . Множество всех цепочек в Σ , за исключением e , обозначается Σ^+ .

Пример 0.11. Рассмотрим язык L_1 , содержащий все цепочки из нуля или более символов a . Его можно обозначить $\{a^i \mid i \geq 0\}$. Ясно, что $L_1 = \{a\}^*$. \square

Соглашение. В тех случаях, когда это не может привести к путанице, мы часто будем обозначать множество, состоящее из одного элемента, самим этим элементом.

В соответствии с этим соглашением $a^* = \{a\}^*$.

Определение. Если язык L таков, что никакая цепочка в L является собственным префиксом (суффиксом) никакой другой

цепочки в L , то говорят, что L обладает *префиксным* (суффиксным) свойством.

Например, a^* не обладает префиксным свойством, а $\{a^i b \mid i \geq 0\}$ обладает.

0.2.3. Операции над языками

Нам часто будут нужны различные операции над языками. Сейчас мы рассмотрим основные из них.

Так как язык—это множество, то операции объединения, пересечения, нахождения разности и дополнения применимы и к языкам. Операцию конкатенации можно применять к языкам так же, как к цепочкам.

Определение. Пусть L_1 —язык в алфавите Σ_1 , а L_2 —язык в алфавите Σ_2 . Тогда язык $L_1 L_2$, называемый *конкатенацией* (а также *сцеплением* или *произведением*) языков L_1 и L_2 ,—это язык $\{xy \mid x \in L_1 \text{ и } y \in L_2\}$.

В некоторых случаях нам нужно будет образовать сцепления произвольного числа цепочек из языка. Эта ситуация охватывается понятием итерации языка.

Определение. Итерация языка L , обозначаемая через L^* , определяется следующим образом:

- (1) $L^0 = \{e\}$,
- (2) $L^n = LL^{n-1}$ для $n \geq 1$,
- (3) $L^* = \bigcup_{n \geq 0} L^n$.

Позитивная итерация языка L , обозначаемая через L^+ ,—это язык $\bigcup_{n \geq 1} L^n$. Заметим, что $L^+ = LL^* = L^*L$ и $L^* = L^+ \cup \{e\}$.

Нас будут также интересовать отображения, определенные на языках. Примером отображения, которое часто встречается при работе с языками, служит гомоморфизм. Его можно определить так:

Определение. Пусть Σ_1 и Σ_2 —алфавиты. Гомоморфизмом называется любое отображение $h: \Sigma_1 \rightarrow \Sigma_2$. Область определения гомоморфизма h можно расширить до Σ_1^* , полагая $h(e) = e$ и $h(xa) = h(x)h(a)$ для всех $x \in \Sigma_1^*$ и $a \in \Sigma_1$.

Применяя гомоморфизм к языку L , мы получаем другой язык $h(L)$, который представляет собой множество цепочек $\{h(w) \mid w \in L\}$.

Пример 0.12. Допустим, что мы хотим заменить каждое вхождение в цепочку символа 0 на символ a , а каждое вхождение 1

на bb . Тогда можно определить гомоморфизм h так, что $h(0) = a$ и $h(1) = bb$. Если $L = \{0^n 1^n \mid n \geq 1\}$, то $h(L) = \{a^n b^{2n} \mid n \geq 1\}$. \square

Хотя гомоморфизмы не всегда взаимно однозначны, часто бывает полезно говорить об их обращениях (как отношениях).

Определение. Если $h: \Sigma_1 \rightarrow \Sigma_2^*$ — гомоморфизм, то отношение $h^{-1}: \Sigma_2^* \rightarrow \mathcal{P}(\Sigma_1)$, определенное ниже, называется *обращением гомоморфизма*. Если $y \in \Sigma_2^*$, то $h^{-1}(y)$ — это множество цепочек в алфавите Σ_1 , которые отображаются гомоморфизмом h в цепочку y , т. е. $h^{-1}(y) = \{x \mid h(x) = y\}$. Если L — язык в Σ_2 , то $h^{-1}(L)$ — язык в Σ_1 , состоящий из тех цепочек, которые h отображает в цепочки из L . Формально $h^{-1}(L) = \bigcup_{y \in L} h^{-1}(y) = \{x \mid h(x) \in L\}$.

Пример 0.13. Пусть h — гомоморфизм, для которого $h(0) = a$ и $h(1) = a$. Тогда $h^{-1}(a) = \{0, 1\}$ и $h^{-1}(a^*) = \{0, 1\}^*$.

В качестве второго примера возьмем такой гомоморфизм h , что $h(0) = a$ и $h(1) = e$. Тогда $h^{-1}(e) = 1^*$ и $h^{-1}(a) = 1^*01^*$. Здесь 1^*01^* обозначает язык $\{1^*01^* \mid i, j \geq 0\}$; это согласуется с нашими определениями и соглашением, отождествляющим a с $\{a\}$. \square

УПРАЖНЕНИЯ

0.2.1. Выпишите все (а) префиксы, (б) суффиксы и (в) подцепочки цепочки abc .

0.2.2. Докажите или опровергните, что $L^+ = L^* - \{e\}$.

0.2.3. Пусть h — гомоморфизм, определенный равенствами $h(0) = a$, $h(1) = bb$ и $h(2) = e$. Опишите язык $h(L)$, где $L = \{012\}^{*1}$.

0.2.4. Пусть гомоморфизм h тот же, что и в упр. 0.2.3. Опишите язык $h^{-1}(\{ab\}^*)$.

***0.2.5.** Докажите или опровергните следующие утверждения:

- (а) $h^{-1}(h(L)) = L$,
- (б) $h(h^{-1}(L)) = L$.

0.2.6. Может ли язык L^* или L^+ быть пустым? При каких условиях L^* и L^+ конечны?

***0.2.7.** Постройте полные порядки на следующих языках:

- (а) $\{a, b\}^*$,
- (б) $a^*b^*c^*$,
- (в) $\{\omega \mid \omega \in \{a, b\}^* \text{ и число символов } a \text{ в } \omega \text{ равно числу символов } b\}$.

0.2.8. Какие из следующих языков обладают префиксным (суффиксным) свойством:

1) Обратите внимание, что $\{012\}^*$ не то же самое, что $\{0, 1, 2\}^*$.

- (а) \emptyset ,
- (б) $\{e\}$,
- (в) $\{a^n b^n \mid n \geq 1\}$,
- (г) L^* , если L обладает префиксным свойством,
- (д) $\{\omega \mid \omega \in \{a, b\}^* \text{ и число символов } a \text{ в } \omega \text{ равно числу символов } b\}$.

0.3. НЕКОТОРЫЕ ПОНЯТИЯ МАТЕМАТИЧЕСКОЙ ЛОГИКИ

В этой книге будет приведен ряд алгоритмов, полезных при работе с языками. Для некоторых функций известно несколько реализующих их алгоритмов, и поэтому желательно описывать алгоритмы в общих терминах, чтобы их можно было оценивать и сравнивать.

Прежде всего, особенно желательно знать, что алгоритм делает именно то, что предполагалось. По этой причине мы будем давать доказательства того, что описываемые нами различные алгоритмы делают то, что было обещано. В этом разделе мы кратко поговорим о том, чтобы такое доказательство, и упомянем о некоторых полезных приемах доказательства.

0.3.1. Доказательства

Формальную математическую систему можно охарактеризовать с помощью следующих основных компонент:

- (1) основные символы,
- (2) правила образования,
- (3) аксиомы,
- (4) правила вывода.

Множество основных символов содержит символы для обозначения констант, операторов и т. д. В соответствии с *правилами образования* из этих основных символов строятся утверждения. Затем определяются примитивные утверждения, справедливость которых принимается без доказательства. Эти утверждения называются *аксиомами* системы.

Далее задаются правила, посредством которых из справедливых утверждений можно выводить новые справедливые утверждения. Такие правила называются *правилами вывода*.

Если требуется доказать, что некоторое утверждение истинно в некоторой формальной системе, то доказательством этого утверждения будет такая последовательность утверждений, что

- (1) каждое утверждение либо является аксиомой, либо его можно получить из одного или более предыдущих утверждений с помощью правила вывода;

(2) последнее утверждение последовательности является тем утверждением, которое надо доказать.

Утверждение, для которого можно найти доказательство, называется *теоремой* этой формальной системы. Очевидно, каждая аксиома формальной системы является теоремой.

Доказательство любой математической теоремы можно по крайней мере мысленно формулировать в этих терминах. Однако если детально проверять, является ли каждое утверждение аксиомой или получается из предыдущих утверждений с помощью исходных правил вывода, то доказательства всех теорем, кроме самых элементарных, станут слишком длинными. Задача нахождения доказательств такого вида сама по себе очень трудоемка даже для вычислительных машин.

Поэтому математики постоянно пользуются различными сокращениями, уменьшающими длину доказательства. Утверждения, ранее доказанные как теоремы, можно вставлять в доказательства. Можно опускать некоторые утверждения, когда (как мы надеемся) ясно, что происходит. Эти приемы практикуются в сущности везде, и данная книга не составляет исключения.

Известно, что невозможно дать универсальный метод доказательства теорем. Однако в последующих разделах мы упомянем о некоторых широко распространенных частных приемах.

0.3.2. Доказательство индукцией

Допустим, мы хотим доказать, что утверждение $S(n)$ о натуральном числе n истинно для всех чисел из множества N .

Если N конечно, то один из методов доказательства заключается в проверке того, что $S(n)$ истинно для каждого значения n из N . Этот метод иногда называют доказательством *исчерпыванием*.

Если N —бесконечное подмножество натуральных чисел, то можно применить *простую математическую индукцию*. Пусть n_0 —наименьшее число из N . Чтобы доказать, что $S(n)$ истинно для всех $n \in N$, надо

(1) показать, что $S(n_0)$ истинно (это называется *базисом индукции*),

(2) предполагая, что $S(m)$ истинно для всех $m < n$, принадлежащих N , показать, что $S(n)$ тоже истинно (это называется *шагом индукции*).

Пример 0.14. Пусть $S(n)$ —утверждение

$$1 + 3 + 5 + \dots + 2n - 1 = n^2$$

Мы хотим показать, что $S(n)$ истинно для всех положительных целых чисел. Таким образом, $N = \{1, 2, 3, \dots\}$.

Базис. Для $n = 1$ имеем $1 = 1^2$.

Шаг индукции. Предполагая, что $S(1), \dots, S(n)$ истинны (в частности, что $S(n)$ истинно), имеем

$$1 + 3 + 5 + \dots + (2n - 1) + (2(n+1) - 1) = n^2 + 2n + 1 = (n+1)^2$$

так что $S(n+1)$ тоже истинно.

Отсюда заключаем, что $S(n)$ истинно для всех положительных целых чисел. \square

Примеры доказательств индукцией, применяемых не к натуральным числам, читатель найдет в разд. 0.5.5.

0.3.3. Логические связки

Утверждения (теоремы) часто формулируются в виде „ P тогда и только тогда, когда Q “ или „ P —необходимое и достаточное условие для Q “, причем P и Q сами являются утверждениями. Термины *если*, *тогда*, *только тогда*, *необходимо*, *достаточно* и др. имеют точный смысл в логике.

Логическая связка—это символ, с помощью которого можно образовать утверждение из более простых утверждений. Примеры логических связок: *или*, *и*, *не*, *влечет*, причем *не*—унарная связка (т. е. присоединяемая к одному утверждению), а остальные—бинарные связки (т. е. связывающие два утверждения). Если P и Q —утверждения, то P и Q , P или Q , $не P$ и P влечет Q —тоже утверждения.

Символ \wedge обозначает связку *и*, \vee —связку *или*, \sim —*не* и \rightarrow —*влечет*.

Существуют точные правила, определяющие истинность или ложность утверждения, содержащего логические связки. Например, утверждение P и Q истинно только тогда, когда истинны оба утверждения P и Q . Свойства логических связок можно резюмировать в виде таблицы, называемой таблицей истинности, в которой даны значения составных утверждений в зависимости от значений их компонент. На рис. 0.5 приведена таблица истинности для логических связок *и*, *или*, *не* и *влечет*.

P	Q	$P \wedge Q$	$P \vee Q$	$\sim P$	$P \rightarrow Q$
Л	Л	Л	Л	И	И
Л	И	Л	И	И	И
И	Л	Л	И	Л	Л
И	И	И	И	Л	И

Рис. 0.5. Таблицы истинности для связок *и*, *или*, *не* и *влечет*.

Из этой таблицы видно, что $P \rightarrow Q$ ложно только тогда, когда P истинно, а Q ложно. Может показаться несколько странным, что если P ложно, то P влечет Q всегда истинно независимо от значения Q . Но в логике это обычное явление: если гипотеза ложна, из нее следует все, что угодно.

Рассмотрим теперь утверждение P если и только если Q . Это утверждение состоит из двух частей: P если Q и P только если Q . Вместо P если Q обычно говорят если Q то P — это лишь другой способ выражения утверждения Q влечет P . Вместо P только если Q часто говорят P только тогда, когда Q .

В действительности следующие шесть утверждений равносильны:

- (1) P влечет Q ,
- (2) если P то Q ,
- (3) P только если Q ,
- (4) P только тогда, когда Q ,
- (5) Q — необходимое условие для P ,
- (6) P — достаточное условие для Q .

Чтобы показать, что утверждение P тогда и только тогда, когда Q (или P если и только если Q) истинно, нужно показать, что одновременно Q влечет P и P влечет Q . Таким образом, утверждение P тогда и только тогда, когда Q истинно в точности тогда, когда P и Q либо оба истинны, либо оба ложны.

Существует несколько различных способов показать, что утверждение P влечет Q всегда истинно. Один из них заключается в том, чтобы показать, что утверждение не Q влечет не P ¹⁾ всегда истинно. Читателю следует проверить, что не Q влечет не P имеет точно ту же таблицу истинности, что и P влечет Q . Утверждение не Q влечет не P называется *контрапозицией* утверждения P влечет Q .

Один из важных методов доказательства — *доказательство от противного*, иногда называемое *косвенным доказательством* или *приведением к противоречию*. Чтобы этим методом показать, что P влечет Q истинно, нужно показать, что утверждение не Q и P влечет ложь истинно. Иначе говоря, мы предполагаем, что Q ложно, и если из предположения, что P истинно, мы сможем получить заведомо ложное утверждение, то P влечет Q должно быть истинно.

Утверждением, обратным к утверждению если P то Q (или его *обращением*), называется утверждение если Q то P . Утвер-

¹⁾ Предполагается, что связка *не* предшествует связке *влечет*. Таким образом, правильная фразировка этого утверждения такая: (*не* Q) влечет (*не* P). Вообще *не* предшествует *и*, и предшествует *или*, или предшествует *влечет*.

ждение P тогда и только тогда, когда Q часто пишут так: если P то Q и обратно. Заметим, что утверждение и его обращение имеют разные таблицы истинности.

УПРАЖНЕНИЯ

Определение. Хороший пример формальной математической системы — исчисление высказываний. Формально исчисление высказываний можно определить как систему \mathcal{S} , состоящую из

- (1) множества исходных символов,
- (2) правил образования правильно построенных утверждений (высказываний),
- (3) множества аксиом,
- (4) правил вывода.

(1) Исходными символами системы \mathcal{S} являются $(,)$, \rightarrow , \sim и бесконечное множество символов переменных a_1, a_2, a_3, \dots . Символ \rightarrow можно трактовать как *влечет*, а символ \sim — как *не*.

(2) Правильно построенное утверждение образуется в результате одного или более применений следующих правил:

- (а) Символ переменной является утверждением.
- (б) Если A и B — утверждения, то $(\sim A)$ и $(A \rightarrow B)$ — тоже утверждения.
- (3) Пусть A , B и C — утверждения. Тогда аксиомы системы \mathcal{S} — это утверждения

$$\begin{aligned} A1: & (A \rightarrow (B \rightarrow A)) \\ A2: & ((A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))) \\ A3: & ((\sim B \rightarrow \sim A) \rightarrow ((\sim B \rightarrow A) \rightarrow B)) \end{aligned}$$

(4) Правилом вывода является *схема заключения* (*modus ponens*), т. е. из утверждений $(A \rightarrow B)$ и A можно вывести утверждение B .

Мы будем опускать некоторые скобки, когда это не вызывает недоразумений. Утверждение $a \rightarrow a$ является теоремой системы \mathcal{S} ; в качестве ее доказательства можно взять последовательность утверждений:

- (i) $(a \rightarrow ((a \rightarrow a) \rightarrow a)) \rightarrow ((a \rightarrow (a \rightarrow a)) \rightarrow (a \rightarrow a))$ получается из A2 при $A = a$, $B = (a \rightarrow a)$ и $c = a$.
- (ii) $a \rightarrow ((a \rightarrow a) \rightarrow a)$ в силу A1.
- (iii) $(a \rightarrow (a \rightarrow a)) \rightarrow (a \rightarrow a)$ по схеме заключения из (i) и (ii).
- (iv) $a \rightarrow (a \rightarrow a)$ из A1.
- (v) $a \rightarrow a$ по схеме заключения из (iii) и (iv).

*0.3.1. Докажите, что $(\sim a \rightarrow a) \rightarrow a$ является теоремой системы \mathcal{S} .

0.3.2. *Тавтологией* называется утверждение, истинное для всевозможных значений входящих в него переменных. Покажите, что каждая теорема системы \mathcal{I} является тавтологией. *Указание:* Докажите это индукцией по числу шагов вывода, необходимых для получения теоремы.

****0.3.3.** Докажите обращение утверждения, сформулированного в упр. 0.3.2, а именно: каждая тавтология является теоремой. Таким образом, простой метод выяснения того, является ли утверждение исчисления высказываний теоремой, заключается в том, чтобы выяснить, является ли это утверждение тавтологией.

0.3.4. Постройте таблицу истинности утверждения *если P то если Q то R* .

Определение. Булеву алгебру можно трактовать как систему, в которой можно комбинировать переменные, принимающие значения истина и ложь, с помощью логических связок, неформально интерпретируемых как *и*, *или* и *нет*. Формально булева алгебра — это множество B вместе с операциями \cdot (*и*), $+$ (*или*) и $\bar{}$ (*нет*). Аксиомами булевой алгебры служат следующие утверждения: Для всех a, b и c , принадлежащих B ,

$$(1) \quad a + (b + c) = (a + b) + c \quad (\text{ассоциативность})$$

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c,$$

$$(2) \quad a + b = b + a \quad (\text{коммутативность})$$

$$a \cdot b = b \cdot a,$$

$$(3) \quad a \cdot (b + c) = (a \cdot b) + (a \cdot c) \quad (\text{дистрибутивность})$$

$$a + (b \cdot c) = (a + b) \cdot (a + c).$$

В множестве B имеются два выделенных элемента, 1 и 0 (в наиболее распространенной булевой алгебре они представляют соответственно истину и ложь), подчиняющиеся следующим законам:

$$(4) \quad a + 0 = a$$

$$a \cdot 1 = a,$$

$$(5) \quad a + \bar{a} = 1$$

$$a \cdot \bar{a} = 0.$$

Правилом вывода является замена равного равным.

***0.3.5.** Покажите, что следующие утверждения являются теоремами в любой булевой алгебре:

$$(a) \quad 0 = \bar{1},$$

$$(b) \quad a + (b \cdot \bar{a}) = a + b,$$

$$(v) \quad \bar{a} = a.$$

Какова неформальная интерпретация этих теорем?

0.3.6. Пусть A — множество. Покажите, что $\mathcal{P}(A)$ — булева алгебра, если операциями $+$, \cdot , и $\bar{}$ служат \cup , \cap и дополнение относительно универсального множества A .

****0.3.7.** Пусть B — булева алгебра и $\#B = n$. Покажите, что $n = 2^m$ для некоторого натурального числа m .

0.3.8. Докажите индукцией, что

$$1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

0.3.9. Докажите индукцией, что

$$(1 + 2 + \dots + n)^2 = 1^3 + 2^3 + \dots + n^3$$

***0.3.10.** Что неверно в следующем рассуждении?

Теорема. *Все кошки одного цвета.*

Доказательство. Пусть A — множество, состоящее из n кошек, $n \geq 1$. „Докажем“ индукцией по n , что все кошки в A одного цвета.

Базис. Если $n = 1$, то все кошки в A , очевидно, одного цвета.

Шаг индукции. Предположим, что если A — любое множество, состоящее из n кошек, то все они одного цвета. Пусть A' — множество, состоящее из $n+1$ кошек, $n \geq 1$. Выбросим из A' одну кошку. Тогда у нас останется множество A'' , состоящее из n кошек, которые по предположению индукции все одного цвета. Выбросим из A'' вторую кошку и вернем туда ранее выброшенную. Снова имеем множество, состоящее из n кошек, которые по предположению индукции все одного цвета. Так как две выброшенные кошки должны иметь один и тот же цвет, то множество A' должно состоять из кошек одного цвета. Таким образом, в любом множестве, состоящем из n кошек, все они одного цвета. \square

***0.3.11.** Пусть R — полный порядок на множестве A и $S(a)$ — утверждение об элементе $a \in A$. Предположим, что если $S(b)$ истинно для всех таких $b \neq a$, что bRa , то $S(a)$ тоже истинно. Покажите, что тогда $S(a)$ истинно для всех $a \in A$. Заметим, что это — обобщение принципа простой математической индукции.

0.3.12. Покажите, что существуют только четыре унарных логических связок. Постройте для них таблицы истинности.

0.3.13. Покажите, что существуют 16 бинарных логических связок.

0.3.14. Два логических утверждения *эквивалентны*, если они имеют одинаковые таблицы истинности. Покажите, что

- (a) $\sim(P \wedge Q)$ эквивалентно $\sim P \vee \sim Q$,
- (б) $\sim(P \vee Q)$ эквивалентно $\sim P \wedge \sim Q$.

0.3.15. Покажите, что $P \rightarrow Q$ эквивалентно $\sim Q \rightarrow \sim P$.

0.3.16. Покажите, что $P \rightarrow Q$ эквивалентно $P \wedge \sim Q \rightarrow \text{ложь}$.

***0.3.17.** Множество логических связок *полно*, если для любого логического утверждения можно найти эквивалентное утверждение, содержащее только эти логические связки. Покажите, что $\{\wedge, \sim\}$ и $\{\vee, \sim\}$ —полные множества логических связок.

Замечания по литературе

Чёрч [1956] и Мендельсон [1968] написали хорошие учебники по математической логике¹). Халмошу [1963] принадлежит удачное введение в булевы алгебры.

0.4. АЛГОРИТМЫ (ЧАСТИЧНЫЕ И ВСЮДУ ОПРЕДЕЛЕННЫЕ)

Понятие *алгоритма*—центральное в вопросах, связанных с вычислениями. К определению этого понятия можно подойти с разных сторон. В этом разделе мы обсудим неформально термин *алгоритм* и укажем, как получить более формальное определение.

0.4.1. Частичные алгоритмы

Начнем с несколько более общего понятия *частичного алгоритма*. Вообще говоря, частичный алгоритм состоит из конечного числа команд, каждая из которых может выполняться механически за фиксированное время и с фиксированными затратами. У частичного алгоритма может быть любое число входов и выходов.

Чтобы быть точными, надо определить термины *команда*, *вход* и *выход*. Однако мы не будем углубляться здесь в детали такого определения, так как для наших целей годится любое „разумное“ определение².

Хорошим примером частичного алгоритма служит программа в каком-нибудь машинном языке. Программа состоит из конечного числа машинных команд, причем для выполнения каждой команды требуется вычисление фиксированного объема. Однако часто бывает очень трудно понять алгоритмы, записанные на языке машинных команд. Поэтому в нашей книге принята более описательная запись алгоритмов. Приведем пример записи того типа, который мы будем использовать для описания алгоритмов.

¹⁾ Рекомендуются также книги Клини [1952, 1967].—Прим. перев.

²⁾ Термин *вход* здесь и далее понимается двояко: как набор входных переменных (аргументов) алгоритма (с указанием их типа) и как любое конкретное значение входной переменной (или набор значений всех переменных). Аналогично понимается термин *выход*. Из контекста обычно легко установить, что имеется в виду.—Прим. перев.

Пример 0.15. Рассмотрим алгоритм Евклида для определения наибольшего общего делителя двух положительных целых чисел p и q .

Алгоритм 1. Алгоритм Евклида.

Вход. p и q , положительные целые числа.

Выход. g , наибольший общий делитель чисел p и q .

Метод.

Шаг 1. Найти r , остаток от деления p на q .

Шаг 2. Если $r = 0$, положить $g = q$ и остановиться. В противном случае положить $p = q$, затем $q = r$ и перейти к шагу 1. □

Посмотрим, является ли алгоритм 1 частичным алгоритмом в смысле нашего определения. Алгоритм 1, несомненно, состоит из конечного множества команд (каждый шаг алгоритма можно считать одной командой) и имеет вход и выход. Но можно ли каждую команду выполнить механически с фиксированными затратами времени и памяти?

Строго говоря, ответ на этот вопрос должен быть отрицательным, потому что если p и q достаточно велики, то затраты, которые могут потребоваться для вычисления остатка от деления p на q , будут в каком-то смысле пропорциональны величинам p и q .

Однако мы могли бы заменить шаг 1 последовательностью шагов, которые в совокупности вычисляют остаток от деления p на q , причем количество ресурсов, необходимых для выполнения одного такого шага, фиксировано и не зависит от p и q . (При этом число повторений каждого шага возрастает вместе с p и q .) Эти шаги могли бы, например, быть реализацией обычного метода деления с помощью карандаша и бумаги.

Таким образом, мы допускаем, чтобы шаг частичного алгоритма сам был частичным алгоритмом. При таком свободном понимании рассмотренный выше алгоритм 1 можно считать частичным алгоритмом.

Вообще удобно предполагать, что целые числа—это элементарные объекты, и впредь мы так и будем поступать. Любое целое число можно хранить в одной ячейке памяти, любую арифметическую операцию над целыми числами можно выполнить за один шаг. Это предположение оправдано только в том случае, когда целые числа меньше 2^k , где k —число битов машинного слова; такая ситуация часто встречается на практике. Однако читатель должен помнить, что если элементарные шаги связаны только с числами ограниченной величины, то для работы с произвольными числами могут потребоваться дополнительные ресурсы.

Теперь перед нами встает, по-видимому, самая важная проблема: доказательство того, что частичный алгоритм делает именно то, что он должен делать. Действительно ли то число g , которое алгоритм 1 вычисляет по каждой паре целых чисел p и q , является их наибольшим общим делителем? Ответ на этот вопрос утверждательный; доказательство этого утверждения мы оставляем в качестве упражнения.

Можно, однако, мимоходом отметить один полезный способ доказательства того, что частичный алгоритм работает как надо,— индукцию по числу выполненных шагов.

0.4.2. Алгоритмы

Теперь мы наложим на частичные алгоритмы очень важное ограничение, чтобы получить то, что можно назвать *всюду определенным алгоритмом* (или просто *алгоритмом*).

Определение. Частичный алгоритм *останавливается* на данном входе, если существует такое натуральное число t , что после выполнения t (не обязательно различных) элементарных команд этого алгоритма либо не окажется ни одной команды, которую теперь можно выполнить, либо последней выполненной командой была команда „остановиться“. Частичный алгоритм, который останавливается на всех входах, т. е. на всех значениях входных данных, называется *всюду определенным алгоритмом* или просто *алгоритмом*.

Пример 0.16. Рассмотрим частичный алгоритм из примера 0.15. Заметим, что шаги 1 и 2 должны выполняться поочередно. После шага 1 должен выполняться шаг 2. После шага 2 либо выполняется шаг 1, либо следующий шаг невозможен, т. е. алгоритм останавливается. Можно доказать, что для каждого входа p и q алгоритм останавливается не более чем через $2q$ шагов¹⁾ и, значит, этот частичный алгоритм является просто алгоритмом. Доказательство основано на том обстоятельстве, что величина r , вычисляемая на шаге 1, меньше q , откуда вытекает, что последовательные значения переменной q , получающиеся после выполнения шага 1, образуют монотонно убывающую последовательность. Таким образом, к тому моменту, когда шаг 2 выполнится в q -й раз, величина r , которая всегда меньше текущего значения q и не может быть отрицательной, должна стать равной нулю. Когда $r=0$, алгоритм останавливается. \square

¹⁾ На самом деле верхняя граница для числа шагов при $q > 1$ равна $4 \log_2 q$. Доказательство этого утверждения оставляем читателю в качестве упражнения.

0.4. АЛГОРИТМЫ (ЧАСТИЧНЫЕ И ВСЮДУ ОПРЕДЕЛЕННЫЕ)

Частичный алгоритм может не остановиться на некоторых входах по некоторым причинам. Может случиться, что процесс владет в бесконечный цикл. Например; если частичный алгоритм содержит команду:

Шаг 1. Если $x=0$, то перейти к шагу 1; в противном случае остановиться,

то для $x=0$ этот частичный алгоритм никогда не остановится. Эту ситуацию можно варьировать бесконечно.

Мы будем заниматься главным образом алгоритмами, т. е. всюду определенными алгоритмами. Нас интересует не только доказательство корректности алгоритмов, но и оценка их сложности. Два главных критерия при оценке того, насколько хорошо работает алгоритм, следующие:

(1) число выполненных элементарных механических операций как функция от величины входа (*временная сложность*);

(2) объем вспомогательной памяти, требуемый для хранения промежуточных результатов, возникающих в ходе вычисления,— тоже как функция от величины входа (*емкостная сложность*).

Пример 0.17. В примере 0.16 мы видели, что число шагов алгоритма 1 (пример 0.15), требуемое для входа (p, q) , ограничено сверху величиной $2q$. Объем используемой памяти равен трем ячейкам, по одной для p , q и r , если считать, что любое целое число может храниться в одной ячейке. Если предположить, что объем памяти, необходимый для хранения целого числа, зависит от длины бинарного представления этого числа, то объем используемой памяти пропорционален $\log_2 n$, где n —наибольшее из чисел p и q . \square

0.4.3. Рекурсивные функции

Частичный алгоритм определяет некоторое отображение множества всех подходящих входов во множество выходов. Отображение, определяемое частичным алгоритмом, называется *частично-рекурсивной функцией* или просто *рекурсивной функцией*. Если алгоритм всюду определен, то отображение называется *общерекурсивной функцией*.

С помощью частичного алгоритма можно определить также и язык. Возьмем алгоритм, которому можно предъявить произвольную цепочку x . После некоторого вычисления алгоритм выдаст выход „да“, если цепочка x принадлежит языку. Если x не принадлежит языку, то алгоритм может остановиться и сказать „нет“, а может никогда не остановиться. Такой алгоритм определяет язык L как множество входных цепочек, для кото-

ГЛ. 0. ПРЕДВАРИТЕЛЬНЫЕ МАТЕМАТИЧЕСКИЕ СВЕДЕНИЯ

рых он выдает выход „да“. Поведение частичного алгоритма на цепочке, не принадлежащей языку, нельзя считать допустимым с практической точки зрения. Если по прошествии некоторого времени частичный алгоритм все еще не остановился на входе x , мы не знаем, то ли x принадлежит языку, но алгоритм еще не закончил вычисление, то ли x не принадлежит языку и алгоритм никогда не остановится.

Если бы мы определили язык с помощью всюду определенного алгоритма, то последний останавливался бы на всех входах. Следовательно, по отношению к такому алгоритму наше терпение оправдано, так как нам известно, что, если ждать достаточно долго, алгоритм в конце концов остановится и скажет либо „да“, либо „нет“.

Множество, определяемое частичным алгоритмом, называется *рекурсивно перечислимым*. Множество, определяемое всюду определенным алгоритмом, называется *рекурсивным*.

Если использовать более точные определения, можно строго доказать, что существуют множества, которые не являются рекурсивно перечислимыми. Можно также показать, что существуют рекурсивно перечислимые множества, которые не являются рекурсивными.

То же самое можно сформулировать по-другому. Существуют отображения, которые нельзя задать никакими частичными алгоритмами, а также такие отображения, которые можно задать частичными алгоритмами, но нельзя всюду определенными.

Мы увидим далее, что эти понятия имеют фундаментальное значение для теории программирования. В разд. 0.4.5 будет приведен пример частичного алгоритма, для которого, как можно показать, не существует эквивалентного (всюду определенного) алгоритма.

0.4.4. Задание алгоритмов

В предыдущем разделе было неформально определено, что мы понимаем под алгоритмом, частичным или всюду определенным. Можно дать строгие определения этих терминов, используя разные формализмы. Существует много формальных способов описания алгоритмов. Отметим следующие:

- (1) Машины Тьюринга [Тьюринг, 1936—1937].
- (2) Грамматики Хомского типа 0 [Хомский, 1959 а, 1963].
- (3) Алгоритмы Маркова [Марков, 1951].
- (4) Лямбда-исчисление [Чёрч, 1941].
- (5) Системы Поста [Пост, 1943].
- (6) ТАГ-системы [Пост, 1965].
- (7) Большинство языков программирования [Саммет, 1969].

Этот список можно продолжить. Здесь важно отметить, что алгоритм, записанный в одном из этих формализмов, можно промоделировать алгоритмом, записанным в любом другом из этих же формализмов. В этом смысле все перечисленные формализмы эквивалентны.

Много лет назад Чёрч и Тьюринг выдвинули гипотезу, что любой вычислительный процесс, который можно разумно назвать алгоритмом, можно промоделировать на машине Тьюринга. Эта гипотеза известна как *тезис Чёрча—Тьюринга* и является общеизвестной. Таким образом, наиболее общий класс множеств, с которым нам приходится встречаться на практике, должен содержаться в классе рекурсивно перечислимых множеств.

Большинство языков программирования, по крайней мере в принципе, пригодны для задания любого частичного алгоритма. В гл. 11 (том 2) мы увидим, к чему приводит эта их способность, когда мы пытаемся оптимизировать программы.

Мы не будем здесь обсуждать детали формализмов, предназначенных для описания алгоритмов, хотя некоторые из них появятся в упражнениях. Очень доступное введение в эту область содержится в книге Минского [1967]¹⁾.

В нашей книге, как мы уже видели, для описания алгоритмов используется неформальная запись.

0.4.5. Проблемы

Слово *проблема* мы будем употреблять довольно специфическим образом.

Определение. *Проблема* (или *вопрос*)—это утверждение (предикат), истинное или ложное в зависимости от значений входящих в него неизвестных (переменных) определенного типа. Проблема обычно формулируется как вопрос, и мы говорим, что ответ на вопрос—„да“, если это утверждение истинно, и „нет“, если утверждение ложно.

Пример 0.18. Примером проблемы служит следующее утверждение: „Целое число x меньше целого числа y “. Это утверждение можно выразить в более разговорном виде, устранив упоминание о типе переменных x и y и придав ему форму вопроса: „ x меньше y ?“ □

Определение. *Частный случай* проблемы—это набор допустимых значений ее неизвестных.

Например, частными случаями примера 0.18 являются упорядоченные пары целых чисел.

¹⁾ Можно порекомендовать также книгу Мальцева [1965].—Прим. перев.

Отображение множества частных случаев проблемы во множество {да, нет} называется *решением* проблемы. Если это отображение можно задать алгоритмом, то проблема называется (алгоритмически) *разрешимой*. Если алгоритма, определяющего это отображение, нет, то проблема называется (алгоритмически) *неразрешимой*.

Одним из выдающихся достижений математики XX века было открытие алгоритмически неразрешимых проблем. Позднее мы увидим, что неразрешимые проблемы серьезно затрудняют развитие широко применимой теории вычислений.

Пример 0.19. Исследуем проблему: „Является ли частичный алгоритм P (всюду определенным) алгоритмом?“ Анализ этой проблемы покажет, почему некоторые проблемы оказываются неразрешимыми. Сначала предположим, что все частичные алгоритмы описываются в некоторой формальной системе вроде тех, что были упомянуты ранее в этом разделе.

В любом формальном языке, предназначенном для задания алгоритмов, можно, оказывается, описать только счетное число алгоритмов. Хотя мы не можем доказать это здесь в общем виде, рассмотрим один пример формализма — язык „абсолютных“ машинных программ, а другие упомянутые формализмы отложим до упражнений. Любая программа в абсолютном машинном языке представляет собой конечную последовательность нулей и единиц (которые можно представлять себе сгруппированными в машинные слова по 32, 36, 48 или другому числу знаков).

Допустим, у нас есть цепочка из нулей и единиц, представляющая программу в машинном языке. Этой программе можно присвоить натуральное число (номер), соответствующее ее положению в некотором полном упорядочении всех цепочек из нулей и единиц. Одно такое упорядочение можно получить, расположив цепочки по возрастанию их длин, а цепочки равной длины упорядочив лексикографически (при этом каждая цепочка рассматривается как двоичное число). Так как число цепочек любой заданной длины конечно, то любой цепочке из множества $\{0, 1\}^*$ будет поставлено в соответствие некоторое целое положительное число. Первые несколько пар этого биективного соответствия приведены в табл. 0.1. Таким образом, мы видим, что для каждой программы в машинном языке можно однозначно найти соответствующее целое положительное число и для каждого такого числа можно найти некоторую программу.

Какой именно формализм выбирается для описания частичных алгоритмов, не имеет значения: всегда можно установить взаимно однозначное соответствие между частичными алгоритмами и целыми положительными числами. Таким образом, имеет смысл говорить об i -м частичном алгоритме в любом данном

алгоритмическом формализме. Более того, это соответствие между алгоритмами и числами достаточно просто, так что по данному числу i можно записать соответствующий алгоритм, а по данному алгоритму найти соответствующее число.

Предположим, что существует частичный алгоритм P_j , который является (всюду определенным) алгоритмом, на его вход

Таблица 0.1

Число	Цепочка
1	e
2	0
3	1
4	00
5	01
6	10
7	11
8	000
9	001

подаются описания частичных алгоритмов в нашем формализме и он выдает ответ „да“ для данного входа тогда и только тогда, когда этот вход описывает (всюду определенный) алгоритм. Все известные формализмы, используемые для описания частичных алгоритмов, обладают тем свойством, что можно простыми способами комбинировать алгоритмы, получая таким образом новые алгоритмы. В частности, если дан гипотетический алгоритм P_j , то можно построить алгоритм P_k , который работает следующим образом:

Алгоритм P_k .

Вход. Любой частичный алгоритм P , имеющий одну входную переменную.

Выход. (1) „Нет“, если (а) P не является алгоритмом или (б) P является алгоритмом и $P(P) =$ „да“.

(2) „Да“ в остальных случаях.

Запись $P(P)$ означает, что мы применяем частичный алгоритм P к описанию его самого.

Метод. (1) Если $P_j(P) =$ „да“, то перейти к шагу (2); в противном случае выдать на выходе „нет“ и остановиться.

(2) Если P — алгоритм и его входы — описания частичных алгоритмов, а выходы — ответы „да“ и „нет“, то алгоритм P_k применяет алгоритм P к самому себе, т. е. к своему описанию.

(При этом предполагается, что по описанию частичного алгоритма можно установить, имеют ли его входы и выходы указанный вид. Для известных формализмов это предположение верно.)

(3) P_k выдает „нет“ или „да“, если P выдает „да“ или „нет“ соответственно. \square

Мы видим, что P_k —(всюду определенный) алгоритм, если P_j —алгоритм. При этом P_k имеет одну входную переменную. Но что же делает P_k , когда он сам подается на свой вход? По предположению алгоритм P_j определяет, что P_k —алгоритм (т. е. $P_j(P_k) = \text{да}$), затем P_k моделирует себя на себе, т. е. свое поведение на своем описании, взятом в качестве входа. Но тогда P_k не может дать непротиворечивый результат. Если P_k устанавливает, что моделирование дает результат „да“, то P_k дает на выходе результат „нет“. Но алгоритм P_k определен так, что в применении к самому себе он должен давать ответ „да“. Аналогичный парадокс возникает, если P_k обнаруживает, что моделирование дает результат „нет“. Отсюда мы должны заключить, что предположение о существовании алгоритма P_j ошибочно, и, следовательно, проблема „является ли P алгоритмом?“ неразрешима для любого известного алгоритмического формализма. \square

Следует подчеркнуть, что проблема разрешима тогда и только тогда, когда существует алгоритм, который для любого частного случая этой проблемы дает ответ „да“ или „нет“. Для конкретных частных случаев проблемы иногда можно получить ответ „да“ или „нет“. Но это еще не делает проблему разрешимой. Для того чтобы можно было говорить о разрешимости проблемы, нужно дать единый алгоритм, работающий для всех частных случаев проблемы.

Следует особо отметить, что очень важную роль играет кодирование частных случаев проблемы. Обычно подразумевается некоторое „стандартное“ кодирование (кодирование, для которого существует алгоритм, отображающий коды описаний алгоритмов в эквивалентные программы машин Тьюринга). Если используются нестандартные кодирования, то неразрешимые проблемы могут стать разрешимыми. Но в таких случаях не существует алгоритма, с помощью которого можно перейти от стандартного кодирования к нестандартному (см. упр. 0.4.21)¹⁾.

¹⁾ Кодирование алгоритмов натуральными числами часто называют *нумерацией* алгоритмов. Одна такая нумерация упоминается в упр. 0.4.10. „Стандартным“ кодированием соответствуют так называемые главные или допустимые нумерации частично рекурсивных функций (см. [Мальцев, 1965], [Роджерс, 1967]).—Прим. перев.

0.4.6. Проблема соответствий Поста

В этом разделе мы приведем один пример неразрешимой проблемы, а именно проблему соответствий Поста. Далее эта проблема будет применяться для доказательства неразрешимости других проблем.

Определение. Частный случай проблемы соответствий Поста над алфавитом Σ —это конечное множество пар из $\Sigma^+ \times \Sigma^+$ (т. е. множество пар непустых цепочек в алфавите Σ). Проблема заключается в выяснении того, существует ли конечная последовательность (принадлежащих этому множеству и не обязательно различных) пар $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$, удовлетворяющих условию $x_1 x_2 \dots x_m = y_1 y_2 \dots y_m$. Такую последовательность мы будем называть *решающей последовательностью* для этого частного случая проблемы соответствий (или просто *решением*).

Пример 0.20. Рассмотрим следующий частный случай проблемы соответствий над алфавитом $\{a, b\}$:

$$\{(abb, b), (a, aab), (ba, b)\}$$

Последовательность $(a, aab), (a, aab), (ba, b), (abb, b)$ —решающая, так как

$$(a)(a)(ba)(abb) = (aab)(aab)(b)(b)$$

Частный случай $\{(ab, aba), (aba, baa), (baa, aa)\}$ проблемы соответствий не имеет решающих последовательностей, так как любая такая последовательность должна начинаться парой (ab, aba) , а тогда общее число букв a в первых компонентах пар, входящих в последовательность, будет меньше числа букв a во вторых компонентах. \square

Существует частичный алгоритм, который в некотором смысле „решает“ проблему соответствий. А именно, можно линейно упорядочить всевозможные последовательности пар цепочек, которые можно построить по данному частному случаю проблемы. Затем можно начать проверять для каждой последовательности, является ли она решающей. Обнаружив первую решающую последовательность, алгоритм остановится и ответит „да“. Если решающей последовательности не окажется, то этот частичный алгоритм будет работать бесконечно.

Однако всюду определенного алгоритма, решающего проблему соответствий¹⁾ (т. е. выдающего правильный ответ „да“ или „нет“ для любого частного случая проблемы), не существует—можно

¹⁾ Над алфавитом Σ , содержащим не менее двух символов.—Прим. перев.

показать, что если бы такой алгоритм был, то с его помощью можно было бы разрешить и проблему остановки для машин Тьюринга (упр. 0.4.22), а эта проблема неразрешима (упр. 0.4.14).

УПРАЖНЕНИЯ

0.4.1. Совершенное число — это натуральное число, равное сумме всех своих делителей (включая 1, но исключая само число). Например, $6 = 1 + 2 + 3$ и $28 = 1 + 2 + 4 + 7 + 14$ — первые два совершенных числа (следующие три: 496, 8 128 и 33 550 336). Постройте частичный алгоритм, который для входа i выдает на выходе i -е совершенное число. (До сих пор неизвестно, конечно или бесконечно множество совершенных чисел.)

0.4.2. Докажите корректность алгоритма Евклида (пример 0.15).

0.4.3. Опишите алгоритм сложения двух n -значных десятичных чисел. Сколько времени и места требует этот алгоритм (т. е. каковы соответствующие функции от n)? (См. [Виноград, 1965], где обсуждается временная сложность сложения.)

0.4.4. Опишите алгоритм умножения двух n -значных чисел. Сколько времени и места требует этот алгоритм? (См. [Виноград, 1967] и [Кук и Аандераа, 1969], где обсуждается временная сложность умножения.)

0.4.5. Дайте алгоритм умножения двух целочисленных ($n \times n$)-матриц. Предположим, что каждую целочисленную арифметическую операцию можно выполнить за один шаг. Какова тогда скорость (число шагов) Вашего алгоритма? Если она пропорциональна n^3 , посмотрите работу Штрассена [1969], где дается асимптотически более быстрый алгоритм.

0.4.6. Пусть $L \subseteq \{a, b\}^*$. Характеристической функцией множества L называется такое отображение $f_L: \{a, b\}^* \rightarrow \{0, 1\}$, что $f_L(w) = 1$, если $w \in L$, и $f_L(w) = 0$ в противном случае. Покажите, что L рекурсивно тогда и только тогда, когда f_L — общерекурсивная функция.

0.4.7. Покажите, что L — рекурсивное множество тогда и только тогда, когда оба множества L и \bar{L} рекурсивно перечислимы.

0.4.8. Пусть P — частичный алгоритм, определяющий рекурсивно перечислимое множество $L \subseteq \{a, b\}^*$. С помощью P постройте алгоритм P' , который порождает все элементы множества L и только их, т. е. его выходом должна быть бесконечная цепочка вида $x_1 \# x_2 \# \dots$, где $L = \{x_1, x_2, \dots\}$. Указание: Постройте алгоритм P' так, чтобы для каждой пары (i, j) из

некоторого разумного упорядочения всех пар натуральных чисел он применял частичный алгоритм P к j -й цепочке из множества $\{a, b\}^*$ в течение i шагов.

Определение. Машина Тьюринга состоит из конечного множества состояний (Q), конечного множества символов ленты (Γ) и функции δ (функции переходов, или программы), которая отображает некоторое подмножество произведения $Q \times \Gamma$ в множество $\Sigma Q \times \Gamma \times \{L, R\}$. В множестве Γ выделяется подмножество Σ входных символов и один символ из $\Gamma - \Sigma$ называется пустым. Одно состояние q_0 называется начальным. Машина Тьюринга работает на ленте, разделенной на ячейки, одну из которых обозревает головка. В любой момент времени все ячейки, кроме конечного числа, заняты пустыми символами. Конфигурацией машины Тьюринга называется пара $(q, \alpha \uparrow \beta)$, где q — состояние, $\alpha \beta$ — непустая часть ленты, \uparrow — специальный символ, показывающий, что головка обозревает ячейку, расположенную непосредственно справа от него (символ \uparrow не является символом ленты и не занимает ячейки).

Следующая конфигурация (после $(q, \alpha \uparrow \beta)$) определяется с помощью символа A , обозреваемого головкой (это самый левый символ цепочки β или пустой символ, если $\beta = e$), и значения функции $\delta(q, A)$. Допустим, что $\delta(q, A) = (p, A', D)$, где p — состояние, A' — символ ленты и $D \in \{L, R\}$. Тогда следующей конфигурацией будет $(p, \alpha' \uparrow \beta')$, где $\alpha' \beta'$ образуется из $\alpha \beta$ заменой символа A , стоящего справа от \uparrow , на A' и сдвигом символа \uparrow на одну позицию в направлении D (влево, если $D = L$, и вправо, если $D = R$). Для того чтобы передвинуть символ \uparrow , стоящий на конце, может оказаться необходимым добавить на этом конце пустой символ.

Машину Тьюринга можно представлять себе как формальную систему, определяющую частичный алгоритм. Входом этого алгоритма может быть любая конечная цепочка $w \in \Sigma^*$. Алгоритм работает, начиная с конфигурации $(q_0, \uparrow w)$ и последовательно вычисляя следующие конфигурации. Если машина Тьюринга останавливается, т. е. достигает конфигурации, для которой следующий шаг невозможен (напомним, что функция δ определена не обязательно для всех пар из $Q \times \Gamma$), то выходом является непустая часть ее ленты.

***0.4.9.** Постройте машину Тьюринга, которая для данного входа $w \in \{0, 1\}^*$ напишет на ленте слово ДА, если w — палиндром (т. е. $w = w^R$), и НЕТ в противном случае, причем в любом случае останавливается.

***0.4.10.** Предположим, что каждая машина Тьюринга использует в качестве состояний и символов ленты конечное подмно-

жество некоторого счетного множества символов $\{a_1, a_2, a_3, \dots\}$. Покажите, что существует взаимно однозначное соответствие между натуральными числами и машинами Тьюринга.

****0.4.11.** Покажите, что не существует машины Тьюринга, которая останавливается на всех входах (т. е. задает алгоритм) и определяет по данному числу i , записанному на ее ленте в двоичной форме, останавливается ли когда-нибудь i -я машина Тьюринга (см. упр. 0.4.10).

***0.4.12.** Пусть $\{a_1, a_2, \dots\}$ — счетное множество символов. Покажите, что множество всех цепочек конечной длины, составленных из символов этого множества, счетно.

***0.4.13.** Опишите неформально машину Тьюринга, входами которой являются пары натуральных чисел и которая для данного входа (i, j) останавливается тогда и только тогда, когда i -я машина Тьюринга останавливается на j -й входной цепочке (см. упр. 0.4.12). Если при этом она дает тот же выход, что и i -я машина на j -й цепочке, будем называть ее *универсальной*. Универсальные машины Тьюринга используются во многих доказательствах.

****0.4.14.** Покажите, что не существует машины Тьюринга, останавливающейся на любом входе, который является парой натуральных чисел, и для данного входа (i, j) печатающей на ленте ДА, если i -я машина Тьюринга останавливается на входе j , и НЕТ в противном случае. *Указание:* Предположите, что такая машина существует, и получите противоречие, как в примере 0.19.

****0.4.15.** Покажите, что не существует машины Тьюринга, входами которой являются машины Тьюринга (т. е. их описания) и которая для машин, задающих алгоритмы, выдает ответ ДА и останавливается, а для остальных машин не останавливается.

***0.4.16.** Покажите, что проблема „остановится ли машина Тьюринга, начав работу на пустой ленте?“ неразрешима.

0.4.17. Покажите, что проблема определения того, является ли высказывание теоремой в исчислении высказываний, разрешима. *Указание:* См. упр. 0.3.2 и 0.3.3.

0.4.18. Покажите, что проблема выяснения, входит ли цепочка в данное рекурсивное множество, разрешима.

0.4.19. Существуют ли решающие последовательности для следующих частных случаев проблемы соответствий Поста:

- (01, 011), (10,000), (00,0);
- (1,11), (11,101), (101,011), (011, 1011)?

Как согласовать возможность ответа в этом упражнении с фактом неразрешимости проблемы соответствий Поста?

0.4.20. Покажите, что проблема соответствий Поста над алфавитом $\{a\}$ разрешима. Как согласовать этот результат с неразрешимостью проблемы соответствий Поста над алфавитом, содержащим больше символов?

***0.4.21.** Пусть P_1, P_2, \dots — перечень (нумерация) всех частичных алгоритмов в некотором формализме. Определим новый перечень P'_1, P'_2, \dots следующим образом:

- (1) Пусть P'_{2i-1} — i -й (всюду определенный) алгоритм из списка P_1, P_2, \dots
- (2) Пусть P'_{2i} — i -й (всюду определенный) алгоритм из списка P_1, P_2, \dots

Тогда существует простой алгоритм определения по данному j , является ли P'_j алгоритмом: достаточно посмотреть, четно или нечетно число j . Более того, каждый P'_i совпадает с некоторым P_j . Как согласовать существование такого взаимно однозначного соответствия между натуральными числами и частичными алгоритмами с результатом примера 0.19?

****0.4.22.** Докажите неразрешимость проблемы соответствий Поста. *Указание:* Для данной машины Тьюринга постройте такой частный случай проблемы Поста, что решающая последовательность для него существует тогда и только тогда, когда эта машина Тьюринга останавливается, начав работу на пустой ленте.

***0.4.23.** Покажите, что алгоритм Евклида из примера 0.15 останавливается не более, чем через $4 \log_2 q$ шагов, если начинает работу с такими входами p и q , что $q > 1$.

Определение. Вариантом проблемы соответствий Поста является *проблема частичного соответствия* над алфавитом Σ . Эта проблема состоит в том, чтобы для любого данного конечного множества пар из $\Sigma^+ \times \Sigma^+$ определить, существует ли для каждого $m > 0$ такая последовательность не обязательно различных пар $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$, что первые m символов цепочки $x_1 x_2 \dots x_m$ совпадают с первыми m символами цепочки $y_1 y_2 \dots y_m$.

****0.4.24.** Докажите, что проблема частичного соответствия неразрешима.

Замечания по литературе

Дэвис [1965] собрал хорошую антологию многих ранних работ по теории алгоритмов. Работа Тьюринга [1936], в которой впервые появились машины Тьюринга, читается с особым интересом, если иметь в виду, что она написана до того, как были придуманы современные электронные вычислительные машины.

Исследование рекурсивных и частично-рекурсивных функций составляет часть теперь уже очень развитой области математики, называемой теорией

рекурсивных функций. Хорошие источники в этой области — книги Роджерса [1957], Клини [1952] и Дэвиса [1958]¹⁾.

Проблема соответствий Поста впервые появилась в [Пост, 1947]. Проблема частичного соответствия, сформулированная перед упр. 0.4.24, взята из [Кнут, 1965].

Исследование сложности вычислений — это изучение алгоритмов с точки зрения измерения числа элементарных операций (временная сложность) или объема вспомогательной памяти (емкостная сложность), требуемых для вычисления данной функции. Бородин [1970] и Хартманис и Хопкрофт [1970] написали хорошие обзоры по этой теме, а Эрланд и Фишер [1970] составили аннотированный библиографию.

Решения многих упражнений данного раздела, отмеченные звездочками, можно найти в книгах Минского [1967] и Хопкрофта и Ульмана [1969].

0.5. НЕКОТОРЫЕ ПОНЯТИЯ ТЕОРИИ ГРАФОВ

Многие структуры, полезные при выполнении вычислений, удобно описывать с помощью графов и деревьев. В этом разделе мы рассмотрим ряд понятий теории графов, которые нам понадобятся в дальнейшем.

0.5.1. Ориентированные графы

Графы могут быть ориентированные и неориентированные, упорядоченные и неупорядоченные. Нас будут интересовать главным образом упорядоченные и неупорядоченные ориентированные графы.

Определение. Неупорядоченный ориентированный граф G — это пара (A, R) , где A — множество элементов, называемых *вершинами* (или *узлами*), а R — отношение на множестве A . Если не оговорено противное, термин *граф* будет обозначать *ориентированный граф*.

Пример 0.21. Пусть $G = (A, R)$, где $A = \{1, 2, 3, 4\}$ и $R = \{(1, 1), (1, 2), (2, 3), (2, 4), (3, 4), (4, 1), (4, 3)\}$. Можно изобразить этот граф, занумеровав четыре точки числами 1, 2, 3, 4 и проведя стрелку из точки a в точку b , если $(a, b) \in R$ (рис. 0.6). \square

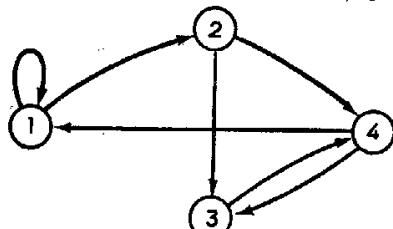


Рис. 0.6. Пример ориентированного графа.

¹⁾ См. также книгу Мальцева [1965]. — Прим. перев.

Пара $(a, b) \in R$ называется *дугой* (или *ребром*) графа G . Говорят, что дуга *выходит* из вершины a и *входит* в вершину b . Например, $(1, 2)$ — дуга приведенного выше графа. Если (a, b) — дуга, то говорят, что вершина a *предшествует* вершине b , а вершина b *следует за* вершиной a .

Не вполне точно можно сказать, что два графа равны, если их можно изобразить одинаково, пренебрегая именами (обозначениями) вершин. Формально равенство неупорядоченных ориентированных графов определяется следующим образом.

Определение. Пусть $G_1 = (A_1, R_1)$ и $G_2 = (A_2, R_2)$ — графы. Будем говорить, что G_1 и G_2 *равны* (или *изоморфны*), если существует такое биективное отображение $f: A_1 \rightarrow A_2$, что aR_1b тогда и только тогда, когда $f(a)R_2f(b)$. Иначе говоря, в графе G_1 из вершины a в вершину b ведет дуга тогда и только тогда, когда в графе G_2 из вершины, соответствующей a , ведет дуга в вершину, соответствующую b .

Часто вершинам и/или дугам графа приписывают некоторую информацию. Мы будем называть такую информацию *разметкой*, а соответствующий граф — *помеченным графом*.

Определение. Пусть (A, R) — граф. *Разметкой* графа называется пара функций f и g , где f (*разметка вершин*) отображает A в некоторое множество, а g (*разметка дуг*) отображает R в некоторое (возможно, отличное от первого) множество. Пусть $G_1 = (A_1, R_1)$ и $G_2 = (A_2, R_2)$ — *помеченные графы*¹⁾ с разметками (f_1, g_1) и (f_2, g_2) соответственно. Тогда G_1 и G_2 — *равные помеченные графы*, если существует такое биективное отображение $h: A_1 \rightarrow A_2$, что

(1) aR_1b тогда и только тогда, когда $h(a)R_2h(b)$ (т. е. G_1 и G_2 равны как непомеченные графы);

(2) $f_1(a) = f_2(h(a))$ (т. е. соответствующие вершины имеют одинаковые метки);

(3) $g_1((a, b)) = g_2((h(a), h(b)))$ (т. е. соответствующие дуги имеют одинаковые метки).

Во многих случаях метками снабжаются только вершины или только дуги. В такой ситуации считается, что множество значений функции f или соответственно g состоит из единственного элемента. Тогда условие (2) или соответственно (3) выполняется тривиальным образом.

Пример 0.22. Пусть $G_1 = (\{a, b, c\}, \{(a, b), (b, c), (c, a)\})$ и $G_2 = (\{0, 1, 2\}, \{(1, 0), (2, 1), (0, 2)\})$. Разметка графа G_1 опреде-

¹⁾ Говорят еще: *нагруженные графы*. — Прим. перев.

ляется формулами $f_1(a) = f_1(b) = X$, $f_1(c) = Y$, $g_1((a, b)) = g_1((b, c)) = \alpha$, $g_1((c, a)) = \beta$. Разметка графа G_2 определяется формулами $f_2(0) = f_2(2) = X$, $f_2(1) = Y$, $g_2((0, 2)) = g_2((2, 1)) = \alpha$, $g_2((1, 0)) = \beta$. Графы G_1 и G_2 показаны на рис. 0.7.

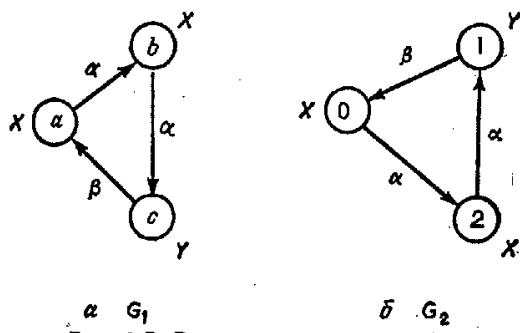


Рис. 0.7. Равные помеченные графы.

G_1 и G_2 равны. Соответствующее биективное отображение h определяется формулами $h(a) = 0$, $h(b) = 2$, $h(c) = 1$. \square

Определение. Последовательность вершин (a_0, a_1, \dots, a_n) , $n \geq 1$, называется *путем* (или *маршрутом*) длины n из вершины a_0 в вершину a_n , если для каждого $1 \leq i \leq n$ существует дуга, выходящая из вершины a_{i-1} и входящая в вершину a_i . Например, $(1, 2, 4, 3)$ — путь в графе G_1 , изображенном на рис. 0.6. Если существует путь из вершины a_0 в вершину a_n , то говорят, что a_n *достижима* из a_0 .

Циклом называется путь (a_0, a_1, \dots, a_n) , в котором $a_0 = a_n$. В графе на рис. 0.6 есть цикл $(1, 1)$ длины 1 и цикл $(1, 2, 4, 1)$ длины 3.

Граф называется *сильно связным*, если для любых двух разных вершин a и b существует путь из a в b .

Введем, наконец, понятие степени вершины. *Степенью по входу* вершины a назовем число входящих в нее дуг, а *степенью по выходу* — число выходящих из нее дуг.

0.5.2. Ориентированные ациклические графы

Определение. Ациклическим графом называется (ориентированный) граф, не имеющий циклов. На рис. 0.8 показан пример ациклического графа.

Вершину, степень по входу которой равна 0, назовем *базовой*. Вершину, степень по выходу которой равна 0, назовем *листом* (или *концевой* вершиной). На рис. 0.8 вершины 1, 2, 3 и 4 — базовые, а вершины 2, 4, 7, 8 и 9 — листья.

Если (a, b) — дуга ациклического графа, то a называется *прямым предком* b , а b — *прямым потомком* a .

Если в ациклическом графе существует путь из a в b , то говорят, что a — *предок* b , а b — *потомок* a . На рис. 0.8 вершина 9 — *потомок* вершины 1, вершина 1 — *предок* вершины 9.

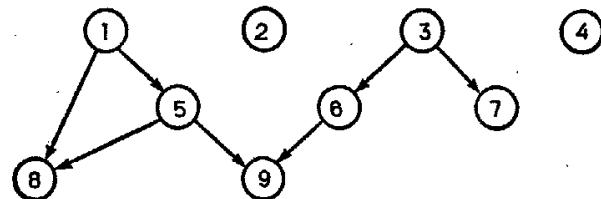


Рис. 0.8. Пример ациклического графа.

Заметим, что если R — частичный порядок на множестве A , то (A, R) — ациклический граф. Более того, если (A, R) — ациклический граф и R' — отношение „являться потомком“, определенное на A , то R' — частичный порядок на A .

0.5.3. Деревья

Дерево — это ациклический граф специального типа, имеющий много приложений в теории компиляторов.

Определение. (Ориентированным) деревом T называется (ориентированный) граф $G = (A, R)$ со специальной вершиной $r \in A$, называемой *корнем*, у которого

- (1) степень по входу вершины r равна 0,
- (2) степень по входу всех остальных вершин дерева T равна 1,
- (3) каждая вершина $a \in A$ достижима из r .

На рис. 0.9, *a* изображено дерево с шестью вершинами. Корень обозначен числом 1. Рисуя деревья, мы будем помещать корень вверху и направлять все дуги вниз. Приняв это соглашение, мы не будем указывать стрелки.

Теорема 0.3. Дерево T обладает следующими свойствами:

- (1) T — ациклический граф,
- (2) для каждой вершины дерева T существует единственный путь, ведущий из корня в эту вершину.

Доказательство. Упражнение. \square

Определение. Поддеревом дерева $T = (A, R)$ называется любое дерево $T' = (A', R')$, у которого

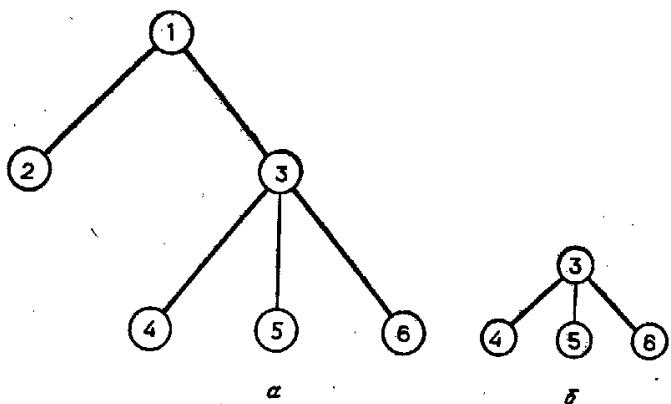


Рис. 0.9. Пример дерева.

- (1) A' непусто и содержится в A ,
- (2) $R' = (A' \times A') \cap R$,
- (3) ни одна вершина из множества $A - A'$ не является потомком вершины из множества A' .

Например, β на рис. 0.9—поддерево дерева α . Мы будем говорить, что корень поддерева *доминирует* над этим поддеревом.

0.5.4. Упорядоченные графы

Упорядоченным графом называется пара (A, R) , где A обозначает, как и раньше, множество вершин, а R —множество линейно упорядоченных списков дуг, каждый элемент которого имеет вид $((a, b_1), (a, b_2), \dots, (a, b_n))$. Этот элемент указывает, что из вершины a выходят n дуг, причем первой из них считается дуга, входящая в вершину b_1 , второй—дуга, входящая в вершину b_2 , и т. д.

Пример 0.23. На рис. 0.10 изображен упорядоченный граф. Линейное упорядочение дуг, выходящих из вершины, указывается

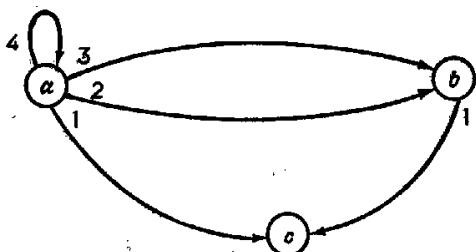


Рис. 0.10. Упорядоченный граф.

с помощью нумерации их числами $1, 2, \dots, n$, где n —степень по выходу этой вершины.

Формально упорядоченный граф на рис. 0.10 определяется как пара (A, R) , где $A = \{a, b, c\}$ и $R = \{((a, c), (a, b), (a, b), (a, a)), ((b, c))\}$. \square

Заметим, что упорядоченный граф на рис. 0.10 не является ориентированным в смысле нашего определения, так как в нем из вершины a в вершину b ведут две дуги. (Напомним, что во множестве дуг графа каждый элемент встречается только один раз.)

Как и для неупорядоченных графов, определим понятия разметки и равенства помеченных графов.

Определение. *Разметкой* упорядоченного графа $G = (A, R)$ назовем такую пару функций f и g , что

- (1) $f: A \rightarrow S$ для некоторого множества S (f помечает вершины);
- (2) g отображает R в последовательности символов из некоторого множества T так, что образом списка $((a_1, b_1), \dots, (a, b_n))$ является последовательность из n символов (g помечает дуги).

Помеченные графы $G_1 = (A_1, R_1)$ и $G_2 = (A_2, R_2)$ с разметками (f_1, g_1) и (f_2, g_2) соответственно назовем *равными*, если существует такое биективное отображение $h: A_1 \rightarrow A_2$, что

- (1) R_1 содержит $((a, b_1), \dots, (a, b_n))$ тогда и только тогда, когда R_2 содержит $((h(a), h(b_1)), \dots, (h(a), h(b_n)))$;
- (2) $f_1(a) = f_2(h(a))$ для всех $a \in A_1$;
- (3) $g_1(((a, b_1), \dots, (a, b_n))) = g_2(((h(a), h(b_1)), \dots, (h(a), h(b_n))))$.

Неформально можно сказать, что два помеченных упорядоченных графа равны, если существует взаимно однозначное соответствие между их вершинами, сохраняющее метки вершин и дуг. Если множества значений обеих разметок состоят из единственного элемента, то граф по существу не помечен, и надо проверять только условие (1). Аналогично, если одним элементом помечены только все вершины или только все дуги, то соответственно становится тривиальным условие (2) или (3).

Для каждого упорядоченного графа (A, R) существует неупорядоченный граф (A, R') (называемый его *основой*), дугами которого служат дуги графа (A, R) без учета порядка и без повторений, т. е. R' состоит из пар (a, b) , для которых существует список $((a, b_1), \dots, (a, b_n)) \in R$ и $b = b_i$ для некоторого i , $1 \leq i \leq n$.

Упорядоченный ациклический граф—это упорядоченный граф, основой которого является ациклический граф.

Упорядоченное дерево—это упорядоченный граф (A, R) , основой которого является дерево, и если $((a, b_1), \dots, (a, b_n)) \in R$, то $b_i \neq b_j$ для $i \neq j$.

Если не оговорено противное, то предполагается, что на диаграмме упорядоченного ациклического графа или дерева прямые потомки вершины всегда линейно упорядочены слева направо.



Рис. 0.11. Два дерева.

Когда рассматривается вопрос о равенстве двух графов, существенно, являются они упорядоченными или нет.

Например, два дерева T_1 и T_2 , изображенные на рис. 0.11, равны, если они неупорядоченные, и различны в противном случае.

0.5.5. Индукция по ациклическому графу

Многие теоремы об ациклических графах и особенно о деревьях можно доказывать индукцией, но часто бывает не ясно, по какому множеству вести индукцию. Теоремы, которые поддаются доказательствам такого sorta, часто имеют вид утверждений о том, что нечто истинно для всех вершин дерева или для некоторого подмножества вершин. Таким образом, нужно доказать утверждение о вершинах дерева и требуется какой-то параметр вершин, к которому можно применять шаг индукции.

В качестве таких параметров можно брать глубину вершины, т. е. длину минимального пути (или в случае дерева просто длину пути) от базовой вершины (в случае дерева от корня) до данной вершины, и высоту (или уровень) вершины, т. е. длину максимального пути от вершины до листа.

Другой подход к индукции по конечным упорядоченным деревьям состоит в том, чтобы каким-то образом упорядочить вершины и вести индукцию по позиции вершины в полученной последовательности. Опишем два распространенных способа упорядочения.

Определение. Пусть T — конечное упорядоченное дерево. Прямой порядок вершин дерева T представляет собой список (последовательность), который получается рекурсивным применением

Шаг 1: Пусть данное применение шага 1 относится к вершине a . Если a — лист, включить вершину a в список. Если a не лист и ее прямые потомки — вершины a_1, a_2, \dots, a_n , включить a в список и затем применить шаг 1 к вершинам a_1, a_2, \dots, a_n в этом порядке.

Обратный порядок вершин дерева T получится, если заключительную часть последнего предложения в описании шага 1 заменить на „применить шаг 1 к вершинам a_1, a_2, \dots, a_n в этом порядке, а затем включить в список вершину a “.

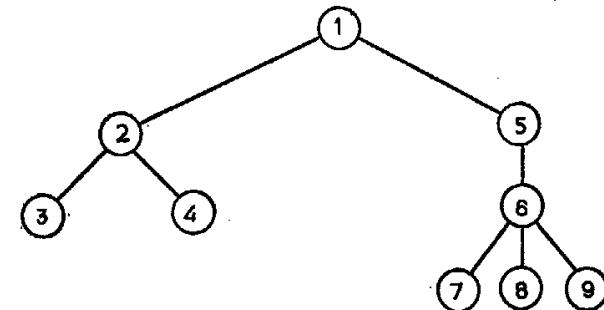


Рис. 0.12. Упорядоченное дерево.

Пример 0.24. Рассмотрим упорядоченное дерево на рис. 0.12. Прямой порядок вершин: 123456789; обратный порядок: 342789651. □

Иногда можно провести индукцию по месту, которое занимает вершина в том или ином упорядочении.

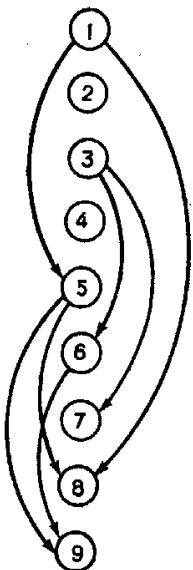
0.5.6. Вложение частичного порядка в линейный

Если задан частичный порядок R на множестве A , часто бывает нужен линейный порядок, содержащий этот частичный порядок. Эта проблема вложения частичного порядка в линейный называется *топологической сортировкой*.

Интуитивно топологическая сортировка заключается в том, что надо взять ациклический граф, который в сущности и является заданным частичным порядком, и расположить его вершины в столбец так, чтобы все дуги были направлены вниз. Линейный порядок задается положением вершин в этом столбце.

Например, при такой деформации граф, изображенный на рис. 0.8, может превратиться в столбец, показанный на рис. 0.13.

Формально можно сказать, что частичный порядок R на множестве A вложен в линейный порядок R' , если R' — линейный порядок и $R \subseteq R'$, т. е. aRb влечет $aR'b$ для всех a и b из A .



Для данного частичного порядка R существует много линейных порядков, в которые его можно вложить (упр. 0.5.5). Следующий алгоритм находит один из таких линейных порядков:

Алгоритм 0.1. Топологическая сортировка.

Вход. Частичный порядок R на конечном множестве A .

Выход. Линейный порядок R' на A , для которого $R \subseteq R'$.

Метод. Так как A — конечное множество, линейный порядок R' на A можно представить в виде списка a_1, a_2, \dots, a_n , для которого $a_i R' a_j$, если $i < j$, и $A = \{a_1, a_2, \dots, a_n\}$. Эта последовательность элементов строится с помощью следующих шагов:

- (1) Положить $i = 1$, $A_i = A$ и $R_i = R$.
- (2) Если A_i пусто, остановиться и выдать a_1, a_2, \dots, a_{i-1} в качестве искомого линейного порядка. В противном случае выбрать в A_i такой элемент a_i , что $a R_i a_i$ должно для всех $a \in A_i$.

(3) Положить $A_{i+1} = A_i - \{a_i\}$ и $R_{i+1} = R_i \cap (A_{i+1} \times A_{i+1})$. Затем увеличить i на единицу и повторить шаг 2. \square

Рис. 0.13. Линейный порядок, полученный из ациклического графа на рис. 0.8.

Если частичный порядок представлен в виде ациклического графа, то алгоритм 0.1 допускает особенно простую интерпретацию. На каждом шаге алгоритма (A_i, R_i) является ациклическим графом и a_i — его базовая вершина. Ациклический граф (A_{i+1}, R_{i+1}) образуется из (A_i, R_i) вычеркиванием вершины a_i и всех выходящих из нее дуг.

Пример 0.25. Пусть $A = \{a, b, c, d\}$ и $R = \{(a, b), (a, c), (b, d), (c, d)\}$. Так как a — единственная вершина, для которой $a' Ra$ можно при всех $a' \in A$, надо взять $a_1 = a$.

Тогда $A_2 = \{b, c, d\}$ и $R_2 = \{(b, d), (c, d)\}$; теперь в качестве a_2 можно взять либо b , либо c . Выберем $a_2 = b$. Тогда $A_3 = \{c, d\}$ и $R_3 = \{(c, d)\}$. Продолжая аналогично, найдем $a_3 = c$ и $a_4 = d$.

В результате получился линейный порядок $R' = \{(a, b), (b, c), (c, d), (a, c), (b, d), (a, d)\}$. \square

Теорема 0.4. Алгоритм 0.1 дает линейный порядок R' , в который вложен данный частичный порядок R .

Доказательство. Простое упражнение на индукцию. \square

0.5.7. Представления деревьев

Дерево является двумерной структурой, но во многих ситуациях удобно пользоваться лишь одномерными структурами данных. Следовательно, мы заинтересованы в том, чтобы иметь для деревьев одномерные представления, сохраняющие всю информацию, которая содержится в двумерных картинках. Под этим мы подразумеваем, что двумерную картинку можно воспроизвести по ее одномерному представлению.

Очевидно, что одно из одномерных представлений дерева $T = (A, R)$ — это запись самих множеств A и R .

Существуют и другие представления. Например, для указания глубины вершин дерева можно использовать вложенные скобки. Напомним, что глубиной вершины называется длина пути от корня до этой вершины. Глубина вершины 1 на рис. 0.9 равна 0, глубина вершины 3 равна 1, а вершина 6 находится на глубине 2. Глубиной (или высотой) дерева называется длина наибольшего пути. Глубина дерева на рис. 0.9 равна 2.

Используя скобки для указания глубины, можно представить изображенное на рис. 0.9 дерево в виде $1(2, 3(4, 5, 6))$. Такую запись будем называть левым скобочным представлением, так как поддерево представляется выражением, заключенным в скобки, а его корень записывается непосредственно слева от левой скобки.

Определение. Левое скобочное представление дерева T (означается $lter(T)$) можно получить, применяя к нему следующие рекурсивные правила.

(1) Если корнем дерева T служит вершина a с поддеревьями T_1, T_2, \dots, T_k , расположенными в этом порядке (их корни — прямые потомки вершины a), то $lter(T) = a(lter(T_1), lter(T_2), \dots, lter(T_k))$.

(2) Если корнем дерева T служит вершина a , не имеющая прямых потомков, то $lter(T) = a$.

Если в левом скобочном представлении стереть все скобки, получится прямой порядок вершин дерева.

Аналогично можно определить правое скобочное представление $rter(T)$ дерева T :

(1) Если корнем дерева T служит вершина a с поддеревьями T_1, T_2, \dots, T_k , то $rter(T) = (rter(T_1), rter(T_2), \dots, rter(T_k))a$.

(2) Если корнем дерева T служит вершина a , не имеющая прямых потомков, то $rter(T) = a$.

Таким образом, для дерева T на рис. 0.12 $rter(T) = ((3, 4) 2, ((7, 8, 9) 6) 5) 1$. В этом представлении прямой предок

вершины расположены непосредственно справа от первой правой скобки, заключающей эту вершину.

Другое представление дерева можно получить, составив список прямых предков его вершин $1, 2, \dots, n$ именно в этом порядке. Чтобы опознать корень, будем считать, что его предок — это 0.

Пример 0.26. Дерево, показанное на рис. 0.14, можно представить в виде 0122441777. Здесь 0 на первом месте указывает

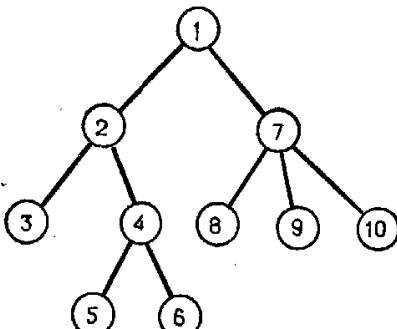


Рис. 0.14. Дерево.

на то, что прямым предком вершины 1 является „вершина 0“ (т. е. что вершина 1 — корень). Число 1 на седьмом месте говорит о том, что прямым предком вершины 7 является вершина 1. \square

0.5.8. Пути в графе

В этом разделе мы опишем эффективный с вычислительной точки зрения алгоритм построения транзитивного замыкания отношения R , определенного на множестве A . Если это отношение представлять себе в виде (неупорядоченного) графа (A, R) , то его транзитивное замыкание будет множеством пар вершин (a, b) , для которых существует путь из a в b .

Другая возможная интерпретация состоит в том, чтобы смотреть на отношение (или граф) как на квадратную булеву матрицу (т. е. матрицу из нулей и единиц), называемую *матрицей смежностей*, в которой на пересечении i -й строки и j -го столбца стоит 1 тогда и только тогда, когда элемент, соответствующий i -й строке, находится в отношении R с элементом, соответствующим j -му столбцу. На рис. 0.15 показана матрица смежностей M , соответствующая графу на рис. 0.6. Если M — матрица смеж-

ностей отношения R , то $M^+ = \sum_{n=1}^{\infty} M^n$ (где M^n — матрица M , умноженная¹⁾ на себя n раз) — матрица смежностей транзитивного замыкания этого отношения. Таким образом, алгоритм нахождения транзитивного замыкания можно применять и для вычисления M^+ .

Для матрицы M , изображенной на рис. 0.15, матрица M^+ состоит из одних единиц.

	1	2	3	4
1	1	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	1	0

Рис. 0.15. Матрица смежностей графа, изображенного на рис. 0.6.

Фактически мы приведем здесь несколько более общий алгоритм. Будем предполагать, что задан (неупорядоченный ориентированный) граф, в котором каждой дуге, ведущей из вершины i в вершину j , приписан вес (или стоимость) c_{ij} . (Если из i в j не ведет ни одна дуга, вес c_{ij} считается бесконечным.) Алгоритм будет вычислять для каждой пары вершин минимальный вес пути, ведущего из первой вершины пары во вторую. В том случае, когда мы хотим вычислить только транзитивное замыкание отношения R на множестве $\{a_1, a_2, \dots, a_n\}$, надо положить $c_{ij} = 0$, если $a_i Ra_j$, и $c_{ij} = \infty$ в противном случае.

Алгоритм 0.2. Минимальный вес (стоимость) путей в графе.

Вход. Граф с n вершинами, запущенными $1, 2, \dots, n$, и с весами $c_{ij} \geq 0$ для всех $1 \leq i, j \leq n$.

Выход. Матрица $M = [m_{ij}]$, в которой m_{ij} — минимальный из весов путей, ведущих из вершины i в вершину j ($1 \leq i, j \leq n$).

Метод.

- (1) Положить $m_{ij} = c_{ij}$ для всех i и j ($1 \leq i, j \leq n$).
- (2) Положить $k = 1$.

¹⁾ При этом используется обычная формула для умножения матриц, но с булевыми операциями · и + в качестве умножения и сложения.

- (3) Для всех i и j , если $m_{ij} > m_{ik} + m_{kj}$, положить $m_{ij} = m_{ik} + m_{kj}$.
 (4) Если $k < n$, увеличить k на единицу и перейти к шагу (3). Если $k = n$, остановиться. \square

Ядром алгоритма 0.2 является шаг (3), на котором проверяется, можно ли уменьшить стоимость (вес) перехода из вершины i в вершину j , перейдя сначала из вершины i в вершину k , а затем из вершины k в вершину j . Так как шаг (3) выполняется по одному разу для всевозможных значений i , j , k , то временная сложность алгоритма 0.2 равна n^3 .

Сразу не ясно, что алгоритм 0.5 находит минимальный вес путей, ведущих из i в j . Таким образом, надо доказать, что алгоритм 0.2 делает то, что требуется.

Теорема 0.5. *Когда алгоритм 0.2 останавливается, m_{ij} — наименьшая из величин, представимых в виде суммы $c_{v_1 v_2} + \dots + c_{v_{m-1} v_m}$, где $v_1 = i$ и $v_m = j$. (Эта сумма — вес пути v_1, v_2, \dots, v_m , ведущего из вершины i в вершину j .)*

Доказательство. Чтобы доказать эту теорему, докажем индукцией по значению l переменной k в шаге (3) нашего алгоритма следующее утверждение:

(0.5.1) После того как шаг (3) выполнен для $k = l$, значением m_{ij} служит наименьшая из величин, представимых в виде суммы $c_{v_1 v_2} + \dots + c_{v_{m-1} v_m}$, где $v_1 = i$, $v_m = j$ и ни одно из v_2, \dots, v_{m-1} не превосходит l .

Назовем эту наименьшую величину *корректным* значением m_{ij} для $k = l$. Эта величина — вес самого легкого (или стоимость самого дешевого) пути, ведущего из вершины i в вершину j , среди путей, не проходящих через вершины, номера которых больше l .

Базис. Рассмотрим начальное условие, которое имеет вид $l=0$. (Если угодно, можно рассматривать шаг (1) как шаг (3) для $k=0$.) Если $l=0$, то $m=2$, и значение $m_{ij} = c_{ij}$ является корректным начальным значением.

Шаг индукции. Предположим, что утверждение (0.5.1) истинно для $l < l_0$. Рассмотрим значение m_{ij} после того, как шаг (3) выполнен для $k = l_0$.

Допустим сначала, что значением m_{ij} для $k = l_0$ является такая наименьшая сумма $c_{v_1 v_2} + \dots + c_{v_{m-1} v_m}$, что ни одно v_p ($2 \leq p \leq m-1$) не равно l_0 . По предположению индукции $c_{v_1 v_2} + \dots + c_{v_{m-1} v_m}$ — корректное значение m_{ij} для $k = l_0 - 1$, и потому $c_{v_1 v_2} + \dots + c_{v_{m-1} v_m}$ — корректное значение m_{ij} также и для $k = l_0$.

Теперь предположим, что значением m_{ij} для $k = l_0$ является такая наименьшая сумма $s_{ij} = c_{v_1 v_2} + \dots + c_{v_{m-1} v_m}$, что $v_p = l_0$ для некоторого p , $2 \leq p \leq m-1$. Это значит, что s_{ij} — это вес пути v_1, v_2, \dots, v_m , для которой $q \neq p$ и $v_q = l_0$. В противном случае путь v_1, v_2, \dots, v_m содержал бы цикл и можно было бы удалить по крайней мере один член суммы $c_{v_1 v_2} + \dots + c_{v_{m-1} v_m}$, не увеличив значения s_{ij} . Таким образом, для s_{ij} всегда можно найти сумму, в которой $v_p = l_0$ только для одного значения p , $2 \leq p \leq m-1$.

Пусть $2 < p < m-1$. (Случай $p=2$ и $p=m-1$ оставляем читателю.) Рассмотрим суммы $s_{iv_p} = c_{v_1 v_2} + \dots + c_{v_{p-1} v_p}$ и $s_{v_p j} = c_{v_p v_{p+1}} + \dots + c_{v_{m-1} v_m}$, которые являются весами путей, ведущих из вершины i в вершину v_p и из вершины v_p в вершину j соответственно. По предположению индукции можно считать, что s_{iv_p} — корректное значение m_{iv_p} для $k = l_0 - 1$, а $s_{v_p j}$ — корректное значение $m_{v_p j}$ для $k = l_0 - 1$. Таким образом, когда шаг (3) выполняется для $k = l_0$, переменной m_{ij} присваивается корректное значение $m_{iv_p} + m_{v_p j}$.

Итак, мы показали, что утверждение (0.5.1) истинно для всех l . Когда $l=n$, утверждение (0.5.1) говорит о том, что в конце работы алгоритма 0.2 значением m_{ij} является наименьшая из возможных величин. \square

Распространенный частный случай нахождения минимума весов путей в графе — это случай, когда мы хотим найти множество вершин, достижимых из данной вершины. Эквивалентная формулировка: дано отношение R на множестве A и элемент $a \in A$; надо найти множество таких $b \in A$, что aR^+b , где R^+ — транзитивное замыкание отношения R . Для этой цели можно применить следующий алгоритм, имеющий квадратичную временную сложность.

Алгоритм 0.3. Нахождение множества вершин, достижимых из данной вершины a (ориентированного) графа.

Вход. Граф (A, R) , в котором A — конечное множество, и $a \in A$.
Выход. Множество таких вершин $b \in A$, что aR^+b .

Метод. Будет образован список L , меняющийся в ходе работы алгоритма. Кроме того, будут отмечаться некоторые элементы множества A . Вначале все элементы из A не отмечены.

(1) Положить $L = a$.

(2) Если список L пуст, остановиться. В противном случае вычеркнуть из L первый элемент b и отметить его в A .

(3) Поместить в конец списка L все неотмеченные элементы $c \in A$, для которых bRc , и перейти к шагу (2). \square

Доказательство того, что алгоритм 0.3 работает правильно, оставляем в качестве упражнения.

УПРАЖНЕНИЯ

0.5.1. Каково максимальное число дуг, которые может иметь ациклический граф с n вершинами?

0.5.2. Докажите теорему 0.3.

0.5.3. Постройте прямой и обратный порядки вершин дерева, изображенного на рис. 0.14. Напишите левое и правое скобочное представления этого дерева.

***0.5.4.** (а) Придумайте алгоритм, отображающий левое скобочное представление дерева в правое.

(б) Придумайте алгоритм, отображающий правое скобочное представление дерева в левое.

0.5.5. Во сколько разных линейных порядков можно вложить частичный порядок, заданный ациклическим графом на рис. 0.8?

0.5.6. Докажите теорему 0.4.

0.5.7. Укажите верхние границы времени и емкости, необходимых для реализации алгоритма 0.1, считая, что требуется одна ячейка памяти для хранения имени вершины или целого числа и один элементарный шаг для каждой операции из некоторого разумного множества примитивных операций, включающего арифметические операции и операции проверки или изменения ячейки массива, индексом которой является известное целое число.

0.5.8. Пусть $A = \{a, b, c, d\}$ и $R = \{(a, b), (b, c), (a, c), (b, d)\}$. Найдите линейный порядок R' , для которого $R \equiv R'$. Сколько существует таких линейных порядков?

Определение. Неориентированный граф G — это тройка (A, E, f) , где A — множество вершин, E — множество имен ребер и f — отображение множества E в множество неупорядоченных пар вершин. Запись $f(e) = \{a, b\}$ означает, что ребро e соединяет вершины a и b . Путем в неориентированном графе называется такая последовательность вершин $a_0, a_1, a_2, \dots, a_n$, что для каждого i , $1 \leq i \leq n$, найдется ребро, соединяющее a_{i-1} с a_i . Неориентированный граф называется связным, если для каждого двух его вершин существует соединяющий их путь.

Определение. Неориентированное дерево можно определить рекурсивно следующим образом. Неориентированное дерево — это

множество, состоящее из одной или более вершин, в котором выделена одна вершина r , называемая корнем. Остальные вершины разбиваются на нуль или более множеств T_1, \dots, T_k , каждое из которых образует дерево. Деревья T_1, \dots, T_k называются поддеревьями корня r и корень каждого такого поддерева соединен неориентированным ребром с r .

Остовом связного неориентированного графа G называется дерево с ребрами из графа G , содержащее все его вершины.

0.5.9. Напишите алгоритм построения остова связного неориентированного графа.

0.5.10. Пусть (A, R) — (неупорядоченный) граф, в котором $A = \{1, 2, 3, 4\}$ и $R = \{(1, 2), (2, 3), (4, 1), (4, 3)\}$. Найдите транзитивное замыкание R^+ отношения R . По матрице смежностей M отношения R вычислите M^+ и покажите, что M^+ — матрица смежностей графа (A, R^+) .

0.5.11. Покажите, что алгоритм 0.2 требует порядка n^3 элементарных шагов, подобных тем, которые упоминались в упр. 0.5.7.

0.5.12. Докажите, что алгоритм 0.3 отмечает вершину b тогда и только тогда, когда aR^+b .

0.5.13. Покажите, что алгоритму 0.3 требуется время, пропорциональное большему из чисел $\#A$ и $\#R$.

0.5.14. Какие из следующих трех неупорядоченных ориентированных графов равны?

$$G_1 = (\{a, b, c\}, \{(a, b), (b, c), (c, a)\})$$

$$G_2 = (\{a, b, c\}, \{(b, a), (a, c), (b, c)\})$$

$$G_3 = (\{a, b, c\}, \{(c, b), (c, a), (b, a)\})$$

0.5.15. Даны три упорядоченных ориентированных графа, у которых помечены только вершины. Какие из них равны?

$$G_1 = (\{a, b, c\}, \{((a, b), (a, c)), ((b, a), (b, c)), ((c, b))\})$$

с разметкой $l_1(a) = X$, $l_1(b) = Z$ и $l_1(c) = Y$.

$$G_2 = (\{a, b, c\}, \{((a, c)), ((b, c), (b, a)), ((c, b), (c, a))\})$$

с разметкой $l_2(a) = Y$, $l_2(b) = X$ и $l_2(c) = Z$.

$$G_3 = (\{a, b, c\}, \{((a, c), (a, b)), ((b, c)), ((c, a), (c, b))\})$$

с разметкой $l_3(a) = Y$, $l_3(b) = X$ и $l_3(c) = Z$.

0.5.16. Закончите доказательство теоремы 0.5.

0.5.17. Дайте алгоритм, определяющий, связан ли неориентированный граф.

***0.5.18.** Дайте алгоритм, определяющий, равны ли два графа.

*0.5.19. Постройте эффективный алгоритм, выясняющий, лежат ли две данные вершины дерева на одном пути. Указание: Рассмотрите прямой порядок вершин.

**0.5.20. Постройте эффективный алгоритм нахождения ближайшего общего предка двух данных вершин дерева.

Упражнения на программирование

0.5.21. Напишите программу, которая будет строить матрицу смежностей по представлению графа, заданному в виде связанного списка.

0.5.22. Напишите программу, которая по матрице смежностей будет строить представление графа в виде связанного списка.

0.5.23. Напишите программы, реализующие алгоритмы 0.1—0.3.

Замечания по литературе

Графы представляют собой старую и почетную часть математики. Теория графов излагается в книгах Харари [1969], Оре [1962] и Бержа [1958]¹⁾. Техника обращения с графами и деревьями внутри вычислительной машины хорошо освещена Кнутом [1968].

Алгоритм 0.2 — это алгоритм Уоршолла в том виде, как он описан у Флойдом [1962а]. Один интересный результат, касающийся вычисления транзитивного замыкания отношения, приведен в работе Мунро [1971], где показано, что транзитивное замыкание можно вычислить за время, требуемое для вычисления произведения двух матриц над булевым кольцом. Отсюда, используя результат Штрассена [1969], получаем, что временная сложность транзитивного замыкания не превосходит $n^2.81$, а не n^3 как для алгоритма 0.2.

¹⁾ Рекомендуется также книга Зыкова [1969]. — Прим. перев.

ВВЕДЕНИЕ В КОМПИЛЯЦИЮ

В этой книге рассматриваются проблемы, связанные с отображением одного представления алгоритма в другое. Один из самых распространенных примеров такого отображения — компиляция исходной программы, написанной на языке программирования высокого уровня, в объектный код конкретной вычислительной машины.

Мы обсудим технику трансляции алгоритмов, применяемую при конструировании компиляторов и других устройств, предназначенных для работы с языками. Чтобы показать эту технику в надлежащей перспективе, мы резюмируем здесь узловые аспекты процесса компиляции и упомянем о других областях, где синтаксический анализ или трансляция играют главную роль.

Как и в предыдущей главе, те, кто уже знаком с материалом (в данном случае с компиляторами), обнаружат, что изложение довольно элементарно. Такие читатели могут пропустить эту главу или только просмотреть ее для ознакомления с терминологией.

1.1. ЯЗЫКИ ПРОГРАММИРОВАНИЯ

В этом разделе мы кратко обсудим понятие языка программирования, а затем коснемся проблем, связанных с заданием языка программирования и построением транслятора для такого языка.

1.1.1. Задание языков программирования

Операции машинного языка цифровой вычислительной машины неизменно оказываются гораздо более примитивными по сравнению со сложными функциями, встречающимися в математике, технике и других областях. Хотя любую функцию, которую можно задать алгоритмом, можно реализовать в виде последовательности чрезвычайно простых команд машинного языка,

в большинстве приложений гораздо предпочтительнее использовать язык высокого уровня, элементарные команды которого приближаются к типу операций, встречающихся в приложениях. Например, если выполняются матричные операции, то для выражения того обстоятельства, что матрица A получается перемножением матриц B и C, удобнее написать команду вида

$$A = B * C$$

чем длинную последовательность операций машинного языка, предназначенную для выполнения того же умножения.

Языки программирования могут значительно облегчить тяжелую и нудную работу, связанную с программированием на машинном языке, но вместе с ними появляется ряд новых присущих им проблем. Само собой разумеется, что так как вычислительные машины пока могут „понимать“ только машинный язык, то программу, написанную на языке высокого уровня, надо в конце концов перевести (транслировать) на машинный язык. Устройство, выполняющее этот перевод, называют компилятором.

Другая проблема, связанная с языком программирования,— это проблема задания самого языка. Задавая язык программирования, мы, как минимум, должны определить

- (1) множество символов, которые можно использовать для записи правильных программ,
- (2) множество правильных программ,
- (3) „смысл“ каждой правильной программы.

Допустимое множество символов определить легко. Однако надо иметь в виду, что в некоторых языках, таких, как Снобол или Фортран, должны прииматься во внимание начало и/или конец перфокарты и, следовательно, они должны рассматриваться как символы. Пробел тоже в некоторых случаях считается символом. Определить множество программ, которые следует считать „правильными“, гораздо более трудно. Во многих случаях как раз трудно решить, считать ли правильной данную программу.

При задании языков программирования стало уже обычным определять класс допустимых программ с помощью грамматических правил, позволяющих строить и такие программы, правильность которых сомнительна. Например, многие определения Фортрана допускают оператор вида

L GOTO L

в качестве части „правильной“ программы Фортрана. Однако задать множество, содержащее все действительно правильные программы, но не только их, часто бывает гораздо проще, чем задать все те и только те программы, которые считаются правильными в самом строгом смысле слова.

Третий и самый трудный аспект задания языка—определение смысла каждой правильной программы. К решению этой проблемы было предпринято несколько подходов. Один из методов заключается в определении отображения, связывающего с каждой правильной программой предложение в языке, смысл которого мы понимаем. Например, в качестве „полне понятного“ языка можно взять функциональное исчисление или лямбда-исчисление. Тогда можно определить смысл программы, записанной на любом языке программирования, в терминах эквивалентной „программы“ в функциональном исчислении или лямбда-исчислении. Под *эквивалентной программой* подразумевается программа, определяющая ту же функцию.

Другой способ придать смысл программам заключается в определении идеализированной машины. Тогда смысл программы выражается в тех действиях, к которым она побуждает эту машину после того, как та начинает работу в некоторой предопределенной начальной конфигурации. В этой схеме интерпретатором данного языка становится абстрактная машина.

Третий подход—вообще игнорировать глубокие вопросы о „смысле“, и именно этот подход мы выберем здесь. Для нас „смысл“ исходной программы состоит просто в выходе компилятора, когда он применяется к этой программе.

Мы будем исходить из предположения, что компилятор задает как множество пар (x, y) , где x —программа в исходном языке, а y —программа в том языке, на который нужно перевести x . Предполагается, что мы заранее знаем это множество и наша главная забота—построить эффективное устройство, которое по данному входу x выдает выход y . Мы будем называть это множество пар (x, y) *переводом*. Если x —цепочка в алфавите Σ , а y —цепочка в алфавите Δ , то перевод—это просто отображение множества Σ^* в Δ^* .

1.1.2. СИНТАКСИС И СЕМАНТИКА

При определении и реализации переводов часто бывает удобнее рассматривать перевод как композицию двух более простых отображений. Первое из них, называемое *синтаксическим отображением*, связывает с каждым входом (программой в исходном языке) некоторую структуру, которая служит аргументом второго отображения, называемого *семантическим*. Сразу не ясно, что должна существовать какая-то структура, помогающая осуществить перевод, но почти всегда той структурой, которую полезно придать входной программе, оказывается помеченное дерево. Не углубляясь в философию о том, почему это так, мы посвятим большую часть книги алгоритмам эффективного построения подходящих деревьев для входных программ.

В качестве примера того, как для цепочек строятся древовидные структуры, рассмотрим разбиение любого английского предложения на синтаксические категории согласно грамматическим правилам. Например, предложение

The pig is in the rep

имеет грамматическую структуру, представленную в виде дерева на рис. 1.1. Неконцевые вершины этого дерева помечены синтаксическими категориями, а концевые вершины, т. е. листья,

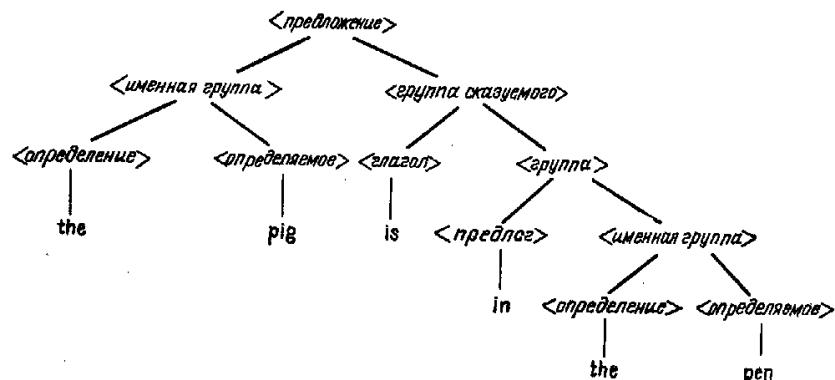


Рис. 1.1. Древовидная структура английского предложения.

помечены концевыми, или терминальными, символами, которые в данном случае являются английскими словами.

Подобным же образом программу, написанную на языке программирования, можно расчленить на синтаксические компоненты в соответствии с синтаксическими правилами, управляющими этим языком. Например, цепочка

$$a + b * c$$

может иметь синтаксическую структуру, заданную деревом на рис. 1.2¹⁾). Процесс нахождения синтаксической структуры заданного предложения называют *синтаксическим анализом* или *разбором*. Синтаксическая структура предложения помогает понять взаимоотношения между различными частями предложения.

¹⁾ Использование трех синтаксических категорий <выражение>, <терм> и <множитель> вместо одной категории <выражение> вызвано желанием обеспечить однозначность синтаксической структуры арифметического выражения. Читатель должен иметь это в виду, иначе ему покажутся чересчур сложными наши последующие примеры синтаксического анализа арифметических выражений.

Термином *синтаксис* языка будем называть отношение, связывающее с каждым предложением языка некоторую синтаксическую структуру. Тогда правильное предложение языка можно определить как цепочку символов, синтаксическая структура

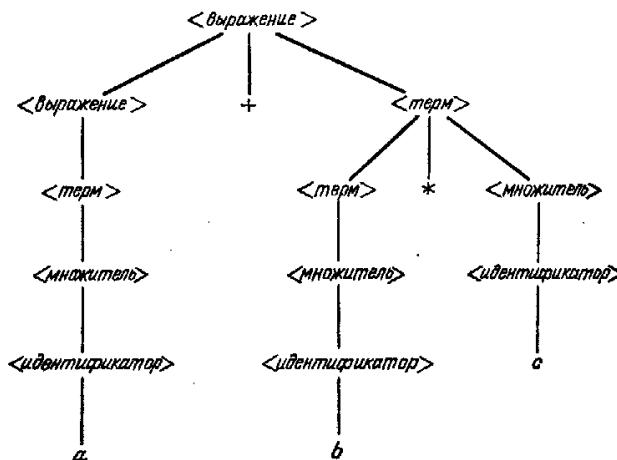


Рис. 1.2. Дерево арифметического выражения.

которой соответствует категория <предложение>. В следующей главе мы рассмотрим несколько методов строгого определения синтаксиса языка.

Вторая часть перевода называется *семантическим отображением*; оно отображает структурированный вход в выход, который обычно является программой в машинном языке. Термином *семантика языка* будем называть отображение, связывающее синтаксической структурой каждой входной цепочки цепочку в некотором языке (возможно, в том же самом), рассматриваемую как „смысл“ первоначальной цепочки. Задание семантики языка — очень трудная проблема, пока еще далекая от полного решения, в особенности для естественных языков, например английского.

Даже задание синтаксиса и семантики языка программирования — задача нетривиальная. Хотя универсально применимых методов нет, в теории языков есть два понятия, которые можно использовать для разработки части необходимого описания.

Первое из них — понятие контекстно-свободной грамматики. В виде контекстно-свободной грамматики можно формализовать большую часть правил, предназначенных для описания синтаксической структуры. Более того, контекстно-свободная грамматика обеспечивает описание, достаточно точное для того, чтобы его можно было использовать как часть определения самого

компилятора. Соответствующие понятия теории контекстно-свободных языков будут изложены в гл. 2.

Второе понятие — схема синтаксически управляемого перевода, с помощью которой можно задавать отображения одного языка в другой. Схемы синтаксически управляемого перевода мы изучим довольно подробно в гл. 3 и 9.

В этой книге предпринята попытка изложить аспекты теории языков и других формальных теорий, имеющие отношение к созданию языков программирования и компиляторов для них. Воздействие теории состоит в том, что в одних случаях она обеспечивает язык, в рамках которого можно говорить о проблемах, возникающих при построении компиляторов, а в других — единообразные и практически применимые решения некоторых из этих проблем.

Замечания по литературе

Языки программирования высокого уровня появились в начале 1950-х годов. В это время вычислительные машины нуждались в арифметических операциях с плавающей точкой, так что первые языки программирования служили для представления этих операций. Первым из главных языков программирования был Фортрај, который появился в середине 1950-х годов. Тогда же было создано несколько других алгебраических языков, но Фортрај нашел наиболее широкое применение. С тех пор появились сотни языков программирования высокого уровня. В книге Саммета [1969] дается обзор многих языков, существовавших в середине 1960-годов.

Теория языков программирования и компиляторов значительно отставала от практики. Сильным стимулом для развития теории формальных языков стало использование при определении синтаксиса Алгола 60 [Наур, 1963] того, что теперь называют формой Бэкуса — Наура (БНФ)¹. Сообщение об Алголе 60 вместе с ранними работами Хомского [1959а, 1963] вызвало бурное развитие теории формальных языков в 1960-е годы. Большая часть нашей книги посвящена результатам теории языков, связанным с построением и пониманием трансляторов для языков программирования.

Большинство ранних работ по теории языков касалось синтаксического аспекта определения языков. Определение семантики языков — проблема гораздо более трудная — привлекло меньше внимания и даже к моменту написания данной книги не было вполне решенным делом². Хорошие антологии по вопросам формального задания семантики — [Стил, 1966] и [Эигелер, 1971]. Определение языка ПЛ/1, разработанное в Венской лаборатории фирмы ИБМ [Лукаш и Вальк, 1969], служит примером полностью формального подхода к определению большого языка программирования.

Очень интересным результатом в области языков программирования было создание расширяемых языков, синтаксис и семантику которых можно менять внутри программы. Одна из самых ранних и наиболее известных схем расширения языка — макропределение (см., например, [Мак-Илрой, 1960], [Ливенворт, 1936] и [Читэм, 1966]). Галлер и Перлис [1957] предложили

¹⁾ То же самое чаще неправильно, как отметил Кнут [1964], называют формальной формой Бэкуса. — Прим. перев.

²⁾ С тех пор появилась обширная литература, посвященная вопросам формальной семантики. Однако их рассмотрение выходит за рамки данной книги. — Прим. перев.

схему расширения, посредством которой в Алгол можно ввести новые типы данных и новые операторы. Более поздние результаты, касающиеся расширяемых языков, содержатся в работах Кристенсена и Шоу [1969] и Вегбрейта [1970]. В качестве примера большого языка программирования, в котором есть средства расширения языка, можно привести Алгол 68 [ван Вейнгаарден, 1969].

1.2. ОВЗОР ПРОЦЕССА КОМПИЛЯЦИИ

Мы рассмотрим технические приемы и алгоритмы, применимые при построении компиляторов и других средств обработки языков. Чтобы представить эти алгоритмы в надлежащей перспективе, мы набросаем в этом разделе общую картину процесса компиляции.

1.2.1. Основные части компилятора

Во многих компиляторах для многих языков программирования есть общие процессы. Попытаемся выделить сущность некоторых из этих процессов. При этом мы постараемся устраниТЬ из них по возможности все, что связано с конкретной реализацией и зависит от машины и операционной системы. Хотя соображения, относящиеся к реализации, важны (плохая реализация может испортить хороший алгоритм), нам кажется, что понимание фундаментальной природы проблемы существенно само по себе и позволяет применить технику, созданную для решения этой проблемы, к другим проблемам, по существу сходным с нею.

Исходная программа, написанная на некотором языке программирования, есть не что иное, как цепочка знаков. Компилятор в конечном итоге превращает эту цепочку знаков в цепочку битов — объектный код. В этом процессе часто можно выделить подпроцессы со следующими названиями:

- (1) Лексический анализ.
- (2) Работа с таблицами.
- (3) Синтаксический анализ, или разбор.
- (4) Генерация кода, или трансляция в промежуточный код (например, язык ассемблера).
- (5) Оптимизация кода.
- (6) Генерация объектного кода (например, ассемблирование).

В конкретных компиляторах порядок этих процессов может несколько отличаться от указанного, а некоторые из них могут объединяться в одну фазу. Кроме того, никакая входная цепочка не должна нарушать работу компилятора, т.е. он должен обладать способностью реагировать на любую из них. Для входных цепочек, не являющихся синтаксически правильными програм-

мами, компилятор должен выдать соответствующие сообщения об ошибках.

Мы кратко опишем первые пять фаз компиляции. В реальном компиляторе они не обязательно разделены. Однако методически часто оказывается удобным расчленить компилятор на эти фазы, чтобы изолировать проблемы, присущие именно этим частям процесса компиляции.

1.2.2. Лексический анализ

Первая фаза — лексический анализ. Входом компилятора, а следовательно, и лексического анализатора, служит цепочка символов некоторого алфавита. В версии языка ПЛ/I, предназначенной для публикаций, алфавит терминальных символов содержит 60 знаков:

```
A B C ... Z $ @ #
0 1 2 ... 9 _ пробел
= + - * / ( ) , . ; : ' & | ¬ > < ? %
```

В программе некоторые комбинации символов часто рассматриваются как единые объекты. Среди типичных примеров можно указать следующие:

1) В таких языках, как ПЛ/I, цепочка, состоящая из одного или более пробелов, обычно рассматривается как один пробел.

2) В некоторых языках есть ключевые слова, такие, как BEGIN, END, GOTO, DO, INTEGER и т. д., каждое из которых считается одним символом.

3) Каждая цепочка, представляющая числовую константу, рассматривается как один элемент текста.

4) Идентификаторы, используемые как имена переменных, функций, процедур, меток и т. п., также считаются лексическими единицами языка программирования.

Работа лексического анализатора состоит в том, чтобы группировать определенные терминальные символы в единые синтаксические объекты, называемые **лексемами**. Какие объекты считать лексемами, зависит от определения языка программирования. Лексема — это цепочка терминальных символов, с которой мы связываем лексическую структуру, состоящую из пары вида (тип-лексемы, некоторые данные). Первой компонентой пары является синтаксическая категория, такая, как „константа“ или „идентификатор“, а вторая — указатель: в ней указывается адрес ячейки, хранящей информацию об этой конкретной лексеме. Для данного языка число типов лексем предполагается конечным. Пару (тип-лексемы, указатель) тоже будем называть лексемой, когда это не будет вызывать недоразумений.

Таким образом, лексический анализатор — это транслятор, входом которого служит цепочка символов, представляющая исходную программу, а выходом — последовательность лексем. Этот выход образует вход синтаксического анализатора.

Пример 1.1. Рассмотрим следующий оператор присваивания из языка, подобного Фортрану:

$$\text{COST} = (\text{PRICE} + \text{TAX}) * 0.98$$

На этапе лексического анализа будет обнаружено, что COST, PRICE и TAX — лексемы типа идентификатора, а 0.98 — лексема типа константы. Знаки = (+) * сами являются лексемами. Допустим, что все константы и идентификаторы нужно отобразить в лексемы типа **<ид>**. Мы предполагаем, что вторая компонента лексемы представляет собой указатель элемента таблицы, содержащего фактическое имя идентификатора вместе с другими собранными нами данными об этом конкретном идентификаторе. Первая компонента используется синтаксическим анализатором для разбора. Вторая компонента используется на этапе генерации кода для изготовления подходящего машинного кода.

Таким образом, выходом лексического анализатора, работающего на нашей входной цепочке, будет последовательность лексем

$$\langle \text{id} \rangle_1 = \langle \text{id} \rangle_2 + \langle \text{id} \rangle_3 * \langle \text{id} \rangle_4$$

Здесь вторая компонента лексемы (указатель данных) показана в виде нижнего индекса. Символы = (+) * трактуются как лексемы, тип которых представлен ими самими. Они не имеют связанных с ними данных и, значит, не имеют указателей. □

Лексический анализ провести легко, если лексемы, состоящие более чем из одного знака, изолированы с помощью знаков, которые сами являются лексемами. В примере 1.1 знаки = (+) * не могут быть частью идентификатора, так что COST, PRICE и TAX легко выделяются как лексемы.

Однако в общем случае лексический анализ может оказаться не таким легким. Например, рассмотрим следующие правильные операторы Фортрана:

- (1) DO 10 I = 1.15
- (2) DO 10 I = 1, 15

В операторе (1) цепочка DO 10 I — переменная¹), а цепочка 1.15 — константа. В операторе (2) DO — ключевое слово, 10 — константа, I — переменная, 1 и 15 — константы.

Если бы лексический анализатор был реализован как сопрограмма [Джентльмен, 1971; Мак-Илрой, 1968] и должен был на-

¹⁾ Напомним, что в Фортране пробелы игнорируются.

чать работу, находясь в начале одного из этих операторов, с выполнения такой команды, как „найти очередную лексему“, то он до тех пор не мог бы определить, является этой лексемой DO или DO 10 I, пока не дошел бы до запятой или точки.

Таким образом, лексический анализатор иногда должен заглядывать вперед за интересующую его в данный момент лексему. Еще худший пример встречается в ПЛ/І, где ключевые слова могут быть переменными. Глядя на входную цепочку вида

DECLARE (X1, X2, ..., X_n)

упомянутый выше лексический анализатор не знал бы, что ему сказать: то ли DECLARE—идентификатор функции и X₁, X₂, ..., X_n—аргументы этой функции, то ли DECLARE—ключевое слово, требующее, чтобы у идентификаторов X₁, X₂, ..., X_n был атрибут (или атрибуты), расположенный непосредственно справа от правой скобки. Здесь различие лексем должно осуществляться с помощью части текста, которая идет после правой скобки. Но так как число n может быть сколь угодно большим¹⁾, то, работая с языком ПЛ/І, этот лексический анализатор должен заглядывать сколь угодно далеко. Однако существует другой подход к лексическому анализу, менее удобный, но позволяющий избежать проблемы произвольно далекого заглядывания вперед.

Мы определим два крайних подхода к лексическому анализу. Большинство известных способов основано на том или другом из этих подходов, а некоторые на их комбинации.

(1) Говорят, что лексический анализатор работает *прямо*, если для данного входного текста (цепочки) и положения указателя в этом тексте анализатор определяет лексему, расположенную непосредственно справа от указываемого места, и сдвигает указатель вправо от части текста, образующей эту лексему.

(2) Говорят, что лексический анализатор работает *не прямо*, если для данного текста, положения указателя в этом тексте и типа лексемы он определяет, образуют ли знаки, расположенные непосредственно справа от указателя, лексему этого типа. Если да, то указатель передвигается вправо от части текста, образующей эту лексему.

Пример 1.2. Рассмотрим текст из Фортрана

DO 10 I=1, 15

с указателем, расположенным на левом конце. Непрямой лексический анализатор ответит „да“, если его спросят о лексеме

¹⁾ В определении языка верхняя граница для n не указывается, однако в каждом конкретном компиляторе с языка ПЛ/І она, конечно, существует.

типа DO или о лексеме типа *идентификатор*. Но в первом случае указатель передвинется на два символа вправо, а в последнем—на пять символов.

Прямой лексический анализатор обследует текст вплоть до запятой и сделает заключение, что очередная лексема должна быть типа DO. Указатель при этом передвинется только на два символа вправо, хотя было просмотрено гораздо больше символов. □

Вообще мы будем описывать алгоритмы синтаксического анализа в предположении, что лексический анализ прямой. В случае непрямого лексического анализа можно использовать „недетерминированные“ алгоритмы или алгоритмы с возвратами. Синтаксический анализ этого типа мы обсудим в гл. 4 и 6.

1.2.3. Работа с таблицами

После того как в результате лексического анализа лексемы распознаны, информация о некоторых из них собирается и запасается в одной или нескольких таблицах. Какова эта информация, зависит от языка. В случае Фортрана, например, мы хотели бы знать, что COST, PRICE и TAX—переменные с плавающей точкой, а 0.98—константа с плавающей точкой.

Допустим, что COST, PRICE и TAX не описаны оператором описания типа. Тогда необходимую информацию об этих переменных можно извлечь из того, что COST, PRICE и TAX начинаются с букв, отличных от I, J, K, L, M, N.

В качестве другого примера на сбор информации о переменных рассмотрим оператор Фортрана DIMENSION вида

DIMENSION A(10,20)

Встретив этот оператор, мы должны запастись информацией о том, что A—идентификатор, являющийся именем двумерного массива с размерами 10 и 20.

В сложных языках, таких, как ПЛ/І, объем сведений, которые надо запомнить о данной переменной, может быть очень велик.

Рассмотрим несколько упрощенный пример таблицы, в которой хранится информация об идентификаторах. Такую таблицу часто называют *таблицей имен* (а также *таблицей идентификаторов* и *таблицей символов*). В ней перечислены, в частности, все идентификаторы вместе с относящейся к ним информацией.

Допустим, что в тексте встречается оператор

COST = (PRICE + TAX) * 0.98

После просмотра этого оператора таблица может иметь вид табл. 1.1.

Если позднее во входной цепочке попадется идентификатор, надо справиться в этой таблице, не появлялся ли он раньше.

Если да, то лексема, соответствующая новому вхождению этого идентификатора, будет той же, что и у предыдущего вхождения. Например, если в программе, написанной на Фортране, после

Таблица 1.1

Номер элемента	Идентификатор	Информация
1	COST	Переменная с плавающей точкой
2	PRICE	Переменная с плавающей точкой
3	TAX	Переменная с плавающей точкой
4	0.98	Константа с плавающей точкой

указанного выше оператора следует оператор, содержащий переменную COST, то лексемой для второго вхождения COST должна быть $\langle\text{ид}\rangle_1$ — та же, что и для первого вхождения.

Таким образом, эта таблица должна обеспечивать

- (1) быстрое добавление новых идентификаторов и новых сведений о них,
- (2) быстрый поиск информации, относящейся к данному идентификатору.

Обычно применяют метод хранения данных с помощью таблиц расстановки; они будут обсуждаться в гл. 10 (том 2).

1.2.4. Синтаксический анализ

Как уже упоминалось, выходом лексического анализатора является цепочка лексем. Эта цепочка образует вход синтаксического анализатора, исследующего только первые компоненты лексем — их типы. Информация о каждой лексеме (вторая компонента) используется на более позднем этапе процесса компиляции для генерации машинного кода.

Синтаксический анализ, или разбор, как его еще называют, — это процесс, в котором исследуется цепочка лексем и устанавливается, удовлетворяет ли она структурным условиям, явно сформулированным в определении синтаксиса языка.

Какова синтаксическая структура данной цепочки, существенно знать также и при генерации кода. Например, синтаксическая структура выражения $A + B * C$ должна отражать тот факт, что сначала перемножаются B и C , а потом результат складывается с A . При любом другом порядке операций нужное вычисление не получится.

Разбор — одна из наиболее понятных фаз компиляции. По совокупности синтаксических правил можно автоматически построить синтаксический анализатор, который будет проверять, имеет ли исходная программа синтаксическую структуру, определяемую этими правилами. В гл. 4—7 мы изложим несколько различных методов разбора и алгоритмов построения синтаксических анализаторов по заданной грамматике.

Выходом анализатора служит дерево, которое представляет синтаксическую структуру, присущую исходной программе. В некотором отношении синтаксический анализ программы напоминает разбор предложений, который все мы проводили в школе.

Пример 1.3. Допустим, что выход лексического анализатора — цепочка лексем

$$(1.2.1) \quad \langle\text{ид}\rangle_1 = (\langle\text{ид}\rangle_2 + \langle\text{ид}\rangle_3) * \langle\text{ид}\rangle_4$$

Эта цепочка передает информацию о том, что необходимо выполнить в точности следующее:

- (1) $\langle\text{ид}\rangle_3$ прибавить к $\langle\text{ид}\rangle_2$,
- (2) результат (1) умножить на $\langle\text{ид}\rangle_4$,
- (3) результат (2) поместить в ячейку, зарезервированную для $\langle\text{ид}\rangle_1$.

Эту последовательность шагов можно представить паглядно с помощью помеченного дерева, показанного на рис. 1.3. Внут-

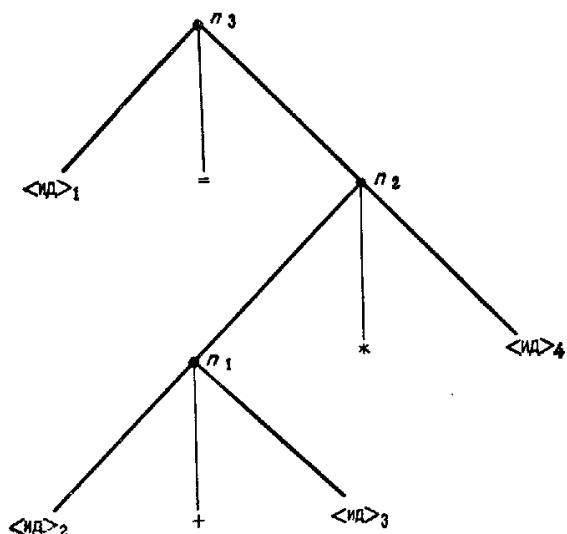


Рис. 1.3. Древовидная структура.

ренние вершины дерева представляют те действия, которые надо выполнить. Прямые потомки каждой вершины либо представляют аргументы, к которым нужно применить действие (если соответствующая вершина помечена идентификатором или является внутренней), либо помогают определить, каким должно быть это действие (в частности, это делают знаки $+$, $*$, $=$). Заметим, что скобки в цепочке (1.2.1) в дереве явно не указаны, хотя мы могли бы изобразить их в качестве прямых потомков вершины n_1 . Роль скобок только в том, что они влияют на порядок операций. Если бы в цепочке (1.2.1) их не было, следовало бы поступить согласно обычному соглашению о том, что умножение „предшествует“ сложению, и на первом шаге перемножить $\langle \text{ид} \rangle_3$ и $\langle \text{ид} \rangle_4$. \square

1.2.5. Генерация кода

Дерево, построенное синтаксическим анализатором, используется для того, чтобы получить перевод входной программы. Этот перевод может быть программой в машинном языке, но чаще он бывает программой в промежуточном языке, таком, как язык ассемблера или „трехадресный код“ (последний образуется из простых операторов, каждый из которых включает не более трех идентификаторов; например, $A = B$, $A = B + C$ или $\text{GOTO } A$).

Если требуется, чтобы компилятор произвел существенную оптимизацию кода, то предпочтительнее код трехадресного типа. Так как трехадресный код не привязывает вычисления к конкретным регистрам вычислительной машины, регистры легче использовать для более эффективной оптимизации. Если предполагается малая оптимизация или никакая, то в качестве промежуточного языка лучше взять язык ассемблера или даже машинный код. Для того чтобы проиллюстрировать узловые моменты процесса трансляции, рассмотрим бегло пример трансляции на язык ассемблерного типа.

Пусть в этом примере наша машина имеет один рабочий регистр (сумматор, или регистр результата) и команды языка ассемблера, вид которых определен в табл. 1.2. Запись „ $c(m) \rightarrow$ сумматор“ означает, что содержимое ячейки памяти m надо поместить в сумматор. Через $= m$ обозначено численное значение m . Из этих замечаний ясен смысл всех семи команд.

Выходом синтаксического анализатора служит дерево (или некоторое представление дерева), представляющее синтаксическую структуру цепочки лексем, полученной на выходе лексического анализатора. С помощью этого дерева и информации, хранящейся в таблице имен, можно построить объектный код. На практике построение дерева и генерация кода часто осущест-

вляются одновременно, но методически удобнее считать, что они происходят последовательно.

Существует несколько методов построения промежуточного кода по синтаксическому дереву. Один из них, называемый *синтаксически управляемым переводом (трансляцией)*, особенно изящен и эффективен. В нем с каждой вершиной n связывается

Таблица 1.2

Команда	Действие
LOAD m	$c(m) \rightarrow$ сумматор
ADD m	$c(\text{сумматор}) + c(m) \rightarrow$ сумматор
MPY m	$c(\text{сумматор}) * c(m) \rightarrow$ сумматор
STORE m	$c(\text{сумматор}) \rightarrow m$
LOAD $=m$	$m \rightarrow$ сумматор
ADD $=m$	$c(\text{сумматор}) + m \rightarrow$ сумматор
MPY $=m$	$c(\text{сумматор}) * m \rightarrow$ сумматор

Предполагается, что ADD и MPY — операции с плавающей точкой.

цепочка $C(n)$ промежуточного кода. Код для вершины n строится сцеплением в фиксированном порядке кодовых цепочек, связанных с прямыми потомками вершины n , и некоторых других фиксированных цепочек. Процесс перевода идет, таким образом, снизу вверх (т. е. от листьев к корню). Фиксированные цепочки и фиксированный порядок задаются используемым для перевода алгоритмом. Подробнее об этом будет изложено в гл. 3 и 9.

Здесь возникает важная проблема: для каждой вершины n выбрать код $C(n)$ так, чтобы код, приписываемый корню, оказался искомым кодом всего оператора. Вообще говоря, нужна какая-то интерпретация кода $C(n)$, которой можно было бы единообразно пользоваться во всех ситуациях, где может встретиться вершина n .

Для арифметических операторов присваивания нужная интерпретация получается весьма естественно; мы опишем ее в следующих абзацах. В общем случае при применении синтаксически управляемой трансляции интерпретация должна задаваться создателем компилятора. Эта задача может оказаться легкой или трудной, и в трудных случаях, возможно, придется учитывать структуру всего дерева.

В качестве характерного примера опишем синтаксически управляемую трансляцию арифметических выражений. Заметим, что на рис. 1.3 есть три типа внутренних вершин, зависящих от

того, каким из знаков $=$, $+$, $*$ помечен средний потомок. Эти три типа вершин показаны на рис. 1.4, где треугольниками изображены произвольные поддеревья (возможно, состоящие из единственной вершины). Для любого арифметического оператора присваивания, включающего только арифметические операции $+$ и $*$,

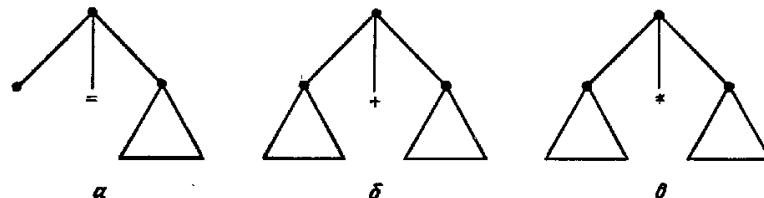


Рис. 1.4. Типы внутренних вершин.

можно построить дерево с одной вершиной (корнем) типа *a* и остальными внутренними вершинами только типов *b* и *v*.

Код, соответствующий вершине *n*, будет иметь следующую интерпретацию:

(1) Если *n*—вершина типа *a*, то $C(n)$ будет кодом, который вычисляет значение выражения, соответствующего правому поддереву, и помещает его в ячейку, зарезервированную для идентификатора, которым помечен левый потомок.

(2) Если *n*—вершина типа *b* или *v*, то цепочка $\text{LOAD } C(n)$ будет кодом, засыпающим в сумматор значение выражения, соответствующего поддереву, над которым доминирует вершина *n*.

Так, для дерева, изображенного на рис. 1.3, код $\text{LOAD } C(n_1)$ засыпает в сумматор значение выражения $\langle \text{id} \rangle_2 + \langle \text{id} \rangle_3$, код $\text{LOAD } C(n_2)$ засыпает в сумматор значение выражения $(\langle \text{id} \rangle_2 + \langle \text{id} \rangle_3) * \langle \text{id} \rangle_4$, а код $C(n_3)$ засыпает в сумматор значение последнего выражения и помещает его в ячейку, предназначенную для $\langle \text{id} \rangle_1$.

Теперь надо показать, как код $C(n)$ строится из кодов потомков вершины *n*. В дальнейшем мы будем предполагать, что операторы языка ассемблера записываются в виде одной цепочки и отделяются друг от друга точкой с запятой или началом новой строки. Кроме того, мы будем предполагать, что каждой вершине *n* дерева приписано число $l(n)$, называемое ее *уровнем*, которое означает максимальную длину пути от этой вершины до листа. Таким образом, $l(n)=0$, если *n*—лист, а если *n* имеет потомков n_1, \dots, n_k , то $l(n)=\max_{1 \leq i \leq k} l(n_i)+1$. Уровни $l(n)$ можно

вычислять снизу вверх одновременно с вычислением кодов $C(n)$. Уровни записываются для того, чтобы контролировать использование временных ячеек памяти. Две нужные нам величины нельзя

засыпать в одну и ту же ячейку памяти. На рис. 1.5 показаны уровни вершин дерева, изображенного на рис. 1.3.

Теперь определим синтаксически управляемый алгоритм генерации кода, предназначенный для вычисления кодов $C(n)$ всех

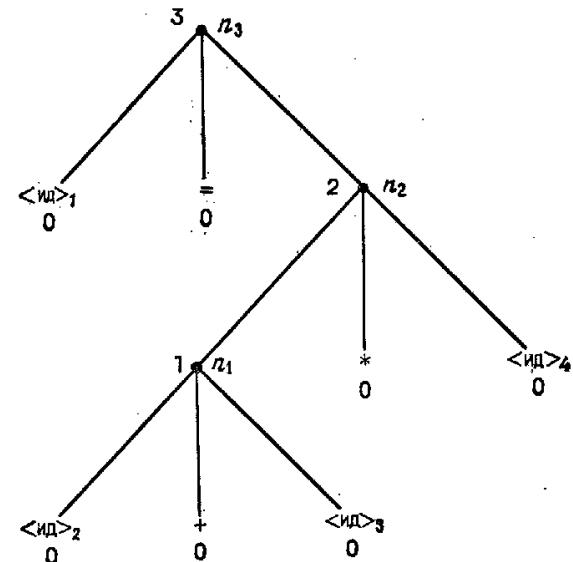


Рис. 1.5. Дерево с уровнями.

вершин дерева, состоящего из листьев, корня типа *a* и внутренних вершин типов *b* и *v*.

Алгоритм 1.1. Синтаксически управляемая трансляция простых операторов присваивания.

Вход. Помеченное упорядоченное дерево, представляющее оператор присваивания, включающий только арифметические операции $+$ и $*$. Предполагается, что уровни всех вершин уже вычислены.

Выход. Код в языке ассемблера, вычисляющий этот оператор присваивания.

Метод. Делать шаги (1) и (2) для всех вершин уровня 0, затем для вершин уровня 1 и т. д., пока не будут обработаны все вершины дерева.

(1) Пусть *n*—лист с меткой $\langle \text{id} \rangle_j$.

(i) Допустим, что элемент *j* таблицы идентификаторов является переменной. Тогда $C(n)$ —имя этой переменной.

(ii) Допустим, что элемент j таблицы идентификаторов является константой k . Тогда $C(n)$ — цепочка $= k$.

(2) Если n — лист с меткой $=$, $*$ или $+$, то $C(n)$ — пустая цепочка. (В этом алгоритме нам не нужно или мы не хотим выдавать выход для листьев, помеченных $=$, $*$ или $+$.)

(3) Если n — вершина типа a и ее прямые потомки — это вершины n_1, n_2 и n_3 , то $C(n)$ — цепочка $\text{LOAD } C(n_3); \text{STORE } C(n_1)$.

(4) Если n — вершина типа b и ее прямые потомки — вершины n_1, n_2 и n_3 , то $C(n)$ — цепочка $C(n_3); \text{STORE } \$l(n); \text{LOAD } C(n_1); \text{ADD } \$l(n)$.

Эта последовательность команд занимает временную ячейку, именем которой служит знак $\$$ вместе со следующим за ним уровнем вершины n . Непосредственно видно, что если перед этой последовательностью поставить LOAD , то значение, которое она в конце концов поместит в сумматор, будет суммой значений выражений поддеревьев, над которыми доминируют вершины n_1 и n_3 .

Сделаем два замечания относительно выбора временных имен. Во-первых, эти имена должны начинаться знаком $\$$, так что их нельзя перепутать с именами идентификаторов в Фортране. Во-вторых, в силу способа выбора $l(n)$ можно утверждать, что $C(n)$ не содержит временного имени $\$i$, если i больше $l(n)$. В частности, $C(n_1)$ не содержит $\$l(n)$. Можно, таким образом, гарантировать, что значение, помещенное в ячейку $\$l(n)$, будет еще находиться там в тот момент, когда его надо прибавить к содержимому сумматора.

(5) Если n — вершина типа b , а все остальное, как в (4), то $C(n)$ — цепочка

$$C(n_3); \text{STORE } \$l(n); \text{LOAD } C(n_1); \text{MPY } \$l(n)$$

Этот код делает то, что надо, и в сумматоре появится нужный результат. \square

Доказательство корректности алгоритма 1.1 оставляем в качестве упражнения. Оно проводится рекурсивно по уровню вершины.

Пример 1.4. Применим алгоритм 1.1 к дереву, изображенному на рис. 1.3. То же дерево, на котором явно выписаны коды, сопоставляемые с каждой его вершиной, показано на рис. 1.6. С вершинами, помеченными $\langle\text{id}\rangle_1, \dots, \langle\text{id}\rangle_4$, связаны соответственно коды COST , PRICE , TAX и $= 0.98$. Теперь мы можем вычислить $C(n_1)$. Так как $l(n_1) = 1$, то формула из правила (4) дает

$$C(n_1) = \text{TAX}; \text{STORE } \$1; \text{LOAD PRICE}; \text{ADD } \$1$$

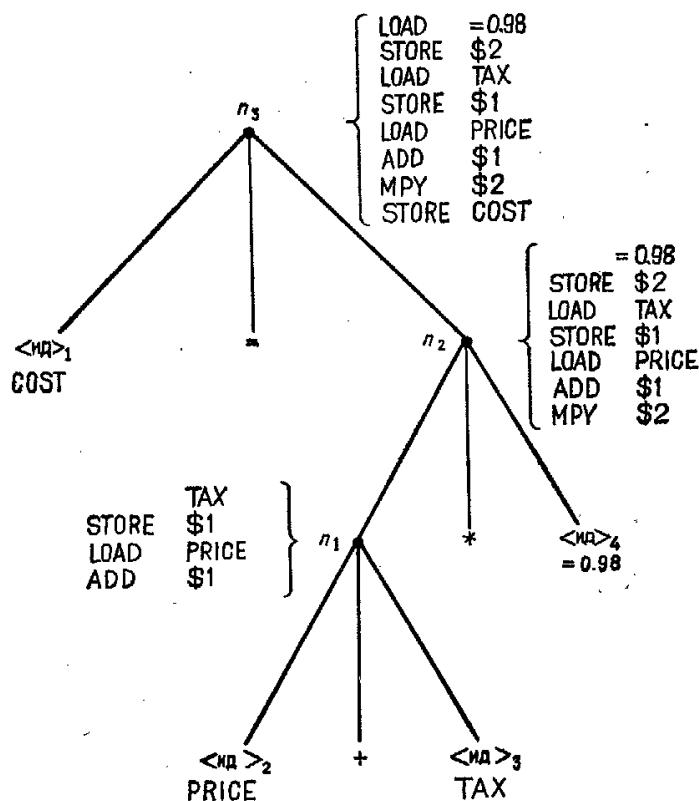


Рис. 1.6. Дерево с генерированными кодами.

Таким образом, код $\text{LOAD } C(n_1)$ вычисляет в сумматоре сумму значений переменных PRICE и TAX , хотя и делает это неуклюже. Неуклюжесть отчасти можно сгладить в процессе оптимизации кода; кроме того, можно разработать правила построения кода, принимающие во внимание особые случаи.

Далее можно вычислить $C(n_2)$ по правилу (5) и получить

$$C(n_2) = = 0.98; \text{STORE } \$2; \text{LOAD } C(n_1); \text{MPY } \$2$$

Здесь $C(n_1)$ — цепочка, построенная для вершины n_1 , а $\$2$ используется как имя временной ячейки, поскольку $l(n_2) = 2$.

Затем вычисляем $C(n_3)$ по правилу (3) и получаем

$$C(n_3) = \text{LOAD } C(n_2); \text{STORE COST}$$

Список команд языка ассемблера (вместо точки с запятой разделителем в нем служит новая строка), который является

переводом нашего первоначального оператора COST = (PRICE + TAX) * 0.98, таков:

(1.2.2)

LOAD	= 0.98
STORE	\$2
LOAD	TAX
STORE	\$1
LOAD	PRICE
ADD	\$1
MPY	\$2
STORE	COST

□

1.2.6. Оптимизация кода

Во многих ситуациях желательно иметь компилятор, который создает эффективно работающие объектные программы. Термин *оптимизация кода* обычно применяется к попыткам сделать объектные программы более „эффективными“, т. е. быстрее работающими или более компактными.

Для оптимизации кода существует широкий спектр возможностей. На одном его конце находится истинно оптимизирующий алгоритм. В этом случае компилятор пытается составить представление о функции, определяемой алгоритмом, программа которого записана на исходном языке. Если он догадается, что это за функция, то может попытаться заменить прежний алгоритм новым, более эффективным алгоритмом, вычисляющим ту же функцию, и уже для этого алгоритма генерировать машинный код.

К сожалению, оптимизация этого типа чрезвычайно трудна. Дело в том, что нет алгоритмического способа нахождения самой короткой или самой быстрой программы, эквивалентной данной. На самом деле можно математически доказать, что существуют алгоритмы, вычисления которых можно ускорять во сколько угодно раз. Иначе говоря, существуют рекурсивные функции, обладающие тем свойством, что для любого алгоритма, вычисляющего такую функцию, найдется другой вычисляющий ее алгоритм, который для достаточно больших входов работает в произвольное число раз быстрее.

Таким образом, термин *оптимизация* совершенно неправильный — на практике мы должны довольствоваться *улучшением кода*. На разных стадиях процесса компиляции применяются различные приемы улучшения кода.

Вообще все, что мы можем делать, это выполнить над данной программой последовательность преобразований в надежде повысить ее эффективность. Эти преобразования должны, разу-

меется, сохранять эффект, создаваемый во внешнем мире исходной программой. Преобразования можно производить в различные моменты процесса компиляции. Например, можно оперировать с самой входной программой, со структурами, порождаемыми на стадии синтаксического анализа, с кодом, порождаемым в качестве выхода фазы генерации кода. Подробнее оптимизация кода будет обсуждаться в гл. 11.

Остальную часть этого раздела мы посвятим следующим преобразованиям, с помощью которых можно сделать код (1.2.2) более коротким:

(1) Если + — коммутативная операция, можно заменить последовательность команд вида LOAD α ; ADD β последовательностью LOAD β ; ADD α . Требуется, однако, чтобы в других местах программы не было перехода к оператору ADD β .

(2) Подобным же образом, если * — коммутативная операция, можно заменить LOAD α ; MPY β на LOAD β ; MPY α .

(3) Последовательность операторов вида STORE α ; LOAD α можно удалить из программы при условии, что либо ячейка α не будет далее использоваться, либо перед использованием α будет заполнена заново. (Чаще можно удалить один лишь оператор LOAD α ; для этого только требуется, чтобы к оператору LOAD α не было перехода в других местах программы.)

(4) Последовательность LOAD α ; STORE β можно удалить, если за ней следует другой оператор LOAD и нет перехода к оператору STORE β , а последующие вхождения β будут заменены на α вплоть до того места, где появляется другой оператор STORE β (но исключая его).

Пример 1.5. Эти четыре преобразования выбраны из-за их применимости к программе (1.2.2). Вообще таких преобразований много и надо пробовать применять их в разных комбинациях. Заметим, что в программе (1.2.2) правило (1) применимо к последовательности LOAD PRICE; ADD \$1, и можно попробовать временно заменить ее на LOAD \$1; ADD PRICE, получив при этом код

(1.2.3)

LOAD	= 0.98
STORE	\$2
LOAD	TAX
STORE	\$1
LOAD	\$1
ADD	PRICE
MPY	\$2
STORE	COST

Теперь ясно, что в (1.2.3) можно удалить последовательность STORE \$1; LOAD \$1 по правилу (3). Таким образом, мы получаем код¹⁾

(1.2.4)

LOAD	= 0.98
STORE	\$2
LOAD	TAX
ADD	PRICE
MPLY	COST
STORE	COST

Теперь к последовательности LOAD = 0.98; STORE \$2 можно применить правило (4). Эти две команды удаляются и \$2 в команде MPLY \$2 заменяется на = 0.98. Окончательный код таков:

(1.2.5)

LOAD	TAX
ADD	PRICE
MPLY	= 0.98
STORE	COST

Код (1.2.5)—самый короткий, какой можно получить с помощью наших четырех и любых других разумных преобразований. □

1.2.7. Анализ и исправление ошибок

До сих пор мы предполагали, что входом компилятора служит правильно построенная программа и что каждую фазу компиляции можно осмысленно довести до конца. На практике, однако, это во многих случаях не так. Программирование в большой степени является искусством, и поистине неограниченны возможности для проникновения в большинство программ различного рода ошибок. Даже если мы чувствуем, что понимаем проблему, для решения которой пишем программу, и даже если мы выбрали подходящий алгоритм, часто нельзя быть уверенными в том, что написанная программа правильно реализует этот алгоритм.

Компилятор имеет возможность обнаруживать ошибки в программе по крайней мере на трех этапах компиляции: во время лексического анализа, синтаксического анализа, при генерации кода. Если встретилась ошибка, то компилятору трудно по неправильной программе решить, что имел в виду ее автор. Эта задача граничит с приложениями искусственного интеллекта. Но в некоторых случаях легко сделать подходящее предположение.

¹⁾ Аналогичного упрощения можно было бы достичь, применяя сразу правило (4). Но мы стараемся дать примеры того, как использовать разные типы преобразований.

Например, если исходный оператор выглядит как $A = B * 2C$, то вполне правдоподобно, что подразумевалось $A = B * 2 * C$.

Вообще в тех случаях, когда процесс разбора доходит до того места во входной цепочке, где он не может дальше правильно продолжаться, некоторые компиляторы пытаются проинформировать "минимальное" изменение во входной цепочке, чтобы продолжить разбор. Перечислим несколько возможных изменений:

(1) Замена одного знака. Например, если лексический анализатор выдает синтаксическому анализатору "идентификатор" INTEJER в неподходящем для появления идентификатора месте программы, то компилятор может догадаться, что подразумевалось ключевое слово INTEGER.

(2) Вставка одной лексемы. Например, компилятор может заменить $2C$ на $2*C$ ($2+C$ тоже годится, но в данном случае мы "знаем", что $2*C$ правдоподобнее).

(3) Устранение одной лексемы. Например, в таком операторе Фортрана, как DO 10 I=1, 20, часто неправильно ставят запятую после 10.

(4) Простая перестановка лексем. Например, вместо INTEGER I может быть неправильно написано I INTEGER.

Во многих языках программирования операторы легко опознаются. Если разбор конкретного неправильно построенного оператора становится безнадежным—даже после того, как произведены изменения вроде описанных выше, часто можно полностью игнорировать этот неправильный оператор и продолжить разбор, как будто его и не было.

Вообще, результатов математического характера об алгоритмах, обнаруживающих ошибки, и алгоритмах, выдающих "хорошую" диагностику, мало. В гл. 4 и 5 мы обсудим несколько алгоритмов синтаксического анализа: LL-алгоритм, LR-алгоритм и алгоритм Эрли, обладающие тем свойством, что как только выясняется, что просмотренную часть входной цепочки нельзя продолжить до правильной цепочки, алгоритм объявляет об этом. Это свойство полезно для обнаружения и анализа ошибок, однако далее мы будем рассматривать и такие алгоритмы, которые этим свойством не обладают.

1.2.8. Резюме

На рис. 1.7 приведена наша принципиальная модель компилятора. Здесь фаза оптимизации следует за фазой генерации кода, но, как мы уже отмечали, разнообразные попытки оптимизации кода могут делаться по ходу всего процесса компиляции.

К процедуре анализа и исправления ошибок можно обращаться на этапах лексического анализа, синтаксического анализа и генерации кода, и если исправление закончилось успешно, то процесс продолжается с того места, где произошло обращение к этой процедуре. Ошибки, при которых в некотором месте входной цепочки не оказывается никакой лексемы, обнаруживаются в ходе лексического анализа. Ошибки, при которых входную программу можно разбить на лексемы, но к этой последовательности лексем не подходит никакое синтаксическое дерево, обнаруживаются в ходе синтаксического анализа. Наконец, ошибки, при которых входная цепочка имеет синтаксическую структуру, но для нее не получается осмысленный код, обнаруживаются в процессе генерации кода. Примером такой ситуации может служить переменная, используемая без описания. Синтаксический анализатор игнорирует информационную компоненту лексемы, так что он не заметит этой ошибки.

Таблицы имен образуются в процессе лексического анализа, а иногда также и в ходе синтаксического анализа, когда, скажем, атрибуты и идентификаторы, к которым они относятся, связываются друг с другом в формируемой древовидной структуре. Эти таблицы используются при генерации кода и, возможно, на этапе ассемблирования.

На рис. 1.7 показана также заключительная фаза, которую мы называем ассемблированием. На этом этапе промежуточный код обрабатывается для получения окончательного представления объектной программы в машинном языке. Некоторые компиляторы могут выдавать код на машинном языке непосредственно как результат процесса генерации кода, так что фазу ассемблирования можно явно не указывать.

Модель компилятора, изображенная на рис. 1.7, есть лишь первое приближение к реальному компилятору. Например, некоторые компиляторы проектируются так, чтобы они занимали небольшой объем памяти. В результате получается много фаз компиляции, которые выполняются последовательно, постепенно преобразуя исходную программу в объектную.

Наша цель состоит не в том, чтобы перечислить все способы построения компиляторов. Мы скорее интересуемся фундаментальными проблемами, возникающими при построении компиляторов и других устройств, предназначенных для обработки языков.

УПРАЖНЕНИЯ

*1.2.1. Опишите синтаксис и семантику операторов присваивания Фортрана.

*1.2.2. Можно ли на Вашем любимом языке программирования написать программу, определяющую произвольное рекур-

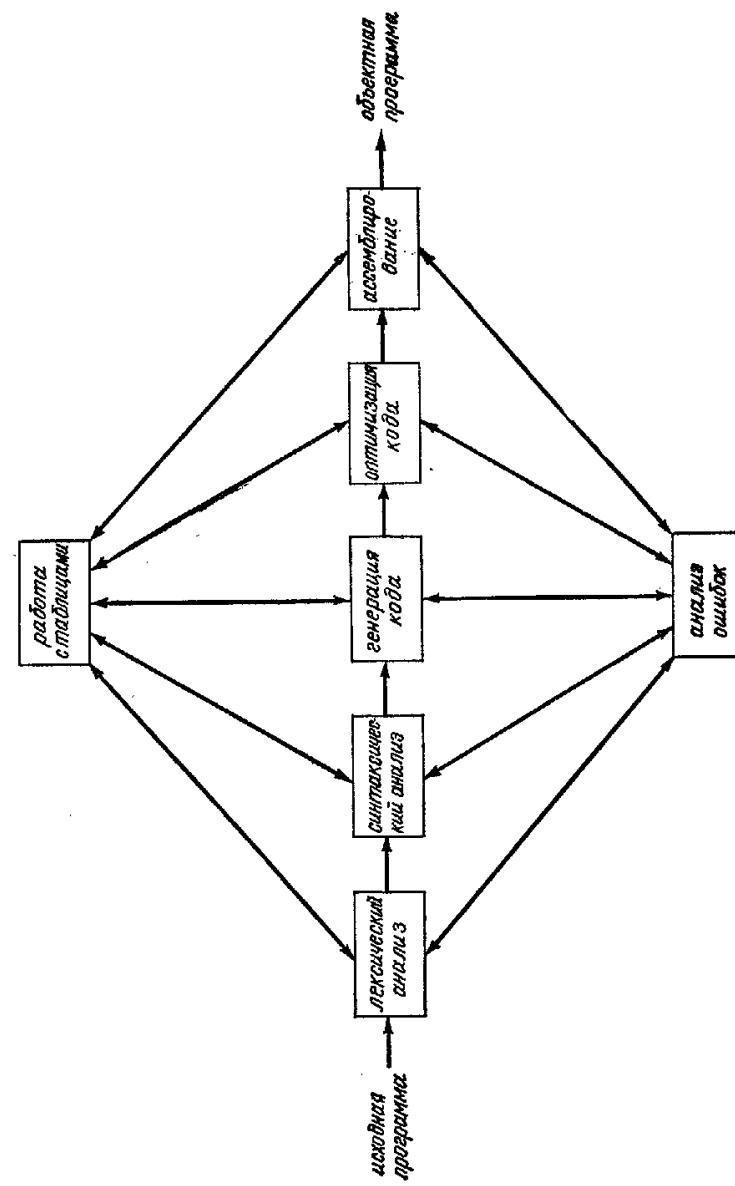


Рис. 1.7. Модель компилятора.

К процедуре анализа и исправления ошибок можно обращаться на этапах лексического анализа, синтаксического анализа и генерации кода, и если исправление закончилось успешно, то процесс продолжается с того места, где произошло обращение к этой процедуре. Ошибки, при которых в некотором месте входной цепочки не оказывается никакой лексемы, обнаруживаются в ходе лексического анализа. Ошибки, при которых входную программу можно разбить на лексемы, но к этой последовательности лексем не подходит никакое синтаксическое дерево, обнаруживаются в ходе синтаксического анализа. Наконец, ошибки, при которых входная цепочка имеет синтаксическую структуру, но для нее не получается осмысленный код, обнаруживаются в процессе генерации кода. Примером такой ситуации может служить переменная, используемая без описания. Синтаксический анализатор игнорирует информационную компоненту лексемы, так что он не заметит этой ошибки.

Таблицы имен образуются в процессе лексического анализа, а иногда также и в ходе синтаксического анализа, когда, скажем, атрибуты и идентификаторы, к которым они относятся, связываются друг с другом в формируемой древовидной структуре. Эти таблицы используются при генерации кода и, возможно, на этапе ассемблирования.

На рис. 1.7 показана также заключительная фаза, которую мы называем ассемблированием. На этом этапе промежуточный код обрабатывается для получения окончательного представления объектной программы в машинном языке. Некоторые компиляторы могут выдавать код на машинном языке непосредственно как результат процесса генерации кода, так что фазу ассемблирования можно явно не указывать.

Модель компилятора, изображенная на рис. 1.7, есть лишь первое приближение к реальному компилятору. Например, некоторые компиляторы проектируются так, чтобы они занимали небольшой объем памяти. В результате получается много фаз компиляции, которые выполняются последовательно, постепенно преобразуя исходную программу в объектную.

Наша цель состоит не в том, чтобы перечислить все способы построения компиляторов. Мы скорее интересуемся фундаментальными проблемами, возникающими при построении компиляторов и других устройств, предназначенных для обработки языков.

УПРАЖНЕНИЯ

*1.2.1. Опишите синтаксис и семантику операторов присваивания Фортрана.

*1.2.2. Можно ли на Вашем любимом языке программирования написать программу, определяющую произвольное рекур-

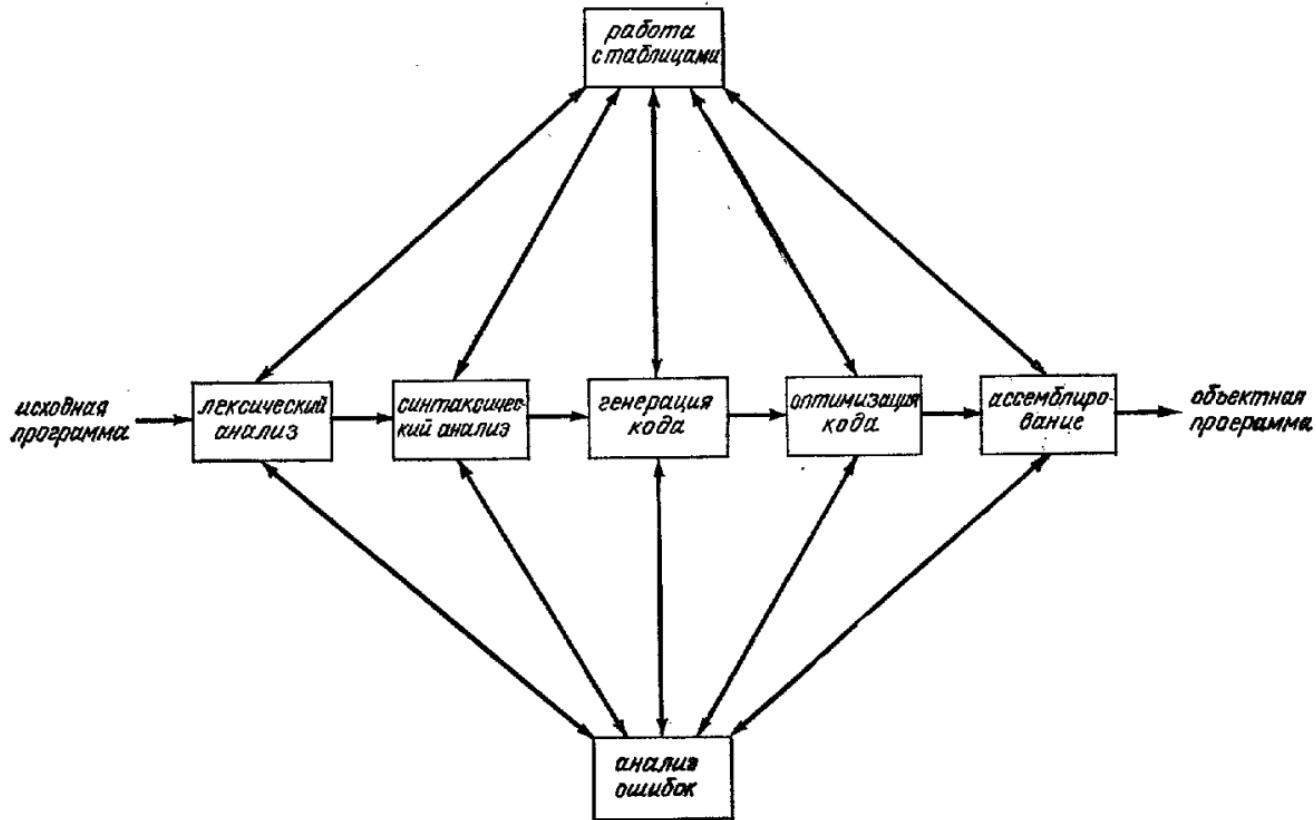


Рис. 1.7. Модель компилятора.

сивно перечислимое множество? Обязательно ли скомпилирует эту программу заданный компилятор?

1.2.3. Приведите пример программы Фортрана, которая синтаксически построена правильно, но не задает (всюду определенный) алгоритм.

****1.2.4.** На какое максимальное число символов требуется заглядывать вперед при прямом лексическом анализе Фортрана? (Имеется в виду число символов, которые просматриваются анализатором, но не составляют часть лексемы, для обнаружения которой осуществляется этот просмотр.)

****1.2.5.** На какое максимальное число символов требуется заглядывать вперед при лексическом анализе Алгола 60? (Можно считать, что лишние пробелы и концевые маркеры перфокарт уже удалены.)

1.2.6. Разберите оператор $X = A * B + C * D$, используя дерево с внутренними вершинами, как на рис. 1.4. Указание: Вспомните, что по известному соглашению при отсутствии скобок умножения выполняются перед сложениями.

1.2.7. Разберите так же, как в упр. 1.2.6, оператор $X = A * (B + C) * D$. Указание: При перемножении нескольких операндов можно считать, что порядок умножений не играет роли¹⁾. Выберите любой, какой Вам нравится.

1.2.8. Примените правила генерации кодов, изложенные в разд. 1.2.5, для синтаксически управляемого перевода деревьев разбора из упр. 1.2.6 и 1.2.7.

***1.2.9.** Сохраняет ли преобразование последовательности LOAD α ; STORE β ; LOAD γ ; STORE δ в последовательность LOAD γ ; STORE δ ; LOAD α ; STORE β отношение между входом и выходом, определяемое программой? Если нет, то какие ограничения надо наложить на идентификаторы α , β , γ , δ ? (Предполагается, что к операторам преобразуемой последовательности не происходит переходов извне.)

1.2.10. Приведите примеры преобразований кодов ассемблера, сохраняющих определяемое программой отношение между входом и выходом.

***1.2.11.** Сделайте синтаксически управляемый перевод, отображающий разбор, приведенный на рис. I.3, прямо в код ассемблера (1.2.5).

¹⁾ Строго говоря, из-за переполнения и/или округления порядок может оказаться важным.

***1.2.12.** Постройте схему синтаксически управляемой трансляции арифметических выражений, содержащих как целые, так и вещественные переменные. (Считайте, что тип каждого идентификатора известен и операция над вещественным значением и целым дает вещественное.)

***1.2.13.** Докажите, что алгоритм 1.1 работает правильно. Сначала нужно определить, когда входной оператор присваивания эквивалентен выходному коду ассемблера.

Проблема для исследования

С компиляцией и трансляцией алгоритмов связано много областей исследования и открытых проблем. Их мы еще будем упоминать в других главах. Но об одной из них мы скажем здесь, так как эту область мы рассматривать не будем.

1.2.14. Разработайте технику доказательства корректности (правильности) компиляторов. В этой и более широкой области доказательства корректности программ и алгоритмов уже проделана некоторая работа (см. ниже замечания по литературе). Однако ясно, что необходимы дальнейшие исследования.

Совсем иной подход к проблеме создания надежных компиляторов состоит в том, чтобы развить теорию, применимую к эмпирическим испытаниям компиляторов. Иначе говоря, предполагается, что мы „знаем“, что наш алгоритм компиляции корректен, и хотим проверить, правильно ли реализует его конкретная программа. При первом из упомянутых выше подходов надо попытаться доказать эквивалентность написанной программы и абстрактного алгоритма компиляции. Второй из предлагаемых подходов состоит в том, чтобы придумать конечное множество таких входных цепочек, что если они компилируются правильно, то с разумной уверенностью (скажем, на 99%) можно сказать, что в программе компилятора ошибок нет. Очевидно, нужно сделать некоторые предположения о частоте и природе ошибок программирования, встречающихся в программе самого компилятора.

Замечания по литературе

Развитие компиляторов и техники компиляции шло параллельно с развитием языков программирования. Первым компилятором, который давал эффективный объектный код, был компилятор с Фортрана [Бэкус и др., 1957]. С тех пор были написаны многочисленные компиляторы и появилось несколько новых методик компиляции. Большие успехи были достигнуты в лексическом и синтаксическом анализах и в понимании техники генерации кодов.

Статьи, относящиеся к построению компиляторов, многочисленны. Мы не будем пытаться перечислить здесь все источники. Исчерпывающие обзоры истории компиляторов и их развития написаны Розеном [1967], Фельдманом и Грисом [1968], Коком и Шварцем [1970]. В ряде книг описывается техника

построения компиляторов: [Реиделл и Рассел, 1964], [Мак-Киман и др., 1970], [Кок и Шварц, 1970], [Грис, 1971]. Хопгуд [1969] дает краткий, но хорошо написанный обзор техники компиляции. Элементарное изложение вопросов компиляции см. в [Ли, 1967].

Написано несколько компиляторов (таких, как DITRAN [Моултон и Мюллер, 1967], PTRAN [Дьюар и др. 1969]), в которых особое внимание уделяется всесторонней диагностике ошибок. Написаны также компиляторы, которые пытаются исправить каждую ошибку и выполнить объективную программу независимо от того, сколько оказалось ошибок. Идея здесь состоит в том, чтобы несмотря на ошибки продолжать компиляцию и выполнение программы и выявить возможно больше ошибок. Такими компиляторами являются CORC [Коивей и Максвелл, 1963; Фримэн, 1964], CUPL [Коивей и Максвелл, 1968] и PL/C [Коивей и др., 1970].

В программе часто встречаются орфографические ошибки. Фримэн [1964] и Морган [1970] описывают технику, которую они нашли эффективной для обнаружения и исправления таких ошибок.

Общий обзор методов обнаружения ошибок при компиляции можно найти в [Элспас и др., 1971].

Попытки создания теоретических основ доказательства корректности компиляторов излагаются в [Мак-Карти, 1963], [Мак-Карти и Пейнтер, 1967], [Пейнтер, 1970] и [Флойд, 1967а].

Построение компилятора — задача, требующая больших затрат труда. Попытки сделать эту работу менее трудоемкой привели к появлению большого числа систем программирования, называемых компиляторами компиляторов. [Брукер и Моррис, 1963], [Читэм и Стэндиш, 1970], [Ингерман, 1966], [Айронс, 1963б], [Фельдмаи, 1966], [Мак-Клюр, 1965], [Мак-Кимай и др., 1970], [Рейнольдс, 1965], [Шорре, 1964] и [Уоршолл и Шапиро, 1964] — только часть литературы по этому предмету. На компилятор компиляторов можно смотреть просто как на язык программирования, в котором исходная программа — это описание компилятора для некоторого языка, а объектная программа — сам компилятор для этого языка.

Исходная программа компилятора компиляторов — это просто формализм, служащий для описания компиляторов. Следовательно, исходная программа должна явно или неявно содержать описание лексического анализатора, синтаксического анализатора, генератора кодов и разных других частей создаваемого компилятора. Компилятор компиляторов отражает попытку создать спектру, с помощью которых все это можно легко записать.

В иескольких компиляторах компиляторов для задания компиляторов используется какой-нибудь вариант схемы синтаксически управляемого перевода, некоторые из них обеспечивают автоматический механизм синтаксического анализа. Система TMG [Мак-Клюр, 1965] — первая из систем этого типа. Другие компиляторы компиляторов, такие, как TGS [Читэм, 1965], вместо этого обеспечивают искусственный язык высокого уровня, на котором удобно описывать алгоритмы, используемые при создании компиляторов. Фельдман и Грис [1968] написали хороший обзор по компиляторам компиляторов.

1.3. ДРУГИЕ ПРИЛОЖЕНИЯ АЛГОРИТМОВ РАЗБОРА И ПЕРЕВОДА

В этом разделе мы коснемся двух отличных от компиляции областей, в которых главную роль могут играть иерархические структуры, подобные тем, что встречаются в алгоритмах разбора и трансляции. Это перевод с естественных языков и распознавание образов.

1.3.1. Естественные языки

Казалось бы, текст, написанный на естественном языке, можно переводить на другой естественный язык или на машинный язык (если этот текст описывает алгоритм) точно так же, как транслируются языки программирования. Однако проблемы возникают уже на стадии синтаксического анализа. Языки, предназначенные для вычислительных машин, имеют точное определение (не без исключений, разумеется) и легко распознаваемую структуру утверждений („предложений“), из которых они состоят. Обычная модель этой структуры — дерево, описанное в разд. 1.2.4.

Естественные языки прежде всего страдают двусмысленностью — как синтаксической, так и семантической. Возьмем в качестве очевидного примера естественного языка английский; предложение *I have drawn butter* имеет по крайней мере два смысла в зависимости от того, является *drawn* в этом предложении прилагательным или глаголом¹⁾. Таким образом, однозначный разбор можно провести не для всякого английского предложения, особенно если предложение рассматривается вне контекста.

Более трудная проблема, касающаяся естественных языков, возникает из-за того, что слова, т. е. терминальные символы языка, связаны с другими словами внутри предложения, вне его и, возможно, с общим окружением. Поэтому для описания всей информации об английском предложении, которую хотелось бы иметь, когда приходится переводить на другой язык (аналог генерации кода в случае языков программирования), не всегда достаточно простая древовидная структура.

Распространенный пример — английское существительное *pen*, имеющее по крайней мере два различных значения — ручка (в выражении *fountain pen* — *pen* — автоматическая ручка) и загон (в выражении *pig pen* — загон для свиней²). Допустим, мы хотим переводить с английского на язык (например, русский), в котором *fountain pen* и *pig pen* обозначаются разными словами. Если нам дано для перевода предложение *This pen leaks*, то как будто бы ясно, что правильным значением *pen* здесь будет „ручка“. Однако если это предложение взято из доклада „Преду-

¹⁾ Приведем пример двусмыслиности в русском языке, извлеченный из одной статьи о синтаксическом анализе языков программирования: „Подъем по полю слов захваченных понятий“. Разные смыслы получаются в зависимости от того, к чему относится „слов“ — к „полю“ или к „понятий“. — *Прим. перев.*

Есть более живые примеры, скажем иззвание одного из органов в Академии иаук: „Комиссия по изучению четвертичного периода АН СССР“. — Прим. ред.

2) Пример омонимии в русском языке: слово „ключ“ имеет разные значения в выражениях „ключ в замке“ и „прозрачный ключ“. — Прим. перев.

преждение насморка у свиней", то нам, вероятно, захочется пересмотреть наше решение.

Отметим особо, что смысл и структуру предложения можно определить, только исследовав все его окружение: окружающие предложения, свойства предмета (например, во фразе „прозрачный ключ“ имеется в виду не ключ от замка, так как он скорее всего не прозрачный) и даже информацию о говорящем и пишущем.

Чтобы подробнее описать информацию, которую можно извлечь из предложений естественного языка, лингвисты используют более сложные структурные системы, чем древовидные структуры, достаточные для языков программирования. Многие из них охватываются понятиями контексто-зависимой грамматики и трансформационной грамматики. Мы не будем рассматривать их подробно, хотя контексто-зависимые грамматики определяются в следующей главе, а некоторую упрощенную форму трансформационной грамматики можно трактовать как обобщение синтаксически управляемого перевода, заданного на деревьях. Это понятие будет определено в гл. 9. В замечаниях по литературе к этому разделу даны ссылки на литературу, из которой можно больше узнать о синтаксическом анализе естественных языков.

1.3.2. Структурное описание образов

Некоторые важные классы образов допускают естественные описания, которые в каком-то смысле поддаются синтаксическому анализу. Шоу [1970], например, проанализировал фотографии, полученные с помощью камеры Вильсона, придав изображенным на них кривым подходящую древовидную структуру. Мы опишем особенно привлекательный способ определения множеств графов с помощью так называемых грамматик, порождающих графы [Пфальц и Розенфельд, 1969]. Полное их описание требует знакомства с разд. 2.1, так что здесь мы лишь приведем простой пример, иллюстрирующий основные идеи.

Пример 1.6. Наш пример относится к графикам, называемым d-схемами¹), которые можно представлять себе как блок-схемы программ некоторого языка программирования, определяемых следующими правилами:

- (1) Простой оператор присваивания является программой.
- (2) Если S_1 и S_2 — программы, то $S_1 ; S_2$ — тоже программа.
- (3) Если S_1 и S_2 — программы и A — предикат, то

if A then S_1 else S_2 end

является программой.

¹⁾ В честь Э. Дейкстры.

- (4) Если S — программа и A — предикат, то
while A do S end

является программой.

Все такие программы можно изобразить блок-схемами, вершины которых (блоки) представляют коды, либо проверяющие предикаты, либо выполняющие простые присваивания. Любую d-схему можно построить, начиная с одной вершины, представляющей программу, и заменяя несколько раз вершины, представляющие программы, одной из трех структур, показанных

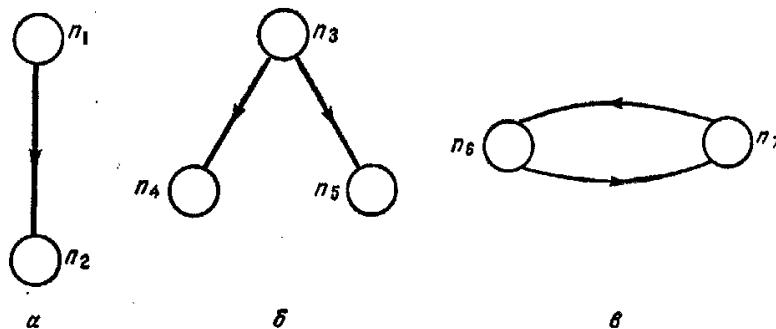


Рис. 1.8. Структуры, представляющие подпрограммы блок-схемы.

на рис. 1.8. Эти правила замены, или подстановки, соответствуют приведенным выше правилам (2) — (4).

Подставленные структуры соединяются с остальной частью графа по следующим правилам. Допустим, что вершина n_0 заменяется одной из структур a , b или c , изображенных на рис. 1.8.

(1) Дуги, входившие в n_0 , теперь входят соответственно в n_1 , n_3 или n_6 .

(2) Дуга, ведущая из n_0 в n , заменяется либо (в случае a) дугой из n_2 в n , либо (в случае b) дугами, ведущими из обеих вершин n_4 и n_5 в вершину n , либо (в случае c) дугой из n_6 в n .

В вершинах n_3 и n_6 проверяются предикаты, поэтому их нельзя заменять структурами. Другие вершины представляют программы и их можно заменять далее.

Построим d-схему, соответствующую программе вида

```
if  $B_1$  then
  while  $B_2$  do
    if  $B_3$  then  $S_1$  else  $S_2$  end end;
     $S_3$ 
```

```

else if  $B_4$  then
   $S_4$ ;
   $S_5$ ;
  while  $B_5$  do  $S_6$  end
else  $S_7$  end end

```

Всю программу можно представить в виде `if B_i then S_8 , else S_9 end`, где S_8 представляет всю часть от первого `while` до S_3 , а S_9 представляет `if $B_4 \dots S_7$ end`. Это первый шаг анализа программы, соответствующий замене одной вершины структурой b на рис. 1.8.

Продолжая анализ, представим S_8 в виде S_{10} ; S_3 (S_{10} — это `while $B_2 \dots S_2$ end end`). Таким образом, этот шаг анализа соответствует замене вершины n_4 на рис. 1.8, б графом a . Результат

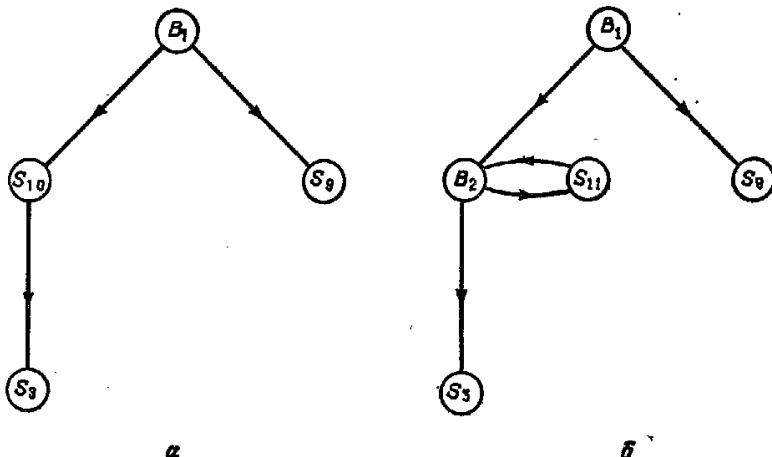


Рис. 1.9. Построение d-схемы.

показан на рис. 1.9, а. Далее, S_{10} имеет вид `while B_2 do S_{11} end`, где S_{11} — это `if $B_3 \dots S_2$ end`. Тогда можно заменить левого прямого потомка корня дерева на рис. 1.9, а структурой, изображенной на рис. 1.8, б. Результат показан на рис. 1.9, б.

Окончательный результат нашего анализа данной программы показан на рис. 1.10. Здесь мы позволили себе нарисовать контуры вокруг заменяемых вершин; заметим, что каждая из последовательных замен осуществляется внутри контура. Таким образом, на d-схему можно наложить естественную древовидную структуру, в которой вершины d-схемы будут листьями, а контуры — внутренними вершинами дерева. Прямым предком вершины дерева будет контур, непосредственно окружающий ее представление в d-схеме. Соответствующее дерево показано на

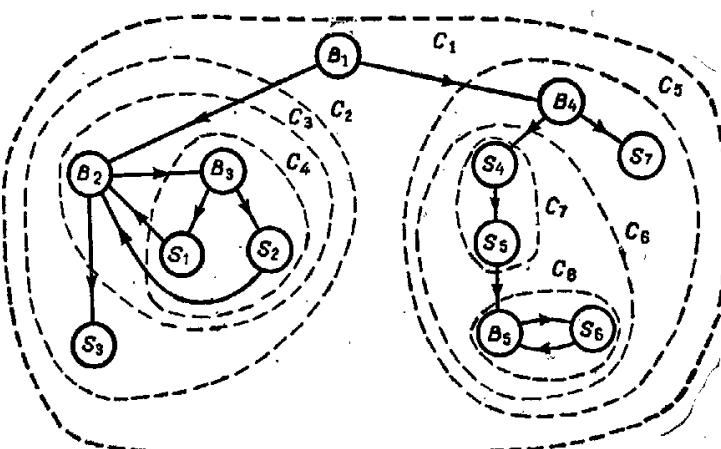


Рис. 1.10. Окончательная d-схема.

рис. 1.11. Вершины дерева помечены соответствующими вершинами или контурами d-схемы. □

Приведенный пример в некотором смысле является подделкой. Структура d-схемы, изображенная на рис. 1.11, — это по существу структура, которую построит для первоначальной программы компилятор на этапе синтаксического анализа. Таким образом, похоже, что мы обсуждаем тот же тип синтаксического анализа, что и в разд. 1.2.3. Однако следует иметь в виду, что этот структурный анализ можно проделать, не ссылаясь на программу, а глядя только на d-схему. Более того, мы привели этот пример графопорождающей грамматики из-за его связи

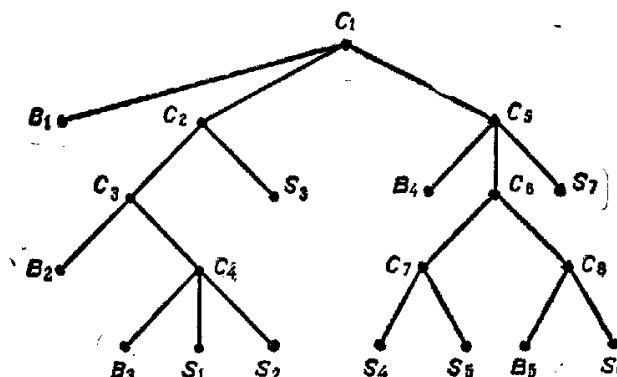


Рис. 1.11. Дерево, описывающее структуру d-схемы.

с языками программирования, но существует много чисто теоретико-графовых понятий, которые можно определить с помощью грамматик, порождающих графы (обобщив подходящим образом пример 1.8), — класс планарных графов, например, или класс бинарных деревьев.

Замечания по литературе

Трудности, возникающие при попытках найти удовлетворительную грамматическую модель для английского языка, хорошо представлены в книге Хомского [1965]. Бобров [1963] дает обзор усилий, направленных на использование английского языка, или, точнее, некоторого его подмножества, в качестве языка программирования. Обзор теоретических аспектов лингвистики содержится в книге Бар-Хиллела [1964]¹).

Понятие грамматики, порождающей графы, взято из работы Пфальца и Розенфельда [1969], теории этих грамматик посвящены статьи Монтанари [1970] и Павлидиса [1972]. Одна из первых работ по синтаксическому анализу образов принадлежит Шоу [1970]. Обзор некоторых результатов, полученных в этой области, можно найти у Миллера и Шоу [1968]²).

¹) Рекомендуем также сборник „Автоматический перевод“ [1971], книгу Винограда [1972] и статьи Цейтина [1971] и Вудса [1970]. — Прим. перев.

²) См. также [Абэ и др., 1973] и [Фу, 1974]. — Прим. перев.

2 ЭЛЕМЕНТЫ ТЕОРИИ ЯЗЫКОВ

В этой главе мы займемся аспектами теории формальных языков, существенными с точки зрения синтаксического анализа и перевода. Вначале рассмотрим синтаксические аспекты языка. Но так как большую часть синтаксиса современных языков программирования можно описать с помощью контекстно-свободных грамматик, мы обратим особое внимание на теорию контекстно-свободных языков.

Сначала изучим один важный подкласс контекстно-свободных языков, а именно регулярные множества. Понятия теории регулярных множеств находят широкое применение и встречаются во многих разделах этой книги.

Другой важный подкласс языков образуют детерминированные контекстно-свободные языки. Это контекстно-свободные языки, грамматики которых допускают простой синтаксический анализ. По счастливому случаю или из-за того, что их преднамеренно создают такими, современные языки программирования можно с хорошим, хотя и не полным приближением считать детерминированными контекстно-свободными языками.

Для этих трех классов языков — контекстно-свободных, регулярных и детерминированных контекстно-свободных — мы дадим их точные определения и опишем основные свойства. Так как теория языков очень обширна и не все в ней имеет отношение к синтаксическому анализу и переводу, доказательства некоторых ее важных теорем приводятся в виде наброска или выполняются в упражнения. Мы стараемся особо выделять только те аспекты теории языков, которые полезны для изложения дальнейшего материала книги.

Как и в гл. 0 и 1, читатель, знакомый с теорией языков, может пропустить или бегло просмотреть эту главу.

2.1. СПОСОБЫ ОПРЕДЕЛЕНИЯ ЯЗЫКОВ

В этом разделе с общей точки зрения будут рассмотрены два основных механизма определения языков — механизм порождения, или генератор, и механизм распознавания, или распозна-

ватель. Мы обсудим только генераторы самого распространенного типа — грамматики Хомского, а распознаватели рассмотрим с несколько большей степенью общности и введем в последующих разделах некоторые из многих типов распознавателей, изучавшихся в литературе.

2.1.1. Мотивировка

Мы определяем язык L как множество цепочек конечной длины в алфавите Σ . Первый важный вопрос: как описать язык L в том случае, когда он бесконечен. Разумеется, если L состоит из конечного числа цепочек, то самый очевидный способ — составить список всех цепочек из L .

Однако для многих языков нельзя (или, быть может, нежелательно) установить верхнюю границу длины самой длинной цепочки языка. Следовательно, приходится рассматривать языки, содержащие сколь угодно много цепочек. Очевидно, что такие языки нельзя определить исчерпывающим перечислением входящих в них цепочек, и, стало быть, надо искать другие способы их описания. Как и прежде, мы хотим, чтобы описание языка было конечным (имело конечный „объем“), хотя описываемый язык может быть и бесконечным.

Известно несколько методов описания языков, удовлетворяющих этому требованию. Один из них состоит в использовании порождающей системы, называемой грамматикой. Цепочки языка строятся точно определенными способами с применением правил грамматики. Одно из преимуществ определения языка с помощью грамматики в том, что операции, производимые в ходе синтаксического анализа и перевода, можно сделать проще, если воспользоваться структурой, которую грамматика приписывает цепочкам („предложениям“) языка. Грамматики, особенно контекстно-свободные, мы изучим очень подробно.

Второй метод описания языка использует частичный алгоритм, который для произвольной входной цепочки останавливается и ответит „да“ после конечного числа шагов, если эта цепочка принадлежит языку. В самом общем случае мы могли бы позволить частичному алгоритму либо остановиться и ответить „нет“, либо продолжать работать бесконечно, если цепочка не принадлежит языку. Однако в практических ситуациях мы должны потребовать, чтобы этот частичный алгоритм был алгоритмом, т. е. чтобы он прекращал работу для всех входных цепочек.

Мы будем представлять частичные алгоритмы, определяющие языки, в виде нескольких схематизированного устройства. Это устройство, называемое *распознавателем*, будет введено в разд. 2.1.4.

2.1.2. Грамматики

Грамматики образуют, по-видимому, наиболее важный класс генераторов языков. Грамматика — это математическая система, определяющая языки. Одновременно она является устройством, которое придает цепочкам („предложениям“) языка полезную структуру. В этом разделе мы рассмотрим класс грамматик, называемых грамматиками Хомского или грамматиками составляющих.

В грамматике, определяющей язык L , используются два конечных непересекающихся множества символов — множество нетерминальных символов, которое часто будет обозначаться буквой N^1 , и множество терминальных символов, обозначаемое Σ . Из терминальных символов образуются слова (цепочки) определяемого языка. Нетерминальные символы служат для порождения слов языка L способом, который будет объяснен позднее.

Сердцевину грамматики составляет конечное множество P правил образования, которые описывают процесс порождения цепочек языка. Правило — это просто пара цепочек, или, точнее, элемент множества $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$. Иначе говоря, первой компонентой правила является любая цепочка, содержащая хотя бы один нетерминал, а второй компонентой — любая цепочка.

Например, правилом может быть пара (AB, CDE) . Если уже установлено, что некоторая цепочка α порождается грамматикой (или „выводится“ в ней) и α содержит AB , т. е. левую часть этого правила, в качестве своей подцепочки, то можно образовать новую цепочку β , заменив одно вхождение AB в α на CDE . Тогда говорят, что β выводима (или выводится) в данной грамматике. Например, если цепочка $FGABH$ выводима, то $FGCDEH$ тоже выводима. Язык, определяемый грамматикой, — это множество цепочек, которые состоят только из терминалов и выводятся, начиная с одной особой цепочки, состоящей из одного выделенного символа, обычно обозначаемого S .

Соглашение. Правило (α, β) будем записывать в виде $\alpha \rightarrow \beta$.

Дадим теперь формальное определение грамматики.

Определение. Грамматикой называется четверка $G = (N, \Sigma, P, S)$, где

(1) N — конечное множество нетерминальных символов, или нетерминалов (иногда называемых вспомогательными символами, синтаксическими переменными или понятиями);

¹) В соответствии с нашим соглашением об обозначениях алфавитов этот символ является прописной греческой буквой „ию“, хотя читателю, вероятно, захочется произносить его как „эн“.

(2) Σ — не пересекающееся с N конечное множество *терминальных символов*, или *терминалов*;

(3) P — конечное подмножество множества

$$(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

(элемент (α, β) множества P называется *правилом* (или *продукцией*) и записывается в виде $\alpha \rightarrow \beta$);

(4) S — выделенный символ из N , называемый *начальным* (или *исходным*) *символом*.

Пример 2.1. Примером грамматики служит четверка $G_1 = (\{A, S\}, \{0, 1\}, P, S)$, где P состоит из правил

$$S \rightarrow 0A1$$

$$0A \rightarrow 00A1$$

$$A \rightarrow e$$

Нетерминальными символами являются A и S , а терминальными — 0 и 1. \square

Грамматика определяет язык рекурсивным образом. Рекурсивность проявляется в задании особого рода цепочек, называемых *выводимыми цепочками* грамматики $G = (N, \Sigma, P, S)$:

(1) S — выводимая цепочка.

(2) Если $\alpha\beta\gamma$ — выводимая цепочка и $\beta \rightarrow \delta$ содержится в P , то $\alpha\delta\gamma$ — тоже выводимая цепочка.

Выводимая цепочка грамматики G , не содержащая нетерминальных символов, называется *терминальной цепочкой, порождаемой грамматикой G*.

Язык, порождаемый грамматикой G (обозначается $L(G)$), — это множество терминальных цепочек, порождаемых грамматикой G .

Теперь введем терминологию, которая окажется далее полезной. Пусть $G = (N, \Sigma, P, S)$ — грамматика. Определим отношение \Rightarrow_G на множестве $(N \cup \Sigma)^*$ ($\psi \Rightarrow_G \phi$ читается ϕ непосредственно выводима из ψ) следующим образом: если $\alpha\beta\gamma$ — цепочка из $(N \cup \Sigma)^*$ и $\beta \rightarrow \delta$ — правило из P , то $\alpha\beta\gamma \Rightarrow_G \alpha\delta\gamma$.

Транзитивное замыкание отношения \Rightarrow_G обозначим через \Rightarrow_G^+ ($\varphi \Rightarrow_G^+ \psi$ читается: ψ выводима из φ нетривиальным образом), а его рефлексивное и транзитивное замыкание — через \Rightarrow_G^* ($\varphi \Rightarrow_G^* \psi$ читается: ψ выводима из φ). В тех случаях, когда из контекста ясно, о какой грамматике идет речь, нижний индекс G будем опускать. Таким образом, $L(G) = \{\omega | \omega \in \Sigma^*, S \Rightarrow^* \omega\}$.

Через \Rightarrow^k будем обозначать k -ю степень отношения \Rightarrow . Иначе говоря, $\alpha \Rightarrow^k \beta$, если существует последовательность $\alpha_0, \alpha_1, \dots, \alpha_k$, состоящая из $k + 1$ цепочек (не обязательно различных), для

которых $\alpha = \alpha_0$, $\alpha_{i-1} \Rightarrow \alpha_i$ при $1 \leq i \leq k$ и $\alpha_k = \beta$. Эта последовательность цепочек называется *выводом длины k цепочки* β из цепочки α в грамматике G . Отметим, что $\alpha \Rightarrow^i \beta$ тогда и только тогда, когда $\alpha \Rightarrow^j \beta$ для некоторого $i \geq 0$, и $\alpha \Rightarrow^+ \beta$ тогда и только тогда, когда $\alpha \Rightarrow^j \beta$ для некоторого $j \geq 1$.

Пример 2.2. Рассмотрим грамматику G_1 из примера 2.1 и вывод $S \Rightarrow 0A1 \Rightarrow 00A1 \Rightarrow 0011$. На первом шаге этого вывода S заменяется на $0A1$ в соответствии с правилом $S \rightarrow 0A1$. На втором шаге $0A$ заменяется на $00A1$, и на третьем шаге A заменяется на e . Можно сказать, что $S \Rightarrow^3 0011$, $S \Rightarrow^+ 0011$, $S \Rightarrow^* 0011$, и что 0011 принадлежит языку $L(G_1)$. Можно показать, что

$$L(G_1) = \{0^n 1^n | n \geq 1\}$$

Доказательство оставляем в качестве упражнения. \square

Соглашение. Для обозначения n правил

$$\begin{aligned} \alpha &\rightarrow \beta_1 \\ \alpha &\rightarrow \beta_2 \\ &\vdots \\ \alpha &\rightarrow \beta_n \end{aligned}$$

удобно пользоваться сокращенной записью

$$\alpha \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$$

Примем еще следующие соглашения относительно различных символов и цепочек, связанных с грамматикой:

(1) a, b, c, d и цифры $0, 1, \dots, 9$ обозначают терминалы;

(2) A, B, C, D, S обозначают нетерминалы; S — начальный символ;

(3) U, V, \dots, Z обозначают либо нетерминалы, либо терминалы;

(4) α, β, \dots обозначают цепочки, которые могут содержать как терминалы, так и нетерминалы;

(5) u, v, \dots, z обозначают цепочки, состоящие только из терминалов.

Эти соглашения распространяются также и на буквы с нижними и верхними индексами. Мы не будем напоминать об этих соглашениях, когда рассматриваемые символы им удовлетворяют. Таким образом, грамматику, все терминалы и нетерминалы которой подчиняются соглашениям (1) и (2), можно определить, просто выписав все ее правила. Например, грамматику G_1 можно задать списком правил

$$\begin{aligned} S &\rightarrow 0A1 \\ 0A &\rightarrow 00A1 \\ A &\rightarrow e \end{aligned}$$

Теперь нет необходимости говорить о множествах терминалов и нетерминалов или о начальном символе.

Приведем еще примеры грамматик.

Пример 2.3. Пусть $G = (\{\langle \text{цифра} \rangle\}, \{0, 1, \dots, 9\}, \{\langle \text{цифра} \rangle \rightarrow 0|1|\dots|9\}, \langle \text{цифра} \rangle)$. Здесь $\langle \text{цифра} \rangle$ рассматривается как единственный нетерминальный символ. $L(G)$ — это, очевидно, множество, состоящее из десяти цифр. Заметим, что $L(G)$ — конечное множество. \square

Пример 2.4. Пусть $G_0 = (\{E, T, F\}, \{a, +, *, (), \}\), P, E)$, где P — состоит из правил

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

Вот пример вывода в этой грамматике

$$\begin{aligned} E &\Rightarrow E + T \\ &\Rightarrow T + T \\ &\Rightarrow F + T \\ &\Rightarrow a + T \\ &\Rightarrow a + T * F \\ &\Rightarrow a + F * F \\ &\Rightarrow a + a * F \\ &\Rightarrow a + a * a \end{aligned}$$

Язык $L(G_0)$ представляет собой множество арифметических выражений, построенных из символов $a, +, *, ()$. \square

Грамматика из примера 2.4 не раз встретится нам в этой книге; мы всегда будем обозначать ее G_0 .

Пример 2.5. Пусть G определяется правилами

$$\begin{aligned} S &\rightarrow aSBC \mid abC \\ CB &\rightarrow BC \\ bB &\rightarrow bb \\ bC &\rightarrow bc \\ cC &\rightarrow cc \end{aligned}$$

В грамматике G возможен вывод

$$\begin{aligned} S &\Rightarrow aSBC \\ &\Rightarrow aabCBC \\ &\Rightarrow aabBCC \\ &\Rightarrow aabbCC \\ &\Rightarrow aabbcC \\ &\Rightarrow aabbcc \end{aligned}$$

Эта грамматика порождает язык $\{a^n b^n c^n \mid n \geq 1\}$. \square

Пример 2.6. Пусть G — грамматика с правилами

$$\begin{array}{ll} S \rightarrow CD & Ab \rightarrow bA \\ C \rightarrow aCA & Ba \rightarrow aB \\ C \rightarrow bCB & Bb \rightarrow bB \\ AD \rightarrow aD & C \rightarrow e \\ BD \rightarrow bD & D \rightarrow e \\ Aa \rightarrow aA & \end{array}$$

Пример вывода в грамматике G :

$$\begin{aligned} S &\Rightarrow CD \\ &\Rightarrow aCAD \\ &\Rightarrow abCBAD \\ &\Rightarrow abBAD \\ &\Rightarrow abBaD \\ &\Rightarrow abaBD \\ &\Rightarrow ababD \\ &\Rightarrow abab \end{aligned}$$

Покажем, что $L(G) = \{ww \mid w \in \{a, b\}^*\}$, т. е. $L(G)$ состоит из цепочек четной длины, составленных из букв a и b , причем первая половина каждой цепочки совпадает со второй половиной.

Так как $L(G)$ — множество, то самый легкий способ показать, что $L(G) = \{ww \mid w \in \{a, b\}^*\}$, состоит в том, чтобы показать, что $\{ww \mid w \in \{a, b\}^*\} \subseteq L(G)$ и $L(G) \subseteq \{ww \mid w \in \{a, b\}^*\}$.

Чтобы показать, что $\{ww \mid w \in \{a, b\}^*\} \subseteq L(G)$, надо показать, что каждую цепочку вида ww можно вывести из S . Простой индукцией можно доказать, что в G возможны такие выводы:

- (1) $S \Rightarrow CD$.
- (2) Для $n \geq 0$

$$\begin{aligned} C &\Rightarrow^n c_1 c_2 \dots c_n CX_{n-1} X_{n-2} \dots X_1 \\ &\Rightarrow c_1 c_2 \dots c_n X_n X_{n-1} \dots X_1 \end{aligned}$$

где для всех $1 \leq i \leq n$ $c_i = a$ тогда и только тогда, когда $X_i = A$, и $c_i = b$ тогда и только тогда, когда $X_i = B$.

$$\begin{aligned} (3) X_n \dots X_2 X_1 D &\Rightarrow X_n \dots X_2 c_1 D \\ &\Rightarrow^{n-1} c_1 X_n \dots X_2 D \\ &\Rightarrow c_1 X_n \dots X_3 c_2 D \\ &\Rightarrow^{n-2} c_1 c_2 X_n \dots X_3 D \\ &\vdots \\ &\Rightarrow c_1 c_2 \dots c_{n-1} X_n D \\ &\Rightarrow c_1 c_2 \dots c_{n-1} c_n D \\ &\Rightarrow c_1 c_2 \dots c_{n-1} c_n \end{aligned}$$

Детали доказательства мы опускаем, так как они проверяются непосредственно.

В выводе (2) из C выводится цепочка, составленная из букв a и b , за которой следует ее зеркальное отражение, составленное соответственно из букв A и B . В выводе (3) нетерминалы A и B перемещаются к правому концу цепочки, где A становится терминалом a , а B становится терминалом b , вступая в контакт с нетерминалом D , который действует как правый концевой маркер. Нетерминалы A и B могут превратиться в терминалы единственным способом — только передвинувшись к правому концу цепочки. При этом цепочка, составленная из букв A и B , будет обращена и совпадет, таким образом, с цепочкой букв a и b , выведенной из C в выводе (2).

Комбинируя выводы (1)–(3), получаем для $n \geq 0$

$$S \Rightarrow^* c_1 c_2 \dots c_n c_1 c_2 \dots c_n$$

где $c_i \in \{a, b\}$ для $1 \leq i \leq n$. Итак, $\{ww \mid w \in \{a, b\}^*\} \subseteq L(G)$.

Теперь покажем, что $L(G) \subseteq \{ww \mid w \in \{a, b\}^*\}$. Для этого надо показать, что из S выводятся только те терминальные цепочки, которые имеют вид ww . Вообще говоря, показать, что грамматика порождает цепочки только данного вида (т. е. что она не может породить других цепочек), гораздо труднее, чем показать, что она может породить все цепочки данного вида.

Здесь удобно задать два гомоморфизма g и h , удовлетворяющие условиям

$$g(a) = a, \quad g(b) = b, \quad g(A) = g(B) = e$$

и

$$h(a) = h(b) = e, \quad h(A) = A, \quad h(B) = B$$

Для нашей грамматики G можно показать индукцией по $m \geq 1$, что если $S \Rightarrow^m \alpha$, то α можно представить в виде $c_1 c_2 \dots c_n U \beta V$, где

- (1) c_i для всех $i = 1, 2, \dots, n$ — либо a , либо b ;
- (2) U — либо C , либо e ;
- (3) β — такая цепочка из языка $\{a, b, A, B\}^n$, что

$$g(\beta) = c_1 c_2 \dots c_n, \quad h(\beta) = X_n X_{n-1} \dots X_1$$

$X_j = A$, если $c_j = a$, и $X_j = B$, если $c_j = b$ ($i < j \leq n$);

- (4) V — либо D , либо e .

Детали индукции опускаем.

Заметим, что все выводимые цепочки грамматики G , состоящие лишь из терминальных символов, имеют вид $c_1 c_2 \dots c_n c_1 c_2 \dots c_n$, где $c_i \in \{a, b\}$ для всех $i = 1, \dots, n$. Таким образом, $L(G) \subseteq \{ww \mid w \in \{a, b\}^*\}$.

Наконец, заключаем, что $L(G) = \{ww \mid w \in \{a, b\}^*\}$. \square

2.1.3. ГРАММАТИКИ С ОГРАНИЧЕНИЯМИ НА ПРАВИЛА

Грамматики можно классифицировать по виду их правил. Пусть $G = (N, \Sigma, P, S)$ — грамматика.

Определение. Грамматика G называется

- (1) **праволинейной**, если каждое правило из P имеет вид $A \rightarrow xB$ или $A \rightarrow x$, где $A, B \in N$, $x \in \Sigma^*$;
- (2) **контекстно-свободной** (или **бесконтекстной**), если каждое правило из P имеет вид $A \rightarrow \alpha$, где $A \in N$, $\alpha \in (N \cup \Sigma)^*$;
- (3) **контекстно-зависимой** (или **неукорачивающей**), если каждое правило из P имеет вид $\alpha \rightarrow \beta$, где $|\alpha| \leq |\beta|$.

Грамматика, не удовлетворяющая ни одному из указанных ограничений, называется грамматикой **общего вида** (или **без ограничений**).

Грамматика примера 2.3 праволинейная. Другой пример праволинейной грамматики — грамматика с правилами

$$S \rightarrow 0S \mid 1S \mid e$$

Эта грамматика порождает язык $\{0, 1\}^*$.

Важным примером контекстно-свободной грамматики служит грамматика примера 2.4. Заметим, что, согласно нашему определению, каждая праволинейная грамматика контекстно-свободная.

В примере 2.5 грамматика, очевидно, контекстно-зависимая. Следует подчеркнуть, что определение контекстно-зависимой грамматики не допускает правил вида $A \rightarrow e$, обычно называемых **e-правилами**. Таким образом, контекстно-свободная грамматика, содержащая **e-правила**, не является контекстно- зависимой.

Запрещение **e-правил** в контекстно-зависимой грамматике вызвано желанием гарантировать рекурсивность порождаемого ею языка. Иначе говоря, мы хотим иметь алгоритм, который для произвольной контекстно-зависимой грамматики G и входной цепочки w определял бы, принадлежит ли эта цепочка языку $L(G)$ (см. упр. 2.1.18).

Если допустить, что среди правил контекстно-зависимой грамматики есть только одно **e-правило** (не снимая с грамматики остальных ограничений), то расширенный класс грамматик уже способен порождать все рекурсивно-перечислимые множества (см. упр. 2.1.20). Грамматика примера 2.6 — это грамматика без ограничений. Заметьте, что она не является ни праволинейной, ни контекстно-свободной, ни контекстно-зависимой.

Соглашение. Если язык L порождается грамматикой типа x , то L называется языком типа x . Это соглашение относится ко всем „типам x “, которые мы уже определили, и к тем, которые еще определим.

Таким образом, язык $L(G)$ примера 2.3—праволинейный, язык $L(G_0)$ примера 2.4—контекстно-свободный, а язык $L(G)$ примера 2.5—контекстно-зависимый. Язык, порожденный грамматикой примера 2.6,—это язык без ограничений, а язык $\{ww \mid w \in \{a, b\}^*\text{ и }w \neq e\}$ —контекстно-зависимый.

Определенные нами четыре типа грамматик и языков часто называют *иерархией Хомского*.

Соглашение. Далее мы будем пользоваться сокращениями КС и КЗ для терминов „контекстно-свободный“ и „контекстно-зависимый“ соответственно.

Каждый праволинейный язык является КС-языком, и существуют КС-языки (например, $\{0^n1^n \mid n \geq 1\}$), которые не являются праволинейными. КС-языки, не содержащие пустой цепочки, образуют собственный подкласс КЗ-языков. А они в свою очередь образуют собственный подкласс рекурсивных множеств, которые собственно включены в класс рекурсивно-перечислимых множеств. Последние порождаются грамматиками общего вида. Доказательства этих фактов оставляем в качестве упражнений.

Часто определяют контекстно-зависимые языки как языки, определенные нами ранее, плюс все языки вида $L \cup \{e\}$, где L —контекстно-зависимый язык в смысле нашего определения. В таком случае КС-языки образуют собственное подмножество КЗ-языков.

Следует подчеркнуть, что, если язык задан какой-то грамматикой, это еще не значит, что его нельзя породить менее мощной грамматикой. Например, контекстно-свободная грамматика

$$\begin{aligned} S &\rightarrow AS | e \\ A &\rightarrow 0 | 1 \end{aligned}$$

порождает язык $\{0, 1\}^*$, который, как мы уже видели, можно породить и праволинейной грамматикой.

Следует также упомянуть о том, что в последнее время были введены грамматические модели, не входящие в иерархию Хомского. Введение новых моделей грамматик отчасти объясняется поисками порождающего механизма, который лучше представлял бы весь синтаксис и/или семантику языков программирования. Некоторые из этих моделей приведены в упражнениях.

2.1.4. Распознаватели

Второй распространенный метод, обеспечивающий задание языка конечными средствами, состоит в использовании распознавателей. В сущности, распознаватель—это очень схематизированный алгоритм, определяющий некоторое множество. Распознаватель можно изобразить так, как показано на рис. 2.1.

Распознаватель состоит из трех частей—входной ленты, управляющего устройства с конечной памятью и вспомогательной, или рабочей, памяти.



Рис. 2.1. Распознаватель.

Входную ленту можно рассматривать как линейную последовательность клеток, или ячеек, причем каждая ячейка содержит точно один входной символ из некоторого конечного входного алфавита. Самую левую и самую правую ячейки могут занимать особые концевые маркеры; маркер может стоять только на правом конце ленты; маркеров может не быть совсем.

Входная головка в каждый данный момент читает, или, как иногда говорят, обозревает, одну входную ячейку. За один шаг работы распознавателя входная головка может сдвинуться на одну ячейку влево, оставаясь неподвижной или сдвинуться на одну ячейку вправо. Распознаватель, который никогда не передвигает свою входную головку влево, называется *односторонним*.

Обычно предполагается, что входная головка *только читает*, т. е. в ходе работы распознавателя символы на входной ленте не меняются. Но можно рассматривать и такие распознаватели, входная головка которых и читает и пишет.

Памятью распознавателя может быть любого типа хранилище информации, или данных. Мы предполагаем, что *алфавит памяти* конечен и хранящаяся в памяти информация построена, или образована, только из символов этого алфавита. Предпо-

лагается также, что в любой момент времени можно конечными средствами описать содержимое и структуру памяти, хотя с течением времени объем памяти может становиться сколь угодно большим. Важный пример вспомогательной памяти — магазинная память (или попросту магазин), которую можно описать абстрактно как цепочку символов памяти; например, $Z_1Z_2\dots Z_n$, где Z_i для всех $i = 1, 2, \dots, n$ принадлежит конечному алфавиту памяти Γ и Z_1 — „верхний“ символ магазина (или, как еще говорят, находится наверху или на вершине магазина).

Поведение вспомогательной памяти для заданного класса распознавателей можно охарактеризовать с помощью двух функций: функции доступа к памяти и функции преобразования памяти. *Функция доступа к памяти* — это отображение множества возможных состояний, или конфигураций, памяти в конечное множество информационных символов, которое может совпадать с алфавитом памяти.

Например, единственная информация, доступная в каждый данный момент в магазине, — верхний символ. Таким образом, функция доступа к магазинной памяти f — это такое отображение Γ^+ в Γ , что $f(Z_1\dots Z_n) = Z_1$.

Функция преобразования памяти — это отображение, описывающее изменения памяти. Она отображает состояние памяти и *управляющую цепочку* в состояние памяти. Если предполагается, что операция над магазинной памятью заменяет верхний символ конечной цепочки символов памяти, то соответствующую функцию преобразования памяти можно определить как такое отображение g : $\Gamma^+ \times \Gamma^* \rightarrow \Gamma^*$, что

$$g(Z_1Z_2\dots Z_n, Y_1\dots Y_k) = Y_1\dots Y_kZ_2\dots Z_n$$

Если верхний символ магазина Z_1 заменяется пустой цепочкой, то Z_2 станет верхним символом и объектом очередного доступа к памяти.

Вообще именно тип памяти определяет название распознавателя. Например, распознаватель, память которого организована как магазин, можно назвать распознавателем с магазинной памятью (обычно его называют автоматом с магазинной памятью).

Ядром распознавателя является *управляющее устройство с конечной памятью*, под которым можно понимать программу, управляющую поведением распознавателя. Управляющее устройство представляет собой конечное множество состояний вместе с отображением, которое описывает, как меняются состояния в соответствии с текущим входным символом (т. е. находящимся под входной головкой) и текущей информацией, извлеченной из памяти. Управляющее устройство определяет также, в каком направлении сдвинуть входную головку и какую информацию поместить в память.

Распознаватель работает, проделывая некоторую последовательность шагов, или тактов. В начале *такта* читается текущий входной символ и с помощью функции доступа исследуется память. Текущий входной символ и информация, извлеченная из памяти, вместе с текущим состоянием управляющего устройства определяют, каким должен быть этот такт. Собственно такт состоит из следующих моментов:

- (1) входная головка сдвигается на одну ячейку влево, одну ячейку вправо или сохраняется в исходном положении,
- (2) в память помещается некоторая информация,
- (3) изменяется состояние управляющего устройства.

Поведение распознавателя удобно описывать в терминах конфигураций распознавателя. *Конфигурация* — это как бы мгновенный снимок распознавателя, на котором изображены

- (1) состояние управляющего устройства,
- (2) содержимое входной ленты вместе с положением входной головки,
- (3) содержимое памяти.

Управляющее устройство распознавателя может быть детерминированным или недетерминированным. Если управляющее устройство *недетерминированное*, то для каждой конфигурации существует конечное множество возможных следующих шагов, любой из которых распознаватель может сделать, исходя из этой конфигурации.

Управляющее устройство называется *детерминированным*, если для каждой конфигурации существует не более одного возможного следующего шага. Недетерминированные распознаватели — удобная математическая абстракция, но, к сожалению, их обычно трудно моделировать на практике. В следующих разделах мы дадим несколько примеров и укажем приложения недетерминированных распознавателей.

Конфигурация называется *начальной*, если управляющее устройство находится в заданном начальном состоянии, входная головка обозревает самый левый символ на входной ленте и память имеет заранее установленное начальное содержимое.

Конфигурация называется *заключительной*, если управляющее устройство находится в одном из состояний заранее выделенного множества заключительных состояний, а входная головка обозревает правый концевой маркер или, если маркер отсутствует, сошла с правого конца входной ленты. Часто требуют, чтобы память в заключительной конфигурации тоже удовлетворяла некоторым условиям.

Говорят, что распознаватель *допускает входную цепочку w* , если, начиная с начальной конфигурации, в которой цепочка w

записана на входной ленте, распознаватель может проделать последовательность шагов, заканчивающуюся заключительной конфигурацией.

Следует указать, что, начиная с данной начальной конфигурации, недетерминированный распознаватель может проделать много различных последовательностей шагов. Если хотя бы одна из этих последовательностей заканчивается заключительной конфигурацией, то начальная входная цепочка будет допущена.

Язык, определяемый распознавателем, — это множество входных цепочек, которые он допускает.

Для каждого класса грамматик из иерархии Хомского существует естественный класс распознавателей, определяющий тот же класс языков. Этими распознавателями являются конечные автоматы, автоматы с магазинной памятью, линейно ограниченные автоматы и машины Тьюринга. Точнее, языки из иерархии Хомского можно охарактеризовать так:

(1) Язык L праволинейный тогда и только тогда, когда он определяется (односторонним детерминированным) конечным автоматом.

(2) Язык L контекстно-свободный тогда и только тогда, когда он определяется (односторонним недетерминированным) автоматом с магазинной памятью.

(3) Язык L контекстно-зависимый тогда и только тогда, когда он определяется (двусторонним недетерминированным) линейно ограниченным автоматом.

(4) Язык L рекурсивно перечислимый тогда и только тогда, когда он определяется машиной Тьюринга.

Точные определения этих распознавателей можно найти в упражнениях и в дальнейших разделах. Конечные автоматы и автоматы с магазинной памятью играют важную роль в теории компиляции, и в этой главе мы их подробно изучим.

УПРАЖНЕНИЯ

2.1.1. Постройте праволинейные грамматики для языков, состоящих из

(а) идентификаторов, которые могут быть произвольной длины, но должны начинаться с буквы (как в Алголе);

(б) идентификаторов, которые должны содержать от одного до шести символов и начинаться с буквы I , J , K , L , M или N (как идентификаторы целых переменных в Фортране);

(в) вещественных констант, как в ПЛ/І или Фортране, например 10.8, 3.14159, 2., 6.625E-27;

(г) всех цепочек из нулей и единиц, имеющих четное число нулей и четное число единиц.

2.1.2. Постройте КС-грамматики, порождающие
а) все цепочки из нулей и единиц с одинаковым числом тех и других;

б) $\{a_1a_2\dots a_n a_n \dots a_2 a_1 \mid a_i \in \{0, 1\}, 1 \leq i \leq n\}$;

в) правильно построенные формулы исчисления высказываний;

г) $\{0^i 1^j \mid i \neq j \text{ и } i, j \geq 0\}$;

д) всевозможные последовательности правильно расставленных скобок.

***2.1.3.** Опишите язык, порождаемый правилами $S \rightarrow bSS \mid a$. Заметьте, что не всегда легко описать язык, порождаемый конкретной грамматикой.

***2.1.4.** Постройте КЗ-грамматики, порождающие

(а) $\{a^n \mid n \geq 1\}$;

(б) $\{\omega\omega \mid \omega \in \{a, b\}^+\}$;

(в) $\{\omega \mid \omega \in \{a, b, c\}^+ \text{ и число букв } a \text{ в цепочке } \omega \text{ равно числу букв } b, \text{ равному числу букв } c\}$;

г) $\{a^m b^n a^m b^n \mid m, n \geq 1\}$.

Указание: Представьте себе множество правил КЗ-грамматики как программу. Вы можете взять специальные нетерминальные символы в роли комбинаций „входной головки“ с терминальными символами.

***2.1.5.** „Истинно“ контекстно-зависимая грамматика G — это грамматика (N, Σ, P, S) , в которой каждое правило имеет вид $\alpha A \beta \rightarrow \alpha \gamma \beta$, где $\alpha, \beta \in (N \cup \Sigma)^*$, $\gamma \in (N \cup \Sigma)^+$ и $A \in N$. Такое правило можно истолковать в том смысле, что A можно заменить на γ только в контексте α , β . Покажите, что каждый КЗ-язык может порождаться „истинно“ контекстно- зависимой грамматикой.

****2.1.6.** Какой класс языков может порождаться грамматиками, использующими только левый контекст, т. е. грамматиками, в которых каждое правило имеет вид $\alpha A \rightarrow \alpha \beta$, где $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Sigma)^+$?

2.1.7. Покажите, что каждый КС-язык может порождаться грамматикой $G = (N, \Sigma, P, S)$, в которой каждое правило имеет вид либо $A \rightarrow \alpha$, $\alpha \in N^*$, либо $A \rightarrow \omega$, $\omega \in \Sigma^*$.

2.1.8. Покажите, что каждый КЗ-язык может порождаться грамматикой $G = (N, \Sigma, P, S)$, в которой каждое правило имеет вид либо $\alpha \rightarrow \beta$, где $\alpha, \beta \in N^+$, либо $A \rightarrow \omega$, где $A \in N$ и $\omega \in \Sigma^+$.

2.1.9. Докажите, что $L(G) = \{a^n b^n c^n \mid n \geq 1\}$, где G — грамматика из примера 2.5.

***2.1.10.** Можете ли Вы описать множество КС-грамматик с помощью КЗ-грамматики?

***2.1.11.** Покажите, что каждое рекурсивно перечислимое множество может порождаться грамматикой, использующей не более двух нетерминальных символов. Может ли произвольное рекурсивно перечислимое множество порождаться грамматикой, использующей только один нетерминальный символ?

2.1.12. Покажите, что если грамматика $G = (N, \Sigma, P, S)$ такова, что $\# N = n$ и Σ не содержит символов A_1, A_2, \dots , то существует эквивалентная ей грамматика $G' = (N', \Sigma, P', A_1)$, у которой $N' = \{A_1, A_2, \dots, A_n\}$.

2.1.13. Докажите, что грамматика G_1 из примера 2.1 порождает язык $\{0^n 1^n \mid n \geq 1\}$. Указание: Обратите внимание, что каждая выводимая цепочка содержит не более одного нетерминала. Таким образом, правила можно применять только в одном месте цепочки.

Определение. В грамматике без ограничений G данную цепочку можно вывести многими способами, которые по существу одинаковы и отличаются лишь порядком, в котором применяются правила. Если грамматика G контекстно-свободная, то эти по существу одинаковые выводы можно представить с помощью одного дерева вывода. Однако если грамматика G контекстно-зависимая или без ограничений, то классы эквивалентных выводов можно определить следующим образом.

Пусть $G = (N, \Sigma, P, S)$ —грамматика без ограничений. Пусть D —множество всех выводов вида $S \Rightarrow^+ w$, т. е. элементами множества D служат такие последовательности вида $(\alpha_0, \alpha_1, \dots, \alpha_n)$, что $\alpha_0 = S$, $\alpha_n \in \Sigma^*$ и $\alpha_{i-1} \Rightarrow \alpha_i$, $1 \leq i \leq n$.

Определим отношение R_0 на множестве D , положив $(\alpha_0, \alpha_1, \dots, \alpha_n) R_0 (\beta_0, \beta_1, \dots, \beta_n)$ тогда и только тогда, когда найдется такой индекс i между 1 и $n-1$, что

$$(1) \alpha_j = \beta_j \text{ для всех } 1 \leq j \leq n, j \neq i;$$

(2) можно писать $\alpha_{i-1} = \gamma_1 \gamma_2 \gamma_3 \gamma_4 \gamma_5$ и $\alpha_{i+1} = \gamma_1 \delta \gamma_3 \epsilon \gamma_5$, причем в P есть правила $\gamma_2 \rightarrow \delta$ и $\gamma_4 \rightarrow \epsilon$, и либо $\alpha_i = \gamma_1 \delta \gamma_3 \gamma_4 \gamma_5$ и $\beta_i = \gamma_1 \gamma_2 \gamma_3 \epsilon \gamma_5$, либо наоборот.

Пусть R —наименьшее отношение эквивалентности, содержащее R_0 . Каждый класс эквивалентности отношения R образован по существу одинаковыми выводами данной цепочки.

****2.1.14.** Каков максимальный объем класса эквивалентности отношения R (как функция от n и $|\alpha_n|$), если грамматика G

(а) праволинейная?

(б) контекстно-свободная?

(в) такова, что каждое ее правило имеет вид $\alpha \rightarrow \beta$ и $|\alpha| < |\beta|$?

***2.1.15.** Пусть G определяется правилами

$$S \rightarrow AOB \mid B1A$$

$$A \rightarrow BB \mid 0$$

$$B \rightarrow AA \mid 1$$

Каков объем класса эквивалентности отношения R , содержащего вывод $S \Rightarrow AOB \Rightarrow BB0B \Rightarrow 1B0B \Rightarrow 1AA0B \Rightarrow 10A0B \Rightarrow 1000B \Rightarrow 10001$?

Определение. Грамматика G называется *однозначной*, если каждая цепочка w из $L(G)$ появляется в качестве последней компоненты вывода в одном и только в одном классе эквивалентности определенного выше отношения R .

Например, грамматика

$$S \rightarrow abC \mid aB$$

$$B \rightarrow bc$$

$$bC \rightarrow bc$$

неоднозначная, так как последовательности (S, abC, abc) и (S, aB, abc) принадлежат двум различным классам эквивалентности.

***2.1.16.** Покажите, что каждый праволинейный язык определяется однозначной праволинейной грамматикой.

***2.1.17.** Пусть $G = (N, \Sigma, P, S)$ —КЗ-грамматика и $N \cup \Sigma$ содержит m элементов. Пусть w —слово из языка $L(G)$. Покажите, что $S \Rightarrow_G^n w$, где $n \leq (m+1)^{|w|}$.

2.1.18. Покажите, что каждый КЗ-язык рекурсивен. Указание: С помощью результата упр. 2.1.17 постройте алгоритм, определяющий для произвольного слова w и КЗ-грамматики G , принадлежит ли w языку $L(G)$.

2.1.19. Покажите, что каждый КС-язык рекурсивен. Указание: Используйте упр. 2.1.18, но обратите внимание на пустое слово.

***2.1.20.** Покажите, что если $G = (N, \Sigma, P, S)$ —грамматика без ограничений, то существует такая КЗ-грамматика $G' = (N', \Sigma \cup \{c\}, P', S')$, что $w \in L(G)$ тогда и только тогда, когда $wc^i \in L(G')$ для некоторого $i \geq 0$. Указание: Дополните каждое не контекстно-зависимое правило грамматики G буквами c . Затем добавьте правила, позволяющие буквам c сдвигаться к правому концу любой выводимой цепочки.

2.1.21. Покажите, что если $L = L(G)$ для произвольной грамматики G , то существуют такой КЗ-язык L_1 и такой гомоморфизм h , что $L = h(L_1)$.

2.1.22. Пусть $\{A_1, A_2, \dots\}$ — счетное множество нетерминальных символов, не содержащее символы 0 и 1. Покажите, что каждый К3-язык $L \subseteq \{0, 1\}^*$ имеет К3-грамматику $G = (N, \{0, 1\}, P, A_i)$, в которой $N = \{A_1, A_2, \dots, A_i\}$ для некоторого i . Мы будем называть такую К3-грамматику *нормализованной*.

***2.1.23.** Покажите, что множество определенных выше нормализованных К3-грамматик счетно.

***2.1.24.** Покажите, что существует рекурсивное множество, содержащееся в $\{0, 1\}^*$, которое не является К3-языком. Указание: Упорядочите нормализованные К3-грамматики, чтобы можно было говорить об i -й грамматике. Аналогично задайте лексикографический порядок на множестве $\{0, 1\}^*$, чтобы можно было говорить об i -й цепочке в $\{0, 1\}^*$. Затем определите $L = \{w_i \mid w_i \in L(G_i)\}$ и покажите, что язык L рекурсивный, но не контекстно-зависимый.

****2.1.25.** Покажите, что язык определяется грамматикой тогда и только тогда, когда он распознается машиной Тьюринга (машина Тьюринга определена в упражнениях разд. 0.4).

2.1.26. Определите недетерминированный распознаватель, памятью которого является первоначально пустая лента машины Тьюринга, причем ее длина не должна становиться больше длины входной цепочки. Покажите, что язык определяется таким распознавателем тогда и только тогда, когда он контекстно-зависимый. Этот распознаватель называется *линейно ограниченным автоматом* (сокращенно ЛО-автоматом).

Определение. Индексная грамматика — это пятерка $G = (N, \Sigma, \Delta, P, S)$, где N, Σ и Δ — конечные множества нетерминалов, терминалов и посредников соответственно, $S \in N$ — начальный символ, P — конечное множество правил вида

$$A \rightarrow X_1 \Psi_1 X_2 \Psi_2 \dots X_n \Psi_n \quad (n \geq 0)$$

и

$$Af \rightarrow X_1 \Psi_1 X_2 \Psi_2 \dots X_n \Psi_n \quad (n \geq 0)$$

где $A \in N$, $X_i \in N \cup \Sigma$, $f \in \Delta$ и $\Psi_i \in \Delta^*$, причем, если $X_i \in \Sigma$, то $\Psi_i = e$. Пусть α и β — цепочки из множества $(N \Delta^* \cup \Sigma)^*$, $A \in N$, $\theta \in \Delta^*$, и пусть $A \rightarrow X_1 \Psi_1 \dots X_n \Psi_n$ принадлежит P . Тогда мы пишем

$$\alpha A \theta \beta \Rightarrow_G \alpha X_1 \Psi_1 \theta'_1 X_2 \Psi_2 \theta'_2 \dots X_n \Psi_n \theta'_n \beta$$

где $\theta'_i = \theta$, если $X_i \in N$, и $\theta'_i = e$, если $X_i \in \Sigma$. Таким образом, цепочка посредников, следующая за нетерминалом в левой части правила, распределяется по нетерминалам правой части,

но ие по терминалам, за которыми никогда не могут идти посредники. Если $Af \rightarrow X_1 \Psi_1 \dots X_n \Psi_n$ принадлежит P , то

$$\alpha A f \theta \beta \Rightarrow_G \alpha X_1 \Psi_1 \theta'_1 \dots X_n \Psi_n \theta'_n \beta$$

где θ'_i определяются так же, как и выше. Этот шаг „поглощает“ посредника, следующего за A , а в остальном он такой же, как и шаг первого типа. Пусть \Rightarrow_G^* обозначает рефлексивное и транзитивное замыкание отношения \Rightarrow_G , и пусть, наконец,

$$L(G) = \{w \mid w \in \Sigma^* \text{ и } S \Rightarrow_G^* w\}$$

Пример 2.7. Пусть $G = (S, T, A, B, C, \{a, b, c\}, \{f, g\}, P, S)$, где P состоит из правил

$$\begin{aligned} S &\rightarrow Tg \\ T &\rightarrow Tf \\ T &\rightarrow ABC \\ Af &\rightarrow aA \\ Bf &\rightarrow bB \\ Cf &\rightarrow cC \\ Ag &\rightarrow a \\ Bg &\rightarrow b \\ Cg &\rightarrow c \end{aligned}$$

Тогда $L(G) = \{a^n b^n c^n \mid n \geq 1\}$. Например, цепочка $aabbcc$ имеет вывод

$$\begin{aligned} S &\Rightarrow Tg \\ &\Rightarrow Tf \\ &\Rightarrow AfgBfgCfg \\ &\Rightarrow aAgBfgCfg \\ &\Rightarrow aaBfgCfg \\ &\Rightarrow aabBfgCfg \\ &\Rightarrow aabbCfg \\ &\Rightarrow aabbcCg \\ &\Rightarrow aabbcc \quad \square \end{aligned}$$

***2.1.27.** Постройте индексные грамматики для языков

- (а) $\{ww \mid w \in \{a, b\}^*\}$,
- (б) $\{a^n b^n \mid n \geq 1\}$.

****2.1.28.** Покажите, что каждый индексный язык контекстно-зависимый.

2.1.29. Покажите, что каждый КС-язык индексный.

2.1.30. Давайте постулируем распознаватель, память которого представляет собой одно целое неотрицательное число (записанное, если хотите, в двоичной форме). Предположим, что управляющими цепочками, описанными в разд. 2.1.4, являются

только X и Y . Какие из следующих функций могут быть функциями доступа к памяти для этого распознавателя?

$$(a) f(i) = \begin{cases} 0, & \text{если } i \text{ четно,} \\ 1, & \text{если } i \text{ нечетно.} \end{cases}$$

$$(b) f(i) = \begin{cases} a, & \text{если } i \text{ четно,} \\ b, & \text{если } i \text{ нечетно.} \end{cases}$$

$$(b) f(i) = \begin{cases} 0, & \text{если } i \text{ четно и под входной головкой} \\ & \text{находится входной символ } a, \\ 1 & \text{в противном случае.} \end{cases}$$

2.1.31. Какие из следующих функций могут быть функциями преобразования памяти для распознавателя из упр. 2.1.30?

$$(a) g(i, X) = 0,$$

$$g(i, Y) = i + 1.$$

$$(b) g(i, X) = 0,$$

$$g(i, Y) = \begin{cases} i + 1, & \text{если предыдущая управляемая} \\ & \text{цепочка была } X. \\ i + 2, & \text{если предыдущая управляемая} \\ & \text{цепочка была } Y. \end{cases}$$

Определение. ТАГ-система состоит из двух конечных алфавитов N и Σ и конечного множества правил вида (α, β) , где $\alpha, \beta \in (N \cup \Sigma)^*$. Если γ — произвольная цепочка из $(N \cup \Sigma)^*$ и (α, β) — правило, то мы пишем $\alpha\gamma\beta$, т. е. префикс α в начале любой цепочки можно стереть при условии, что потом в конце цепочки будет приписано β . Пусть \vdash^* — рефлексивное и транзитивное замыкание отношения \vdash . Для любой цепочки $\gamma \in (N \cup \Sigma)^*$ обозначим через L_γ язык $\{w \mid w \in \Sigma^* \text{ и } \gamma \vdash^* w\}$.

****2.1.32.** Покажите, что L_γ всегда можно определить некоторой грамматикой. Указание: Используйте упр. 2.1.25 или посмотрите [Минский, 1967].

****2.1.33.** Покажите, что для любой грамматики G язык $L(G)$ можно определить ТАГ-системой описанным выше способом. См. указание к упр. 2.1.32.

Открытые проблемы

2.1.34. Является ли дополнение любого контекстно-зависимого языка контекстно- зависимым?

¹⁾ То, что здесь определено, обычно называют канонической системой Поста (в нормальной форме). ТАГ-система должна удовлетворять еще некоторым дополнительным условиям (см. [Минский, 1967]). — Прим. перев.

Если распознаватель из упр. 2.1.26 детерминированный, то он называется *детерминированным линейно ограниченным (ДЛО) автоматом*.

2.1.35. Распознается ли любой КЗ-язык ДЛО-автоматом?

2.1.36. Распознается ли любой индексный язык ДЛО-автоматом?

Из упр. 2.1.28 следует, что утвердительный ответ в упр. 2.1.35 влечет утвердительный ответ в упр. 2.1.36.

Замечания по литературе

Теория формальных языков была в значительной степени стимулирована работами Хомского в конце 1950-х годов [Хомский, 1956, 1957, 1959а, 1959б]. Хорошими примерами ранних работ по порождающим системам служат [Хомский, 1963] и [Бар-Хиллел, 1964].

Иерархия Хомского грамматик и языков хорошо изучена. Многие из основных результатов, касающихся иерархии Хомского, приведены в упражнениях. Большинство из них доказаны в книгах Хопкрофта и Ульмана [1969] и С. Гинзбурга [1966].

С тех пор как Хомский ввел грамматики составляющих, в литературе появились многие другие модели грамматик. В некоторых из них используются специальные виды правил. В качестве примеров таких грамматик назовем индексные грамматики [Ахо, 1968], макрограмматики [Фишер, 1968] и грамматики с рассеянным контекстом [Грейбах и Хопкрофт, 1969]. В других грамматических моделях налагаются ограничения на порядок применения правил, например в программных грамматиках [Розенкранц, 1968]¹⁾.

Распознаватели для языков тоже изучались очень широко. Машинны Тьюринга были введены А. Тьюрингом в 1936 г. Несколько позже в работе Мак-Каллока и Питтса [1943] появилось понятие конечного автомата. Изучение распознавателей было стимулировано работами Мура [1956], Рабина и Скотта [1959]²⁾.

В теории языков значительные усилия были приложены для выяснения алгебраических свойств классов языков и для получения результатов, связанных с проблемой разрешимости для разных классов грамматик и распознавателей. Для каждого из четырех классов грамматик в иерархии Хомского существует класс распознавателей, определяющий в точности те языки, которые порождаются грамматиками этого класса. Эти наблюдения привели к изучению абстрактных семейств языков и распознавателей, причем классы языков определяются в терминах их алгебраических свойств. Определенные алгебраические свойства класса языков оказываются необходимыми и достаточными для существования соответствующего класса распознавателей для этих языков. Работа в этой области была начата Гинзбургом и Грейбах [1969] и Хопкрофтом и Ульманом [1967]. Хороший обзор теории языков к 1970 г. дан Буком [1970].

Хейнс [1970] утверждает, что левоконтекстные грамматики из упр. 2.1.6 порождают в точности контекстно-зависимые языки³⁾. Упр. 2.1.28 взято из [Ахо, 1968].

¹⁾ А также в матричных грамматиках Абрахама [1965]. — Прим. перев.

²⁾ Следовало бы упомянуть здесь и работу Клини [1956]. — Прим. перев.

³⁾ Доказательство этого утверждения см. в § 3.5 книги Гладкого [1973]. — Прим. перев.

2.2. РЕГУЛЯРНЫЕ МНОЖЕСТВА, ИХ РАСПОЗНАВАНИЕ И ПОРОЖДЕНИЕ

Регулярные множества образуют класс языков, занимающий центральное положение по отношению к значительной части теории языков. В этом разделе мы опишем несколько методов задания языков, каждый из которых определяет в точности регулярные множества. Среди них — регулярные выражения, праволинейные грамматики, детерминированные и недетерминированные конечные автоматы.

2.2.1. Регулярные множества и регулярные выражения

Определение. Пусть Σ — конечный алфавит. Регулярное множество в алфавите Σ определяется рекурсивно следующим образом:

(1) \emptyset (пустое множество) — регулярное множество в алфавите Σ ;

(2) $\{e\}$ — регулярное множество в алфавите Σ ;

(3) $\{a\}$ — регулярное множество в алфавите Σ для каждого $a \in \Sigma$;

(4) если P и Q — регулярные множества в алфавите Σ , то таковы же и множества

(a) $P \cup Q$,

(b) PQ ,

(в) P^* ;

(5) ничто другое не является регулярным множеством в алфавите Σ .

Итак, множество в алфавите Σ регулярно тогда и только тогда, когда оно либо \emptyset , либо $\{e\}$, либо $\{a\}$ для некоторого $a \in \Sigma$, либо его можно получить из этих множеств применением конечного числа операций объединения, конкатенации и итерации.

Определим теперь удобный способ обозначения регулярных множеств в конечном алфавите Σ .

Определение. Регулярные выражения в алфавите Σ и регулярные множества, которые они обозначают, определяются рекурсивно следующим образом:

(1) \emptyset — регулярное выражение, обозначающее регулярное множество \emptyset ,

(2) e — регулярное выражение, обозначающее регулярное множество $\{e\}$,

(3) если $a \in \Sigma$, то a — регулярное выражение, обозначающее регулярное множество $\{a\}$,

2.2. РЕГУЛЯРНЫЕ МНОЖЕСТВА, ИХ РАСПОЗНАВАНИЕ И ПОРОЖДЕНИЕ

(4) если p и q — регулярные выражения, обозначающие регулярные множества P и Q соответственно, то

(a) $(p + q)$ — регулярное выражение, обозначающее $P \cup Q$;

(б) (pq) — регулярное выражение, обозначающее PQ ;

(в) $(p)^*$ — регулярное выражение, обозначающее P^* ;

(5) ничто другое не является регулярным выражением.

Мы будем пользоваться записью p^+ для сокращенного обозначения выражения pP^* и, кроме того, будем устранять из регулярных выражений лишние скобки там, где это не может привести к недоразумениям. В этой связи предполагается, что наивысшим приоритетом обладает операция $*$, затем идет конкатенация и далее операция $+$. Так, $0+10^*$ означает $(0+(1(0^*)))$.

Пример 2.8. Вот несколько примеров регулярных выражений и обозначаемых ими множеств:

(1) 01 обозначает $\{01\}$.

(2) 0^* обозначает $\{0\}^*$.

(3) $(0+1)^*$ обозначает $\{0, 1\}^*$.

(4) $(0+1)^*011$ обозначает множество всех цепочек, составленных из нулей и единиц и оканчивающихся цепочкой 011 .

(5) $(a+b)(a+b+0+1)^*$ обозначает множество всех цепочек из $\{0, 1, a, b\}^*$, начинающихся с a или b .

(6) $(00+11)^*((01+10)(00+11)^*(01+10)(00+11)^*)^*$ обозначает множество всех цепочек нулей и единиц, содержащих четное число нулей и четное число единиц. \square

Совершенно ясно, что для каждого регулярного множества можно найти по крайней мере одно регулярное выражение, обозначающее это множество. И обратно, для каждого регулярного выражения можно построить регулярное множество, обозначаемое этим выражением. К сожалению, для каждого регулярного множества существует бесконечно много обозначающих его регулярных выражений.

Будем говорить, что два регулярных выражения равны ($=$), если они обозначают одно и то же множество.

В следующей лемме устанавливаются некоторые основные алгебраические свойства регулярных выражений.

Лемма 2.1. Пусть α, β и γ — регулярные выражения. Тогда

(1) $\alpha + \beta = \beta + \alpha$

(2) $\emptyset^* = e$

(3) $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$

(4) $\alpha(\beta\gamma) = (\alpha\beta)\gamma$

(5) $\alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma$

(6) $(\alpha + \beta)\gamma = \alpha\gamma + \beta\gamma$

(7) $\alpha e = e\alpha = \alpha$

(8) $\emptyset\alpha = \alpha\emptyset = \emptyset$

(9) $\alpha^* = \alpha + \alpha^*$

(10) $(\alpha^*)^* = \alpha^*$

(11) $\alpha + \alpha = \alpha$

(12) $\alpha + \emptyset = \alpha$

Доказательство. (1) Пусть α и β обозначают множества L_1 и L_2 соответственно. Тогда $\alpha + \beta$ обозначает $L_1 \cup L_2$, а $\beta + \alpha$ обозначает $L_2 \cup L_1$. Но $L_1 \cup L_2 = L_2 \cup L_1$ по определению объединения. Следовательно, $\alpha + \beta = \beta + \alpha$.

Доказательство остальных равенств оставляем в качестве упражнения. \square

В дальнейшем мы не будем различать регулярное выражение и обозначаемое им множество, если это не приводит к недоразумениям. Например, в силу этого соглашения символ a будет представлять множество $\{a\}$.

При работе с языками часто бывает удобно пользоваться уравнениями, коэффициентами и неизвестными которых служат множества. Мы рассмотрим здесь системы уравнений, коэффициенты которых — регулярные выражения. Такие уравнения будем называть *уравнениями с регулярными коэффициентами*.

Рассмотрим, например, уравнение с регулярными коэффициентами

$$(2.2.1) \quad X = aX + b$$

где a и b — регулярииые выражения. Легко проверить прямой подстановкой, что $X = a^*b$ — решение уравнения (2.2.1). Иначе говоря, если в обе части уравнения (2.2.1) подставить a^*b вместо X , то слева и справа будет одно и то же множество.

Можно рассматривать также системы уравнений, определяющие языки. Возьмем, например, пару уравнений

$$(2.2.2) \quad \begin{aligned} X &= a_1X + a_2Y + a_3 \\ Y &= b_1X + b_2Y + b_3 \end{aligned}$$

где a_i и b_i для всех $i = 1, 2, 3$ являются регулярииыми выражениями. Покажем, как решить эту систему уравнений и получить решение

$$\begin{aligned} X &= (a_1 + a_2b_2^*b_1)^*(a_3 + a_2b_2^*b_3) \\ Y &= (b_2 + b_1a_1^*a_2)^*(b_3 + b_1a_1^*a_3) \end{aligned}$$

Однако сначала отметим, что не все уравнения с регулярииыми коэффициентами обладают единственным решением. Например, если

$$(2.2.3) \quad X = \alpha X + \beta$$

— уравнение с регулярииими коэффициентами и α обозначает множество, содержащее пустую цепочку, то $X = \alpha^*(\beta + \gamma)$ будет решением уравнения (2.2.3) для любого γ (γ не обязано даже быть регулярииым, см. упр. 2.2.7). Таким образом, уравнение (2.2.3) имеет бесконечно много решений. В такого рода ситуациях мы будем брать наименьшее решение, которое назовем

наименьшей неподвижной точкой. Наименьшая неподвижная точка уравнения (2.2.3) — это множество $X = \alpha^*\beta$.

Определение. Систему уравнений с регулярииими коэффициентами назовем *стандартной* системой с множеством неизвестных $\Delta = \{X_1, X_2, \dots, X_n\}$, если она имеет вид

$$\begin{aligned} X_1 &= \alpha_{10} + \alpha_{11}X_1 + \alpha_{12}X_2 + \dots + \alpha_{1n}X_n \\ &\vdots \\ X_n &= \alpha_{n0} + \alpha_{n1}X_1 + \alpha_{n2}X_2 + \dots + \alpha_{nn}X_n \end{aligned}$$

где все α_{ij} — регулярииые выражения в алфавите, не пересекающимся с Δ .

Коэффициентами уравнений являются выражения α_{ij} . Заметим, что если $\alpha_{ij} = \emptyset$ (такое регулярие выражение возможно), то в уравнении для X_i нет члена, содержащего X_j . Аналогично, если $\alpha_{ij} = e$, то в уравнении для X_i член, содержащий X_j , — это просто X_j . Иными словами, \emptyset играет роль коэффициента 0, а e — роль коэффициента 1 в обычных линейных уравнениях.

Алгоритм 2.1. Решение стандартной системы уравнений с регулярииими коэффициентами.

Вход. Стандартная система Q уравнений с регулярииими коэффициентами в алфавите Σ и с множеством неизвестных $\Delta = \{X_1, \dots, X_n\}$.

Выход. Решение системы Q в виде $X_1 = \alpha_1, \dots, X_n = \alpha_n$, где α_i — регулярие выражение в алфавите Σ .

Метод. Метод напоминает решение системы линейных уравнений методом исключения Гаусса.

Шаг 1. Положить $i = 1$.

Шаг 2. Если $i = n$, перейти к шагу 4. В противном случае с помощью тождеств леммы 2.1 записать уравнение для X_i в виде $X_i = \alpha X_i + \beta$, где α — регулярие выражение в алфавите Σ , а β — регулярие выражение вида $\beta_0 + \beta_1 X_{i+1} + \dots + \beta_n X_n$, причем все β_i — регулярииые выражения в алфавите Σ . (Мы увидим, что это всегда возможно.) Затем в правых частях уравнений для X_{i+1}, \dots, X_n заменить X_i регулярииим выражением $\alpha^*\beta$.

Шаг 3. Увеличить i на 1 и вернуться к шагу 2.

Шаг 4. Записать уравнение для X_n в виде $X_n = \alpha X_n + \beta$, где α и β — регулярииые выражения в алфавите Σ . (После выполнения шага 2 для каждого $i < n$ в правой части уравнения для X_i не будет неизвестных X_1, \dots, X_{i-1} . В частности, на шаге 4 этим свойством будет обладать и уравнение для X_n .) Перейти к шагу 5 (при этом $i = n$).

Шаг 5. Уравнение для X_i имеет вид $X_i = \alpha X_i + \beta$, где α и β — регулярииые выражения в алфавите Σ . Записать на выходе

$X_i = \alpha^* \beta$ и в уравнения для X_{i-1}, \dots, X_1 подставить $\alpha^* \beta$ вместо X_i .

Шаг 6. Если $i = 1$, остановиться. В противном случае уменьшить i на 1 и вернуться к шагу 5. \square

Пример 2.9. Пусть $\Delta = \{X_1, X_2, X_3\}$. Рассмотрим систему уравнений

$$(2.2.4) \quad X_1 = 0X_2 + 1X_1 + e$$

$$(2.2.5) \quad X_2 = 0X_3 + 1X_2$$

$$(2.2.6) \quad X_3 = 0X_1 + 1X_3$$

Из уравнения (2.2.4) получаем $X_1 = 1X_1 + (0X_2 + e)$. Затем в остальные уравнения подставляем $1^*(0X_2 + e)$ вместо X_1 . Уравнение (2.2.6) принимает вид

$$X_3 = 01^*(0X_2 + e) + 1X_3$$

или с учетом леммы 2.1

$$(2.2.7) \quad X_3 = 01^*0X_2 + 1X_3 + 01^*$$

Если теперь из уравнения (2.2.5), которое мы еще не трогали, выразить X_2 через X_3 и в (2.2.7) подставить 1^*0X_3 вместо X_2 , то получим

$$(2.2.8) \quad X_3 = (01^*01^*0 + 1)X_3 + 01^*$$

Теперь в алгоритме 2.1 мы дошли до шага 5. Из уравнения (2.2.8) находим решение для X_3 :

$$(2.2.9) \quad X_3 = (01^*01^*0 + 1)^*01^*$$

Подстановка (2.2.9) в (2.2.5) дает

$$(2.2.10) \quad X_2 = 0(01^*01^*0 + 1)^*01^* + 1X_2$$

Так как X_3 не входит в уравнение (2.2.4), то оно не меняется. Затем решаем (2.2.10) и получаем

$$(2.2.11) \quad X_2 = 1^*0(01^*01^*0 + 1)^*01^*$$

Подставляем (2.2.11) в (2.2.4):

$$(2.2.12) \quad X_1 = 01^*0(01^*01^*0 + 1)^*01^* + 1X_1 + e$$

Решением уравнения (2.2.12) будет

$$(2.2.13) \quad X_1 = 1^*(01^*0(01^*01^*0 + 1)^*01^* + e)$$

Выход алгоритма 2.1 — равенства (2.2.9), (2.2.11) и (2.2.13). \square

Мы должны показать, что выход алгоритма 2.1 действительно является решением данной системы уравнений в том смысле, что если подставить решение вместо неизвестных, то в каждом урав-

нении обе его части будут обозначать одно и то же множество. Как уже указывалось, решение стандартной системы уравнений не всегда единствено. Однако мы увидим, что в таком случае алгоритм 2.1 дает наименьшую неподвижную точку.

Определение. Пусть Q — стандартная система уравнений с множеством неизвестных Δ и с коэффициентами в алфавите Σ . Отображение f множества Δ в множество языков в алфавите Σ называют *решением* системы Q , если после подстановки в каждое уравнение $f(X)$ вместо X для каждого $X \in \Delta$ уравнения становятся равенствами множеств. Отображение $f: \Delta \rightarrow \mathcal{P}(\Sigma^*)$ называют *наименьшей неподвижной точкой* системы Q , если f — решение и для любого другого решения g

$$f(X) \equiv g(X) \text{ для всех } X \in \Delta$$

Следующие две леммы содержат полезную информацию о наименьших неподвижных точках.

Лемма 2.2. Каждая стандартная система уравнений Q с неизвестными Δ обладает единственной наименьшей неподвижной точкой.

Доказательство. Пусть $f(X) = \{\omega \mid \omega \in g(X)\}$ для всех решений g системы Q для всех $X \in \Delta$. Можно прямо показать, что f — решение и $f(X) \equiv g(X)$ для всех решений g . Таким образом, f — единственная наименьшая неподвижная точка системы Q . \square

Теперь дадим некоторую характеристику наименьшей неподвижной точки.

Лемма 2.3. Пусть Q — стандартная система уравнений с неизвестными $\Delta = \{X_1, \dots, X_n\}$ и уравнение для X_i имеет вид

$$X_i = \alpha_{i0} + \alpha_{i1}X_1 + \alpha_{i2}X_2 + \dots + \alpha_{in}X_n$$

Тогда наименьшей неподвижной точкой системы Q будет такое отображение f , что $f(X_i) = \{\omega_1 \dots \omega_m \mid \omega_m \in \alpha_{i,m}^0\}$ и $\omega_k \in \alpha_{i,j_k k+1}^0$ для некоторой последовательности чисел j_1, \dots, j_m , где $m \geq 1$, $1 \leq k \leq m$ и $j_1 = i$.

Доказательство. Легко проверить непосредственно, что для всех i

$$f(X_i) = \alpha_{i0} \cup \alpha_{i1}f(X_1) \cup \dots \cup \alpha_{in}f(X_n)$$

Таким образом, f — решение.

Чтобы показать, что f — наименьшая неподвижная точка, предположим, что g — решение и для некоторого i существует цепочка $\omega \in f(X_i) — g(X_i)$. Так как $\omega \in f(X_i)$, то можно записать

$w = w_1 \dots w_m$, где для некоторой последовательности чисел j_1, \dots, j_m выполнены условия $w_m \in \alpha_{j_m 0}$ и $w_k \in \alpha_{j_k j_{k+1}}$ для $1 \leq k < m$ и $j_1 = i$. Так как g —решение, то

$$g(X_j) = \alpha_{j_0} \cup \alpha_{j_1} g(X_1) \cup \dots \cup \alpha_{j_n} g(X_n) \text{ для всех } j$$

В частности, $\alpha_{j_0} \subseteq g(X_j)$ и $\alpha_{j_k} g(X_k) \subseteq g(X_j)$ для всех j и k . Тогда $w_m \in g(X_{j_m})$, $w_{m-1} w_m \in g(X_{j_{m-1}})$ и т. д. В конечном счете получаем, что цепочка $w = w_1 w_2 \dots w_m$ принадлежит множеству $g(X_i) = g(X_j)$. Пришли к противоречию, так как предположили, что w не принадлежит $g(X_i)$. Таким образом, заключаем, что $f(X_i) \subseteq g(X_i)$ для всех i .

Отсюда непосредственно следует, что f —наименьшая неподвижная точка системы Q . \square

Лемма 2.4. Пусть Q_1 и Q_2 —системы уравнений до и после одного применения шага 2 алгоритма 2.1. Тогда Q_1 и Q_2 имеют одну и ту же наименьшую неподвижную точку.

Доказательство. Допустим, что на шаге 2 рассматривается уравнение

$$A_i = \alpha_{i0} + \alpha_{ii} A_i + \alpha_{i, i+1} A_{i+1} + \dots + \alpha_{in} A_n$$

(Заметим, что для $1 \leq h < i$ коэффициент при A_h равен \emptyset .) В системах Q_1 и Q_2 уравнения для A_h при $h \leq i$ совпадают.

Допустим, что

$$(2.2.14) \quad A_j = \alpha_{j0} + \sum_{k=1}^n \alpha_{jk} A_k$$

—уравнение для A_j в системе Q_1 при $j > i$. В системе Q_2 уравнением для A_j будет

$$(2.2.15) \quad A_j = \beta_0 + \sum_{k=i+1}^n \beta_k A_k$$

где

$$\begin{aligned} \beta_0 &= \alpha_{j0} + \alpha_{ji} \alpha_{ii}^* \alpha_{i0} \\ \beta_k &= \alpha_{jk} + \alpha_{ji} \alpha_{ii}^* \alpha_{ik} \text{ для } i < k \leq n \end{aligned}$$

С помощью леммы 2.3 можно получить представления для наименьших неподвижных точек систем Q_1 и Q_2 , которые мы обозначим соответственно через f_1 и f_2 . Из уравнения (2.2.15) видно, что каждая цепочка, принадлежащая $f_2(A_j)$, принадлежит $f_1(A_j)$. Это вытекает из того, что любую цепочку $w \in \alpha_{ji} \alpha_{ii}^* \alpha_{ik}$ можно представить в виде $w_1 w_2 \dots w_m$, где $w_1 \in \alpha_{ji}$, $w_m \in \alpha_{ik}$ и $w_2, \dots, w_{m-1} \in \alpha_{ii}$. Таким образом, цепочка w является конкатнацией последовательности цепочек, каждая из которых принад-

лежит множеству, обозначаемому некоторым коэффициентом системы Q_1 , и индексы этих коэффициентов удовлетворяют условию леммы 2.3. Аналогично для цепочек, принадлежащих $\alpha_{ji} \alpha_{ii}^* \alpha_{ik}$. Так можно показать, что $f_2(A_j) \subseteq f_1(A_j)$.

Чтобы доказать обратное включение, предположим, что $w \in f_1(A_j)$. Тогда по лемме 2.3 $w = w_1 \dots w_m$, где для некоторой последовательности ненулевых индексов l_1, \dots, l_m выполнены условия $w_m \in \alpha_{l_m 0}$ и $w_p \in \alpha_{l_p l_{p+1}}$ для $1 \leq p < m$ и $l_1 = j$. Можно однозначно сгруппировать цепочки w_p так, чтобы было $w = y_1 \dots y_r$, где $y_p = w_t \dots w_s$ и

(1) если $l_p \leq i$, то $s = t + 1$,

(2) если $l_p > i$, то s выбирается так, чтобы все l_{t+1}, \dots, l_s были равны i и $l_{s+1} \neq i$.

Отсюда следует, что в любом случае y_p —коэффициент при $A_{l_{s+1}}$ в уравнении для A_{l_p} системы Q_2 , и, значит, $w \in f_2(A_j)$. В итоге получаем, что $f_1(A_j) = f_2(A_j)$ для всех j . \square

Лемма 2.5. Пусть Q_1 и Q_2 —системы уравнений до и после одного применения шага 5 алгоритма 2.1. Тогда Q_1 и Q_2 имеют одну и ту же наименьшую неподвижную точку.

Доказательство. Доказательство аналогично доказательству леммы 2.4 и остается в качестве упражнения. \square

Теорема 2.1. Алгоритм 2.1 находит наименьшую неподвижную точку стандартной системы уравнений.

Доказательство. После того как шаг 5 применен для всех i , все уравнения принимают вид $X_i = \alpha_i$, где α_i —регулярное выражение в алфавите Σ . Ясно, что отображение f , для которого $f(X_i) = \alpha_i$, и будет наименьшей неподвижной точкой этой системы. \square

2.2.2. Регулярные множества и праволинейные грамматики

Покажем, что язык определяется праволинейной грамматикой тогда и только тогда, когда он является регулярным множеством. Чтобы доказать, что для каждого регуляярного множества существует праволинейная грамматика, начнем с некоторых наблюдений. Пусть Σ —конечный алфавит.

Лемма 2.6. Множества (i) \emptyset , (ii) $\{e\}$ и (iii) $\{a\}$ для всех $a \in \Sigma$ являются праволинейными языками.

Доказательство. (i) $G = (\{S\}, \Sigma, \emptyset, S)$ —праволинейная грамматика, для которой $L(G) = \emptyset$.

(ii) $G = (\{S\}, \Sigma, \{S \rightarrow e\}, S)$ — праволинейная грамматика, для которой $L(G) = \{e\}$.

(iii) $G_a = (\{S\}, \Sigma, \{S \rightarrow a\}, S)$ — праволинейная грамматика, для которой $L(G_a) = \{a\}$. \square

Лемма 2.7. Если L_1 и L_2 — праволинейные языки, то языки

- (i) $L_1 \cup L_2$, (ii) $L_1 L_2$ и (iii) L_1^* тоже праволинейные.

Доказательство. Так как языки L_1 и L_2 праволинейные, можно считать, что существуют праволинейные грамматики $G_1 = (N_1, \Sigma, P_1, S_1)$ и $G_2 = (N_2, \Sigma, P_2, S_2)$, для которых $L(G_1) = L_1$ и $L(G_2) = L_2$. Будем также предполагать, что алфавиты N_1 и N_2 не пересекаются. Так как нетерминалы грамматики можно как угодно переименовывать, это предположение не приведет к потере общности.

(i) Пусть G_3 — праволинейная грамматика

$$(N_1 \cup N_2 \cup \{S_3\}, \Sigma, P_1 \cup P_2 \cup \{S_3 \Rightarrow S_1 | S_2\}, S_3)$$

где S_3 — новый нетерминальный символ, не принадлежащий ни N_1 , ни N_2 . Ясно, что $L(G_3) = L(G_1) \cup L(G_2)$, так как для каждого вывода $S_3 \Rightarrow_{G_3}^+ w$ существует либо вывод $S_1 \Rightarrow_{G_1}^+ w$, либо вывод $S_2 \Rightarrow_{G_2}^+ w$, и обратно. Так как G_3 — праволинейная грамматика, то $L(G_3)$ — праволинейный язык.

(ii) Пусть G_4 — праволинейная грамматика $(N_1 \cup N_2, \Sigma, P_4, S_1)$, в которой P_4 определяется так:

- (1) Если $A \rightarrow xB$ принадлежит P_1 , то $A \rightarrow xB$ принадлежит P_4 .
- (2) Если $A \rightarrow x$ принадлежит P_1 , то $A \rightarrow xS_2$ принадлежит P_4 .
- (3) Все правила из P_2 принадлежат P_4 .

Заметим, что если $S_1 \Rightarrow_{G_4}^+ w$, то $S_1 \Rightarrow_{G_4}^+ wS_2$, а если $S_2 \Rightarrow_{G_4}^+ x$, то $S_2 \Rightarrow_{G_4}^+ x$. Таким образом, $L(G_1)L(G_2) \subseteq L(G_4)$. Предположим теперь, что $S_1 \Rightarrow_{G_4}^+ w$. Так как в P_4 нет правил вида $A \rightarrow x$, которые попали туда из P_1 , то этот вывод можно записать в виде $S_1 \Rightarrow_{G_4}^+ xS_2 \Rightarrow_{G_4}^+ xy$, где $w = xy$ и все правила, используемые в выводе $S_1 \Rightarrow_{G_4}^+ xS_2$, попали в P_4 с помощью шагов (1) и (2). Следовательно, должны быть выводы $S_1 \Rightarrow_{G_4}^+ x$ и $S_2 \Rightarrow_{G_4}^+ y$. Отсюда $L(G_4) \subseteq L(G_1)L(G_2)$. Итак, $L(G_4) = L(G_1)L(G_2)$.

(iii) Пусть грамматика $G_5 = (N_1 \cup \{S_5\}, \Sigma, P_5, S_5)$ такова, что S_5 не принадлежит N_1 , а P_5 строится следующим образом:

- (1) Если $A \rightarrow xB$ принадлежит P_1 , то $A \rightarrow xB$ принадлежит P_5 .
- (2) Если $A \rightarrow x$ принадлежит P_1 , то $A \rightarrow xS_5$ и $A \rightarrow x$ принадлежат P_5 .
- (3) $S_5 \rightarrow S_1 | e$ принадлежат P_5 .

Доказательство того, что $S_5 \Rightarrow_{G_5}^+ x_1 S_5 \Rightarrow_{G_5}^+ x_1 x_2 S_5 \Rightarrow_{G_5}^+ \dots \dots \Rightarrow_{G_5}^+ x_1 x_2 \dots x_{n-1} S_5 \Rightarrow_{G_5}^+ x_1 x_2 \dots x_{n-1} x_n$ тогда и только тогда, когда $S_1 \Rightarrow_{G_5}^+ x_1$, $S_1 \Rightarrow_{G_5}^+ x_2$, ..., $S_1 \Rightarrow_{G_5}^+ x_n$, оставляем в качестве упражнения.

Отсюда заключаем, что $L(G_5) = (L(G_1))^*$. \square

Теперь можно доказать, что класс праволинейных языков совпадает с классом регулярных множеств.

Теорема 2.2. Язык является регулярным множеством тогда и только тогда, когда он праволинейный.

Доказательство. Необходимость. Эта часть доказательства получается из лемм 2.6 и 2.7, если воспользоваться индукцией по числу шагов построения регулярного множества, где один шаг — это применение одного из правил, определяющих регулярные множества.

Достаточность. Пусть $G = (N, \Sigma, P, S)$ — праволинейная грамматика и $N = \{A_1, \dots, A_n\}$. Можно построить стандартную систему уравнений с регулярными коэффициентами, неизвестными которой служат нетерминалы из N . Уравнение для A_i имеет вид $A_i = \alpha_{i0} + \alpha_{i1}A_1 + \dots + \alpha_{in}A_n$, где

(1) $\alpha_{i0} = w_1 + \dots + w_k$, если $A_i \rightarrow w_1 | \dots | w_k$ — все правила с левой частью A_i и правой частью, состоящей только из терминалов (если $k = 0$, полагаем $\alpha_{i0} = \emptyset$);

(2) для $j > 0$ $\alpha_{ij} = x_1 + \dots + x_m$, если $A_i \rightarrow x_1 A_j | \dots | x_m A_j$ — все правила с левой частью A_i и правой частью, оканчивающейся на A_j (как и раньше, если $m = 0$, то $\alpha_{ij} = \emptyset$).

С помощью леммы 2.3 можно показать, что $L(G) = f(S)$, где f — наименьшая неподвижная точка построенной системы уравнений (это предлагаем в качестве упражнения). Но алгоритм 2.1 строит $f(S)$ как язык, обозначаемый некоторым регулярным выражением. Таким образом, $L(G)$ — регулярное множество. \square

Пример 2.10. Пусть грамматика G определяется правилами

$$\begin{aligned} S &\rightarrow 0A | 1S | e \\ A &\rightarrow 0B | 1A \\ B &\rightarrow 0S | 1B \end{aligned}$$

Тогда соответствующая система уравнений получается из системы примера 2.9, если X_1, X_2, X_3 заменить соответственно на S, A, B . $L(G)$ — это множество цепочек, в которых число нулей делится на 3. Нетрудно показать, что это множество обозначается регулярииым выражением (2.2.13). \square

2.2.3. Конечные автоматы

Мы рассмотрели три способа определения класса регулярных множеств:

(1) класс регулярных множеств — наименьший класс языков, содержащий множества \emptyset , $\{e\}$ и $\{a\}$ для всех символов a и замкнутый относительно операций объединения, конкатенации и итерации;

(2) регулярные множества — множества, определяемые регулярными выражениями;

(3) регулярииые множества — языки, порождаемые праволинейными грамматиками.

Теперь опишем четвертый способ их определения с помощью конечных автоматов. Конечный автомат является одним из простейших распознавателей. „Бесконечной“ памяти у него нет. Обычно конечный автомат состоит только из входной ленты и управляющего устройства с конечной памятью. Здесь мы позволим управляющему устройству быть недетерминированным, но входная головка будет односторонней. Фактически мы потребуем, чтобы входная головка сдвигалась вправо на каждом такте¹⁾. Двусторонний конечный автомат рассматривается в упражнениях.

Мы определим конечный автомат, задав конечное множество его управляющих состояний, допустимые входные символы, начальное состояние и множество заключительных состояний, т. е. состояний, указывающих, что входная цепочка допускается. Задается также функция переходов состояний, которая по данному „текущему“ состоянию и „текущему“ входному символу указывает все возможные следующие состояния. Подчеркнем, что это устройство — недетерминированное в теоретико-автоматном смысле, т. е. оно переходит во все свои следующие состояния, если угодно, дублируя себя так, что в каждом из возможных следующих состояний находится один экземпляр этого устройства. Недетерминированный автомат допускает входную цепочку, если какой-нибудь из его параллельно работающих экземпляров достигает допускающего состояния.

Недетерминизм конечного автомата не следует смешивать со „случайностью“, при которой автомат может случайно выбрать одно из следующих состояний с фиксированными вероятностями, но этот автомат всегда имеется только в одном экземпляре.

1) Напомним, что по определению односторонний распознаватель не сдвигает свою входную головку влево; она, однако, может оставаться неподвижной в течение такта. Если позволить конечному автоматау оставлять свою входную головку неподвижной, это не даст ему возможности распознать язык, не распознаваемый обычным конечным автоматом.

2.2. РЕГУЛЯРНЫЕ МНОЖЕСТВА, ИХ РАСПОЗНАВАНИЕ И ПОРОЖДЕНИЕ

Такой автомат называется „вероятностным“ и здесь изучаться не будет.

Дадим формальное определение недетерминированного конечного автомата.

Определение. Недетерминированный конечный автомат — это пятерка $M = (Q, \Sigma, \delta, q_0, F)$, где

- (1) Q — конечное множество состояний;
- (2) Σ — конечное множество допустимых входных символов;
- (3) δ — отображение множества $Q \times \Sigma$ в множество $\mathcal{P}(Q)$, определяющее поведение управляющего устройства; функцию δ иногда называют *функцией переходов*;
- (4) $q_0 \in Q$ — начальное состояние управляющего устройства;
- (5) $F \subseteq Q$ — множество заключительных состояний.

Работа конечного автомата представляет собой некоторую последовательность шагов, или тактов. Такт определяется текущим состоянием управляющего устройства и входным символом, обозреваемым в данный момент входной головкой. Сам шаг состоит из изменения состояния и сдвига входной головки на одну ячейку вправо.

Для того чтобы определить будущее поведение конечного автомата, нужно знать лишь

- (1) текущее состояние управляющего устройства и
- (2) цепочку символов на входной ленте, состоящую из символа под головкой и всех символов, расположенных вправо от него.

Эти два элемента информации дают мгновенное описание конечного автомата, которое мы будем называть *конфигурацией*.

Определение. Если $M = (Q, \Sigma, \delta, q_0, F)$ — конечный автомат, то пара $(q, w) \in Q \times \Sigma^*$ называется *конфигурацией* автомата M . Конфигурация (q_0, w) называется *начальной*, а пара (q, e) , где $q \in F$, называется *заключительной* (или *допускающей*).

Такт автомата M представляется бинарным отношением \vdash_M (или \vdash , если M подразумевается), определенным на конфигурациях. Если $\delta(q, a)$ содержит q' , то $(q, aw) \vdash_M (q', w)$ для всех $w \in \Sigma^*$.

Это говорит о том, что если M находится в состоянии q и входная головка обозревает входной символ a , то автомат M может делать такт, за который он переходит в состояние q' и сдвигает головку на одну ячейку вправо. Так как автомат M , вообще говоря, недетерминированный, могут быть и другие

состояния, отличные от q' , в которые он тоже может перейти за один такт.

Запись $C \vdash_M^k C'$ означает, что $C = C'$, а $C_0 \vdash_M^k C_k$ (для $k \geq 1$) — что существуют такие конфигурации C_1, \dots, C_{k-1} , что $C_i \vdash_M C_{i+1}$ для всех $0 \leq i < k$. $C \vdash_M^* C'$ означает, что $C \vdash_M^k C'$ для некоторого $k \geq 1$, а $C \vdash_M^* C'$ — что $C \vdash_M^k C'$ для некоторого $k \geq 0$. Таким образом, отношения \vdash_M и \vdash_M^* являются соответственно транзитивным и рефлексивно-транзитивным замыканием отношения \vdash_M . Будем опускать нижний индекс M там, где это не приведет к недоразумениям.

Говорят, что автомат M допускает цепочку w , если $(q_0, w) \vdash^*(q, e)$ для некоторого $q \in F$. Языком, определяемым (распознаваемым, допускаемым) автоматом M (обозначается $L(M)$), называется множество входных цепочек, допускаемых автоматом M , т. е.

$$L(M) = \{w \mid w \in \Sigma^* \text{ и } (q_0, w) \vdash^*(q, e) \text{ для некоторого } q \in F\}$$

Приведем два примера конечных автоматов. Первый — простой „детерминированный“ автомат; второй пример показывает использование недетерминизма.

Пример 2.11. Пусть $M = (\{p, q, r\}, \{0, 1\}, \delta, p, \{r\})$ — конечный автомат, где δ задается табл. 2.1.

Таблица 2.1

δ	Вход	
	0	1
Состояние p	$\{q\}$	$\{p\}$
q	$\{r\}$	$\{p\}$
r	$\{r\}$	$\{r\}$

M допускает все цепочки из нулей и единиц, содержащие два стоящих рядом нуля. Начальное состояние p можно интерпретировать так: „два стоящих рядом нуля еще не появились и предыдущий символ не был нулем“. Состояние q означает, что „два стоящих рядом нуля еще не появились, но предыдущий символ был нулем“. Состояние r означает, что „два стоящих рядом нуля уже появились“. Заметим, что, попав в состояние r , автомат M остается в этом состоянии.

Для входа 01001 единственной возможной последовательностью конфигураций, начинающейся конфигурацией $(p, 01001)$,

будет

$$\begin{aligned} (p, 01001) &\vdash (q, 1001) \\ &\vdash (p, 001) \\ &\vdash (q, 01) \\ &\vdash (r, 1) \\ &\vdash (r, e) \end{aligned}$$

Таким образом, $01001 \in L(M)$. \square

Пример 2.12. Построим недетерминированный конечный автомат, допускающий цепочки в алфавите $\{1, 2, 3\}$, у которых последний символ цепочки уже появлялся в ней раньше. Иными словами, цепочка 121 допускается, а 31312 — нет. Введем состояние q_0 , смысл которого в том, что автомат в этом состоянии не пытается ничего распознать, он (или какой-то его экземпляр) находится в так называемом нейтральном состоянии. Введем также состояния q_1, q_2 и q_3 , смысл которых в том, что они „делают предположение“ о том, что последний символ цепочки совпадает с индексом состояния. Кроме того, пусть будет одно заключительное состояние q_f . Находясь в состоянии q_0 , автомат может остаться в нем или перейти в состояние q_a , если a — очередной символ. Находясь в состоянии q_a , экземпляр автомата может перейти в состояние q_j , если видит символ a . Из состояния q_f автомат никуда не переходит, так как вопрос о том, допускается ли входная цепочка, решается один только раз,

Таблица 2.2

δ	Вход		
	1	2	3
Состояние q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_3\}$
q_1	$\{q_1, q_f\}$	$\{q_1\}$	$\{q_1\}$
q_2	$\{q_2\}$	$\{q_2, q_f\}$	$\{q_2\}$
q_3	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_f\}$
q_f	\emptyset	\emptyset	\emptyset

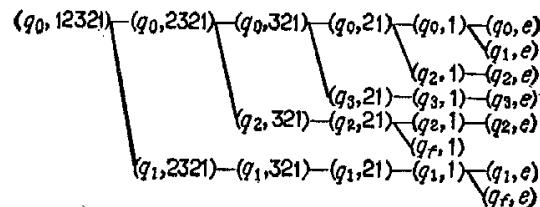
когда автомат считает входной символ „последним“. Формально автомат M определяется как пятерка

$$M = (\{q_0, q_1, q_2, q_3, q_f\}, \{1, 2, 3\}, \delta, q_0, \{q_f\})$$

где δ задается табл. 2.2.

Процесс порождения конфигураций для входа 12321 показан на рис. 2.2.

Так как $(q_0, 12321) \vdash^* (q_f, e)$, то $12321 \in L(M)$. Заметим, что некоторые конфигурации на рис. 2.2 повторяются. Поэтому для представления конфигураций, в которые попадает автомат M , по-видимому, больше подходит ориентированный ациклический граф. \square

Рис. 2.2. Конфигурации автомата M .

Часто бывает удобно графическое представление конечного автомата.

Определение. Пусть $M = (Q, \Sigma, \delta, q_0, F)$ — недетерминированный конечный автомат. Диаграммой (или графиком переходов) автомата M называют неупорядоченный помеченный граф, вершины которого помечены именами состояний и в котором есть дуга (p, q) , если существует такой символ $a \in \Sigma$, что $q \in \delta(p, a)$. Кроме того, дуга (p, q) помечается списком, состоящим из таких a , что $q \in \delta(p, a)$.

На рис. 2.3 показаны диаграммы автоматов из примеров 2.11 и 2.12. Начальное состояние указывается на диаграммах направленной в него стрелкой, помеченной словом „начало“, а заключительные состояния обводятся кружком.

Определим детерминированный конечный автомат как частный случай недетерминированного.

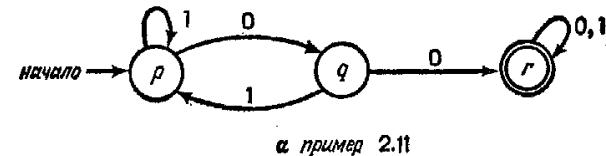
Определение. Пусть $M = (Q, \Sigma, \delta, q_0, F)$ — недетерминированный конечный автомат. Назовем автомат M детерминированным, если множество $\delta(q, a)$ содержит не более одного состояния для любых $q \in Q$ и $a \in \Sigma$. Если $\delta(q, a)$ всегда содержит точно одно состояние, то автомат M назовем полностью определенным.

Таким образом, автомат из примера 2.11 — полностью определенный детерминированный конечный автомат. В дальнейшем конечным автоматом будем называть полностью определенный детерминированный конечный автомат.

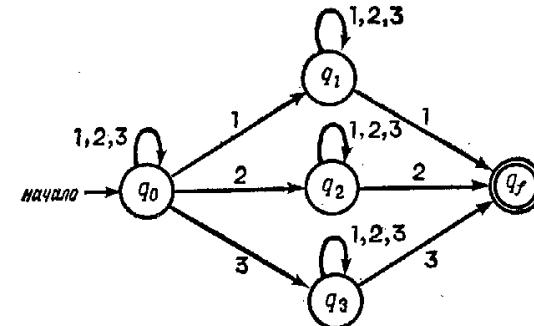
Один из наиболее важных результатов теории конечных автоматов состоит в том, что класс языков, определяемых недетерминированными конечными автоматами, совпадает с классом

языков, определяемых полностью определенными детерминированными конечными автоматами. Сейчас мы это докажем.

Соглашение. Так как нам придется рассматривать в основном детерминированные конечные автоматы, мы будем писать $\delta(q, a) = p$ вместо $\delta(q, a) = \{p\}$, когда автомат с функцией пере-



а пример 2.11



б пример 2.12

Рис. 2.3. Диаграммы автоматов.

ходов δ детерминированный. Если $\delta(q, a) = \emptyset$, то часто будем говорить, что $\delta(q, a)$ не определено.

Теорема 2.3. Если $L = L(M)$ для некоторого недетерминированного конечного автомата M , то $L = L(M')$ для некоторого конечного автомата M' .

Доказательство. Пусть $M = (Q, \Sigma, \delta, q_0, F)$. Построим $M' = (Q', \Sigma, \delta', q'_0, F')$ следующим образом:

- (1) $Q' = \mathcal{P}(Q)$, т. е. состояниями автомата M' являются множества состояний автомата M ;
- (2) $q'_0 = \{q_0\}$;
- (3) F' состоит из всех таких подмножеств S множества Q , что $S \cap F \neq \emptyset$;
- (4) $\delta(S, a) = S'$ для всех $S \subseteq Q$, где $S' = \{p \mid \delta(q, a) \text{ содержит } p \text{ для некоторого } q \in S\}$.

Оставляем в качестве упражнения доказательство индукцией по i утверждения

$$(2.2.16) \quad (S, w) \vdash_M^i (S', e) \text{ тогда и только тогда, когда } S' = \{p \mid (q, w) \vdash_M^i (p, e) \text{ для некоторого } q \in S\}.$$

Отсюда, в частности, следует, что $(\{q_0\}, w) \vdash_M^i (S', e)$ для некоторого $S' \in F'$ тогда и только тогда, когда $(q_0, w) \vdash_M^i (p, e)$ для некоторого $p \in F$. Итак, $L(M') = L(M)$. \square

Пример 2.13. Построим конечный автомат $M' = (Q, \{1, 2, 3\}, \delta', \{q_0\}, F)$, допускающий язык $L(M)$, определяемый автоматом M из примера 2.12. Так как M имеет 5 состояний, то, казалось бы, M' должен иметь 32 состояния. Однако не все они достижимы из начального состояния. Состояние r называется *достижимым*, если существует такая цепочка w , что $(q_0, w) \vdash^*(r, e)$,

ход		
1	2	3
состояния $A = \{q_0\}$	B	C
$B = \{q_0, q_1\}$	E	F
$C = \{q_0, q_2\}$	F	H
$D = \{q_0, q_3\}$	G	I
$E = \{q_0, q_1, q_f\}$	E	F
$F = \{q_0, q_1, q_2\}$	K	K
$G = \{q_0, q_1, q_3\}$	M	L
$H = \{q_0, q_2, q_f\}$	F	H
$I = \{q_0, q_2, q_f\}$	L	N
$J = \{q_0, q_3, q_f\}$	G	I
$K = \{q_0, q_1, q_2, q_f\}$	K	L
$L = \{q_0, q_1, q_2, q_3\}$	P	P
$M = \{q_0, q_1, q_3, q_f\}$	M	L
$N = \{q_0, q_2, q_3, q_f\}$	L	N
$P = \{q_0, q_1, q_2, q_3, q_f\}$	P	P

Рис. 2.4. Функция переходов автомата M' .

где q_0 — начальное состояние. Мы будем строить только достижимые состояния.

Начнем с замечания, что состояние $\{q_0\}$ достижимо. $\delta'(\{q_0\}, a) = \{q_0, q_a\}$ для $a = 1, 2$ и 3 . Рассмотрим состояние $\{q_0, q_1\}$. Имеем $\delta'(\{q_0, q_1\}, 1) = \{q_0, q_1, q_f\}$. Продолжая в том же духе, получаем, что множество состояний автомата M (оно является состоянием автомата M') достижимо тогда и только тогда, когда

- (1) оно содержит q_0 и
- (2) если оно содержит q_f , то содержит также и q_1, q_2 или q_3 .

Множество всех достижимых состояний автомата M' вместе с функцией δ' приведено на рис. 2.4.

Начальным состоянием автомата M' является A , а множество заключительных состояний — $\{E, H, J, K, M, N, P\}$. \square

2.2.4. Конечные автоматы и регулярные множества

Покажем, что язык является регулярным множеством тогда и только тогда, когда он определяется конечным автоматом. Для этого мы докажем сначала, что конечно-автоматный язык определяется праволинейной грамматикой, а затем — что множество конечно-автоматных языков содержит языки $\emptyset, \{e\}, \{a\}$ для всех символов a и замкнуто относительно объединения, конкатенации и итерации. Таким образом, каждое регулярное множество оказывается конечно-автоматным языком.

Лемма 2.8. Если $L = L(M)$ для некоторого конечного автомата M , то $L = L(G)$ для некоторой праволинейной грамматики G .

Доказательство. Пусть $M = (Q, \Sigma, \delta, q_0, F)$ — конечный автомат (детерминированный, разумеется). Возьмем грамматику $G' = (Q, \Sigma, P, q_0)$, где P определяется так:

- (1) Если $\delta(q, a) = r$, то P содержит правило $q \rightarrow ar$.
- (2) Если $p \in F$, то $p \rightarrow e \in P$.

Можно показать, что каждый шаг вывода в грамматике G имитирует торт автомата M . Докажем индукцией по i , что

$$(2.2.17) \quad q \Rightarrow^{i+1} w \text{ для } q \in Q \text{ тогда и только тогда, когда } (q, w) \vdash^i (r, e) \text{ для некоторого } r \in F.$$

Базис. Для $i = 0$ очевидно, что $q \Rightarrow e$ тогда и только тогда, когда $(q, e) \vdash^0 (q, e)$ для $q \in F$.

Шаг индукции. Предположим, что (2.2.17) истинно для i , и возьмем $w = ax$, где $|x| = i$. Тогда $q \Rightarrow^{i+1} w$ равносильно тому, что $q \Rightarrow as \Rightarrow^i ax$ для некоторого $s \in Q$. Но $q \Rightarrow as$ равносильно $\delta(q, a) = s$. По предположению индукции $s \Rightarrow^i x$ тогда и только тогда, когда $(s, x) \vdash^{i-1} (r, e)$ для некоторого $r \in F$. Следовательно, $q \Rightarrow^{i+1} w$ равносильно $(q, w) \vdash^i (r, e)$ для некоторого $r \in F$. Итак, утверждение (2.2.17) истинно для всех $i \geq 0$.

Отсюда заключаем, что $q_0 \Rightarrow^+ w$ тогда и только тогда, когда $(q_0, w) \vdash^*(r, e)$ для некоторого $r \in F$. Таким образом, $L(G) = L(M)$. \square

Лемма 2.9. Пусть Σ — конечный алфавит. Множества (i) \emptyset , (ii) $\{e\}$ и (iii) $\{a\}$ для всех $a \in \Sigma$ являются конечно-автоматными языками.

Доказательство. (i) Любой конечный автомат с пустым множеством заключительных состояний допускает \emptyset .

(ii) Пусть $M = (\{q_0\}, \Sigma, \delta, q_0, \{q_0\})$, где $\delta(q_0, a)$ не определено ни для каких $a \in \Sigma$. Тогда $L(M) = \{e\}$.

(iii) Пусть $M = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$, где $\delta(q_0, a) = q_1$, а в остальных случаях функция δ не определена. Тогда $L(M) = \{a\}$. \square

Лемма 2.10. Пусть $L_1 = L(M_1)$ и $L_2 = L(M_2)$ для конечных автоматов M_1 и M_2 . Множества (i) $L_1 \cup L_2$, (ii) $L_1 L_2$ и (iii) L_1^* являются конечно-автоматными языками.

Доказательство. Пусть $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ и $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. Без потери общности можно считать, что $Q_1 \cap Q_2 = \emptyset$, так как в противном случае состояния можно было бы переименовать.

(i) Пусть $M = (Q_1 \cup Q_2 \cup \{q_0\}, \Sigma, \delta, q_0, F)$ — недетерминированный конечный автомат, где

- (1) q_0 — новое состояние,
- (2) $F = F_1 \cup F_2$, если $e \notin L_1 \cup L_2$, и $F = F_1 \cup F_2 \cup \{q_0\}$, если $e \in L_1 \cup L_2$,
- (3) (а) $\delta(q_0, a) = \delta(q_1, a) \cup \delta(q_2, a)$ для всех $a \in \Sigma$,
- (б) $\delta(q, a) = \delta_1(q, a)$ для всех $q \in Q_1$ и $a \in \Sigma$,
- (в) $\delta(q, a) = \delta_2(q, a)$ для всех $q \in Q_2$ и $a \in \Sigma$.

Таким образом, автомат M вначале как бы угадывает, какой из автоматов M_1 , M_2 ему моделировать. Так как M — недетерминированный автомат, то фактически он моделирует и тот, и другой. Можно показать индукцией по $i \geq 1$, что $(q_0, w) \vdash_M^i (q, e)$ тогда и только тогда, когда $q \in Q_1$ и $(q_1, w) \vdash_{M_1}^i (q, e)$ или $q \in Q_2$ и $(q_2, w) \vdash_{M_2}^i (q, e)$. Отсюда и из определения множества F вытекает, что $L(M) = L(M_1) \cup L(M_2)$.

(ii) Чтобы построить конечный автомат M , распознающий язык $L_1 L_2$, положим $M = (Q_1 \cup Q_2, \Sigma, \delta, q_1, F)$, где

$$F = \begin{cases} F_2, & \text{если } q_2 \notin F_2, \\ F_1 \cup F_2, & \text{если } q_2 \in F_2, \end{cases}$$

а функция δ определяется равенствами

- (1) $\delta(q, a) = \delta_1(q, a)$ для всех $q \in Q_1 — F_1$,
- (2) $\delta(q, a) = \delta_1(q, a) \cup \delta_2(q_2, a)$ для всех $q \in F_1$,
- (3) $\delta(q, a) = \delta_2(q, a)$ для всех $q \in Q_2$.

Таким образом, M начинает работать, моделируя M_1 . Когда M достигает заключительного состояния автомата M_1 , он может недетерминированно вообразить, что находится в начальном состоянии автомата M_2 (равенство (2)), и моделировать M_2 . Пусть $x \in L_1$ и $y \in L_2$. Тогда $(q_1, xy) \vdash_M^*(q, y)$ для некоторого $q \in F_1$. Если $x = e$, то $q = q_1$. Если $y \neq e$, то, применяя один раз равенство (2) и нуль или более раз равенство (3), получаем $(q, y) \vdash_M^*(r, e)$ для некоторого $r \in F_2$. Если $y = e$, то $q \in F$, так как $q_2 \notin F_2$. Отсюда $xy \in L(M)$. Допустим, что $w \in L(M)$. Тогда

$(q_1, w) \vdash_M^*(q, e)$ для некоторого $q \in F$. Рассмотрим отдельно два случая: $q \in F_2$ и $q \in F_1$. Пусть $q \in F_2$. Тогда $w = xay$ для некоторого $a \in \Sigma$, удовлетворяющего условиям

$$(q_1, xay) \vdash_M^*(r, ay) \vdash_M(s, y) \vdash_M^*(q, e)$$

где $r \in F_1$, $s \in Q_2$ и $\delta_2(r, a)$ содержит s . Следовательно, $x \in L_1$ и $ay \in L_2$. Пусть теперь $q \in F_1$. Тогда $q_2 \in F_2$ и $e \in L_2$. Таким образом, $w \in L_1$. Отсюда заключаем, что $L(M) = L_1 L_2$.

(iii) Построим автомат $M = (Q_1 \cup \{q'\}, \Sigma, \delta, q', F_1 \cup \{q'\})$, где q' — новое состояние, не принадлежащее Q_1 , который допускает язык L_1^* . Определим функцию δ равенствами

- (1) $\delta(q, a) = \delta_1(q, a)$, если $q \in Q — F_1$ и $a \in \Sigma$,
- (2) $\delta(q, a) = \delta_1(q, a) \cup \delta_1(q_1, a)$, если $q \in F_1$ и $a \in \Sigma$,
- (3) $\delta(q', a) = \delta_1(q_1, a)$ для $a \in \Sigma$.

Таким образом, когда M попадает в заключительное состояние автомата M_1 , он может на выбор либо продолжать моделирование M_1 , либо начать заново моделирование M_1 с начального состояния. Доказательство того, что $L(M) = L_1^*$, аналогично доказательству части (ii). Заметим, что $e \in L(M)$, так как q' — заключительное состояние. \square

Теорема 2.4. Язык допускается конечным автоматом тогда и только тогда, когда он является регулярным множеством.

Доказательство. Теорема непосредственно следует из теоремы 2.2 и лемм 2.8—2.10. \square

2.2.5. Резюме

Результаты разд. 2.2 можно сформулировать в виде следующей теоремы.

Теорема 2.5. Утверждения

- (1) L — регулярное множество,
- (2) L — праволинейный язык,
- (3) L — конечно-автоматный язык,
- (4) L — недетерминированный конечно-автоматный язык,
- (5) L обозначается регулярным выражением

эквивалентны. \square

УПРАЖНЕНИЯ

2.2.1. Какие из следующих множеств регулярны? Для тех, которые регулярны, напишите регулярные выражения.

- (а) Множество цепочек с равным числом нулей и единиц.
- (б) Множество цепочек из $\{0, 1\}^*$ с четным числом нулей и нечетным числом единиц.

- (в) Множество цепочек из $\{0, 1\}^*$, длины которых делятся на 3.
 (г) Множество цепочек из $\{0, 1\}^*$, не содержащих подцепочки 101.

2.2.2. Покажите, что множество регулярных выражений в алфавите Σ является КС-языком.

2.2.3. Покажите, что если L —произвольное регулярное множество, то существует бесконечно много регулярных выражений, обозначающих L .

2.2.4. Пусть L —регулярное множество. Прямо из определения регулярного множества получите, что L^R —регулярное множество. *Указание:* Докажите это индукцией по числу применений определения регуляриого множества, использованных при построении L как регулярного множества.

2.2.5. Докажите следующие тождества для произвольных регулярных выражений α , β и γ :

- | | |
|---------------------------------------------------------------|-------------------------------------------------------------|
| (а) $\alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma$, | (ж) $(\alpha + \beta)\gamma = \alpha\gamma + \beta\gamma$, |
| (б) $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$, | (з) $\emptyset^* = e$, |
| (в) $\alpha(\beta\gamma) = (\alpha\beta)\gamma$, | (и) $\alpha^* = \alpha = \alpha^*$, |
| (г) $\alpha e = e\alpha = \alpha$, | (к) $(\alpha^*)^* = \alpha^*$, |
| (л) $\emptyset\alpha = \alpha\emptyset = \emptyset$, | (л) $(\alpha + \beta)^* = (\alpha^*\beta^*)^*$, |
| (е) $\alpha + \alpha = \alpha$, | (м) $\alpha + \emptyset = \alpha$. |

2.2.6. Решите систему уравнений с регулярными коэффициентами

$$\begin{aligned} A_1 &= (01^* + 1)A_1 + A_2 \\ A_2 &= 11 + 1A_1 + 00A_3 \\ A_3 &= e = A_1 + A_2 \end{aligned}$$

2.2.7. Рассмотрите уравнение

$$(2.2.18) \quad X = \alpha X + \beta$$

где α и β —регулярные выражения в алфавите Σ и $X \notin \Sigma$. Покажите, что

(а) если $e \notin \alpha$, то $X = \alpha^*\beta$ —единственное решение уравнения (2.2.18);

(б) если $e \in \alpha$, то $\alpha^*\beta$ —наименьшая неподвижная точка уравнения (2.2.18), но решений бесконечно много;

(в) в любом случае каждое решение уравнения (2.2.18) имеет вид $\alpha^*(\beta \cup L)$, где L —некоторый (не обязательно регулярный) язык.

2.2.8. Решите стандартную систему, состоящую из двух уравнений общего вида

$$\begin{aligned} X &= \alpha_1 X + \alpha_2 Y + \alpha_3 \\ Y &= \beta_1 X + \beta_2 Y + \beta_3 \end{aligned}$$

2.2.9. Восполните детали доказательства леммы 2.4.

2.2.10. Докажите лемму 2.5.

2.2.11. Найдите праволинейные грамматики для тех множеств упр. 2.2.1, которые регулярны.

Определение. Грамматика $G = (N, \Sigma, P, S)$ называется *леволинейной*, если каждое правило множества P имеет вид $A \rightarrow Bw$ или $A \rightarrow w$.

2.2.12. Покажите, что язык является регулярным множеством тогда и только тогда, когда он порождается леволинейной грамматикой. *Указание:* Воспользуйтесь упр. 2.2.4.

Определение. Праволинейная грамматика $G = (N, \Sigma, P, S)$ называется *регулярной* (или *автоматной*), если

(1) все ее правила, за исключением $S \rightarrow e$, имеют вид $A \rightarrow aB$ или $A \rightarrow a$, где $B \in N$, $a \in \Sigma$,

(2) если $S \rightarrow e$ принадлежит P , то S не встречается в правых частях правил.

2.2.13. Покажите, что каждое регулярное множество порождается регулярной грамматикой. *Указание:* Это можно сделать несколькими способами. Один из них состоит в том, чтобы применить к праволинейной грамматике G последовательность преобразований, которая переведет G в эквивалентную регулярную грамматику. Другой способ—построить регулярную грамматику по конечному автомату.

2.2.14. Постройте регулярную грамматику для регулярного множества, порождаемого праволинейной грамматикой с правилами

$$\begin{aligned} A &\rightarrow B|C \\ B &\rightarrow 0B|1B|011 \\ C &\rightarrow 0D|1C|e \\ D &\rightarrow 0C|1D \end{aligned}$$

2.2.15. Опишите алгоритм, который по данной регулярной грамматике G и цепочке w определяет, принадлежит ли w языку $L(G)$.

2.2.16. Докажите утверждение (2.2.16) из доказательства теоремы 2.3.

2.2.17. Дополните доказательство леммы 2.7 (iii).

Определение. Правило $A \rightarrow \alpha$ праволинейной грамматики $G = (N, \Sigma, P, S)$ называется *бесполезным*, если в множестве Σ^* не существует таких цепочек w и x , что

$$S \Rightarrow^* wA \Rightarrow^* wa \Rightarrow^* wx$$

2.2.18. Постройте алгоритм, преобразующий праволинейную грамматику в эквивалентную ей грамматику без бесполезных правил.

***2.2.19.** Пусть $G = (N, \Sigma, P, S)$ —праволинейная грамматика. Пусть $N = \{A_1, \dots, A_n\}$ и $\alpha_{ij} = x_1 + x_2 + \dots + x_m$, где $A_i \rightarrow x_1 A_j | \dots | x_m A_j$ —это все правила вида $A_i \rightarrow y A_j$. Положим $\alpha_{io} = x_1 + \dots + x_m$, где $A_i \rightarrow x_1 | \dots | x_m$ —это все правила вида $A_i \rightarrow y$. Пусть, наконец, Q будет системой уравнений в стандартной форме $A_i = \alpha_{io} + \alpha_{i1} A_1 + \alpha_{i2} A_2 + \dots + \alpha_{in} A_n$. Покажите, что $L(G)$ —наименьшая неподвижная точка системы Q . Указание: Воспользуйтесь леммой 2.3.

2.2.20. Покажите, что язык $L(G)$ из примера 2.10—это множество цепочек в алфавите $\{0, 1\}$, длина которых делится на 3.

2.2.21. Найдите детерминированные и недетерминированные конечные автоматы для тех множеств из упр. 2.2.1, которые регулярны.

2.2.22. Покажите, что конечный автомат из примера 2.11 допускает язык $(0+1)^*00(0+1)^*$.

2.2.23. Докажите, что конечный автомат из примера 2.12 допускает язык $\{wa \mid a \in \{1, 2, 3\}\}$ и a входит в w .

2.2.24. Дополните доказательство леммы 2.10 (iii).

***2.2.25.** Двусторонний конечный автомат состоит из (недетерминированного) управляющего устройства с конечным числом состояний и входной головки, которая может двигаться по входной цепочке влево, вправо или оставаться неподвижной. Покажите, что язык допускается двусторонним конечным автоматом тогда и только тогда, когда он является регулярным множеством. Указание: Постройте детерминированный односторонний конечный автомат, который, прочитав входную цепочку $w \neq e$, помещает во внутреннюю память управляющего устройства конечную таблицу, указывающую для каждого состояния q двустороннего автомата, в каком состоянии (если таковое существует) он сойдет с правого конца цепочки w , начав работу на этом правом конце в состоянии q .

***2.2.26.** Покажите, что если позволить односторонним конечным автоматам сдвигать входную головку не на каждом такте, то это не увеличит класса определяемых ими языков.

****2.2.27.** Покажите, что для любого n существует регулярное множество, которое допускается недетерминированным конечным автоматом с n состояниями, но для распознавания которого детерминированным конечным автоматом требуется 2^n состояний.

2.2.28. Покажите, что каждый язык, допускаемый двусторонним конечным автоматом с n состояниями, допускается односторонним конечным автоматом с $2^{n(n+1)}$ состояниями.

****2.2.29.** Сколько различных языков в алфавите $\{0, 1\}$ определяются (а) недетерминированными, (б) детерминированными и (в) двусторонними конечными автоматами с двумя состояниями?

Определение. Множество S целых чисел образует *арифметическую прогрессию*, если его можно записать в виде $S = \{c, c+p, c+2p, \dots, c+ip, \dots\}$. Пусть $S(L) = \{i \mid |w| = i\}$ для некоторого $w \in L$ для любого языка L .

****2.2.30.** Покажите, что для каждого регулярного языка L множество $S(L)$ можно представить в виде объединения конечного числа арифметических прогрессий.

Открытая проблема

2.2.31. Насколько приведенная в упр. 2.2.28 верхняя граница числа состояний одностороннего автомата, моделирующего двусторонний конечный автомат, близка к наименьшему возможному числу состояний? ¹⁾

Замечания по литературе

Регулярные выражения были определены Клинн [1956]. Дополнительную информацию о регулярных выражениях можно найти в работах Мак-Нотона и Ямады [1960] и Бжозовского [1962]²⁾. Саломаа [1966a] описал две системы аксиом для регулярных выражений³⁾. Хомский и Миллер [1958] доказали эквивалентность регулярных грамматик и регулярных выражений, а Рабин и Скотт [1959]—эквивалентность детерминированных и недетерминированных конечных автоматов. Упр. 2.2.25 взято из работ Рабина и Скотта [1959] и Шепердсона [1959].

2.3. СВОЙСТВА РЕГУЛЯРНЫХ МНОЖЕСТВ

В этом разделе мы докажем несколько полезных результатов о конечных автоматах и регулярных множествах. Особенно важный результат состоит в том, что для каждого регулярного множества по существу однозначно находится определяющий его конечный автомат с минимальным числом состояний.

2.3.1. Минимизация конечных автоматов

По данному конечному автомату M можно найти наименьший эквивалентный ему конечный автомат, исключив все недостижимые состояния и затем склеив лишние состояния. Лишние

¹⁾ В связи с этой проблемой см. работу Хартманна [1970]. —Прим. перев.

²⁾ О регулярных выражениях написано много, см. монографии, упомянутые в замечаниях по литературе к разд. 2.3. —Прим. перев.

³⁾ В связи с этим см. также работы Редько [1964] и Саломаа [1966b]. —Прим. перев.

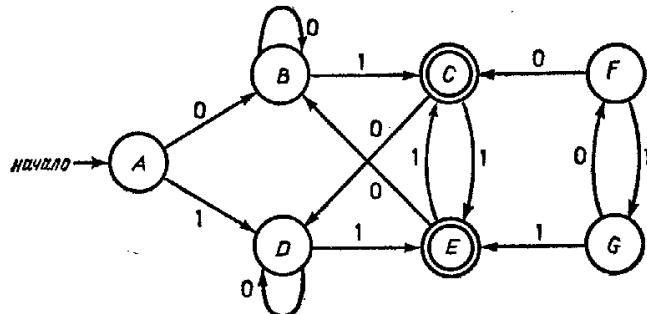
состояния определяются с помощью разбиения множества всех достижимых состояний на классы эквивалентности так, что каждый класс содержит неразличимые состояния и выбирается как можно шире. Потом из каждого класса берется один представитель в качестве состояния сокращенного, или приведенного, автомата. Таким способом можно сократить объем автомата M , если M содержит недостижимые состояния или два и более неразличимых состояний. Мы покажем, что этот приведенный автомат — наименьший из конечных автоматов, распознающих регулярное множество, определяемое первоначальным автоматом M .

Определение. Пусть $M = (Q, \Sigma, \delta, q_0, F)$ — конечный автомат, а q_1 и q_2 — различные его состояния. Будем говорить, что цепочка $x \in \Sigma^*$ различает состояния q_1 и q_2 , если $(q_1, x) \vdash^*(q_2, e)$, $(q_2, x) \vdash^*(q_1, e)$ и одно из состояний q_3 и q_4 , принадлежит F , а другое нет. Будем говорить, что q_1 и q_2 k -неразличимы, и писать $q_1 \equiv^k q_2$, если не существует такой цепочки x , различающей q_1 и q_2 , у которой $|x| \leq k$. Будем говорить, что состояния q_1 и q_2 неразличимы, и писать $q_1 \equiv q_2$, если они k -неразличимы для любого $k \geq 0$.

Состояние $q \in Q$ называется недостижимым, если не существует такой входной цепочки x , что $(q_0, x) \vdash^*(q, e)$.

Автомат M называется приведенным, если в Q нет недостижимых состояний и нет двух неразличимых состояний.

Пример 2.14. Рассмотрим конечный автомат M , диаграмма которого показана на рис. 2.5.

Рис. 2.5. Диаграмма автомата M .

Чтобы сократить M , заметим сперва, что состояния F и G недостижимы из начального состояния A , так что их можно устраниć. Позднее, применяя алгоритм 2.2, мы увидим, что классами эквивалентности отношения \equiv будут $\{A\}$, $\{B, D\}$ и $\{C, E\}$. Тогда, взяв представителями этих множеств состояния

p, q и r соответственно, можно получить конечный автомат, изображенный на рис. 2.6, который является приведенным автоматом, эквивалентным M . \square

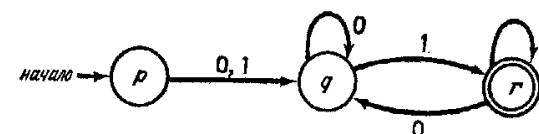


Рис. 2.6. Приведенный автомат.

Лемма 2.11. Пусть $M = (Q, \Sigma, \delta, q_0, F)$ — конечный автомат с n состояниями. Состояния q_1 и q_2 неразличимы тогда и только тогда, когда они $(n-2)$ -неразличимы.

Доказательство. Необходимость условия тривиальна. Достаточность тривиальна в тех случаях, когда F имеет 0 или n элементов. Поэтому рассмотрим случай, когда число элементов множества F отлично от 0 или n .

Покажем, что

$$\equiv \subseteq \equiv^{n-2} \subseteq \equiv^{n-3} \subseteq \dots \subseteq \equiv^2 \subseteq \equiv^1 \subseteq \equiv^0$$

Для этого заметим, что для любых состояний q_1 и q_2

(1) $q_1 \equiv^0 q_2$ тогда и только тогда, когда q_1 и q_2 оба либо принадлежат, либо не принадлежат F ,

(2) $q_1 \equiv^k q_2$ тогда и только тогда, когда $q_1 \equiv^{k-1} q_2$ и $\delta(q_1, a) = \delta(q_2, a)$ для всех $a \in \Sigma$.

Отношение \equiv^0 грубейшее; оно разбивает Q на два класса: F и $Q - F$. Если $\equiv^{k+1} \neq \equiv^k$, то отношение \equiv^{k+1} тоньше, чем \equiv^k , т. е. в нем по крайней мере на один класс эквивалентности больше, чем в \equiv^k . Так как каждое из множеств F и $Q - F$ содержит не более $n-1$ элементов, можно получить не более $n-2$ последовательных уточнений отношения \equiv^0 . Если $\equiv^{k+1} = \equiv^k$ для некоторого k , то в силу (2) $\equiv^{k+1} = \equiv^{k+2} = \dots$. Таким образом, \equiv — это первое из отношений \equiv^k , для которых $\equiv^{k+1} = \equiv^k$. \square

Можно дать интересную интерпретацию леммы 2.11: если два состояния можно различить, то их можно различить с помощью входной цепочки, длина которой меньше числа состояний конечного автомата. Приведем алгоритм, описывающий процесс минимизации числа состояний конечного автомата.

Алгоритм 2.2. Построение канонического конечного автомата.

Вход. Конечный автомат $M = (Q, \Sigma, \delta, q_0, F)$.

Выход. Эквивалентный приведенный конечный автомат M' .

Метод.

Шаг 1: Применив к диаграмме автомата M алгоритм 0.3, найти состояния, недостижимые из q_0 . УстраниТЬ все недостижимые состояния.

Шаг 2: Строить отношения эквивалентности $\equiv^0, \equiv^1, \dots$, как описано в лемме 2.11, до тех пор, пока два из них не совпадут: $\equiv^{k+1} = \equiv^k$. Взять в качестве \equiv отношение \equiv^k .

Шаг 3: Построить конечный автомат $M' = (Q', \Sigma, \delta', q'_0, F')$, где
(а) Q' —множество классов эквивалентности отношения \equiv (обозначим через $[p]$ класс эквивалентности отношения \equiv , содержащий состояние p),

(б) $\delta'([p], a) = [q]$, если $\delta(p, a) = q$,

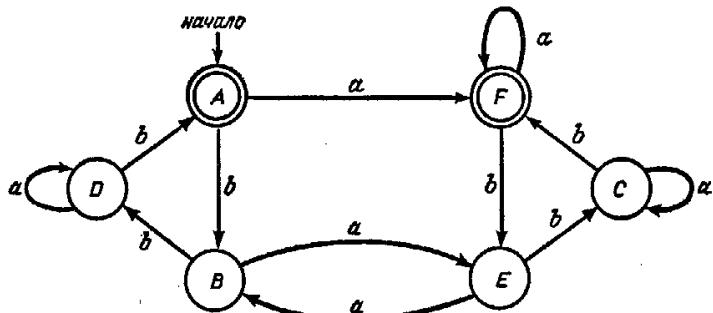
(в) q'_0 —это $[q_0]$,

(г) $F' = \{[q] \mid q \in F\}$. \square

Можно непосредственно показать, что шаг 3(б) непротиворечив, т. е. какой элемент класса $[p]$ ни взять, для $\delta([p], a)$ будет один и тот же класс. Доказательство равенства $L(M) = L(M')$ тоже просто и оставляется в качестве упражнения. Докажем теперь, что автомат с меньшим числом состояний, чем у M' , не может допускать $L(M)$.

Теорема 2.6. Автомат M' , который строится алгоритмом 2.2, имеет наименьшее число состояний среди всех конечных автоматов, допускающих язык $L(M)$.

Доказательство. Предположим, что M'' имеет меньше состояний, чем M' , и что $L(M'') = L(M)$. В силу шага 1 алгоритма 2.2 каждое состояние автомата M' достижимо. Так как

Рис. 2.7. Диаграмма автомата M .

M'' имеет меньше состояний, то найдутся цепочки w и x , переводящие состояние q'_0 в разные состояния, а q''_0 (начальное состояние автомата M'')—в одно и то же: $(q'_0, w) \vdash_{M''}^*(q, e)$ и $(q''_0, x) \vdash_{M''}^*(q, e)$. Следовательно, w и x переводят автомат M в раз-

личимые состояния, скажем p и r . Это значит, что существует такая цепочка y , что точно одна из цепочек wy и xy принадлежит $L(M)$. Но wy и xy должны переводить M'' в одно и то же состояние s , для которого $(q, y) \vdash_{M''}^*(s, e)$. Таким образом, точно одна из цепочек wy и xy не может принадлежать $L(M'')$, а это противоречит предположению о том, что $L(M'') = L(M)$. \square

Таблица 2.3

	a	b
$[A]$	$[A]$	$[B]$
$[B]$	$[B]$	$[C]$
$[C]$	$[C]$	$[A]$

Пример 2.15. Найдем приведенный конечный автомат, эквивалентный автомата M , диаграмма которого показана на рис. 2.7. Отношения \equiv^k для $k \geq 0$ имеют следующие классы эквивалентности:

Классы отношения \equiv^0 : $\{A, F\}, \{B, C, D, E\}$

Классы отношения \equiv^1 : $\{A, F\}, \{B, E\}, \{C, D\}$

Классы отношения \equiv^2 : $\{A, F\}, \{B, E\}, \{C, D\}$

Так как $\equiv^2 = \equiv^1$, то $\equiv = \equiv^1$. Приведенным автоматом M' будет автомат $(\{[A], [B], [C]\}, \{a, b\}, \delta', A, \{[A]\})$, где функция δ' определяется табл. 2.3. Здесь мы выбрали $[A]$ для представления класса $\{A, F\}$, $[B]$ —для представления $\{B, E\}$ и $[C]$ —для $\{C, D\}$. \square

2.3.2. Лемма о разрастании для регулярных множеств

Теперь мы хотим получить характеристику регулярных множеств, которая будет полезна для доказательства того, что некоторые языки не являются регулярными. Следующую теорему назовем леммой о „разрастании“, потому что она в сущности говорит о том, что если даны регулярное множество и достаточно длинная цепочка в нем, то в этой цепочке можно найти непустую подцепочку, которую можно повторить сколько угодно раз (т. е. она „разрастается“), и все полученные таким образом „новые“ цепочки будут принадлежать тому же регулярному множеству. С помощью этой леммы часто приводят к противоречию предположение о том, что некоторое множество регулярно.

Теорема 2.7. Лемма о разрастании для регулярных множеств. Пусть L — регулярное множество. Существует такая константа p , что если $w \in L$ и $|w| \geq p$, то цепочку w можно записать в виде xuz , где $0 < |y| \leq p$ и $xy^iz \in L$ для всех $i \geq 0$.

Доказательство. Пусть $M = (Q, \Sigma, \delta, q_0, F)$ — конечный автомат с n состояниями и $L(M) = L$. Пусть $p = n$. Если $w \in L$ и $|w| \geq n$, рассмотрим последовательность конфигураций, которую проходит автомат M , допуская цепочку w . Так как в этой последовательности по крайней мере $n+1$ конфигураций, то среди первых $n+1$ конфигураций найдутся две с одинаковыми состояниями. Поэтому должна быть такая последовательность тактов, что

$$(q_0, xuz) \vdash^* (q_1, yz) \vdash^{-k} (q_1, z) \vdash^* (q_2, e)$$

для некоторого q_1 и $0 < k \leq n$. Отсюда $0 < |y| \leq n$. Но тогда для любого $i > 0$ автомат может проделать последовательность тактов

$$\begin{aligned} (q_0, xy^iz) &\vdash^* (q_1, y^iz) \\ &\vdash^+ (q_1, y^{i-1}z) \\ &\vdots \\ &\vdash^+ (q_1, yz) \\ &\vdash^+ (q_1, z) \\ &\vdash^* (q_2, e) \end{aligned}$$

Так как цепочка $w = xuz$ принадлежит L , то и $xy^iz \in L$ для всех $i > 1$. Случай $i = 0$ исследуется аналогично. \square

Пример 2.16. С помощью леммы о разрастании докажем, что язык $L = \{0^n 1^n \mid n \geq 1\}$ не является регулярным множеством. Допустим, что L регулярен. Тогда для достаточно большого n цепочку $0^n 1^n$ можно записать в виде xuz , причем $y \neq e$ и $xy^iz \in L$ для всех $i \geq 0$. Если $y \in 0^+$ или $y \in 1^+$, то $xz = xy^0z \notin L$. Если $y \in 0^+ 1^+$, то $xyuz \notin L$. Полученное противоречие доказывает, что язык L не может быть регулярным. \square

2.3.3. Свойства замкнутости регулярных множеств

Множество A называется замкнутым относительно n -местной операции θ , если $\theta(a_1, a_2, \dots, a_n) \in A$ всегда, когда $a_i \in A$ для всех $1 \leq i \leq n$. Например, множество целых чисел замкнуто относительно операции сложения.

В этом разделе мы рассмотрим несколько операций, относительно которых класс регулярных множеств замкнут. С помощью этих свойств замкнутости можно будет определить, регулярны ли некоторые языки. Мы уже знаем, что если L_1 и L_2 — регулярные множества, то множества $L_1 \cup L_2$, $L_1 L_2$ и L_1^* тоже регулярны.

Определение. Класс множеств называется *булевой алгеброй множеств*, если он замкнут относительно объединения, пересечения и дополнения.

Теорема 2.8. Для любого алфавита Σ класс регулярных множеств, содержащихся в Σ^* , является булевой алгеброй множеств.

Доказательство. Докажем замкнутость относительно дополнения. Замкнутость относительно объединения мы уже имеем, а замкнутость относительно пересечения будет следовать из теоретико-множественного закона $\overline{A \cap B} = \overline{A} \cup \overline{B}$ (упр. 0.1.4). Пусть $M = (Q, \Delta, \delta, q_0, F)$ — конечный автомат, у которого $\Delta \subseteq \Sigma$. Легко показать, что каждое регулярие множество $L \subseteq \Sigma^*$ допускается некоторым таким автоматом. Тогда конечный автомат $M' = (Q, \Delta, \delta, q_0, Q - F)$ допускает $\Delta^* - L(M)$. Заметим, что здесь использован тот факт, что автомат M полностью определен. Далее, дополнение $\overline{L(M)}$ относительно Σ^* можно представить в виде $\overline{L(M)} = L(M') \cup \Sigma^*(\Sigma - \Delta) \Sigma^*$. Так как множество $\Sigma^*(\Sigma - \Delta) \Sigma^*$ регулярно, то регулярность множества $L(M)$ следует из замкнутости регулярных множеств относительно объединения. \square

Теорема 2.9. Класс регулярных множеств замкнут относительно обращения.

Доказательство. Пусть $M = (Q, \Sigma, \delta, q_0, F)$ — конечный автомат, определяющий регулярное множество L . Чтобы определить L^R , „запустим M в обратном направлении“. Иными словами, возьмем недетерминированный конечный автомат $M' = (Q \cup \{q'_0\}, \Sigma, \delta', q'_0, F')$, где $F' = \{q_0\}$, если $e \notin L$, и $F' = \{q_0, q'_0\}$, если $e \in L$. Функцию δ' определим так:

- (1) $\delta'(q'_0, a)$ содержит q , если $\delta(q, a) \in F$.
- (2) $\delta'(q', a)$ для всех $q' \in Q$ и $a \in \Sigma$ содержит q , если $\delta(q, a) = q'$.

Легко показать, что $(q_0, w) \vdash_M^+ (q, e)$ для $q \in F$ тогда и только тогда, когда $(q'_0, w^R) \vdash_{M'}^+ (q_0, e)$. Таким образом, $L(M') = (L(M))^R = L^R$. \square

Класс регулярных множеств замкнут относительно самых распространенных операций над языками. Многие из этих свойств замкнутости изучаются в упражнениях.

2.3.4. Разрешимые проблемы, связанные с регулярными множествами

Мы изложили несколько способов описания регулярных множеств, таких, как регулярные выражения и конечные автоматы. В связи с этими способами представления языков естественно возникают алгоритмические проблемы. Мы коснемся здесь трех проблем:

Проблема принадлежности: „Даны определенного типа описание языка и цепочка w ; принадлежит ли w этому языку?“

Проблема пустоты: „Дано определенного типа описание языка; пуст ли этот язык?“

Проблема эквивалентности: „Даны два описания одинакового типа; определяют ли они один и тот же язык?“

Мы рассмотрим следующие типы описаний регулярных множеств:

- (1) регулярные выражения,
- (2) праволинейные грамматики,
- (3) конечные автоматы.

Сначала дадим алгоритмы, решающие три упомянутые выше проблемы в том случае, когда языки описываются с помощью конечных автоматов.

Алгоритм 2.3. Решение проблемы принадлежности для конечных автоматов.

Вход. Конечный автомат $M = (Q, \Sigma, \delta, q_0, F)$ и цепочка $w \in \Sigma^*$.

Выход. „ДА“, если $w \in L(M)$; „НЕТ“, если $w \notin L(M)$.

Метод. Пусть $w = a_1 a_2 \dots a_n$. Найти последовательно состояния $q_1 = \delta(q_0, a_1), q_2 = \delta(q_1, a_2), \dots, q_n = \delta(q_{n-1}, a_n)$. Если $q_n \in F$, сказать „ДА“; если $q_n \notin F$, сказать „НЕТ“. \square

Корректность алгоритма 2.3 слишком очевидна, чтобы ее обсуждать. Однако стоит обсудить временнюю и емкостную сложности этого алгоритма. Естественными мерами сложности в данном случае служат число шагов и число ячеек памяти, требуемых вычислительной машине с произвольным доступом к памяти, у которой каждая ячейка памяти может хранить целое число произвольной величины. (Фактически в реальных вычислительных машинах величина чисел ограничена, но эта граница так велика, что, несомненно, для тех конечных автоматов, которые имеет смысл рассматривать, она никогда не будет достигнута. Таким образом, предположение о неограниченности чисел является здесь разумным математическим упрощением.)

Легко видеть, что время работы алгоритма линейно зависит от длины цепочки. Однако не совсем ясно, влияет ли на время работы „объем“ автомата M . Мы должны предположить, что фак-

тически описание автомата M представляет собой цепочку символов, взятых из некоторого конечного алфавита. Поэтому можно предположить, что именами состояний являются $q_0, q_1, \dots, q_i, \dots$, где индексы записаны в виде двоичных чисел. Аналогично обстоит дело с именами входных символов a_1, a_2, \dots . Если предполагается машина обычного типа, то для хранения функции δ можно построить двумерный массив, такой, что в ячейке (i, j) находится $\delta(q_i, a_j)$. Итак, общее время работы алгоритма будет состоять из времени, необходимого для построения этой таблицы, которое пропорционально длине описания M , и времени выполнения самого алгоритма, которое пропорционально $|w|$.

Требуемый объем памяти — это главным образом число ячеек, необходимых для хранения таблицы, оно пропорционально длине описания M .

Теперь дадим алгоритмы решения проблем пустоты и эквивалентности, когда описаниями языков служат конечные автоматы.

Алгоритм 2.4. Решение проблемы пустоты для конечных автоматов.

Вход. Конечный автомат $M = (Q, \Sigma, \delta, q_0, F)$.

Выход. „ДА“, если $L(M) \neq \emptyset$, „НЕТ“ в противном случае.

Метод. Вычислить множество состояний, достижимых из q_0 . Если это множество содержит какое-нибудь заключительное состояние, сказать „ДА“, в противном случае сказать „НЕТ“. \square

Алгоритм 2.5. Решение проблемы эквивалентности для конечных автоматов.

Вход. Два конечных автомата $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ и $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$, таких, что $Q_1 \cap Q_2 = \emptyset$.

Выход. „ДА“, если $L(M_1) = L(M_2)$, „НЕТ“ в противном случае.

Метод. Построить конечный автомат

$$M = (Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta_1 \cup \delta_2, q_1, F_1 \cup F_2)$$

С помощью леммы 2.11 определить, различимы ли состояния q_1 и q_2 . Если да, то сказать „НЕТ“, в противном случае сказать „ДА“. \square

Отметим, что для решения проблемы эквивалентности можно было бы воспользоваться алгоритмом 2.4, так как $L(M_1) = L(M_2)$ тогда и только тогда, когда

$$(L(M_1) \cup \overline{L(M_2)}) \cup (\overline{L(M_1)} \cap L(M_2)) = \emptyset$$

Теперь займемся вопросом разрешимости проблем принадлежности, пустоты и эквивалентности в тех случаях, когда регулярные множества представляются регулярными выражениями

или праволинейными грамматиками. Легко показать, что для этих способов представления языков все три проблемы тоже разрешимы. Регулярное выражение можно превратить в эквивалентный конечный автомат с помощью алгоритма, который извлекается из доказательств лемм 2.9 и 2.10. Затем к полученному автомату можно применить один из алгоритмов 2.3—2.5. Праволинейную грамматику можно превратить в эквивалентное регулярное выражение с помощью алгоритма 2.1 и алгоритма, который неявно содержится в доказательстве теоремы 2.2. Ясно, что эти алгоритмы слишком непрямые, чтобы их можно было использовать на практике. Прямые быстро работающие алгоритмы будут предметом нескольких упражнений.

Сформулируем эти результаты в виде теоремы.

Теорема 2.10. *Если множества определяются конечными автоматами, регулярными выражениями или праволинейными грамматиками, то проблемы принадлежности, пустоты и эквивалентности для регулярных множеств алгоритмически разрешимы.* \square

Подчеркнем, что эти три проблемы разрешимы не для любого способа представления регулярных множеств. Рассмотрим, в частности, следующий пример.

Пример 2.17. Можно устроить перечисление машин Тьюринга (см. упражнения к разд. 0.4) в виде списка M_1, M_2, \dots . Затем можно определить представление регулярных множеств с помощью натуральных чисел, а именно

(1) если M_i допускает регулярное множество, то пусть число i представляет это множество,

(2) если M_i допускает не регулярное множество, то пусть число i представляет множество $\{\epsilon\}$.

Каждое положительное целое число представляет, таким образом, некоторое регулярное множество, и каждое регулярное множество представляется хотя бы одним таким числом. Известно, что для использованного здесь представления машин Тьюринга проблема пустоты неразрешима (упр. 0.4.16). Допустим, что разрешима проблема: „Представляет ли число i пустое множество \emptyset ?“ Легко видеть, что машина M_i допускает \emptyset тогда и только тогда, когда число i представляет \emptyset . Следовательно, если регулярные множества определяются только что описанным способом, то проблема пустоты для них неразрешима. \square

УПРАЖНЕНИЯ

2.3.1. Дан конечный автомат с n достижимыми состояниями. Каково наименьшее число состояний эквивалентного ему приведенного автомата?

2.3.2. Найдите конечный автомат с минимальным числом состояний для языка, определяемого автоматом $M = \{A, B, C, D, E, F\}, \{0, 1\}, \delta, A, \{E, F\}$, где функция δ задается табл. 2.4.

Таблица 2.4

δ	Вход	
	0	1
Состояние A	B	C
B	E	F
C	A	A
D	F	E
E	D	F
F	D	E

2.3.3. Покажите, что для любого n существует такой конечный автомат с n состояниями, что $=^{n-2} \neq =^{n-3}$.

2.3.4. Докажите, что для автомата M' , который строится алгоритмом 2.2, $L(M') = L(M)$.

Определение. Отношение R , определенное на Σ^* , называется *правоинвариантным*, если xRy влечет $xxRyz$ для всех x, y, z из Σ^* .

2.3.5. Покажите, что L — регулярное множество тогда и только тогда, когда L можно представить в виде объединения некоторого числа классов эквивалентности правоинвариантного отношения эквивалентности R конечного индекса. **Указание:** Для доказательства необходимости определите отношение R , положив xRy тогда и только тогда, когда $(q_0, x) \xrightarrow{*} (p, e), (q_0, y) \xrightarrow{*} (q, e)$ и $p = q$ (т. е. цепочки x и y переводят конечный автомат, определяющий L , в одно и то же состояние). Покажите, что R — правоинвариантное отношение эквивалентности конечного индекса. Для доказательства достаточности постройте конечный автомат, определяющий L , беря в качестве его состояний классы эквивалентности отношения R .

Определение. Отношение E назовем *грубейшим правоинвариантным* отношением эквивалентности для языка $L \subseteq \Sigma^*$, если xEy тогда и только тогда, когда для всех $z \in \Sigma^*$ цепочка xz принадлежит L в точности в тех случаях, когда $yz \in L$.

Следующее упражнение устанавливает, что каждое правоинвариантное отношение эквивалентности, определяющее язык L , содержится в E .

2.3.6. Пусть L — объединение некоторых классов эквивалентности правоинвариантного отношения эквивалентности R , опре-

деленного на Σ^* . Пусть E — грубейшее правонвариантное отношение эквивалентности для языка L . Покажите, что $E \equiv R$.

*2.3.7. Покажите, что грубейшее правонвариантное отношение эквивалентности для языка L имеет конечный индекс тогда и только тогда, когда L — регулярное множество.

2.3.8. Пусть $M = (Q, \Sigma, \delta, q_0, F)$ — приведенный конечный автомат. Определим отношение E на Σ^* , положив xEy тогда и только тогда, когда $(q_0, x) \vdash^* (p, e)$, $(q_0, y) \vdash^* (q, e)$ и $p = q$. Покажите, что E — грубейшее правонвариантное отношение эквивалентности для $L(M)$.

Определение. Отношение эквивалентности R на Σ^* называется *отношением конгруэнтности*, если R одновременно лево- и правоинвариантно (т. е. если xRy , то $wxzRwyg$ для всех w, x, y, z из Σ^*).

2.3.9. Покажите, что L — регулярное множество тогда и только тогда, когда L можно представить в виде объединения некоторых классов эквивалентности отношения конгруэнтности конечного индекса.

2.3.10. Покажите, что если M_1 и M_2 — приведенные конечные автоматы, для которых $L(M_1) = L(M_2)$, то их диаграммы равны.

*2.3.11. Покажите, что нижняя граница временной сложности алгоритма 2.2 равна n^2 (т. е. существует такой автомат M с n состояниями, что алгоритму 2.2 требуется произвести n^2 операций, чтобы найти приведенный автомат, эквивалентный M). Какова верхняя граница временной сложности алгоритма 2.2?

Можно найти алгоритм минимизации числа состояний конечного автомата, время работы которого всегда не превосходит $n \log n$, где n — число состояний минимизируемого автомата. Больше всего времени требует тот этап алгоритма 2.2, который на шаге 2 находит классы эквивалентности отношения \equiv методом леммы 2.11. Однако на шаге 2 можно использовать другой алгоритм, который позволяет сократить время работы алгоритма 2.2 до $n \log n$.

Этот новый алгоритм несколько иначе, чем в лемме 2.11, измельчает разбиения множества состояний. Вначале все состояния разбиты на заключительные и незаключительные. Далее допустим, что некоторое разбиение состоит из классов $\pi_1, \pi_2, \dots, \pi_{k-1}$. Для того чтобы сделать это разбиение более мелким, выберем класс π_i и входной символ a . Каждый класс π_j , содержащий такое состояние q , что $\delta(q, a) \in \pi_i$, расщепляется на два класса: $\pi'_j = \{q \mid q \in \pi_j \text{ и } \delta(q, a) \in \pi_i\}$ и $\pi''_j = \pi_j - \pi'_j$. Таким обра-

зом, в отличие от метода леммы 2.11 здесь происходит расщепление класса, как только выясняется, что данный вход переводит некоторые состояния этого класса в такие, неэквивалентность которых установлена ранее.

Алгоритм 2.6. Нахождение классов неразличимых состояний конечного автомата.

Вход. Конечный автомат $M = (Q, \Sigma, \delta, q_0, F)$.

Выход. Классы эквивалентности отношения \equiv .

Метод.

(1) Найти $\delta^{-1}(q, a) = \{p \mid \delta(p, a) = q\}$ для всех $q \in Q$ и $a \in \Sigma$. Положить $\pi_{i,a} = \{q \mid q \in \pi_i \text{ и } \delta^{-1}(q, a) \neq \emptyset\}$.

(2) Положить $\pi_1 = F$ и $\pi_2 = Q - F$.

(3) Для всех $a \in \Sigma$ определить множества индексов

$$I(a) = \begin{cases} \{1\}, & \text{если } \#\pi_{1,a} \leq \#\pi_{2,a}, \\ \{2\} & \text{в противном случае.} \end{cases}$$

(4) Положить $k = 3$.

(5) Выбрать $a \in \Sigma$ и $i \in I(a)$. Если $I(a) = \emptyset$ для всех $a \in \Sigma$, остановиться. Выход — множество классов $\pi_1, \pi_2, \dots, \pi_{k-1}$.

(6) Удалить i из множества $I(a)$.

(7) Для всех $j < k$, для которых существует $q \in \pi_j$ и $\delta(q, a) \in \pi_i$, сделать шаги (a) — (г):

(a) Положить $\pi'_j = \{q \mid \delta(q, a) \in \pi_i \text{ и } q \in \pi_j\}$ и $\pi''_j = \pi_j - \pi'_j$.

(б) Заменить π_j на π'_j и положить $\pi_k = \pi''_j$; построить новые $\pi_{j,a}$ и $\pi_{k,a}$ для всех $a \in \Sigma$.

(в) Для всех $a \in \Sigma$ изменить $I(a)$, положив

$$I(a) = \begin{cases} I(a) \cup \{j\}, & \text{если } j \notin I(a) \text{ и } 0 < \#\pi_{j,a} \leq \#\pi_{k,a}, \\ I(a) \cup \{k\} & \text{в противном случае.} \end{cases}$$

(г) Положить $k = k + 1$.

(8) Перейти к шагу 5. \square

2.3.12. Примените алгоритм 2.6 к конечным автоматам из примера 2.15 и упр. 2.3.2.

2.3.13. Докажите, что алгоритм 2.6 правильно находит классы неразличимых состояний конечного автомата.

**2.3.14. Покажите, что алгоритм 2.6 можно реализовать за время $n \log n$.

2.3.15. Покажите, что следующие множества не регулярны:

(а) $\{0^n 1 0^n \mid n \geq 1\}$,

(б) $\{\omega\omega \mid \omega \in \{0, 1\}^*\}$,

(в) $L(G)$, где G определяется правилами $S \rightarrow aSbS \mid c$,

(г) $\{a^{n^2} \mid n \geq 1\}$,

- (д) $\{a^p \mid p \text{ — простое число}\},$
 (е) $\{w \mid w \in \{0, 1\}^*\text{ и }w \text{ имеет одинаковые числа нулей и единиц}\}.$

2.3.16. Пусть $f(m)$ — монотонно возрастающая функция и для каждого n существует такое m , что $f(m+1) \geq f(m) + n$. Покажите, что множество $\{a^{f(m)} \mid m \geq 1\}$ не регулярно.

Определение. Пусть L_1 и L_2 — языки. Определим следующие операции:

- (1) $L_1/L_2 = \{w \mid wx \in L_1 \text{ для некоторого } x \in L_2\},$
- (2) $1N1T(L_1) = \{w \mid wx \in L_1 \text{ для некоторого } x\},$
- (3) $FIN(L_1) = \{w \mid xw \in L_1 \text{ для некоторого } x\},$
- (4) $SUB(L_1) = \{w \mid xwy \in L_1 \text{ для некоторых } x \text{ и } y\},$
- (5) $MIN(L_1) = \{w \mid w \in L_1 \text{ и никакой собственный префикс цепочки } w \text{ не принадлежит } L_1\},$
- (6) $MAX(L_1) = \{w \mid w \in L_1 \text{ и не существует такого } x \neq e, \text{ что } wx \in L_1\}.$

Пример 2.18. Пусть $L_1 = \{0^n 1^n 0^m \mid n, m \geq 1\}$ и $L_2 = 1^* 0^*$. Тогда

$$L_1/L_2 = L_1 \cup \{0^i 1^j \mid i \geq 1, j \leq i\}$$

$$L_2/L_1 = \emptyset$$

$$INIT(L_1) = L_1 \cup \{0^i 1^j \mid i \geq 1, j \leq i\} \cup 0^*$$

$$FIN(L_1) = \{0^i 1^j 0^k \mid k \geq 1, j \geq 1, i \leq j\} \cup 1^+ 0^+ \cup 0^*$$

$$SUB(L_1) = \{0^i 1^j 0^k \mid i \leq j\} \cup 1^* 0^* \cup 0^* 1^*$$

$$MIN(L_1) = \{0^n 1^n 0^m \mid n \geq 1\}$$

$$MAX(L_1) = \emptyset \quad \square$$

***2.3.17.** Пусть L_1 и L_2 — регулярные множества. Покажите, что следующие множества регулярны:

- (а) $L_1/L_2,$
- (б) $INIT(L_1),$
- (в) $FIN(L_1),$
- (г) $SUB(L_1),$
- (д) $MIN(L_1),$
- (е) $MAX(L_1).$

***2.3.18.** Пусть L_1 — регулярное множество, а L_2 — произвольный язык. Покажите, что множество L_1/L_2 регулярно. Существует ли алгоритм, который находит конечный автомат для L_1/L_2 , если дан автомат для L_1 ?

Определение. Производную $D_x \alpha$ регулярного выражения α по $x \in \Sigma^*$ можно определить рекурсивно:

- (1) $D_e \alpha = \alpha.$
- (2) Для $a \in \Sigma$
 - (а) $D_a \emptyset = \emptyset,$
 - (б) $D_a e = \emptyset,$

- (в) $D_a b = \begin{cases} \emptyset, & \text{если } a \neq b, \\ e, & \text{если } a = b, \end{cases}$
- (г) $D_a(\alpha + \beta) = D_a \alpha + D_a \beta,$
- (д) $D_a(\alpha \beta) = \begin{cases} (D_a \alpha) \beta, & \text{если } e \notin \alpha, \\ (D_a \alpha) \beta + D_a \beta, & \text{если } e \in \alpha, \end{cases}$
- (е) $D_a \alpha^* = (D_a \alpha) \alpha^*,$

$$(3) D_{ax} \alpha = D_x(D_a \alpha) \text{ для } a \in \Sigma \text{ и } x \in \Sigma^*.$$

2.3.19. Покажите, что если $\alpha = 10^* 1$, то

- (а) $D_e = 10^* 1,$
- (б) $D_0 \alpha = \emptyset,$
- (в) $D_1 \alpha = 0^* 1.$

***2.3.20.** Покажите, что если α — регулярное выражение, обозначающее регулярное множество R , то $D_x \alpha$ обозначает

$$x \setminus R = \{w \mid xw \in R\}.$$

****2.3.21.** Пусть L — регулярное множество. Покажите, что множество $\{x \mid xy \in L \text{ для некоторого } y, \text{ такого, что } |x| = |y|\}$ регулярно.

Следующее упражнение обобщает упр. 2.3.21.

****2.3.22.** Пусть L — регулярное множество и $f(x)$ — полином от x с неотрицательными целыми коэффициентами. Покажите, что множество $\{w \mid wy \in L \text{ для некоторого } y, \text{ такого, что } |y| = f(|w|)\}$ регулярно.

***2.3.23.** Пусть L — регулярное множество и h — гомоморфизм. Покажите, что $h(L)$ и $h^{-1}(L)$ — регулярные множества.

2.3.24. Докажите корректность алгоритмов 2.4 и 2.5.

2.3.25. Оцените временную и емкостную сложности алгоритмов 2.4 и 2.5.

2.3.26. Дайте формальное доказательство теоремы 2.10. Обратите внимание, что недостаточно просто показать, что, скажем, для каждого регулярного выражения существует конечный автомат, допускающий обозначаемое им множество. Нужно показать, что существует алгоритм, который по регулярному выражению строит этот автомат. В этой связи см. пример 2.17.

***2.3.27.** Дайте эффективный алгоритм минимизации числа состояний не полностью определенного детерминированного конечного автомата.

2.3.28. Дайте эффективные алгоритмы, решающие проблемы принадлежности, пустоты и эквивалентности для

- (а) регулярных выражений,

- (б) праволинейных грамматик,
 (в) недетерминированных конечных автоматов.

****2.3.29.** Покажите, что проблемы принадлежности и эквивалентности неразрешимы для способа представления регулярных множеств, указанного в примере 2.17.

***2.3.30.** Покажите, что проблема: „Бесконечен ли язык $L(M)$?“ разрешима для конечных автоматов. *Указание:* Покажите, что для автомата M с n состояниями язык $L(M)$ бесконечен тогда и только тогда, когда $L(M)$ содержит цепочку ω , для которой $n \leq |\omega| < 2n$.

***2.3.31.** Докажите алгоритмическую разрешимость проблемы включения $L(M_1) \subseteq L(M_2)$, где M_1 и M_2 — конечные автоматы.

Открытая проблема

2.3.32. Найдите быстрый алгоритм (скажем, такой, который для автомата с n состояниями требует времени не более n^k , где k — константа), дающий недетерминированный конечный автомат с минимальным числом состояний, эквивалентный данному.

Упражнения на программирование

2.3.33. Напишите программу, которая воспринимает в качестве входа конечный автомат, праволинейную грамматику или регуляриое выражение и выдает на выходе эквивалентное регуляриое выражение, конечный автомат или праволинейную грамматику. С помощью такой программы можно, например, по регуляриому выражению построить эквивалентный ему конечный автомат.

2.3.34. Постройте программу, которая воспринимает в качестве входа описание конечного автомата и выдает на выходе эквивалентный ему приведенный автомат.

2.3.35. Напишите программу, которая будет моделировать недетерминированный конечный автомат.

2.3.36. Постройте программу, определяющую, эквивалентны ли два описания регуляриого множества (тип описаний надо взять такой, чтобы проблема эквивалентности была разрешима).

Замечания по литературе

Минимизация конечного автомата впервые изучалась Хафменом [1954] и Муром [1956]. Свойства замкнутости регуляриых множеств и результаты о разрешимости взяты из статьи [Рабин и Скотт, 1959].

Упражнения содержат лишь некоторые из многих результатов, касающихся конечных автоматов и регуляриых выражений. Алгоритм 2.6 взят из

работы Хопкрофта [1971]. Результат из упр. 2.3.22 получен Коцарю [1970]. Производная регулярия выражения была определена Бжозовским [1964].

Существует много методов минимизации не полностью определенного конечного автомата (упр. 2.3.27). Этую проблему рассматривали С. Гинзбург [1962] и Пратер [1969]. Частичное решение проблемы, сформулированной в упр. 2.3.32, дано в работе Камеды и Вайнера [1968].

Достаточно полно теория конечных автоматов излагается в книгах Гилла [1962], С. Гинзбурга [1962], Харрисона [1965], Минского [1967], Бута [1967], А. Гинзбурга [1968], Арбика [1969] и Саломаа [1969а]¹⁾.

Томпсон [1968] описывает технику программирования, полезную при построении распознавателя по регуляриому выражению.

2.4. КОНТЕКСТНО-СВОБОДНЫЕ ЯЗЫКИ

Из четырех классов грамматик иерархии Хомского класс контекстно-свободных грамматик наиболее важен с точки зрения приложений к языкам программирования и компиляции. С помощью КС-грамматики можно определить большую часть синтаксической структуры языка программирования. Кроме того, она служит осевой различий схем задания переводов.

В ходе самого процесса компиляции синтаксическую структуру, придаваемую входной программе КС-грамматикой, можно использовать при построении перевода этой программы. Синтаксическую структуру входной цепочки можно определить по последовательности правил, примененных при выводе этой цепочки. Таким образом, на часть компилятора, называемую синтаксическим анализатором, можно смотреть как на устройство, которое пытается выяснить, существует ли в некоторой фиксированной КС-грамматике вывод входной цепочки. Однако это нетривиальная задача — по данной КС-грамматике G и входной цепочке ω выяснить, принадлежит ли ω языку $L(G)$, и если да, то найти вывод цепочки ω в грамматике G . В гл. 4—7 эта проблема будет исследована подробно.

В данном разделе мы построим фундамент, на котором будет основано наше изучение синтаксического анализа. В частности, определим деревья выводов и изучим преобразования, которым можно подвергнуть КС-грамматики, чтобы привести их к более удобному виду.

2.4.1. Деревья выводов

В грамматике может быть несколько выводов, эквивалентных в том смысле, что во всех них применяются одни и те же правила в одних и тех же местах, но в различном порядке. Определить понятие эквивалентности двух выводов для грамматик

¹⁾ См. также книги Глушкова [1962] и Трахтенброта и Барадзина [1970]. — Прим. перев.

произвольного вида сложно (см. упражнения к разд. 2.2), но в случае КС-грамматик можно ввести удобное графическое представление класса эквивалентных выводов, называемое деревом вывода.

Дерево вывода в КС-грамматике $G = (N, \Sigma, P, S)$ — это помеченное упорядоченное дерево, каждая вершина которого помечена символом из множества $N \cup \Sigma \cup \{e\}$. Если внутренняя вершина помечена символом A , а ее прямые потомки — символами X_1, X_2, \dots, X_n , то $A \rightarrow X_1 X_2 \dots X_n$ — правило этой грамматики.

Определение. Помеченное упорядоченное дерево D называется деревом вывода (или деревом разбора) в КС-грамматике $G(A) = (N, \Sigma, P, A)$, если выполнены следующие условия:

(1) Корень дерева D помечен A .

(2) Если D_1, \dots, D_k — поддеревья, над которыми доминируют прямые потомки корня дерева, и корень дерева D_i помечен X_i , то $A \rightarrow X_1 \dots X_k$ — правило из множества P . D_i должно быть деревом вывода в грамматике $G(X_i) = (N, \Sigma, P, X_i)$, если X_i — нетерминал, и D_i состоит из единственной вершины, помеченной X_i , если X_i — терминал.

(3) Если корень дерева имеет единственного потомка, помеченного e , то этот потомок образует дерево, состоящее из единственной вершины, и $A \rightarrow e$ — правило из множества P .

Пример 2.19. На рис. 2.8 изображены деревья выводов в грамматике $G = G(S)$ с правилами $S \rightarrow aSbS \mid bSaS \mid e$. \square

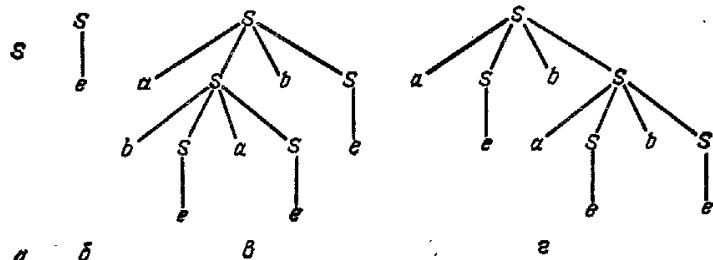


Рис. 2.8. Деревья выводов.

Заметим, что существует естественное упорядочение вершин упорядоченного дерева, при котором прямые потомки вершины упорядочиваются „слева направо“, как определено в разд. 0.5.4. Расширим это упорядочение следующим образом. Допустим, что n — вершина и n_1, \dots, n_k — ее прямые потомки. Тогда если $i < j$, то вершина n_i и все ее потомки считаются расположенными левее вершины n_j и всех ее потомков. Доказательство непроти-

воречивости этого упорядочения оставляем в качестве упражнения. Надо лишь показать, что либо любые две вершины упорядоченного дерева лежат на одном пути, либо одна из них расположена левее другой.

Определение. Кроной дерева вывода назовем цепочку, которая получится, если выписать слева направо метки листьев.

Например, кроны деревьев выводов, показанных на рис. 2.8, таковы: (а) S , (б) e , (в) $abab$ и (г) $abab$.

Покажем теперь, что деревья выводов адекватно представляют выводы в том смысле, что для каждого вывода выводимой цепочки α в КС-грамматике G можно построить дерево вывода в G с кроной α , и обратно. Для этого введем несколько новых понятий. Пусть D — дерево вывода в КС-грамматике $G = (N, \Sigma, P, S)$.

Определение. Сечением дерева D назовем такое множество C вершин дерева D , что

- (1) никакие две вершины из C не лежат на одном пути в D ,
- (2) ни одну вершину дерева D нельзя добавить к C , не нарушив свойства (1).

Пример 2.20. Множество вершин дерева, состоящее из одного корня, является сечением. Листья тоже образуют сечение. Еще

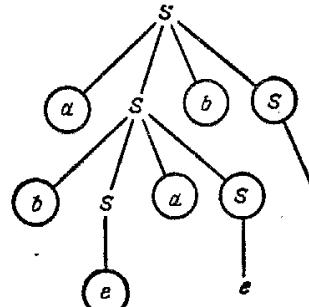


Рис. 2.9. Пример сечения.

один пример сечения — множество на рис. 2.9, состоящее из вершин дерева, обведенных кружками. \square

Определение. Определим крону сечения дерева D как цепочку, которая получается конкатенацией (в порядке слева направо) меток вершин, образующих некоторое сечение.

Например, $abaSbS$ — крана сечения дерева вывода, показанного на рис. 2.9.

Лемма 2.12. Пусть $S = \alpha_0, \alpha_1, \dots, \alpha_n$ — вывод цепочки α_n из S в КС-грамматике $G = (N, \Sigma, P, S)$. Тогда в G можно построить дерево вывода D , для которого α_n — крона, а $\alpha_0, \alpha_1, \dots, \alpha_{n-1}$ — некоторые из крон сечений.

Доказательство. Построим такую последовательность деревьев выводов D_i ($0 \leq i \leq n$), что α_i — крона дерева D_i . Пусть D_0 — дерево, состоящее из единственной вершины, помеченной S .

Допустим, что $\alpha_i = \beta_i A \gamma_i$ и после применения правила $A \rightarrow X_1 X_2 \dots X_k$ к выделенному вхождению A получается $\alpha_{i+1} = \beta_i X_1 X_2 \dots X_k \gamma_i$. Тогда дерево D_{i+1} получается из D_i добавлением к листу, помеченному выделенным вхождением A (он является $(|\beta_i| + 1)$ -м символом кроны дерева D_i), k прямых

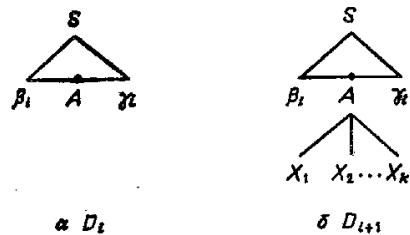


Рис. 2.10. Построение деревьев выводов.

потомков, которые помечаются X_1, X_2, \dots, X_k соответственно. Очевидно, что кроной дерева D_{i+1} будет α_{i+1} . Построение D_{i+1} по D_i показано на рис. 2.10. Дерево D_n будет искомым деревом вывода D . \square

Докажем обращение леммы 2.12, т. е. покажем, что для каждого дерева вывода в грамматике G существует хотя бы один соответствующий вывод.

Лемма 2.13. Пусть D — дерево вывода в КС-грамматике $G = (N, \Sigma, P, S)$ с кроной α . Тогда $S \Rightarrow^* \alpha$.

Доказательство. Пусть $C_0, C_1, C_2, \dots, C_n$ — такая последовательность сечений дерева D , что

- (1) C_0 содержит только корень дерева D ,
- (2) C_{i+1} для $0 \leq i < n$ получается из C_i заменой одной нетерминальной вершины ее прямыми потомками,
- (3) C_n — крона дерева D .

Ясно, что хотя бы одна такая последовательность существует.

Если α_i — крона сечения C_i , то $\alpha_0, \alpha_1, \dots, \alpha_n$ — вывод цепочки α_n из α_0 в G . \square

Среди выводов, которые можно построить по данному дереву, два вывода нас особенно интересуют.

Определение. Если в доказательстве леммы 2.13 сечение C_{i+1} получается из C_i заменой самой левой нетерминальной вершины в C_i ее прямыми потомками, то соответствующий вывод $\alpha_0, \alpha_1, \dots, \alpha_n$ называется *левым* выводом цепочки α_n из α_0 в грамматике G . *Правый* вывод определяется аналогично, надо только в предыдущем предложении читать „самой правой“ вместо „самой левой“. Заметим, что левый (или правый) вывод определяется по дереву вывода однозначно.

Если $S = \alpha_0, \alpha_1, \dots, \alpha_n = w$ — левый вывод терминальной цепочки w , то каждая цепочка α_i ($0 \leq i < n$) имеет вид $x_i A_i \beta_i$,

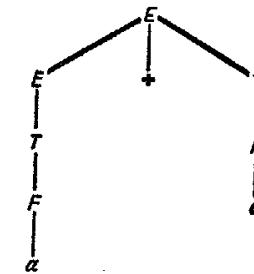


Рис. 2.11. Пример дерева.

где $x_i \in \Sigma^*$, $A_i \in N$ и $\beta_i \in (N \cup \Sigma)^*$. Каждая следующая цепочка α_{i+1} левого вывода получается из предыдущей α_i заменой самого левого нетерминала A_i правой частью некоторого правила. В правом выводе заменяется самый правый нетерминал.

Пример 2.21. Рассмотрим КС-грамматику G_0 с правилами

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

Дерево вывода, показанное на рис. 2.11, служит представлением десяти эквивалентных выводов цепочки $a + a$. Ее левый вывод —

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + F \Rightarrow a + a$$

а правый вывод —

$$E \Rightarrow E + T \Rightarrow E + a \Rightarrow T + a \Rightarrow F + a \Rightarrow a + a \quad \square$$

Определение. Цепочку α будем называть *левовыводимой* (в грамматике G), если существует левый вывод $S = \alpha_0, \alpha_1, \dots, \alpha_n = \alpha$, и писать $S \Rightarrow^*_L \alpha$ (или $S \Rightarrow^*_L \alpha_n$), когда ясно, какая грамматика G имеется в виду). Аналогично α будем называть *правовыводимой*, если существует правый вывод $S = \alpha_0, \alpha_1, \dots, \alpha_n = \alpha$, и писать $S \Rightarrow^*_R \alpha$ (или $S \Rightarrow^*_R \alpha_n$). Один шаг левого вывода обозначим через \Rightarrow_L , а шаг правого вывода — через \Rightarrow_R .

Леммы 2.12 и 2.13 можно объединить в одну теорему:

Теорема 2.11. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. $S \Rightarrow^* \alpha$ тогда и только тогда, когда в G существует дерево вывода с кроной α .

Доказательство. Это непосредственно следует из лемм 2.12 и 2.13. \square

Заметим, что мы остерегались говорить, что если дан вывод $S \Rightarrow^* \alpha$ в КС-грамматике, то можно найти единственное дерево вывода в G с кроной α . Причина этого заключается в том, что есть КС-грамматики, у которых может быть несколько различных деревьев выводов с одной и той же кроной. Такова грамматика из примера 2.19. На рис. 2.8 показаны разные деревья выводов (в) и (г) с одной и той же кроной.

Определение. КС-грамматику G называют *неоднозначной*, если существует хотя бы одна цепочка $w \in L(G)$, которая является кроной двух или более различных деревьев выводов в G . Это равносильно тому, что некоторая цепочка $w \in L(G)$ имеет два или более разных левых (правых) вывода (упр. 2.4.4). В противном случае КС-грамматика G называется *однозначной*.

Неоднозначность будет подробнее рассмотрена в разд. 2.6.5.

2.4.2. Преобразования КС-грамматик

Данную грамматику часто требуется модифицировать так, чтобы порождаемый ею язык приобрел нужную структуру. Рассмотрим, например, язык $L(G_0)$. Этот язык может порождаться грамматикой G с правилами

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

Но грамматика G имеет два недостатка. Прежде всего она неоднозначна из-за наличия в ней правил $E \rightarrow E + E \mid E * E$. Этую неоднозначность можно устраниТЬ, взяв вместо G грамматику G_1 с правилами

$$E \rightarrow E + T \mid E * T \mid T$$

$$T \rightarrow (E) \mid a$$

Другой недостаток грамматики G , которым обладает также и G_1 , заключается в том, что операции $+$ и $*$ имеют один и тот же приоритет. Иначе говоря, структура выражений $a + a * a$ и $a * a + a$, которую придает им грамматика G_1 , подразумевает тот же порядок выполнения операций, что и в выражениях $(a + a) * a$ и $(a * a) + a$ соответственно.

Чтобы получить обычный приоритет операций $+$ и $*$, при котором $*$ предшествует $+$ и выражение $a + a * a$ понимается как $a + (a * a)$, надо перейти к грамматике G_0 .

Общего алгоритмического метода, который придавал бы данному языку произвольную структуру, не существует. Но с помощью ряда преобразований можно видоизменить грамматику, не испортив порождаемого ею языка. В данном разделе в разд. 2.4.8—2.4.5 мы укажем несколько преобразований такого рода.

Начнем с очевидных, но важных преобразований. В некоторых случаях КС-грамматика может содержать бесполезные символы и правила. Например, в грамматике $G = (S, A), \{a, b\}, P, S$, где $P = \{S \rightarrow a, A \rightarrow b\}$, нетерминал A и терминал b не могут появиться ни в какой выводимой цепочке. Таким образом, эти символы не имеют отношения к языку $L(G)$, и их можно устранить из определения грамматики G , не затронув языка $L(G)$.

Определение. Назовем символ $X \in N \cup \Sigma$ бесполезным в КС-грамматике $G = (N, \Sigma, P, S)$, если в ней нет вывода вида $S \Rightarrow^* wXy \Rightarrow^* wxy$, где w, x, y принадлежат Σ^* .

Чтобы установить, бесполезен ли нетерминал A , построим сначала алгоритм, выясняющий, может ли нетерминал порождать какие-нибудь терминальные цепочки, т. е. решающий проблему пустоты множества $\{w \mid A \Rightarrow^* w, w \in \Sigma^*\}$. Из существования такого алгоритма следует разрешимость проблемы пустоты для КС-грамматик.

Алгоритм 2.7. Непуст ли язык $L(G)$?

Вход. КС-грамматика $G = (N, \Sigma, P, S)$.

Выход. „ДА“, если $L(G) \neq \emptyset$, „НЕТ“ в противном случае.

Метод. Строим множества N_0, N_1, \dots рекурсивно:

- (1) Положить $N_0 = \emptyset$ и $i = 1$.
 - (2) Положить $N_i = \{A \mid A \rightarrow \alpha \in P \text{ и } \alpha \in (N_{i-1} \cup \Sigma)^*\} \cup N_{i-1}$.
 - (3) Если $N_i \neq N_{i-1}$, положить $i = i + 1$ и перейти к шагу (2).
- В противном случае положить $N_e = N_i$.
- (4) Если $S \in N_e$, выдать выход „ДА“, в противном случае „НЕТ“. \square

Так как $N_e \subseteq N$, то алгоритм 2.7 должен остановиться самое большое после $n+1$ повторений шага (2), если N содержит n нетерминалов. Докажем корректность алгоритма 2.7. Доказательство простое и послужит в качестве модели для нескольких аналогичных доказательств.

Теорема 2.12. Алгоритм 2.7 говорит „ДА“ тогда и только тогда, когда $S \Rightarrow^* w$ для некоторой цепочки $w \in \Sigma^*$.

Доказательство. Сначала докажем индукцией по i , что (2.4.1) если $A \in N_i$, то $A \Rightarrow^* w$ для некоторой цепочки $w \in \Sigma^*$.

Базис, $i=0$, не нуждается в доказательстве, так как $N_0 \neq \emptyset$. Предположим, что утверждение (2.4.1) истинно для i , и возьмем $A \in N_{i+1}$. Если A принадлежит также и N_i , то шаг индукции тривиален. Если $A \in N_{i+1} - N_i$, то существует правило $A \rightarrow X_1 \dots X_k$, где каждый символ X_i принадлежит либо Σ , либо N_i . Таким образом, для каждого X_j можно найти такую цепочку w_j , что $X_j \Rightarrow^* w_j$: если $X_j \in \Sigma$, то $w_j = X_j$, в противном случае существование w_j следует из (2.4.1). Легко видеть, что

$$A \Rightarrow X_1 \dots X_k \Rightarrow^* w_1 X_2 \dots X_k \Rightarrow^* \dots \Rightarrow^* w_1 \dots w_k$$

Случай $k=0$ (т. е. правило $A \rightarrow e$) не составляет исключения. Шаг индукции окончен.

Определение множеств N_i гарантирует, что если $N_i = N_{i-1}$, то $N_i = N_{i+1} = \dots$. Мы должны показать, что если $A \Rightarrow^* w$ для некоторой цепочки $w \in \Sigma^*$, то $A \in N_e$. В силу сделанного выше замечания все, что нужно показать, это что $A \in N_i$ для некоторого i . Индукцией по n докажем, что

(2.4.2) если $A \Rightarrow^n w$, то $A \in N_i$ для некоторого i .

Базис, $n=1$, тривиален — в этом случае $i=1$. Допустим, что (2.4.2) истинно для n , и пусть $A \Rightarrow^{n+1} w$. Тогда можно написать $A \Rightarrow X_1 \dots X_k \Rightarrow^n w$, где цепочка $w = w_1 \dots w_k$ такова, что $X_j \Rightarrow^{n_j} w_j$ для каждого j и $n_j \leq n$ ¹⁾.

Согласно (2.4.2), если $X_j \in N$, то $X_j \in N_{i_j}$ для некоторого i_j . Если $X_j \in \Sigma$, то $i_j=0$. Пусть $i=1+\max(i_1, \dots, i_k)$. Тогда по определению $A \in N_i$. Индукция завершена. Положив $A=S$ в (2.4.1) и (2.4.2), получим утверждение теоремы. \square

Следствие. Для КС-грамматики G проблема пустоты языка $L(G)$ разрешима. \square

Определение. Символ $X \in N \cup \Sigma$ назовем недостижимым в КС-грамматике $G=(N, \Sigma, P, S)$, если X не появляется ни в одной выводимой цепочке.

Недостижимые символы можно устраниć из КС-грамматики с помощью следующего алгоритма, который легко получить из алгоритма 0.3.

¹⁾ Это „очевидное“ замечание требует тем не менее некоторого осмысливания: представьте себе дерево вывода $A \Rightarrow^{n+1} w$, в нем w_i — крана поддерева с корнем X_j .

Алгоритм 2.8. Устранение недостижимых символов.

Вход. КС-грамматика $G=(N, \Sigma, P, S)$.

Выход. КС-грамматика $G'=(N', \Sigma', P', S)$, у которой

- (i) $L(G') = L(G)$,
- (ii) для всех $X \in N' \cup \Sigma'$ существуют такие цепочки α и β из $(N' \cup \Sigma')^*$, что $S \Rightarrow_G^* \alpha X \beta$.

Метод.

(1) Положить $V_0 = \{S\}$ и $i=1$.

(2) Положить $V_i = \{X \mid$ в P есть $A \rightarrow \alpha X \beta$ и $A \in V_{i-1}\} \cup V_{i-1}$.

(3) Если $V_i \neq V_{i-1}$, положить $i=i+1$ и перейти к шагу (2).

В противном случае пусть

$$N' = V_i \cap N,$$

$$\Sigma' = V_i \cap \Sigma,$$

P' состоит из правил множества P , содержащих только символы из V_i ,

$$G' = (N', \Sigma', P', S). \quad \square$$

Алгоритмы 2.7 и 2.8 очень похожи. Заметим, что шаг (2) алгоритма 2.8 можно повторить только конечное число раз, так как $V_i \subseteq N \cup \Sigma$. Кроме того, прямое доказательство индукцией по i показывает, что $S \Rightarrow_G^* \alpha X \beta$ тогда и только тогда, когда $X \in V_i$ для некоторого i .

Теперь мы готовы устранить из КС-грамматики все бесполезные символы.

Алгоритм 2.9. Устранение бесполезных символов.

Вход. КС-грамматика $G=(N, \Sigma, P, S)$, у которой $L(G) \neq \emptyset$.

Выход. КС-грамматика $G'=(N', \Sigma', P', S)$, у которой $L(G') = L(G)$ и в $N' \cup \Sigma'$ нет бесполезных символов.

Метод.

(1) Применив к G алгоритм 2.7, получить N_e . Положить $G_1 = (N \cap N_e, \Sigma, P_1, S)$, где P_1 состоит из правил множества P , содержащих только символы из $N_e \cup \Sigma$.

(2) Применив к G_1 алгоритм 2.8, получить $G' = (N', \Sigma', P', S)$. \square

На шаге (1) алгоритма 2.9 из G устраняются все нетерминалы, которые не могут порождать терминальных цепочек. Затем на шаге (2) устраняются все недостижимые символы. Каждый символ X результирующей грамматики должен появиться хотя бы в одном выводе вида $S \Rightarrow^* w X y \Rightarrow^* w x y$. Заметим, что если сначала применить алгоритм 2.8, а потом алгоритм 2.7, то не всегда результатом будет грамматика, не содержащая бесполезных символов.

Теорема 2.13. Грамматика G' , которую строит алгоритм 2.9, не содержит бесполезных символов и $L(G) = L(G')$.

Доказательство. Доказательство того, что $L(G') = L(G)$, оставляем в качестве упражнения. Предположим, что $A \in N'$ — бесполезный символ. Тогда по определению бесполезности символа могут представиться два случая:

Случай 1: Вывод $S \Rightarrow_{G'}^* \alpha A \beta$ ни для каких α и β невозможен. В этом случае символ A устраняется на шаге (2) алгоритма 2.9.

Случай 2: $S \Rightarrow_{G'}^* \alpha A \beta$ для некоторых α и β , но вывода $A \Rightarrow_{G'}^* w$ для $w \in \Sigma^*$ не существует. Тогда A не устраивается на шаге (2) и, кроме того, если $A \Rightarrow_G^* yB\delta$, то и B не устраивается на шаге (2). Таким образом, если $A \Rightarrow_G^* w$, то $A \Rightarrow_{G'}^* w$. Тогда можно заключить, что вывода $A \Rightarrow_{G'}^* w$ для $w \in \Sigma^*$ не существует и A устраивается на шаге (1).

Доказательство того, что ни один терминал в G' не может быть бесполезным, проводится аналогично; мы оставляем его в качестве упражнения. \square

Пример 2.22. Рассмотрим грамматику $G = (\{S, A, B\}, \{a, b\}, P, S)$, где P состоит из правил

$$\begin{aligned} S &\rightarrow a | A \\ A &\rightarrow AB \\ B &\rightarrow b \end{aligned}$$

Применим к G алгоритм 2.9. На шаге (1) получим $N_e = \{S, B\}$ и $G_1 = (\{S, B\}, \{a, b\}, \{S \rightarrow a, B \rightarrow b\}, S)$. Применив алгоритм 2.8, получим $V_2 = V_1 = \{S, a\}$. Итак, $G' = (\{S\}, \{a\}, \{S \rightarrow a\}, S)$.

Если применить к G сначала алгоритм 2.8, то окажется, что все символы достижимы, так что грамматика не изменится. Затем применение алгоритма 2.7 дает $N_e = \{S, B\}$, и результирующей будет грамматика G_1 , отличная от G' . \square

Часто бывает удобно устранить из КС-грамматики G e -правила, т. е. правила вида $A \rightarrow e$. Но если $e \in L(G)$, то очевидно, что без правил вида $A \rightarrow e$ не обойтись.

Определение. Назовем КС-грамматику $G = (N, \Sigma, P, S)$ грамматикой без e -правил (или неукорачивающей), если либо

(1) P не содержит e -правил, либо

(2) есть точно одно e -правило $S \rightarrow e$ и S не встречается в правых частях остальных правил из P .

Алгоритм 2.10. Преобразование в грамматику без e -правил.

Вход. КС-грамматика $G = (N, \Sigma, P, S)$.

Выход. Эквивалентная КС-грамматика $G' = (N', \Sigma, P', S')$ без e -правил.

Метод.

(1) Построить $N_e = \{A \mid A \in N \text{ и } A \Rightarrow_G^* e\}$. Это аналогично тому, что было в алгоритмах 2.7 и 2.8, и остается в качестве упражнения.

(2) Построить P' так:

(а) Если $A \rightarrow \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \dots B_k \alpha_k$ принадлежит P , $k \geq 0$ и $B_i \in N_e$ для $1 \leq i \leq k$, но ни один символ в цепочках α_j ($0 \leq j \leq k$) не принадлежит N_e , то включить в P' все правила вида

$$A \rightarrow \alpha_0 X_1 \alpha_1 X_2 \dots \alpha_{k-1} X_k \alpha_k$$

где X_i — либо B_i , либо e , но не включать правило $A \rightarrow e$ (это могло бы произойти в случае, если все α_i равны e).

(б) Если $S \in N_e$, включить в P' правила

$$S' \rightarrow e | S$$

где S' — новый символ, и положить $N' = N \cup \{S'\}$. В противном случае положить $N' = N$ и $S' = S$.

(3) Положить $G' = (N', \Sigma, P', S')$. \square

Пример 2.23. Рассмотрим грамматику из примера 2.19 с правилами

$$S \rightarrow aSbS | bSaS | e$$

Применяя к этой грамматике алгоритм 2.10, получаем грамматику

$$S' \rightarrow S | e$$

$$S \rightarrow aSbS | bSaS | aSb | abS | ab | bSa | baS | ba \quad \square$$

Теорема 2.14. Алгоритм 2.10 дает грамматику без e -правил, эквивалентную входной грамматике.

Доказательство. Непосредственно видно, что алгоритм 2.10 дает грамматику G' без e -правил. Чтобы показать, что $L(G) = L(G')$, достаточно доказать индукцией по длине цепочки w , что

(2.4.3) $A \Rightarrow_G^* w$ тогда и только тогда, когда $w \neq e$ и $A \Rightarrow_{G'}^* w$.

Доказательство этого утверждения оставляем в качестве упражнения. Подставив S вместо A в (2.4.3), видим, что $w \in L(G)$ для $w \neq e$ тогда и только тогда, когда $w \in L(G')$. Очевидно, что $e \in L(G)$ тогда и только тогда, когда $e \in L(G')$. Таким образом, $L(G) = L(G')$. \square

Другое полезное преобразование грамматик — устранение правил вида $A \rightarrow B$, которые мы будем называть цепными.

Алгоритм 2.11. Устранение цепных правил.

Вход. КС-грамматика G без e -правил.

Выход. Эквивалентная КС-грамматика G' без e -правил и без цепных правил.

Метод.

(1) Для каждого $A \in N$ построить $N_A = \{B \mid A \Rightarrow^* B\}$ следующим образом:

- Положить $N_0 = \{A\}$ и $i = 1$.
- Положить $N_i = \{C \mid B \rightarrow C \text{ принадлежит } P \text{ и } B \in N_{i-1}\} \cup N_{i-1}$.
- Если $N_i \neq N_{i-1}$, положить $i = i + 1$ и повторить шаг (б).

В противном случае положить $N_A = N_i$.

(2) Построить P' так: если $B \rightarrow \alpha$ принадлежит P и не является цепным правилом, включить в P' правило $A \rightarrow \alpha$ для всех таких A , что $B \in N_A$.

(3) Положить $G' = (N, \Sigma, P', S)$. \square

Пример 2.24. Применим алгоритм 2.11 к грамматике G_0 с правилами

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

На шаге (1) $N_E = \{E, T, F\}$, $N_T = \{T, F\}$, $N_F = \{F\}$. После шага (2) множество P' станет таким:

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

Теорема 2.15. Грамматика G' , которую строит алгоритм 2.11, не имеет цепных правил и $L(G) = L(G')$.

Доказательство. Непосредственно видно, что алгоритм 2.11 дает грамматику G' без цепных правил. Покажем сначала, что $L(G') \subseteq L(G)$. Пусть $w \in L(G')$. Тогда в грамматике G' существует вывод $S \Rightarrow \alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_n = w$. Если при переходе от α_i к α_{i+1} применяется правило $A \rightarrow \beta$, то существует такой символ $B \in N$ (возможно, $B = A$), что $A \Rightarrow^*_0 B$ и $B \Rightarrow_G \beta$. Таким образом, $A \Rightarrow^*_0 \beta$ и $\alpha_i \Rightarrow^*_G \alpha_{i+1}$. Отсюда следует, что $S \Rightarrow^*_G w$ и $w \in L(G)$, так что $L(G') \subseteq L(G)$.

Теперь покажем, что $L(G) \subseteq L(G')$. Пусть $w \in L(G)$ и $S = \alpha_0 \Rightarrow_l \alpha_1 \Rightarrow_l \dots \Rightarrow_l \alpha_n = w$ — левый вывод цепочки w в грамматике G . Можно найти последовательность индексов i_1, i_2, \dots, i_k , состоящую в точности из тех j , для которых на шаге $\alpha_{j-1} \Rightarrow_l \alpha_j$ применяется не цепное правило. В частности, $i_k = n$, так как вывод терминальной цепочки не может оканчиваться цепным правилом.

Так как мы рассматриваем левый вывод, то последовательные применения цепных правил заменяют символ, занимающий одну и ту же позицию в левовыводимых цепочках, из которых состоит соответствующая часть вывода. Отсюда видно, что $S \Rightarrow^{G'} \alpha_{i_1} \Rightarrow^{G'} \alpha_{i_2} \Rightarrow^{G'} \dots \Rightarrow^{G'} \alpha_{i_k} = w$. Таким образом, $w \in L(G')$ и, значит, $L(G') = L(G)$. \square

Определение. КС-грамматика $G = (N, \Sigma, P, S)$ называется грамматикой без циклов, если в ней нет выводов $A \Rightarrow^+ A$ для $A \in N$. Грамматика G называется приведенной, если она без циклов, без e -правил и без бесполезных символов¹⁾.

Грамматики с e -правилами или циклами иногда труднее анализировать, чем грамматики без e -правил и циклов. Кроме того, в любой практической ситуации бесполезные символы без необходимости увеличивают объем анализатора. Поэтому для некоторых алгоритмов синтаксического анализа, обсуждаемых в этой книге, мы будем требовать, чтобы грамматики, фигурирующие в них, были приведенными. Докажем, что это требование все же позволяет рассматривать все КС-языки.

Теорема 2.16. Если L — КС-язык, то $L = L(G)$ для некоторой приведенной КС-грамматики G .

Доказательство. Применить к КС-грамматике, определяющей язык L , алгоритмы 2.8—2.11. \square

Определение. A -правилом КС-грамматики называется правило вида $A \rightarrow \alpha$ (не путайте A -правило с e -правилом, которое имеет вид $B \rightarrow e$).

Введем еще преобразование, с помощью которого можно удалить из грамматики одно правило вида $A \rightarrow \alpha B \beta$. Чтобы устранить это правило, надо добавить к грамматике новые правила, получающиеся из него заменой нетерминала B правыми частями всех B -правил.

Лемма 2.14. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика и P содержит правило $A \rightarrow \alpha B \beta$, где $B \in N$, а α и β принадлежат $(N \cup \Sigma)^*$. Пусть $B \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_k$ — все B -правила этой грамматики. Пусть $G' = (N, \Sigma, P', S)$, где

$$P' = (P - \{A \rightarrow \alpha B \beta\}) \cup \{A \rightarrow \alpha \gamma_1 \beta \mid \alpha \gamma_2 \beta \mid \dots \mid \alpha \gamma_k \beta\}$$

Тогда $L(G) = L(G')$.

Доказательство. Упражнение. \square

¹⁾ Часто грамматику называют приведенной, просто если в ней нет бесполезных символов. — Прим. перев.

Пример 2.25. Устраним правило $A \rightarrow aAA$ из грамматики G , имеющей два правила $A \rightarrow aAA \mid b$. Применим лемму 2.14, полагая $\alpha = a$, $B = A$ и $\beta = A$, и получим грамматику G' с правилами

$$A \rightarrow aaAAA \mid aba \mid b$$

Деревья выводов цепочки $aabb$ в грамматиках G и G' показаны на рис. 2.12. Заметим, что эффект этого преобразования

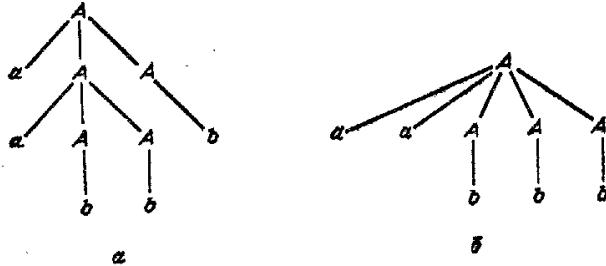


Рис. 2.12. Деревья выводов: а — в грамматике G ; б — в грамматике G' .

состоит в "склеивании" корня дерева на рис. 2.12, а с его вторым прямым потомком. \square

2.4.3. Нормальная форма Хомского

Определение. КС-грамматика $G = (N, \Sigma, P, S)$ называется грамматикой в *нормальной форме Хомского* (или в *бинарной нормальной форме*), если каждое правило из P имеет один из следующих видов:

- (1) $A \rightarrow BC$, где A, B и C принадлежат N ,
- (2) $A \rightarrow a$, где $a \in \Sigma$,
- (3) $S \rightarrow e$, если $e \in L(G)$, причем S не встречается в правых частях правил.

Покажем, что каждый КС-язык порождается грамматикой в нормальной форме Хомского. Этот результат полезен в тех случаях, когда требуется простая форма представления КС-языка.

Алгоритм 2.12. Преобразование к нормальной форме Хомского.

Вход. Приведенная КС-грамматика $G = (N, \Sigma, P, S)$.

Выход. КС-грамматика G' в нормальной форме Хомского, эквивалентная G , т. е. $L(G') = L(G)$.

Метод. Грамматика G' строится по G следующим образом:

- (1) Включить в P' каждое правило из P вида $A \rightarrow a$.
- (2) Включить в P' каждое правило из P вида $A \rightarrow BC$.
- (3) Включить в P' правило $S \rightarrow e$, если оно было в P .

(4) Для каждого правила из P вида $A \rightarrow X_1 \dots X_k$, где $k > 2$, включить в P' правила

$$\begin{aligned} A &\rightarrow X'_1 \langle X_2 \dots X_k \rangle \\ \langle X_2 \dots X_k \rangle &\rightarrow X'_2 \langle X_3 \dots X_k \rangle \\ &\vdots \\ \langle X_{k-2} X_{k-1} X_k \rangle &\rightarrow X'_{k-2} \langle X_{k-1} X_k \rangle \\ \langle X_{k-1} X_k \rangle &\rightarrow X'_{k-1} X'_k \end{aligned}$$

где $X'_i = X_i$, если $X_i \in N$; X'_i — новый нетерминал, если $X_i \in \Sigma$; $\langle X_1 \dots X_k \rangle$ — новый нетерминал.

(5) Для каждого правила из P вида $A \rightarrow X_1 X_2$, где хотя бы один из символов X_1 и X_2 принадлежит Σ , включить в P' правило $A \rightarrow X'_1 X'_2$.

(6) Для каждого нетерминала вида a' , введенного на шагах (4) и (5), включить в P' правило $a' \rightarrow a$. Наконец, пусть N' — это N вместе со всеми новыми нетерминалами, введенными при построении P' . Тогда искомой грамматикой будет $G' = (N', \Sigma, P', S)$ ¹⁾. \square

Теорема 2.17. Пусть L — КС-язык. Тогда $L = L(G')$ для некоторой КС-грамматики G' в нормальной форме Хомского.

Доказательство. По теореме 2.16 L определяется приведенной грамматикой G . Алгоритм 2.12 строит по ней грамматику G' , которая, очевидно, имеет нормальную форму Хомского. Остается показать, что $L(G) = L(G')$. Это доказывается применением леммы 2.14 к каждому правилу грамматики G' , в правую часть которого входит a' , а затем к правилам с нетерминалами вида $\langle X_1 \dots X_j \rangle$. \square

Пример 2.26. Пусть G — приведенная КС-грамматика, определяемая правилами

$$\begin{aligned} S &\rightarrow aAB \mid BA \\ A &\rightarrow BBB \mid a \\ B &\rightarrow AS \mid b \end{aligned}$$

Строим P' алгоритмом 2.12, сохраняя правила $S \rightarrow BA$, $A \rightarrow a$, $B \rightarrow AS$ и $B \rightarrow b$. Заменяем $S \rightarrow aAB$ правилами $S \rightarrow a' \langle AB \rangle$ и $\langle AB \rangle \rightarrow AB$, а $A \rightarrow BBB$ — правилами $A \rightarrow B \langle BB \rangle$ и $\langle BB \rangle \rightarrow BB$. Наконец, добавляем $a' \rightarrow a$. В результате получаем грамматику

¹⁾ Авторы забыли исключить S из правых частей правил. Надо либо модифицировать алгоритм 2.12, либо, проще, ввести новые символы S' и правило $S' \rightarrow S$ и прежде, чем применять алгоритм 2.12, сделать приведение грамматики. — Прим. ред.

$G' = (N', \{a, b\}, P', S)$, где $N' = \{S, A, B, \langle AB \rangle, \langle BB \rangle, a'\}$, а P' состоит из правил

$$\begin{aligned} S &\rightarrow a' \langle AB \rangle | BA \\ A &\rightarrow B \langle BB \rangle | a \\ B &\rightarrow AS | b \\ \langle AB \rangle &\rightarrow AB \\ \langle BB \rangle &\rightarrow BB \\ a' &\rightarrow a^1) \quad \square \end{aligned}$$

2.4.4. Нормальная форма Грейбах

Теперь покажем, что для каждого КС-языка можно найти грамматику, в которой все правые части правил начинаются с терминалов. Построение такой грамматики основано на устранении так называемой левой рекурсии.

Определение. Нетерминал A КС-грамматики $G = (N, \Sigma, P, S)$ называется *рекурсивным*, если $A \Rightarrow^+ \alpha A \beta$ для некоторых α и β . Если $\alpha = e$, то A называется *леворекурсивным*. Аналогично, если $\beta = e$, то A называется *праворекурсивным*. Грамматика, имеющая хотя бы один леворекурсивный нетерминал, называется *леворекурсивной*. Аналогично определяется *праворекурсивная* грамматика. Грамматика, в которой все нетерминалы, кроме, быть может, начального символа, рекурсивные, называется *рекурсивной*.

Некоторые из обсуждаемых далее алгоритмов разбора не могут работать с леворекурсивными грамматиками. Покажем, что каждый КС-язык определяется хотя бы одной не леворекурсивной грамматикой. Начнем с устранения в КС-грамматике непосредственной леворекурсивности.

Лемма 2.15. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, в которой

$$A = A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$$

— все A -правила из P и ни одна из цепочек β_i не начинается с A . Пусть

$$G' = (N \cup \{A'\}, \Sigma, P', S)$$

где A' — новый нетерминал, а P' получается из P заменой A -правил правилами¹⁾

$$\begin{aligned} A &\rightarrow \beta_1 | \beta_2 | \dots | \beta_n | \beta_1 A' | \beta_2 A' | \dots | \beta_n A' \\ A' &\rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_m | \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' \end{aligned}$$

Тогда $L(G') = L(G)$.

¹⁾ Надо сюда устраниТЬ S из правой части правила $B \rightarrow AS$. — Прим. ред.

²⁾ Заметим, что правила $A \rightarrow \beta_i$ содержатся как в исходном, так и в результатеирующем множестве A -правил.

Доказательство. Цепочки, которые можно получить в грамматике G из нетерминала A применением A -правил лишь к самому левому нетерминалу, образуют регулярное множество $(\beta_1 + \beta_2 + \dots + \beta_n)(\alpha_1 + \alpha_2 + \dots + \alpha_m)^*$. Это в точности те цепочки, которые можно получить в G' из A с помощью правых выводов, применив один раз A -правило и несколько раз A' -правила. (В результате весь вывод уже не будет левым.) Все шаги вывода в G , на которых не используются A -правила, можно непосредственно сделать в G' , так как не A -правила в G и G' одни и те же. Отсюда можно заключить, что $L(G') \subseteq L(G)$.

Обратное включение доказывается по существу так же. В G' берется правый вывод и рассматриваются последовательности шагов, состоящие из одного применения A -правила и нескольких применений A' -правил. Таким образом, $L(G) = L(G')$. \square

На рис. 2.13 показано, как действует преобразование, описанное в лемме 2.15, на деревья выводов.

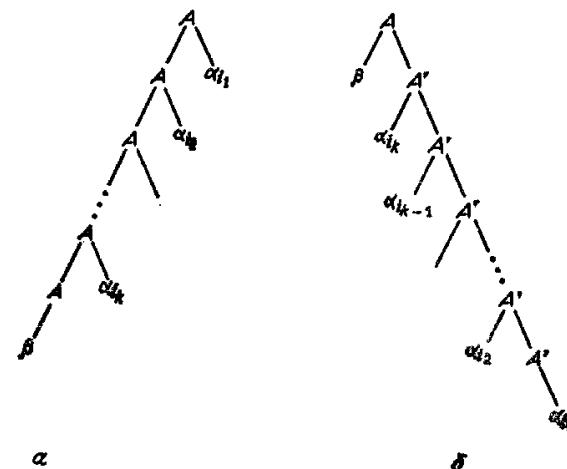


Рис. 2.13. Части деревьев: а — часть дерева в G ; б — соответствующая часть в G' .

Пример 2.27. Пусть G_0 — наша обычная грамматика с правилами

$$\begin{aligned} E &\rightarrow E + T | T \\ T &\rightarrow T * F | F \\ F &\rightarrow (E) | a \end{aligned}$$

Если применить к ней конструкцию леммы 2.15, то получится эквивалентная ей грамматика G' с правилами

$$\begin{aligned} E &\rightarrow T \mid TE' \\ E' &\rightarrow +T \mid +TE' \\ T &\rightarrow F \mid FT' \\ T' &\rightarrow *F \mid *FT' \\ F &\rightarrow (E) \mid a \quad \square \end{aligned}$$

Теперь мы готовы описать алгоритм, устраниющий левую рекурсию из приведенной КС-грамматики. По идеи этот алгоритм подобен алгоритму решения уравнений с регулярными коэффициентами.

Алгоритм 2.13. Устранение левой рекурсии.

Вход. Приведенная КС-грамматика $G = (N, \Sigma, P, S)$.

Выход. Эквивалентная КС-грамматика G' без левой рекурсии.
Метод.

(1) Пусть $N = \{A_1, \dots, A_n\}$. Преобразуем G так, чтобы в правиле $A_i \rightarrow \alpha$ цепочка α начиналась либо с терминала, либо с такого A_j , что $j > i$. С этой целью положим $i = 1$.

(2) Пусть множество A_i -правил — это $A_i \rightarrow A_i \alpha_1 | \dots | A_i \alpha_m | \beta_1 | \dots | \beta_p$, где ни одна из цепочек β_j не начинается с A_k , если $k \leq i$. (Это всегда можно сделать.) Заменим A_i -правила правилами

$$\begin{aligned} A_i &\rightarrow \beta_1 | \dots | \beta_p | \beta_1 A'_i | \dots | \beta_p A'_i \\ A'_i &\rightarrow \alpha_1 | \dots | \alpha_m | \alpha_1 A'_i | \dots | \alpha_m A'_i \end{aligned}$$

где A'_i — новый нетерминал. Правые части всех A_i -правил начинаются теперь с терминала или с A_k для некоторого $k > i$.

(3) Если $i = n$, полученную грамматику G' считать результатом и остановиться. В противном случае положить $i = i + 1$ и $j = 1$.

(4) Заменить каждое правило вида $A_i \rightarrow A_j \alpha$ правилами $A_i \rightarrow \beta_1 \alpha | \dots | \beta_m \alpha$, где $A_j \rightarrow \beta_1 | \dots | \beta_m$ — все A_j -правила. Так как правая часть каждого A_j -правила начинается уже с терминала или с A_k для $k > j$, то и правая часть каждого A_i -правила будет теперь обладать этим свойством.

(5) Если $j = i - 1$, перейти к шагу (2). В противном случае положить $j = j + 1$ и перейти к шагу (4). \square

Теорема 2.18. Каждый КС-язык определяется не леворекурсивной грамматикой.

Доказательство. Пусть G — приведенная грамматика, порождающая КС-язык L . При применении к ней алгоритма 2.13

используются только те преобразования, которые упоминаются в леммах 2.14 и 2.15. Поэтому результирующая грамматика G' порождает L .

Сформулируем два утверждения, которые по существу уже встречались в конце описаний шагов (2) и (4) алгоритма 2.13:

(2.4.4) После выполнения шага (2) для i правая часть каждого A_i -правила начинается с терминала или с такого A_k , что $k > i$.

(2.4.5) После выполнения шага (4) для i и j правая часть каждого A_i -правила начинается с терминала или с такого A_k , что $k > j$.

Докажем, что (2.4.4) истинно для всех $i \leq n$, а (2.4.5) истинно для всех таких пар (i, j) , что $i \leq n$ и $i > j$. Рассмотрим значения параметров i и (i, j) утверждений (2.4.4) и (2.4.5) в том порядке, в каком они встречаются в ходе работы алгоритма 2.13 на шаге (2) и шаге (4) соответственно:

$$\begin{aligned} 1, (2, 1), 2, \dots, i-1, (i, 1), \dots, (i, j), \dots, \\ (i, i-1), i, \dots, (n, n-1), n \end{aligned}$$

и проведем доказательство индукцией по значениям параметров в этом упорядочении.

Базис. Так как на шаге (2) цепочки β_1, \dots, β_p не могут начинаться с A_1 , то для $i = 1$ утверждение (2.4.4) истинно.

Шаг индукции. Предположим, что (2.4.4) и (2.4.5) истинны для значений параметров, предшествующих (i, j) . Так как $j < i$, то по предположению индукции (2.4.4) истинно для j . Поэтому на шаге (4) каждая из цепочек β_1, \dots, β_m начинается с терминала или с такого A_k , что $k > j$. Но после завершения шага (4) цепочки β_1, \dots, β_m становятся префиксами правых частей A_i -правил. Следовательно, (2.4.5) истинно для (i, j) .

Так же просто доказывается, что если (2.4.4) и (2.4.5) истинны для значений параметров, предшествующих i , то (2.4.4) истинно для i . Оставляем это в качестве упражнения.

Из (2.4.4) заключаем, что нетерминалы A_1, \dots, A_n не являются леворекурсивными, так как если $A_i \Rightarrow^+ A_k \alpha$ для некоторого α , то $k > i$. Теперь нужно показать, что нетерминалы A'_i , появляющиеся на шаге (2), не могут быть леворекурсивными. Это непосредственно следует из того, что грамматика G приведенная. На шаге (2) все цепочки $\alpha_1, \dots, \alpha_m$ не пустые, и потому A'_i не может быть первым символом правой части правила. \square

Пример 2.28. Пусть G определяется правилами

$$A \rightarrow BC | a$$

$$B \rightarrow CA | Ab$$

$$C \rightarrow AB | CC | a$$

Положим $A_1 = A$, $A_2 = B$ и $A_3 = C$. После каждого применения шага (2) или (4) алгоритма 2.13 получаются такие грамматики (мы указываем только новые правила):

Шаг (2) для $i=1$: G не меняется

Шаг (4) для $i=2$, $j=1$: $B \rightarrow CA | BCb | ab$

Шаг (2) для $i=2$: $B \rightarrow CA | ab | CAB' | abB'$

$$B' \rightarrow CbB' | Cb$$

Шаг (4) для $i=3$, $j=1$: $C \rightarrow BCB | aB | CC | a$

Шаг (4) для $i=3$, $j=2$:

$$C \rightarrow CACB | abCB | CAB'CB | abB'CB | aB | CC | a$$

Шаг (2) для $i=3$:

$$C \rightarrow abCB | abB'CB | aB | a | abCBC' | abB'CBC' | aBC' | aC'$$

$$C' \rightarrow ACBC' | AB'CBC' | CC' | ACB | AB'CB | C \quad \square$$

Интересный частный случай не леворекурсивной грамматики — грамматика в нормальной форме Грейбах.

Определение. КС-грамматика $G=(N, \Sigma, P, S)$ называется грамматикой в *нормальной форме Грейбах*, если в ней нет e -правил и каждое правило из P , отличное от $S \rightarrow e$, имеет вид $A \rightarrow a\alpha$, где $a \in \Sigma$ и $\alpha \in N^*$.

Если грамматика не леворекурсивна, то на множестве ее нетерминалов можно определить естественный частичный порядок. Этот частичный порядок можно вложить в линейный порядок, полезный при преобразовании грамматики к нормальной форме Грейбах.

Лемма 2.16. Пусть $G=(N, \Sigma, P, S)$ — не леворекурсивная грамматика. Существует такой линейный порядок $<$ на N , что если $A \rightarrow B\alpha$ принадлежит P , то $A < B$.

Доказательство. Пусть R — такое отношение на N , что ARB тогда и только тогда, когда $A \Rightarrow^+ B\alpha$ для некоторого α . Так как грамматика G не леворекурсивна, то R — частичный порядок (транзитивность легко доказывается). Отношение R можно расширить до линейного порядка $<$, обладающего нужным свойством (см. алгоритм 0.1). \square

Алгоритм 2.14. Преобразование к нормальной форме Грейбах.

Вход. Не леворекурсивная приведенная КС-грамматика $G=(N, \Sigma, P, S)$.

Выход. Эквивалентная грамматика G' в нормальной форме Грейбах.

Метод.

(1) Построить с помощью леммы 2.16 такой линейный порядок $<$ на N , что каждое A -правило начинается либо с терминала, либо с такого нетерминала B , что $A < B$. Упорядочить $N=\{A_1, \dots, A_n\}$ так, что $A_1 < A_2 < \dots < A_n$.

(2) Положить $i=n-1$.

(3) Если $i=0$, перейти к шагу (5). В противном случае заменить каждое правило вида $A_i \rightarrow A_j\alpha$, где $j > i$, правилами $A_i \rightarrow \beta_1\alpha | \dots | \beta_m\alpha$, где $A_j \rightarrow \beta_1 | \dots | \beta_m$ — все A_j -правила. Позже мы убедимся, что каждая из цепочек β_1, \dots, β_m начинается терминалом.

(4) Положить $i=i-1$ и вернуться к шагу (3).

(5) Сейчас правая часть каждого правила (кроме, возможно, $S \rightarrow e$) начинается терминалом. В каждом правиле $A \rightarrow aX_1\dots X_k$ заменить $X_i \in \Sigma$ новым нетерминалом X'_i .

(6) Для новых нетерминалов X'_i , введенных на шаге (5), добавить правила $X'_i \rightarrow X_i$. \square

Теорема 2.19. Если L — КС-язык, то $L=L(G)$ для некоторой грамматики G в нормальной форме Грейбах.

Доказательство. Индукцией по $n-i$ (т. е. по i , но в обратном порядке, начиная с $i=n-1$ и кончая $i=1$) можно показать, что после выполнения шага (3) алгоритма 2.14 для i правая часть каждого A_i -правила начинается терминалом. Ключевой момент здесь — использование линейного порядка $<$. После шага (5) грамматика преобразуется к нормальной форме Грейбах и по лемме 2.14 порождаемый ею язык не изменится. \square

Пример 2.29. Рассмотрим грамматику G с правилами

$$E \rightarrow T | TE'$$

$$E' \rightarrow + T | + TE'$$

$$T \rightarrow F | FT'$$

$$T' \rightarrow * F | * FT'$$

$$F \rightarrow (E) | a$$

Упорядочим нетерминалы следующим образом: $E' < E < T' < T < F$.

Правая часть каждого F -правила начинается терминалом, как и должно быть, так как F — наибольший нетерминал в этом упорядочении. Предшествующий ему нетерминал T имеет правила $T \rightarrow F | FT'$, так что, заменив в них F , получаем $T \rightarrow (E) | a(E) T' | aT'$. Переходя к T' , обнаруживаем, что здесь ничего менять не надо. Затем заменяем E -правила правилами

$$E \rightarrow (E) | a | (E) T' | aT' | (E) E' | aE' | (E) T'E' | aT'E'$$

В E' -правилах ничего менять не надо.

На шагах (5) и (6) появляются новый нетерминал $)'$ и правило $)' \rightarrow)$, поэтому все вхождения $)$ в предыдущих правилах надо заменить на $)'$. Таким образом, в результате получится грамматика в нормальной форме Грейбах с правилами

$$\begin{aligned} E &\rightarrow (E)' | a | (E)' T' | aT' | (E)' E' | aE' | (E)' T'E' | aT'E' \\ E' &\rightarrow +T | +TE' \\ T &\rightarrow (E)' | a | (E)' T' | aT' \\ T' &\rightarrow *F | *FT' \\ F &\rightarrow (E)' | a \\)' &\rightarrow) \quad \square \end{aligned}$$

Недостаток описанной техники преобразования грамматики к нормальной форме Грейбах в том, что она дает много новых правил. Чтобы не вводить слишком много новых правил, можно применить другой метод, который излагается в следующем разделе. Однако он может давать больше нетерминалов.

2.4.5. Другой метод преобразования к нормальной форме Грейбах

Изложим другой способ построения грамматики, в которой каждое правило имеет вид $A \rightarrow aa$. Здесь грамматика переписывается только один раз. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, в которой нет e -правил (даже вида $S \rightarrow e$) и цепных правил.

Вместо того чтобы описывать этот метод в терминах правил, воспользуемся системами определяющих уравнений того типа, что был введен в разд. 2.2.2. Например, множество правил

$$\begin{aligned} A &\rightarrow AaB | BB | b \\ B &\rightarrow aA | BAa | Bd | c \end{aligned}$$

можно представить в виде системы уравнений

$$(2.4.6) \quad \begin{aligned} A &= AaB + BB + b \\ B &= aA + BAa + Bd + c \end{aligned}$$

где A и B — неизвестные, представляющие множества.

Определение. Пусть Δ и Σ — два непересекающихся алфавита. Системой определяющих уравнений в алфавитах Σ и Δ назовем систему уравнений вида $A = \alpha_1 + \alpha_2 + \dots + \alpha_k$, где $A \in \Delta$ и $\alpha_i \in (\Delta \cup \Sigma)^*$. Если $k = 0$, то уравнение имеет вид $A = \emptyset$. Для каждого $A \in \Delta$ в системе есть одно уравнение. Решением системы определяющих уравнений назовем такое отображение f множества Δ в $\mathcal{P}(\Sigma^*)$, что если подставить $f(A)$ в каждое уравнение вместо каждого $A \in \Delta$, уравнения станут равенствами. Решение f назовем наименьшей неподвижной точкой, если $f(A) \subseteq f(A)$ для любого решения g и любого $A \in \Delta$.

Чтобы получить КС-грамматику, соответствующую системе определяющих уравнений, надо для каждого уравнения $A = \alpha_1 + \dots + \alpha_k$ построить правила $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$. Нетерминалами будут символы алфавита Δ . Очевидно, что это соответствие взаимно однозначно. Приведем несколько результатов о системах определяющих уравнений, обобщающих результаты о стандартных системах уравнений с регулярными коэффициентами (частном случае систем определяющих уравнений). Доказательства оставим в качестве упражнений.

Лемма 2.17. Наименьшая неподвижная точка системы определяющих уравнений в алфавитах Δ и Σ единственна и имеет вид $f(A) = \{w \mid A \Rightarrow_G^* w \text{ и } w \in \Sigma^*\}$, где G — соответствующая КС-грамматика.

Доказательство. Упражнение. \square

Мы будем пользоваться матричным представлением систем определяющих уравнений. Допустим, что $\Delta = \{A_1, A_2, \dots, A_n\}$. Матричное уравнение

$$A = AR + B$$

представляет n уравнений. Здесь A — вектор-строка $[A_1, A_2, \dots, A_n]$, R — матрица порядка n , элементами которой служат регулярные выражения, и B — вектор-строка, состоящая из n регулярных выражений.

В качестве «скалярного» умножения возьмем конкатенацию, а в качестве сложения — операцию $+$ (т. е. объединение). Сложение и умножение векторов и матриц определяются, как обычно. Элементом матрицы R , стоящим в i -й строке и j -м столбце, будет регулярное выражение $\alpha_1 + \dots + \alpha_k$, если $A_i \alpha_1, \dots, A_i \alpha_k$ — все члены уравнения для A_j , первым символом которых является A_i . В качестве j -й компоненты вектора B возьмем сумму тех членов уравнения для A_j , которые начинаются символом из множества Σ . Таким образом, B_j и R_{ij} — такие выражения, что уравнение для A_j (в КС-грамматике ему соответствует множе-

ство A_j -правил) можно записать в виде

$$A_j = A_1 R_{1j} + A_2 R_{2j} + \dots + A_i R_{ij} + \dots + A_n R_{nj} + B_j$$

где B_j —сумма выражений, начинающихся терминалами.

Система определяющих уравнений (2.4.6) примет вид

$$(2.4.7) \quad [A, B] = [A, B] \begin{bmatrix} aB & \emptyset \\ B & Aa + d \end{bmatrix} + [b, aA + c]$$

Теперь для матричного уравнения $\mathbf{A} = \mathbf{AR} + \mathbf{B}$ найдем такую эквивалентную систему определяющих уравнений, что все правые части соответствующих ей правил начинаются терминальными символами.

Этот переход основан на следующей лемме:

Лемма 2.18. Пусть $\mathbf{A} = \mathbf{AR} + \mathbf{B}$ —система определяющих уравнений. Тогда ее наименьшей неподвижной точкой будет $\mathbf{A} = \mathbf{BR}^*$, где $\mathbf{R}^* = \mathbf{I} + \mathbf{R} + \mathbf{R}^2 + \mathbf{R}^3 + \dots$, \mathbf{I} —единичная матрица (е на диагонали и \emptyset —в остальных местах), $\mathbf{R}^2 = \mathbf{RR}$, $\mathbf{R}^3 = \mathbf{RRR}$ и т. д.

Доказательство. Упражнение. \square

Если положить $\mathbf{R}^+ = \mathbf{RR}^*$, то наименьшую неподвижную точку матричного уравнения $\mathbf{A} = \mathbf{AR} + \mathbf{B}$ можно записать в виде $\mathbf{A} = \mathbf{B}(\mathbf{R}^+ + \mathbf{I}) = \mathbf{BR}^+ + \mathbf{BI} = \mathbf{BR}^+ + \mathbf{B}$. К сожалению, для этой системы нельзя найти соответствующую грамматику: она не является системой определяющих уравнений, так как элементы матрицы \mathbf{R}^+ могут быть бесконечными суммами членов исходных уравнений. Однако \mathbf{R}^+ можно заменить новой матрицей “неизвестных” \mathbf{Q} , каждый элемент q_{ij} которой, расположенный в i -й строке и j -м столбце,—это новый символ.

Тогда, заметив, что $\mathbf{R}^+ = \mathbf{RR}^+ + \mathbf{R}$, можно получить систему определяющих уравнений $\mathbf{Q} = \mathbf{RQ} + \mathbf{R}$ с неизвестными q_{ij} . Отметим, что если \mathbf{Q} и \mathbf{R} —матрицы порядка n , то система состоит из n^2 уравнений.

Лемма 2.19. Пусть $\mathbf{A} = \mathbf{AR} + \mathbf{B}$ —система определяющих уравнений в алфавитах Δ и Σ . Пусть \mathbf{Q} —матрица того же порядка, что и \mathbf{R} , причем все ее элементы—различные новые символы. Тогда система определяющих уравнений, состоящая из $\mathbf{A} = \mathbf{BQ} + \mathbf{B}$ и $\mathbf{Q} = \mathbf{RQ} + \mathbf{R}$, имеет наименьшую неподвижную точку, которая совпадает на Δ с наименьшей неподвижной точкой системы $\mathbf{A} = \mathbf{AR} + \mathbf{B}$.

Доказательство. Упражнение. \square

Дадим другой алгоритм преобразования приведенной грамматики к нормальной форме Грейбах.

Алгоритм 2.15. Преобразование к нормальной форме Грейбах.

Вход. Приведенная грамматика $G = (N, \Sigma, P, S)$ без правила вида $S \rightarrow e$.

Выход. Эквивалентная грамматика $G' = (N', \Sigma, P', S)$ в нормальной форме Грейбах.

Метод.

(1) По грамматике G построить систему определяющих уравнений $\mathbf{A} = \mathbf{AR} + \mathbf{B}$ в алфавитах N и Σ .

(2) Пусть \mathbf{Q} —матрица порядка n , состоящая из новых символов, и $\# N = n$. Построить новую систему определяющих уравнений $\mathbf{A} = \mathbf{BQ} + \mathbf{B}$, $\mathbf{Q} = \mathbf{RQ} + \mathbf{R}$ и соответствующую грамматику G_1 . Так как в векторе \mathbf{B} каждая компонента, отличная от \emptyset , начинается терминалом, а такие компоненты должны быть в силу приведенности грамматики G , то для $A \in N$ все A -правила грамматики G_1 будут начинаться терминалами.

(3) Так как G —приведенная грамматика, то e не является элементом матрицы \mathbf{R} . Поэтому для каждого элемента q матрицы \mathbf{Q} все соответствующие q -правила грамматики G_1 начинаются символами из $N \cup \Sigma$. В тех правых частях этих правил, которые начинаются нетерминалом, заменить этот нетерминал A правыми частями всех A -правил. В результате получится грамматика, у которой правые части всех правил начинаются терминалом.

(4) Если в правой части некоторого правила терминал a встречается не на первом месте, заменить его новым нетерминалом a' и добавить правило $a' \rightarrow a$. Результирующую грамматику обозначить через G' . \square

Теорема 2.20. Алгоритм 2.15 дает грамматику G' в нормальной форме Грейбах и $L(G) = L(G')$.

Доказательство. Так как грамматика G приведенная и не содержит $S \rightarrow e$, то G' —грамматика в нормальной форме Грейбах, поскольку e не является компонентой вектора \mathbf{B} и матрицы \mathbf{R} . Из лемм 2.14, 2.17 и 2.19 следует, что $L(G') = L(G)$. \square

Пример 2.30. Рассмотрим грамматику, соответствующую системе определяющих уравнений (2.4.7):

$$\begin{aligned} A &\rightarrow AaB \mid BB \mid b \\ B &\rightarrow aA \mid BAa \mid Bd \mid c \end{aligned}$$

Перепишем систему (2.4.7), согласно шагу (2) алгоритма 2.15, в виде

$$(2.4.8) \quad [A, B] = [b, aA + c] \begin{bmatrix} W & X \\ Y & Z \end{bmatrix} + [b, aA + c]$$

Затем добавим систему

$$(2.4.9) \quad \begin{bmatrix} W & X \\ Y & Z \end{bmatrix} = \begin{bmatrix} aB & \emptyset \\ B & Aa+d \end{bmatrix} \begin{bmatrix} W & X \\ Y & Z \end{bmatrix} + \begin{bmatrix} aB & \emptyset \\ B & Aa+d \end{bmatrix}$$

Системам (2.4.8) и (2.4.9) соответствует грамматика

$$\begin{aligned} A &\rightarrow bW | aAY | cY | b \\ B &\rightarrow bX | aAZ | cZ | aA | c \\ W &\rightarrow aBW | aB \\ X &\rightarrow aBX \\ Y &\rightarrow BW | AaY | dY | B \\ Z &\rightarrow BX | AaZ | dZ | Aa | d \end{aligned}$$

Заметим, что X — бесполезный символ. На шаге (3) в правилах $Y \rightarrow BW | AaY | B$ и $Z \rightarrow BX | AaZ | Aa$ нетерминалы A и B заменяются соответствующими правыми частями правил. Так как это преобразование и преобразование на шаге (4) уже знакомы читателю, мы их опускаем. \square

УПРАЖНЕНИЯ

2.4.1. Пусть G определяется правилами

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow Aa | bB \\ B &\rightarrow a | Sb \end{aligned}$$

Постройте деревья выводов следующих выводимых цепочек:

- (а) $baabaab$,
- (б) $bBABb$,
- (в) $baSb$.

2.4.2. Постройте левый и правый выводы цепочки $baabaab$ в грамматике из упр. 2.4.1.

2.4.3. Постройте все сечения дерева, изображенного на рис. 2.14.

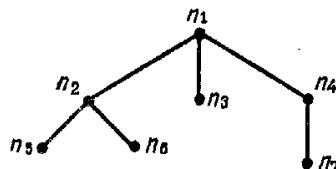


Рис. 2.14. Непомеченное дерево вывода.

2.4.4. Покажите, что следующие утверждения о КС-грамматике G и цепочке w эквивалентны:

- (а) w — крона двух различных деревьев выводов в G ,
- (б) w имеет два различных левых вывода в G ,
- (в) w имеет два различных правых вывода в G .

****2.4.5.** Каково наибольшее число выводов, которые представляются одним и тем же деревом с n вершинами?

2.4.6. Преобразуйте грамматику

$$\begin{aligned} S &\rightarrow A | B \\ A &\rightarrow aB | bS | b \\ B &\rightarrow AB | Ba \\ B &\rightarrow AS | b \end{aligned}$$

в эквивалентную КС-грамматику, не содержащую бесполезных символов.

2.4.7. Докажите, что алгоритм 2.8 правильно устраниет недостижимые символы.

2.4.8. Дополните доказательство теоремы 2.13.

2.4.9. Оцените временную и емкостную сложности алгоритма 2.8. В качестве вычислительной модели возьмите машину с произвольным доступом к памяти.

2.4.10. Постройте алгоритм, вычисляющий для КС-грамматики $G = (N, \Sigma, P, S)$ множество $\{A \mid A \Rightarrow^* e, A \in N\}$. Насколько быстр Ваш алгоритм?

2.4.11. Найдите грамматику без e -правил, эквивалентную грамматике

$$\begin{aligned} S &\rightarrow ABC \\ A &\rightarrow BB | e \\ B &\rightarrow CC | a \\ C &\rightarrow AA | b \end{aligned}$$

2.4.12. Дополните доказательство теоремы 2.14.

2.4.13. Найдите приведенную грамматику, эквивалентную грамматике

$$\begin{aligned} S &\rightarrow A | B \\ A &\rightarrow C | D \\ B &\rightarrow D | E \\ C &\rightarrow S | a | e \\ D &\rightarrow S | b \\ E &\rightarrow S | c | e \end{aligned}$$

2.4.14. Докажите теорему 2.16.

2.4.15. Докажите лемму 2.14.

2.4.16. Преобразуйте следующие грамматики к нормальной форме Хомского:

$$(a) S \rightarrow 0S1 | 01$$

$$(b) S \rightarrow aB | bA$$

$$A \rightarrow aS | bAA | a$$

$$B \rightarrow bS | aBB | b$$

2.4.17. Если $G = (N, \Sigma, P, S)$ — грамматика в нормальной форме Хомского, $S \Rightarrow_G^k w$, $|w| = n$ и $w \in \Sigma^*$, то каково k ?

2.4.18. Дайте подробное доказательство теоремы 2.17.

2.4.19. Преобразуйте к нормальной форме Грейбах грамматику

$$S \rightarrow Ba | Ab$$

$$A \rightarrow Sa | AAb | a$$

$$B \rightarrow Sb | BBa | b$$

используя

(а) алгоритм 2.14,

(б) алгоритм 2.15.

***2.4.20.** Постройте быстрый алгоритм, проверяющий, леворекурсивна ли КС-грамматика G .

2.4.21. Постройте быстрый алгоритм, устраняющий в КС-грамматике правую рекурсию.

2.4.22. Дополните доказательство леммы 2.15.

***2.4.23.** Докажите леммы 2.17—2.19.

2.4.24. Завершите преобразования примера 2.30 и получите приведенную грамматику в нормальной форме Грейбах.

2.4.25. Сравните относительные достоинства алгоритмов 2.14 и 2.15, особенно с точки зрения объема результирующей грамматики.

***2.4.26.** Покажите, что каждый КС-язык, не содержащий пустой цепочки e , определяется грамматикой, все правила которой имеют вид $A \rightarrow aBC$, $A \rightarrow aB$ или $A \rightarrow a$.

Определение. КС-грамматика называется *операторной*, если в правых частях ее правил нет двух стоящих рядом нетерминалов.

***2.4.27.** Покажите, что каждый КС-язык определяется операторной грамматикой. **Указание:** Начните с грамматики в нормальной форме Грейбах.

***2.4.28.** Покажите, что каждый КС-язык порождается грамматикой, все правила которой имеют вид $A \rightarrow aBbC$, $A \rightarrow aBb$,

$A \rightarrow aB$ или $A \rightarrow a$. Если $e \in L(G)$, то допускается еще правило $S \rightarrow e$.

****2.4.29.** Рассмотрим грамматику с двумя правилами $S \rightarrow SS | a$. Покажите, что число X_n различных левых выводов цепочки a^n находится из соотношения

$$X_n = \sum_{\substack{i+j=n \\ i \neq 0 \\ j \neq 0}} X_i X_j$$

где $X_1 = 1$. Покажите, что

$$X_{n+1} = \frac{1}{n+1} \binom{2n}{n}$$

(числа такого вида называются числами Каталана).

***2.4.30.** Покажите, что КС-язык L , не содержащий цепочек, длина которых меньше 2, порождается грамматикой с правилами вида $A \rightarrow aab$.

2.4.31. Покажите, что каждый КС-язык определяется грамматикой, удовлетворяющей условию: если $X_1 X_2 \dots X_k$ — правая часть правила, то все X_1, \dots, X_k различны.

Определение. КС-грамматика $G = (N, \Sigma, P, S)$ называется *линейной*, если каждое ее правило имеет вид $A \rightarrow wBx$ или $A \rightarrow w$, где $B \in N$, а w и x принадлежат Σ^* .

2.4.32. Покажите, что каждый линейный язык, не содержащий пустой цепочки, определяется грамматикой, все правила которой имеют вид $A \rightarrow aB$, $A \rightarrow Ba$ или $A \rightarrow a$.

***2.4.33.** Покажите, что каждый КС-язык определяется такой грамматикой $G = (N, \Sigma, P, S)$, что если $A \in N - \{S\}$, то язык $\{w \mid A \Rightarrow^* w \text{ и } w \in \Sigma^*\}$ бесконечен.

2.4.34. Покажите, что каждый КС-язык определяется рекурсивной грамматикой. **Указание:** Используйте лемму 2.14 и упр. 2.4.33.

***2.4.35.** Назовем КС-грамматику $G = (N, \Sigma, P, S)$ *квазилинейной*, если для каждого правила $A \rightarrow X_1 \dots X_k$ существует не более одного символа X_i , порождающего бесконечное множество терминальных цепочек. Покажите, что каждая квазилинейная грамматика порождает линейный язык.

Определение. Графом КС-грамматики $G = (N, \Sigma, P, S)$ назовем такой ориентированный неупорядоченный граф $(N \cup \Sigma \cup \{e\}, R)$, что ARX тогда и только тогда, когда $A \rightarrow \alpha X \beta$ — правило грамматики для некоторых α и β .

2.4.36. Покажите, что если грамматика не содержит бесполезных символов, то все вершины ее графа достижимы из S . Верно ли обратное утверждение?

2.4.37. Пусть T — преобразование КС-грамматик, определенное в лемме 2.14, т. е. T отображает G в G' , где G и G' — грамматики из леммы 2.14. Покажите, что алгоритмы 2.10 и 2.11 можно реализовать повторными применениями преобразования T .

Упражнения на программирование

2.4.38. Постройте программу, устраниющую в КС-грамматике бесполезные символы.

2.4.39. Постройте программу, которая преобразует КС-грамматику в эквивалентную приведенную КС-грамматику.

2.4.40. Постройте программу, устраниющую в КС-грамматике левую рекурсию.

2.4.41. Постройте программу, которая решает, является ли данное дерево деревом вывода в КС-грамматике.

Замечания по литературе

Дерево вывода имеет немало других названий, среди которых: дерево по-рождения, диаграмма разбора, дерево разбора, дерево анализа, синтаксиче- ское дерево, дерево составляющих. Аналогичное понятие хорошо известно в лингвистике. Понятие левого вывода появилось в работе Эви [1963].

Многие алгоритмы этой главы были известны еще в начале 1960-х годов, хотя часть из них появилась в литературе значительно позже. Теорема 2.17 (нормальная форма Хомского) впервые доказана Хомским [1959а]. Теорема 2.18 (нормальная форма Грейбаха) впервые доказана Грейбахом [1965]. Алгоритм 2.15 и результат, сформулированный в упр. 2.4.30, принадлежат Розенкранцу [1967]. Алгоритм 2.14 приписывают М. Паулу.

Представление КС-языков с помощью систем уравнений использовалось в работах Хомского [1963], Хомского и Шютценберже [1963], Гиззбурга и Райса [1962].

Операторные грамматики впервые рассмотрены Флойдом [1963]. Нормаль- ные формы, приведенные в упр. 2.4.26—2.4.28, были получены Грейбахом [1965].

2.5. АВТОМАТЫ С МАГАЗИННОЙ ПАМЯТЬЮ

Теперь мы введем понятие автомата с магазинной памятью — тип распознавателя, представляющий собой естественную модель синтаксических анализаторов контексто-свободных языков. Автомат с магазинной памятью — это односторонний недетерминированный распознаватель, в потенциально бесконечной памяти которого элементы информации хранятся и используются так же, как патроны в магазине автоматического оружия, т. е. в каж- дый момент доступен только верхний элемент магазина. Распо- знаватель этого типа изображен на рис. 2.15.

Мы докажем один фундаментальный результат, относящийся к автоматам с магазинной памятью, а именно: язык контексто-свободен тогда и только тогда, когда он допускается недетерми-нированным автоматом с магазинной памятью. Рассмотрим также

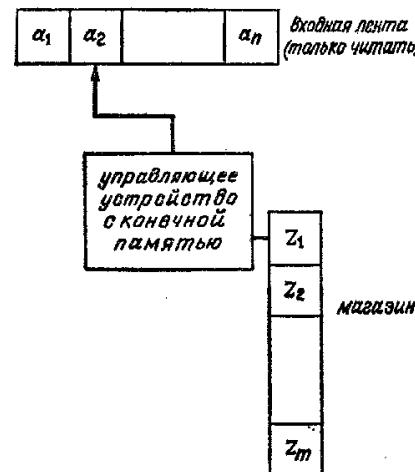


Рис. 2.15. Автомат с магазинной памятью.

один подкласс КС-языков, особенно важный с точки зрения ана- лизируемости языков. Он состоит из детерминированных КС-язы- ков, т. е. языков, распознаваемых детерминированными автома- тами с магазинной памятью.

2.5.1. Основное определение

Будем представлять магазин (или магазинный список, или магазинную ленту) в виде цепочки символов, причем верхним элемен- том магазина будем считать самый левый или самый пра- вый символ цепочки в зависимости от того, что удобнее в данной ситуации. Пока будем считать верхним самый левый символ цепочки, представляющей магазинный список.

Определение. Автомат с магазинной памятью (сокращенно МП-автомат) — это семерка

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

где

- (1) Q — конечное множество символов состояний, представ- ляющих всевозможные состояния управляющего устройства;
- (2) Σ — конечный входной алфавит;
- (3) Γ — конечный алфавит магазинных символов;

- (4) δ — отображение множества $Q \times (\Sigma \cup \{e\}) \times \Gamma$ в множество конечных подмножеств множества $Q \times \Gamma^*$;
- (5) $q_0 \in Q$ — начальное состояние управляющего устройства;
- (6) $Z_0 \in \Gamma$ — символ, находящийся в магазине в начальный момент (начальный символ);
- (7) $F \subseteq Q$ — множество заключительных состояний.

Конфигурацией МП-автомата P называется тройка $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$, где

- (1) q — текущее состояние управляющего устройства;
- (2) w — неиспользованная часть входной цепочки; первый символ цепочки w находится под входной головкой; если $w = e$, то считается, что вся входная лента прочитана;
- (3) α — содержимое магазина; самый левый символ цепочки α считается верхним символом магазина; если $\alpha = e$, то магазин считается пустым.

Такт работы МП-автомата P будем представлять в виде бинарного отношения \vdash_P (или \vdash , когда P подразумевается), определенного на конфигурациях. Будем писать

$$(2.5.1) \quad (q, aw, Za) \vdash (q', w, \gamma\alpha)$$

если множество $\delta(q, a, Z)$ содержит (q', γ) , где $q, q' \in Q$, $a \in \Sigma \cup \{e\}$, $w \in \Sigma^*$, $Z \in \Gamma$ и $\alpha, \gamma \in \Gamma^*$.

Если $a \neq e$, то (2.5.1) говорит о том, что МП-автомат P , находясь в состоянии q и имея a в качестве текущего входного символа, расположенного под входной головкой, а Z — в качестве верхнего символа магазина, может перейти в новое состояние q' , сдвинуть входную головку на одну ячейку вправо и заменить верхний символ магазина цепочкой γ магазинных символов. Если $\gamma = e$, то верхний символ удаляется из магазина, и тем самым магазинный список сокращается.

Если $a = e$, будем называть этот такт *e-тактом*. В *e-такте* текущий входной символ не принимается во внимание и входная головка не сдвигается. Однако состояние управляющего устройства и содержимое памяти могут измениться. Заметим, что *e-такт* может происходить и тогда, когда вся входная цепочка прочитана.

Следующий такт невозможен, если магазин пуст.

Можно обычным образом определить отношения \vdash^i для $i \geq 0$, \vdash^* и \vdash^+ , а именно: \vdash^* и \vdash^+ — это соответственно рефлексивно-транзитивное и транзитивное замыкания отношения \vdash .

Начальной конфигурацией МП-автомата P называется конфигурация вида (q_0, w, Z_0) , где $w \in \Sigma^*$, т. е. управляющее устройство находится в начальном состоянии, входная лента содержит цепочку, которую нужно распознать, а в магазине есть только

начальный символ Z_0 . Заключительная конфигурация — это конфигурация вида (q, e, α) , где $q \in F$ и $\alpha \in \Gamma^*$.

Говорят, что цепочка w допускается МП-автоматом P , если $(q_0, w, Z_0) \vdash^* (q, e, \alpha)$ для некоторых $q \in F$ и $\alpha \in \Gamma^*$. Языком, определяемым (или допускаемым) автоматом P (обозначается $L(P)$), называют множество цепочек, допускаемых автоматом P .

Пример 2.31. Построим МП-автомат, определяющий язык $L = \{0^n 1^n \mid n \geq 0\}$. Пусть $P = (Q, q_0, q_1, q_2, \{0, 1\}, \{Z, 0\}, \delta, q_0, Z, \{q_0\})$, где

$$\begin{aligned} \delta(q_0, 0, Z) &= \{(q_1, 0Z)\} \\ \delta(q_1, 0, 0) &= \{(q_1, 00)\} \\ \delta(q_1, 1, 0) &= \{(q_2, e)\} \\ \delta(q_2, 1, 0) &= \{(q_2, e)\} \\ \delta(q_2, e, Z) &= \{(q_0, e)\} \end{aligned}$$

Работа МП-автомата P состоит в том, что он копирует в магазине начальную часть входной цепочки, состоящую из нулей, а затем устраняет из магазина по одному нулю на каждую единицу, которую он видит на входе. Кроме того, переходы состояний гарантируют, что все нули предшествуют единицам. Например, для входной цепочки 0011 автомат P проделает такую последовательность тактов:

$$\begin{aligned} (q_0, 0011, Z) \vdash (q_1, 011, 0Z) \\ \vdash (q_1, 11, 00Z) \\ \vdash (q_2, 1, 0Z) \\ \vdash (q_2, e, Z) \\ \vdash (q_0, e, e) \end{aligned}$$

Вообще можно показать, что

$$\begin{aligned} (q_0, 0, Z) \vdash (q_1, e, 0Z) \\ (q_1, 0^i, 0Z) \vdash^i (q_1, e, 0^{i+1}Z) \\ (q_1, 1, 0^{i+1}Z) \vdash (q_2, e, 0^iZ) \\ (q_2, 1^i, 0^iZ) \vdash^i (q_2, e, Z) \\ (q_2, e, Z) \vdash (q_0, e, e) \end{aligned}$$

Объединяя все это, получаем для $n \geq 1$

$$(q_0, 0^n 1^n, Z) \vdash^{2n+1} (q_0, e, e)$$

и

$$(q_0, e, Z) \vdash^0 (q_0, e, Z)$$

Таким образом, $L \subseteq L(P)$.

Покажем, что $L \supseteq L(P)$, т. е. что P допускает только цепочки вида $0^n 1^n$. Эта часть доказательства труднее. Обычно легче доказать, что такие-то цепочки распознаватель допускает, и так

же, как для грамматик, гораздо труднее доказать, что он допускает цепочки только определенного вида.

В данном случае заметим, что если P допускает непустую цепочку, то он должен пройти через состояния q_0, q_1, q_2, q_3 именно в этом порядке.

Далее, если $(q_0, w, Z) \vdash^i (q_1, e, \alpha)$ для $i \geq 1$, то $w = 0^i$ и $\alpha = 0^i Z$. Аналогично, если $(q_2, w, \alpha) \vdash^i (q_3, e, \beta)$, то $w = 1^i$ и $\alpha = 0^i \beta$. К тому же $(q_1, w, \alpha) \vdash (q_2, e, \beta)$ только тогда, когда $w = 1$ и $\alpha = 0\beta$, а $(q_2, w, Z) \vdash^* (q_3, e, e)$ только тогда, когда $w = e$. Таким образом, если $(q_0, w, Z) \vdash^i (q_3, e, \alpha)$ для некоторого $i \geq 0$, то либо $w = e$ и $i = 0$, либо $w = 0^n 1^n$, $i = 2n + 1$, и $\alpha = e$. Следовательно, $L \subseteq L(P)$. \square

Подчеркнем еще раз, что МП-автомат, как мы его определили, может продолжать работу (делать e -такты), даже если он прочел уже всю входную цепочку. Но он не может сделать следующий такт, если его магазин пуст.

Пример 2.32. Построим МП-автомат, допускающий язык

$$L = \{ww^R \mid w \in \{a, b\}^+\}$$

Пусть $P = (\{q_0, q_1, q_2\}, \{a, b\}, \{Z, a, b\}, \delta, q_0, Z, \{q_2\})$, где

- (1) $\delta(q_0, a, Z) = \{(q_0, aZ)\}$
- (2) $\delta(q_0, b, Z) = \{(q_0, bZ)\}$
- (3) $\delta(q_0, a, a) = \{(q_0, aa), (q_1, e)\}$
- (4) $\delta(q_0, a, b) = \{(q_0, ab)\}$
- (5) $\delta(q_0, b, a) = \{(q_0, ba)\}$
- (6) $\delta(q_0, b, b) = \{(q_0, bb), (q_1, e)\}$
- (7) $\delta(q_1, a, a) = \{(q_1, e)\}$
- (8) $\delta(q_1, b, b) = \{(q_1, e)\}$
- (9) $\delta(q_1, e, Z) = \{(q_2, e)\}$

МП-автомат P вначале копирует в магазине какую-то часть входной цепочки по правилам (1), (2), (4) и (5) и первым альтернативам правил (3) и (6). Однако P — недетерминированный распознаватель. В любой момент, когда ему захочется, лишь бы текущий входной символ совпадал с верхним символом магазина, он может перейти в состояние q_1 и начать сравнивать цепочку в магазине с оставшейся частью входной цепочки. Этот выбор осуществляют вторые альтернативы правил (3) и (6), а по правилам (7) и (8) происходит сравнение. Если P обнаруживает несовпадение очередных символов, то этот экземпляр МП-автомата P „умирает“, т. е. перестает работать. Однако, так как автомат P — недетерминированный, то разные его экземпляры могут делать все возможные для него такты. Если какой-то выбор тактов приводит к тому, что Z снова оказывается верх-

ним (и единственным) символом магазина, то по правилу (9) P стирает Z и попадает в состояние q_2 . Итак, P допускает цепочку тогда и только тогда, когда все сравнения обнаружили совпадение символов.

Например, для входной цепочки $abba$ автомат P может среди прочих сделать следующие последовательности тактов:

- (1) $(q_0, abba, Z) \vdash (q_0, bba, aZ)$
 $\vdash (q_0, ba, baZ)$
 $\vdash (q_0, a, bbaZ)$
 $\vdash (q_0, e, abbaZ)$
- (2) $(q_0, abba, Z) \vdash (q_0, bba, aZ)$
 $\vdash (q_0, ba, baZ)$
 $\vdash (q_1, a, aZ)$
 $\vdash (q_1, e, Z)$
 $\vdash (q_2, e, e)$

Так как последовательность (2) оканчивается заключительным состоянием q_2 , то МП-автомат P допускает входную цепочку $abba$.

Как и раньше, относительно легко показать, что если $w = c_1c_2\dots c_n c_{n-1}\dots c_1$, где $c_i \in \{a, b\}$ для $1 \leq i \leq n$, то

$$\begin{aligned} (q_0, w, Z) &\vdash^n (q_0, c_n c_{n-1}\dots c_1, c_n c_{n-1}\dots c_1 Z) \\ &\vdash (q_1, c_{n-1}\dots c_1, c_{n-1}\dots c_1 Z) \\ &\vdash^{n-1} (q_1, e, Z) \\ &\vdash (q_2, e, e) \end{aligned}$$

Таким образом, $L \subseteq L(P)$.

Несколько труднее доказать, что если $(q_0, w, Z) \vdash^* (q_2, e, \alpha)$ для некоторого $\alpha \in \Gamma^*$, то $w = xx^R$ для некоторого $x \in (a+b)^+$ и $\alpha = e$. Оставляем доказательство в качестве упражнения и, считая это утверждение истинным, заключаем, что $L(P) = L$. \square

В примере 2.32 ясно видна недетерминированная природа МП-автомата P . Для любой конфигурации вида $(q_0, aw, a\alpha)$ автомат P может сделать один из двух тактов: либо поместить в магазин еще один символ a , либо устраниить из магазина верхний символ a .

Подчеркнем, что хотя недетерминированный МП-автомат может обеспечивать удобное абстрактное определение языка, для реализации на практике нужно промоделировать его детерминировано. В гл. 4 мы рассмотрим методы моделирования недетерминированных МП-автоматов.

2.5.2. Варианты МП-автоматов

В этом разделе мы определим некоторые варианты понятия МП-автомата и установим связь между определяемыми ими языками и языками, определяемыми МП-автоматами в смысле первоначального определения. Но сначала нам хотелось бы выявить один фундаментальный аспект поведения МП-автомата, который интуитивно представляется совершенно ясным. Его можно сформулировать так: „То, что происходит с верхним символом магазина, не зависит от того, что находится в магазине под ним“.

Лемма 2.20. Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — МП-автомат. Если $(q, w, A) \vdash^n (q', e, e)$, то $(q, w, A\alpha) \vdash^{n'} (q', e, \alpha)$ для всех $A \in \Gamma$ и $\alpha \in \Gamma^*$.

Доказательство. Доказательство индукцией по n довольно элементарно. Для $n=1$ лемма, очевидно, верна. Допустим, что она верна для всех $1 \leq n < n'$, и пусть $(q, w, A) \vdash^{n'} (q', e, e)$. Тогда соответствующая последовательность тактов должна иметь вид

$$\begin{aligned} (q, w, A) \vdash & (q_1, w_1, X_1 \dots X_k) \\ \vdash^{n_1} & (q_2, w_2, X_2 \dots X_k) \\ & \cdot \\ & \cdot \\ & \cdot \\ \vdash^{n_{k-1}} & (q_k, w_k, X_k) \\ \vdash^{n_k} & (q', e, e) \end{aligned}$$

где $k \geq 1$ и $n_i < n'$ для $1 \leq i \leq k$ ¹⁾.

Тогда для любых $\alpha \in \Gamma^*$ также возможна последовательность

$$\begin{aligned} (q, w, A\alpha) \vdash & (q_1, w_1, X_1 \dots X_k \alpha) \\ \vdash^{n_1} & (q_2, w_2, X_2 \dots X_k \alpha) \\ & \cdot \\ & \cdot \\ & \cdot \\ \vdash^{n_{k-1}} & (q_k, w_k, X_k \alpha) \\ \vdash^{n_k} & (q', e, \alpha) \end{aligned}$$

¹⁾ Это еще один пример „очевидного“ утверждения, которое может потребовать некоторого размышления. Представьте себе МП-автомат, проходящий указанную последовательность конфигураций. Рано или поздно длина магазинной цепочки впервые станет равной $k-1$. Так как ни один из символов $X_2 \dots X_k$ не был верхним символом магазина, они так и останутся в нем, так что n_1 — число сделанных к этому моменту тактов. Затем ждем, когда длина магазина впервые станет равной $k-2$, и берем в качестве n_2 число дополнительных потребовавшихся тактов. Продолжаем в том же духе, пока магазин не станет пустым.

Всюду, кроме первого такта, мы пользовались здесь предположением индукции. \square

Теперь мы слегка расширим определение МП-автомата, позволив ему заменять за один такт цепочку символов ограниченной длины, расположенную в верхней части магазина, другой цепочкой конечной длины. Напомним, что МП-автомат в первоначальной версии мог на данном такте заменять лишь один верхний символ магазина.

Определение. Расширенным МП-автоматом назовем семерку $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, где δ — отображение конечного подмножества множества $Q \times (\Sigma \cup \{e\}) \times \Gamma^*$ в множество конечных подмножеств множества $Q \times \Gamma^*$, а все другие символы имеют тот же смысл, что и раньше.

Конфигурация определяется так же, как прежде, и мы пишем $(q, aw, \alpha\gamma) \vdash (q', w, \beta\gamma)$, если $\delta(q, a, \alpha)$ содержит (q', β) , где $q \in Q$, $a \in \Sigma \cup \{e\}$ и $\alpha \in \Gamma^*$. В этом такте цепочка α , расположенная в верхней части магазина, заменяется цепочкой β . Как и прежде, языком $L(P)$, определяемым автоматом P , называется множество

$$\{w \mid (q_0, w, Z_0) \vdash^* (q, e, \alpha) \text{ для некоторых } q \in F \text{ и } \alpha \in \Gamma^*\}$$

Заметим, что в отличие от обычного МП-автомата расширенный МП-автомат обладает способностью продолжать работу и тогда, когда магазин пуст.

Пример 2.33. Определим расширенный МП-автомат P , распознающий язык $L = \{ww^R \mid w \in \{a, b\}^*\}$. Пусть $P = (\{q, p\}, \{a, b\}, \{a, b, S, Z\}, \delta, q, Z, \{p\})$, где

- (1) $\delta(q, a, e) = \{(q, a)\}$
- (2) $\delta(q, b, e) = \{(q, b)\}$
- (3) $\delta(q, e, e) = \{(q, S)\}$
- (4) $\delta(q, e, aSa) = \{(q, S)\}$
- (5) $\delta(q, e, bSb) = \{(q, S)\}$
- (6) $\delta(q, e, SZ) = \{(p, e)\}$

На входе $aabbba$ автомат P может сделать следующую последовательность тактов:

$$\begin{aligned} (q, aabbba, Z) \vdash & (q, abbaa, aZ) \\ \vdash & (q, bbaa, aaZ) \\ \vdash & (q, baa, baaZ) \\ \vdash & (q, baa, SbaaZ) \\ \vdash & (q, aa, bSbaaZ) \end{aligned}$$

$$\begin{aligned} & \vdash (q, aa, SaaZ) \\ & \vdash (q, a, aSaaZ) \\ & \vdash (q, a, SaZ) \\ & \vdash (q, e, aSaZ) \\ & \vdash (q, e, SZ) \\ & \vdash (p, e, e) \end{aligned}$$

Работа автомата P состоит в том, что вначале он запасает в магазине некоторый префикс входной цепочки. Затем верхним символом магазина делается маркер S , указывающий предполагаемую середину входной цепочки. Далее P помещает в магазин очередной входной символ и заменяет в магазине aSa или bSb на S . Автомат P работает до тех пор, пока не исчерпается вся входная цепочка. Если после этого в магазине останется SZ , то P сорнет SZ и попадет в заключительное состояние. \square

Покажем, что язык L определяется МП-автоматом тогда и только тогда, когда он определяется расширенным МП-автоматом. Необходимость условия очевидна; докажем достаточность.

Лемма 2.21. Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — расширенный МП-автомат. Тогда существует такой МП-автомат P_1 , что $L(P_1) = L(P)$.

Доказательство. Положим $m = \max\{|\alpha| \mid \delta(q, a, \alpha) \neq \emptyset\}$ для некоторых $q \in Q, a \in \Sigma \cup \{e\}$. Построим МП-автомат P_1 , который будет моделировать автомат P , храня верхние m символов его магазина в „буфере“ длины m , занимающем часть конечной памяти управляющего устройства автомата P_1 . Тогда P_1 сможет сказать в начале каждого такта, каковы m верхних символов магазина автомата P . Если в некотором такте P заменяет k верхних символов магазина цепочкой из l символов, то P_1 заменит k первых символов в буфере этой цепочкой длины l . Если $l < k$, то P_1 сделает $k-l$ вспомогательных e -тактов, в течение которых $k-l$ символов перейдут из верхней части магазина в буфер управляющего устройства. После этого буфер окажется заполненным, и P_1 готов моделировать очередной такт автомата P . Если $l > k$, то символы передаются из буфера в магазин.

Итак, положим $P_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_1, Z_1, F_1)$, где

- (1) $Q_1 = \{[q, \alpha] \mid q \in Q, \alpha \in \Gamma_1^*, 0 \leq |\alpha| \leq m\}$;
- (2) $\Gamma_1 = \Gamma \cup \{Z_1\}$;
- (3) δ_1 определяется так:

(а) Допустим, что $\delta(q, a, X_1 \dots X_k)$ содержит $(r, Y_1 \dots Y_l)$.

(и) Если $l \geq k$, то для всех $Z \in \Gamma_1$ и $\alpha \in \Gamma_1^*, |\alpha| = m-k$,

$\delta_1([q, X_1 \dots X_k \alpha], a, Z)$ содержит $([r, \beta], \gamma Z)$

где $\beta \gamma = Y_1 \dots Y_l \alpha$ и $|\beta| = m$.

(ii) Если $l < k$, то для всех $Z \in \Gamma_1$ и $\alpha \in \Gamma_1^*, |\alpha| = m-k$,

$\delta_1([q, X_1 \dots X_k \alpha], a, Z)$ содержит $([r, Y_1 \dots Y_l \alpha Z], e)$

(б) Для всех $q \in Q, Z \in \Gamma_1$ и $\alpha \in \Gamma_1^*, |\alpha| < m$,

$\delta_1([q, \alpha], e, Z) = \{([q, \alpha Z], e)\}$

Эти правила осуществляют заполнение буфера управляющего устройства, которой содержит m символов.

(4) $q_1 = [q_0, Z_0 Z_1^{m-1}]$. В начальный момент буфер содержит Z_0 наверху и $m-1$ символов Z_1 пониже. Символы Z_1 используются как специальные маркеры, отмечающие „дно“, т. е. нижний конец магазина

(5) $F_1 = \{[q, \alpha] \mid q \in F, \alpha \in \Gamma_1^*\}$.

Нетрудно показать, что

$(q, aw, X_1 \dots X_k X_{k+1} \dots X_n) \vdash_P (r, w, Y_1 \dots Y_l X_{k+1} \dots X_n)$

тогда и только тогда, когда $([q, \alpha], aw, \beta) \vdash_{P_1} ([r, \alpha'], w, \beta')$, где

(1) $\alpha \beta = X_1 \dots X_n Z_1^m$,

(2) $\alpha' \beta' = Y_1 \dots Y_l X_{k+1} \dots X_n Z_1^m$,

(3) $|\alpha| = |\alpha'| = m$,

(4) между двумя указанными конфигурациями МП-автомата P_1 нет ни одной конфигурации, состояния которой имело бы вторую компоненту (буфер) длины m .

Для этого достаточно непосредственно применить правила, определяющие P_1 .

Таким образом, $(q_0, w, Z_0) \vdash_P^* (q, e, \alpha)$ для некоторых $q \in F$ и $\alpha \in \Gamma^*$ тогда и только тогда, когда

$([q_0, Z_0 Z_1^{m-1}], w, Z_1) \vdash_{P_1}^* ([q, \beta], e, \gamma)$

где $|\beta| = m$ и $\beta \gamma = \alpha Z_1^m$. Отсюда $L(P_1) = L(P)$. \square

Теперь исследуем входные цепочки, заставляющие МП-автомат опустошать магазин.

Определение. Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — МП-автомат или расширенный МП-автомат. Будем говорить, что P допускает цепочку $w \in \Sigma^*$ опустошением магазина, если $(q_0, w, Z_0) \vdash^+ (q, e, e)$ для некоторого $q \in Q$. Пусть $L_e(P)$ — множество цепочек, допускаемых автоматом P опустошением магазина.

Лемма 2.22. Пусть $L = L(P)$ для некоторого МП-автомата $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. Можно построить такой МП-автомат P' , что $L_e(P') = L$.

Доказательство. Пусть P' моделирует P . Всякий раз, когда P переходит в заключительное состояние, P' решает, про-

должать ли моделирование P или перейти в специальное состояние q_e , которое опустошит магазин. Но есть одно осложнение. Для некоторой входной цепочки w автомат P может сделать последовательность тактов, приводящую к опустошению магазина, а управляющее устройство окажется при этом не в заключительном состоянии. Тогда для того, чтобы помешать P' допустить в этом случае цепочку w , добавим к P' специальный маркер, отмечающий дно магазина, который автомат P' может устранить только в состоянии q_e . Формально положим $P' = (Q \cup \{q_e\}, \Sigma, \Gamma \cup \{Z'\}, \delta', q', Z', \emptyset)^1$, где δ' определяется так:

- (1) если $\delta(q, a, Z)$ содержит (r, γ) , то $\delta'(q, a, Z)$ содержит (r, γ) для всех $q \in Q$, $a \in \Sigma \cup \{e\}$ и $Z \in \Gamma$;
- (2) $\delta'(q', e, Z') = \{(q_0, Z_0 Z')\}$, т. е. на первом такте автомат P' записывает в магазин $Z_0 Z'$ и переходит в начальное состояние автомата P (Z' играет роль маркера, отмечающего дно магазина);
- (3) для всех $q \in F$ и $Z \in \Gamma \cup \{Z'\}$ множество $\delta'(q, e, Z)$ содержит (q_e, e) ;
- (4) $\delta'(q_e, e, Z) = \{(q_e, e)\}$ для всех $Z \in \Gamma \cup \{Z'\}$. Легко видеть, что

$$\begin{aligned} (q', w, Z') &\vdash_{P'} (q_0, w, Z_0 Z') \\ &\vdash_P^-(q, e, Y_1 \dots Y_r) \\ &\vdash_{P'} (q_e, e, Y_2 \dots Y_r) \\ &\vdash_P^{+1}(q_e, e, e) \end{aligned}$$

где $Y_r = Z'$, тогда и только тогда, когда

$$(q_0, w, Z_0) \vdash_P^-(q, e, Y_1 \dots Y_{r-1})$$

для $q \in F$ и $Y_1 \dots Y_{r-1} \in \Gamma^*$. Следовательно, $L_e(P') = L_e(P)$. \square

Справедливо также и обращение леммы 2.22.

Лемма 2.23. Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \emptyset)$ — МП-автомат. Можно построить такой МП-автомат P' , что $L(P') = L_e(P)$.

Доказательство. P' будет моделировать P , используя в надлежащий момент специальный символ Z' находящийся на дне магазина. В тот момент, когда автомат P' может прочесть Z' , он будет переходить в новое заключительное состояние q_f . Формальное построение оставляем в качестве упражнения. \square

¹⁾ Для МП-автомата, допускающего язык опустошением магазина, множество заключительных состояний обычно берется пустым. Его, очевидно, можно сделать каким угодно.

2.5.3. Эквивалентность МП-автоматов и КС-грамматик

С помощью полученных результатов можно теперь показать, что языки, определяемые МП-автоматами — это в точности КС-языки. Начнем с построения естественного (недетерминированного) „исходящего“ распознавателя, эквивалентного данной КС-грамматике.

Лемма 2.24. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. По грамматике G можно построить такой МП-автомат R , что $L_e(R) = L(G)$.

Доказательство. Построим R так, чтобы он моделировал все левые выводы в G .

Пусть $R = (\{q\}, \Sigma, N \cup \Sigma, \delta, q, S, \emptyset)$, где δ определяется следующим образом:

- (1) если $A \rightarrow \alpha$ принадлежит P , то $\delta(q, e, A)$ содержит (q, α) ;
- (2) $\delta(q, e, A) = \{(q, e)\}$ для всех $a \in \Sigma$.

Мы хотим показать, что

$$(2.5.2) \quad A \Rightarrow^m w \text{ тогда и только тогда, когда } (q, w, A) \vdash^m (q, e, e) \text{ для некоторых } m, n \geq 1$$

Необходимость условия докажем индукцией по m . Допустим, что $A \Rightarrow^m w$. Если $m=1$ и $w=a_1 \dots a_k$ ($k \geq 0$), то

$$\begin{aligned} (q, a_1 \dots a_k, A) &\vdash (q, a_1 \dots a_k, a_1 \dots a_k) \\ &\vdash^k (q, e, e) \end{aligned}$$

Теперь предположим, что $A \Rightarrow^m w$ для некоторого $m > 1$. Первый шаг этого вывода должен иметь вид $A \Rightarrow X_1 X_2 \dots X_k$, где $X_i \Rightarrow^{m_i} x_i$ для некоторого $m_i < m$, $1 \leq i \leq k$, и $x_1 x_2 \dots x_k = w$. Тогда

$$(q, w, A) \vdash (q, w, X_1 X_2 \dots X_k)$$

Если $X_i \in N$, то по предположению индукции

$$(q, x_i, X_i) \vdash^* (q, e, e)$$

Если $X_i = x_i \in \Sigma$, то

$$(q, x_i, X_i) \vdash (q, e, e)$$

Объединяя вместе эти последовательности тактов, видим, что $(q, w, A) \vdash^+(q, e, e)$.

Для доказательства достаточности покажем индукцией по n , что если $(q, w, A) \vdash^n (q, e, e)$, то $A \Rightarrow^+ w$.

Если $n=1$, то $w=e$ и $A \rightarrow e$ принадлежит P . Предположим, что утверждение верно для всех $n' < n$. Тогда первый такт,

сделанный МП-автоматом R , должен иметь вид

$$(q, w, A) \vdash (q, w, X_1 \dots X_k)$$

причем $(q, x_i, X_i) \vdash^{n_i} (q, e, e)$ для $1 \leq i \leq k$ и $w = x_1 x_2 \dots x_k$ (лемма 2.20). Тогда $A \rightarrow X_1 \dots X_k$ — правило из P , и по предположению индукции $X_i \Rightarrow^+ x_i$ для $X_i \in N$. Если $X_i \in \Sigma$, то $X_i \Rightarrow^0 x_i$. Таким образом,

$$\begin{aligned} A &\Rightarrow X_1 \dots X_k \\ &\Rightarrow^* x_1 X_2 \dots X_k \\ &\quad \vdots \\ &\quad \vdots \\ &\Rightarrow^* x_1 x_2 \dots x_{k-1} X_k \\ &\Rightarrow^* x_1 x_2 \dots x_{k-1} x_k = w \end{aligned}$$

— вывод цепочки w из A в грамматике G .

Из (2.5.2), в частности, вытекает, что $S \Rightarrow^+ w$ тогда и только тогда, когда $(q, w, S) \vdash^+ (q, e, e)$. Следовательно, $L_e(R) = L(G)$. \square

Пример 2.34. Построим МП-автомат P , для которого $L_e(P) = L(G_0)$, где G_0 — наша обычная грамматика, определяющая арифметические выражения. Пусть $P = (\{q\}, \Sigma, \Gamma, \delta, q, E, \emptyset)$, где δ определяется так:

- (1) $\delta(q, e, E) = \{(q, E + T), (q, T)\};$
- (2) $\delta(q, e, T) = \{(q, T * F), (q, F)\};$
- (3) $\delta(q, e, F) = \{(q, (E)), (q, a)\};$
- (4) $\delta(q, b, b) = \{(q, e)\}$ для всех $b \in \{a, +, *, ()\}$.

При входе $a + a * a$ для P возможна среди других последовательность тактов

$$\begin{aligned} (q, a + a * a, E) &\vdash (q, a + a * a, E + T) \\ &\vdash (q, a + a * a, T + T) \\ &\vdash (q, a + a * a, F + T) \\ &\vdash (q, a + a * a, a + T) \\ &\vdash (q, + a * a, + T) \\ &\vdash (q, a * a, T) \\ &\vdash (q, a * a, T * F) \\ &\vdash (q, a * a, F * F) \\ &\vdash (q, a * a, a * F) \\ &\vdash (q, * a, * F) \\ &\vdash (q, a, F) \\ &\vdash (q, a, a) \\ &\vdash (q, e, e) \end{aligned}$$

Заметим, что вычисление МП-автомата P соответствует левому выводу цепочки $a + a * a$ из E в КС-грамматике G_0 . \square

Тип синтаксического анализа, проводимого для КС-грамматики МП-автоматом, построенным в лемме 2.24, называется „нисходящим анализом“ („анализом сверху вниз“) или „предсказывающим анализом“, потому что при этом дерево вывода строится по существу сверху (от корня) вниз, а каждый торт вида (1) можно трактовать как предсказание очередного шага левого вывода. Подробно нисходящий синтаксический анализ будет обсуждаться в гл. 3—5.

Можно построить расширенный МП-автомат, который работает „снизу вверх“ как „восходящий анализатор“, моделируя в обратном порядке правые выводы в КС-грамматике. Рассмотрим цепочку $a + a * a \in L(G_0)$ и ее правый вывод из E в грамматике G_0 :

$$\begin{aligned} E &\Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * a \Rightarrow E + F * a \\ &\Rightarrow E + a * a \Rightarrow T + a * a \Rightarrow F + a * a \Rightarrow a + a * a \end{aligned}$$

Предположим, что мы записываем этот вывод в обратном порядке. Если считать, что переход от цепочки $a + a * a$ к цепочке $F + a * a$ происходит по правилу $F \rightarrow a$, примененному „в обратном направлении“, то можно сказать, что цепочка $a + a * a$ „свертывается слева“ к цепочке $F + a * a$. Более того, это единственное возможное левая свертка. Подобным же способом можно правовыводимую цепочку $F + a * a$ свернуть слева к цепочке $T + a * a$ с помощью правила $T \rightarrow F$ и т. д. Определим формально левую свертку.

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика и

$$S \Rightarrow_*^* \alpha A \omega \Rightarrow_r \alpha \beta \omega \Rightarrow_*^* \chi \omega$$

— правый вывод в ней. Будем говорить, что правовыводимую цепочку $\alpha \beta \omega$ можно *свернуть слева* (или что она *левосвертывается*) к правовыводимой цепочке $\alpha A \omega$ с помощью правила $A \rightarrow \beta$. Указанное вхождение цепочки β в цепочку $\alpha \beta \omega$ назовем основой цепочки $\alpha \beta \omega$.

Таким образом, основа правовыводимой цепочки — это ее любая подцепочка, которая является правой частью некоторого правила, причем после замены ее левой частью этого правила тоже получается правовыводимая цепочка.

Пример 2.35. Рассмотрим грамматику с правилами

$$\begin{aligned} S &\rightarrow Ac \mid Bd \\ A &\rightarrow aAb \mid ab \\ B &\rightarrow aBbb \mid abb \end{aligned}$$

Она порождает язык $\{a^n b^n c \mid n \geq 1\} \cup \{a^n b^{2n} d \mid n \geq 1\}$.

Рассмотрим правовыводимую цепочку $aabbdd$. Единственная ее основа — подцепочка abb , так как $aBbb$ — правовыводимая цепочка. Заметим, что хотя ab — правая часть правила $A \rightarrow ab$, она не будет основой цепочки $aabbdd$, так как $aBbb$ — не правовыводимая цепочка. \square

Основу правовыводимой цепочки можно было бы определить другим способом, сказав, что основа — это крона самого левого поддерева глубины 1 некоторого дерева вывода этой правовыводимой цепочки. (Дерево глубины 1, состоящее из вершины и ее прямых потомков, которые являются листьями, называют *кустом* или *реером*.)

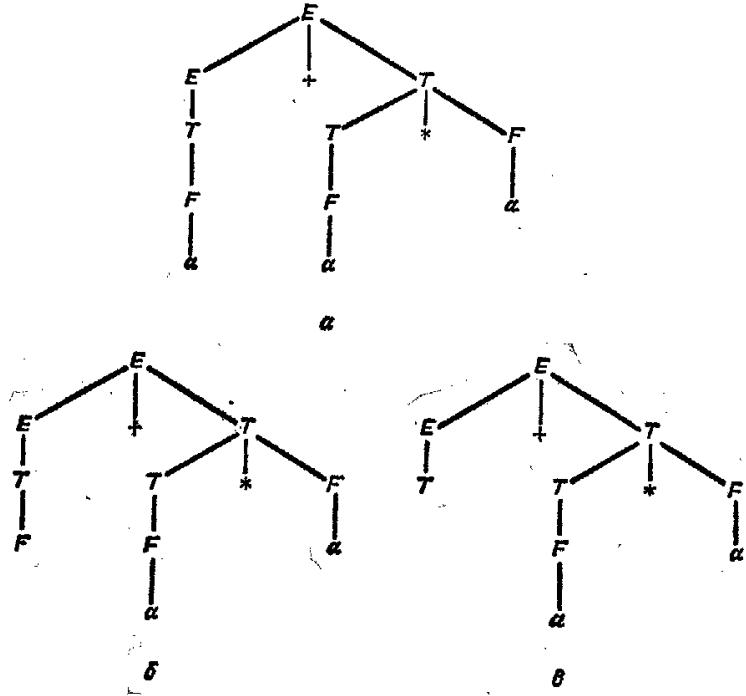


Рис. 2.16. Отсечение основ.

Дерево вывода цепочки $a+a*a$ в грамматике G_0 показано на рис. 2.16, а. Самый левый куст состоит из самой левой вершины, помеченной F , которая является его корнем, и кроны a .

Если удалить единственный лист самого левого куста, то останется дерево, показанное на рис. 2.16, б. Крона $F+a*a$ этого дерева и есть результат левой свертки цепочки $a+a*a$,

а его основа — крона F поддерева с корнем T . Опять удалив основу, получим дерево на рис. 2.16, в.

Описанный процесс свертки деревьев назовем *отсечением основ*.

По КС-грамматике G можно построить эквивалентный расширенный МП-автомат P , работа которого заключается в отсечении основ. Здесь удобно представлять себе магазин в виде цепочки, верхним символом которой является самый правый, а не самый левый символ. В силу этого соглашения конфигурации (расширенного) МП-автомата $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ определяются точно так же, как раньше, а отношение \vdash несколько иначе. Если $\delta(q, a, \alpha)$ содержит (p, β) , то будем писать $(q, aw, \gamma\alpha) \vdash (p, w, \gamma\beta)$ для всех $w \in \Sigma^*$ и $\gamma \in \Gamma^*$.

Таким образом, утверждение „ $\delta(q, a, YZ)$ содержит (p, VWX) “ имеет разные смыслы в зависимости от того, справа или слева расположена верхняя часть магазина. Если слева, то верхними символами до и после этого такта будут Y и V , а если справа, то верхними символами будут Z и X . По данному МП-автомату, верхний символ которого расположен слева, можно построить МП-автомат, делающий то же самое, но с верхним символом, расположенным справа, просто записывая в обратном порядке все цепочки из Γ^* . Например, если было $(p, VWX) \in \delta(q, a, YZ)$, то станет $(p, XWV) \in \delta(q, a, ZY)$. Разумеется, нужно оговорить тот факт, что верх магазина находится теперь справа. Обратно, МП-автомат с верхним символом, расположенным справа, легко превратить в МП-автомат с верхним символом слева.

Мы видим, что обозначение МП-автомата в виде семерки можно интерпретировать как два разных МП-автомата в зависимости от того, какой из символов — правый или левый — считается верхним. Нам кажется, что удобство обозначений, создаваемое этими двумя соглашениями, перевешивает возможность путаницы в исходных понятиях. В дальнейшем, если не оговорено противное, будем считать, что у обычного МП-автомата верх магазина расположен слева, а у расширенного — справа.

Лемма 2.25. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. По G можно построить такой расширенный МП-автомат R , что $L(R) = L(G)$ ¹⁾.

Работу автомата R „разумно“ рассматривать как процесс отсечения основ.

Доказательство. Пусть $R = (\{q, r\}, \Sigma, N \cup \Sigma \cup \{\$\}, \delta, q, \$, \{r\})$ — расширенный МП-автомат²⁾, в котором δ определяется следующим образом:

¹⁾ Лемма 2.25 очевидно следует из лемм 2.23 и 2.24, но здесь интересна сама конструкция.

²⁾ По нашему соглашению верх его магазиналожен справа.

(1) $\delta(q, a, e) = \{(q, a)\}$ для всех $a \in \Sigma$. (На этих тактах входные символы переносятся в магазин.)

(2) Если $A \rightarrow \alpha$ принадлежит P , то $\delta(q, e, \alpha)$ содержит (q, A) .

(3) $\delta(q, e, \$S) = \{(r, e)\}$.

Покажем, что процесс вычисления в автомате R заключается в построении правовыводимых цепочек грамматики G , начиная с терминальной цепочки (на входе R) и кончая цепочкой S . Индукцией по n докажем, что

(2.5.3) $S \Rightarrow_r^* \alpha A y \Rightarrow_r^n xy$ влечет $(q, xy, \$) \vdash^* (q, y, \$\alpha A)$

Базис, $n=0$, тривиален: R не делает ни одного такта. Предположим, что (2.5.3) верно для всех значений параметра, меньших n . Можно написать $\alpha A y \Rightarrow_r \alpha \beta y \Rightarrow_r^{n-1} xy$. Допустим, что цепочка $\alpha \beta$ состоит только из терминалов. Тогда $\alpha \beta = x$ и $(q, xy, \$) \vdash^* (q, y, \$\alpha \beta) \vdash (q, y, \$\alpha A)$.

Если $\alpha \beta \notin \Sigma^*$, то можно писать $\alpha \beta = \gamma B z$, где B — самый правый нетерминал. По (2.5.3) из $S \Rightarrow_r^* \gamma B z y \Rightarrow_r^{n-1} xy$ следует $(q, xy, \$) \vdash^* (q, zy, \$\gamma B)$. Кроме того, $(q, zy, \$\gamma B) \vdash^* (q, y, \$\gamma B z) \vdash (q, y, \$\alpha A)$ — тоже возможная последовательность тактов.

Мы заключаем, что (2.5.3) верно для всех n . Так как $(q, e, \$S) \vdash (r, e, e)$, то $L(G) \equiv L(R)$.

Для того чтобы доказать обратное включение $L(R) \equiv L(G)$ и, следовательно, равенство $L(G) = L(R)$, докажем, что

(2.5.4) $(q, xy, \$) \vdash^n (q, y, \$\alpha A)$ влечет $\alpha A y \Rightarrow^* xy$

Базис, $n=0$, выполняется автоматически. Для проверки шага индукции предположим, что (2.5.4) верно для всех $n < m$. Если верхний символ магазина автомата R нетерминальный, то последний такт был сделан по правилу (2) определения функции δ . Поэтому можно писать

$(q, xy, \$) \vdash^{m-1} (q, y, \$\alpha \beta) \vdash (q, y, \$\alpha A)$

где $A \rightarrow \beta$ принадлежит P . Если цепочка $\alpha \beta$ содержит нетерминал, то по предположению индукции $\alpha \beta y \Rightarrow^* xy$. Отсюда $\alpha A y \Rightarrow \alpha \beta y \Rightarrow^* xy$, что и утверждалось.

В качестве частного случая утверждения (2.5.4) получаем, что $(q, w, \$) \vdash^* (q, e, \$S)$ влечет $S \Rightarrow_r^* w$. Так как R допускает w только тогда, когда $(q, w, \$) \vdash^* (q, e, \$S) \vdash (r, e, e)$, то отсюда следует, что $L(R) \equiv L(G)$. Таким образом, $L(R) = L(G)$. \square

Заметим, что сразу после свертки правовыводимая цепочка вида αAx представлена в R так, что αA находится в магазине, а x занимает непрочитанную часть входной ленты. После этого R может продолжать переносить символы цепочки x в магазин до тех пор, пока в его верхней части не образуется основа.

Тогда R может сделать следующую свертку. Синтаксический анализ этого типа называют „восходящим анализом“ („анализом снизу вверх“) или „анализом сверткой“.

Пример 2.36. Построим восходящий анализатор¹⁾ R для грамматики G_0 . Пусть R — расширенный МП-автомат $(\{q, r\}, \Sigma, \Gamma, \delta, q, \$, \{r\})$, где δ определяется так:

- | | |
|----------------------------------------|---------------------------------------|
| (1) $\delta(q, b, e) = \{(q, b)\}$ | для всех $b \in \{a, +, *, (), *\}$; |
| (2) $\delta(q, e, E + T) = \{(q, E)\}$ | |
| $\delta(q, e, T) = \{(q, E)\}$ | |
| $\delta(q, e, T * F) = \{(q, T)\}$ | |
| $\delta(q, e, F) = \{(q, T)\}$ | |
| $\delta(q, e, (E)) = \{(q, F)\}$ | |
| $\delta(q, e, a) = \{(q, F)\};$ | |
| (3) $\delta(q, e, \$E) = \{(r, e)\}$. | |

Для входа $a + a * a$ распознаватель R может сделать следующую последовательность тактов:

$(q, a + a * a, \$) \vdash (q, + a * a, \$a)$
 $\vdash (q, + a * a, \$F)$
 $\vdash (q, + a * a, \$T)$
 $\vdash (q, + a * a, \$E)$
 $\vdash (q, a * a, \$E +)$
 $\vdash (q, * a, \$E + a)$
 $\vdash (q, * a, \$E + F)$
 $\vdash (q, * a, \$E + T)$
 $\vdash (q, a, \$E + T *)$
 $\vdash (q, e, \$E + T * a)$
 $\vdash (q, e, \$E + T * F)$
 $\vdash (q, e, \$E + T)$
 $\vdash (q, e, \$E)$
 $\vdash (r, e, e)$

Заметим, что для входа $a + a * a$ распознаватель R может сделать много различных последовательностей тактов. Однако выписанная последовательность — единственная, которая переводит начальную конфигурацию в заключительную. \square

Покажем теперь, что язык, определяемый МП-автоматом, контекстно-свободен.

Лемма 2.26. Пусть $R = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — МП-автомат. Можно построить такую КС-грамматику G , что $L(G) = L_e(R)$.

¹⁾ Автомат, который строится в этом примере (а также и в примере 2.34), — это еще не анализатор, а только распознаватель, но его легко превратить в анализатор (см. гл. 4 и 5). — Прим. перев.

Доказательство. Построим G так, чтобы левый вывод цепочки w в грамматике G прямо соответствовал последовательности тиков, которую делает R при обработке цепочки w . Нетерминальные символы будут иметь вид $[qZr]$, где $q, r \in Q$ и $Z \in \Gamma$. Потом мы покажем, что $[qZr] \Rightarrow^+ w$ тогда и только тогда, когда $(q, w, Z) \vdash^+ (r, e, e)$.

Формально пусть $G = (N, \Sigma, P, S)$, где

$$(1) N = \{[qZr] \mid q, r \in Q, Z \in \Gamma\} \cup \{S\};$$

(2) правила множества P строятся так:

- (a) если $\delta(q, a, Z)$ содержит $(r, X_1 \dots X_k)^1$ ($k \geq 1$), добавим к P все правила вида

$$[qZs_k] \rightarrow a[rX_1s_1][s_1X_2s_2] \dots [s_{k-1}X_ks_k]$$

для каждой последовательности s_1, s_2, \dots, s_k состояний из Q ,

- (b) если $\delta(q, a, Z)$ содержит (r, e) , добавим к P правило $[qZr] \rightarrow a$,
- (v) добавим к P правила $S \rightarrow [q_0Z_0q]$ для каждого $q \in Q$.

Индукцией по m и n легко доказать, что для любых $q, r \in Q$ и $Z \in \Gamma$

$$[qZr] \Rightarrow^m w \text{ тогда и только тогда, когда } (q, w, Z) \vdash^n (r, e, e).$$

Доказательство оставляем в качестве упражнения. Из этого утверждения следует, что $S \Rightarrow [q_0Z_0q] \Rightarrow^+ w$ тогда и только тогда, когда $(q_0, w, Z_0) \vdash^+ (q, e, e)$ для $q \in Q$. Таким образом, $L_e(R) = L(G)$. \square

Результаты двух последних разделов можно сформулировать в виде следующей теоремы:

Теорема 2.21. Утверждения

- (1) $L = L(G)$ для КС-грамматики G ,
- (2) $L = L(P_1)$ для МП-автомата P_1 ,
- (3) $L = L_e(P_2)$ для МП-автомата P_2 ,
- (4) $L = L(P_3)$ для расширенного МП-автомата P_3

эквивалентны.

Доказательство. Из (3) следует (1) по лемме 2.26. Из (1) следует (3) по лемме 2.24. Из (4) следует (2) по лемме 2.21, а (4) тривиально следует из (2). Из (2) следует (3) по лемме 2.22, и из (3) следует (2) по лемме 2.23. \square

¹⁾ Верхний символ магазина R расположен слева, так как мы не оговаривали противное.

2.5.4. Детерминированные МП-автоматы

Мы уже видели, что для каждой КС-грамматики G можно построить МП-автомат, распознающий $L(G)$. Однако построенный МП-автомат был недетерминированным. В практических приложениях нас больше интересуют детерминированные МП-автоматы, т. е. такие, которые в каждой конфигурации могут сделать не более одного очередного такта. В этом разделе мы займемся детерминированными МП-автоматами и в дальнейшем увидим, что, к сожалению, детерминированные МП-автоматы не так мощны по своей распознавательной способности, как недетерминированные МП-автоматы. Существуют КС-языки, которые нельзя определить детерминированными МП-автоматами.

Язык, определяемый детерминированным МП-автоматом, называется *детерминированным КС-языком*. В гл. 5 мы введем подкласс КС-грамматик, называемых $LR(k)$ -грамматиками, а в гл. 8 покажем, что каждая $LR(k)$ -грамматика порождает детерминированный КС-язык и каждый детерминированный КС-язык определяется $LR(1)$ -грамматикой.

Определение. МП-автомат $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ называется *детерминированным* (сокращенно *ДМП-автоматом*), если для каждого $q \in Q$ и $Z \in \Gamma$ либо

- (1) $\delta(q, a, Z)$ содержит не более одного элемента для каждого $a \in \Sigma$ и $\delta(q, e, Z) = \emptyset$, либо
- (2) $\delta(q, a, Z) = \emptyset$ для всех $a \in \Sigma$ и $\delta(q, e, Z)$ содержит не более одного элемента.

В силу этих двух ограничений ДМП-автомат в любой конфигурации может выбрать не более одного такта. Это приводит к тому, что на практике гораздо легче моделировать детерминированный МП-автомат, чем недетерминированный. По этой причине класс детерминированных КС-языков важен для практических приложений.

Соглашение. Так как для ДМП-автомата $\delta(q, a, Z)$ содержит не более одного элемента, мы будем писать $\delta(q, a, Z) = (r, \gamma)$ вместо $\delta(q, a, Z) = \{(r, \gamma)\}$.

Пример 2.37. Построим ДМП-автомат, распознающий язык $L = \{wcz^R \mid w \in \{a, b\}^+\}$. Пусть $P = (\{q_0, q_1, q_2\}, \{a, b, c\}, \{Z, a, b\}, \delta, q_0, Z, \{q_2\})$, где δ определяется так:

$$\delta(q_0, X, Y) = (q_0, XY) \text{ для всех } X \in \{a, b\} \text{ и } Y \in \{Z, a, b\}$$

$$\delta(q_0, c, Y) = (q_1, Y) \text{ для всех } Y \in \{a, b\}$$

$$\delta(q_1, X, X) = (q_1, e) \text{ для всех } X \in \{a, b\}$$

$$\delta(q_1, e, Z) = (q_2, e)$$

До тех пор пока P не увидит маркер c , отмечающий середину, он запасает в магазине символы входной цепочки. Когда P достигает c , он переходит в состояние q_1 и далее сравнивает оставшуюся часть входной цепочки с содержимым магазина. Доказательство того, что $L(P) = L$, оставляем в качестве упражнения. \square

Определение. Расширенный МП-автомата можно естественным образом расширить, чтобы включить в него те расширенные МП-автоматы, которые естественно считать детерминированными.

Определение. Расширенный МП-автомат $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ называется *детерминированным*, если выполнены следующие условия:

- (1) $\#\delta(q, a, \gamma) \leq 1$ для всех $q \in Q, a \in \Sigma \cup \{e\}$ и $\gamma \in \Gamma^*$;
- (2) если $\delta(q, a, \alpha) \neq \emptyset, \delta(q, a, \beta) \neq \emptyset$ и $\alpha \neq \beta$, то ни одна из цепочек α и β не является суффиксом другой¹;
- (3) если $\delta(q, a, \alpha) \neq \emptyset$ и $\delta(q, e, \beta) \neq \emptyset$, то ни одна из цепочек α и β не является суффиксом другой.

Легко видеть, что в том частном случае, когда расширенный МП-автомат является обычным ДМП-автоматом, это определение согласуется с прежним. Кроме того, если конструкция леммы 2.21 применяется к расширенному МП-автомату, результатом будет детерминированный МП-автомат тогда и только тогда, когда детерминированным был исходный расширенный МП-автомат.

При моделировании синтаксического анализатора желательно иметь ДМП-автомат P ,читывающий всю входную цепочку, даже если она не приадлежит языку $L(P)$. Покажем, что такой ДМП-автомат всегда можно найти.

Сначала модифицируем ДМП-автомат так, чтобы для любой конфигурации, в которой часть входа осталась непрочитанной, был всегда возможен очередной тиктак. Следующая лемма показывает, как это сделать.

Лемма 2.27. Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — ДМП-автомат. Можно построить такой эквивалентный ДМП-автомат $P' = (Q', \Sigma, \Gamma', \delta', q'_0, Z'_0, F)$, что

- (1) для всех $a \in \Sigma, q \in Q'$ и $Z \in \Gamma'$ либо
 - (а) $\delta'(q, a, Z)$ содержит точно один элемент и $\delta'(q, e, Z) = \emptyset$, либо

¹⁾ Если верх магазина расширенного МП-автомата расположен слева, то здесь надо заменить „суффикс“ на „предфикс“.

(б) $\delta'(q, a, Z) = \emptyset$ и $\delta'(q, e, Z)$ содержит точно один элемент;

- (2) если $\delta'(q, a, Z'_0) = (r, \gamma)$ для некоторого $a \in \Sigma \cup \{e\}$, то $\gamma = \alpha Z'_0$ для некоторой цепочки $\alpha \in \Gamma^*$.

Доказательство. Символ Z'_0 будет действовать как маркер, указывающий конец (дно) магазина и предотвращающий полное опустошение магазина. Пусть $\Gamma' = \Gamma \cup \{Z'_0\}$ и $Q' = \{q'_0, q_e\} \cup Q$, а δ' определяется так:

- (1) $\delta'(q'_0, e, Z'_0) = (q_0, Z_0 Z'_0)$;
- (2) для всех $q \in Q, a \in \Sigma \cup \{e\}$ и $Z \in \Gamma$, таких, что $\delta(q, a, Z) \neq \emptyset$, полагаем $\delta'(q, a, Z) = \delta(q, a, Z)$;
- (3) если $\delta(q, e, Z) = \emptyset$ и $\delta(q, a, Z) = \emptyset$ для некоторых $a \in \Sigma$ и $Z \in \Gamma$, полагаем $\delta'(q, a, Z) = (q_e, Z)$;
- (4) для всех $Z \in \Gamma'$ и $a \in \Sigma$ полагаем $\delta'(q_e, a, Z) = (q_e, Z)$.

Благодаря правилу (1) P' запишет в верхней части магазина $Z_0 Z'_0$ и, перейдя в состояние q_0 , начнет моделировать P . Правила (2) позволяют P' моделировать P , пока очередной тиктак станет невозможным. В этой ситуации P' (по правилу 3) перейдет в незаключительное состояние q_e и останется в нем, не изменяя содержимого магазина, пока не будет прочитана оставшаяся часть входа. Доказательство того, что $L(P') = L(P)$, оставляем в качестве упражнения. \square

ДМП-автомат может, исходя из некоторой конфигурации, проделать бесконечное число e -тиктаков, не прочитав больше ни одного входного символа. Такую конфигурацию мы называем *зацикливающей*.

Определение. Конфигурация (q, w, α) ДМП-автомата P называется *зацикливающей*, если для каждого $i \geq 1$ найдется такая конфигурация (p_i, w, β_i) , что $|\beta_i| \geq |\alpha|$ и

$$(q, w, \alpha) \vdash (p_1, w, \beta_1) \vdash (p_2, w, \beta_2) \vdash \dots$$

Таким образом, конфигурация зацикливающая, если P может делать бесконечное число e -тиктаков, не укорачивая магазин; при этом магазин может либо бесконечно расти, либо циклически совпадать с несколькими различными цепочками.

Заметим, что существуют незацикливающие конфигурации, которые после ряда e -тиктаков, укорачивающих магазин, переходят в зацикливающую конфигурацию. Мы покажем, что нельзя сделать бесконечное число e -тиктаков, исходя из любой данной конфигурации, не попав через конечное и притом вычислимое число тиктаков в зацикливающую конфигурацию.

Если P попадает в зацикливающую конфигурацию в середине входной цепочки, то он больше не будет использовать

входную цепочку, даже если удовлетворяет лемме 2.27. Мы хотим преобразовать данный ДМП-автомат P в эквивалентный ДМП-автомат P' , который никогда не попадает в зацикливающую конфигурацию.

Алгоритм 2.16. Обнаружение зацикливающих конфигураций.

Вход. ДМП-автомат $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$.

Выход. (1) $C_1 = \{(q, A) \mid (q, e, A) \text{ — зацикливающая конфигурация и не существует такого } r \in F, \text{ что } (q, e, A) \vdash^*(r, e, \alpha) \text{ для некоторой цепочки } \alpha \in \Gamma^*\}$ и

(2) $C_2 = \{(q, A) \mid (q, e, A) \text{ — зацикливающая конфигурация и } (q, e, A) \vdash^*(r, e, \alpha) \text{ для некоторых } r \in F \text{ и } \alpha \in \Gamma^*\}$.

Метод. Пусть $\# Q = n_1$, $\# \Gamma = n_2$ и l — длина самой длинной цепочки, которую P может записать в магазин за один такт. Пусть $n_3 = n_1(n_2^{n_1 n_2 l + 1} - n_2)/(n_2 - 1)$, если $n_2 > 1$, и $n_3 = n_1$, если $n_2 = 1$. Число n_3 — это максимальное число e -тактов, которое может сделать P , не зациклившись.

(1) Для всех $q \in Q$ и $A \in \Gamma$ выяснить, выполняется ли $(q, e, A) \vdash^{n_3}(r, e, \alpha)$ для каких-нибудь $r \in Q$ и $\alpha \in \Gamma^+$. При этом используется прямое моделирование P . Если да, то (q, e, A) — зацикливающая конфигурация, так как в этом случае — мы это покажем — должна быть такая пара (q', A') , где $q' \in Q$ и $A' \in \Gamma$, что

$$\begin{array}{c} (q, e, A) \vdash^* (q', e, A'\beta) \\ \vdash^m (q', e, A'\gamma\beta) \\ \vdash^{m(l-1)} (q', e, A'\gamma^l\beta) \end{array}$$

где $m > 0$ и $l > 0$. Заметим, что γ может быть e .

(2) Если (q, e, A) — зацикливающая конфигурация, выяснить, существует ли такое $r \in F$, что $(q, e, A) \vdash^j(r, e, \alpha)$ для некоторого $0 \leq j \leq n_3$. При этом снова используется прямое моделирование. Если да, то включить (q, A) в C_2 . В противном случае включить (q, A) в C_1 . Мы утверждаем, что если P может достичь заключительной конфигурации, исходя из (q, e, A) , то это должно произойти за n_3 или меньшее число тактов. \square

Теорема 2.22. Алгоритм 2.16 правильно находит множества C_1 и C_2 .

Доказательство. Сначала докажем, что шаг (1) правильно определяет множество $C_1 \cup C_2$. Если $(q, A) \in C_1 \cup C_2$, то очевидно, что $(q, e, A) \vdash^{n_3}(r, e, \alpha)$. Обратно, допустим, что $(q, e, A) \vdash^{n_3}(r, e, \alpha)$.

Случай 1: Существует такая цепочка $\beta \in \Gamma^*$, что $|\beta| > n_1 n_2 l$ и $(q, e, A) \vdash^*(p, e, \beta) \vdash^*(r, e, \alpha)$ для некоторого $p \in Q$. Если в

последовательности тактов $(q, e, A) \vdash^*(p, e, \beta)$ для каждого $j = 1, 2, \dots, n_1 n_2 l + 1$ выделить конфигурацию, в которой оказывается P , когда длина его магазина в последний раз становится равной j , то можно заметить, что в выделенной последовательности конфигураций некоторое состояние q' и символ A' должны дважды встречаться в качестве текущего состояния и верхнего символа магазина; другими словами, можно писать $(q, e, A) \vdash^*(q', e, A'\delta) \vdash^+(q', e, A'\gamma\delta) \vdash^*(p, e, \beta)$. Таким образом, по лемме 2.20 $(q, e, A) \vdash^*(q', e, A'\delta) \vdash^{mj}(q', e, A'\gamma^j\delta) \vdash^*(p, e, \beta)$ для всех $j \geq 0$. Здесь $m \geq 0$, поэтому, исходя из конфигурации (q, e, A) , можно сделать бесконечно много e -тактов и, следовательно, $(q, A) \in C_1 \cup C_2$.

Случай 2: Допустим, что в противоположность случаю 1 $|\beta| \leq n_1 n_2 l$ для всех таких β , что $(q, e, A) \vdash^*(p, e, \beta) \vdash^*(r, e, \alpha)$. Так как в этой последовательности конфигураций имеется $n_3 + 1$ различных β , число возможных состояний равно n_1 и число возможных магазинных цепочек длины, не большей $n_1 n_2 l$, равно $n_2 + n_2^2 + n_2^3 + \dots + n_2^{n_1 n_2 l} = (n_2^{n_1 n_2 l + 1} - n_2)/(n_2 - 1)$, то некоторая конфигурация должна повторяться. Отсюда непосредственно следует, что $(q, A) \in C_1 \cup C_2$.

Доказательство того, что шаг (2) правильно разбивает множество $C_1 \cup C_2$ на C_1 и C_2 , оставляем в качестве упражнения. \square

Определение. ДМП-автомат $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ назовем *дочитывающим*, если для каждой цепочки $w \in \Sigma^*$ существуют такие $p \in Q$ и $\alpha \in \Gamma^*$, что $(q_0, w, Z_0) \vdash^*(p, e, \alpha)$. Интуитивно, дочитывающим называется ДМП-автомат, способный дочивать до конца каждую входную цепочку.

Лемма 2.28. Пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — ДМП-автомат. Тогда существует эквивалентный ему дочитывающий ДМП-автомат P' .

Доказательство. В силу леммы 2.27 можно считать, что для P всегда возможен очередной такт. Пусть $P' = (Q \cup \{p, r\}, \Sigma, \Gamma, \delta', q_0, Z_0, F \cup \{p\})$, где p и r — новые состояния, а δ' определяется так:

- (1) для всех $q \in Q$, $a \in \Sigma$ и $Z \in \Gamma$ положим $\delta'(q, a, Z) = \delta(q, a, Z)$;
- (2) для всех $q \in Q$ и $Z \in \Gamma$, таких, что, (q, e, Z) не является зацикливающей конфигурацией, положим $\delta'(q, e, Z) = \delta(q, e, Z)$;
- (3) для всех $(q, \bar{Z}) \in C_1$, где C_1 — множество, построенное алгоритмом 2.16, положим $\delta'(q, e, \bar{Z}) = (r, Z)$;
- (4) для всех $(q, Z) \in C_2$, где C_2 — множество, построенное алгоритмом 2.16, положим $\delta'(q, e, \bar{Z}) = (p, Z)$;
- (5) для всех $a \in \Sigma$ и $Z \in \Gamma$ положим $\delta'(p, a, Z) = (r, Z)$ и $\delta'(r, a, Z) = (r, Z)$.

Таким образом, P' моделирует P , но если P попадает в зацикливающую конфигурацию, то P' в очередном такте перейдет в состояние r или t в зависимости от того, встречается ли в циклической последовательности конфигураций заключительное состояние. Затем, какова бы ни была входная цепочка, P' переходит в состояние r и остается в нем, ничего не меняя в магазине. Отсюда $L(P') = L(P)$.

Надо показать, что P' — дочитывающий ДМП-автомат. Правила (3)–(5) гарантируют, что условие „дочитывания“ для P' выполняется, если P попадает в зацикливающую конфигурацию. Остается только заметить, что если P находится в незацикливающей конфигурации, то через конечное число тактов он должен либо

- (1) сделать не e -такт, либо
- (2) попасть в конфигурацию, укорачивающую магазин.

Кроме того, ситуация (2) не может повторяться бесконечно, так как в исходной конфигурации магазин имеет конечную длину. Таким образом, либо в конце концов произойдет (1), либо после нескольких повторений (2) P попадет в зацикливающую конфигурацию. Итак, можно заключить, что ДМП-автомат P' дочитывающий. \square

Теперь докажем одно важное свойство ДМП-автоматов, а именно, что класс распознаваемых ими языков замкнут относительно дополнения. В следующем разделе мы увидим, что для класса всех КС-языков это не так.

Теорема 2.23. Если $L = L(P)$ для некоторого ДМП-автомата P , то $\bar{L} = L(P')$ для некоторого ДМП-автомата P' .

Доказательство. По лемме 2.28 можно считать, что P — дочитывающий. Построим P' так, чтобы он моделировал P и между сдвигами входной головки выяснял, попал ли P в допускающее состояние. Так как P' должен допускать дополнение языка $L(P)$, то P' допускает входную цепочку, если P не допускает ее и собирается сдвинуть свою входную головку (и, следовательно, уже не допустит ее и позже).

Формально пусть $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ и $P' = (Q', \Sigma, \Gamma, \delta', q'_0, Z_0, F')$, где

- (1) $Q' = \{[q, i] \mid q \in Q, i \in \{0, 1, 2\}\}$,
- (2) $q'_0 = [q_0, 0]$, если $q_0 \notin F$, и $q'_0 = [q_0, 1]$, если $q_0 \in F$,
- (3) $F' = \{[q, 2] \mid q \in Q\}$.

Состояния вида $[q, 0]$ предизначены для обозначения того, что P не был в заключительном состоянии после последнего не e -такта. Состояния вида $[q, 1]$ указывают, что в данный момент

P перешел в заключительное состояние. Состояния вида $[q, 2]$ используются только для обозначения заключительных состояний. Если P' находится в состоянии $[q, 0]$ и в соответствующий момент моделируемый распознаватель P собирается сделать e -такт, то P' сначала переходит в состояние $[q, 2]$, а затем моделирует P . Таким образом, P' допускает тогда и только тогда, когда P не допускает. Тот факт, что ДМП-автомат P — дочитывающий, гарантирует, что и у P' всегда есть шанс допустить входную цепочку, если P не допускает ее. Формальное определение функции δ таково:

(i) если $q \in Q, a \in \Sigma$ и $Z \in \Gamma$, то

$$\delta'([q, 1], a, Z) = \delta'([q, 2], a, Z) = ([p, i], \gamma)$$

где $\delta(q, a, Z) = (p, \gamma)$, $i = 0$, если $p \notin F$, и $i = 1$, если $p \in F$;

(ii) если $q \in Q, Z \in \Gamma$ и $\delta(q, e, Z) = (p, \gamma)$, то

$$\delta'([q, 1], e, Z) = ([p, 1], \gamma)$$

$$\delta'([q, 0], e, Z) = ([p, i], \gamma)$$

где $i = 0$, если $p \notin F$, и $i = 1$, если $p \in F$;

(iii) если $\delta(q, e, Z) = \emptyset$, то $\delta'([q, 0], e, Z) = ([q, 2], Z)$.

Правило (i) относится к не e -тактам. Вторая компонента состояния правильно принимает значение 0 или 1. Правило (ii) относится к e -тактам, и здесь оно управляет со второй компонентой состояния, как задумано. Правило (iii) позволяет P' допускать входную цепочку в точности тогда, когда P не допускает ее. Формальное доказательство того, что $L(P') = \bar{L}(P)$, мы опускаем. \square

Детерминированные КС-языки обладают и другими важными свойствами: Мы рассмотрим их в упражнениях и в следующем разделе.

УПРАЖНЕНИЯ

2.5.1. Постройте МП-автоматы, допускающие дополнения (относительно $\{a, b\}^*$) следующих языков:

- (a) $\{a^n b^n a^n \mid n \geq 1\}$,
- (б) $\{ww^R \mid w \in \{a, b\}^*\}$,
- (в) $\{a^m b^n a^m b^n \mid m, n \geq 1\}$,
- (г) $\{ww \mid w \in \{a, b\}^*\}$.

Указание: Пусть недетерминированный МП-автомат делает предположение относительно того, почему его входная цепочка не принадлежит данному языку, и проверяет, верно ли это предположение.

2.5.2. Докажите, что МП-автомат из примера 2.31 допускает язык $\{ww^R \mid w \in \{a, b\}^*\}$.

2.5.3. Покажите, что каждый КС-язык допускается МП-автоматом, который за один такт увеличивает длину магазина не более чем на единицу.

2.5.4. Покажите, что каждый КС-язык допускается таким МП-автоматом, что если $(p, \gamma) \in \delta(q, a, Z)$, то либо $\gamma = e$, либо $\gamma = Z$, либо $\gamma = YZ$ для некоторого $Y \in \Gamma$. Указание: Рассмотрите конструкцию леммы 2.21.

2.5.5. Докажите, что каждый КС-язык допускается МП-автоматом, не делающим e -тактов. Указание: Вспомните, что каждый КС-язык порождается грамматикой в нормальной форме Грейбах.

2.5.6. Покажите, что для каждого КС-языка L найдется такой МП-автомат P с двумя состояниями, что $L = L(P)$.

2.5.7. Дополните доказательство леммы 2.23.

2.5.8. Найдите восходящие и нисходящие распознаватели (МП-автоматы) для следующих грамматик:

- (а) $S \rightarrow aSb \mid e$
- (б) $S \rightarrow AS \mid b$
- $A \rightarrow SA \mid a$
- (в) $S \rightarrow SS \mid A$
- $A \rightarrow 0A1 \mid S \mid 01$

2.5.9. Найдите грамматику, порождающую $L(P)$, где

$$P = (\{q_0, q_1, q_2\}, \{a, b\}, \{Z_0, A\}, \delta, q_0, Z_0, \{q_2\})$$

и δ задается равенствами

$$\begin{aligned}\delta(q_0, a, Z_0) &= (q_1, AZ_0) \\ \delta(q_0, a, A) &= (q_1, AA) \\ \delta(q_1, a, A) &= (q_0, AA) \\ \delta(q_1, e, A) &= (q_2, A) \\ \delta(q_2, b, A) &= (q_2, e)\end{aligned}$$

Указание: Для бесполезных нетерминалов строить правила не обязательно.

***2.5.10.** Покажите, что если $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — МП-автомат, то множество цепочек, которые могут появиться в его магазине, регулярно. Иначе говоря, покажите, что множество $\{\alpha \mid (q_0, w, Z_0) \vdash^* (q, x, \alpha) \text{ для некоторых } q, w \text{ и } x\}$ регулярно.

2.5.11. Дополните доказательство леммы 2.26.

2.5.12. Пусть P — МП-автомат, для которого существует такая константа k , что в его магазине всегда не более k символов. Покажите, что $L(P)$ — регулярное множество.

2.5.13. Постройте ДМП-автоматы, допускающие следующие языки:

- (а) $\{0^i 1^j \mid j \leq i\}$,
- (б) $\{w \mid w \text{ состоит из равного числа символов } a \text{ и } b\}$,
- (в) $L(G_0)$, где G_0 — обычная грамматика, определяющая простые арифметические выражения.

2.5.14. Покажите, что ДМП-автомат из примера 2.36 допускает язык $\{ww^R \mid w \in \{a, b\}^+\}$.

2.5.15. Покажите, что если конструкцию леммы 2.21 применить к расширенному ДМП-автомату, то получится ДМП-автомат.

2.5.16. Докажите, что P и P' в лемме 2.27 допускают один и тот же язык.

2.5.17. Докажите, что шаг (2) алгоритма 2.16 действительно отличает C_1 от C_2 .

2.5.18. Дополните доказательство теоремы 2.23.

2.5.19. Определенные нами МП-автоматы делают такт независимо от входа только тогда, когда при этом не сдвигается входная головка. Можно ослабить это ограничение и позволить обозреваемому входному символу влиять на то, что делается в данном такте, и тогда, когда входная головка остается неподвижной. Покажите, что при таком расширении класса МП-автоматов они все еще допускают только КС-языки.

***2.5.20.** Можно еще более расширить класс МП-автоматов, позволив входной головке двигаться по ленте в обе стороны и снабдив входную ленту концевыми маркерами. Назовем такой автомат 2МП-автоматом (двусторонним МП-автоматом), а если он детерминированный, то 2ДМП-автоматом. Покажите, что следующие языки распознаются 2ДМП-автоматами:

- (а) $\{a^n b^n c^n \mid n \geq 1\}$,
- (б) $\{ww \mid w \in \{a, b\}^*\}$,
- (в) $\{a^{2^n} \mid n \geq 1\}$.

2.5.21. Покажите, что 2МП-автомат может распознать язык $\{wxw \mid w, x \in \{0, 1\}^*\}$.

Открытые проблемы

2.5.22. Существует ли язык, который допускается 2МП-автоматом, но не допускается 2ДМП-автоматом?

2.5.23. Существует ли КС-язык, который не допускается 2ДМП-автоматом?

Упражнения на программирование

2.5.24. Напишите программу, моделирующую детерминированный МП-автомат.

***2.5.25.** Придумайте язык программирования, пригодный для задания МП-автоматов. Постройте компилятор для Вашего языка. Исходная программа в этом языке должна определять МП-автомат P . Объектная программа должна описывать распознаватель, разумным способом моделирующий поведение P на входных цепочках w .

2.5.26. Напишите программу, которая в качестве входа воспринимает КС-грамматику и строит для нее недетерминированный нисходящий (или восходящий) распознаватель.

Замечания по литературе

Важность магазинов (известных также под названием стеков) в процессах обработки языков была осознана в начале 1950-х годов. Эттигер [1961] и Шютценберже [1963] первыми formalизовали понятие автомата с магазинной памятью. Эквивалентность МП-автоматов и КС-грамматик была показана Хомским [1962] и Эви [1963].

Двусторонние МП-автоматы изучались Хартмансом и др. [1965], Грэем и др. [1967], Ахо и др. [1968], Куком [1971].

2.6. СВОЙСТВА КОНТЕКСТНО-СВОБОДНЫХ ЯЗЫКОВ

В этом разделе мы исследуем некоторые из основных свойств КС-языков. Упоминаемые здесь результаты на самом деле образуют малую долю огромного богатства знаний о КС-языках. В частности, мы рассмотрим некоторые операции, относительно которых замкнут класс КС-языков, некоторые результаты о разрешимости и кое-что о неоднозначных КС-грамматиках и языках.

2.6.1. Лемма Огдена

Начнем с доказательства одной теоремы (леммы Огдена) о КС-грамматиках, из которой можно будет извлечь ряд результатов о КС-языках и, в частности, „лемму о разрастании“ для КС-языков.

Определение. Позицией в цепочке длины k назовем такое целое число i , что $1 \leq i \leq k$. Будем говорить, что символ a занимает позицию i (или i -ю позицию) в цепочке w , если $w = w_1 a w_2$ и $|w_1| = i - 1$. Например, символ a занимает третью позицию в цепочке $baacc$.

Теорема 2.24. Для каждой КС-грамматики $G = (N, \Sigma, P, S)$ существует такое целое число $k \geq 1$, что если $z \in L(G)$, $|z| \geq k$ и в цепочке z выделены k или более различных позиций, то z можно записать в виде $uwxy$, причем

- (1) w содержит хотя бы одну выделенную позицию;
- (2) либо u и v обе содержат выделенные позиции, либо x и y обе содержат выделенные позиции;
- (3) uw содержит не более k выделенных позиций;
- (4) существует такой нетерминал A , что

$$S \Rightarrow_G^+ uAy \Rightarrow_G^+ uvAxy \Rightarrow_G^+ \dots \Rightarrow_G^+ uv^i Ax^i y \Rightarrow_G^+ uv^i wx^i y$$

для всех $i \geq 0$ (в случае $i = 0$ вывод имеет вид $S \Rightarrow_G^+ uAy \Rightarrow_G^+ uw$).

Доказательство. Пусть $m = \#N$ и l — длина самой длинной из правых частей правил множества P . Выберем $k = l^{2m+3}$ и рассмотрим дерево вывода T некоторой цепочки $z \in L(G)$, где $|z| \geq k$. Пусть в цепочке z выделены по крайней мере k позиций. Заметим, что T должно содержать хотя бы один путь длины, не меньшей $2m+3$. Выделим листья дерева T , которые в кроне z дерева T занимают выделенные позиции.

Назовем вершину n дерева T ветвящейся, если среди ее прямых потомков есть хотя бы два, скажем n_1 и n_2 , таких, что среди потомков каждого из них есть выделенные листья.

Построим путь n_1, n_2, \dots в дереве T следующим образом:

- (1) n_1 — корень дерева T ;
- (2) если мы нашли n_i и только один прямой потомок этой вершины имеет выделенные листья среди своих потомков (т. е. n_i — неветвящаяся вершина), то возьмем в качестве n_{i+1} этого прямого потомка;
- (3) если n_i — ветвящаяся вершина, то выберем в качестве n_{i+1} того прямого потомка вершины n_i , который имеет среди своих потомков наибольшее число выделенных листьев (если таких прямых потомков несколько, выберем самый правый, но можно взять и любой);
- (4) если n_i — лист, то путь окончен.

Пусть n_1, n_2, \dots, n_p — путь, построенный описанным способом. Простой индукцией по i можно показать, что если среди вершин n_1, \dots, n_i есть r ветвящихся, то n_{i+1} имеет по крайней мере l^{2m+3-r} выделенных потомков. Базис, $i = 0$, тривиален: $r = 0$ и n_1 имеет по крайней мере $k = l^{2m+3}$ выделенных потомков. Для доказательства шага индукции заметим, что если n_i — неветвящаяся вершина, то n_i и n_{i+1} имеют одно и то же число выделенных потомков, и что если n_i — ветвящаяся вершина, то n_{i+1} имеет по крайней мере $(1/l)$ -ю часть выделенных потомков вершины n_i .

Так как n_i имеет l^{2m+3} выделенных потомков, то путь n_i, \dots, n_p содержит по крайней мере $2m+3$ ветвящихся вершин. Кроме того, n_p — лист, и потому он не является ветвящейся вершиной. Следовательно, $p > 2m+3$.

Пусть $b_1, b_2, \dots, b_{2m+3}$ — это последние $2m+3$ вершины, принадлежащие пути n_1, \dots, n_p . Назовем b_i левой ветвящейся вершиной, если прямой потомок вершины b_i , не принадлежащий этому пути, имеет выделенного потомка слева от n_p . В противном случае будем называть b_i правой ветвящейся вершиной.

Предположим, что по крайней мере $m+2$ вершины из b_1, \dots, b_{2m+3} левые ветвящиеся. Случай, когда по крайней мере $m+2$ вершины правые ветвящиеся, исследуется аналогично.

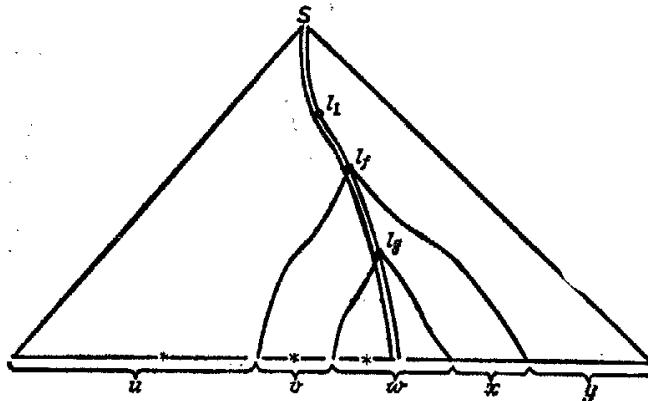


Рис. 2.17. Дерево вывода T .

Пусть l_1, l_2, \dots, l_{m+2} — последние $m+2$ левые ветвящиеся вершины последовательности b_1, \dots, b_{2m+3} . Так как $\#N = m$, то среди l_1, \dots, l_{m+2} можно найти две вершины, скажем l_f и l_g , с одной и той же меткой, скажем A , и $f < g$. Эта ситуация изображена на рис. 2.17. Двойной линией показан путь n_1, \dots, n_p , звездочки указывают выделенные листья (кроме этих могут быть и другие).

Если удалить всех потомков вершины l_j , то получится дерево вывода с короной uAy , где u состоит из листьев, расположенных слева от l_j , а y — из листьев, расположенных справа. Таким образом, $S \Rightarrow^+ uAy$. Если мы рассмотрим поддерево с корнем l_j , из которого удалены потомки вершины l_g , то увидим, что $A \Rightarrow^+ vXx$, где v и x — части короны этого под дерева, состоящие из листьев, расположенных соответственно слева и справа от l_g . Наконец, пусть w — корона под дерева с корнем l_g . Тогда $A \Rightarrow^+ w$ и, значит, $z = uwxy$.

Объединяя эти выводы, получаем $S \Rightarrow^+ uAy \Rightarrow^+ uwu$ и для всех $i \geq 1$

$$S \Rightarrow^+ uAy \Rightarrow^+ uvAx y \Rightarrow^+ uv^2Ax^2y \Rightarrow^+ \dots \Rightarrow^+ uv^iAx^iy \Rightarrow^+ uv^iwx^iy$$

Таким образом, условие (4) выполнено. Кроме того, цепочка u имеет хотя бы одну выделенную позицию, которую занимает потомок некоторого прямого потомка вершины l_1 . Аналогично цепочка v имеет хотя бы одну выделенную позицию, которую занимает потомок вершины l_j . Следовательно, условие (2) тоже выполнено. Условие (1) выполняется потому, что цепочка w имеет выделенную позицию, а именно ту, которую занимает n_{ν} .

Чтобы проверить выполнение условия (3), состоящего в том, что цепочка u_{\max} имеет не более k выделенных позиций, заметим, что цепочка b_1 , будучи $(2m+3)$ -й ветвящейся вершиной от конца пути n_1, \dots, n_p , имеет не более k выделенных позиций. Так как l_f — потомок вершины b_1 , отсюда непосредственно следует нужный результат.

Надо было бы рассмотреть еще противоположный случай, когда по крайней мере $m+2$ вершин из b_1, \dots, b_{2m+3} правые ветвящиеся. Однако здесь можно рассуждать по симметрии, и в итоге мы обнаружим, что условие (2) выполняется, так как каждая из цепочек x и y имеет выделенные позиции. \square

Докажем важное следствие леммы Огдена, которое иногда называют леммой о разрастании для КС-языков.

Следствие. Пусть L — КС-язык. Тогда существует такая константа k , что если $|z| \geq k$ и $z \in L$, то z можно представить в виде $z = uvwx$, где $uv \neq e$, $|vwx| \leq k$ и $uv^iwx^i \in L$ для всех $i \geq 0$.

Доказательство. В теореме 2.24 взять какую-нибудь КС-грамматику для языка L и считать все позиции во всех цепочках выделенными. \square

Именно этим следствием из теоремы 2.24 мы будем чаще всего пользоваться при доказательстве того, что некоторые языки не контекстно-свободны. Самой теоремой 2.24 мы воспользуемся, когда в разд. 2.6.5 речь пойдет о существенной неоднозначности КС-языков.

Пример 2.38. С помощью леммы о разрастании покажем, что язык $L = \{a^{n^2} \mid n \geq 1\}$ не является КС-языком. Если бы он был КС-языком, то нашлось бы такое число k , что если $n^2 \geq k$, то $a^{n^2} = uxxy$, где цепочки u и x не могут быть обе пустыми и $|ux| \leq k$. Пусть, в частности, $n = k$. Так как $k^2 \geq k$, то по предположению $uv^2wx^2y \in L$. Но так как $|vwx| \leq k$, то $1 \leq |vx| \leq k$ и $k^2 < |uv^2wx^2y| \leq k^2 + k$. Заметим, что следующий после k^2 квадрат целого числа — число $(k+1)^2 = k^2 + 2k + 1$. Так как

$k^2 + k < k^2 + 2k + 1$, то $|uv^2wx^2y|$ не равно квадрату целого числа. Но по лемме о разрастании $uv^2wx^2y \in L$; получили противоречие. \square

Пример 2.39. Покажем, что язык $L = \{a^n b^n c^n \mid n \geq 1\}$ — не КС-язык. Если бы он был КС-языком, то мы взяли бы константу k , которая определяется в лемме о разрастании. Пусть $z = a^k b^k c^k$. Тогда $z = uwxy$. Так как $|uw| \leq k$, то в цепочке uw не могут быть вхождения каждого из символов a , b и c . Таким образом, цепочка uw , которая по лемме о разрастании принадлежит L , содержит либо k символов a , либо k символов c . Но она не может иметь k вхождений каждого из символов a , b и c , потому что $|uw| < 3k$. Значит, вхождений какого-то из этих символов в uw больше, чем другого, и, следовательно, $uw \notin L$. Полученное противоречие позволяет заключить, что L — не КС-язык. \square

2.6.2. Свойства замкнутости класса КС-языков

Свойства замкнутости часто помогают доказать, что некоторые языки не контекстно-свободны; кроме того, они интересны и с теоретической точки зрения. В этом разделе мы приведем несколько основных свойств замкнутости класса КС-языков.

Определение. Пусть \mathcal{L} — класс языков и язык $L \subseteq \Sigma^*$ принадлежит \mathcal{L} . Допустим, что $\Sigma = \{a_1, \dots, a_n\}$ и языки L_{a_1}, \dots, L_{a_n} принадлежат \mathcal{L} . Класс \mathcal{L} замкнут относительно подстановки, если для любого набора языков $L, L_{a_1}, \dots, L_{a_n}$ язык

$$L' = \{x_1 \dots x_k \mid a_{j_1} \dots a_{j_k} \in L, x_i \in L_{a_{j_1}}, \dots, x_k \in L_{a_{j_k}}\}$$

принадлежит \mathcal{L} .

Пример 2.40. Пусть $L = \{0^n 1^n \mid n \geq 1\}$, $L_0 = \{a\}$ и $L_1 = \{b^m c^m \mid m \geq 1\}$. Тогда результатом подстановки языков L_0 и L_1 в язык L будет язык

$$L' = \{a^n b^m c^m b^m c^m \dots b^m c^m \mid n \geq 1, m_i \geq 1\} \quad \square$$

Теорема 2.25. Класс КС-языков замкнут относительно подстановки.

Доказательство. Пусть $L \subseteq \Sigma^*$ — КС-язык и $\Sigma = \{a_1, \dots, a_n\}$. Пусть $L_a \subseteq \Sigma_a^*$ — КС-язык для каждого $a \in \Sigma$ и L' — результат подстановки языков L_a вместо a в цепочки языка L . Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика языка L и $G_a = (N_a, \Sigma_a, P_a, a')$ —

КС-грамматика языка L_a . Предполагаем, что N и все N_a попарно не пересекаются. Возьмем $G' = (N', \Sigma', P', S)$, где

$$(1) N' = \bigcup_{a \in \Sigma} N_a \cup N;$$

$$(2) \Sigma' = \bigcup_{a \in \Sigma} \Sigma_a;$$

(3) пусть h — гомоморфизм, определенный на $N \cup \Sigma$ и такой, что $h(A) = A$ для всех $A \in N$ и $h(a) = a'$ для $a \in \Sigma$; положим $P' = \{A \rightarrow h(A) \mid A \rightarrow a \in P\} \cup \bigcup_{a \in \Sigma} P_a$.

Итак, P' состоит из правил всех грамматик G_a и правил грамматики G , в которых все терминалы сделаны нетерминалами со штрихами. Пусть $a_{j_1} \dots a_{j_k} \in L$ и $x_i \in L_{a_{j_i}}$ для $1 \leq i \leq k$. Тогда

$S \Rightarrow_G^* a_{j_1}' \dots a_{j_k}' \Rightarrow_{G'}^* x_1 a_{j_1}' \dots a_{j_k}' \Rightarrow_{G'}^* \dots \Rightarrow_{G'}^* x_1 \dots x_k$. Следовательно, $L' \subseteq L(G')$.

Допустим, что $w \in L(G')$, и рассмотрим дерево вывода T цепочки w . Так как N и N_a не пересекаются, каждый лист с меткой, отличной от e , имеет по крайней мере одного предка, помеченного a' , где $a \in \Sigma$. Если удалить из T каждую вершину, у которой есть предок, отличный от неё и помеченный a' для $a \in \Sigma$, то получим дерево вывода T' с короной $a_{j_1}' \dots a_{j_k}'$, где $a_{j_1} \dots a_{j_k} \in L$. Если x_i — корона поддерева дерева T , над которыми доминирует i -й лист дерева T' , то $w = x_1 \dots x_k$ и $x_i \in L_{a_{j_i}}$. Таким образом, $L(G') = L'(G)$. \square

Следствие. Класс КС-языков замкнут относительно (1) объединения, (2) конкатенации, (3) итерации, (4) позитивной итерации и (5) гомоморфизма.

Доказательство. Пусть L_a и L_b — КС-языки.

(1) Подставим L_a вместо a и L_b вместо b в КС-язык $\{a, b\}$.

(2) Подставим L_a вместо a и L_b вместо b в $\{a b\}$.

(3) Подставим L_a вместо a в a^* .

(4) Подставим L_a вместо a в a^+ .

(5) Для гомоморфизма h возьмем $L_a = \{h(a)\}$ и, подставив L_a вместо a в L , получим $h(L)$. \square

Теорема 2.26. Класс КС-языков замкнут относительно пересечения с регулярными множествами.

Доказательство. Нетрудно показать, что МП-автомат P и параллельно работающий конечный автомат A можно моделировать подходящим МП-автоматом P' . Этот составной МП-автомат P' прямо моделирует P и изменяет состояние автомата A

каждый раз, когда P делает не e -такт. P' допускает цепочку тогда и только тогда, когда ее допускает P и A находится при этом в заключительном состоянии. Детали доказательства оставляем в качестве упражнения. \square

В отличие от регулярных множеств КС-языки не образуют булеву алгебру множеств.

Теорема 2.27. Класс КС-языков не замкнут относительно пересечения и дополнения.

Доказательство. $L_1 = \{a^n b^n c^i \mid n \geq 1, i \geq 1\}$ и $L_2 = \{a^i b^n c^n \mid i \geq 1, n \geq 1\}$ — КС-языки. Однако $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 1\}$ — не КС-язык (см. пример 2.39). Таким образом, класс КС-языков не замкнут относительно пересечения.

Отсюда можно также заключить, что класс КС-языков не замкнут относительно дополнения. В самом деле, в силу закона де Моргана любой класс языков, замкнутый относительно объединения и дополнения, должен быть замкнут относительно пересечения. Но по следствию из теоремы 2.25 класс КС-языков замкнут относительно объединения. \square

Существует много операций, относительно которых замкнут класс КС-языков. Некоторые из них приведены в упражнениях. В заключение этого раздела дадим несколько примеров применения свойств замкнутости к доказательству того, что некоторые множества не являются КС-языками.

Пример 2.41. $L = \{ww \mid w \in \{a, b\}^+\}$ не КС-язык. Допустим, что это не так. Тогда по теореме 2.26 язык $L' = L \cap a^+ b^+ a^+ b^+ = \{a^m b^n a^m b^n \mid m, n \geq 1\}$ контекстно-свободен. Но в упр. 2.6.3(д) утверждается, что L' не КС-язык. \square

Пример 2.42. $L = \{ww \mid w \in \{c, f\}^+\}$ не КС-язык. Пусть h — такой гомоморфизм, что $h(c) = a$ и $h(f) = b$. Тогда $h(L) = \{ww \mid w \in \{a, b\}^+\}$ не КС-язык (см. предыдущий пример). Так как класс КС-языков замкнут относительно гомоморфизмов (следствие из теоремы 2.25), то L не КС-язык. \square

Пример 2.43. Алгол не является КС-языком. Рассмотрим следующий класс программ Алгола:

$$L = \{\text{begin integer } w; w := 1; \text{ end} \mid w \in \{c, f\}^+\}$$

Пусть L_A — множество всех правильных программ Алгола и R — регулярное множество, обозначаемое регулярным выражением

$$\text{begin integer } (c+f)^+; (c+f)^+ := 1; \text{ end}$$

Тогда $L = L_A \cap R$. Наконец, пусть h — такой гомоморфизм, что $h(c) = c$, $h(f) = f$ и $h(X) = e$ в остальных случаях. Тогда $h(L) = \{ww \mid w \in \{c, f\}^+\}$.

Следовательно, если L_A — КС-язык, то $h(L_A \cap R)$ — тоже КС-язык. Однако мы знаем, что язык $h(L_A \cap R)$ не контекстно-свободен, и поэтому заключаем, что множество L_A всех правильных программ Алгола не является КС-языком. \square

Пример 2.43 показывает, что язык программирования, требующий описания идентификаторов, длина которых может быть произвольно большой, не контекстно-свободен. Однако в компиляторе идентификаторы обычно обрабатываются лексическим анализатором и свертываются в лексемы прежде, чем достигают синтаксического анализатора. Поэтому язык, который должен распознаваться синтаксическим анализатором, обычно можно считать контекстно-свободным.

В Алголе и многих других языках встречаются и другие явления, характерные для не контекстно-свободных языков. Например, каждая процедура имеет одно и то же число аргументов в каждом месте, где она упоминается. Поэтому можно показать, что подаваемый на вход синтаксического анализатора язык не контекстно-свободен, отобразив множество программ с тремя вызовами одной и той же процедуры на не контекстно-свободный язык $\{0^n 10^n 10^n \mid n \geq 0\}$. Обычно, однако, для проверки того, что число аргументов процедуры не противоречит ее определению, используется некоторый процесс, не входящий в собственно синтаксический анализ.

2.6.3. Результаты о разрешимости

Мы уже видели, что для КС-грамматик проблема пустоты разрешима. Алгоритм 2.7 получает на входе КС-грамматику G и выясняет, пуст или нет язык $L(G)$.

Рассмотрим проблему принадлежности для КС-грамматик. Нам надо найти алгоритм, который по данной КС-грамматике $G = (N, \Sigma, P, S)$ и цепочке $w \in \Sigma^*$ выясняет, принадлежит ли w языку $L(G)$. Получение эффективного алгоритма, решающего эту проблему, внесло бы существенный вклад в содержание гл. 4—7. С чисто теоретической точки зрения можно сразу сказать, что проблема принадлежности для КС-грамматик разрешима: с помощью преобразований, указанных в разд. 2.4.2, всегда можно превратить G в эквивалентную приведенную КС-грамматику G' , а приведенные КС-грамматики (без учета пустой цепочки) контекстно-зависимы, так что можно применить общий грубый алгоритм, решающий проблему принадлежности (см. упр. 2.1.18).

Рассмотрим проблему эквивалентности для КС-грамматик. К сожалению, эта проблема неразрешима. Мы докажем, что не

существует алгоритма, который по двум данным КС-грамматикам G_1 и G_2 мог бы определить, совпадают ли языки $L(G_1)$ и $L(G_2)$. На самом деле будет доказано, что не существует даже алгоритма, который по КС-грамматике G_1 и праволинейной грамматике G_2 выяснял бы, выполняется ли равенство $L(G_1) = L(G_2)$. Как и для большинства других неразрешимых проблем, мы покажем, что если бы проблема эквивалентности для КС-грамматик была разрешима, то была бы разрешима и проблема соответствий Поста. По частному случаю проблемы соответствий мы будем строить два естественно связанных с ним КС-языка.

Определение. Пусть $C = (x_1, y_1), \dots, (x_n, y_n)$ — частный случай проблемы соответствий над алфавитом Σ и $I = \{1, 2, \dots, n\}$, $I \cap \Sigma = \emptyset$. Положим $L_C = \{x_{i_1} x_{i_2} \dots x_{i_m} i_m i_{m-1} \dots i_1 | i_1, \dots, i_m \in I, m \geq 1\}$ и $M_C = \{y_{i_1} y_{i_2} \dots y_{i_m} i_m i_{m-1} \dots i_1 | i_1, \dots, i_m \in I, m \geq 1\}$.

Лемма 2.29. Пусть $C = (x_1, y_1), \dots, (x_n, y_n)$ — частный случай проблемы соответствий над алфавитом Σ , где $\Sigma \cap \{1, 2, \dots, n\} = \emptyset$. Тогда

(1) можно найти расширенные ДМП-автоматы, допускающие L_C и M_C ,

(2) $L_C \cap M_C = \emptyset$ тогда и только тогда, когда C не имеет решения.

Доказательство. (1) Легко построить расширенный ДМП-автомат (верхняя часть его магазина расположена справа), который все входные символы, принадлежащие Σ , переносит в магазин, а когда на входе появляются символы из $\{1, 2, \dots, n\}$, он удаляет из магазина x_i , если на входе появилось число i . Если на верху магазина в этот момент не x_i , то ДМП-автомат останавливается. Кроме того, с помощью управляющего устройства с конечной памятью он проверяет, принадлежит ли входная цепочка множеству $\Sigma^+ \{1, \dots, n\}^+$, и допускает ее, если все символы множества Σ удалены из магазина. Таким образом допускается L_C . Аналогично можно найти расширенный ДМП-автомат, допускающий M_C .

(2) Если язык $L_C \cap M_C$ содержит цепочку $w i_m \dots i_1$, где $w \in \Sigma^+$, то ясно, что w — решая последовательность. Если $x_{i_1} \dots x_{i_m} = y_{i_1} \dots y_{i_m} = w$, то $w i_m \dots i_1 \in L_C \cap M_C$. \square

Вернемся к проблеме эквивалентности для КС-грамматик. Нам понадобятся еще два языка, связанные с частным случаем проблемы соответствий.

Определение. Пусть $C = (x_1, y_1), \dots, (x_n, y_n)$ — частный случай проблемы соответствий над алфавитом Σ и $I = \{1, \dots, n\}$,

причем $\Sigma \cap I = \emptyset$. Положим $Q_C = \{w \# w^R | w \in \Sigma^+ I^+\}$, где $\#\notin \Sigma \cup I$ и $P_C = L_C \# M_C^R$.

Лемма 2.30. Пусть C — определенная выше последовательность. Тогда

(1) можно найти расширенные ДМП-автоматы, допускающие Q_C и P_C ,

(2) $Q_C \cap P_C = \emptyset$ тогда и только тогда, когда C не имеет решения.

Доказательство. (1) ДМП-автомат, допускающий Q_C , построить легко. Что касается P_C , то в силу леммы 2.29 существует ДМП-автомат, скажем M_1 , допускающий L_C . Найти ДМП-автомат M_2 , допускающий M_C^R , не намного труднее; он переносит со входа в магазин символы из I и затем сравнивает их с той частью входной цепочки, которая принадлежит Σ^+ . Поэтому можно построить ДМП-автомат M_3 , моделирующий M_1 , потом проверяющий, есть ли $\#$, и, наконец, моделирующий M_2 .

(2) Если $uv \# wx \in Q_C \cap P_C$, где $u, x \in \Sigma^+$ и $v, w \in I^+$, то $u = x^R$ и $v = w^R$, поскольку $uv \# wx \in Q_C$. С другой стороны, u — решая последовательность, так как $uv \# wx \in P_C$. Таким образом, C имеет решение. Обратно, если $x_{i_1} \dots x_{i_m} = y_{i_1} \dots y_{i_m}$, то $x_{i_1} \dots x_{i_m} i_m \dots i_1 \# i_1 \dots i_m x_{i_m}^R \dots x_{i_1}^R \in Q_C \cap P_C$. \square

Лемма 2.31. Пусть C — определенная выше последовательность. Тогда

(1) можно найти КС-грамматику для языка $\overline{Q_C} \cup \overline{P_C}$,

(2) $\overline{Q_C} \cup \overline{P_C} = (\Sigma \cup I)^*$ тогда и только тогда, когда C не имеет решения.

Доказательство. (1) В силу замкнутости класса детерминированных КС-языков относительно дополнения (теорема 2.23) можно найти ДМП-автоматы для языков $\overline{Q_C}$ и $\overline{P_C}$. В силу эквивалентности КС-грамматик и МП-автоматов (лемма 2.26) можно найти для этих языков КС-грамматики. В силу замкнутости класса КС-языков относительно объединения можно найти КС-грамматику для $\overline{Q_C} \cup \overline{P_C}$.

(2) Непосредственное следствие леммы 2.30(2) и закона де Моргана. \square

Теперь докажем, что проблема, порождающая ли две КС-грамматики один и тот же язык, неразрешима. Фактически будет доказано более сильное утверждение: эта проблема неразрешима, даже если одна из грамматик праволинейная.

Теорема 2.28. Не существует алгоритма, который для КС-грамматики G_1 и праволинейной грамматики G_2 отвечал бы на вопрос: „ $L(G_1) = L(G_2)?$ “

Доказательство. Если бы такой алгоритм существовал, то можно было бы следующим образом решать проблему соответствий Поста:

(1) По данному частному случаю C построить КС-грамматику G_1 , порождающую $\overline{Q_C} \cup \overline{P_C}$ (по лемме 2.31), и праволинейную грамматику G_2 , порождающую регулярное множество $(\Sigma \cup I)^*$, где Σ — алфавит последовательности C , n — ее длина и $I = \{1, \dots, n\}$. Возможно, придется сначала переименовать некоторые символы, но это не повлияет на наличие или отсутствие решения.

(2) С помощью гипотетического алгоритма получить ответ на вопрос: „ $L(G_1) = L(G_2)?$ “ По лемме 2.31 (2) это равенство выполняется тогда и только тогда, когда C не имеет решения. \square

Так как существуют алгоритмы, преобразующие КС-грамматику в эквивалентный МП-автомат и обратно, то из теоремы 2.28 следует также неразрешимость таких проблем: распознают ли два МП-автомата (или МП-автомат и конечный автомат) один и тот же язык; распознает ли данный МП-автомат множество, обозначаемое данным регулярным выражением, и т. д.

2.6.4. Свойства детерминированных КС-языков

Класс детерминированных КС-языков замкнут лишь относительно малого числа из тех операций, относительно которых замкнут класс всех КС-языков. Мы уже знаем, что класс детерминированных КС-языков замкнут относительно дополнения. Так как $L_1 = \{a^i b^j c^j \mid i, j \geq 1\}$ и $L_2 = \{a^i b^j c^k \mid i, j, k \geq 1\}$ — детерминированные КС-языки, а $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 1\}$ не КС-язык (пример 2.39), то отсюда вытекают сразу два свойства незамкнутости:

Теорема 2.29. Класс детерминированных КС-языков не замкнут относительно пересечения и объединения.

Доказательство. Незамкнутость относительно пересечения непосредственно следует из сказанного перед формулировкой теоремы. Незамкнутость относительного объединения следует из закона де Моргана и замкнутости относительно дополнения. \square

Приведем пример, показывающий, что детерминированные КС-языки образуют собственный подкласс КС-языков.

Пример 2.44. Легко показать, что дополнение языка $L = \{a^n b^n c^n \mid n \geq 1\}$ — КС-язык. Цепочка w принадлежит \bar{L} тогда

и только тогда, когда выполняются одно или несколько из следующих условий:

- (1) $w \notin a^+ b^+ c^+$,
- (2) $w = a^i b^j c^k$ и $i \neq j$,
- (3) $w = a^i b^j c^k$ и $j \neq k$.

Множество, удовлетворяющее условию (1), регулярно, каждое из множеств, удовлетворяющих условиям (2) или (3) — КС-язык, в чем легко убедиться, построив распознающие их недетерминированные МП-автоматы. Так как класс КС-языков замкнут относительно объединения, то \bar{L} — КС-язык.

Но если бы \bar{L} был детерминированным КС-языком, то по теореме 2.23 L был бы таким же. Но L даже не КС-язык. \square

Для детерминированных КС-языков справедливы те же положительные результаты о разрешимости, что и для КС-языков, т. е. по данному ДМП-автомату P можно определить, пуст ли язык $L(P)$, и по данной входной цепочке w можно легко определить, принадлежит ли w языку $L(P)$.

Кроме того, по данному детерминированному МП-автомату P и регулярному множеству R можно определить, совпадают ли множества $L(P)$ и R , так как $L(P) = R$ тогда и только тогда, когда $(L(P) \cap \bar{R}) \cup (\bar{L(P)} \cap R) = \emptyset$. Легко видеть, что язык $(L(P) \cap \bar{R}) \cup (\bar{L(P)} \cap R)$ контексто-свободен. Другие результаты, касающиеся разрешимости, приведены в упражнениях.

2.6.5. Неоднозначность

Напомним, что КС-грамматика $G = (N, \Sigma, P, S)$ неоднозначна, если существует цепочка $w \in L(G)$, имеющая два или более различных деревьев выводов, или, что эквивалентно, если цепочка w имеет два различных левых или правых вывода.

Если грамматика используется для определения языка программирования, желательно, чтобы она была однозначной. В противном случае программист и компилятор могут по-разному понять смысл некоторых программ.

Пример 2.45. Самый известный пример неоднозначности в языках программирования — это, по-видимому, „кочующее else“. Рассмотрим грамматику G с правилами

$$S \rightarrow \text{if } b \text{ then } S \text{ else } S \mid \text{if } b \text{ then } S \mid a$$

Эта грамматика неоднозначна, так как предложение

$$\text{if } b \text{ then if } b \text{ then } a \text{ else } a$$

имеет два дерева вывода, показанные на рис. 2.18. Дерево a предполагает интерпретацию

$\text{if } b \text{ then } (\text{if } b \text{ then } a) \text{ else } a$

тогда как дерево b дает

$\text{if } b \text{ then } (\text{if } b \text{ then } a \text{ else } a) \quad \square$

Нам хотелось бы иметь алгоритм, который по произвольной КС-грамматике выясняет, однозначна ли она, но, к сожалению, такого алгоритма не существует.

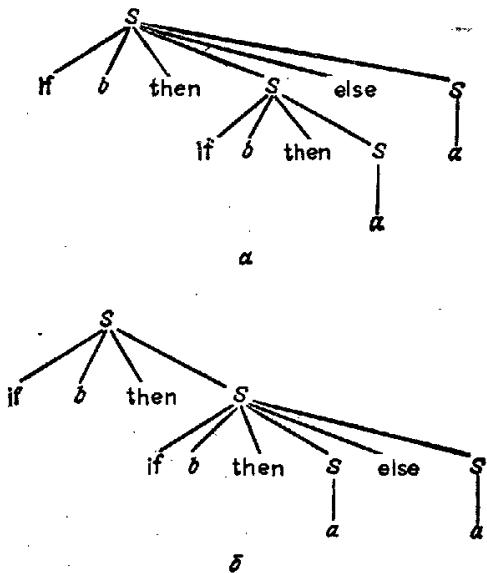


Рис. 2.18. Два дерева вывода.

Теорема 2.30. Проблема, однозначна ли КС-грамматика G , неразрешима.

Доказательство. Пусть $C = (x_1, y_1), \dots, (x_n, y_n)$ — частный случай проблемы соответствий над алфавитом Σ . Возьмем КС-грамматику $G = (\{S, A, B\}, \Sigma \cup I, P, S)$, где $I = \{1, 2, \dots, n\}$ и P содержит правила

$$S \rightarrow A | B$$

$$A \rightarrow x_i A_i | x_i i \quad \text{для } 1 \leq i \leq n$$

$$B \rightarrow y_i B_i | y_i i \quad \text{для } 1 \leq i \leq n$$

Нетерминалы A и B порождают соответственно языки L_C и M_C , определенные в разд. 2.6.3. Легко видеть, что не существует цепочки с двумя различными левыми выводами из нетерминала A .

или из нетерминала B . Следовательно, если существует цепочка с двумя различными левыми выводами из S , то один должен начинаться шагом $S \Rightarrow_t A$, а другой — шагом $S \Rightarrow_t B$. Но по лемме 2.29 некоторая цепочка выводится из A и из B тогда и только тогда, когда частный случай C проблемы соответствий Поста имеет решение.

Таким образом, КС-грамматика G неоднозначна тогда и только тогда, когда C имеет решение. Отсюда сразу заключаем, что если бы существовал алгоритм, решающий проблему однозначности для КС-грамматик, то была бы разрешима проблема соответствий. \square

Определенная нами неоднозначность — это свойство грамматики, а не языка. Для некоторых неоднозначных грамматик можно построить эквивалентные им однозначные грамматики.

Пример 2.46. Рассмотрим грамматику и язык из предыдущего примера. Грамматика G неоднозначна потому, что else можно ассоциировать с двумя разными then . По этой причине языки программирования, в которых разрешаются как операторы вида if—then , так и операторы вида if—then—else , могут быть неоднозначными. Неоднозначность можно устраниТЬ, если договориться, что else должно ассоциироваться с последним из предшествующих ему then , как на рис. 2.18, б.

Можно подправить грамматику из примера 2.45, введя два нетерминала S_1 и S_2 , и потребовав, чтобы S_2 порождал операторы вида if—then—else , тогда как S_1 может порождать операторы обоих видов. Правила новой грамматики таковы:

$$\begin{aligned} S_1 &\rightarrow \text{if } b \text{ then } S_1 | \text{if } b \text{ then } S_2 \text{ else } S_1 | a \\ S_2 &\rightarrow \text{if } b \text{ then } S_2 \text{ else } S_2 | a \end{aligned}$$

Тот факт, что слову else предшествует только S_2 , гарантирует появление внутри конструкции then—else либо символа a , либо другого else . Таким образом, структура, изображенная на рис. 2.18, а, не возникнет. В гл. 5 мы изложим детерминированные методы разбора для различных грамматик, в том числе для грамматики этого примера, и там мы сможем доказать, что наша новая грамматика однозначна. \square

Хотя нет общего алгоритма, выясняющего, однозначна ли грамматика, можно указать некоторые встречающиеся в правилах, конструкциях, приводящие к неоднозначности. Поскольку неоднозначные грамматики часто анализируются труднее, чем однозначные, мы назовем здесь несколько наиболее распространенных конструкций такого рода, так что их можно будет распознать на практике.

Приведенная грамматика, содержащая правила $A \rightarrow AA | \alpha$, неоднозначна, так как подцепочка AAA допускает два разных разбора (рис. 2.19).

Эта неоднозначность исчезнет, если вместо правил $A \rightarrow AA | \alpha$ взять правила

$$\begin{aligned} A &\rightarrow AB | B \\ B &\rightarrow \alpha \end{aligned}$$

или правила

$$\begin{aligned} A &\rightarrow BA | B \\ B &\rightarrow \alpha \end{aligned}$$

Другой пример неоднозначности — правило $A \rightarrow A\alpha A$. Пара правил $A \rightarrow \alpha A | A\beta$ тоже создает неоднозначность, так как цепь



Рис. 2.19. Два дерева разбора.

цепочка $\alpha A\beta$ имеет два разных левых вывода $A \Rightarrow \alpha A \Rightarrow \alpha A\beta$ и $A \Rightarrow A\beta \Rightarrow \alpha A\beta$. В качестве более тонкого примера пары правил, из-за которых грамматика может стать неоднозначной, приведем $A \rightarrow \alpha A | \alpha A\beta A$. Другие примеры неоднозначных грамматик можно найти в упражнениях.

КС-язык называется *неоднозначным* (или *существенно неоднозначным*), если он не порождается никакой однозначной КС-грамматикой. С первого взгляда не видно, существуют ли вообще неоднозначные КС-языки, но нашим следующим примером и будет как раз такой язык. Проблема, порождает ли данная КС-грамматика однозначный язык (т. е. существует ли эквивалентная ей однозначная грамматика), на самом деле неразрешима, но для некоторых больших подклассов КС-языков известно, что они однозначны; все придуманные до сих пор языки программирования тоже однозначны. Важнее всего то, что, как мы увидим в гл. 8, каждый детерминированный КС-язык определяется однозначной КС-грамматикой.

Пример 2.47. Пусть $L = \{a^i b^j c^l \mid i = j \text{ или } j = l\}$. Этот КС-язык неоднозначен. Интуитивно это объясняется тем, что цепочки с $i = j$ должны порождаться группой правил, отличных от правил, порождающих цепочки с $j = l$. По крайней мере некоторые из цепочек с $i = j = l$ должны порождаться обоими механизмами.

Одна из КС-грамматик, порождающих L , такова:

$$\begin{aligned} S &\rightarrow AB | DC \\ A &\rightarrow aA | e \\ B &\rightarrow bBc | e \\ C &\rightarrow cC | e \\ D &\rightarrow aDb | e \end{aligned}$$

Ясно, что она неоднозначна.

С помощью леммы Огдена можно показать, что язык L неоднозначен. Пусть G — произвольная КС-грамматика, порождающая L , и k — число, связанное с G (см. теорему 2.24). Можно считать, что $k \geq 3$. Рассмотрим цепочку $z = a^k b^k c^{k+k!}$, в которой выделены все символы a . Ее можно записать в виде $z = uwxy$. Так как w содержит выделенные позиции, то u и v состоят только из символов a . Если x содержит два различных символа, то, конечно, $uv^2wx^2y \notin L$, так что либо $x \in a^*$, либо $x \in b^*$, либо $x \in c^*$.

Если $x \in a^*$, то цепочка uv^2wx^2y имеет вид $a^{k+p} b^k c^{k+k!}$ для некоторого $1 \leq p \leq k$ и потому не принадлежит L . Если $x \in c^*$, то uv^2wx^2y имеет вид $a^{k+p_1} b^k c^{k+k!+p_2}$, где $1 \leq p_1 \leq k$. Эта цепочка тоже не принадлежит L .

Если $x \in b^*$, то $uv^2wx^2y = a^{k+p_1} b^{k+p_2} c^{k+k!}$, где $1 \leq p_1 \leq k$. Если эта цепочка принадлежит L , то либо $p_1 = p_2$, либо $p_1 \neq p_2$ и $p_2 = k!$. В последнем случае цепочка $uv^3wx^3y = a^{k+2p_1} b^{k+2p_2} c^{k+k!}$, конечно, не принадлежит L . Поэтому заключаем, что $p_1 = p_2$. Заметим, что $p_1 = |v|$ и $p_2 = |x|$.

По теореме 2.24 для любого $m \geq 0$ найдется вывод

$$(2.6.1) \quad S \Rightarrow^+ uAy \Rightarrow^+ uv^m Ax^m y \Rightarrow^+ uv^m wx^m y$$

Пусть, в частности, $m = k!/p_1$. Так как $1 \leq p_1 \leq k$, то m — целое число. Тогда $uv^m wx^m y = a^{k+k!} b^{k+k!} c^{k+k!}$.

Для цепочки $a^{k+k!} b^{k+k!} c^{k+k!}$ можно показать по симметрии, что существуют u_1, v_1, w_1, x_1, y_1 , где только u_1 содержит символ a , $v_1 \in b^*$, и что найдется такой нетерминал B , что

$$\begin{aligned} (2.6.2) \quad S &\Rightarrow^+ u_1 By_1 \Rightarrow^+ u_1 v_1^m Bx_1^m y_1 \\ &\Rightarrow^+ u_1 v_1^m w_1 x_1^m y_1 = a^{k+k!} b^{k+k!} c^{k+k!} \end{aligned}$$

Если мы сумеем показать, что этим двум выводам цепочки $a^{k+k!} b^{k+k!} c^{k+k!}$ соответствуют разные деревья, то тем самым докажем, что L — неоднозначный язык, поскольку грамматика G была выбрана произвольно и оказалась неоднозначной.

Допустим, что выводам (2.6.1) и (2.6.2) соответствует одно и то же дерево. Так как A порождает цепочки из символов a и b , а B порождает цепочки из символов b и c , то A (соответственно B) не может быть меткой вершины, которая является

потомком вершины, помеченной B (соответственно A). Следовательно, найдется выводимая цепочка $t_1At_2Bt_3$, где t_1, t_2, t_3 — терминальные цепочки. Для всех i и j цепочка $t_1v^iwx^it_2v^jwx_1t_3$ должна по предположению принадлежать L . Но $|v|=|x|$ и $|v_i|=|x_1|$ и, кроме того, x и x_1 состоят только из символов b , v — из символов a и x_1 — из символов c . Тогда, выбрав i и j равными и достаточно большими, можно считать, что в соответствующей цепочке символов b больше, чем символов a или c . Отсюда заключаем, что грамматика G неоднозначна, а стало быть, и язык L неоднозначен. \square

УПРАЖНЕНИЯ

2.6.1. Пусть L — КС-язык и R — регулярное множество. Покажите, что следующие языки контекстно-свободны:

- (а) $\text{INIT}(L)$;
- (б) $\text{FIN}(L)$;
- (в) $\text{SUB}(L)$;
- (г) L/R ;
- (д) $L \cap R$.

Определения операций (а) — (г) даны перед упр. 2.3.17.

2.6.2. Покажите, что если L — КС-язык и h — гомоморфизм, то $h^{-1}(L)$ — КС-язык. Указание: Пусть P — МП-автомат, допускающий L . Постройте МП-автомат P' , который применяет h по очереди к каждому входному символу, запасает результат в буфере управляющего устройства и моделирует P на символах из буфера. Позаботьтесь о том, чтобы Ваш буфер был конечной длины.

2.6.3. Покажите, что следующие языки не контекстно-свободны:

- (а) $\{a^ib^jc^l \mid j \leq i\}$;
- (б) $\{a^ib^jc^k \mid i < j < k\}$;
- (в) множество цепочек с одинаковым числом символов a , b и c ;
- (г) $\{a^ib^ja^ib^i \mid j \leq i\}$;
- (д) $\{a^mb^n a^mb^n \mid m, n \geq 1\}$;
- (е) $\{a^ib^jc^k \mid$ все числа i, j, k разные};
- (ж) $\{nHa^n \mid n \geq 1$ — число в десятичной записи}. (Эта конструкция представляет холлеритовы поля в Фортране.)

****2.6.4.** Покажите, что каждый КС-язык в однобуквенном алфавите регулярен. Указание: Воспользуйтесь леммой о разрастании.

***2.6.5.** Покажите, что следующие языки не всегда являются КС-языками, когда L — КС-язык:

- (а) $\text{MAX}(L)$;
- (б) $\text{MIN}(L)$;
- (в) $L^{1/2} = \{x \mid xy \in L \text{ для некоторого } y \text{ и } |x|=|y|\}$.

***2.6.6.** Докажите следующую лемму о разрастании для линейных языков. Если L — линейный язык, то найдется такая константа k , что если $z \in L$ и $|z| \geq k$, то $z = uxhy$, где $|uxhy| \leq k$, $ux \neq e$ и $ux^iwx^i y \in L$ для всех i .

2.6.7. Покажите, что язык $\{a^n b^n a^m b^m \mid n, m \geq 1\}$ не линейный.

***2.6.8.** МП-автоматом с одним поворотом называется МП-автомат, который для любой входной цепочки сначала только пишет в магазин, а потом только стирает, т. е. начав стирать символы в магазине, он уже не может сюда записывать (это и есть один поворот в работе с магазином). Покажите, что КС-язык линеен тогда и только тогда, когда он распознается МП-автоматом с одним поворотом.

***2.6.9.** Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Покажите, что следующие языки контекстно-свободны:

- (а) $\{\alpha \mid S \Rightarrow^* \alpha\}$;
- (б) $\{\alpha \mid S \Rightarrow^* \alpha\}$;
- (в) $\{\alpha \mid S \Rightarrow^* \alpha\}$.

2.6.10. Дайте подробное доказательство следствия из теоремы 2.25.

2.6.11. Дополните доказательство теоремы 2.26.

2.6.12. Дайте более формальное описание ДМП-автоматов, фигурирующих в доказательствах лемм 2.29 (I) и 2.30 (I).

***2.6.13.** Покажите, что язык $Q_C \cap P_C$ из разд. 2.6.3 контекстно-свобден тогда и только тогда, когда он пуст.

2.6.14. Покажите, что следующие проблемы для КС-грамматик G неразрешимы:

- (а) $\overline{L(G)}$ — КС-язык?
- (б) $L(G)$ — регулярный язык?
- (в) $L(G)$ — детерминированный КС-язык?

Указание: Используйте упр. 2.6.13 и рассмотрите КС-грамматику для языка $\overline{Q_C} \cup \overline{P_C}$.

2.6.15. Докажите, что проблема, порождает ли контекстно-зависимая грамматика КС-язык, неразрешима.

2.6.16. Пусть G_1 и G_2 — КС-грамматики. Докажите неразрешимость проблемы „ $L(G_1) \cap L(G_2) = \emptyset$ ”

***2.6.17.** Пусть G_1 — КС-грамматика и G_2 — праволинейная грамматика. Докажите

- (а) неразрешимость проблемы „ $L(G_2) \subseteq L(G_1)$?“
 (б) разрешимость проблемы „ $L(G_1) \subseteq L(G_2)$?“

*2.6.18. Пусть P_1 и P_2 — ДМП-автоматы. Докажите неразрешимость следующих проблем:

- (а) $L(P_1) \cup L(P_2)$ — детерминированный КС-язык?
 (б) $L(P_1) \cap L(P_2)$ — детерминированный КС-язык?
 (в) $L(P_1) \subseteq L(P_2)$?
 (г) $L(P_1)^*$ — детерминированный КС-язык?

**2.6.19. Покажите, что для детерминированных МП-автоматов P проблема, регулярен ли язык $L(P)$, разрешима (в противоположность упр. 2.6.14 (б)).

**2.6.20. Пусть L — детерминированный КС-язык и R — регулярное множество. Покажите, что следующие языки являются детерминированными КС-языками;

- (а) LR ;
 (б) L/R ;
 (в) $L \cup R$;
 (г) $\text{MAX}(L)$;
 (д) $\text{MIN}(L)$;
 (е) $L \cap R$.

Указание: В пунктах а, б, д, е пусть P — ДМП-автомат, распознающий L , а M — конечный автомат, распознающий R . Надо показать, что существует ДМП-автомат P' , который моделирует P , но при этом в каждой ячейке магазина хранит такую информацию: „Для каких состояний p распознавателя M и q распознавателя P существует цепочка w , переводящая M из состояния p в заключительное состояние и допускаемая распознавателем P , если он начинает работу в состоянии q и данная ячейка расположена наверху магазина?“ Надо показать, что для каждой ячейки эта информация имеет лишь конечный объем и что P' может следить за ее сохранением, когда магазин растет или убывает. Как только P' построен, четыре упомянутые ДМП-автомата строятся относительно легко.

2.6.21. Покажите, что для детерминированного КС-языка L и регулярного множества R следующие языки могут не быть детерминированными КС-языками:

- (а) RL ;
 (б) $\{x \mid xR \subseteq L\}$;
 (в) $\{x \mid yx \in L \text{ для некоторого } y \in R\}$;
 (г) $h(L)$, где h — гомоморфизм.

2.6.22. Покажите, что если h — гомоморфизм и L — детерминированный КС-язык, то $h^{-1}(L)$ — тоже детерминированный КС-язык.

**2.6.23. Покажите, что если язык $\overline{Q_c} \cup \overline{P_c}$ не пуст, то он неоднозначный КС-язык.

**2.6.24. Докажите, что проблема, порождающая КС-грамматика G неоднозначный язык, неразрешима.

*2.6.25. Покажите, что грамматика из примера 2.46 однозначна.

**2.6.26. Докажите неоднозначность языка $L_1 \cup L_2$, где $L_1 = \{a^m b^n a^m b^n \mid m, n \geq 1\}$ и $L_2 = \{a^n b^m a^m b^n \mid m, n \geq 1\}$.

**2.6.27. Покажите, что КС-грамматика с правилами $S \rightarrow aSbSc \mid aSb \mid bSc \mid d$ неоднозначна. Однозначен ли порождаемый ею язык?

*2.6.28. Покажите, что для ДМП-автомата P проблема, обладает ли язык $L(P)$ префиксным свойством, разрешима. Разрешима ли эта проблема для произвольных КС-грамматик?

Определение. Языком Дика называется КС-язык L , порождаемый грамматикой $G = (\{S\}, \Sigma, P, S)$, где $\Sigma = \{a_1, \dots, a_k, b_1, \dots, b_k\}$ для некоторого $k \geq 1$ и P состоит из правил $S \rightarrow SS \mid a_1 S b_1 \mid a_2 S b_2 \mid \dots \mid a_k S b_k \mid e$.

**2.6.29. Покажите, что для данного алфавита Σ можно найти такие алфавит Σ' , язык Дика $L_D \subseteq \Sigma'^*$ и гомоморфизм h из Σ'^* в Σ^* , что для любого КС-языка $L \subseteq \Sigma^*$ существует регулярное множество R , для которого $h(L_D \cap R) = L$.

*2.6.30. Пусть L — КС-язык и $S(L) = \{i \mid i = |\omega| \text{ для некоторой цепочки } \omega \in L\}$. Покажите, что $S(L)$ можно представить в виде объединения конечного числа арифметических прогрессий.

Определение. Назовем n -вектором набор из n неотрицательных целых чисел. Если $v_1 = (a_1, \dots, a_n)$ и $v_2 = (b_1, \dots, b_n)$ — n -векторы и c — неотрицательное целое число, то $v_1 + v_2 = (a_1 + b_1, \dots, a_n + b_n)$ и $cv_1 = (ca_1, \dots, ca_n)$. Множество S n -векторов называется линейным, если существуют такие n -векторы v_0, \dots, v_k , что $S = \{v \mid v = v_0 + c_1 v_1 + \dots + c_k v_k \text{ для некоторых неотрицательных целых чисел } c_1, \dots, c_k\}$. Множество n -векторов называется полулинейным, если его можно представить в виде объединения конечного числа линейных множеств.

**2.6.31. Пусть $\Sigma = \{a_1, \dots, a_n\}$. Обозначим через $\#_b(x)$ число вхождений символа b в цепочку x . Покажите, что для каждого КС-языка $L \subseteq \Sigma^*$ множество $\{(\#_{a_1}(w), \#_{a_2}(w), \dots, \#_{a_n}(w)) \mid w \in L\}$ полулинейно.

Определение. Индексом (или активной емкостью) вывода называется наибольшее из чисел вхождений нетерминалов в цепочки, образующие этот вывод. Индексом $I(w)$ цепочки $w \in L(G)$ называется наименьший из индексов всевозможных выводов этой цепочки в G . Индекс грамматики G — это $I(G) = \max\{I(w) \mid w \in L(G)\}$. Индекс КС-языка L — это $I(L) = \min\{I(G) \mid L = L(G)\}$.

**2.6.32. Покажите, что индекс грамматики G с правилами

$$S \rightarrow SS \mid 0S1 \mid e$$

бесконечен. Покажите, что индекс языка $L(G)$ бесконечен.

*2.6.33. КС-грамматика $G = (N, \Sigma, P, S)$ называется *граммойкой с самовставлением*, если $A \Rightarrow^* uAv$ для некоторых u и v из Σ^* (обе цепочки u и v непустые). Покажите, что КС-язык L не регулярен тогда и только тогда, когда все порождающие его грамматики являются грамматиками с самовставлением.

Определение. Пусть \mathcal{L} — класс языков и языки $L_1 \subseteq \Sigma_1^*$ и $L_2 \subseteq \Sigma_2^*$ принадлежат \mathcal{L} . Пусть a и b — новые символы, не входящие в $\Sigma_1 \cup \Sigma_2$. Говорят, что класс \mathcal{L} замкнут относительно

- (1) *маркированного объединения*, если $aL_1 \cup bL_2 \in \mathcal{L}$;
- (2) *маркированной конкатенации*, если $L_1 a L_2 \in \mathcal{L}$;
- (3) *маркированной итерации*, если $(aL_1)^* \in \mathcal{L}$.

2.6.34. Покажите, что класс детерминированных КС-языков замкнут относительно маркированных объединения, конкатенации и итерации.

2.6.35. Пусть $G = (N, \Sigma, P, S)$ — (не обязательно контекстно-свободная) грамматика, где каждое правило из P имеет вид $xAy \rightarrow xy\gamma$, причем $x, y \in \Sigma^$, $A \in N$ и $\gamma \in (N \cup \Sigma)^*$. Покажите, что $L(G)$ — КС-язык.

**2.6.36. Пусть $G_1 = (N_1, \Sigma_1, P_1, S_1)$ и $G_2 = (N_2, \Sigma_2, P_2, S_2)$ — КС-грамматики. Докажите неразрешимость проблем „ $\{\alpha \mid S_1 \Rightarrow^*_G \alpha\} = \{\beta \mid S_2 \Rightarrow^*_G \beta\}$ “ и „ $\{\alpha \mid S_1 \Rightarrow^*_G \alpha\} = \{\beta \mid S_2 \Rightarrow^*_G \beta\}$ “.

Открытая проблема

2.6.37. Разрешима ли для ДМП-автоматов P_1 и P_2 проблема эквивалентности, т. е. „ $L(P_1) = L(P_2)$ “?

Проблемы для исследования

2.6.38. Разработайте методы доказательства того, что некоторые грамматики однозначны. В силу теоремы 2.30 нельзя найти метод, который работал бы для произвольной однозначной грамматики. Однако хорошо было бы получить технику, применимую к широкому классу КС-грамматик.

2.6.39. Родственная область исследования — поиск больших классов однозначных КС-языков. Следует иметь в виду, что, как будет показано в гл. 8, детерминированные КС-языки образуют один из таких классов.

2.6.40. Найдите преобразования, превращающие грамматики из некоторых классов неоднозначных грамматик в эквивалентные однозначные.

Замечания по литературе

Мы не будем пытаться указать здесь все многочисленные работы, касающиеся контекстно-свободных языков. В книгах Хопкрофта и Ульмана [1969], С. Гинзбурга [1966], Гросса и Лантеиа [1970]¹⁾ и в работе Бука [1970] приведены большие библиографии по теории КС-языков.

Теорема 2.24 (лемма Огдена) взята из работы Огдена [1968]. Бар-Хиллел и др. [1961] доказали несколько из основных теорем о свойствах замкнутости и алгоритмических проблемах для КС-языков. Гинзбург и Грэйбах [1966] исследовали многие из основных свойств детерминированных КС-языков.

Кантор [1962], Флойд [1962a], Хомский и Шютценберже [1963] независимо друг от друга обнаружили, что проблема однозначности для КС-грамматик неразрешима. Существование неоднозначных КС-языков было замечено Париком [1996]. Неоднозначные КС-языки подробно рассматриваются в книгах С. Гинзбурга [1966] и Хопкрофта и Ульмана [1969]¹⁾.

Многие из результатов, сформулированных в упражнениях, опубликованы. Упр. 2.6.19 взято из работы Стирнза [1967]. Конструкции, указанные в упр. 2.6.20, подробно описаны у Хопкрофта и Ульмана [1969]. Теорема из упр. 2.6.29 доказана С. Гинзбургом [1966]. Результат из упр. 2.6.31 известен как теорема Парика и впервые был получен Париком [1966]. Упр. 2.6.32 взято из работы Саломаа [1969b], упр. 2.6.33 — из статьи Хомского [1959a], а результат из упр. 2.6.36 принадлежит Блатнеру [1972]²⁾.

¹⁾ А также в книге Гладкого [1973]. — Прим. перев.

²⁾ Отметим также, что теорема из упр. 2.6.24 впервые доказана Гладким [1965] и независимо Гинзбургом и Уллианом [1966]. — Прим. перев.

3

ТЕОРИЯ ПЕРЕВОДА

Перевод (или трансляция)¹⁾ — это некоторое отношение между цепочками, или, другими словами, это некоторое множество пар цепочек. Компилятор определяет перевод, образуемый парами вида (исходная программа, объектная программа). Если компилятор состоит из трех фаз — лексического анализа, синтаксического анализа и генерации кода, то каждая из них сама является переводом. Как отмечалось в гл. 1, лексический анализ можно рассматривать как перевод, при котором цепочки, представляющие исходные программы, отображаются в цепочки лексем. Синтаксический анализатор отображает цепочки лексем в цепочки, представляющие деревья. Затем генератор кода переводит эти цепочки на машинный язык или язык ассемблера.

В этой главе мы изложим элементарные методы определения перевода. Кроме того, опишем устройства, с помощью которых можно реализовать эти переводы, и алгоритмы, позволяющие автоматически строить такие устройства по описанию перевода.

Сначала мы исследуем переводы с абстрактной точки зрения, а затем рассмотрим применимость моделей перевода к лексическому и синтаксическому анализу. Процесс генерации кода, который служит главным объектом приложения теории перевода, мы в основном изложим в гл. 9.

Вообще, когда создаются большие системы, такие, как компилятор, следует разбить всю систему на части, свойства и поведение которых можно точно определить и понять. Тогда можно сравнивать алгоритмы, реализующие функцию, которую должна выполнять данная составная часть, и выбрать для нее наиболее подходящий алгоритм. Как только компоненты системы выделены и точно описаны, можно устанавливать стандарты, которым должно удовлетворять функционирование компонент, и критерий

для их оценки. Поэтому надо понять способы задания и реализации перевода, прежде чем применять к компиляторам критерии инженерного проектирования.

3.1. ФОРМАЛИЗМЫ, ИСПОЛЬЗУЕМЫЕ ДЛЯ ОПРЕДЕЛЕНИЯ ПЕРЕВОДА

В этом разделе мы изложим два фундаментальных метода определения перевода. Один из них — „схема перевода“, которая представляет собой грамматику, снабженную механизмом, обеспечивающим выход для каждой порождаемой цепочки. В другом методе основную роль играет „преобразователь“, т. е. распознаватель с выходом, который на каждом такте может выдавать цепочку выходных символов ограниченной длины. Сначала мы рассмотрим схемы перевода, основанные на контексто-свободных грамматиках, а потом займемся конечными преобразователями и преобразователями с магазинной памятью.

3.1.1. Перевод и семантика

В гл. 2 изучались только синтаксические аспекты языков. Там мы познакомились с несколькими методами определения правильно построенных цепочек, или предложений, языка. Теперь мы хотим исследовать механизмы, связывающие с каждым предложением (цепочкой) языка другую цепочку, которая должна быть выходом, или результатом перевода, этого предложения. Для обозначения этого отношения между предложениями и соответствующими выходными цепочками, определяющими их „смыслы“, или „значения“, иногда употребляют термин „семантика“.

Определение. Пусть Σ — входной алфавит и Δ — выходной алфавит. *Переводом с языка $L_1 \subseteq \Sigma^*$ на язык $L_2 \subseteq \Delta^*$* назовем отношение T из Σ^* в Δ^* , для которого L_1 — область определения, а L_2 — множество значений.

Если $(x, y) \in T$, то цепочка y называется *выходом* для цепочки x . Заметим, что в общем случае в переводе T для данной входной цепочки может быть более одной выходной цепочки. Однако перевод, предназначенный для описания языка программирования, должен быть функцией, т. е. для каждого входа должно быть не более одного выхода.

Можно привести много примеров переводов. По-видимому, самый примитивный тип перевода — перевод, задаваемый гомоморфизмом.

Пример 3.1. Допустим, что мы хотим перевести каждую греческую букву цепочки из множества Σ^* в ее английское название. Это можно сделать с помощью такого гомоморфизма h , что

¹⁾ В дальнейшем мы будем считать эти слова синонимами, отдавая предпочтение термину „перевод“. — Прим. ред.

(1) $h(a) = a$, если a принадлежит Σ , но не является буквой греческого алфавита;

(2) $h(a)$ определяется табл. 3.1, если a —греческая буква.

Например, переводом предложения $a = \pi r^2$ будет цепочка $a = \text{pi } r^2$. \square

Таблица 3.1

Греческая буква	h	Греческая буква	h
Α α	alpha	Ν ν	nu
Β β	beta	Ξ ξ	xi
Γ γ	gamma	Ο ο	omicron
Δ δ	delta	Π π	pi
Ε ε	epsilon	Ρ ρ	rho
Ζ ζ	zeta	Σ σ	sigma
Η η	eta	Τ τ	tau
Θ θ	theta	Υ υ	upsilon
Ι ι	iota	Φ φ	phi
Κ κ	kappa	Χ χ	chi
Λ λ	lambda	Ψ ψ	psi
Μ μ	mu	Ω ω	omega

Другой пример перевода, полезный при описании одного процесса, часто встречающегося при компиляции,—отображение обычной (инфиксной) записи арифметических выражений в польскую запись.

Определение. Запись обычных (или инфиксных) арифметических выражений без использования скобок называют *польской (бесскобочной) записью*¹). Пусть Θ —множество знаков бинарных операций (например, $\{+, *\}$) и Σ —множество operandов. Определим рекурсивно две формы польской записи, *префиксную* и *постфиксную*:

(1) Если инфиксное выражение E является единственным operandом $a \in \Sigma$, то как префиксная, так и постфиксная польские записи выражения E —это просто a .

(2) Если $E_1 \theta E_2$ —инфиксное выражение, где θ —знак операции, а E_1 и E_2 —инфиксные выражения, operandы для θ , то
 (а) $\theta E'_1 E'_2$ —префиксная польская запись выражения $E_1 \theta E_2$, где E'_1 и E'_2 —префиксные польские записи выражений E_1 и E_2 соответственно;

¹⁾ Этот метод впервые описал польский математик Лукасевич, фамилию которого пронзести труднее, чем слово „польская“.

3.1. ФОРМАЛИЗМЫ, ИСПОЛЬЗУЕМЫЕ ДЛЯ ОПРЕДЕЛЕНИЯ ПЕРЕВОДА

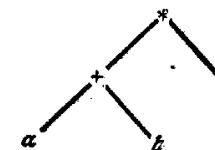
(б) $E'_1 E'_2 \theta$ —постфиксная польская запись выражения $E_1 \theta E_2$, где E'_1 и E'_2 —постфиксные польские записи выражений E_1 и E_2 соответственно.

- (3) Если (E) —инфиксное выражение, то
 (а) префиксная польская запись выражения (E) —это префиксная польская запись выражения E ;
 (б) постфиксная польская запись выражения (E) —это постфиксная польская запись выражения E .

Пример 3.2. Рассмотрим инфиксное выражение $(a + b) * c$. Это выражение вида $E_1 * E_2$, где $E_1 = (a + b)$ и $E_2 = c$. Тогда c —префиксная и постфиксная польская запись выражения E_2 . Префиксная запись выражения E_1 , т. е. выражения $a + b$, это $+ab$. Таким образом, префиксной записью выражения $(a + b) * c$ является $* + abc$.

Аналогично постфиксной записью выражения $a + b$ будет $ab +$, а постфиксной записью выражения $(a + b) * c$ будет $ab + c *$. \square

От префиксного или постфиксного выражения можно однозначно вернуться к инфиксному выражению. Это не совсем очевидно, но можно доказать, используя упр. 3.1.16 и 3.1.17.

Рис. 3.1. Древовидное представление выражения $(a + b) * c$.

Арифметические выражения удобно представлять с помощью деревьев. Представление выражения $(a + b) * c$ в виде дерева показано на рис. 3.1. В этом дереве каждая внутренняя вершина помечена знаком операции из множества Θ , а каждый лист—operandом из Σ . Префиксная польская запись—это просто левое скобочное представление дерева, из которого удалены все скобки. Аналогично постфиксная польская запись—это правое скобочное представление дерева, из которого удалены все скобки.

Два важных примера переводов—множества пар

$$\{(x, y) \mid x\text{—инфиксное выражение и } y\text{—префиксная (постфиксная) запись выражения } x\}$$

Эти переводы нельзя задать с помощью гомоморфизмов. Нужны более мощные способы задания переводов, и мы займемся теперь формализмами, позволяющими удобно описывать эти и другие переводы.

3.1.2. Схемы синтаксически управляемого перевода

Проблема задания бесконечного перевода конечными средствами аналогична проблеме задания бесконечного языка. Известно несколько возможных подходов к определению переводов. Аналогично порождению языка с помощью грамматики можно использовать систему, порождающую пары цепочек, принадлежащие переводу. Можно также воспользоваться распознавателем с двумя лентами, распознающим пары, принадлежащие переводу, или же определить автомат, который принимает в качестве входа цепочку x и выдаёт (недетерминировано, если нужно) все цепочки y , являющиеся переводами цепочки x . Этот список не исчерпывает всех возможностей, но охватывает наиболее распространенные модели.

Назовем устройство, которое по данной входной цепочке x вычисляет такую выходную цепочку y , что $(x, y) \in T$, транслятором, реализующим перевод T . Хотелось бы, чтобы определение перевода обладало „хорошими“ свойствами, в частности такими:

(1) в нем легко разобраться, т. е. легко установить, какие пары цепочек принадлежат переводу,

(2) прямо по определению перевода можно механически построить эффективный транслятор, реализующий этот перевод.

Желательные качества трансляторов таковы:

(1) эффективность трансляции — время, необходимое для обработки входной цепочки w длины n линейно зависит от n ,

(2) небольшой объем,

(3) корректность — желательно иметь небольшой конечный тест, такой, что если транслятор прошел через него, то правильность работы транслятора гарантирована на всех входных цепочках.

Одним из формализмов, используемых для определения переводов, является схема синтаксически управляемого перевода (трансляции). Интуитивно такая схема представляет собой просто грамматику, в которой к каждому правилу присоединяется элемент перевода. Всякий раз, когда правило участвует в выводе входной цепочки, с помощью элемента перевода вычисляется часть выходной цепочки, соответствующая части входной цепочки, порожденной этим правилом.

Пример 3.3. Рассмотрим схему, определяющую перевод $\{(x, x^R) \mid x \in \{0, 1\}^*\}$ (для каждого входа x выходом служит обращенная цепочка) по правилам, выписанным в табл. 3.2.

В переводе, определяемом этой схемой, пару вход—выход можно получить, порождая последовательность выводимых пар

цепочек (α, β) , где α — входная выводимая цепочка, а β — выходная выводимая цепочка. Начинается последовательность парой (S, S) . Затем к этой паре можно применить первое правило. Тогда первое S заменяется на $0S$ по правилу $S \rightarrow 0S$, а второе S заменяется на $S0$ в соответствии с элементом перевода

Таблица 3.2

Правило	Элемент перевода
(1) $S \rightarrow 0S$	$S = S0$
(2) $S \rightarrow 1S$	$S = S1$
(3) $S \rightarrow e$	$S = e$

$S = S0$. Пока можно рассматривать этот элемент перевода просто как правило $S \rightarrow S0$. Так получается, выводимая пара $(0S, S0)$. Снова применяя правило (1) к символу S в этой новой паре, получаем $(00S, S00)$. Затем правило (2) дает $(001S, S100)$. Если здесь применить правило (3), то будет $(001, 100)$. К последней паре никакого правила применить нельзя, и потому она принадлежит переводу, определяемому этой схемой. □

Схема трансляции T определяет некоторый перевод $\tau(T)$. По схеме T можно построить транслятор, реализующий перевод $\tau(T)$, который работает так. По данной входной цепочке x с помощью правил схемы трансляции транслятор находит (если это возможно) некоторый вывод цепочки x из S . Допустим, что $S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n = x$ — такой вывод. Затем транслятор строит вывод

$$(\alpha_0, \beta_0) \Rightarrow (\alpha_1, \beta_1) \Rightarrow \dots \Rightarrow (\alpha_n, \beta_n)$$

состоящий из выводимых пар цепочек, для которого $(\alpha_0, \beta_0) = (S, S)$, $(\alpha_n, \beta_n) = (x, y)$ и каждая цепочка β_i получается из β_{i-1} с помощью элемента перевода, соответствующего правилу, примененному в надлежащем месте при переходе от α_{i-1} к α_i . Цепочка y служит выходом для цепочки x .

Часто выходную цепочку можно получить за время, необходимое для разбора входной цепочки.

Пример 3.4. Рассмотрим схему перевода, отображающую арифметические выражения из языка $L(G_0)$ в соответствующие постфиксные польские записи. Она изображена в табл. 3.3.

Правилу $E \rightarrow E + T$ соответствует элемент перевода $E = ET +$. Этот элемент говорит о том, что перевод, порождаемый символом E , стоящим в левой части правила, получается

из перевода, порождаемого символом E , стоящим в правой части правила, за которым идут перевод, порождаемый символом T , и знак $+$.

Определим выход, соответствующий входу $a + a * a$. Для этого сначала по правилам схемы перевода найдем левый вывод цепочки

Таблица 3.3

Правило	Элемент перевода
$E \rightarrow E + T$	$E = ET +$
$E \rightarrow T$	$E = T$
$T \rightarrow T * F$	$T = TF *$
$T \rightarrow F$	$T = F$
$F \rightarrow (E)$	$T = E$
$F \rightarrow a$	$F = a$

почки $a + a * a$ из S . Затем вычислим соответствующую последовательность выводимых пар цепочек:

$$\begin{aligned} (E, E) &\Rightarrow (E + T, ET +) \\ &\Rightarrow (T + T, TT +) \\ &\Rightarrow (F + T, FT +) \\ &\Rightarrow (a + T, aT +) \\ &\Rightarrow (a + T * F, aTF * +) \\ &\Rightarrow (a + F * F, aFF * +) \\ &\Rightarrow (a + a * F, aaF * +) \\ &\Rightarrow (a + a * a, aaa * +) \end{aligned}$$

Каждая выходная цепочка этой последовательности получается из предыдущей выходной цепочки заменой подходящего нетерминала правой частью элемента перевода, присоединенного к правилу, примененному при выводе соответствующей входной цепочки. \square

Схемы перевода в примерах 3.3 и 3.4 относятся к важному классу схем, называемых схемами синтаксически управляемого перевода.

Определение. Схемой синтаксически управляемого перевода (или трансляции) (сокращенно СУ-схемой) называется пятерка $T = (N, \Sigma, \Delta, R, S)$, где

- (1) N — конечное множество нетерминальных символов,
- (2) Σ — конечный входной алфавит,
- (3) Δ — конечный выходной алфавит,

3.1. ФОРМАЛИЗМЫ, ИСПОЛЬЗУЕМЫЕ ДЛЯ ОПРЕДЕЛЕНИЯ ПЕРЕВОДА

(4) R — конечное множество правил вида $A \rightarrow \alpha, \beta$, где $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$ и вхождения нетерминалов в цепочку β образуют перестановку вхождений нетерминалов в цепочку α ,

(5) S — начальный символ, выделенный нетерминал из N .

Пусть $A \rightarrow \alpha, \beta$ — правило. Каждому вхождению нетерминала в цепочку α соответствует некоторое вхождение того же нетерминала в цепочку β . Если нетерминал B входит в цепочку α только один раз, то соответствие очевидно. Если B входит более одного раза, то для указания соответствия мы будем пользоваться верхними целочисленными индексами. Это соответствие является (если оно явно не указано, то подразумеваемой) частью правила. Например, в правиле $A \rightarrow B^{(1)}CB^{(2)}$, $B^{(2)}B^{(1)}C$ 1-й, 2-й и 3-й позиции цепочки $B^{(1)}CB^{(2)}$ соответствуют 2-я, 3-я и 1-я позиции цепочки $B^{(2)}B^{(1)}C$.

Определим выводимую пару цепочек схемы T :

(1) (S, S) — выводимая пара, в которой символы S соответствуют друг другу.

(2) если $(\alpha A \beta, \alpha' A' \beta')$ — выводимая пара, в которой два выделенных вхождения нетерминала A соответствуют друг другу, и $A \rightarrow \gamma, \gamma'$ — правило из R , то $(\alpha \gamma \beta, \alpha' \gamma' \beta')$ — выводимая пара. Вхождения нетерминалов в γ и γ' соответствуют друг другу точно так же, как они соответствовали в правиле. Вхождения нетерминалов в α и β соответствуют вхождениям нетерминалов в α' и β' в новой выводимой паре точно так же, как они соответствовали в старой выводимой паре. Когда надо, это соответствие будет указываться верхними индексами; оно составляет существенную часть выводимой пары.

Если между парами $(\alpha A \beta, \alpha' A' \beta')$ и $(\alpha \gamma \beta, \alpha' \gamma' \beta')$ с учетом соответствия их нетерминалов установлена описанная выше связь, то будем писать $(\alpha A \beta, \alpha' A' \beta') \Rightarrow_T (\alpha \gamma \beta, \alpha' \gamma' \beta')$. Транзитивное замыкание, рефлексивно-транзитивное замыкание и k -ю степень отношения \Rightarrow_T будем обозначать \Rightarrow_T^f , \Rightarrow_T^* и \Rightarrow_T^k соответственно. Когда можно, будем опускать индекс T .

Переводом, определяемым схемой T (обозначается $\tau(T)$), называют множество пар

$$\{(x, y) | (S, S) \Rightarrow^*(x, y), x \in \Sigma^* \text{ и } y \in \Delta^*\}$$

Пример 3.5. Рассмотрим СУ-схему $T = (\{S\}, \{a, +\}, \{a, +\}, R, S)$, где R состоит из правил

$$\begin{aligned} S &\rightarrow + S^{(1)} S^{(2)}, \quad S^{(1)} + S^{(2)} \\ S &\rightarrow a, \quad a \end{aligned}$$

и рассмотрим вывод

$$\begin{aligned}
 (S, S) &\Rightarrow (+ S^{(1)} S^{(2)}, S^{(1)} + S^{(2)}) \\
 &\Rightarrow (+ + S^{(3)} S^{(4)} S^{(2)}, S^{(3)} + S^{(4)} + S^{(2)}) \\
 &\Rightarrow (+ + a S^{(4)} S^{(2)}, a + S^{(4)} + S^{(2)}) \\
 &\Rightarrow (+ + a a S, a + a + S) \\
 &\Rightarrow (+ + a a a, a + a + a)
 \end{aligned}$$

$\tau(T) = \{(x, a(+a)^i) \mid i \geq 0\}$ и x — префиксная польская запись выражения $a(+a)^i$ с некоторым заданным порядком выполнения операций $+$. \square

Определение. Если $T = (N, \Sigma, \Delta, R, S)$ — СУ-схема, то $\tau(T)$ называется *синтаксически управляемым переводом* (СУ-переводом). Грамматика $G_i = (N, \Sigma, P, S)$, где $P = \{A \rightarrow \alpha \mid A \rightarrow \alpha, \beta \text{ принадлежит } R\}$, называется *входной грамматикой* СУ-схемы T . Грамматика $G_o = (N, \Delta, P', S)$, где $P' = \{A \rightarrow \beta \mid A \rightarrow \alpha, \beta \text{ принадлежит } R\}$, называется *выходной грамматикой* схемы T^1 .

Синтаксически управляемый перевод можно трактовать еще по-другому, считая его методом преобразования деревьев выводов входной грамматики G_i в деревья выводов выходной грамматики G_o . Перевод данной входной цепочки x можно получить, построив ее дерево вывода, затем преобразовав это дерево в дерево вывода в выходной грамматике и, наконец, взяв крону выходного дерева в качестве перевода цепочки x .

Алгоритм 3.1. Преобразование деревьев посредством СУ-схемы.

Вход. СУ-схема $T = (N, \Sigma, \Delta, R, S)$ с входной грамматикой $G_i = (N, \Sigma, P_i, S)$ и выходной грамматикой $G_o = (N, \Delta, P_o, S)$ и дерево вывода D в G_i с кроной, принадлежащей Σ^* .

Выход. Некоторое дерево вывода D' в G_o , такое, что если x и y — кроны деревьев D и D' соответственно, то $(x, y) \in \tau(T)$.

Метод.

(1) Применять шаг (2) рекурсивно, начиная с корня дерева D .

(2) Пусть этот шаг применяется к вершине n , внутренней в дереве D , и пусть n_1, \dots, n_k — ее прямые потомки.

(а) УстраниТЬ из множества вершин n_1, \dots, n_k листья (т. е. вершины, помеченные терминалами или e).

(б) Пусть $A \rightarrow \alpha$ — правило грамматики G_i , соответствующее вершине n и ее прямым потомкам, т. е. A — метка вершины n и α образуется конкатенацией меток вершин n_1, \dots, n_k . Выбрать из R некоторое правило вида $A \rightarrow \alpha, \beta^2$.

¹⁾ Здесь нижние индексы i и o — первые буквы слов *input* (вход) и *output* (выход). — Прим. ред.

²⁾ Заметим, что β может определяться по A и α не единственным образом. Если подходящих правил несколько, то выбор произволен.

3.1. ФОРМАЛИЗМЫ, ИСПОЛЬЗУЕМЫЕ ДЛЯ ОПРЕДЕЛЕНИЯ ПЕРЕВОДА

Переставить оставшихся прямых потомков вершины n (если они есть) согласно соответствуанию между вхождениями нетерминалов в α и β . (Поддеревья, корнями которых служат эти потомки, переставляются вместе с ними.)

(в) Добавить в качестве прямых потомков вершины n листья с метками так, чтобы метки всех ее прямых потомков образовали цепочку β .

(г) Применить шаг (2) к прямым потомкам вершины n , не являющимся листьями, в порядке слева направо.

3) Результирующим деревом будет D' . \square

Пример 3.6. Рассмотрим СУ-схему $T = (\{S, A\}, \{0, 1\}, \{a, b\}, R, S)$, где R состоит из правил

$$\begin{aligned}
 S &\rightarrow 0AS, \quad SAa \\
 A &\rightarrow 0SA, \quad ASA \\
 S &\rightarrow 1, \quad b \\
 A &\rightarrow 1, \quad b
 \end{aligned}$$

На рис. 3.2, а показано дерево вывода во входной грамматике. Применение к корню этого дерева шага (2) алгоритма 3.1

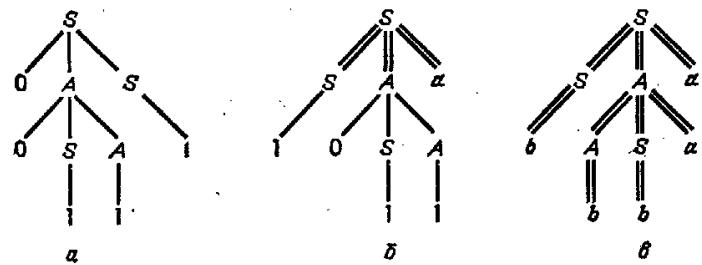


Рис. 3.2. Применение алгоритма 3.1.

устраивает левый лист, помеченный 0. Далее, так как корню соответствует правило $S \rightarrow 0AS$ и у этого правила только один элемент перевода SAa , нужно поменять местами оставшихся прямых потомков корня. Затем надо добавить в самой правой позиции третьего прямого потомка, помеченного a . Результатом будет дерево, показанное на рис. 3.2, б.

Снова применим шаг (2) уже к первым двум потомкам корня. Применение шага (2) ко второму из них приводит еще к двукратному повторению шага (2). Окончательный результат показан на рис. 3.2, в. Заметим, что $(00111, bbbbaa) \in \tau(T)$. \square

Чтобы установить связь между процессом перевода, осуществляемым алгоритмом 3.1, и СУ-схемой, которая служит входом этого алгоритма, докажем следующую теорему.

Теорема 3.1. (1) Если x и y — соответственно кроны деревьев D и D' из алгоритма 3.1, то $(x, y) \in \tau(T)$.

(2) Если $(x, y) \in \tau(T)$, то существуют дерево вывода D с кроной x и такая последовательность выборов при обращении к шагу (2б), что в результате получается дерево D' с кроной y .

Доказательство. (1) Индукцией по числу внутренних вершин дерева E докажем, что

(3.1.1) если E — дерево вывода в G_i с кроной x и корнем, помеченным A , и применение к E шага (2) дает дерево E' с кроной y , то $(A, A) \Rightarrow^*(x, y)$.

Базис (дерево, содержащее одну внутреннюю вершину) — три-вилен. Все прямые потомки являются листьями, и в R должно быть правило $A \rightarrow x$.

Для доказательства шага индукции допустим, что утверждение (3.1.1) верно для деревьев с меньшим числом внутренних вершин и корень дерева E имеет прямых потомков с метками X_1, \dots, X_k . Тогда $x = x_1 \dots x_k$, где $X_j \Rightarrow_{G_i}^* x_j$ для $1 \leq j \leq k$. Пусть прямые потомки корня дерева E' помечены Y_1, \dots, Y_l . Тогда $y = y_1 \dots y_l$, где $Y_j \Rightarrow_{G_0}^* y_j$ для $1 \leq j \leq l$. Кроме того, в R есть правило $A \rightarrow X_1 \dots X_k, Y_1 \dots Y_l$.

Если X_j — нетерминал, то ему соответствует некоторый символ $Y_{p_j} = X_j$. По предположению индукции $(X_j, X_{p_j}) \Rightarrow^* (x_j, y_{p_j})$. Так как шаг (2б) приводит к перестановке вершин, то

$$\begin{aligned} (A, A) &\Rightarrow (X_1 \dots X_k, Y_1 \dots Y_l) \\ &\Rightarrow^* (x_1 X_2 \dots X_k, \alpha_1^{(1)} \dots \alpha_l^{(1)}) \\ &\vdots \\ &\Rightarrow^* (x_1 \dots x_k, \alpha_1^{(k)} \dots \alpha_l^{(k)}) \end{aligned}$$

где

$$\alpha_j^{(m)} = \begin{cases} y_j, & \text{если } Y_j \in N \text{ и соответствует одному из } X_1, \dots, X_m, \\ Y_j & \text{в противном случае.} \end{cases}$$

Таким образом, утверждение (3.1.1) справедливо.

Часть (2) теоремы — это частный случай следующего утверждения:

(3.1.2) если $(A, A) \Rightarrow^f (x, y)$, то существуют дерево вывода D в G_i с корнем, помеченным A , и кроной x и такая последовательность выборов при обращении к шагу (2б), что применение шага (2) к дереву D дает дерево с кроной y .

Доказательство этого утверждения индукцией по f оставляем в качестве упражнения. \square

Заметим, что порядок применений шага (2) алгоритма 3.1 к вершинам дерева не важен. Можно было бы выбрать любой порядок, при котором каждая внутренняя вершина рассматривается точно один раз. Проверку этого утверждения тоже оставляем в качестве упражнения.

Определение. СУ-схема $T = (N, \Sigma, \Delta, R, S)$ называется *простой*, если для каждого правила $A \rightarrow \alpha, \beta$ из R соответствующие друг другу вхождения нетерминалов встречаются в α и β в одном и том же порядке. Перевод, определяемый простой СУ-схемой, называется *простым синтаксически управляемым переводом* (простым СУ-переводом).

Все СУ-схемы в примерах 3.3—3.5 простые, а в примере 3.6 — нет.

Соответствие нетерминалов в выводимой паре простой СУ-схемы самое простое, оно определяется порядком, в каком эти нетерминалы появляются в цепочках.

Простые СУ-переводы образуют важный класс переводов, потому что для каждого из них легко построить транслятор, представляющий собой преобразователь с магазинной памятью. Это построение будет проведено в разд. 3.1.4. Многие, но не все, полезные переводы можно описать как простые СУ-переводы. В гл. 9 мы дадим несколько обобщений схемы синтаксически управляемого перевода, которые можно использовать для определения более широких классов переводов КС-языков. В заключение этого раздела приведем еще один пример простого СУ-перевода.

Пример 3.7. Следующая простая СУ-схема отображает арифметические выражения из языка $L(G_0)$ в арифметические выражения, не содержащие избыточных скобок:

- | | |
|-------------------------------|-----------|
| (1) $E \rightarrow (E)$, | E |
| (2) $E \rightarrow E + E$, | $E + E$ |
| (3) $E \rightarrow T$, | T |
| (4) $T \rightarrow (T)$, | T |
| (5) $T \rightarrow A * A$, | $A * A$ |
| (6) $T \rightarrow a$, | a |
| (7) $A \rightarrow (E + E)$, | $(E + E)$ |
| (8) $A \rightarrow T$, | T |

Например, для выражения $((a + (a * a)) * a)$ эта СУ-схема дает перевод $(a + a * a) * a^1$. \square

¹) Заметим, что входная грамматика неоднозначна, но каждой входной цепочке соответствует точно одна выходная.

3.1.3. Конечные преобразователи

Введем наш простейший транслятор — конечный преобразователь. Преобразователь — это просто распознаватель, выдающий на каждом такте выходную цепочку (она может быть и пустой). Конечный преобразователь получится, если конечному автомату (распознавателю) позволить выдавать на каждом такте цепочку выходных символов (рис. 3.3). В разд. 3.3 мы будем поль-

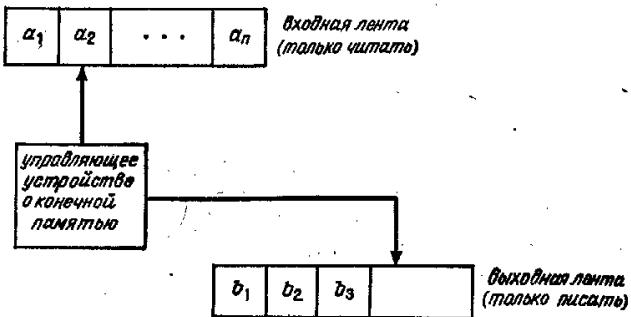


Рис. 3.3. Конечный преобразователь.

ватьсяся конечным преобразователем как моделью лексического анализатора.

Для большей общности рассмотрим в качестве основы конечного преобразователя недетерминированный конечный автомат, способный делать e -такты.

Определение. Конечным преобразователем называется шестерка $M = (Q, \Sigma, \Delta, \delta, q_0, F)$, где

- (1) Q — конечное множество состояний,
- (2) Σ — конечный входной алфавит,
- (3) Δ — конечный выходной алфавит,
- (4) δ — отображение множества $Q \times (\Sigma \cup \{e\})$ в множество конечных подмножеств множества $Q \times \Delta^*$,
- (5) $q_0 \in Q$ — начальное состояние,
- (6) $F \subseteq Q$ — множество заключительных состояний.

Определим конфигурацию преобразователя M как тройку (q, x, y) , где

- (1) $q \in Q$ — текущее состояние управляющего устройства,
- (2) $x \in \Sigma^*$ — оставшаяся непрочитанной часть входной цепочки, причем самый левый символ цепочки x расположен под входной головкой,
- (3) $y \in \Delta^*$ — часть выходной цепочки, выданная вплоть до текущего момента.

3.1. ФОРМАЛИЗМЫ, ИСПОЛЬЗУЕМЫЕ ДЛЯ ОПРЕДЕЛЕНИЯ ПЕРЕВОДА

Определим бинарное отношение \vdash_M (или \vdash , когда ясно, о каком M идет речь) на конфигурациях, соответствующее одному такту работы преобразователя M : для всех $q \in Q, a \in \Sigma \cup \{e\}$, $x \in \Sigma^*$ и $y \in \Delta^*$, таких, что $\delta(q, a)$ содержит (r, z) , будем писать

$$(q, ax, y) \vdash (r, x, yz)$$

Обычным образом далее определяются \vdash^i , \vdash^* и \vdash^+ .

Цепочку y назовем выходом для цепочки x , если $(q_0, x, e) \vdash^*(q, e, y)$ для некоторого $q \in F$. Переводом, определяемым преобразователем M (обозначается $\tau(M)$), назовем множество $\{(x, y) | (q_0, x, e) \vdash^*(q, e, y)$ для некоторого $q \in F\}$. Перевод, определяемый конечным преобразователем, будем называть регулярным переводом или конечным преобразованием.

Заметим, что для того, чтобы выходную цепочку y можно было считать переводом входной цепочки x , цепочка x должна перевести преобразователь M из начального состояния в заключительное.

Пример 3.8. Построим конечный преобразователь, который распознает арифметические выражения, порождаемые правилами

$$S \rightarrow a + S \mid a - S \mid + S \mid - S \mid a$$

и устраниет из этих выражений избыточные унарные операции. Например, выражение $-a + -a - + -a$ он переведет в $-a - a + a$. Во входном языке символ a представляет идентификатор и перед идентификатором допускается произвольная последовательность знаков унарных операций $+$ и $-$. Заметим, что входной язык является регулярным множеством. Пусть $M = (Q, \Sigma, \Delta, \delta, q_0, F)$, где

- (1) $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
- (2) $\Sigma = \{a, +, -\}$,
- (3) $\Delta = \Sigma$,

(4) δ определяется диаграммой, изображенной на рис. 3.4; метка x/y на дуге, ведущей из вершины, помеченной q_i , в вершину, помеченную q_j , указывает, что $\delta(q_i, x)$ содержит (q_j, y) ,

- (5) $F = \{q_1\}$.

Преобразователь M начинает работу в состоянии q_0 и, чередуя состояния q_0 и q_4 на входном символе $-$, определяет, четное или нечетное число знаков $-$ предшествует первому символу a . Когда появляется a , преобразователь M переходит в состояние q_1 , допуская вход, и выдает a или $-a$ в зависимости от того, четно или нечетно число появившихся минусов. Для следующих символов a он подсчитывает, четно или нечетно число предшествующих минусов, с помощью состояний q_2 и q_3 . Един-

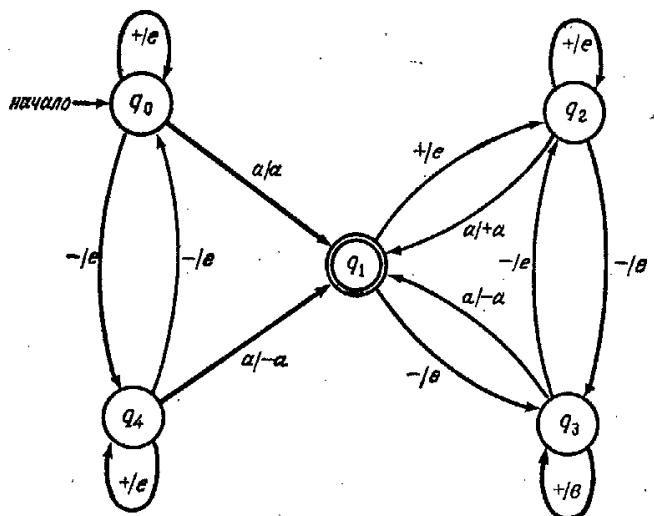


Рис. 3.4. Граф переходов.

ственное различие между парами \$q_2, q_3\$ и \$q_0, q_4\$ состоит в том, что если символу \$a\$ предшествует четное число минусов, то первая из них выдает \$+a\$, а не только \$a\$.

Для входа \$-\bar{a}+\bar{a}-\bar{a}+\bar{a}-\bar{a}\$ последовательность тактов преобразователя \$M\$ такова:

$$\begin{aligned}
 & (q_0, -\bar{a}+\bar{a}-\bar{a}+\bar{a}-\bar{a}, e) \vdash (q_4, a+\bar{a}-\bar{a}+\bar{a}-\bar{a}, e) \\
 & \vdash (q_1, +\bar{a}-\bar{a}+\bar{a}-\bar{a}, -a) \\
 & \vdash (q_2, -\bar{a}-\bar{a}+\bar{a}, -a) \\
 & \vdash (q_3, a-\bar{a}-\bar{a}, -a) \\
 & \vdash (q_1, -\bar{a}-\bar{a}, -a-a) \\
 & \vdash (q_3, +\bar{a}, -a-a) \\
 & \vdash (q_2, -a, -a-a) \\
 & \vdash (q_2, a, -a-a) \\
 & \vdash (q_1, e, -a-a+a)
 \end{aligned}$$

Отсюда ясно, что \$M\$ отображает цепочку \$-\bar{a}+\bar{a}-\bar{a}+\bar{a}-\bar{a}\$ в \$-\bar{a}-\bar{a}+\bar{a}\$, поскольку \$q_1\$ — заключительное состояние. \$\square\$

Конечный преобразователь \$M\$ назовем *детерминированным*, если для всех \$q \in Q\$

- (1) либо \$\delta(q, a)\$ содержит не более одного элемента для каждого \$a \in \Sigma\$ и \$\delta(q, e)\$ пусто, либо
- (2) \$\delta(q, e)\$ содержит один элемент и \$\delta(q, a)\$ пусто для всех \$a \in \Sigma\$.

В примере 3.8 конечный преобразователь детерминированный. Заметим, что детерминированный конечный преобразователь может давать несколько переводов для одного входа.

Пример 3.9. Пусть \$M = (\{q_0, q_1\}, \{a\}, \{b\}, \delta, q_0, \{q_1\})\$ и \$\delta(q_0, a) = \{(q_1, b)\}\$, \$\delta(q_1, e) = \{(q_1, b)\}\$. Тогда

$$\begin{aligned}
 (q_0, a, e) & \vdash (q_1, e, b) \\
 & \vdash^i (q_1, e, b^{i+1})
 \end{aligned}$$

— допустимая последовательность тактов для всех \$i \geq 0\$. Таким образом, \$\tau(M) = \{(a, b^i) | i \geq 1\}\$. \$\square\$

Известно несколько простых модификаций определения детерминизма для конечных преобразователей, гарантирующих единственность выхода. Мы предлагаем потребовать, чтобы в заключительном состоянии были невозможны \$e\$-такты.

Для некоторых классов языков можно получить ряд свойств замкнутости, если рассматривать преобразователи как операторы, определенные на языках. Например, если \$M\$ — конечный преобразователь и язык \$L\$ содержится в области определения отношения \$\tau(M)\$, то полагаем

$$M(L) = \{y | x \in L \text{ и } (x, y) \in \tau(M)\}$$

Можно также определить *обратное конечное преобразование*, а именно, если \$M\$ — конечный преобразователь, положим \$M^{-1}(L) = \{x | y \in L \text{ и } (x, y) \in \tau(M)\}\$.

Нетрудно показать, что конечные преобразования (прямые и обратные) сохраняют свойства регулярности и контекстной свободности. Иными словами, если \$L\$ — регулярное множество (КС-язык) и \$M\$ — конечный преобразователь, то \$M(L)\$ и \$M^{-1}(L)\$ — регулярные множества (КС-языки). Доказательства этих фактов оставляем в качестве упражнений. С их помощью можно доказать, что некоторые языки не регулярны (или не контекстно-свободны).

Пример 3.10. Язык, порождаемый грамматикой \$G\$ с правилами

$$S \rightarrow \text{if } S \text{ then } S | a$$

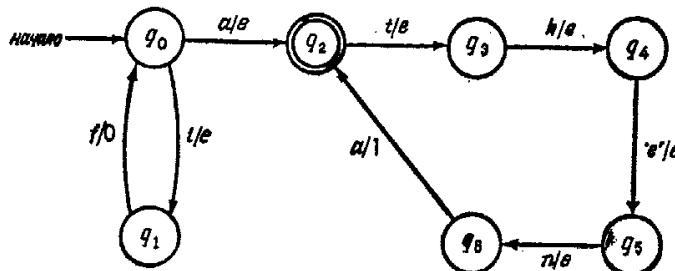
не регулярен. Обозначим

$$L_1 = L(G) \cap (\text{if})^* a (\text{then } a)^* = \{(\text{if})^n a (\text{then } a)^n | n \geq 0\}$$

Рассмотрим конечный преобразователь \$M = (Q, \Sigma, \Delta, \delta, q_0, F)\$, где

- (1) \$Q = \{q_i | 0 \leq i \leq 6\}\$,
- (2) \$\Sigma = \{a, i, f, t, h, "e", n\}\$,
- (3) \$\Delta = \{0, 1\}\$,
- (4) \$\delta\$ определяется диаграммой, изображенной на рис. 3.5,
- (5) \$F = \{q_2\}\$.

Здесь “ e ” означает букву e в отличие от пустой цепочки. Таким образом, $M(L_1) = \{0^k 1^k \mid k \geq 0\}$, а это, как мы знаем, не регулярное множество. Так как класс регулярных множеств замкнут относительно пересечения и конечных преобразований, то $L(G)$ не регулярное множество. \square

Рис. 3.5. Диаграмма преобразователя M .

3.1.4. Преобразователи с магазинной памятью

Теперь введем другой важный класс трансляторов, называемых преобразователями с магазинной памятью. Эти преобразователи получаются из автоматов с магазинной памятью, если их снабдить выходом и разрешить на каждом такте выдавать выходную цепочку конечной длины.

Определение. Преобразователем с магазинной памятью (МП-преобразователем) называется восьмерка $P = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$, где все символы имеют тот же смысл, что в определении МП-автомата, за исключением того, что Δ —конечный выходной алфавит, а δ —отображение множества $Q \times (\Sigma \cup \{e\}) \times \Gamma$ в множество конечных подмножеств множества $Q \times \Gamma^* \times \Delta^*$.

Определим конфигурацию преобразователя P как четверку (q, x, α, y) , где q, x и α те же, что у МП-автомата, а y —выходная цепочка, выданная вплоть до настоящего момента. Если $\delta(q, a, Z)$ содержит (r, α, z) , то будем писать $(q, ax, Z\gamma, y) \vdash (r, x, \alpha\gamma, yz)$ для любых $x \in \Sigma^*$, $\gamma \in \Gamma^*$ и $y \in \Delta^*$.

Цепочку y назовем выходом для x , если $(q_0, x, Z_0, e) \vdash^* (q, e, \alpha, y)$ для некоторых $q \in F$ и $\alpha \in \Gamma^*$. Переводом (или преобразованием), определяемым МП-преобразователем P (обозначается $\tau(P)$), назовем множество

$$\{(x, y) \mid (q_0, x, Z_0, e) \vdash^* (q, e, \alpha, y) \text{ для некоторых } q \in F \text{ и } \alpha \in \Gamma^*\}$$

Аналогично МП-автоматам можно говорить о преобразовании входа x в выход y опустошением магазина, если $(q_0, x, Z_0, e) \vdash^* (q, e, e, y)$ для некоторого $q \in Q$. Переводом, определяемым преобразователем P опустошением магазина (обозначается $\tau_e(P)$),

назовем множество

$$\{(x, y) \mid (q_0, x, Z_0, e) \vdash^* (q, e, e, y) \text{ для некоторого } q \in Q\}$$

Аналогично расширенным МП-автоматам можно определить расширенные МП-преобразователи (у них верх магазина расположена справа).

Пример 3.11. Рассмотрим МП-преобразователь

$$P = (\{q\}, \{a, +, *\}, \{+, *, E\}, \{a, +, *\}, \delta, q, E, \{q\})$$

где δ определяется равенствами

$$\begin{aligned}\delta(q, a, E) &= \{(q, e, a)\} \\ \delta(q, +, E) &= \{(q, EE +, e)\} \\ \delta(q, *, E) &= \{(q, EE *, e)\} \\ \delta(q, e, +) &= \{(q, e, +)\} \\ \delta(q, e, *) &= \{(q, e, *)\}\end{aligned}$$

Для входа $+ * aaa$ МП-преобразователь P сделает такую последовательность тактов:

$$\begin{aligned}(q, + * aaa, E, e) &\vdash (q, * aaa, EE +, e) \\ &\vdash (q, aaa, EE * E +, e) \\ &\vdash (q, aa, E * E +, a) \\ &\vdash (q, a, * E +, aa) \\ &\vdash (q, a, E +, aa *) \\ &\vdash (q, e, +, aa * a) \\ &\vdash (q, e, e, aa * a +)\end{aligned}$$

Таким образом, P переводит цепочку $+ * aaa$ в цепочку $aa * a +$, опустошая магазин. Можно проверить, что

$$\tau_e(P) = \{(x, y) \mid x \text{—префиксное польское арифметическое выражение в алфавите } \{+, *, a\} \text{ и } y \text{—соответствующая постфиксная польская запись}\} \quad \square$$

Определение. Если $P = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$ —МП-преобразователь, то МП-автомат $(Q, \Sigma, \Gamma, \delta', q_0, Z_0, F)$, где $\delta'(q, a, Z)$ содержит (r, γ) тогда и только тогда, когда $\delta(q, a, Z)$ содержит (r, γ, y) для некоторого y , назовем МП-автоматом, лежащим в основе преобразователя P (или просто основой P).

Будем говорить, что МП-преобразователь $P = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$ детерминированный (ДМП-преобразователь), если

(1) для всех $q \in Q$, $a \in \Sigma \cup \{e\}$ и $Z \in \Gamma$ множество $\delta(q, a, Z)$ содержит не более одного элемента,

(2) если $\delta(q, e, Z) \neq \emptyset$, то $\delta(q, a, Z) = \emptyset$ для всех $a \in \Sigma^1$.

Очевидно, что если L — область определения отношения $\tau(P)$ для некоторого МП-преобразователя P , то $L = L(P')$, где P' — МП-автомат, лежащий в основе P .

Многие из результатов, доказанных в разд. 2.5 для МП-автоматов, естественным образом распространяются на МП-преобразователи. В частности, аналогично леммам 2.22 и 2.23 можно доказать следующую лемму.

Лемма 3.1. $T = \tau(P_1)$ для МП-преобразователя P_1 тогда и только тогда, когда $T = \tau_e(P_2)$ для подходящего МП-преобразователя P_2 .

Доказательство. Упражнение. \square

МП-преобразователь, в частности детерминированный МП-преобразователь, служит полезной моделью для фазы синтаксического анализа процесса компиляции. Мы используем его в этой роли в разд. 3.4.

Докажем теперь, что перевод является простым СУ-переводом тогда и только тогда, когда он определяется МП-преобразователем. Таким образом, МП-преобразователи характеризуют класс простых СУ-переводов так же, как МП-автоматы характеризуют класс КС-языков.

Лемма 3.2. Пусть $T = (N, \Sigma, \Delta, R, S)$ — простая СУ-схема. Существует такой МП-преобразователь P , что $\tau_e(P) = \tau(T)$.

Доказательство. Пусть G_i — входная грамматика схемы T . Построим P так, чтобы он распознавал $L(G_i)$ сверху вниз, как в лемме 2.24.

При моделировании правила $A \rightarrow \alpha, \beta$ схемы T преобразователь P заменит в магазине верхний символ A цепочкой α , в которую вставлены выходные символы из цепочки β , т. е. если $\alpha = x_0 A_1 x_1 \dots A_n x_n$ и $\beta = y_0 A_1 y_1 \dots A_n y_n$, то символ A будет заменен цепочкой $x_0 y_0 A_1 x_1 y_1 \dots A_n x_n y_n$. Надо уметь, однако, отличать символы алфавита Σ от символов алфавита Δ , чтобы можно было правильно разбивать цепочки $x_i y_i$ на две части. Если Σ и Δ не пересекаются, то никакой проблемы нет. В общем случае определим новый алфавит Δ' , соответствующий Δ , но заведомо не пересекающийся с Σ . Будем считать, что Δ' состоит

¹⁾ Заметим, что это определение несколько сильнее утверждения о том, что лежащий в основе МП-автомат детерминированный. Последнее может быть и тогда, когда (1) не выполняется из-за того, что МП-преобразователь может выдать два разных выхода на двух шагах, которые во всем остальном совпадают. Заметим также, что из условия (2) следует, что если $\delta(q, a, Z) = \emptyset$ для некоторого $a \in \Sigma$, то $\delta(q, e, Z) = \emptyset$.

3.1. ФОРМАЛИЗМЫ, ИСПОЛЬЗУЕМЫЕ ДЛЯ ОПРЕДЕЛЕНИЯ ПЕРЕВОДА

из новых символов a' , соответствующих $a \in \Delta$. Тогда $\Delta' \cap \Sigma = \emptyset$ и естественно определить такой гомоморфизм h , что $h(a) = a'$ для каждого $a \in \Delta$.

Пусть $P = (\{q\}, \Sigma, N \cup \Sigma \cup \Delta', \Delta, \delta, q, S, \emptyset)$, где δ определяется так:

(1) если $A \rightarrow x_0 B_1 x_1 \dots B_k x_k, y_0 B_1 y_1 \dots B_k y_k$ — правило из R для $k \geq 0$, то $\delta(q, e, A)$ содержит $(q, x_0 y_0 B_1 x_1 y_1 \dots B_k x_k y_k, e)$, где $y_i = h(y_i)$ для $0 \leq i \leq k$;

(2) $\delta(q, a, a) = \{(q, e, e)\}$ для всех $a \in \Sigma$;

(3) $\delta(q, e, a') = \{(q, e, a)\}$ для всех $a \in \Delta$.

Индукцией по m и n можно показать, что для $A \in N$ и $m, n \geq 1$ справедливо следующее утверждение:

(3.1.3) $(A, A) \Rightarrow^m (x, y)$ для некоторого m тогда и только тогда, когда $(q, x, A, e) \vdash^n (q, e, e, y)$ для некоторого n .

Необходимость. Базис, $m = 1$, выполняется тривиально, так как правило $A \rightarrow x, y$ должно принадлежать R . Тогда $(q, x, A, e) \vdash (q, x, xh(y), e) \vdash^* (q, e, h(y), e) \vdash^* (q, e, e, y)$.

Для доказательства шага индукции допустим, что (3.1.3) справедливо для значений, меньших m , и пусть $(A, A) \Rightarrow (x_0 B_1 x_1 \dots B_k x_k, y_0 B_1 y_1 \dots B_k y_k) \Rightarrow^{m-1} (x, y)$. Так как в простой СУ-схеме порядок вхождений нетерминалов не меняется, можно писать $x = x_0 u_1 x_1 \dots u_k x_k$ и $y = y_0 v_1 y_1 \dots v_k y_k$, причем $(B_i, B_i) \Rightarrow^{m_i} (u_i, v_i)$ для $1 \leq i \leq k$, где $m_i < m$ для каждого i . Таким образом, по предположению индукции $(q, u_i, B_i, e) \vdash^* (q, e, e, v_i)$. Объединяя эти последовательности тактов, получаем

$$\begin{aligned} (q, x, A, e) \vdash & (q, x, x_0 h(y_0) B_1 \dots B_k x_k h(y_k), e) \\ & \vdash^* (q, u_1 x_1 \dots u_k x_k, h(y_0) B_1 \dots B_k x_k h(y_k), e) \\ & \vdash^* (q, u_1 x_1 \dots u_k x_k, B_1 \dots B_k x_k h(y_k), y_0) \\ & \vdash^* (q, x_1 \dots u_k x_k, x_1 h(y_1) \dots B_k x_k h(y_k), y_0 v_1) \\ & \vdash^* \dots \vdash^* (q, e, e, y) \end{aligned}$$

Достаточность. Опять базис, $n = 1$, тривиален, так как правило $A \rightarrow e, e$ должно принадлежать R . Для доказательства шага индукции допустим, что первый такт преобразователя P имеет вид

$$(q, x, A, e) \vdash (q, x, x_0 h(y_0) B_1 x_1 h(y_1) \dots B_k x_k h(y_k), e)$$

где $x_i \in \Sigma^*$, $y_i \in \Delta^*$, $h(y_i) \in \Delta'^*$. Тогда цепочка x_0 должна быть префиксом цепочки x и на следующих тактах работы преобразователя P цепочка x_0 будет прочитана на входе и устранена из магазина, а затем на выходе будет выдана цепочка y_0 . Пусть x' — оставшаяся часть входной цепочки. Тогда цепочка x' должна иметь префикс u_1 , который приводит к тому, что магазинные

символы, расположенные на уровне символа B_1 и выше, устраиваются из магазина. При этом к моменту, когда длина магазина становится меньше $|B_1 \dots B_k x_k h(y_k)|$, на выходе выдается цепочка v_1 . Тогда $(q, u_1, B_1, e) \vdash^*(q, e, e, v_1)$, и соответствующая последовательность состоит менее чем из n тактов. По предложению индукции $(B_1, B_1) \Rightarrow^* (u_1, v_1)$.

Рассуждая таким образом, мы приходим к тому, что x можно записать в виде $x_0 u_1 x_1 \dots u_k x_k$, а y — в виде $y_0 v_1 y_1 \dots v_k y_k$, причем $(B_i, B_i) \Rightarrow^* (u_i, v_i)$ для $1 \leq i \leq k$. Так как правило $A \rightarrow x_0 B_1 x_1 \dots B_k x_k, y_0 B_1 y_1 \dots B_k y_k$, очевидно, принадлежит R , то $(A, A) \Rightarrow^* (x, y)$.

В качестве частного случая утверждения (3.1.3) имеем $(S, S) \Rightarrow^* (x, y)$ тогда и только тогда, когда $(q, x, S, e) \vdash^* (q, e, e, y)$, так что $\tau_e(P) = \tau(T)$. \square

Пример 3.12. От простой СУ-схемы с правилами

$$\begin{aligned} E &\rightarrow +EE, EE+ \\ E &\rightarrow *EE, EE* \\ E &\rightarrow a, a \end{aligned}$$

можно перейти к эквивалентному МП-преобразователю

$$P = (\{q\}, \{a, +, *\}, \{E, a, +, *, a', +', *\}, \{a, +, *\}, \delta, q, E, \emptyset)$$

где δ определяется равенствами

- (1) $\delta(q, e, E) = \{(q, +EE+, e), (q, *EE*, e), (q, aa', e)\}$,
- (2) $\delta(q, b, b) = \{(q, e, e)\}$ для всех $b \in \{a, +, *\}$,
- (3) $\delta(q, e, b') = \{(q, e, b)\}$ для всех $b \in \{a, +, *\}$.

Это недетерминированный МП-преобразователь. Эквивалентный ему детерминированный МП-преобразователь приведен в примере 3.11. \square

Лемма 3.3. Пусть $P = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$ — МП-преобразователь. Существует такая простая СУ-схема, что $\tau(T) = \tau_e(P)$.

Доказательство. Построение схемы аналогично построению КС-грамматики по МП-автомату. Пусть $T = (N, \Sigma, \Delta, R, S)$, где

- (1) $N = \{[pAq] \mid p, q \in Q, A \in \Gamma\} \cup \{S\}$,
- (2) R определяется так:

(а) если $\delta(p, a, A)$ содержит $(r, X_1 X_2 \dots X_k, y)$, то при $k > 0$ в R входят правила

$$\begin{aligned} [pAq_k] &\rightarrow a[rX_1q_1][q_1X_2q_2] \dots [q_{k-1}X_kq_k], \\ &y[rX_1q_1][q_1X_2q_2] \dots [q_{k-1}X_kq_k] \end{aligned}$$

3.1. ФОРМАЛИЗМЫ, ИСПОЛЬЗУЕМЫЕ ДЛЯ ОПРЕДЕЛЕНИЯ ПЕРЕВОДА

для всех последовательностей q_1, q_2, \dots, q_k состояний из Q , а при $k = 0$ в R входит одно правило $[pAr] \rightarrow a, y$;

(б) для каждого $q \in Q$ в R входит правило $S \rightarrow [q_0Z_0q]$.

Ясно, что T — простая СУ-схема. Снова индукцией по m и по n легко доказать, что

- (3.1.4) $([pAq], [pAq]) \Rightarrow^m (x, y)$ тогда и только тогда, когда $(p, x, A, e) \vdash^n (q, e, e, y)$ для всех $p, q \in Q$ и $A \in \Gamma$.

Доказательство утверждения (3.1.4) оставляем в качестве упражнения.

Таким образом, $(S, S) \Rightarrow ([q_0Z_0q], [q_0Z_0q]) \Rightarrow^+ (x, y)$ тогда и только тогда, когда $(q_0, x, Z_0, e) \vdash^+ (q, e, e, y)$. Следовательно, $\tau(T) = \tau_e(P)$. \square

Пример 3.13. С помощью конструкции предыдущей леммы построим по МП-преобразователю из примера 3.11 эквивалентную ему простую СУ-схему. Получим СУ-схему $T = (N, \{a, +, *\}, \{a, +, *\}, R, S)$, где $N = \{[qXq] \mid X \in \{+, *\}\} \cup \{S\}$, а R состоит из правил

$$\begin{aligned} S &\rightarrow [qEq], [qEq] \\ [qEq] &\rightarrow a, a \\ [qEq] &\rightarrow +[qEq][qEq][q+q], [qEq][qEq][q+q] \\ [qEq] &\rightarrow *[qEq][qEq][q*q], [qEq][qEq][q*q] \\ [q+q] &\rightarrow e, + \\ [q*q] &\rightarrow e, * \end{aligned}$$

Заметим, что преобразования, аналогичные тем, которые применялись для устранения из КС-грамматики цепных правил и e -правил, позволяют упростить эту схему и получить правила

$$\begin{aligned} S &\rightarrow a, a \\ S &\rightarrow +SS, SS+ \\ S &\rightarrow *SS, SS* \end{aligned} \quad \square$$

Теорема 3.2. T — простой СУ-перевод тогда и только тогда, когда $T = \tau(P)$ для некоторого МП-преобразователя P .

Доказательство. Теорема непосредственно следует из лемм 3.1—3.3. \square

В гл. 9 мы введем машины, называемые процессорами с магазинной памятью, способные определять любые синтаксически управляемые переводы.

УПРАЖНЕНИЯ

3.1.1. Операция¹⁾ с одним аргументом называется *унарной*, с двумя — *бинарной*, и вообще операция с n аргументами называется *n-арной* (или *n-местной*). Например, операция — может быть унарной (как в $-a$) и бинарной (как в $a-b$). Число аргументов операции иногда называют ее *арностью* (или *местностью*). Пусть Θ — множество операций с заданными арностями и Σ — множество операндов. Постройте КС-грамматики G_1 и G_2 , порождающие префиксные и постфиксные польские записи выражений, составленных из операций множества Θ и операндов множества Σ .

***3.1.2.** „Предшествование“ (приоритет) инфиксных операций определяет порядок, в котором они выполняются. Если бинарная операция θ_1 „предшествует“ операции θ_2 , то $a\theta_2b\theta_1c$ вычисляется как $a\theta_2(b\theta_1c)$. Например, $*$ предшествует $+$, и потому $a+b*c$ означает $a+(b*c)$, а не $(a+b)*c$. Рассмотрим булевы операции \neg (не), \wedge (и), \vee (или), \rightarrow (импликация) и \equiv (эквиваленция). Эти операции перечислены в том порядке, в каком они предшествуют друг другу. Операция \neg — унарная, а остальные — бинарные. Например, в выражении $\neg(a \vee b) \equiv \neg a \wedge \neg b$ подразумевается такая расстановка скобок: $(\neg(a \vee b)) \equiv ((\neg a) \wedge (\neg b))$. Постройте КС-грамматику, порождающую все правильные булевые выражения, составленные из этих операций и операндов a, b , с и не содержащие избыточных скобок.

***3.1.3.** Постройте простую СУ-схему, отображающую инфиксные булевые выражения в префиксные.

***3.1.4.** В Алголе выражения строятся с помощью операций, перечисленных ниже в порядке их приоритетов. Если операций одного приоритета более одной, то они выполняются в порядке слева направо. Например, $a-b+c$ означает $(a-b)+c$.

$$\begin{array}{lll} (1) \uparrow & (2) \times / \div & (3) + - \\ (4) \leqslant < = \neq > \geqslant & (5) \neg & (6) \wedge \\ (7) \vee & (8) \rightarrow & (9) \equiv \end{array}$$

Постройте простую СУ-схему, которая отображает инфиксные выражения, содержащие эти операции, в их постфиксные польские записи.

3.1.5. Рассмотрим следующую СУ-схему, в которой цепочка вида $\langle x \rangle$ означает один нетерминал:

```

<exp> → sum <exp>(1) with <var> ← <exp>(2) to <exp>(3), 1)
      begin local t;
      t ← 0;
      for <var> ← <exp>(2) to <exp>(3) do
      t ← t + <exp>(1); result t
      end
<var> → <id>, <id>
<exp> → <id>, <id>
<id> → a <id>, a <id>
<id> → b <id>, b <id>
<id> → a, a
<id> → b, b
  
```

Постройте переводы предложений

- a) sum aa with a ← b to bb
- b) sum sum a with aa ← aaa to aaaa with b ← bb to bbb

***3.1.6.** Рассмотрим такую схему трансляции:

```

<statement> → for <var> ← <exp>(1) to <exp>(2) do <statement>,
      begin <var> ← <exp>(1);
      L: if <var> ≤ <exp>(2) then
          begin <statement>;
          <var> ← <var> + 1
          go to L
          end
      end
<var> → <id>, <id>
<exp> → <id>, <id>
<statement> → <var> ← <exp>, <var> ← <exp>
<id> → a <id>, a <id>
<id> → b <id>, b <id>
<id> → a, a
<id> → b, b
  
```

Почему эта схема не является СУ-схемой? Каким должен быть выход для входного предложения

for $a \leftarrow b$ to aa do baa \leftarrow bba

¹⁾ Обратите внимание, что эта запятая отделяет две части правила.

¹⁾ Здесь и в дальнейшем мы будем называть операциями как сами *операции* (например, арифметические), так и знаки *операций*. Надеемся, что это не вызовет недоразумений.— Прим. ред.

Указание: Примените алгоритм 3.1, дублируя в выходном дереве вершины, помеченные $\langle \text{var} \rangle$.

Упражнения 3.1.5 и 3.1.6 дают примеры того, как с помощью синтаксических макросов можно расширять язык. В приложении П1 подробно показано, как включить в язык такой механизм расширения.

3.1.7. Докажите, что область определения и множество значений любого СУ-перевода являются КС-языками.

3.1.8. Пусть $L \subseteq \Sigma^*$ — КС-язык и $R \subseteq \Sigma^*$ — регулярное множество. Постройте такую СУ-схему T , что

$$\begin{aligned}\tau(T) = \{(x, y) \mid & \text{ если } x \in L - R, \text{ то } y = 0, \\ & \text{если } x \in L \cap R, \text{ то } y = 1\}\end{aligned}$$

***3.1.9.** Постройте такую СУ-схему T , что

$$\begin{aligned}\tau(T) = \{(x, y) \mid & x \in \{a, b\}^* \text{ и } y = c^t, \text{ где} \\ & t = |\#_a(x) - \#_b(x)| \text{ и } \#_d(x) — \text{число} \\ & \text{символов } d \text{ в цепочке } x\}\end{aligned}$$

***3.1.10.** Покажите, что если L — регулярное множество и M — конечный преобразователь, то $M(L)$ и $M^{-1}(L)$ — регулярные множества.

***3.1.11.** Покажите, что если L — КС-язык и M — конечный преобразователь, то $M(L)$ и $M^{-1}(L)$ — КС-языки.

***3.1.12.** Пусть R — регулярное множество. Постройте такой конечный преобразователь M , что $M(L) = L/R$ для любого языка L . Учитывая упр. 3.1.10 и 3.1.11, заключаем отсюда, что классы регулярных множеств и КС-языков замкнуты относительно операции $/R$ (деление справа на регулярное множество).

***3.1.13.** Пусть R — регулярное множество. Постройте такой конечный преобразователь M , что $M(L) = R/L$ для любого языка L .

3.1.14. СУ-схема $T = (N, \Sigma, \Delta, R, S)$ называется *праволинейной*, если каждое правило из R имеет вид

$$A \rightarrow xB, yB$$

или

$$A \rightarrow x, y$$

где $A, B \in N$, $x \in \Sigma^*$ и $y \in \Delta^*$. Покажите, что если T — праволинейная СУ-схема, то $\tau(T)$ — регулярный перевод (конечное преобразование).

****3.1.15.** Покажите, что если $T \subseteq a^* \times b^*$ — СУ-перевод, то T определяется конечным преобразователем.

3.1.16. Рассмотрим класс префиксных выражений в алфавитах операций Θ и operandов Σ . Если цепочка $a_1 \dots a_n$ принадлежит $(\Theta \cup \Sigma)^*$, то степень s_i ее i -й позиции ($0 \leq i \leq n$) вычисляется так:

- (1) $s_0 = 1$,
- (2) если a_i — m -местная операция, то $s_i = s_{i-1} + m - 1$,
- (3) если $a_i \in \Sigma$, то $s_i = s_{i-1} - 1$.

Докажите, что $a_1 \dots a_n$ — префиксное выражение тогда и только тогда, когда $s_n = 0$ и $s_i > 0$ для всех $i < n$.

***3.1.17.** Пусть $a_1 \dots a_n$ — префиксное выражение, в котором a_1 — m -местная операция. Покажите, что единственный способ записать $a_1 \dots a_n$ как $a_1 w_1 \dots w_m$, где w_1, \dots, w_m — префиксные выражения, — это выбрать выражение w_j ($1 \leq j \leq m$) так, чтобы оно оканчивалось первым из символов a_k , у которого $s_k = m - j$.

***3.1.18.** Покажите, что каждое префиксное выражение с бинарными операциями получается из единственного инфиксного выражения, не содержащего избыточных скобок.

3.1.19. Переформулируйте и выполните упр. 3.1.16—3.1.18 для постфиксных выражений.

3.1.20. Дополните доказательство теоремы 3.1.

***3.1.21.** Докажите, что порядок, в котором шаг (2) алгоритма 3.1 применяется к вершинам, не влияет на результирующее дерево.

3.1.22. Докажите лемму 3.1.

3.1.23. Постройте МП-преобразователи, реализующие простые СУ-переводы, определенные схемами трансляций из примеров 3.5 и 3.7.

3.1.24. Постройте грамматику для операций языка Снобол 4, отражающую ассоциативность и приоритет операций, указанные в приложении П2.

3.1.25. Найдите СУ-схему, определяющую перевод, который определяет (опустошением магазина) МП-преобразователь

$$(\{q, p\}, \{a, b\}, \{Z_0, A, B\}, \{a, b\}, \delta, q, Z_0, \emptyset)$$

где δ задается равенствами

$$\begin{aligned}\delta(q, a, X) &= (q, AX, e) \text{ для всех } X \in \{Z_0, A, B\} \\ \delta(q, b, X) &= (q, BX, e) \text{ для всех } X \in \{Z_0, A, B\} \\ \delta(q, e, A) &= (p, A, a) \\ \delta(p, a, A) &= (p, e, b) \\ \delta(p, b, B) &= (p, e, b) \\ \delta(p, e, Z_0) &= (p, e, a)\end{aligned}$$

****3.1.26.** Рассмотрим два МП-преобразователя, соединенных последовательно, так что выход первого служит входом второго. Покажите, что для такого «тандема» МП-преобразователей множеством всевозможных выходов второго МП-преобразователя может быть любое рекурсивно перечислимое множество.

3.1.27. Покажите, что T — регулярный перевод тогда и только тогда, когда существует такой линейный КС-язык L , что $T = \{(x, y) | xcy^R \in L\}$, где c — новый символ.

***3.1.28.** Докажите, что проблема равенства $T_1 = T_2$ для регулярных переводов T_1 и T_2 неразрешима.

Открытые проблемы

3.1.29. Разрешима ли проблема эквивалентности для детерминированных конечных преобразователей?

3.1.30. Разрешима ли проблема эквивалентности для детерминированных МП-преобразователей?

Проблема для исследования

3.1.31. Известно, что проблема эквивалентности для недетерминированных конечных преобразователей неразрешима (упр. 3.1.28). Поэтому их нельзя «минимизировать» в том смысле, в котором мы в разд. 3.3.1 минимизировали конечные автоматы. Однако с помощью некоторых приемов можно уменьшить число состояний. Попробуйте найти полезный набор таких приемов. То же самое попытайтесь сделать для МП-преобразователей.

Замечания по литературе

Идея синтаксически управляемой трансляции возникла у многих примерно в одно и то же время. Среди тех, кто первыми стали пропагандировать ее применение, были Айронс [1961] и Барнет и Фуртель [1962]. Конечные преобразователи аналогичны обобщенным последовательностным машинам, введенным С. Гинзбургом [1962]. Наши определения СУ-схемы и МП-преобразователя и результат об их эквивалентности (теорема 3.2) подобны тому, что приведено в работе Льюнса и Стирнза [1968]¹. Гриффитс [1968] показал, что проблема эквивалентности для недетерминированных конечных преобразователей без e -выходов тоже неразрешима.

3.2. СВОЙСТВА СИНТАКСИЧЕСКИ УПРАВЛЯЕМЫХ ПЕРЕВОДОВ

В этом разделе мы исследуем несколько теоретических свойств синтаксически управляемых переводов и дадим характеристику простых СУ-переводов.

¹⁾ Следует также упомянуть работы Чулика [1966] и Петроне [1965]. — Прим. перев.

3.2.1. Характеризующие языки

Определение. Будем говорить, что язык L характеризует перевод T , если существуют такие два гомоморфизма h_1 и h_2 , что $T = \{(h_1(w), h_2(w)) | w \in L\}$.

Пример 3.14. Перевод $T = \{(a^n, a^n) | n \geq 1\}$ характеризуется языком 0^+ , так как $T = \{(h_1(w), h_2(w)) | w \in 0^+\}$, где $h_1(0) = h_2(0) = a$. \square

Будем говорить, что язык $L \subseteq (\Sigma \cup \Delta')^*$ сильно характеризует перевод $T \subseteq \Sigma^* \times \Delta^*$, если

- (1) $\Sigma \cap \Delta' = \emptyset$;
- (2) $T = \{(h_1(w), h_2(w)) | w \in L\}$, где
 - (a) $h_1(a) = a$ для всех $a \in \Sigma$ и $h_1(b) = e$ для всех $b \in \Delta'$,
 - (б) $h_2(a) = e$ для всех $a \in \Sigma$ и h_2 взаимно однозначно отображает Δ' на Δ (т. е. h_2 — биективное отображение Δ' в Δ).

Пример 3.15. Перевод $T = \{(a^n, a^n) | n \geq 1\}$ сильно характеризуется языком $L_1 = \{(a^n b^n) | n \geq 1\}$. Он также сильно характеризуется языком $L_2 = \{w | w$ состоит из одинакового числа символов a и $b\}$. Гомоморфизмы в обоих случаях определяются равенствами $h_1(a) = a$, $h_1(b) = e$ и $h_2(a) = e$, $h_2(b) = a$. Язык 0^+ не сильно характеризует перевод T . \square

С помощью понятия характеризующего языка можно исследовать классы переводов, определяемых конечными преобразователями и МП-преобразователями.

Лемма 3.4. Пусть $T = (N, \Sigma, \Delta, R, S)$ — СУ-схема, в которой каждое правило имеет вид $A \rightarrow aB$, bB или $A \rightarrow a$, b , где $a \in \Sigma \cup \{e\}$, $b \in \Delta \cup \{e\}$ и $B \in N$. Тогда $\tau(T)$ — регулярный перевод.

Доказательство. Пусть $M = (N \cup \{f\}, \Sigma, \Delta, \delta, S, \{f\})$ — конечный преобразователь, причем f — новый символ. Пусть $\delta(A, a)$ содержит (B, b) , если правило $A \rightarrow aB$, bB принадлежит R , и содержит (f, b) , если правило $A \rightarrow a$, b принадлежит R . Индукцией по n можно показать, что

$(S, x, e) \vdash^n (A, e, y)$ тогда и только тогда, когда $(S, S) \Rightarrow^n (xA, yA)$

Отсюда следует, что $(S, x, e) \vdash^*(f, e, y)$ тогда и только тогда, когда $(S, S) \Rightarrow^*(x, y)$. Детали доказательства оставляем в качестве упражнения. Таким образом, $\tau(T) = \tau(M)$. \square

Теорема 3.3. T — регулярный перевод тогда и только тогда, когда T сильно характеризуется регулярным языком.

Доказательство. *Достаточность.* Допустим, что $L \subseteq (\Sigma \cup \Delta')^*$ — регулярный язык и h_1 и h_2 — гомоморфизмы, причем $h_1(a) = a$ для $a \in \Sigma$, $h_1(a) = e$ для $a \in \Delta'$, $h_2(a) = e$ для $a \in \Sigma$ и h_2 взаимно однозначно отображает Δ' на Δ . Пусть $T = \{(h_1(w), h_2(w)) \mid w \in L\}$, а $G = (N, \Sigma \cup \Delta', P, S)$ — регулярная грамматика, для которой $L(G) = L$. Рассмотрим СУ-схему $U = (N, \Sigma, \Delta, R, S)$, где R определяется так:

(1) если $A \rightarrow aB$ принадлежит P , то $A \rightarrow h_1(a)B$, $h_2(a)B$ принадлежит R ;

(2) если $A \rightarrow a$ принадлежит P , то $A \rightarrow h_1(a)$, $h_2(a)$ принадлежит R .

Легко доказать с помощью индукции, что $(A, A) \Rightarrow_U^n (x, y)$ тогда и только тогда, когда $A \Rightarrow_G^n w$, $h_1(w) = x$ и $h_2(w) = y$. Отсюда можно заключить, что $(S, S) \Rightarrow_U^n (x, y)$ тогда и только тогда, когда $(x, y) \in T$. Следовательно, $\tau(U) = T$. По лемме 3.4 существует конечный преобразователь M , для которого $\tau(M) = T$.

Необходимость. Допустим, что $T \subseteq \Sigma^* \times \Delta^*$ — регулярный перевод и $M = (Q, \Sigma, \Delta, \delta, q_0, F)$ — конечный преобразователь, для которого $\tau(M) = T$.

Пусть $\Delta' = \{a' \mid a \in \Delta\}$ — алфавит, состоящий из новых символов, а $G = (Q, \Sigma \cup \Delta', P, q_0)$ — праволинейная грамматика, в которой P определяется так:

(1) если $\delta(q, a)$ содержит (r, y) , то $q \rightarrow ah(y)r$ принадлежит P , где h — такой гомоморфизм, что $h(a) = a'$ для всех $a \in \Delta$;

(2) если $q \in F$, то $q \rightarrow e$ принадлежит P .

Определим гомоморфизмы h_1 и h_2 равенствами

$$h_1(a) = a \text{ для всех } a \in \Sigma$$

$$h_1(b) = e \text{ для всех } b \in \Delta'$$

$$h_2(a) = e \text{ для всех } a \in \Sigma$$

$$h_2(b') = b \text{ для всех } b' \in \Delta'$$

Индукцией по m и по n можно показать, что $(q, x, e) \vdash_m (r, e, y)$ для некоторого m тогда и только тогда, когда $q \Rightarrow_G^n w$ для некоторого n , где $h_1(w) = x$ и $h_2(w) = y$. Таким образом, $(q_0, x, e) \vdash^+ (q, e, y)$ для $q \in F$ тогда и только тогда, когда $q_0 \Rightarrow_G^n w \Rightarrow w$, где $h_1(w) = x$ и $h_2(w) = y$. Следовательно, $T = \{(h_1(w), h_2(w)) \mid w \in L(G)\}$ и, значит, $L(G)$ сильно характеризует T . \square

Следствие. *Т — регулярный перевод тогда и только тогда, когда T характеризуется регулярным языком.*

Доказательство. Сильная характеризация — частный случай характеризации. Поэтому в одну сторону («только тогда»)

утверждение очевидно. В другую сторону («тогда») доказательство получается простым обобщением доказательства соответствующей части теоремы.

Похоже доказывается аналогичный результат для простых СУ-переводов.

Теорема 3.4. *Т — простой СУ-перевод тогда и только тогда, когда он сильно характеризуется КС-языком.*

Доказательство. *Достаточность.* Пусть T сильно характеризуется языком, порождаемым грамматикой $G_1 = (N, \Sigma \cup \Delta', P, S)$, где h_1 и h_2 — соответствующие гомоморфизмы. Построим простую СУ-схему $T_1 = (N, \Sigma, \Delta, R, S)$, где R определяется так: для каждого правила $A \rightarrow w_0 B_1 w_1 \dots B_k w_k$ из P правило

$$A \rightarrow h_1(w_0) B_1 h_1(w_1) \dots B_k h_1(w_k), h_2(w_0) B_1 h_2(w_1) \dots B_k h_2(w_k)$$

принадлежит R .

Индукцией по n легко доказать, что

(1) если $A \Rightarrow_{G_1}^n w$, то $(A, A) \Rightarrow_{T_1}^n (h_1(w), h_2(w))$,

(2) если $(A, A) \Rightarrow_{T_1}^n (x, y)$, то существует такая цепочка w , что $A \Rightarrow_{G_1}^n w$ и $h_1(w) = x$, $h_2(w) = y$.

Таким образом, $\tau(T_1) = T$.

Необходимость. Пусть $T = \tau(T_2)$, где $T_2 = (N, \Sigma, \Delta, R, S)$, и $\Delta' = \{a' \mid a \in \Delta\}$ — алфавит, состоящий из новых символов. Построим КС-грамматику $G_2 = (N, \Sigma \cup \Delta', P, S)$, где P содержит правило $A \rightarrow x_0 y'_0 B_1 x_1 y_1 \dots B_k x_k y_k$ для каждого правила $A \rightarrow x_0 B_1 x_1 \dots B_k x_k, y_0 B_1 y_1 \dots B_k y_k$, принадлежащего R , причем y'_i получается из y_i заменой каждого символа $a \in \Delta$ на $a' \in \Delta'$. Пусть h_1 и h_2 — очевидные гомоморфизмы: $h_1(a) = a$ для $a \in \Sigma$, $h_1(a) = e$ для $a \in \Delta'$, $h_2(a) = e$ для $a \in \Sigma$ и $h_2(a') = a$ для $a \in \Delta$. Снова можно элементарно доказать индукцией, что

(1) если $A \Rightarrow_{G_2}^n w$, то $(A, A) \Rightarrow_{T_2}^n (h_1(w), h_2(w))$,

(2) если $(A, A) \Rightarrow_{T_2}^n (x, y)$, то существует такая цепочка w , что $A \Rightarrow_{G_2}^n w$ и $h_1(w) = x$, $h_2(w) = y$. \square

Следствие. *Перевод является простым СУ-переводом тогда и только тогда, когда он характеризуется КС-языком.* \square

Для некоторых переводов можно показать с помощью теорем 3.3 и 3.4, что они не регулярные или же не простые СУ-переводы. Легко показать, что область определения и множество значений любого СУ-перевода принадлежат классу КС-языков. Однако есть и такие СУ-переводы, что их область определения и множество значений регулярны, но сами они не определяются ни конечным преобразователем, ни даже МП-преобразователем.

Пример 3.16. Рассмотрим простую СУ-схему с правилами

$$\begin{aligned} S &\rightarrow 0S, \quad S0 \\ S &\rightarrow 1S, \quad S1 \\ S &\rightarrow e, \quad e \end{aligned}$$

Покажем, что $\tau(T) = \{(w, w^R) \mid w \in \{0, 1\}^*\}$ не является регулярным переводом.

Допустим, что перевод $\tau(T)$ регулярен. Тогда найдется регулярный язык L , сильно характеризующий $\tau(T)$. Без потери общности можно предположить, что $L \subseteq \{0, 1, a, b\}^*$, а соответствующие гомоморфизмы удовлетворяют условиям $h_1(0) = 0$, $h_1(1) = 1$, $h_1(a) = h_1(b) = e$, $h_2(0) = h_2(1) = e$, $h_2(a) = 0$, $h_2(b) = 1$.

Если язык L регулярен, он допускается конечным автоматом

$$M = (Q, \{0, 1, a, b\}, \delta, q_0, F)$$

с s состояниями для некоторого $s \geq 1$. В L должна найтись такая цепочка z , что $h_1(z) = 0^s 1^s$ и $h_2(z) = 1^s 0^s$, поскольку $(0^s 1^s, 1^s 0^s) \in \tau(T)$. Все нули в z предшествуют всем единицам, а все символы b предшествуют всем символам a . Таким образом, первые s символов цепочки z могут быть только нулями или символами b . Состояния, которые пройдет автомат M , прочитав первые s символов цепочки z , не могут все быть разными, так что можно писать $z = uvw$, причем $(q_0, z) \vdash^* (q, vw) \vdash^+ (q, w) \vdash^* (p, e)$, где $|uv| \leq s$, $|v| \geq 1$ и $p \in F$. Тогда $uvw \in L$. Но $h_1(uvw) = 0^{s+m} 1^s$ и $h_2(uvw) = 1^{s+n} 0^s$, где m и n не равны одновременно нулю. Поэтому $(0^{s+m} 1^s, 1^{s+n} 0^s) \in \tau(T)$. Мы пришли к противоречию, так как предположили, что перевод $\tau(T)$ регулярен. \square

Пример 3.17. Рассмотрим СУ-схему T с правилами

$$\begin{aligned} S &\rightarrow A^{(1)} c A^{(2)}, \quad A^{(2)} c A^{(1)} \\ A &\rightarrow 0A, \quad 0A \\ A &\rightarrow 1A, \quad 1A \\ A &\rightarrow e, \quad e \end{aligned}$$

Здесь $\tau(T) = \{(uv, vcu) \mid u, v \in \{0, 1\}^*\}$. Покажем, что $\tau(T)$ не является простым СУ-переводом.

Допустим, что L — КС-язык, сильно характеризующий перевод $\tau(T)$. Можно считать, что $\Delta' = \{c', 0', 1'\}$, $L \subseteq \{0, 1, c\}^* \cup \Delta'^*$ и h_1, h_2 — соответствующие гомоморфизмы. Для любых $u, v \in \{0, 1\}^*$ найдется такая цепочка $z_{uv} \in L$, что $h_1(z_{uv}) = ucv$ и $h_2(z_{uv}) = vcu$. Рассмотрим отдельно два случая, связанные с расположением символов c и c' в цепочках z_{uv} .

Случай 1: Для каждой цепочки u существует такая цепочка v , что c предшествует c' в z_{uv} . Пусть R — регулярное мно-

жество $\{0, 1, 0', 1'\}^* c \{0, 1, 0', 1'\}^* c' \{0, 1, 0', 1'\}^*$. Тогда $L \cap R$ — КС-язык, поскольку класс КС-языков замкнут относительно пересечения с регулярными множествами. Заметим, что множество $L \cap R$ состоит из тех цепочек языка L , в которых c предшествует c' . Пусть M — конечный преобразователь, который до тех пор, пока не прочтет c , передает на выход нули и единицы, пропуская символы со штрихами. После чтения символа c он не выдает ничего, пока не достигнет c' . После этого M прочтает на выходе 0, прочитав на входе 0', и 1 вместо 1', пропуская нули и единицы. Тогда язык $M(L \cap R)$ должен быть КС-языком, так как класс КС-языков замкнут относительно конечных преобразований, но в данном случае $M(L \cap R) = \{uu \mid u \in \{0, 1\}^*\}$, а это, как показано в примере 2.41, не КС-язык.

Случай 2: Для некоторой цепочки u не существует ни одной цепочки v , в которой символ c предшествует c' в z_{uv} . Тогда для каждой цепочки v можно найти такую цепочку u , что c' предшествует c в z_{uv} . Рассуждая, как в случае 1, можно показать, что если бы язык L был контекстно-свободным, то и язык $\{uv \mid v \in \{0, 1\}^*\}$ был бы тоже контекстно-свободным. Это рассуждение мы оставляем в качестве упражнения.

Итак, перевод $\tau(T)$ не сильно характеризуется никаким КС-языком и, следовательно, не является простым СУ-переводом. \square

Пусть \mathcal{T} , обозначает класс регулярных переводов, \mathcal{T}_s — класс простых СУ-переводов и \mathcal{T} — класс СУ-переводов. Из приведенных выше примеров вытекает, что справедлива следующая теорема.

Теорема 3.5. $\mathcal{T} \subseteq \mathcal{T}_s \subseteq \mathcal{T}$.

Доказательство. $\mathcal{T}_s \subseteq \mathcal{T}$ по определению, $\mathcal{T} \subseteq \mathcal{T}_s$, так как конечный преобразователь — частный случай МП-автомата. Из примеров 3.16 и 3.17 следует, что включения собственные. \square

3.2.2. Свойства простых СУ-переводов

С помощью понятия характеризующего языка можно доказать аналоги многих теорем о нормальных формах, приведенных в разд. 2.6. Здесь мы докажем два из этих результатов, а остальные оставим в качестве упражнений. Первый из них — аналог теоремы о нормальной форме Хомского.

Теорема 3.6. Пусть T — простой СУ-перевод. Тогда $T = \tau(T_1)$, где $T_1 = (N, \Sigma, \Delta, R, S)$ — простая СУ-схема, у которой каждое правило из R имеет один из следующих видов:

(1) $A \rightarrow BC, BC$, где A, B и C —(не обязательно различные) нетерминалы из N ,

(2) $A \rightarrow a, b$, где один из символов a и b есть e , а другой принадлежит Σ или Δ соответственно,

(3) $S \rightarrow e, e$, если $(e, e) \in T$, причем S не встречается в правых частях правил.

Доказательство. Применить конструкцию теоремы 3.4 к грамматике в нормальной форме Хомского. \square

Второй результат—аналог теоремы о нормальной форме Грайбах.

Теорема 3.7. Пусть T —простой СУ-перевод. Тогда $T = \tau(T_1)$, где $T_1 = (N, \Sigma, \Delta, R, S)$ —простая СУ-схема, у которой каждое правило из R имеет вид $A \rightarrow a\alpha, b\alpha$, где $\alpha \in N^*$, один из символов a и b есть e , а другой принадлежит Σ или Δ (с тем же исключением, как в случае (3) предыдущей теоремы).

Доказательство. Применить конструкцию теоремы 3.4 к грамматике в нормальной форме Грайбах. \square

Отметим, что в теоремах 3.6 и 3.7 нельзя сделать так, чтобы одновременно было $a \in \Sigma$ и $b \in \Delta$. Тогда перевод сохранял бы длину слова, а для простых СУ-переводов это не всегда так.

3.2.3. Иерархия СУ-переводов

Главный результат этого раздела состоит в том, что для произвольных СУ-переводов нет аналога нормальной формы Хомского. За одним только исключением, всякий раз, когда мы увеличиваем число нетерминалов, допустимое для правых частей правил СУ-схем, соответствующий класс СУ-переводов становится более широким. Попутно мы докажем и другие интересные свойства СУ-переводов.

Определение. Пусть $T = (N, \Sigma, \Delta, R, S)$ —СУ-схема. Будем говорить, что T имеет порядок k , если ни в одном правиле $A \rightarrow \alpha, \beta$ из R цепочка α (а следовательно, и β) не содержит более k вхождений нетерминалов. Будем также говорить, что перевод $\tau(T)$ имеет порядок k . Пусть \mathcal{T}_k —класс всех СУ-переводов порядка k .

Очевидно, что $\mathcal{T}_1 \subseteq \mathcal{T}_2 \subseteq \dots \subseteq \mathcal{T}_i \subseteq \dots$. Мы покажем, что каждое из этих включений собственное, за исключением того, что $\mathcal{T}_3 = \mathcal{T}_2$. Для этого нам понадобится ряд предварительных результатов.

Лемма 3.5. $\mathcal{T}_1 \subsetneq \mathcal{T}_2$.

Доказательство. Легко показать, что область определения СУ-перевода порядка 1 является линейным КС-языком. По теореме 3.6 каждый простой СУ-перевод имеет порядок 2 и каждый КС-язык служит областью определения некоторого простого СУ-перевода (хотя бы тождественного, для которого этот язык—область его определения). Так как линейные языки об разуют собственный подкласс КС-языков (упр. 2.6.7), то включение \mathcal{T}_1 в \mathcal{T}_2 собственное. \square

Для СУ-схем существуют разные нормальные формы. Мы утверждаем, что из СУ-схемы, как и из КС-грамматики, можно устраниТЬ бесполезные нетерминалы. Кроме того, для СУ-схем существует нормальная форма, отчасти похожая на нормальную форму Хомского, а именно: все правила можно привести к такому виду, что каждая правая часть либо целиком состоит из нетерминалов, либо не содержит ни одного нетерминала.

Определение. Нетерминал A в СУ-схеме $T = (N, \Sigma, \Delta, R, S)$ называется бесполезным, если либо

- (1) нет таких $x \in \Sigma^*$ и $y \in \Delta^*$, что $(A, A) \Rightarrow^*(x, y)$, либо
- (2) ни для каких $\alpha_1, \alpha_2 \in (N \cup \Sigma)^*$ и $\beta_1, \beta_2 \in (N \cup \Delta)^*$ не существует вывода $(S, S) \Rightarrow^*(\alpha_1 A \alpha_2, \beta_1 A \beta_2)$.

Лемма 3.6. Каждый СУ-перевод порядка k определяется СУ-схемой порядка k , не содержащей бесполезных нетерминалов.

Доказательство. Упражнение, аналогичное теореме 2.13. \square

Лемма 3.7. Каждый СУ-перевод порядка $k \geq 2$ определяется такой СУ-схемой $T_1 = (N, \Sigma, \Delta, R, S)$, что если $A \rightarrow \alpha, \beta$ принадлежит R , то либо

- (1) $\alpha, \beta \in N^*$, либо
- (2) $\alpha \in \Sigma^*, \beta \in \Delta^*$.

Кроме того, T_1 не содержит бесполезных нетерминалов.

Доказательство. Пусть $T_2 = (N', \Sigma, \Delta, R', S)$ —СУ-схема, не содержащая бесполезных нетерминалов, и $\tau(T_2) = T$. Построим R по R' следующим образом. Пусть

$$A \rightarrow x_0 B_1 x_1 \dots B_k x_k, \quad y_0 C_1 y_1 \dots C_k y_k$$

—правило из R' , причем $k > 0$. Пусть π —такая перестановка чисел $1, \dots, n$, что нетерминал B_i соответствует нетерминалу

$C_{\pi(i)}$. Введем новые нетерминалы A' , D_1, \dots, D_k и E_0, \dots, E_k и заменим это правило правилами

$$A \rightarrow E_0 A', \quad E_0 A'$$

$$E_0 \rightarrow x_0, \quad y_0$$

$A' \rightarrow D_1 \dots D_k, \quad D'_1 \dots D'_k$, где $D_i = D'_{\pi(i)}$ для $1 \leq i \leq k$

$D_i \rightarrow B_i E_i, \quad B_i E_i$ для $1 \leq i \leq k$

$E_i \rightarrow x_i, \quad y_{\pi(i)}$ для $1 \leq i \leq k$

Например, если правило имеет вид

$$A \rightarrow x_0 B_1 x_1 B_2 x_2 B_3 x_3, \quad y_0 B_3 y_1 B_2 y_2 B_2 y_3$$

то $\pi = (2, 3, 1)$ и оно заменяется правилами

$$A \rightarrow E_0 A', \quad E_0 A'$$

$$E_0 \rightarrow x_0, \quad y_0$$

$A' \rightarrow D_1 D_2 D_3, \quad D_3 D_1 D_2$

$D_i \rightarrow B_i E_i, \quad B_i E_i$ для $i = 1, 2, 3$

$E_1 \rightarrow x_1, \quad y_2$

$E_2 \rightarrow x_2, \quad y_3$

$E_3 \rightarrow x_3, \quad y_1$

Так как D_i и E_i для каждого i имеют только по одному правилу, легко видеть, что все новые правила в совокупности действуют так же, как правило, которое они заменяют. Правила из R' , не содержащие нетерминалов в правых частях, прямо помещаются в R . Обозначим через N множество N' вместе с новыми нетерминалами. Тогда $\tau(T_2) = \tau(T_1)$ и T_2 удовлетворяет условиям леммы. \square

Лемма 3.8. $\mathcal{T}_2 = \mathcal{T}_3$.

Доказательство. В силу леммы 3.7 достаточно показать, как заменить правило вида $A \rightarrow B_1 B_2 B_3, C_1 C_2 C_3$ двумя правилами, у которых каждая компонента правой части содержит два нетерминала. Пусть π — перестановка, при которой B_i соответствует $C_{\pi(i)}$. Таких перестановок может быть шесть. В каждом случае можно ввести новый нетерминал D и заменить интересующее нас правило двумя правилами, как показано на рис. 3.6.

Легко проверить, что в каждом случае новые правила в совокупности действуют так же, как старое. \square

Лемма 3.9. Каждый СУ-перевод порядка $k \geq 2$ определяется СУ-схемой $T = (N, \Sigma, \Delta, R, S)$, удовлетворяющей лемме 3.7 и следующим условиям:

- (1) в R нет правила вида $A \rightarrow B, B$,
 (2) в R нет правил вида $A \rightarrow e, e$ (за исключением случая $A = S$, но тогда символ S не должен встречаться в правых частях правил).

Доказательство. Упражнение, аналогичное теоремам 2.14 и 2.15. \square

$\pi(1)$	$\pi(2)$	$\pi(3)$	Правила
1	2	3	$A \rightarrow B_1 D, B_1 D \quad D \rightarrow B_2 B_3, B_2 B_3$
1	3	2	$A \rightarrow B_1 D, B_1 D \quad D \rightarrow B_2 B_3, B_3 B_2$
2	1	3	$A \rightarrow DB_3, DB_3 \quad D \rightarrow B_1 B_2, B_2 B_1$
2	3	1	$A \rightarrow DB_3, B_3 D \quad D \rightarrow B_1 B_2, B_1 B_2$
3	1	2	$A \rightarrow B_1 D, DB_1 \quad D \rightarrow B_2 B_3, B_2 B_3$
3	2	1	$A \rightarrow B_1 D, DB_1 \quad D \rightarrow B_2 B_3, B_3 B_2$

Рис. 3.6. Новые правила.

Теперь определим такое семейство переводов T_k для $k \geq 4$, что T_k имеет порядок k , но не $k-1$, а затем докажем ряд лемм, из которых это будет следовать.

Определение. Пусть $k \geq 4$. Пусть Σ_k обозначает — только до конца этого раздела — множество $\{a_1, \dots, a_k\}$. Определим перестановку π_k для четного k равенствами

$$\pi_k(i) = \begin{cases} \frac{k+i+1}{2} & \text{если } i \text{ нечетно} \\ \frac{i}{2} & \text{если } i \text{ четно} \end{cases}$$

Например, $\pi_4 = (3, 1, 4, 2)$ и $\pi_6 = (4, 1, 5, 2, 6, 3)$. Определим π_k для нечетного k равенствами

$$\pi_k(i) = \begin{cases} \frac{k+1}{2} & \text{если } i=1 \\ k - \frac{i}{2} + 1 & \text{если } i \text{ четно} \\ \frac{i-1}{2} & \text{если } i \text{ нечетно и } i \neq 1 \end{cases}$$

Например, $\pi_5 = (3, 5, 1, 4, 2)$ и $\pi_7 = (4, 7, 1, 6, 2, 5, 3)$.

Пусть T_k — взаимно однозначное отображение, которое переводит

$$a_1^{i_1} a_2^{i_2} \dots a_k^{i_k} \quad \text{в} \quad a_{\pi(1)}^{i_{\pi(1)}} a_{\pi(2)}^{i_{\pi(2)}} \dots a_{\pi(k)}^{i_{\pi(k)}}$$

Например, если a_1, a_2, a_3, a_4 — это a, b, c, d , то

$$T_4 = \{(a^i b^j c^k d^l, a^{\pi(1)} b^{\pi(2)} c^{\pi(3)} d^{\pi(4)}) \mid i, j, k, l \geq 0\}$$

В дальнейшем будем предполагать, что $k \geq 4$ — фиксированное целое число и существует СУ-схема $T = (N, \Sigma_k, \Sigma_k, R, S)$ порядка $k-1$, определяющая T_k . Без потери общности можно считать, что T удовлетворяет лемме 3.9 и, значит, леммам 3.6 и 3.7. Мы докажем, получив противоречие, что T не может существовать.

Определение. Пусть $\Sigma \subseteq \Sigma_k$ и $A \in N$. (Напомним, что имеется в виду СУ-схема T , определяющая по предположению перевод T_k .) Множество Σ будем называть (A, d) -ограниченным в области определения (соответственно в множестве значений), если для каждой пары (x, y) , для которой $(A, A) \Rightarrow_t^*(x, y)$, существует символ $a \in \Sigma$, содержащийся не более d раз в x (соответственно в y). Если множество Σ не является (A, d) -ограниченным в области определения (множестве значений) ни для какого d , то будем говорить, что A покрывает Σ в области определения (множестве значений).

Лемма 3.10. Если нетерминал A покрывает Σ в области определения, то он покрывает Σ в множестве значений, и обратно.

Доказательство. Допустим, что A покрывает множество Σ в области определения, но оно (A, d) -ограничено в множестве значений. По лемме 3.6 найдутся такие $w_1, w_2, w_3, w_4 \in \Sigma_k^*$, что $(S, S) \Rightarrow^* (w_1 A w_2, w_3 A w_4)$. Пусть $m = |w_3 w_4|$. Так как A покрывает Σ в области определения, то найдутся такие $w_5, w_6 \in \Sigma_k^*$, что $(A, A) \Rightarrow^* (w_5, w_6)$ и любой символ $a \in \Sigma$ содержится в w_5 более $m+d$ раз. Но так как Σ (A, d) -ограничено в множестве значений, то найдется символ $b \in \Sigma$, содержащийся не более d раз в w_6 . При этих условиях пара $(w_1 w_5 w_2, w_3 w_6 w_4)$ должна принадлежать T_k , хотя b входит в $w_1 w_5 w_2$ более $m+d$ раз, а в $w_3 w_6 w_4$ — не более $m+d$ раз.

Полученное противоречие показывает, что если нетерминал A покрывает Σ в области определения, то он покрывает Σ также и в множестве значений. Обратное утверждение доказывается по симметрии. \square

Лемма 3.10 дает право говорить, что A покрывает Σ , не упоминая области определения и множества значений.

Лемма 3.11. Пусть A покрывает Σ_k . Тогда существуют правило $A \rightarrow B_1 \dots B_m, C_1 \dots C_m$ из R и такие множества $\Theta_1, \dots, \Theta_m$, объединение которых совпадает с Σ_k , что B_i покрывает Θ_i ($1 \leq i \leq m$).

Доказательство. Пусть d_0 — наибольшее из таких коначных чисел, что для некоторых $\Sigma \subseteq \Sigma_k$ и B_i ($1 \leq i \leq m$) множество Σ (B_i, d_0) -ограничено, но не (B_i, d_0-1) -ограничено. Ясно, что такое d_0 существует. Положим $d_1 = d_0(k-1)+1$. Тогда должны

найтись такие цепочки $x, y \in \Sigma_k^*$, что $(A, A) \Rightarrow^*(x, y)$ и каждый символ $a \in \Sigma_k$ входит в каждую из них по крайней мере d_1 раз (в противном случае Σ_k было бы (A, d_1) -ограниченным).

Пусть первый шаг вывода $(A, A) \Rightarrow^*(x, y)$ был $(A, A) \Rightarrow (B_1 \dots B_m, C_1 \dots C_m)$. Так как по предположению T имеет порядок $k-1$, то $m \leq k-1$. Можно записать x в виде $x_1 \dots x_m$, причем $(B_i, B_j) \Rightarrow^*(x_i, y_i)$ для некоторой цепочки y_i . Для произвольного $a \in \Sigma_k$ должна найтись хотя бы одна цепочка x_i , содержащая a более d_0 раз, потому что в противном случае цепочка x содержала бы a не более $d_0(k-1)-d_1+1$ раз. Пусть $\Theta_i \subseteq \Sigma_k$ состоит из символов, которые входят в x_i более d_0 раз. Из предыдущего ясно, что $\Theta_1 \cup \Theta_2 \cup \dots \cup \Theta_m = \Sigma_k$. Тогда B_i для каждого i покрывает Θ_i , так как в противном случае какое-то множество Θ_i было бы (B_i, d) -ограниченным для некоторого $d > d_0$. Это невозможно в силу нашего выбора числа d_0 . \square

Определение. Пусть a_i, a_j и a_t — различные элементы множества Σ_k . Будем говорить, что a_j находится между a_i и a_t , если либо

- (1) $i < j < t$, либо
- (2) $\pi_k(i) < \pi_k(j) < \pi_k(t)$.

Таким образом, символ формально находится между двумя другими символами, если он действительно расположен между ними в какой-нибудь цепочке, принадлежащей области определения или множеству значений перевода T_k .

Лемма 3.12. Пусть A покрывает Σ_k и $A \rightarrow B_1 \dots B_m, C_1 \dots C_m$ — правило, удовлетворяющее лемме 3.11. Если B_i покрывает $\{a_r\}$ и $\{a_s\}$, а a_t находится между a_r и a_s , то B_i покрывает $\{a_t\}$ и B_j ни для какого $j \neq i$ не покрывает $\{a_t\}$.

Доказательство. Допустим, что $r < t < s$ и B_j покрывает $\{a_t\}$, причем $j \neq i$. Рассмотрим отдельно случаи $j < i$ и $j > i$.

Случай 1: $j < i$. Так как во входной грамматике схемы T из B_i выводится цепочка, содержащая a_r , а из B_j выводится цепочка, содержащая a_t , то $(A, A) \Rightarrow^*(x, y)$, где x содержит вхождение a_t , предшествующее вхождению a_r . Тогда по лемме 3.6 в области определения перевода T_k есть цепочка, которой там не должно быть.

Случай 2: $j > i$. Предположим, что из B_i выводится цепочка, содержащая a_s . Можно аналогично найти в области определения перевода T_k цепочку, в которой a_s предшествует a_t .

Полученное противоречие позволяет исключить возможность того, что $r < t < s$. Единственная оставшаяся возможность —

$\pi_k(r) < \pi_k(t) < \pi_k(s)$ — рассматривается аналогично с привлечением множества значений перевода T_k . Таким образом, B_j ни для какого $j \neq i$ не покрывает $\{a_t\}$. Если B_j покрывает Σ , содержащее a_t , то B_j , разумеется, покрывает $\{a_t\}$. Следовательно, по лемме 3.11 B_i покрывает некоторое множество, содержащее a_t , и, значит, покрывает $\{a_t\}$. \square

Лемма 3.13. Если A покрывает Σ_k ($k \geq 4$), то существует такое правило $A \rightarrow B_1 \dots B_m, C_1 \dots C_m$ и такой индекс i , $1 \leq i \leq m$, что B_i покрывает Σ_k .

Доказательство. Рассмотрим случай, когда k четно. Случай нечетного k рассматривается аналогично и оставляется в качестве упражнения. Пусть $A \rightarrow B_1 \dots B_m, C_1 \dots C_m$ — правило, удовлетворяющее лемме 3.11. Так как по предположению $m \leq k-1$, то некоторый нетерминал B_i должен покрывать два элемента множества Σ_k , скажем $\{a_r, a_s\}$, где $r \neq s$. Следовательно, B_i покрывает $\{a_r\}$ и $\{a_s\}$, а если a_t находится между a_r и a_s , то по лемме 3.12 B_i покрывает $\{a_t\}$ и B_j ни для какого $j \neq i$ не покрывает $\{a_t\}$.

Рассмотрим множество значений перевода T_k . Если нетерминал B_i покрывает $\{a_{k/2}\}$ и $\{a_{k/2+1}\}$, то он покрывает $\{a\}$ для всех $a \in \Sigma_k$ и никакой другой нетерминал B_j не покрывает никакого $\{a\}$. В силу леммы 3.11 B_i покрывает Σ_k . Далее, если B_i покрывает $\{a_m\}$ и $\{a_n\}$, где $m \leq k/2$ и $n > k/2$, то, исследуя область определения, убеждаемся, что B_i покрывает $\{a_{k/2}\}$ и $\{a_{k/2+1}\}$.

Таким образом, если одно из чисел r и s не превосходит $k/2$, а другое превосходит $k/2$, то нужный результат получается сразу.

Остались случаи $r \leq k/2, s \leq k/2$ и $r > k/2, s > k/2$. Но в множестве значений для любых различных r и s , не превосходящих $k/2$, между a_r и a_s найдется символ a_t , для которого $t > k/2$. Аналогично, если $r > k/2$ и $s > k/2$, то между ними найдется символ a_t , для которого $t \leq k/2$. В любом случае лемма доказана. \square

Лемма 3.14. $T_k \in \mathcal{T}_k - \mathcal{T}_{k-1}$ для $k \geq 4$.

Доказательство. Ясно, что $T_k \in \mathcal{T}_k$. Остается показать, что вопреки предположению для T_k не существует СУ-схемы T порядка $k-1$. Так как S , разумеется, покрывает Σ_k , то по лемме 3.13 в N найдется такая последовательность нетерминалов $A_0, A_1, \dots, A_{\#N}$, что $A_0 = S$ и для каждого $0 \leq i \leq \#N$ существует правило $A_i \rightarrow \alpha_i A_{i+1} \beta_i, \gamma_i A_{i+1} \delta_i$, причем A_i покрывает Σ_k . Так как все A_i не могут быть различными, найдутся такие $i < j$, что $A_i = A_j$. По лемме 3.6 можно найти такие

w_1, \dots, w_{10} , что для всех $p \geq 0$

$$\begin{aligned} (S, S) &\Rightarrow^* (w_1 A_i w_2, w_3 A_i w_4) \\ &\Rightarrow^* (w_1 w_5 A_i w_6 w_2, w_3 w_7 A_i w_8 w_4) \\ &\vdots \\ &\Rightarrow^* (w_1 w_5^p A_i w_6^p w_2, w_3 w_7^p A_i w_8^p w_4) \\ &\Rightarrow^* (w_1 w_5^p w_9 w_6^p w_2, w_3 w_7^p w_{10} w_8^p w_4) \end{aligned}$$

По лемме 3.9(1) не все $\alpha_i, \beta_i, \gamma_i$ и δ_i пустые, а по лемме 3.9(2) не все w_5, w_6, w_7 и w_8 пустые.

Для каждого $a \in \Sigma_k$ цепочки $w_5 w_6$ и $w_7 w_8$ должны иметь одно и то же число вхождений a , потому что иначе в переводе $T(T)$ была бы пара, которой нет в T_k . Так как A_i покрывает Σ_k , то если бы w_5, w_6, w_7 и w_8 содержали какой-нибудь символ, кроме a_i или a_k , можно было бы подобрать w_9 так, чтобы получить пару, не принадлежащую T_k . Следовательно, в w_7 или w_8 входит a_i или a_k . Так как A_i покрывает Σ_k , можно выбрать w_{10} так, чтобы получилась пара, не входящая в T_k . Итак, T не существует и $T_k \notin \mathcal{T}_{k-1}$. \square

Теорема 3.8. За исключением $k=2$, для любого $k \geq 1$ класс \mathcal{T}_{k+1} собственно включает \mathcal{T}_k .

Доказательство. Случай $k=1$ доказан в лемме 3.5. Остальные случаи охватываются леммой 3.14. \square

Интересное практическое следствие теоремы 3.8 состоит в том, что, хотя кажется заманчивым сконструировать такую систему построения компиляторов, чтобы входная грамматика была в нормальной форме Хомского, эта система не в состоянии выполнить любую синтаксически управляемую трансляцию, а более общая система смогла бы это сделать. Однако похоже, что на практике СУ-переводы в худшем случае будут припадлежать классу \mathcal{T}_3 (и, следовательно, \mathcal{T}_2).

УПРАЖНЕНИЯ

***3.2.1.** Пусть T — СУ-перевод. Покажите, что существует такая константа c , что для каждой цепочки x из области его определения найдется такая цепочка y , что $(x, y) \in T$ и $|y| \leq c(|x| + 1)$.

***3.2.2.** (a) Покажите, что если T_1 — регулярный перевод и T_2 — СУ-перевод, то $T_1 \circ T_2 = \{(x, z) | (x, y) \in T_1 \text{ и } (y, z) \in T_2 \text{ для некоторой цепочки } y\}$ — СУ-перевод¹.

¹ Часто эта операция над переводами, называемая композицией, записывается с операндами в обратном порядке, так что в нашем определении надо было бы писать $T_2 \circ T_1$, а не $T_1 \circ T_2$. Мы не изменим нашей записи, так как она выглядит более естественной.

(б) Покажите, что если перевод T_2 простой, то перевод $T_1 \circ T_2$ тоже простой.

3.2.3. (а) Покажите, что если T — СУ-перевод, то T^{-1} — тоже СУ-перевод.

(б) Покажите, что если перевод T простой, то перевод T^{-1} тоже простой.

*3.2.4. (а) Пусть T_1 — регулярный перевод, а T_2 — СУ-перевод. Покажите, что $T_2 \circ T_1$ — СУ-перевод.

(б) Покажите, что если перевод T_2 простой, то перевод $T_2 \circ T_1$ тоже простой.

3.2.5. Найдите сильно характеризующие языки для СУ-переводов из

- (а) примера 3.5,
- (б) примера 3.7,
- (в) примера 3.12.

3.2.6. Для СУ-перевода из упр. 3.2.5 найдите язык, характеризующий его, но не сильно.

3.2.7. Дополните доказательство леммы 3.4.

3.2.8. Дополните доказательство случая 2 в примере 3.17.

3.2.9. Покажите, что каждый простой СУ-перевод определяется простой СУ-схемой, не содержащей бесполезных нетерминалов.

3.2.10. Пусть T_1 — простой СУ-перевод, а T_2 — регулярный перевод. Всегда ли $T_1 \cap T_2$ — простой СУ-перевод?

3.2.11. Докажите лемму 3.6.

3.2.12. Докажите лемму 3.9.

3.2.13. Постройте СУ-схему порядка k , определяющую T_k .

3.2.14. Пусть $T = (N, \Sigma, \Delta, R, S)$, где $N = \{S, A, B, C, D\}$, $\Sigma = \{a, b, c, d\}$ и R состоит из правил

$$\begin{aligned} A &\rightarrow aA, aA \\ A &\rightarrow e, e \\ B &\rightarrow bB, bB \\ B &\rightarrow e, e \\ C &\rightarrow cC, cC \\ c &\rightarrow e, e \\ D &\rightarrow dD, dD \\ D &\rightarrow e, e \end{aligned}$$

и одного из приведенных ниже. Найдите наименьший порядок перевода $\tau(T)$ для каждого из этих дополнительных правил:

- | | |
|-----------------------------------|-----------------------------------|
| (а) $S \rightarrow ABCD$, $ABCD$ | (в) $S \rightarrow ABCD$, $DBCA$ |
| (б) $S \rightarrow ABCD$, $BCDA$ | (г) $S \rightarrow ABCD$, $BDAC$ |

3.2.15. Покажите, что если T определяется ДМП-преобразователем, то T сильно характеризуется детерминированным КС-языком.

3.2.16. Верно ли обращение упражнения 3.2.15?

3.2.17. Докажите следствия теорем 3.3 и 3.4.

Замечания по литературе

Понятие характеризующего языка и результаты разд. 3.2.1 и 3.2.2 взяты из работы Ахо и Ульмана [1969 б], а результаты разд. 3.2.3 — из работы Ахо и Ульмана [1969а].

3.3. ЛЕКСИЧЕСКИЙ АНАЛИЗ

Лексический анализ образует первый этап процесса компиляции. На этом этапе символы, составляющие исходную программу, считываются и группируются в отдельные лексические элементы, называемые лексемами. Лексический анализ важен для процесса компиляции по нескольким причинам. Наибольшее значение имеет, по-видимому, то, что замена в программе идентификаторов и констант лексемами делает представление программы удобнее для дальнейшей обработки. Далее, лексический анализ уменьшает длину программы, устранивая из исходного ее представления несущественные пробелы и комментарии. На следующих стадиях компиляции компилятор может несколько раз просматривать внутреннее представление программы. Поэтому, уменьшая с помощью лексического анализа длину этого представления, можно сократить общее время процесса компиляции.

Во многих ситуациях выбор конструкций, которые будут выделяться как лексемы, довольно произволен. Например, если в языке разрешены комплексные константы вида

\langle комплексная константа $\rangle \rightarrow \langle$ вещественное \rangle, \langle вещественное \rangle

то возможны две стратегии. Можно трактовать \langle вещественное \rangle как лексический элемент и отложить распознавание конструкции (\langle вещественное \rangle, \langle вещественное \rangle) как комплексной константы до фазы синтаксического анализа. Другая стратегия состоит в том, чтобы с помощью более сложного лексического анализатора распознать эту конструкцию как комплексную константу на лексическом уровне и передать тип лексемы синтаксическому анализатору. Важно также отметить, что если в вычислительном центре изменяется принятное в нем множество терминальных символов, то это отражается лишь на лексическом анализе.

Большую часть того, что происходит в течение лексического анализа, можно моделировать с помощью конечных преобразователей, работающих последовательно или параллельно. Напри-

мер, лексический анализатор может состоять из ряда последовательно соединенных конечных преобразователей. Первый преобразователь в этой цепи может устранять из исходной программы все несущественные пробелы, второй ликвидирует комментарии, третий ищет константы и т. д. Другая возможность — завести набор конечных преобразователей, каждый из которых ищет определенную лексическую конструкцию.

В этом разделе мы рассмотрим методы, которые можно использовать при построении эффективных лексических анализаторов. Как уже отмечалось в разд. 1.2.1, лексические анализаторы бывают по существу двух видов — прямые и непрямые. Мы покажем, как по регулярным выражениям, описывающим соответствующие лексемы, строятся анализаторы обоих видов.

3.3.1. Язык расширенных регулярных выражений

Множества допустимых цепочек знаков, образующих идентификаторы и другие лексемы языков программирования, почти всегда оказываются регулярными. Например, идентификаторы Фортрана описываются как цепочки, „содержащие от одной до шести латинских букв или цифр и начинающиеся буквой“. Очевидно, что это множество регулярно и его описывает регулярное выражение

$$(A + \dots + Z)(e + (A + \dots + Z + 0 + \dots + 9) \\ (e + (A + \dots + Z + 0 + \dots + 9))(e + (A + \dots + Z + 0 + \dots + 9)) \\ (e + (A + \dots + Z + 0 + \dots + 9))(e + A + \dots + Z + 0 + \dots + 9)))$$

Это выражение чрезвычайно громоздко, так что имеет смысл ввести расширенные регулярии выражения, удобнее описывающие это и другие интересные с практической точки зрения регулярные выражения.

Определение. Расширенные регулярные выражения и множества, которые они обозначают, определяются рекурсивно следующим образом.

- (1) Если R — регулярное выражение, то оно расширенное и обозначает множество R ¹⁾.
- (2) Если R — расширенное регулярное выражение, то
 - (а) R^+ — расширенное регулярное выражение, обозначающее множество RR^* ,
 - (б) R^{**} — расширенное регулярное выражение, обозначающее множество $\{e\} \cup R \cup RR \cup \dots \cup R^n$,

¹⁾ Напомним, что мы не различаем регулярное выражение и обозначаемое им множество, когда ясно, о чем идет речь.

(в) R^{+n} — расширенное регулярное выражение, обозначающее множество $R \cup RR \cup \dots \cup R^n$.

(3) Если R_1 и R_2 — расширенные регулярии выражения, то $R_1 - R_2$ и $R_1 \cap R_2$ — расширенные регулярии выражения, обозначающие соответственно множества $\{x | x \in R_1 \text{ и } x \notin R_2\}$ и $\{x | x \in R_1 \text{ и } x \in R_2\}$.

(4) Ничто другое не является расширенным регулярии выражением.

Соглашение. В расширенных регулярии выражениях мы будем обозначать бинарную операцию объединения знаком $|$ вместо $+$, чтобы различие между этой операцией и унарными операциями $^+$ и $^{+n}$ было более заметным.

Другое полезное качество аппарата, служащего для определения регулярии множеств, — это возможность давать им имена. Нужно позаботиться о том, чтобы определения, в которых участвуют имена, не приводили к порочному кругу или чтобы не получилась по существу система определяющих уравнений, подобная системе, рассмотренной в разд. 2.6, с помощью которой можно определить любой КС-язык. В принципе нет ничего плохого в том, чтобы при определении лексем использовать всю мощь определяющих уравнений (или распознавать лексемы с помощью МП-преобразователя). Однако, как правило, лексический анализатор имеет простую структуру — обычно это структура конечного автомата. Поэтому мы предпочитаем пользоваться механизмом, который может определять только регулярии множества и по которому легко строить соответствующие конечные преобразователи. Существенно контекстно-свободные аспекты языка анализируются на этапе синтаксического разбора, гораздо более сложном, чем лексическая фаза.

Определение. Последовательностью регулярии определений в алфавите Σ назовем список определений $A_1 = R_1, A_2 = R_2, \dots, A_n = R_n$, где A_1, \dots, A_n — различные символы, не принадлежащие Σ , и R_i — расширенное регулярие выражение в алфавите $\Sigma \cup \{A_1, \dots, A_{i-1}\}$ ($1 \leq i \leq n$). Для $1 \leq i \leq n$ определим расширенное регулярие выражение R'_i в алфавите Σ :

- (1) $R'_1 = R_1$,
- (2) R'_i получается из R_i заменой каждого вхождения A_j для $1 \leq j < i$ выражением R'_j .

Множество, обозначаемое символом A_i , это множество, обозначаемое выражением R'_i .

Должно быть ясно, что множества, обозначаемые расширенными регулярии выражениями и последовательностями ре-

гулярных определений, регулярны. Доказательство оставляем в качестве упражнения.

Пример 3.18. Идентификаторы Фортрана задаются последовательностью регулярных определений

$$\langle \text{буква} \rangle = A | B | \dots | Z$$

$$\langle \text{цифра} \rangle = 0 | 1 | \dots | 9$$

$$\langle \text{идентификатор} \rangle = \langle \text{буква} \rangle (\langle \text{буква} \rangle | \langle \text{цифра} \rangle)^{*5}$$

Если мы не хотим допустить, чтобы в качестве идентификаторов использовались ключевые слова Фортрана, то нам придется пересмотреть определение $\langle \text{идентификаторов} \rangle$, чтобы исключить соответствующие цепочки. Тогда оно должно читаться так:

$$\langle \text{идентификатор} \rangle = (\langle \text{буква} \rangle (\langle \text{буква} \rangle | \langle \text{цифра} \rangle)^{*5}) - (\text{DO} | \text{IF} | \dots) \quad \square$$

Пример 3.19. Общепринятый вид записи вещественных констант (например, 3.14159, —682, 6.6E—29 и т. д.) задается следующей последовательностью регуляриых определений¹⁾:

$$\langle \text{цифра} \rangle = 0 | 1 | \dots | 9$$

$$\langle \text{знак} \rangle = + | - | e$$

$$\langle \text{целое} \rangle = \langle \text{знак} \rangle \langle \text{цифра} \rangle^{+}$$

$$\langle \text{десятичное} \rangle = \langle \text{знак} \rangle (\langle \text{цифра} \rangle^{*}. \langle \text{цифра} \rangle^{+} | \langle \text{цифра} \rangle^{+}. \langle \text{цифра} \rangle^{*})$$

$$\langle \text{константа} \rangle = \langle \text{целое} \rangle | \langle \text{десятичное} \rangle | \langle \text{десятичное} \rangle E \langle \text{целое} \rangle \quad \square$$

3.3.2. Непрямой лексический анализ

При непрямом лексическом анализе требуется, прочитав цепочку знаков, определить, появилась ли подцепочка, образующая некоторую конкретную лексему. Если множество возможных цепочек, которые могут образовывать эту лексему, обозначается, как это обычно бывает, регулярным выражением, то проблему построения непрямого лексического анализатора для данной лексемы можно представлять себе как проблему реализации конечного преобразователя. Конечный преобразователь — это почти конечный автомат (распознаватель) в том смысле, что он читает вход, не производя выхода, пока не обнаружит присутствие лексемы данного типа (т. е. достигнет заключительного состояния). Тогда он сигнализирует о том, что эта лексема появилась, и выдает на выходе цепочку символов, образующих эту лексему.

Сигналом, очевидно, служит само заключительное состояние. Однако лексическому анализатору, возможно, требуется иссле-

дователь один или более символов, стоящих после правого конца лексемы. Простой пример: мы не можем определить правый конец идентификатора Алгола, пока не встретим символ, который не является ни буквой, ни цифрой, т. е. обычно не считается частью идентификатора.

При непрямом лексическом анализе можно получить выход лексического анализатора, говорящий о появлении некоторой лексемы, а если позднее обнаружится, что лексема на самом деле отсутствует, то алгоритм синтаксического анализа осуществляет возврат к данной цепочке, обеспечивающий в конце концов работу анализатора для подходящей лексемы. Используя непрямой лексический анализ, нужно заботиться о том, чтобы с таблицами имен не выполнялись ошибочные операции. Обычно идентификатор не помещают в таблицу, пока не убедятся в том, что он действительно встречается. (В противном случае надо обеспечить механизм, удаляющий из таблиц ненужные элементы.)

Проблема непрямого лексического анализа является, таким образом, по существу проблемой построения детерминированного конечного автомата по заданному регулярному выражению и его программной реализации. Результаты гл. 2 убеждают нас в том, что такое построение возможно, хотя и требует немало работы. Нетрудно, оказывается, прямо перейти от регулярного выражения к недетерминированному конечному автомatu. Затем по теореме 2.3 можно превратить этот автомат в детерминированный либо моделировать его работу, прослеживая параллельно все возможные последовательности тактов. При прямом лексическом анализе тоже удобно начинать построение прямого лексического анализатора с компактного недетерминированного конечного автомата для каждой из лексем.

Недетерминированный автомат можно строить с помощью алгоритма, подобного тому, который в разд. 2.2 применялся для построения праволинейной грамматики по регулярному выражению. Распространить эту процедуру на все расширенные регулярные выражения не так-то просто, особенно потому, что операции пересечения и вычитания множеств требуют конструкций, связанных с детерминированными автоматами. (Очень трудно доказать, что $R_1 \cap R_2$ и $R_1 - R_2$ регулярны, если регулярны R_1 и R_2 , не обращаясь так или иначе к детерминированным автоматам. С другой стороны, для доказательства замкнутости относительно операций U , \cdot и $*$ детерминированные автоматы не требуются.) Однако операции $+$, $+^n$ и $*^n$ обрабатываются естественным образом.

¹⁾ Конкретная реализация языка обычно налагает ограничения на длину и величину констант.

Алгоритм 3.2. Построение недетерминированного конечного автомата по расширенному регулярному выражению.

Вход. Расширенное регулярное выражение R в алфавите Σ , не содержащее символа \emptyset и операций \cap и $-$.

Выход. Недетерминированный конечный автомат M , для которого $L(M) = R$.

Метод.

(1) Выполнять шаг (2) рекурсивно, начиная с выражения R . Пусть M — автомат, построенный в результате данного обращения к шагу (2).

(2) Пусть R_0 — расширенное регулярное выражение, к которому применяется этот шаг. Строится недетерминированный конечный автомат M_0 . Возможны следующие случаи:

(а) $R_0 = e$. Тогда $M_0 = (\{q\}, \Sigma, \emptyset, q, \{q\})$, где q — новое состояние.

(б) $R_0 = a$, где $a \in \Sigma$. Тогда $M_0 = (\{q_1, q_2\}, \Sigma, \delta_0, q_1, \{q_2\})$, где $\delta_0(q_1, a) = \{q_2\}$; в остальных случаях δ_0 не определена, q_1 и q_2 — новые состояния.

(в) $R_0 = R_1 | R_2$. Тогда можно применить шаг (2) к R_1 и R_2 и получить соответственно $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ и $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$, где Q_1 и Q_2 не пересекаются, а затем построить $M_0 = (Q_1 \cup Q_2 \cup \{q_0\}, \Sigma, \delta_0, q_0, F_0)$, где

(i) q_0 — новый символ,

(ii) δ_0 включает δ_1 и δ_2 , и $\delta_0(q_0, a) = \delta_1(q_1, a) \cup \delta_2(q_2, a)$,

(iii) $F_0 = F_1 \cup F_2$, если $q_1 \notin F_1$ и $q_2 \notin F_2$, и $F_0 = F_1 \cup F_2 \cup \{q_0\}$ в противном случае.

(г) $R_0 = R_1 R_2$. Применить шаг (2) к R_1 и R_2 и получить M_1 и M_2 , как в случае (в). Построить $M_0 = (Q_1 \cup Q_2, \Sigma, \delta_0, q_1, F_0)$, где

(i) δ_0 включает δ_2 ; $\delta_0(q, a) = \delta_1(q, a)$ для всех $q \in Q$ и $a \in \Sigma$, если $q \notin F_1$, и $\delta_0(q, a) = \delta_1(q, a) \cup \delta_2(q_2, a)$ в противном случае,

(ii) $F_0 = F_2$, если $q_2 \notin F_2$, и $F_0 = F_1 \cup F_2$ в противном случае.

(д) $R_0 = R_1^*$. Применить шаг (2) к R_1 и получить $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$. Построить $M_0 = (Q_1 \cup \{q_0\}, \Sigma, \delta_0, q_0, F_1 \cup \{q_0\})$, где q_0 — новый символ и δ_0 определяется соотношениями

(i) $\delta_0(q_0, a) = \delta_1(q_1, a)$,

(ii) если $q \notin F_1$, то $\delta_0(q, a) = \delta_1(q, a)$,

(iii) если $q \in F_1$, то $\delta_0(q, a) = \delta_1(q, a) \cup \delta_1(q_1, a)$.

(е) $R_0 = R_1^+$. Применить шаг (2) к R_1 и получить M_1 , как в (д). Построить $M_0 = (Q_1, \Sigma, \delta_0, q_1, F_1)$, где $\delta_0(q, a) = \delta_1(q, a)$, если $q \notin F_1$, и $\delta_0(q, a) = \delta_1(q, a) \cup \delta_1(q_1, a)$, если $q \in F_1$.

(ж) $R_0 = R_1^{*n}$. Применить шаг (2) к R_1 и получить M_1 , как в (д). Построить $M_0 = (Q_1 \times \{1, \dots, n\}, \Sigma, \delta_0, [q_1, 1], F_0)$, где

(i) если $q \notin F_1$ или $i = n$, то $\delta_0([q, i], a) = \{[p, i]\}$

$\delta_1(q, a)$ содержит $p\}$,

(ii) если $q \in F_1$ и $i < n$, то $\delta_0([q, i], a) = \{[p, i]\} \cup \delta_1(q, a)$ содержит $p\}$ и $\{[p, i+1]\}$

$\delta_1(q_1, a)$ содержит $p\}$,

(iii) $F_0 = \{[q, i] \mid q \in F_1, 1 \leq i \leq n\} \cup \{[q_1, 1]\}$.

(з) $R_0 = R_1^{+n}$. Делать то же, что и на шаге (ж), но пункт

(iii) заменить на

(iii') $F_0 = \{[q, i] \mid q \in F_1, 1 \leq i \leq n\}$. \square

Теорема 3.9. Алгоритм 3.2 дает такой недетерминированный конечный автомат M , что $L(M) = R$.

Доказательство. Упражнение на индукцию. \square

Заметим, что в пунктах (ж) и (з) алгоритма 3.2 вторую компоненту состояния автомата M_0 часто можно эффективно реализовать при программировании в виде счетчика (в том числе и при переходе к детерминированному автомата). Это возможно потому, что во многих случаях R_1 обладает префиксным свойством и цепочку из R_1^{+n} можно trivialно разбить на цепочки из R_1 . Например, R_1 может быть «цифрой», как в примере 3.18, а для каждого элемента множества «цифра» равна 1.

Пример 3.20. Сконструируем недетерминированный автомат для идентификаторов, определенных в примере 3.18. Чтобы применить шаг (2) алгоритма 3.2 к выражению, названному «идентификатором», надо применить его к выражениям «буква» и «буква» «цифра»^{*b}. Построение автомата для первого из этих выражений фактически содержит 26 применений шага (2б) и 25 применений шага (2в). Однако заметим, что после очевидного отождествления состояний¹⁾ получится автомат $(\{q_1, q_2\}, \Sigma, \delta_1, q_1, \{q_2\})$, где $\Sigma = \{A, \dots, Z, 0, \dots, 9\}$ и $\delta_1(q_1, A) = \delta_1(q_1, B) = \dots = \delta_1(q_1, Z) = \{q_2\}$.

Чтобы получить автомат для выражения «буква» «цифра»^{*5}, нужен еще один автомат для «буквы», скажем $(\{q_3, q_4\}, \Sigma, \delta_2, q_3, \{q_4\})$, и аналогичный автомат для «цифры», скажем $(\{q_5, q_6\}, \Sigma, \delta_3, q_5, \{q_6\})$. Изготавливая автомат для объединения двух последних выражений, мы добавляем новое начальное состояние q_7 и обнаруживаем, что из него нельзя достичь состояний q_3 и q_5 . Кроме того, очевидно, что q_4 и q_6 можно отождествить. В ре-

¹⁾ Два состояния недетерминированного конечного автомата можно отождествить, если они оба либо заключительные, либо незаключительные, и каждый вход переводит их в одно и то же множество состояний. Это все, что требуется здесь, но состояния недетерминированного конечного автомата можно отождествлять и при некоторых других условиях.

зультате получаем автомат $(\{q_4, q_7\}, \Sigma, \delta_4, q_7, \{q_4\})$, где $\delta_4(q_1, A) = \dots = \delta_4(q_7, Z) = \delta_4(q_7, 0) = \dots = \delta_4(q_7, 9) = \{q_4\}$.

Чтобы применить шаг (2ж), введем состояния $[q_4, i]$ и $[q_7, i]$ для $1 \leq i \leq 5$. Заключительными состояниями будут $[q_4, i]$ для $1 \leq i \leq 5$ и $[q_7, 1]$. Последнее состояние является также начальным. Получаем автомат $(Q_5, \Sigma, \delta_5, [q_7, 1], F_5)$, где F_5 то же, что и выше, а

$\delta([q_7, 1], \alpha) = \{[q_4, 1]\}$, $\delta([q_4, i], \alpha) = \{[q_4, i+1]\}$ для всех $\alpha \in \Sigma$ и $i \in \{1, 2, 3, 4\}$. Таким образом, состояния $[q_7, 2], \dots, [q_7, 5]$ недостижимы и не должны быть в Q_5 . Следовательно, $Q_5 = F_5$.

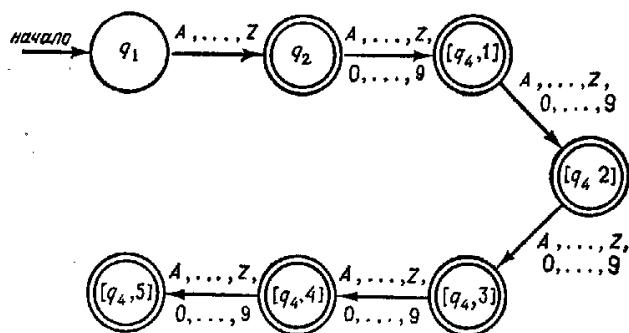


Рис. 3.7. Недетерминированный конечный автомат для идентификаторов.

Чтобы построить окончательный автомат для *(идентификатор)*, применим шаг (2г). В результате получим

$$M = (\{q_1, q_2, [q_4, 1], \dots, [q_4, 5]\}, \Sigma, \delta, q_1, \{q_2, [q_4, 1], \dots, [q_4, 5]\})$$

где δ определяется соотношениями

- (1) $\delta(q_1, \alpha) = \{q_2\}$ для всех $\alpha \in \Sigma$, которые являются буквами,
- (2) $\delta(q_2, \alpha) = \{[q_4, 1]\}$ для всех $\alpha \in \Sigma$,
- (3) $\delta([q_4, i], \alpha) = \{[q_4, i+1]\}$ для всех $\alpha \in \Sigma$ и $1 \leq i < 5$.

Заметим, что состояние $[q_7, 1]$ недостижимо и его можно устраинуть из M . Кроме того, автомат M оказался здесь детерминированным, хотя в общем случае этого может и не быть.

Диаграмма автомата M показана на рис. 3.7. \square

3.3.3. Прямой лексический анализ

При прямом лексическом анализе требуется найти одну из многих лексем. Наиболее эффективный способ заключается, вообще говоря, в том, чтобы вести поиск параллельно, так как он при этом часто довольно быстро сужается. Таким образом,

моделью прямого лексического анализатора служит множество работающих параллельно конечных автоматов, или, точнее, один конечный преобразователь, моделирующий много конечных автоматов и выдающий сигнал о том, какой из них распознал цепочку.

Если нужно моделировать параллельно несколько недетерминированных автоматов, причем множества их состояний не пересекаются, то можно объединить эти множества состояний и функции переходов, построив один недетерминированный конечный автомат, который можно превратить в детерминированный по теореме 2.3. (Начальным состоянием детерминированного автомата будет при этом множество всех начальных состояний составляющих автоматов.) Таким образом, удобнее объединить множества состояний до перехода к детерминированному автомatu, а не после.

Объединенный детерминированный автомат можно рассматривать как конечный преобразователь простого типа. Он выдает имя лексемы и, возможно, информацию, локализующую входжение этой лексемы. В каждом состоянии объединенного автомата представлены состояния всех составляющих автоматов. Понятно, что когда объединенный автомат попадает в состояние, содержащее заключительное состояние одного составляющего автомата и не содержащее никаких других состояний, он должен остановиться и выдать имя лексемы, соответствующей этому составляющему автомату. Однако часто дело обстоит не так просто.

Например, если идентификатором может быть любая цепочка, за исключением ключевых слов, то на практике нецелесообразно определять идентификатор в точности как это регулярное множество, поскольку такое определение сложно и требует много состояний. Вместо этого пользуются простым определением идентификатора (одно из таких дано в примере 3.18) и предоставляют объединенному автомatu принять правильное решение.

В этом случае, если объединенный автомат попадает в состояние, которое содержит заключительное состояние одного из автоматов, соответствующих ключевым словам, и состояние автомата, соответствующего идентификаторам, а следующий входной символ (это может быть пробел или специальный знак) указывает на окончание лексемы, то предпочтение отдается ключевому слову и выдается указание о том, что обнаружено ключевое слово.

Пример 3.21. Рассмотрим несколько абстрактный пример. Допустим, что идентификаторы представляют собой цепочки, составленные из четырех символов D, F, I и O, за которыми должен следовать пробел (б), причем DO и IF—ключевые слова,

которые не являются идентификаторами и за которыми может не следовать пробел и не должны следовать (непосредственно) буквы D, F, I и O.

Упрощенное множество идентификаторов (включающее DO и IF) распознается конечным автоматом, изображенным на рис. 3.8, а, цепочка DO—автоматом на рис. 3.8, б, а IF—автоматом на рис. 3.8, в. (Здесь все автоматы детерминированные, хотя в общем случае это, разумеется, не обязательно.)

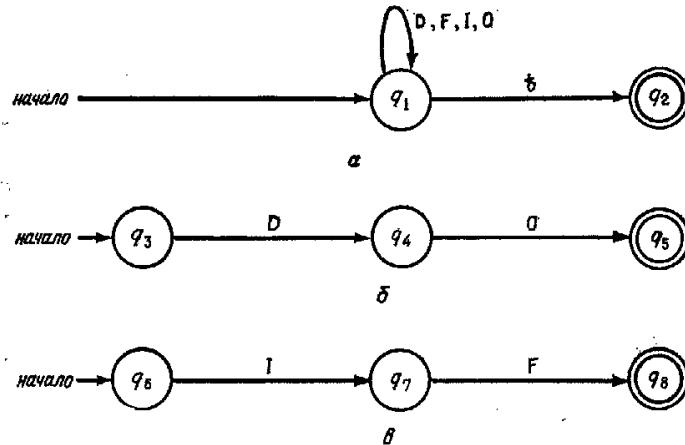


Рис. 3.8. Автоматы для лексического анализа: а—идентификаторы; б—DO; в—IF.

Объединенный автомат показан на рис. 3.9¹⁾. Состояние q_2 указывает, что обнаружен идентификатор. Однако состояния $\{q_1, q_8\}$ и $\{q_1, q_5\}$ нельзя истолковать однозначно. Они могут указывать на IF или DO соответственно, а могут только на то, что появился начальный отрезок какого-то идентификатора, вроде DOOF. Чтобы разрешить возникший конфликт, лексический анализатор должен взглянуть на следующий символ. Если это D, O, I или F, то перед нами префикс идентификатора. Если что-то другое, в том числе и пробел (допустим, что букв больше, чем упомянутых пять), то автомат переходит в новое состояние q_9 или q_{10} и выдает сигнал о том, что обнаружено ключевое слово DO или IF соответственно и оно окончилось на один символ раньше. Если автомат попадает в q_2 , то он выдает сигнал о том, что обнаружен идентификатор, оканчивающийся одним символом раньше.

¹⁾ В противоположность автомatu рис. 3.8 данный автомат не допускает в качестве идентификатора пробел.

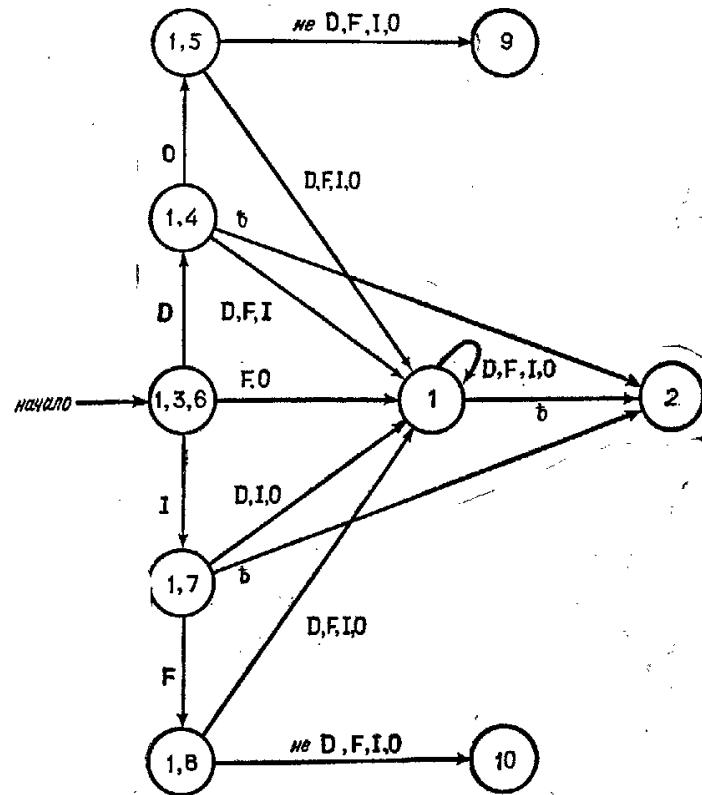


Рис. 3.9. Объединенный лексический анализатор.

Важно отметить, что так как указанное различие проявляется в выходе описанного устройства, а не в состоянии, то состояния q_2 , q_9 и q_{10} можно отождествить, и на самом деле они вообще не будут представлены в программной реализации. □

3.3.4. Программное моделирование конечных преобразователей

Известно несколько подходов к программному моделированию конечных автоматов и преобразователей. Медленный, но экономный с точки зрения расхода памяти способ заключается в том, что кодируется функция переходов соответствующего устройства и работа с ней реализуется в режиме интерпретации. Так как лексический анализ составляет значительную долю процесса трансляции, такой метод часто оказывается слишком медленным, чтобы его можно было принять. Однако некоторые

вычислительные машины имеют команды, распознающие те или иные виды лексем, и хотя этими командами нельзя промоделировать произвольный конечный автомат, они очень хорошо работают, когда лексемами служат ключевые слова или идентификаторы. Другой подход состоит в том, чтобы завести кусок программы для каждого состояния. В функции программы входит определение очередной буквы (для локализации буквы можно использовать подпрограмму), выдача требуемого выхода и переход к тому месту программы, которое соответствует следующему состоянию.

Важную проблему представляет выбор подходящего метода определения очередной буквы. Если для данного текущего состояния функция переходов такова, что большинство различных очередных букв приводят к различным следующим состояниям, то, по-видимому, нет ничего лучше, чем передавать управлениекосвенно через таблицу, основанную на очередной букве. Этот метод по скорости не уступает любому другому, но для него нужна таблица, объем которой пропорционален числу различных букв.

В типичном лексическом анализаторе много состояний, для которых почти все очередные буквы приводят к одному и тому же состоянию. Размещение в памяти полной таблицы для каждого такого состояния привело бы к слишком большому расходу памяти. Для многих состояний разумным компромиссом между соображениями, связанными с затратами времени и памяти, был бы двоичный поиск для выбора тех немногих букв, которые вызывают переход в необычное состояние.

УПРАЖНЕНИЯ

3.3.1. Напишите регулярные выражения, эквивалентные следующим расширенным регулярным выражениям:

- $(a^{+3}b^{+3})^{*2}$,
- $(a|b)^* - (ab)^*$,
- $(aa|bb)^{*4} \cap a(ab|ba)^{+}b$.

3.3.2. Напишите последовательность регулярных определений, приводящую к определению

- идентификаторов Алгола,
- идентификаторов ПЛ/1,
- комплексных констант вида (α, β) , где α и β — вещественные константы Фортрана,
- комментариев в ПЛ/1.

3.3.3. Докажите теорему 3.3.

3.3.4. Постройте непрямые лексические анализаторы для трех регулярных множеств из упр. 3.3.2.

3.3.5. Постройте прямой лексический анализатор, различающий следующие лексемы:

(1) идентификаторы, представляющие собой любые последовательности букв и цифр, содержащие хотя бы одну букву (исключения оговорены в пункте (3)),

(2) константы, как в примере 3.19,

(3) ключевые слова IF, IN и INTEGER (которые не считаются идентификаторами).

3.3.6. Расширьте понятие неразличимых состояний (разд. 2.3), чтобы применить его к недетерминированным конечным автоматам. Если склеить все неразличимые состояния, обязательно ли получится недетерминированный автомат с минимальным числом состояний?

****3.3.7.** Будет ли более легким прямой лексический анализ для Фортрана, если исходная программа считывается в обратном направлении?

Проблема для исследования

3.3.8. Дайте алгоритм выбора реализации для прямых лексических анализаторов. Ваш алгоритм должен допускать задание желательного соотношения между затратами времени и памяти. Возможно, Вы не захотите реализовать работу конечного автомата символ за символом, а предпочтете использовать и другие операции. Например, если бы многие лексемы были арифметическими знаками, которые должны отделяться пробелами, как в Сноболе, то было бы разумно в качестве первого шага лексического анализа отделить эти лексемы от других, проверяя, является ли вторая буква пробелом.

Упражнения на программирование

3.3.9. Постройте лексический анализатор для одного из языков программирования, данных в приложении. Приведите соображения относительно того, как этот анализатор будет обнаруживать лексические ошибки, в частности описки.

3.3.10. Придумайте язык программирования, основанный на расширенных регулярных выражениях. Постройте компилятор для этого языка. Программа в объектном языке должна быть реализацией лексического анализатора, описываемого исходной программой.

Замечания по литературе

Первой большой системой, в которой при построении лексических анализаторов использовалась техника конечных автоматов, была система AED WORD (Read a WORD). Обзор этой системы дан Джонсоном и др. [1968].

Томпсон [1968] приводит алгоритм, который по регулярному выражению строит программу в машинном языке, моделирующую соответствующий недетерминированный конечный автомат. Этот алгоритм применялся для сравнения образцов в мощном языке QED, предназначенном для редактирования текстов.

Лексический анализатор целесообразно проектировать так, чтобы он справлялся с лексическими ошибками. К ним относятся, в частности,

- (1) замена в лексеме правильного символа неправильным,
- (2) вставка в лексему лишнего символа,
- (3) пропуск символа в лексеме,
- (4) перестановка двух смежных символов лексемы.

Техника, позволяющая обнаруживать и исправлять ошибки такого рода, описана Фрименом [1964] и Морганом [1970]. Дополнительная литература по обнаружению и исправлению ошибок при компиляции указана в замечаниях по литературе в конце разд. 1.2.

3.4. СИНТАКСИЧЕСКИЙ АНАЛИЗ

Второй фазой процесса компиляции обычно является фаза синтаксического анализа, или разбора. В данном разделе мы дадим формальные определения двух распространенных типов разбора и кратко сопоставим их возможности. Займемся также вопросом о том, что значит: одна грамматика „покрывает“ другую.

3.4.1. Определение разбора

Мы говорим, что для некоторой КС-грамматики G цепочка $w \in L(G)$ разобрана, или проанализирована, если известно одно (или, быть может, все) из ее деревьев выводов. При трансляции такое дерево можно „физически“ построить в памяти машины, но вероятнее, что будет использован какой-нибудь более искусный способ представления. Проследив шаг за шагом работу синтаксического анализатора, можно получить дерево разбора, хотя связь между этими процессами едва ли сразу очевидна.

К счастью, большинство компиляторов осуществляют разбор моделированием МП-автомата, анализирующего входные цепочки либо сверху вниз, либо снизу вверх (см. разд. 2.5). Мы увидим, что способность МП-автомата проводить нисходящий анализ связана со способностью МП-преобразователя отображать входные цепочки в соответствующие левые выводы. Аналогично восходящий анализ связан с отображением входных цепочек в их обращенные правые выводы. Таким образом, мы будем трактовать проблему разбора как проблему отображения цепочек в их левые или правые выводы. Хотя известно много других стратегий разбора, мы будем опираться на эти две.

В разных частях книги упоминаются другие стратегии разбора. В упражнениях, помещенных в конце разд. 3.4, 4.1 и 5.1, рассматривается анализ по левому участку — метод разбора, который по своему характеру является и восходящим, и нисходящим. В разд. 6.2.1 мы обсудим методы обобщенного нисходящего и восходящего разбора.

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, правила которой занумерованы $1, 2, \dots, p$. Пусть $\alpha \in (N \cup \Sigma)^*$. Тогда

(1) *левым разбором* цепочки α называется последовательность правил, примененных при левом выводе цепочки α из S ,

(2) *правым разбором* цепочки α называется обращение последовательности правил, примененных при правом выводе цепочки α из S .

Эти разборы можно представить в виде последовательности номеров из множества $\{1, 2, \dots, p\}$.

Пример 3.22. Рассмотрим грамматику G_0 с такой нумерацией правил:

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

Левый разбор цепочки $a * (a + a)$ — это последовательность 23465124646, а правый разбор — 64642641532. \square

Для описания левых и правых разборов введем дополнительные обозначения.

Соглашение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, правила которой занумерованы числами от 1 до p . Будем писать $\alpha \Rightarrow_i \beta$, если $\alpha \Rightarrow_i \beta$ и применено i -е правило. Аналогично будем писать $\alpha \Rightarrow_{i,j} \beta$, если $\alpha \Rightarrow_i \beta$ и применено i -е правило. Расширим эти обозначения, положив

- (1) $\alpha \Rightarrow_{i_1, i_2} \gamma$, если $\alpha \Rightarrow_{i_1} \beta$ и $\beta \Rightarrow_{i_2} \gamma$,
- (2) $\alpha \Rightarrow_{i_1, i_2} \gamma$, если $\alpha \Rightarrow_{i_1} \beta$ и $\beta \Rightarrow_{i_2} \gamma$.

3.4.2. Нисходящий разбор

В этом разделе мы хотим исследовать проблему левого анализа для КС-грамматик. Пусть $\pi = i_1 \dots i_n$ — левый разбор цепочки $w \in L(G)$, где G — КС-грамматика. Зная π , можно построить

дерево разбора цепочки w следующим „иисходящим“ образом. Начнем с корня, помеченного S . Тогда i_1 дает правило, которое надо применить к S . Допустим, что i_1 — номер правила $S \rightarrow X_1 \dots X_k$. Присоединим k потомков к вершине, помеченной S , и назовем их X_1, X_2, \dots, X_k . Если X_i — первый слева нетерминал в цепочке $X_1 \dots X_k$, то первыми $i-1$ символами цепочки w должны быть $X_1 \dots X_{i-1}$. Правило с номером i_2 должно тогда иметь вид $X_i \rightarrow Y_1 \dots Y_l$, и можно продолжить построение дерева разбора цепочки w , применяя ту же процедуру к вершине, помеченной X_l . Продолжая в том же духе, можно построить все дерево разбора цепочки w , соответствующее левому разбору π .

Допустим теперь, что даны КС-грамматика $G = (N, \Sigma, P, S)$, в которой правила занумерованы от 1 до p , и цепочка $w \in \Sigma^*$, для которой мы хотим построить левый разбор. Можно считать, что известны корень и крона дерева разбора и нам остается „всего лишь“ восполнить промежуточные вершины. Стратегия левого нисходящего разбора предлагает пытаться заполнять дерево разбора, начиная с корня, и двигаться слева направо, направляясь к кроне.

Легко показать, что существует простая СУ-схема, отображающая цепочки языка $L(G)$ в их левые (или правые, если угодно) разборы. Мы определим здесь такую СУ-схему, хотя предпочитаем исследовать МП-преобразователь, реализующий соответствующий перевод, потому что от него легко перейти к реальному выполнению этого перевода.

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, в которой правила занумерованы от 1 до p . Определим СУ-схему T_l^G (или T_l , если G подразумевается) как пятерку $(N, \Sigma, \{1, \dots, p\}, R, S)$, где R состоит из таких правил $A \rightarrow \alpha, \beta$, что $A \rightarrow \alpha$ — правило из P с номером i , а $\beta = i\alpha'$, где α' — это цепочка α , из которой устраниены все терминалы.

Пример 3.23. Пусть G_0 — обычная наша грамматика с правилами, занумерованными, как в примере 3.22. Тогда $T_l = (\{E, T, F\}, \{+, *, (,), a\}, \{1, \dots, 6\}, R, E)$, где R состоит из правил

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

Пара деревьев, изображенная на рис. 3.10, задает перевод цепочки $a * (a + a)$. \square

Оставляем в качестве упражнения доказательство следующей теоремы:

Теорема 3.10. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Тогда $T_l = \{(w, \pi) | S \xrightarrow{\pi} w\}$.

Доказательство. Можно доказать индукцией, что $(A, A) \Rightarrow_l^* (w, \pi)$ тогда и только тогда, когда $A \xrightarrow{\pi} G w$. \square

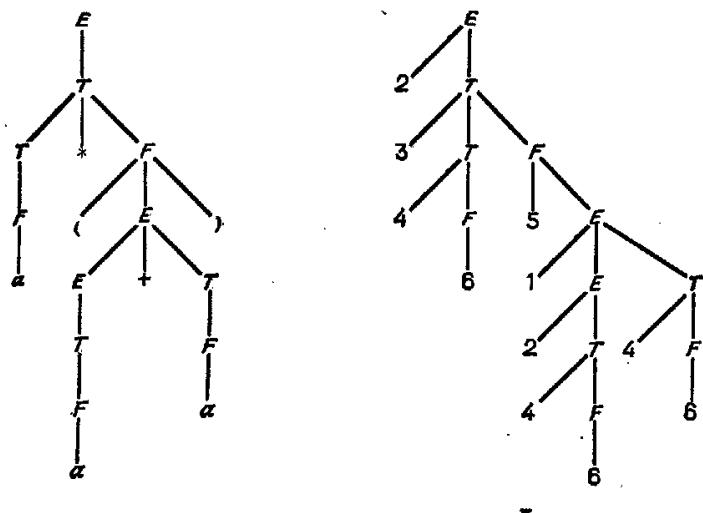


Рис. 3.10. Перевод схемой T_l : a — вход; b — выход.

С помощью конструкции, подобной конструкции леммы 3.2, можно для любой грамматики G построить недетерминированный МП-преобразователь, работающий как левый анализатор для G .

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, в которой правила занумерованы от 1 до p . Пусть M_l^G (или M_l , если G подразумевается) — недетерминированный МП-преобразователь $(\{q\}, \Sigma, N \cup \Sigma, \{1, 2, \dots, p\}, \delta, q, S, \emptyset)$, где δ определяется так:

- (1) $\delta(q, e, A)$ содержит (q, α, i) , если $A \rightarrow \alpha$ — правило из P с номером i ,
- (2) $\delta(q, a, A) = \{(q, e, e)\}$ для всех $\alpha \in \Sigma$.

Назовем M_l^G левым анализатором для G .

Для входа w анализатор M_t моделирует левый вывод в грамматике G цепочки w из S . По правилам (1) M_t , каждый раз „развертывает“ нетерминал, расположенный наверху магазина, в соответствии с некоторым правилом из P и одновременно выдает номер этого правила. Если наверху магазина находится терминальный символ, то M_t , по правилу (2) проверяет, совпадает ли он с текущим входным символом. Таким образом, M_t может производить только левый вывод цепочки w .

Теорема 3.11. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Тогда $\tau_e(M_t) = \{(w, \pi) \mid S \xrightarrow{\pi} w\}$.

Доказательство. Еще одно элементарное упражнение на индукцию. Здесь предположение индукции состоит в том, что $(q, w, A, e) \vdash^*(q, e, e, \pi)$ тогда и только тогда, когда $A \xrightarrow{\pi} w$. \square

Заметим, что M_t^G — это почти, но не совсем ДМП-преобразователь, который получается по лемме 3.2 из СУ-схемы T_t^G .

Пример 3.24. Построим левый анализатор для грамматики G_0 . Здесь

$$M_t = (\{q\}, \Sigma, N \cup \Sigma, \{1, 2, \dots, 6\}, \delta, q, E, \emptyset)$$

где

$$\delta(q, e, E) = \{(q, E + T, 1), (q, T, 2)\}$$

$$\delta(q, e, T) = \{(q, T * F, 3), (q, F, 4)\}$$

$$\delta(q, e, F) = \{(q, (E), 5), (q, a, 6)\}$$

$$\delta(q, b, b) = \{(q, e, e)\} \text{ для всех } b \in \Sigma$$

Для входной цепочки $a + a * a$ левый анализатор M_t^G может среди других сделать такую последовательность тактов:

$$\begin{aligned} (q, a + a * a, E, e) &\vdash (q, a + a * a, E + T, 1) \\ &\vdash (q, a + a * a, T + T, 12) \\ &\vdash (q, a + a * a, F + T, 124) \\ &\vdash (q, a + a * a, a + T, 1246) \\ &\vdash (q, + a * a, + T, 1246) \\ &\vdash (q, a * a, T, 1246) \\ &\vdash (q, a * a, T * F, 12463) \\ &\vdash (q, a * a, F * F, 124634) \\ &\vdash (q, a * a, a * F, 1246346) \\ &\vdash (q, * a, * F, 1246346) \\ &\vdash (q, a, F, 1246346) \\ &\vdash (q, a, a, 12463466) \\ &\vdash (q, e, e, 12463466) \quad \square \end{aligned}$$

Левый анализатор представляет собой, вообще говоря, недетерминированное устройство. Чтобы пользоваться им на практике, нужно уметь моделировать его детерминированно. Для некоторых грамматик (например, для тех, что содержат циклы) полное моделирование невозможно (в данном случае потому, что некоторые цепочки имеют бесконечно много левых разборов). Кроме того, естественное моделирование, которое мы рассмотрим в гл. 4, непригодно для более широкого класса грамматик, а именно для леворекурсивных грамматик. Нисходящий анализ налагает существенное ограничение: должна быть устраниена левая рекурсия.

Существует естественный класс грамматик — они называются LL-грамматиками (ход читается слева (left) направо и выдается левый (left) разбор) и рассматриваются в разд. 5.1, — для которых левый разбор можно сделать детерминированным с помощью простого приема, состоящего в том, что анализатор заглядывает на входе на несколько символов вперед и делает очередной шаг на основе того, что он при этом видит. LL-грамматики — это те, которые „естественным образом“ анализируются детерминированным левым анализатором.

Можно рассмотреть более широкий класс грамматик, для каждой из которых возможен ДМП-преобразователь, реализующий СУ-схему T_t . Он содержит все LL-грамматики и некоторые другие, которые можно анализировать только „неестественным“ способом, т. е. таким, когда содержимое магазина не отражает в отличие от анализатора M_t последовательности шагов левого вывода. С точки зрения нисходящего анализа такие грамматики представляют только теоретический интерес, но мы кратко рассмотрим их в разд. 3.4.4.

3.4.3. Восходящий разбор

Займемся теперь правым разбором. Возьмем правый вывод в грамматике G_0 цепочки $a + a * a$ из E :

$$\begin{aligned} E &\Rightarrow_1 E + T \\ &\Rightarrow_3 E + T * F \\ &\Rightarrow_6 E + T * a \\ &\Rightarrow_4 E + F * a \\ &\Rightarrow_8 E + a * a \\ &\Rightarrow_2 T + a * a \\ &\Rightarrow_4 F + a * a \\ &\Rightarrow_6 a + a * a \end{aligned}$$

Записывая в обратном порядке последовательность правил, участвующих в этом выводе, получаем правый разбор 64264631 цепочки $a + a * a$.

Вообще правый разбор цепочки ω в грамматике $G = (N, \Sigma, P, S)$ — это последовательность правил, с помощью которых можно свернуть цепочку ω к начальному символу S . В терминах деревьев выводов правый анализ цепочки ω представляет собой последовательность отсечений основ, сводящую дерево вывода с кроной ω к одной вершине, помеченной S . В сущности это равнозначно процессу „заполнения“ дерева вывода, начинающемуся с одной только кроны и идущему от листьев к корню. Поэтому с процессом порождения правого разбора часто ассоциируется термин „восходящий“ анализ.

По аналогии с СУ-схемой T_l , отображающей цепочки из $L(G)$ в их левые разборы, можно определить СУ-схему T_r , отображающую цепочки в правые разборы. В ней из элементов перевода устраниены терминалы, а номера правил выводов пишутся справа. Доказательство того, что эта СУ-схема корректно определяет нужный перевод, оставляем в качестве упражнения.

Что касается восходящего анализа, то нас прежде всего интересует МП-преобразователь, реализующий схему T_r . По аналогии с расширенным МП-автоматом определим расширенный МП-преобразователь.

Определение. Расширенным МП-преобразователем назовем восьмерку $P = (Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$, где все символы понимаются так же, как прежде, но только δ теперь обозначает отображение конечного подмножества множества $Q \times (\Sigma \cup \{e\}) \times \Gamma^*$ в множество конечных подмножеств множества $Q \times \Gamma^* \times \Delta^*$. Конфигурации определяются так же, как прежде, но обычно подразумевается, что верх магазина расположен справа, и запись $(q, a\omega, \beta\alpha, x) \vdash (p, \omega, \beta\gamma, xy)$ означает, что $\delta(q, a, \alpha)$ содержит (p, γ, y) .

Расширенный МП-преобразователь P называется *детерминированным*, если

- (1) $\#\delta(q, a, \alpha) \leq 1$ для всех $q \in Q$, $a \in \Sigma \cup \{e\}$ и $\alpha \in \Gamma^*$,
- (2) цепочки α и β не являются суффиксами одна другой, если $\delta(q, a, \alpha) \neq \emptyset$ и $\delta(q, b, \beta) \neq \emptyset$ для $b = a$ или $b = e$.

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Обозначим через M_r^G расширенный недетерминированный МП-преобразователь $(\{q\}, \Sigma, N \cup \Sigma \cup \{\$\}, \{1, \dots, p\}, \delta, q, \$, \emptyset)$, причем верх магазина расположен справа и δ определяется так:

- (1) $\delta(q, e, \alpha)$ содержит (q, A, i) , если $A \rightarrow \alpha$ — правило из P с номером i ,
- (2) $\delta(q, a, e) = \{(q, a, e)\}$ для всех $a \in \Sigma$,
- (3) $\delta(q, e, \$S) = \{(q, e, e)\}$.

Этот МП-преобразователь содержит элементы алгоритма разбора, называемого алгоритмом типа перенос — свертка. На такте, соответствующем правилу (2), M_r переносит входной символ в магазин. Всякий раз, когда наверху магазина появляется основа, M_r может свернуть ее по правилу (1) и выдать номер правила, использованного при свертке. Затем M_r может продолжать переносить в магазин входные символы, пока в его верхней части не появится новая основа. Эта основа свертывается, а на выход выдается номер соответствующего правила. M_r действует так до тех пор, пока в магазине не останется только начальный символ с маркером дна магазина. По правилу (3) M_r может перейти тогда в конфигурацию с пустым магазином.

Теорема 3.12. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Тогда $\tau_e(M_r) = \{(\omega, \pi^R) \mid S \Rightarrow_{\pi} \omega\}$.

Доказательство. Доказательство аналогично доказательству леммы 2.25; оставляем его в качестве упражнения. \square

Пример 3.25. Правым анализатором для грамматики G_0 будет

$$M_r^{G_0} = (\{q\}, \Sigma, N \cup \Sigma \cup \{\$\}, \{1, 2, \dots, 6\}, \delta, q, \$, \emptyset)$$

где

$$\begin{aligned}\delta(q, e, E + T) &= \{(q, E, 1)\} \\ \delta(q, e, T) &= \{(q, E, 2)\} \\ \delta(q, e, T * F) &= \{(q, T, 3)\} \\ \delta(q, e, F) &= \{(q, T, 4)\} \\ \delta(q, e, (E)) &= \{(q, F, 5)\} \\ \delta(q, e, a) &= \{(q, F, 6)\} \\ \delta(q, b, e) &= \{(q, b, e)\} \text{ для всех } b \in \Sigma \\ \delta(q, e, \$E) &= \{(q, e, e)\}\end{aligned}$$

Для входной цепочки $a + a * a$ анализатор $M_r^{G_0}$ может среди других сделать такую последовательность тактов:

$$\begin{aligned}(q, a + a * a, \$, e) \vdash (q, + a * a, \$a, e) \\ \vdash (q, + a * a, \$F, 6) \\ \vdash (q, + a * a, \$T, 64) \\ \vdash (q, + a * a, \$E, 642) \\ \vdash (q, a * a, \$E +, 642) \\ \vdash (q, * a, \$E + a, 642) \\ \vdash (q, * a, \$E + F, 6426) \\ \vdash (q, * a, \$E + T, 64264)\end{aligned}$$

- $\vdash (q, a, \$E + T *, 64264)$
- $\vdash (q, e, \$E + T * a, 64264)$
- $\vdash (q, e, \$E + T * F, 642646)$
- $\vdash (q, e, \$E + T, 6426463)$
- $\vdash (q, e, \$E, 64264631)$
- $\vdash (q, e, e, 64264631)$

Таким образом, для входной цепочки $a+a*a$ анализатор M_f^G выдает правый разбор 64264631. \square

В гл. 4 мы рассмотрим детерминированное моделирование недетерминированного правого анализатора. В разд. 5.2 обсудим важный подкласс КС-грамматик, называемых LR-грамматиками (вход читается слева (*left*) направо и выдается правый (*right*) разбор), для которых соответствующий ДМП-преобразователь можно сделать детерминированным, позволив ему заглядывать на входе на несколько символов вперед. LR-грамматики — это те, которые анализируются снизу вверх (по дереву вывода от кроны к корню) естественным образом и притом детерминированно. Как и в случае левого разбора, существуют грамматики, для которых возможен детерминированный, но не естественный правый анализ; их мы обсудим в следующем разделе.

3.4.4. Сравнение нисходящего разбора с восходящим

Если рассматривать недетерминированные анализаторы, то сравнивать по существу нечего, так как по теоремам 3.11 и 3.12 для каждой КС-грамматики возможен как левый, так и правый анализ. Однако, если поставить важный вопрос о том, существуют ли для данной грамматики детерминированные анализаторы, то тут дело обстоит не так просто.

Определение. Назовем КС-грамматику G левоанализируемой, если существует такой ДМП-преобразователь P , что

$$\tau(P) = \{(x\$, \pi) \mid (x, \pi) \in T_f^G\}$$

и правоанализируемой, если существует такой ДМП-преобразователь P , что

$$\tau(P) = \{(x\$, \pi) \mid (x, \pi) \in T_r^G\}$$

В обоих случаях ДМП-преобразователю разрешается использовать концевой маркер для указания правого конца входной цепочки.

Заметим, что все КС-грамматики лево- и правоанализируемы в неформальном смысле, но в данном формальном определении отражен именно детерминизм.

Мы обнаружим, что классы лево- и правоанализируемых грамматик несравнимы, т. е. ни один из них не является подмножеством другого. Это удивительно с точки зрения разд. 8.1, где будет показано, что LL-грамматики, которые можно детерминированно левоанализировать естественным образом, образуют подмножество LR-грамматик, которые можно детерминированно правоанализировать естественным образом. Приведем примеры грамматик, лево- (право-) анализируемых, но не право- (лево-) анализируемых.

Пример 3.26. Пусть грамматика G_1 определена правилами

- | | |
|--------------------------|--------------------------|
| (1) $S \rightarrow BA b$ | (2) $S \rightarrow CA c$ |
| (3) $A \rightarrow BA$ | (4) $A \rightarrow a$ |
| (5) $B \rightarrow a$ | (6) $C \rightarrow a$ |

$L(G_1) = aa^+b + aa^+c$. Можно показать, что G_1 не является ни LL-, ни LR-грамматикой, потому что до тех пор, пока мы не увидим последний символ цепочки, не известно, каким нетерминалом B или C порождается первый символ a .

Однако можно «неестественно» получить левый разбор произвольной входной цепочки с помощью ДМП-преобразователя. Рассмотрим в качестве входной цепочки $a^{n+2}b$ ($n \geq 0$). Тогда ДМП-преобразователь может сделать левый разбор 15(35) n 4, помещая в магазин все символы a до тех пор, пока он не увидит b . При этом на выходе ничего не порождается. Когда b появится, ДМП-преобразователь может выдать 15(35) n 4, подсчитав n с помощью занесенных в магазин символов a . Подобным же образом для входа $a^{n+2}c$ можно получить 26(35) n 4 на выходе. В любом случае вся хитрость в том, чтобы отложить порождение выхода до того, как появится b или c .

Теперь попытаемся убедить читателя, что не существует ДМП-преобразователя, способного порождать правильные правые разборы для всех входных цепочек. Допустим, что ДМП-преобразователь M порождает правые разборы 55 n 43 n 1 для цепочки $a^{n+2}b$ и 65 n 43 n 2 для цепочки $a^{n+2}c$. Докажем неформально, что такой преобразователь невозможен. Это доказательство существенно опирается на результаты работы Гинзбурга и Грейбах [1966], где показано, что $\{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\}$ не является детерминированным КС-языком. Знакомство с этой работой поможет при построении формального доказательства. Можно доказать следующие утверждения:

(1) Пусть a^i поступает на вход преобразователя M . Тогда либо его выход пуст, либо он выдает 5 или 6, и преобразователь M можно «обмануть», подав на вход c или b соответственно и сделав тем самым его выход ошибочным.

(2) По мере того как символы a поступают на вход M , они должны так или иначе запасаться в магазине. Точнее, можно показать, что найдутся такие числа j и k , магазинные цепочки α и β и состояние q , что $(q_0, a^{k+jp}, Z_0, e) \xrightarrow{*} (q, e, \beta^p\alpha, e^1)$ для всех $p \geq 0$, где q_0 и Z_0 — начальные состояние и магазинный символ преобразователя M .

(3) Если после $k+jp$ символов a на входе M появляется один символ b , то M не может выдать 4 до тех пор, пока не сотрет магазинную ленту вплоть до α . В противном случае мы могли бы «обмануть» M , предварительно поместив на вход дополнительные j символов a , в то время как M будет выдавать то же количество чисел 5, которое он выдавал раньше, так что оба выхода правильными быть не могут.

(4) После того как магазин сократится до цепочки α , преобразователь M уже не может «помнить», сколько символов было на входе, потому что теперь конфигурации преобразователя M для разных значений p (где $k+jp$ — число символов a) отличаются только их состояниями. Таким образом, M не знает, сколько теперь надо выдать чисел 3. \square

Пример 3.27. Пусть G_2 определяется правилами

- | | |
|------------------------|------------------------|
| (1) $S \rightarrow Ab$ | (2) $S \rightarrow Ac$ |
| (3) $A \rightarrow AB$ | (4) $A \rightarrow a$ |
| (5) $B \rightarrow a$ | |

$L(G_2) = a^+b + a^+c$. Легко показать, что G_2 — правоанализируемая грамматика. С помощью рассуждений, аналогичных проведенным в упр. 3.26, можно показать, что G_2 не левоанализируема. \square

Теорема 3.13. Классы лево- и правоанализируемых грамматик несравнимы.

Доказательство. Примеры 3.26 и 3.27. \square

Несмотря на эту теорему, восходящий синтаксический анализ, как правило, привлекательнее нисходящего, так как для данного языка программирования часто легче написать правоанализируемую грамматику, чем левоанализируемую. К тому же, как отмечалось, LL-грамматики содержатся в классе LR-грамматик. В следующей главе мы увидим также, что естественное моделирование недетерминированного МП-преобразователя в случае, когда он является правым анализатором, работает для более общего класса грамматик (в некотором смысле, который там будет разъяснен), чем в случае левого анализатора.

¹⁾ Здесь подразумевается, что верх магазина расположен слева. — *Прим. перев.*

Однако с точки зрения процесса трансляции левый анализ предпочтительнее. Мы покажем, что каждый простой СУ-перевод можно выполнить с помощью

(1) МП-преобразователя, дающего левые разборы входных цепочек, за которым следует

(2) ДМП-преобразование, отображающее левые разборы в выходные цепочки данного СУ-перевода.

Интересно, что существуют простые СУ-переводы, для которых в приведенном выше утверждении нельзя заменить «левый разбор» на «правый разбор».

Если компилятор транслирует так, что вначале строится полный разбор, который потом превращается в объектный код, то указанное обстоятельство достаточно для доказательства того, что существуют переводы, требующие на промежуточной стадии именно левого разбора.

Однако многие компиляторы строят дерево разбора вершина за вершиной и вычисляют перевод в каждой вершине, когда она построена. Мы утверждаем, что если перевод нельзя вычислить прямо по дереву разбора, то его нельзя вычислять вершина за вершиной, когда само построение вершин идет снизу вверх. Это обсуждается подробно в гл. 9, и мы просим читателя подождать до этой главы с формализацией материала, связанного с переводом «вершина за вершиной».

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Определим язык L_l^G левых и язык L_r^G правых разборов грамматики G :

$$\begin{aligned} L_l^G &= \{\pi \mid S \xrightarrow{\pi} w \text{ для некоторой } w \in L(G)\} \\ L_r^G &= \{\pi^R \mid S \xrightarrow{\pi} w \text{ для некоторой } w \in L(G)\} \end{aligned}$$

Обозначения $\xrightarrow{\pi}$ и $\xrightarrow{\pi^R}$ можно распространить на СУ-схемы, договорившись, что $(\alpha, \beta) \xrightarrow{\pi} (\gamma, \delta)$ тогда и только тогда, когда $(\alpha, \beta) \xrightarrow{*} (\gamma, \delta)$, причем на каждом шаге этого вывода заменяется самый левый нетерминал первой компоненты выводимой пары и примененные при этом правила, из которых удалены элементы перевода, образуют последовательность π . Аналогично для СУ-схемы определяется $\xrightarrow{\pi^R}$.

Определение. СУ-схема называется семантически однозначной, если в ней нет двух различных правил вида $A \rightarrow \alpha, \beta$ и $A \rightarrow \alpha, \gamma$.

В семантически однозначной СУ-схеме для каждого правила входной грамматики есть точно один элемент перевода.

Теорема 3.14. Пусть $T = (N, \Sigma, \Delta, R, S)$ — семантически однозначная простая СУ-схема. Тогда существует такой ДМП-преобразователь P , что $\tau_e(P) = \{(\pi, y) \mid (S, S) \xrightarrow{\pi} (x, y) \text{ для некоторой цепочки } x \in \Sigma^*\}$.

Доказательство. Пусть $P = (\{q\}, \{1, \dots, p\}, N \cup \Delta, \Delta, \delta, q, S, \emptyset)$, где $1, \dots, p$ — номера правил входной грамматики, а δ определяется так:

(1) Пусть $A \rightarrow \alpha$ — правило входной грамматики с номером i и $A \rightarrow \alpha, \beta$ — единственное соответствующее правило схемы. Тогда $\delta(q, i, A) = (q, \beta, e)$.

(2) $\delta(q, e, b) = (q, e, b)$ для всех $b \in \Delta$.

МП-преобразователь P детерминированный, потому что правило (1) применяется только тогда, когда наверху магазина находится нетерминал, а правило (2) применяется только тогда, когда наверху выходной символ. Корректность преобразователя P следует из утверждения, которое легко доказывается индукцией: $(q, \pi, A, e) \vdash^* (q, e, e, y)$ тогда и только тогда, когда существует такая цепочка $x \in \Sigma^*$, что $(A, A)_\pi \Rightarrow (x, y)$. Доказательство оставляем в качестве упражнения. \square

Чтобы описать СУ-перевод, не выполнимый никаким ДМП-преобразователем, отображающим L_r^G , где G — входная грамматика, в соответствующие выходы этого перевода, нам понадобится следующая лемма.

Лемма 3.15. Не существует такого ДМП-преобразователя P , что $\tau(P) = \{(wc, w^Rcw) \mid w \in \{a, b\}^*\}$.

Доказательство. Здесь символ c играет роль правого концевого маркера. Допустим, что P для входа w выдает некоторую непустую цепочку, скажем dx , где $d \in \{a, b\}$. Пусть \bar{d} означает символ, противоположный d . Посмотрим, как работает P с цепочкой $w\bar{d}c$ в качестве входа. Он должен выдать некоторую цепочку, но эта цепочка должна начинаться символом d . Следовательно, P не отображает $w\bar{d}c$ в $\bar{d}w^Rcw\bar{d}$, как требуется. Таким образом, P не может ничего выдать на выходе, пока не будет достигнут правый концевой маркер c . В этот момент P будет находиться в состоянии q_w и хранить в магазине цепочку α_w .

Неформально α_w должна быть по существу цепочкой w , и тогда, стирая α_w , P может выдать w^R . Но стоит ему стереть α_w , как он уже не может «вспомнить» всю цепочку w , чтобы напечатать ее на выходе. Формальное доказательство леммы аналогично рассуждениям в примере 3.26. Здесь мы дадим набросок доказательства.

Рассмотрим входные цепочки вида $w = a^l$. Существуют такие числа j и k , состояние q и цепочки α и β , что, прочитав на входе $a^{j+k}c$, P помещает в магазин $\alpha\beta^n$ и переходит в состояние q . Тогда P должен стереть содержимое магазина вплоть до α к тому моменту, когда он выдает w^Rc . Но так как α от w не зависит, то выдать w уже невозможно. \square

Теорема 3.15. Существует простая СУ-схема $T = (N, \Sigma, \Delta, R, S)$, для которой нет такого ДМП-преобразователя P , что $\tau(P) = \{(\pi^R, x) \mid (S, S) \Rightarrow_\pi (w, x)$ для некоторой цепочки $w\}$.

Доказательство. Пусть T определяется правилами

- (1) $S \rightarrow Sa, aSa$
- (2) $S \rightarrow Sb, bSb$
- (3) $S \rightarrow c, c$

Тогда $L_r^G = 3(1+2)^*$, где G — входная грамматика. Если положить $h(1) = a$ и $h(2) = b$, то требуемым переводом $\tau(P)$ будет $\{(3a, h(\alpha)^Rch(\alpha)) \mid \alpha \in \{1, 2\}^*\}$. Если бы ДМП-преобразователь P , о котором говорится в теореме, существовал (с правым концевым маркером или без него), то можно было бы построить ДМП-преобразователь, определяющий перевод $\{(wc, w^Rcw) \mid w \in \{a, b\}^*\}$, а это противоречит лемме 3.15. \square

Итак, мы заключаем, что интересны как левый, так и правый ариал. Оба они будут изучаться в последующих главах. Синтаксический анализ другого типа, обладающий чертами и нисходящего, и восходящего анализа, — ариал по левому участку — рассматривается в упражнениях.

3.4.5. ГРАММАТИЧЕСКОЕ ПОКРЫТИЕ

Пусть G_1 — КС-грамматика. Можно считать, что грамматика G_2 подобна G_1 с точки зрения процесса разбора, если $L(G_2) = L(G_1)$ и левый и/или правый разбор цепочки, порождаемой грамматикой G_1 , можно выразить в терминах ее разбора в грамматике G_2 . В таком случае говорят, что G_2 покрывает G_1 . Покрывающие грамматики можно использовать несколькими способами. Например, если язык программирования описан «трудно анализируемой грамматикой», то было бы желательно найти покрывающую грамматику, анализирующую «легче». К тому же несколько изучаемых далее алгоритмов разбора работают только тогда, когда грамматика находится в некоторой нормальной форме, например в нормальной форме Хомского или в такой, где нет левой рекурсии. Если G_1 — произвольная грамматика, а G_2 — ее какая-то нормальная форма, то желательно, чтобы разборы в грамматике G_1 можно было легко восстановить по разборам в G_2 . В этом случае нет необходимости уметь восстанавливать разборы в G_2 по разборам в G_1 .

Для формального определения того, что подразумевается под «восстановлением» разборов в одной грамматике по разборам в другой, воспользуемся понятием гомоморфизма между разборами, представляющими собой цепочки определенного вида. Можно

привлечь и другие, более сильные, отображения; некоторые из них рассматриваются в упражнениях.

Определение. Пусть $G_1 = (N_1, \Sigma, P_1, S_1)$ и $G_2 = (N_2, \Sigma, P_2, S_2)$ — КС-грамматики и $L(G_1) = L(G_2)$. Будем говорить, что G_2 левопокрывает G_1 , если найдется такой гомоморфизм h , отображающий P_2 в P_1 , что

- (1) если $S_2 \Rightarrow w$, то $S_1 h(w) \Rightarrow w$,
- (2) для каждого π , для которого $S_2 \Rightarrow^\pi w$, существует такой разбор π' , что $S_2 \Rightarrow^{\pi'} w$ и $h(\pi') = \pi$.

Будем говорить, что G_2 правопокрывает G_1 , если найдется такой гомоморфизм h , отображающий P_2 в P_1 , что

- (1) если $S_2 \Rightarrow_\pi w$, то $S_1 \Rightarrow_{h(\pi)} w$,
- (2) для каждого π , для которого $S_2 \Rightarrow_\pi w$, существует такой разбор π' , что $S_2 \Rightarrow_{\pi'} w$ и $h(\pi') = \pi$.

Пример 3.28. Пусть G_1 — грамматика с правилами

- (1) $S \rightarrow 0S1$
- (2) $S \rightarrow 01$

а G_2 — эквивалентная ей грамматика в нормальной форме Хомского с правилами

- (1) $S \rightarrow AB$
- (2) $S \rightarrow AC$
- (3) $B \rightarrow SC$
- (4) $A \rightarrow 0$
- (5) $C \rightarrow 1$

Мы видим, что G_2 левопокрывает G_1 ; соответствующий гомоморфизм определяется равенствами $h(1) = 1$, $h(2) = 2$, $h(3) = h(4) = h(5) = e$. Например,

$$S_{1432455} \Rightarrow_{G_2} 0011, \quad h(1432455) = 12 \quad \text{и} \quad S_{12} \Rightarrow_{G_1} 0011$$

G_2 также и правопокрывает G_1 , причем подходит тот же гомоморфизм h . Например,

$$S \Rightarrow_{1352544} G_2 \Rightarrow 0011, \quad h(1352544) = 12 \quad \text{и} \quad S \Rightarrow_{12} G_1 \Rightarrow 0011$$

Грамматика G_1 не лево- и не правопокрывает G_2 . Так как обе грамматики однозначные, то соответствие между разборами фиксировано. Поэтому гомоморфизм g , доказывающий, что G_1 левопокрывает G_2 , должен отображать $1^n 2$ в $(143)^n 24(5)^{n+1}$, а это, как легко показать, невозможно. \square

Можно показать, что многие из конструкций разд. 2.4, представляющих грамматики к нормальным формам, дают грамматики, лево- или правопокрывающие исходные грамматики.

Пример 3.29. Ключевой шаг при построении нормальной формы Хомского (алгоритм 2.12) состоит в замене правила $A \rightarrow X_1 \dots X_n$ ($n > 2$) правилами $A \rightarrow X_1 B_1$, $B_1 \rightarrow X_2 B_2$, \dots , $B_{n-2} \rightarrow X_{n-1} X_n$. Можно показать, что в результате получается грамматика, левопокрывающая исходную: достаточно отобразить правило $A \rightarrow X_1 B_1$ в $A \rightarrow X_1 \dots X_n$, а каждое из правил $B_1 \rightarrow X_2 B_2$, \dots , $B_{n-2} \rightarrow X_{n-1} X_n$ в пустую цепочку. Если мы хотим получить правое покрытие, то $A \rightarrow X_1 \dots X_n$ можно заменить на $A \rightarrow B_1 X_n$, $B_1 \rightarrow B_2 X_{n-1}$, \dots , $B_{n-2} \rightarrow X_1 X_2$. \square

Другие результаты о покрытии оставляем в качестве упражнений.

УПРАЖНЕНИЯ

3.4.1. Дайте алгоритм построения дерева вывода по левому или правому разбору.

3.4.2. Пусть G — КС-грамматика. Покажите, что L_l^G — детерминированный КС-язык.

3.4.3. Всегда ли L_r^G — детерминированный КС-язык?

***3.4.4.** Постройте детерминированный МП-преобразователь P , для которого $\tau(P) = \{(\pi, \pi') \mid \pi \in L_l^G \text{ и } \pi' — \text{правый разбор для того же дерева вывода}\}$.

***3.4.5.** Можете ли Вы построить такой детерминированный МП-преобразователь P , что $\tau(P) = \{(\pi, \pi') \mid \pi \in L_r^G \text{ и } \pi' — \text{соответствующий левый разбор}\}$.

3.4.6. Постройте левые и правые разборы в грамматике G_0 для цепочек

- (a) $((a))$,
- (b) $a + (a + a)$,
- (b) $a * a * a$.

3.4.7. Пусть КС-грамматика G определяется следующими заимствованными правилами:

- (1) $S \rightarrow \text{if } B \text{ then } S \text{ else } S$
- (2) $S \rightarrow s$
- (3) $B \rightarrow B \wedge B$
- (4) $B \rightarrow B \vee B$
- (5) $B \rightarrow b$

Постройте СУ-схемы, определяющие T_l^G и T_r^G .

3.4.8. Постройте МП-преобразователи, определяющие T_l^G и T_r^G для грамматики G из упр. 3.4.7.

3.4.9. Докажите теорему 3.10.

3.4.10. Докажите теорему 3.11.

3.4.11. Дайте определение СУ-схемы T^G и докажите, что Ваша схема отображает цепочки из $L(G)$ в их правые разборы.

3.4.12. Дайте алгоритм, превращающий расширенный МП-преобразователь в эквивалентный МП-преобразователь. Ваш алгоритм должен быть таким, чтобы при применении его к детерминированному расширенному МП-преобразователю получался ДМП-преобразователь. Докажите, что Ваш алгоритм делает это.

3.4.13. Докажите теорему 3.12.

*3.4.14. Постройте детерминированные правые анализаторы для грамматик

- (a) (1) $S \rightarrow S0$
 (2) $S \rightarrow S1$
 (3) $S \rightarrow e$

(b) (1) $S \rightarrow AB$
 (2) $A \rightarrow 0A1$
 (3) $A \rightarrow e$
 (4) $B \rightarrow B1$
 (5) $B \rightarrow e$

*3.4.15. Постройте детерминированные левые анализаторы для грамматик

- (a) (1) $S \rightarrow 0S$
 (2) $S \rightarrow 1S$
 (3) $S \rightarrow e$

(b) (1) $S \rightarrow 0SI$
 (2) $S \rightarrow A$
 (3) $A \rightarrow A1$
 (4) $A \rightarrow e$

*3.4.16. Какие из грамматик в упр. 3.4.14 имеют детерминированные левые анализаторы? Какие в упр. 3.4.15 имеют детерминированные правые анализаторы?

*3.4.17. Дайте детальное доказательство того, что грамматика из примера 3.26 (3.27) право- (лево-)анализируема, но не лево- (право-)анализируема.

3.4.18. Дополните доказательство теоремы 3.14.

3.4.19. Дополните доказательство леммы 3.15.

3.4.20. Дополните доказательство теоремы 3.15.

Определение. Левым участком правила с непустой правой частью назовем самый левый символ (терминал или нетерминал) его правой части. Анализом (или разбором) цепочки по левому участку называется последовательность правил, соответствующих внутренним вершинам дерева разбора этой цепочки, в котором все вершины упорядочены следующим образом. Если вершина n имеет p прямых потомков n_1, n_2, \dots, n_p , то все вершины поддерева с корнем n_1 предшествуют n . Вершина n предшествует всем другим ее потомкам. Потомки вершины n_2 предшествуют потомкам вершины n_3 , которые в свою очередь предшествуют потомкам вершины n_4 , и т. д.

Грубо говоря, при анализе по левому участку левый участок правила распознается снизу вверх, а остальная часть правила — сверху вниз.

Пример 3.30. На рис. 3.11 показано дерево разбора цепочки $bbaaab$, порожденной грамматикой с правилами

- | | |
|-------------------------|------------------------|
| (1) $S \rightarrow AS$ | (2) $S \rightarrow BB$ |
| (3) $A \rightarrow bAA$ | (4) $A \rightarrow a$ |
| (5) $B \rightarrow b$ | (6) $B \rightarrow e$ |

Упорядочение вершин, о котором идет речь в определении анализа по левому участку, таково, что вершина n , и ее по-

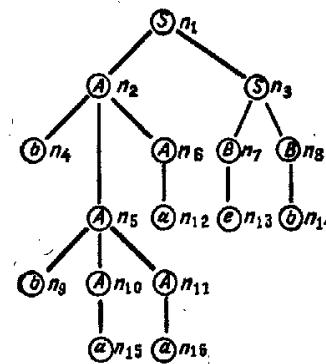


Рис. 3.11. Дерево разбора.

томки предшествуют вершине n_1 , за которой следуют n_3 и ее потомки. Вершина n_4 предшествует вершине n_2 , за которой следуют n_5 , n_6 и их потомки. Вершина n_9 предшествует вершине n_5 , за которой следуют n_{10} , n_{11} и их потомки. Продолжая в том же духе, получаем упорядочение вершин

$$n_4 n_2 n_9 n_5 n_{15} n_{10} n_{16} n_{11} n_{12} n_6 n_1 n_{18} n_7 n_3 n_{14} n_8$$

Разбором по левому участку является последовательность правил, соответствующих внутренним вершинам в этом упорядочении. Таким образом, разбором цепочки $bbaaab$ по левому участку будет последовательность 334441625. \square

Другой метод определения разбора по левому участку для цепочки, порождаемой грамматикой G , заключается в использовании следующей простой СУ-схемы, ассоциируемой с G .

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, в которой правила занумерованы от 1 до p . Пусть T_L^G — простая СУ-схема $(\bar{N}, \Sigma, \{1, \dots, p\}, R, S)$, где для каждого правила из P в множество R включается правило, определяемое так: если i -е правило из P имеет вид $A \rightarrow B\alpha$ или $A \rightarrow a\alpha$, или $A \rightarrow e$, то R содержит правило вида $A \rightarrow B\alpha$, $B\alpha'$ или $A \rightarrow a\alpha$, $a\alpha'$, или $A \rightarrow e$, i соответственно, где α' получается из α удалением всех терминальных символов. Тогда если $(w, \pi) \in \tau(T_L^G)$, то π — разбор по левому участку цепочки w .

Пример 3.31. Для грамматики из предыдущего примера схема T_L^G такова:

$$\begin{array}{ll} S \rightarrow AS, A1S & S \rightarrow BB, B2B \\ A \rightarrow bAA, 3AA & A \rightarrow a, 4 \\ B \rightarrow b, 5 & B \rightarrow e, 6 \end{array}$$

Можно убедиться, что $(bbaaab, 334441625) \in \tau(T_L^G)$. \square

3.4.21. Докажите, что $(w, \pi) \in \tau(T_L^G)$ тогда и только тогда, когда π — разбор по левому участку цепочки w .

3.4.22. Покажите, что для каждой КС-грамматики существует (недетерминированный) МП-преобразователь, отображающий цепочки языка, порождаемого этой грамматикой, в их разборы по левому участку.

3.4.23. Разработайте алгоритмы, отображающие разборы по левому участку в (1) соответствующие левые разборы, (2) соответствующие правые разборы, и обратно.

3.4.24. Покажите, что если G_3 лево- (право-)покрывает G_2 и G_2 лево- (право-)покрывает G_1 , то G_3 лево- (право-)покрывает G_1 .

3.4.25. Пусть G_1 — грамматика без циклов. Покажите, что G_1 лево- и правопокрывается грамматиками, не содержащими цепных правил.

3.4.26. Покажите, что каждая грамматика без циклов лево- и правопокрывается грамматиками в нормальной форме Хомского.

***3.4.27.** Покажите, что не каждая КС-грамматика покрывается грамматикой, не содержащей e -правил.

3.4.28. Покажите, что алгоритм 2.9, устраниющий бесполезные символы, дает грамматику, лево- и правопокрывающую исходную.

****3.4.29.** Покажите, что не каждая приведенная грамматика лево- или правопокрывается грамматикой в нормальной форме Грейбах. *Указание:* Рассмотрите грамматику $S \rightarrow S0|S1|0|1$.

****3.4.30.** Покажите, что утверждение из упр. 3.4.29 остается верным, если в определении покрытия заменить гомоморфизм конечным преобразованием.

***3.4.31.** Останется ли верным утверждение из упр. 3.4.29, если заменить гомоморфизм МП-преобразованием.

Проблема для исследования

3.4.32. Было бы хорошо, если бы всегда, когда G_2 лево- или правопокрывает G_1 , каждая СУ-схема с G_1 в качестве входной грамматики была эквивалентна некоторой СУ-схеме, входной грамматикой которой служит G_2 . К сожалению, это не так. Можете ли Вы найти условия, связывающие G_1 и G_2 , при которых СУ-схемы с входной грамматикой G_1 образуют подмножество СУ-схем с входной грамматикой G_2 ?

Замечания по литературе

Дополнительные подробности, касающиеся грамматического покрытия, можно найти в работах Рейнольдса и Хаскела [1970], Грэя [1969] и Грэя и Харрисона [1969]. В некоторых ранних статьях анализ по левому участку назывался восходящим анализом. Более полное изложение метода анализа по левому участку содержится в монографии Читэма [1967].

4

ОБЩИЕ МЕТОДЫ СИНТАКСИЧЕСКОГО АНАЛИЗА

Эта глава посвящена алгоритмам синтаксического анализа, применимыми ко всему классу контекстно-свободных языков. Не все эти алгоритмы можно применять к любым КС-грамматикам, но каждый КС-язык имеет хотя бы одну грамматику, к которой все они применимы.

Вначале мы обсудим алгоритмы с полным возвратом. Эти алгоритмы детерминированно моделируют недетерминированные анализаторы. Емкость памяти, которую требуют эти возвратные методы, линейно зависит от длины анализируемой цепочки, но время может выражаться экспонентой.

Алгоритмы, рассматриваемые во втором разделе главы, носят табличный характер; это алгоритм Кока—Янгера—Касами и алгоритм Эрли. Они затрачивают емкость n^2 и время n^3 . Алгоритм Эрли работает для любой КС-грамматики и для него требуется время n^2 , если грамматика однозначная.

Алгоритмы этой главы включены в книгу главным образом для того, чтобы пояснить внутренние проблемы, связанные с построением анализаторов. С самого начала следует вполне определенно заявить, что в большинстве практических применений надо избегать возвратных алгоритмов разбора. Даже табличные методы (а они асимптотически гораздо более быстрые, чем алгоритмы с возвратами) неприемлемы, если для интересующего нас языка существует грамматика, к которой применимы более эффективные алгоритмы, описываемые в гл. 5 и 6. Можно почти не сомневаться в том, что фактически для всех языков программирования существуют легко анализируемые грамматики, к которым эти алгоритмы применимы.

Методы данной главы могут оказаться полезными в таких приложениях, когда исходные грамматики не обладают теми специальными свойствами, которых требуют алгоритмы, рассматриваемые в гл. 5 и 6. Например, если требуются неоднозначные

4.1. СИНТАКСИЧЕСКИЙ АНАЛИЗ С ВОЗВРАТАМИ

грамматики и интерес представляют все разборы цепочки, как это бывает при работе с естественными языками, можно обратиться к некоторым методам данной главы.

4.1.1. Моделирование МП-преобразователя

Предположим, что у нас есть недетерминированный МП-преобразователь P и входная цепочка w . Допустим, что все последовательности тактов, которые может сделать P для входной цепочки w , ограничены по длине. Тогда общее число различных последовательностей тактов МП-преобразователя P тоже конечно, хотя, возможно, и экспоненциально зависит от длины цепочки w . Грубый, зато прямой способ детерминированного моделирования МП-преобразователя P состоит в том, чтобы каким-то образом линейно упорядочить последовательности тактов и затем в предписанном порядке промоделировать каждую последовательность.

Если нас интересуют все выходные цепочки для данной входной цепочки w , то мы должны промоделировать все последовательности тактов. Если можно обойтись одной выходной цепочкой, то, обнаружив первую последовательность тактов, оканчивающуюся заключительной конфигурацией, можно прекратить моделирование P . Разумеется, если ни одна последовательность не оканчивается заключительной конфигурацией, придется перепробовать все.

Синтаксический анализ с возвратами можно представлять себе в следующем виде. Последовательности тактов располагают обычно в таком порядке, чтобы к моделированию очередной последовательности можно было перейти, возвратившись по последним сделанным тактам (т. е. прослеживая их) к конфигурации, в которой возможен еще не испытанный альтернативный торт. Этот торт и надо затем сделать. На практике для ускорения процесса возврата пользуются локальными критериями, позволяющими, не моделируя всей последовательности, определить, может ли она привести к заключительной конфигурации.

В этом разделе мы опишем, как можно детерминированно моделировать недетерминированный МП-преобразователь, используя возвраты. Затем исследуем два специальных случая. Первый — нисходящий анализ с возвратами, при котором для входной цепочки строится левый разбор. Во втором случае восходящий анализ с возвратами дает правый разбор.

4.1.1.1. Моделирование МП-преобразователя

Рассмотрим МП-преобразователь P и лежащий в его основе МП-автомат M . Если дана входная цепочка w , то нам было бы удобно знать, что, хотя M может недетерминированно перепро-

бовать много последовательностей тактов, каждая из них ограничена по длине. Если это так, то последовательности можно перепробовать в некотором разумном порядке. Если для входа w возможны бесконечные последовательности тактов, то по крайней мере в одном смысле нельзя осуществить полное прямое моделирование МП-автомата M .

Определение. МП-автомат $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ называется *незацикливающимся*, если для каждой цепочки $w \in \Sigma^*$ найдется такая константа k_w , что если $(q_0, w, Z_0) \vdash^m (q, x, \gamma)$, то $m \leq k_w$. МП-преобразователь назовем *незацикливающимся*, если лежащий в его основе МП-автомат незацикливающийся.

Интересно, при каких условиях, налагаемых на грамматику G , левый или правый анализатор для G будет незацикливающимся. Предоставляем читателю в качестве упражнений доказать, что левый анализатор не зацикливается тогда и только тогда, когда грамматика G не леворекурсивна, а правый анализатор не зацикливается тогда и только тогда, когда G не содержит циклов и e -правил. Мы покажем далее, что именно при этих условиях работают общие нисходящие и восходящие алгоритмы разбора с возвратами, хотя более общие алгоритмы работают на грамматиках из более широкого класса.

Заметим, что условие отсутствия циклов и e -правил на самом деле не очень ограничительно. Для каждого КС-языка существует грамматика, удовлетворяющая этим условиям, и, более того, ее можно получить из произвольной КС-грамматики, порождающей этот язык, с помощью простых преобразований (алгоритмы 2.10 и 2.11). Далее, если исходная грамматика однозначна, то преобразованная грамматика лево- и правопокрывает ее. Отсутствие левой рекурсии в этом смысле более стеснительное условие. Хотя для каждого КС-языка существует грамматика без левой рекурсии (теорема 2.18), может не быть покрывающей грамматики, удовлетворяющей этому условию (см. упр. 3.4.29).

Для того чтобы продемонстрировать, что такое анализ с возвратами и вообще моделирование недетерминированного МП-преобразователя, рассмотрим грамматику G с правилами

- (1) $S \rightarrow aSbS$
- (2) $S \rightarrow aS$
- (3) $S \rightarrow c$

Левым анализатором для G служит МП-преобразователь T с функцией переходов, определяемой равенствами

$$\begin{aligned}\delta(q, a, S) &= \{(q, SbS, 1), (q, S, 2)\} \\ \delta(q, c, S) &= \{(q, e, 3)\} \\ \delta(q, b, b) &= \{(q, e, e)\}\end{aligned}$$

Допустим, что нам надо разобрать входную цепочку $aacbc$. На рис. 4.1 изображено дерево, представляющее возможные последовательности тактов, которые может сделать T для этого входа.

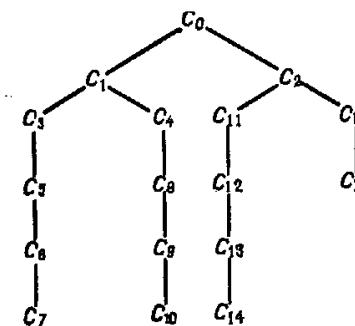


Рис. 4.1. Последовательности тактов анализатора.

Вершина C_0 представляет начальную конфигурацию $(q, aacbc, S, e)$. Правила, определяющие T , показывают, что из C_0 можно перейти в две следующие конфигурации, а именно $C_1 = (q, acbc, SbS, 1)$ и $C_2 = (q, acbc, S, 2)$. (Упорядочение здесь произвольное.) Из C_1 анализатор T может перейти в конфигурации $C_3 = (q, cbc, SbSbS, 11)$ и $C_4 = (q, cbc, SbS, 12)$. Из C_2 можно перейти в конфигурации $C_{11} = (q, cbc, SbS, 21)$ и $C_{15} = (q, cbc, S, 22)$. Остальные конфигурации определяются однозначно.

Один из способов нахождения всех разборов данной входной цепочки состоит в определении всех допускающих конфигураций, достижимых из C_0 в дереве конфигураций. Это можно сделать, прослеживая все возможные пути, начинающиеся в C_0 и заканчивающиеся в конфигурации, из которой дальнейший переход невозможен. Можно ввести порядок, в котором будут перебираться пути, упорядочив выборы очередных тактов, доступных анализатору T , для каждой комбинации состояния, входного символа и верхнего символа магазина. Например, в том случае, когда применимо правило $\delta(q, a, S)$, возьмем $(q, SbS, 1)$ в качестве первого выбора и $(q, S, 2)$ — в качестве второго выбора.

Посмотрим теперь, как можно определить все допускающие конфигурации анализатора T , систематически прослеживая все возможные для T последовательности тактов. Допустим, что из C_0 , делая первый выбор, мы получаем C_1 . Из C_1 , снова делая первый выбор, получаем C_3 . Продолжая в том же духе, мы прослеживаем последовательность конфигураций $C_0, C_1, C_3, C_5, C_6, C_7$. Вершина C_7 представляет заключительную конфигурацию $(q, e, bS, 1133)$, которая не является допускающей. Чтобы

определить, существует ли другая заключительная конфигурация, можно вернуться (сделать «возврат») по дереву, пока не встретится конфигурация, для которой возможен другой, еще не рассмотренный выбор очередного такта. Поэтому мы должны быть в состоянии восстановить конфигурацию C_6 по конфигурации C_7 . Возврат к C_6 из C_7 может включать обратный сдвиг головки на входе, восстановление предыдущего содержимого магазина и удаление выходных символов, выданных при переходе от C_6 к C_7 . Восстановив C_6 , мы должны сделать новый выбор очередного такта (если такой возможен). Так как в вершине C_6 других выборов нет, мы продолжаем возврат к C_5 и далее к C_3 и C_1 .

Из C_1 с помощью второго выбора такта для правила $\delta(q, a, S)$ можно получить конфигурацию C_4 . Затем, переходя через конфигурации C_8 и C_9 , можно получить конфигурацию $C_{10} = (q, e, e, 1233)$, которая оказывается допускающей.

После этого можно выдать в качестве выхода левый разбор 1233. Если мы заинтересованы в получении только одного разбора входной цепочки, то можно остановиться. Но если нас интересуют все разборы, можно вернуться к конфигурации C_0 и затем испробовать все конфигурации, достижимые из C_0 . Вершина C_{14} представляет другую допускающую конфигурацию, а именно $(q, e, e, 2133)$.

Мы остановимся после того, как рассмотрим все последовательности тактов, которые может сделать T . Если входная цепочка построена синтаксически неправильно, то придется рассмотреть все возможные последовательности тактов. Если исчерпаны все последовательности тактов, а допускающая конфигурация не обнаружена, надо выдать сообщение об ошибке.

Описанный нами анализ иллюстрирует характерные черты алгоритма, который иногда называют *недетерминированным*, т. е. на некоторых его шагах допускаются выборы и все их нужно перебрать¹⁾. На самом деле систематически порождаются все конфигурации, к которым могут привести данные, обрабатываемые алгоритмом, пока не встретится нужное решение или не исчерпаются все возможности. Понятие недетерминированного алгоритма применимо, таким образом, не только к моделированию недетерминированного автомата, но также и ко многим другим проблемам. Интересно отметить, что нечто аналогичное проблеме остановки для МП-преобразователей всегда приводит к вопросу о том, можно ли промоделировать недетерминированный алгоритм детерминированно. Конкретные примеры недетерминированных алгоритмов даны в упражнениях.

¹⁾ Лучше было бы назвать такой алгоритм *переборным*. Термин «недетерминированный алгоритм» обычно употребляют в том же смысле, что и «недетерминированный автомат» (произвольного типа). — Прим. перев.

При синтаксическом анализе задается обычно грамматика, а не МП-преобразователь. Поэтому мы обсудим сейчас восходящий и нисходящий анализ прямо в терминах заданной грамматики, а не левого или правого анализатора для нее. Однако способ работы соответствующих алгоритмов состоит именно в последовательном моделировании анализатора с магазинной памятью. Но вместо обхода всевозможных последовательностей тактов анализатора мы будем обходить всевозможные выводы, совместимые с данным входом.

4.1.2. Неформальное описание нисходящего разбора

Происхождение названия нисходящего синтаксического анализа связано с тем, что дерево разбора входной цепочки строится сверху вниз, начиная с корня и нисходя к листьям. Начнем с того, что возьмем какую-нибудь грамматику и для каждого нетерминала занумеруем в некотором порядке все правые части правил, у которых левой частью является данный нетерминал (их называют *альтернативами* этого нетерминала), т. е. если $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ — все A -правила данной грамматики, то припишем некоторый порядок цепочкам α_i (альтернативам нетерминала A).

Например, рассмотрим грамматику, упомянутую в предыдущем разделе. Ее S -правила

$$S \rightarrow aSbS | aS | c$$

будем использовать в том порядке, как они написаны, т. е. $aSbS$ будет первой альтернативой нетерминала S , aS — второй и c — третьей. Допустим, что входная цепочка — $aabc$. В дальнейших рассуждениях мы будем употреблять входной указатель, который в начальный момент указывает на самый левый символ входной цепочки.

Коротко говоря, нисходящий анализатор пытается построить дерево вывода входной цепочки следующим образом. Процесс начинается с дерева, состоящего из одной вершины, помеченной S . Это начальная активная вершина. Затем рекурсивно выполняются такие шаги:

(1) Если активная вершина помечена нетерминалом, скажем A , то выбрать первую его альтернативу, скажем $X_1 \dots X_k$, и добавить k прямых потомков вершины A с метками X_1, \dots, X_k . Сделать X_1 активной вершиной. Если $k = 0$, сделать активной вершину, расположенную непосредственно справа от A .

(2) Если активная вершина помечена терминалом, скажем a , сравнить текущий входной символ с a . Если они совпадают, сделать активной вершину, расположенную непосредственно

справа от a , и сдвинуть входной указатель на один символ вправо. Если a не совпадает с текущим входным символом, вернуться к вершине, к которой применялось предыдущее правило, вернуть, куда надо, входной указатель, если это необходимо, и испытать следующую альтернативу. Если альтернатив больше нет, вернуться к следующей предыдущей вершине, и т. д.

Всякий раз мы пытаемся сохранить совместимость построенного дерева с входной цепочкой, т. е. если xa — крона дерева, построенного к данному моменту, где цепочка α либо пуста,

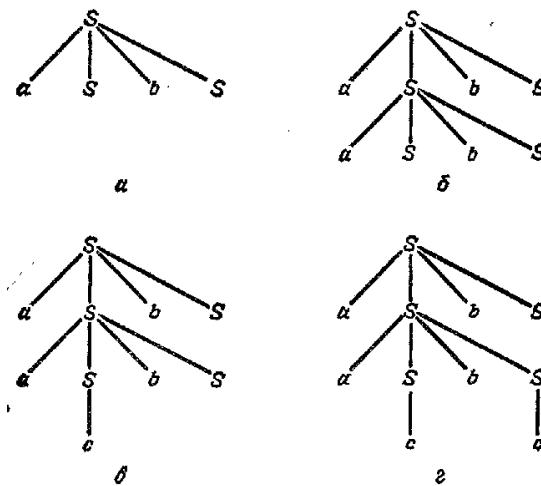


Рис. 4.2. Частичные деревья разбора.

либо начинается нетерминальным символом, то x — префикс входной цепочки.

В нашем примере мы начнем с дерева, состоящего из одной вершины, помеченной S . Затем применим первое S -правило, расширяя дерево так, чтобы оно было совместимо с данной входной цепочкой. Здесь применим правило $S \rightarrow aSbS$, чтобы расширить начальное дерево до дерева, изображенного на рис. 4.2, а. Так как активной вершиной дерева в этот момент станет a и первый входной символ тоже a , мы передвинем входной указатель на второй входной символ и сделаем нетерминал S , стоящий непосредственно справа от a , новым активным символом. Затем развернем этот символ S с помощью первой альтернативы и получим дерево на рис. 4.2, б. Так как новый активный символ a совпадает со вторым входным символом, передвинем входной указатель на третий входной символ.

Теперь развернем самый левый нетерминал S на рис. 4.2, б, но на этот раз нельзя применить ни первую, ни вторую альтер-

нативу, так как в результате получится левовыводимая цепочка, несовместимая с входной цепочкой. Поэтому мы воспользуемся третьей альтернативой и приедем к рис. 4.2, в. Сейчас можно передвинуть входной указатель с третьего на четвертый, а затем на пятый входной символ, так как c и b — следующие два активных символа левовыводимой цепочки, представленной на рис. 4.2, в.

Далее можно с помощью третьей альтернативы для S развернуть самый левый символ S на рис. 4.2, в и получить рис. 4.2, г. (Первые две альтернативы снова несовместимы с входом.) Пятый входной символ — c , и потому входной указатель можно передвинуть на один символ вправо. (Мы предполагаем, что конец входной цепочки обозначается маркером.) Однако дерево на рис. 4.2, г порождает дополнительные символы, а именно bS , которых нет во входной цепочке, так что теперь мы знаем, что при поиске правильного разбора входной цепочки пошли по неверному пути.

Если вспомнить МП-анализатор из разд. 4.1.1, то окажется, что мы прошли через последовательность конфигураций $C_0, C_1, C_3, C_5, C_6, C_7$, и из C_7 переход невозможен.

Итак, нам придется искать какую-то другую левовыводимую цепочку. Сначала посмотрим, нет ли другой альтернативы для правила, использованного при построении дерева на рис. 4.2, г из предыдущего дерева. Оказывается, такой альтернативы нет, так как для получения рис. 4.2, г из рис. 4.2, в использовалось последнее S -правило $S \rightarrow c$.

Тогда мы возвращаемся к дереву рис. 4.2, в и переставляем указатель на позицию 3. Проверяем, есть ли еще альтернативы правила, использованного при построении дерева на рис. 4.2, в из предыдущего дерева. Видим, что их нет, так как мы применяли правило $S \rightarrow c$. Поэтому мы возвращаемся к рис. 4.2, б, переставляя входной указатель на позицию 2. При переходе к рис. 4.2, б от рис. 4.2, а применялась первая альтернатива, так что теперь испытаем вторую альтернативу и получим дерево на рис. 4.3, а.

Входной указатель можно передвинуть вперед на позицию 3, поскольку порожденный символ a совпадает с входным символом a в позиции 2. Далее, чтобы развернуть самый левый символ S на рис. 4.3, а и получить рис. 4.3, б, можно применить только третью альтернативу. Входные символы в позициях 3 и 4 совпадают теперь с порожденными, так что можно передвинуть входной указатель на позицию 5. К нетерминалу S на рис. 4.3, б можно применить только третью альтернативу и получить рис. 4.3, в. Последний входной символ совпадает с самым правым символом на рис. 4.3, в. Таким образом, мы знаем теперь, что на рис. 4.3, в изображен правильный разбор входной цепоч-

ки. Здесь можно либо сделать возврат, чтобы продолжать поиск других разборов, либо остановиться.

Из-за того, что наша грамматика не леворекурсивна, мы с помощью возвратов в конце концов исчерпаем все возможности, т. е. окажемся в корне⁷ и все альтернативы для S будут уже

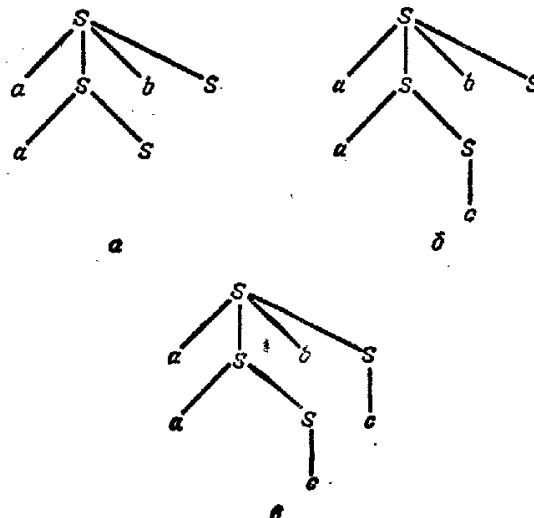


Рис. 4.3. Дальнейшие попытки разбора.

испытаны. В этот момент можно остановиться и, если нужный разбор не обнаружен, выдать сообщение о том, что входная цепочка синтаксически неправильна.

В описанной процедуре есть коварная ловушка. Если грамматика леворекурсивна, то процесс может никогда не остановиться. Например, допустим, что $A\alpha$ — первая альтернатива для A . Тогда это правило будет применяться всякий раз, когда надо развернуть A .

Можно возразить, сказав, что этой проблемы можно было бы избежать, испытывая альтернативу $A\alpha$ последней. Однако левая рекурсия может оказаться гораздо более тонкой и включать несколько правил. Например, первым A -правилом могло бы быть $A \rightarrow SC$. Тогда если $S \rightarrow AB$ — первое S -правило, то мы получим $A \Rightarrow SC \Rightarrow ABC$, и эта картина будет повторяться. Даже если будет найдено подходящее упорядочение правил, на синтаксически неправильных входных цепочках леворекурсивные циклы будут встречаться снова и снова, так как все предыдущие выборы будут безуспешными.

Вторая попытка свести на нет эффект левой рекурсии могла бы состоять в том, чтобы ограничить число вершин проме-

жуточного дерева в зависимости от длины входной цепочки. Если нам дана КС-грамматика $G = (N, \Sigma, P, S)$, у которой $\# N = k$, и входная цепочка w длины n , то можно показать, что для $w \in L(G)$ существует хотя бы одно дерево вывода, в котором длины всех путей не большие kn . Таким образом, можно было бы ограничить поиск деревьями, глубина (длина наибольшего пути) которых не превосходит kn .

Однако для некоторых грамматик число деревьев глубины $\leq d$ может слишком быстро расти с ростом d . Рассмотрим, например, грамматику G с правилами $S \rightarrow SS | e$. Для нее число деревьев выводов глубины d задается рекуррентными соотношениями

$$\begin{aligned} D(1) &= 1 \\ D(d) &= (D(d-1))^2 + 1 \end{aligned}$$

Значения функции $D(d)$ для d от 1 до 6 приведены на рис. 4.4.

$D(d)$ растет очень быстро, быстрее, чем $2^{2^{d-4}}$. (См. также упр. 4.1.4.) В результате любую грамматику, в которой встречаются два правила такого вида, нельзя разумно проанализи-

d	$D(d)$
1	1
2	2
3	5
4	26
5	677
6	458 330

Рис. 4.4. Значения $D(d)$.

ровать с помощью описанного варианта алгоритма нисходящего разбора.

По этим причинам общепринятый подход состоит в том, чтобы применять нисходящий алгоритм разбора только к грамматикам без левой рекурсии.

4.1.3. Алгоритм нисходящего разбора

Теперь мы готовы описать наш алгоритм нисходящего разбора с возвратами. В алгоритме используются два магазина (L_1 и L_2) и счетчик, в котором хранится текущая позиция входного указателя. Чтобы точно описать алгоритм, воспользуемся несколькими стилизованными обозначениями, похожими на те, что

употреблялись при описании конфигураций МП-преобразователя.

Алгоритм 4.1. Нисходящий разбор с возвратами.

Вход. Не леворекурсивная КС-грамматика $G = (N, \Sigma, P, S)$ и входная цепочка $w = a_1 a_2 \dots a_n$ ($n \geq 0$). Предполагается, что правила из P занумерованы числами $1, 2, \dots, p$.

Выход. Один левый разбор цепочки w , если таковой существует. В противном случае — слово „ошибка“.

Метод.

(1) Для каждого нетерминала $A \in N$ упорядочить его альтернативы. Пусть A_i — индекс i -й альтернативы нетерминала A . Например, если $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$ — все A -правила из P и альтернативы упорядочены так, как записаны, то A_1 — индекс альтернативы α_1 , A_2 — индекс альтернативы α_2 и т. д.

(2) Четверкой (s, i, α, β) будем обозначать конфигурацию алгоритма:

- (а) s обозначает состояние алгоритма;
- (б) i указывает позицию входного указателя (предполагается, что $(n+1)$ -м „входным символом“ служит правый концевой маркер $\$$);
- (в) α представляет содержимое первого магазина ($L1$);
- (г) β представляет содержимое второго магазина ($L2$).

Будем считать, что верх первого магазина расположен справа, а верх второго магазина — слева. Магазин $L2$ служит для представления „текущей“ левовыводимой цепочки, т. е. той, которая получилась к данному моменту в результате развертки нетерминалов. В соответствии с неформальным описанием нисходящего разбора в разд. 4.1.1 верхний символ магазина $L2$ будет символом, помечающим активную вершину порожденного к данному моменту дерева вывода. В магазине $L1$ представлены текущая история проделанных выборов альтернатив и выходные символы, по которым прошла входная головка. Алгоритм находится в одном из трех состояний q , b или t , где q — состояние нормальной деятельности, b — состояние возврата, t — заключительное состояние.

(3) Начальная конфигурация алгоритма — $(q, 1, e, \$\$)$.

(4) Существует шесть типов шагов. Эти шаги будут описаны в терминах их воздействия на конфигурацию алгоритма. Ядро алгоритма — вычисление последовательных конфигураций, определяемое через „отношение перехода“ \vdash . Запись $(s, i, \alpha, \beta) \vdash (s', i', \alpha', \beta')$ означает, что если (s, i, α, β) — текущая конфигурация, то нужно перейти в следующую конфигурацию $(s', i', \alpha', \beta')$. Если не оговорено противное, то i — число от 1 до $n+1$,

$\alpha \in (\Sigma \cup I)^*$, где I — множество индексов альтернатив, $\beta \in (N \cup \Sigma)^*$. Шесть типов шагов определяются так:

(а) *Разрастание дерева*

$$(q, i, \alpha, A\beta) \vdash (q, i, \alpha A_1, \gamma_1 \beta)$$

где $A \rightarrow \gamma_1$ — правило из P и γ_1 — первая альтернатива нетерминала A . Этот шаг соответствует разрастанию частичного дерева вывода, при котором применяется первая альтернатива самого левого нетерминала дерева.

(б) *Успешное сравнение входного символа с порожденным символом*

$$(q, i, \alpha, a\beta) \vdash (q, i+1, \alpha a, \beta)$$

при условии, что $a_i = a$ ($i \leq n$). Если i -й входной символ совпадает с очередным порожденным терминальным символом, то он передается из $L2$ в $L1$, а позиция входного указателя увеличивается на единицу.

(в) *Успешное завершение*

$$(q, n+1, \alpha, \$) \vdash (t, n+1, \alpha, e)$$

Достигнут конец входной цепочки и найдена левовыводимая цепочка, совпадающая с входной. Левый разбор входной цепочки восстанавливается по цепочке α с помощью такого гомоиорфизма h , что $h(a) = e$ для всех $a \in \Sigma$, $h(A_i) = p$, где p — номер правила $A \rightarrow \gamma$ и γ — i -я альтернатива нетерминала A .

(г) *Неудачное сравнение входного символа с порожденным символом*

$$(q, i, \alpha, a\beta) \vdash (b, i, \alpha, a\beta) \quad (a_i \neq a)$$

Надо перейти в состояние возврата, как только обнаружится, что порожденная левовыводимая цепочка не совместима с входной цепочкой.

(д) *Возврат по входу*

$$(b, i, \alpha a, \beta) \vdash (b, i-1, \alpha, a\beta)$$

для всех $a \in \Sigma$. В состоянии возврата входные символы переносятся назад из $L1$ в $L2$.

(е) *Испытание очередной альтернативы*

$$(b, i, \alpha A_j, \gamma_j, \beta) \vdash$$

- (i) $(q, i, \alpha A_{j+1}, \gamma_{j+1} \beta)$, если γ_{j+1} — $(j+1)$ -я альтернатива нетерминала A . (Заметьте, что γ_j в $L2$ заменяется на γ_{j+1} .)
- (ii) Следующая конфигурация невозможна, если $i=1$, $A=S$ и S имеет только j альтернатив. (Это условие указывает на то, что всевозможные левовыводимые це-

почки, совместимые с входной цепочкой w , уже исчерпаны, а ее разбор не найден.)

(iii) $(b, i, \alpha, A\beta)$ в оставшихся случаях. (Здесь исчерпаны все альтернативы нетерминала A и дальнейший возврат происходит путем удаления A_j из $L1$ и замены в $L2$ цепочки γ_j на A .)

Алгоритм выполняется следующим образом:

Шаг 1. Исходя из начальной конфигурации, вычислять последующие конфигурации $C_0 \vdash C_1 \vdash \dots \vdash C_i \vdash \dots$, пока их можно вычислить.

Шаг 2. Если последняя вычисленная конфигурация — $(t, n+1, \psi, e)$, выдать $h(\psi)$ и остановиться. В противном случае выдать сигнал об ошибке. \square

Алгоритм 4.1 — это по существу описанный ранее неформально алгоритм, к которому для выполнения возвратов добавлена некоторая „бухгалтерия“.

Пример 4.1. Рассмотрим работу алгоритма 4.1 на грамматике G с правилами

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

Пусть E_1 служит индексом для $T+E$, E_2 — для T , T_1 — для $F*T$, T_2 — для F и F_1 — для a . Для входа $a+a$ алгоритм 4.1 вычисляет такую последовательность конфигураций:

- $$\begin{aligned} (q, 1, e, E \$) &\vdash (q, 1, E_1, T + E \$) \\ &\vdash (q, 1, E_1 T_1, F * T + E \$) \\ &\vdash (q, 1, E_1 T_1 F_1, a * T + E \$) \\ &\vdash (q, 2, E_1 T_1 F_1 a, * T + E \$) \\ &\vdash (b, 2, E_1 T_1 F_1 a, * T + E \$) \\ &\vdash (b, 1, E_1 T_1 F_1, a * T + E \$) \\ &\vdash (b, 1, E_1 T_1, F * T + E \$) \\ &\vdash (q, 1, E_1 T_2, F + E \$) \\ &\vdash (q, 1, E_1 T_2 F_1, a + E \$) \\ &\vdash (q, 2, E_1 T_2 F_1 a, + E \$) \\ &\vdash (q, 3, E_1 T_2 F_1 a +, E \$) \\ &\vdash (q, 3, E_1 T_2 F_1 a + E_1, T + E \$) \\ &\vdash (q, 3, E_1 T_2 F_1 a + E_1 T_1, F * T + E \$) \end{aligned}$$

- $$\begin{aligned} &\vdash (q, 3, E_1 T_2 F_1 a + E_1 T_1 F_1, a * T + E \$) \\ &\vdash (q, 4, E_1 T_2 F_1 a + E_1 T_1 F_1 a, * T + E \$) \\ &\vdash (b, 4, E_1 T_2 F_1 a + E_1 T_1 F_1 a, * T + E \$) \\ &\vdash (b, 3, E_1 T_2 F_1 a + E_1 T_1 F_1, a * T + E \$) \\ &\vdash (b, 3, E_1 T_2 F_1 a + E_1 T_1, F * T + E \$) \\ &\vdash (q, 3, E_1 T_2 F_1 a + E_1 T_2, F + E \$) \\ &\vdash (q, 3, E_1 T_2 F_1 a + E_1 T_2 F_1, a + E \$) \\ &\vdash (q, 4, E_1 T_2 F_1 a + E_1 T_2 F_1 a, + E \$) \\ &\vdash (b, 4, E_1 T_2 F_1 a + E_1 T_2 F_1 a, + E \$) \\ &\vdash (b, 3, E_1 T_2 F_1 a + E_1 T_2 F_1, a + E \$) \\ &\vdash (b, 3, E_1 T_2 F_1 a + E_1 T_2, F + E \$) \\ &\vdash (b, 3, E_1 T_2 F_1 a + E_1, T + E \$) \\ &\vdash (q, 3, E_1 T_2 F_1 a + E_2, T \$) \\ &\vdash (q, 3, E_1 T_2 F_1 a + E_2 T_1, F * T \$) \\ &\vdash (q, 3, E_1 T_2 F_1 a + E_2 T_1 F_1, a * T \$) \\ &\vdash (q, 4, E_1 T_2 F_1 a + E_2 T_1 F_1 a, * T \$) \\ &\vdash (b, 4, E_1 T_2 F_1 a + E_2 T_1 F_1 a, * T \$) \\ &\vdash (b, 3, E_1 T_2 F_1 a + E_2 T_1 F_1, a * T \$) \\ &\vdash (b, 3, E_1 T_2 F_1 a + E_2 T_1, F * T \$) \\ &\vdash (q, 3, E_1 T_2 F_1 a + E_2 T_2, F \$) \\ &\vdash (q, 3, E_1 T_2 F_1 a + E_2 T_2 F_1, a \$) \\ &\vdash (q, 4, E_1 T_2 F_1 a + E_2 T_2 F_1 a, \$) \\ &\vdash (t, 4, E_1 T_2 F_1 a + E_2 T_2 F_1 a, e) \end{aligned}$$

В результате получается левый разбор $h(E_1 T_2 F_1 a + E_2 T_2 F_1 a) = 145245$. \square

Теперь покажем, что алгоритм 4.1 действительно дает левый разбор цепочки w в соответствии с грамматикой G , если он существует.

Определение. Частичным левым разбором назовем последовательность правил, использованных при левом выводе левоуводимой цепочки. Будем говорить, что частичный левый разбор *совместим* с входной цепочкой w , если соответствующая левоуводимая цепочка совместима с w , т. е. ее терминальный префикс до первого нетерминала является префиксом цепочки w .

Пусть $G = (N, \Sigma, P, S)$ — не леворекурсивная грамматика и $w = a_1 \dots a_n$ — входная цепочка. Определим *последовательность* $\pi_0, \pi_1, \dots, \pi_i, \dots$ совместимых с w частичных левых разборов.

(1) $\pi_0 = e$ представляет вывод нетерминала S из S (строго говоря, π_0 — не разбор).

(2) π_i — номер правила $S \rightarrow \alpha$, где α — первая альтернатива нетерминала S .

(3) π_i определяется так. Допустим, что $S_{\pi_{i-1}} \Rightarrow xA\gamma$. Пусть β — такая альтернатива нетерминала A с наименьшим номером, если она существует, что $x\beta\gamma = x\delta b$, где b либо e , либо начинается нетерминалом, и $x\gamma$ — префикс цепочки w . Тогда $\pi_i = \pi_{i-1} A_k$, где k — номер альтернативы β . В этом случае будем называть π_i *продолжением* разбора π_{i-1} . Если, с другой стороны, такой альтернативы β нет или $S_{\pi_{i-1}} \Rightarrow x$ для некоторой терминальной цепочки x , то пусть j — наибольшее из чисел, меньших $i-1$, для которых выполняется следующее условие:

Пусть $S_{\pi_j} \Rightarrow xB\gamma$ и π_{j+1} — продолжение разбора π_j , причем на последнем шаге разбора π_{j+1} нетерминал B заменяется альтернативой α_m . Пусть α_m — первая из альтернатив нетерминала B , которая следует за α_k в упорядочении альтернатив этого нетерминала, и если записать $x\alpha_m\gamma = x\delta b$, где b либо e , либо начинается нетерминалом, то $x\gamma$ — префикс цепочки w .

Тогда $\pi_i = \pi_j B_m$, где B_m — номер правила $B \rightarrow \alpha_m$. В этом случае назовем π_i *модификацией* разбора π_{i-1} .

Если сформулированное условие не выполняется, то разбор π_i не определен.

Пример 4.2. Для грамматики G из примера 4.1 и входной цепочки $a+a$ получается такая последовательность совместимых частичных левых разборов:

e
1
13
14
145
1451
14513
14514
1452
14523
14524
145245
2
23
24

Заметим, что последовательность совместимых частичных левых разборов вплоть до первого корректного разбора тесно связана с последовательностью цепочек, появляющихся в магазине $L1$. Если отвлечься от терминальных символов, которые

могут быть в $L1$, то эти две последовательности совпадают, за исключением того, что $L1$ может содержать последовательности, не совместимые со входом. Когда в $L1$ появляется такая последовательность, сразу происходит возврат. \square

Должно быть очевидно, что последовательность совместимых частичных левых разборов единственна и включает все совместимые частичные левые разборы в естественном лексикографическом порядке.

Лемма 4.1. Пусть $G = (N, \Sigma, P, S)$ — не леворекурсивная грамматика. Тогда найдется такая константа c , что если $A \Rightarrow^i wBa$ и $|w| = n$, то $i \leq c^{n+2}$ ¹.

Доказательство. Пусть $\# N = k$. Рассмотрим дерево D левого вывода $A \Rightarrow^i wBa$. Допустим, что существует путь от корня к листу длины, большей, чем $k(n+2)$. Пусть n_0 — вершина, помеченная нетерминалом B , явно указанным в цепочке wBa . Если

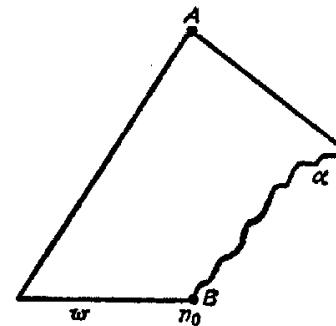


Рис. 4.5. Дерево вывода D .

этот путь оканчивается листом, расположенным справа от n_0 , то путь к n_0 должен быть не менее длинным. Это следует из того, что в левом выводе правило всегда применяется к самому левому нетерминалу. Поэтому прямой предок каждой вершины, расположенной справа от n_0 , является предком вершины n_0 . Дерево D показано на рис. 4.5.

Таким образом, если в D есть путь длины, большей, чем $k(n+2)$, то можно найти один такой путь, достигающий n_0 или вершины, расположенной слева от n_0 . Тогда на этом пути можно найти $k+1$ последовательных вершин n_1, \dots, n_{k+1} , каждая из которых порождает один и тот же кусок цепочки wB . Все пря-

¹) На самом деле можно доказать более сильное утверждение: i линейно зависит от n . Однако пока нам достаточно и этого результата, а потом мы с его помощью докажем более сильное утверждение.

мые потомки вершины n_i ($1 \leq i \leq k$), расположенные слева от n_{i+1} , порождают e . Следовательно, можно найти две вершины из n_1, \dots, n_{k+1} с одной и той же меткой, и легко видеть, что эта метка — леворекурсивный нетерминал.

Отсюда заключаем, что в D нет путей, длина которых больше $k(n+2)$. Пусть l — длина самой длинной из правых частей правил грамматики G . Тогда D имеет не более $l^{k(n+2)}$ внутренних вершин. Поэтому если $A \Rightarrow^i wB\alpha$, то $i \leq l^{k(n+2)}$. Возьмем $c = l^k$; лемма доказана. \square

Следствие. Пусть $G = (N, \Sigma, P, S)$ — не леворекурсивная грамматика. Тогда найдется такая константа c' , что если $S \Rightarrow^i wB\alpha$ и $w \neq e$, то $|\alpha| \leq c' |w|$.

Доказательство. Мы показали (см. рис. 4.5), что длина пути от корня к вершине n_0 не превосходит $k(|w| + 2)$. Поэтому $|\alpha| \leq kl(|w| + 2)$. Возьмем $c' = 3kl$. \square

Лемма 4.2. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, не содержащая бесполезных нетерминалов, и $w = a_1 a_2 \dots a_n \in \Sigma^*$. Последовательность совместимых с w левых разборов конечна тогда и только тогда, когда G не леворекурсивна.

Доказательство. Если G леворекурсивна, то ясно, что для некоторой терминальной цепочки последовательность совместимых с нею левых разборов бесконечна. Допустим, что G не леворекурсивна. Тогда по лемме 4.1 длина каждого совместимого с w левого разбора не превосходит c^{n+2} . Таким образом, число совместимых с w левых разборов конечно. \square

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика и γ — последовательность индексов альтернатив (т. е. нетерминалов с номерами) и терминалов. Пусть π — частичный левый разбор, совместимый с w . Будем говорить, что γ описывает π , если выполняется следующее условие:

(1) Пусть $\pi = p_1 \dots p_k$ и $S = \alpha_0 p_1 \Rightarrow \alpha_1 p_2 \Rightarrow \alpha_2 \dots p_k \Rightarrow \alpha_k$. Пусть $\alpha_i = x_i \beta_i$, где β_i — либо e , либо начинается нетерминалом. Тогда $\gamma = A_i w_1 A_i w_2 \dots A_{i_k} w_k$, где A_{i_j} — индекс правила (альтернативы), примененного при переходе от α_{j-1} к α_j , и w_j — такой суффикс цепочки x_j , что $x_j = x_{j-1} w_j$.

Лемма 4.3. Пусть $G = (N, \Sigma, P, S)$ — не леворекурсивная грамматика и $\pi_0, \pi_1, \dots, \pi_i, \dots$ — последовательность совместимых с w частичных левых разборов. Допустим, что ни один из разборов π_0, \dots, π_i не является левым разбором цепочки w . Пусть $S \pi_i \Rightarrow \alpha$ и $S \pi_{i+1} \Rightarrow \beta$. Запишем α и β в виде $x\alpha_1$ и $y\beta_1$ соответственно, где каждая из цепочек α_1 и β_1 — либо e , либо начинается

нетерминалом. Тогда в алгоритме 4.1

$$(q, 1, e, \$\$) \vdash^* (q, j_1, \gamma_1, \alpha_1\$) \vdash^* (q, j_2, \gamma_2, \beta_1\$)$$

где $j_1 = |x| + 1$, $j_2 = |y| + 1$, а γ_1 и γ_2 описывают соответственно π_i и π_{i+1} .

Доказательство. Доказательство проводится индукцией по i . Базис, $i = 0$, тривиален. Для доказательства шага индукции рассмотрим отдельно два случая.

Случай 1: π_{i+1} — продолжение разбора π_i . Пусть первый символ цепочки α_1 — это нетерминал A , а его альтернативы — $\delta_1, \dots, \delta_k$. Если цепочка $x\delta_j$ не совместима с входной цепочкой, то в случае замены алгоритмом 4.1 нетерминала A цепочкой δ_j правила (г), (д) и (е) гарантируют, что следующей будет испытана альтернатива δ_{j+1} . Так как мы предположили, что π_{i+1} — продолжение разбора π_i , то нужная альтернатива нетерминала A впоследствии будет испытана. После того как по правилу (б) произойдет сдвиг по входу, будет достигнута конфигурация $(q, j_2, \gamma_2, \beta_1\$)$.

Случай 2: π_{i+1} — модификация разбора π_i . Все неиспытанные альтернативы нетерминала A сразу приводят к возврату, и по правилам (д) и (е) iii) содержимое магазина $L1$ в конце концов будет описывать частичный левый разбор π_i , упоминаемый в определении модификации. Тогда, как в случае 1, будет достигнута конфигурация $(q, j_2, \gamma_2, \beta_1\$)$. \square

Теорема 4.1. Алгоритм 4.1 дает левый разбор цепочки w , если он существует, а в противном случае выдает сообщение об ошибке.

Доказательство. Из леммы 4.3 видно, что алгоритм перебирает все совместимые частичные левые разборы до тех пор, пока не обнаружится левый разбор входной цепочки или не будут исчерпаны все совместимые частичные левые разборы. По лемме 4.2 число таких разборов конечно, так что алгоритм рано или поздно остановится. \square

4.1.4. Временная и емкостная сложность исходящего анализатора

Рассмотрим вычислительную машину, у которой емкость памяти, необходимая для хранения конфигураций алгоритма 4.1, пропорциональна сумме длин обоих магазинов; это предположение вполне разумно. Разумно также предположить, что время, затрачиваемое на вычисление конфигурации C_2 по конфигурации C_1 , если $C_1 \vdash C_2$, не зависит от этих конфигураций. Покажем, что при этих предположениях алгоритм 4.1 требует линей-

ной емкости памяти и не более чем экспоненциального времени работы, рассматриваемых как функции от длины входной цепочки. Для доказательства нам понадобится следующая лемма, усиливающая лемму 4.1.

Лемма 4.4. Пусть $G = (N, \Sigma, P, S)$ — не леворекурсивная грамматика. Тогда найдется такая константа c , что если $A \Rightarrow^i \alpha$ и $|\alpha| \geq 1$, то $i \leq c|\alpha|$.

Доказательство. По лемме 4.1 найдется такая константа c_1 , что если $A \Rightarrow^i e$, то $i \leq c_1$. Пусть $\#N = k$ и l — длина самой длинной из правых частей правил. По лемме 2.16 множество N можно представить в виде $\{A_0, A_1, \dots, A_{k-1}\}$, где $j > i$, если $A_i \Rightarrow^+ A_j \alpha$. Индукцией по параметру $p = kn - j$ докажем, что

$$(4.1.1) \quad \text{если } A_j \Rightarrow^i \alpha \text{ и } |\alpha| = n \geq 1, \text{ то } i \leq klc_1|\alpha| - jlc_1$$

Базис. Базис, $p = 0$, выполняется автоматически, так как мы предполагаем, что $n \geq 1$.

Шаг индукции. Допустим, что (4.1.1) истинно для всех значений параметра $kn - j < p$. Рассмотрим это утверждение для параметра $kn - j = p$. Пусть первым шагом упоминаемого в нем вывода был $A_j \Rightarrow X_1 \dots X_r$, для $r \leq l$. Тогда можно записать $\alpha = \alpha_1 \dots \alpha_r$, где $X_m \Rightarrow^{i_m} \alpha_m$, $1 \leq m \leq r$ и $i = 1 + i_1 + \dots + i_m$. Пусть $\alpha_1 = \alpha_2 = \dots = \alpha_{s-1} = e$ и $\alpha_s \neq e$. Так как $\alpha \neq e$, то такое число s существует. Исследуем отдельно два случая.

Случай 1: X_s — нетерминал, скажем A_g . Тогда $A_j \Rightarrow^* A_g X_{s+1} \dots X_r$, так что $g > j$. Так как $k|\alpha_s| - g < p$, то в силу (4.1.1) $i_s \leq klc_1|\alpha_s| - glc_1$. Так как $|\alpha_m| < |\alpha|$ для $s+1 \leq m \leq r$, то из (4.1.1) следует, что $i_m \leq klc_1|\alpha_m|$ всякий раз, когда $s+1 \leq m \leq r$ и $\alpha_m \neq e$. Разумеется, не более $l-1$ цепочек из $\alpha_1, \dots, \alpha_r$ пусты, так что суммирование чисел i_m по тем m , для которых $\alpha_m = e$, даст число, не превосходящее $(l-1)c_1$. Таким образом,

$$\begin{aligned} i &= 1 + i_1 + \dots + i_r \\ &\leq 1 + (l-1)c_1 + klc_1|\alpha| - glc_1 \\ &\leq klc_1|\alpha| - (g-1)lc_1 \\ &\leq klc_1|\alpha| - jlc_1 \end{aligned}$$

Случай 2: X_s — терминал. В качестве упражнения предлагаем показать, что в этом случае $i \leq 1 + klc_1(|\alpha| - 1) \leq klc_1|\alpha| - jlc_1$.

Из (4.1.1) вытекает, что если $S \Rightarrow^i \alpha$ и $|\alpha| \geq 1$, то $i \leq klc_1|\alpha|$. Возьмем $c = klc_1$; лемма доказана. \square

Следствие 1. Пусть $G = (N, \Sigma, P, S)$ — не леворекурсивная грамматика. Тогда найдется такая константа c' , что если $S \Rightarrow^i wA\alpha$ и $w \neq e$, то $i \leq c'|\alpha|$.

Доказательство. Согласно следствию леммы 4.1, найдется такая константа c'' , что $|\alpha| \leq c''|w|$. По лемме 4.4 $i \leq c|wA\alpha|$. Так как $|wA\alpha| \leq (2 + c'')|w|$, то выбор $c' = c(2 + c'')$ дает нужный результат. \square

Следствие 2. Пусть $G = (N, \Sigma, P, S)$ — не леворекурсивная грамматика. Тогда найдется такая константа k , что если π — частичный левый разбор, совместимый с цепочкой w , и $S_\pi \Rightarrow x\alpha$, где α — либо e , либо начинается нетерминалом, то $|\pi| \leq k(|w| + 1)$.

Доказательство. Если $x \neq e$, то по следствию 1 $|\pi| \leq c'|\alpha|$. Так как $|\alpha| \leq |w|$, то $|\pi| \leq c'|\alpha|$. В качестве упражнения можно показать, что если $x = e$, то $|\pi| \leq c'$. Поэтому в любом случае $|\pi| \leq c'(|w| + 1)$. \square

Теорема 4.2. Существует такая константа c , что алгоритм 4.1 для входной цепочки w длины $n \geq 1$ использует не более сп ячеек, если для каждого символа из обеих магазинных цепочек, входящих в конфигурации, требуется только одна ячейка.

Доказательство. Не считая символа $\$$, содержимое магазина $L2$ является частью левовыводимой цепочки α , для которой $S_\pi \Rightarrow \alpha$, где π — частичный левый разбор, совместимый с w . В силу следствия 2 леммы 4.4 $|\pi| \leq k(|w| + 1)$. Так как длины правых частей правил ограничены, скажем величиной l , то $|\alpha| \leq kl(|w| + 1) \leq 2kl|w|$. Следовательно, длина магазина $L2$ превосходит $2kl|w| + 1$.

Магазин $L1$ содержит часть левовыводимой цепочки α (весь терминальный префикс или его часть) и $|\pi|$ индексов. В силу следствия 2 леммы 4.4 длина магазина $L1$ не превосходит $2k(l+1)(|w| + 1)$. Таким образом, сумма длин обоих магазинов пропорциональна $|w|$. \square

Теорема 4.3. Существует такая константа c , что алгоритм 4.1 при обработке входной цепочки w длины $n \geq 1$ делает не более c^n элементарных операций, при условии что вычисление одного шага алгоритма 4.1 требует фиксированного числа элементарных операций.

Доказательство. В силу следствия 2 длина каждого частичного левого разбора, совместимого с w , не превосходит c_1n для некоторого c_1 . Таким образом, число различных частичных левых разборов, совместимых с w , не больше c_2^n , где c_2 — некоторая константа. Алгоритм 4.1 в промежутках между конфигурациями, в которых содержимое магазина $L1$ описывает

последовательные частичные левые разборы, вычисляет самое большое n конфигураций. Общее число конфигураций, вычисляемых алгоритмом 4.1, не превосходит, таким образом, nc_2^n . Из формулы бинома непосредственно следует, что $nc_2^n \leq (c_2 + 1)^n$. Остается выбрать $c = (c_2 + 1)m$, где m — максимальное число элементарных операций, требуемых для вычисления одного шага алгоритма 4.1. \square

Теорему 4.3 в некотором смысле нельзя усилить, т. е. существуют не леворекурсивные грамматики, при работе с которыми алгоритм 4.1 затрачивает экспоненциальное время, поскольку эти грамматики имеют c^n частичных левых разборов, совместимых с некоторыми цепочками длины n .

Пример 4.3. Пусть $G = (\{S\}, \{a, b\}, P, S)$, где P состоит из правил $S \rightarrow aSS | e$. Пусть $X(n)$ — число различных левых разборов цепочки a^n , а $Y(n)$ — число частичных левых разборов, совместимых с a^n . Функции $X(n)$ и $Y(n)$ определяются рекуррентными соотношениями

$$(4.1.2) \quad X(0) = 1 \\ X(n) = \sum_{i=0}^{n-1} X(i)X(n-1-i)$$

$$(4.1.3) \quad Y(0) = 2 \\ Y(n) = Y(n-1) + \sum_{i=0}^{n-1} X(i)Y(n-1-i)$$

Равенство (4.1.2) вытекает из того, что каждый вывод цепочки a^n при $n \geq 1$ начинается правилом $S \rightarrow aSS$. Остальные $n-1$ символов a можно так или иначе распределить между двумя нетерминалами S . В равенстве (4.1.3) $Y(n-1)$ отражает возможность того, что после первого шага $S \Rightarrow aSS$ второй нетерминал S больше не участвует в выводе, а суммирование отражает возможность того, что из первого S выводится a^i для некоторого i . Формула $Y(0) = 2$ вытекает из того, что пустой вывод и вывод $S \Rightarrow e$ совместимы с цепочкой e .

Из упр. 2.4.29 получаем

$$X(n) = \frac{1}{n+1} \binom{2n}{n}$$

так что $X(n) \geq 2^{n-1}$. Поэтому

$$Y(n) \geq Y(n-1) + \sum_{i=0}^{n-1} 2^{i-1} X(n-1-i)$$

откуда, разумеется, следует, что $Y(n) \geq 2^n$. \square

Этот пример выявляет главную проблему, возникающую при исходящем анализе с возвратами. Число шагов, необходимых для разбора с помощью алгоритма 4.1, может быть огромным. Известно несколько приемов, помогающих слегка ускорить этот алгоритм. Мы укажем некоторые из них.

(1) Можно упорядочить правила так, чтобы наиболее вероятные альтернативы испытывались первыми. Однако это не поможет в тех случаях, когда входная цепочка синтаксически неправильна, так что придется перебрать все возможности.

Определение. Для КС-грамматики $G = (N, \Sigma, P, S)$ определим функции

$$\text{FIRST}_k(\alpha) = \{x \mid \alpha \Rightarrow^* x \beta \text{ и } |x| = k \text{ или } \alpha \Rightarrow^* x \text{ и } |x| < k\}$$

Иначе говоря, множество $\text{FIRST}_k(\alpha)$ состоит из всех терминальных префиксов длины k (или меньше, если из α выводится терминальная цепочка длины, меньшей k) терминальных цепочек, выводимых из α .

(2) Можно заглядывать вперед на следующие k входных символов, чтобы определить, надо ли использовать данную альтернативу. Например, с этой целью можно для каждой альтернативы α заготовить множество $\text{FIRST}_k(\alpha)$. Если в нем не содержится ни один префикс оставшейся части входной цепочки, можно сразу отвергнуть α и опровергнуть следующую альтернативу. Этот прием очень полезен, когда данная входная цепочка принадлежит $L(G)$ и когда она не принадлежит $L(G)$. В гл. 5 мы увидим, что для некоторых классов грамматик с помощью заглядывания вперед можно полностью устранить необходимость возвратов.

(3) Можно добавить некоторую «бухгалтерию» (учет), позволяющую ускорить возвраты. Например, если мы знаем, что последние t примененных правил не имеют применимых следующих альтернатив, то в случае неудачи можем при возврате пропустить эти альтернативы, сразу вернувшись к тому месту, где есть применимая альтернатива.

(4) Можно ограничить количество допустимых возвратов. Методы такого рода излагаются в гл. 6.

Другая трудная проблема, связанная с возвратным анализом, — его слабые возможности в смысле локализации ошибок. Если входная цепочка синтаксически неправильна, то компилятор должен объявить, какие входные символы причастны к ошибке. Кроме того, когда найдена ошибка, компилятор должен исправить ее так, чтобы анализ мог продолжаться и обнаруживать другие ошибки, если они попадутся.

Если входная цепочка построена синтаксически неправильно, то алгоритм с возвратами просто объявит об ошибке, оставив входной указатель на первом входном символе. Чтобы получить более подробную информацию об ошибках, в грамматику можно вставить специальные правила, порождающие ошибки. Эти правила применяются для порождения цепочек, содержащих распространенные синтаксические ошибки, и благодаря им синтаксически неправильные цепочки становятся правильными с точки зрения новой грамматики. Появление в выходе номера такого правила служит сигналом, локализующим ошибку во входной цепочке. Однако с практической точки зрения алгоритмы синтаксического анализа, излагаемые в гл. 5, обладают большими способностями к обнаружению ошибок, чем возвратные алгоритмы с правилами, порождающими ошибки.

4.1.5. Восходящий разбор

Существует общий подход к разбору, в некотором смысле противоположный подходу, принятому при исходящем разборе. Алгоритм исходящего разбора можно представлять себе как построение дерева разбора методом проб и ошибок, которое начинается сверху в корне и продолжается вниз к листьям. В противоположность этому алгоритм восходящего разбора начинает с листьев (т. е. с самих входных символов) и пытается построить дерево разбора, восходя от листьев к корню.

Мы опишем восходящий разбор в виде алгоритма синтаксического анализа, который называют алгоритмом типа «перенос—свертка». Он представляет собой то же самое правило анализа, который перебирает все возможные обращенные правые выводы, совместимые с входной цепочкой. Один шаг алгоритма состоит в считывании цепочки, расположенной в верхней части магазина, чтобы выяснить, образуют ли верхние символы правую часть какого-нибудь правила. Если да, то производится свертка, заменяющая эти символы левой частью того самого правила. Если возможна более чем одна свертка, то эти свертки упорядочиваются произвольным образом и применяется первая из них.

Если свертка невозможна, то в магазин переносится следующий входной символ и процесс продолжается, как прежде. Всегда перед переносом делается попытка свертки. Если мы дошли до конца цепочки, а свертка все еще невозможна, то надо вернуться к последнему шагу, на котором была сделана свертка. Если здесь возможна другая свертка, надо испытать ее.

Рассмотрим грамматику с правилами $S \rightarrow AB$, $A \rightarrow ab$ и $B \rightarrow aba$. Пусть $ababa$ — входная цепочка. Сначала перенесем в магазин первый символ a . Так как свертка пока невозможна,

перенесем в магазин символ b . Затем цепочку ab , расположенную наверху магазина, заменим на A . Получили частичное дерево, показанное на рис. 4.6, а.

Так как A нельзя больше свернуть, перенесем в магазин a . Свертка опять невозможна, поэтому перенесем b . Тогда можно свернуть ab к A . Теперь получаем частичное дерево на рис. 4.6, б.

Переносим в магазин символ a и обнаруживаем, что свертка невозможна. Возвращаемся к последней позиции, в которой

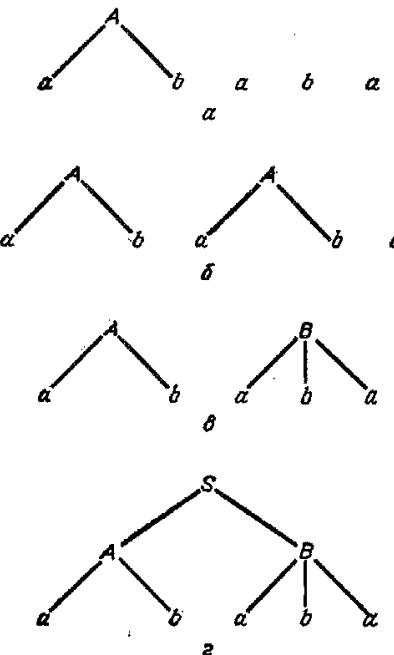


Рис. 4.6. Частичные деревья восходящего разбора.

была сделана свертка, а именно когда в магазине была цепочка Aab (здесь b — верхний символ) и мы заменили ab на A , т. е. частичное дерево было таким, как на рис. 4.6, а. Так как другая свертка здесь невозможна, сделаем перенос вместо свертки. В магазине окажется Aba . Тогда можно свернуть aba к B , получив при этом рис. 4.6, в. Далее заменим AB на S и получим завершенное дерево, показанное на рис. 4.6, г.

Этот метод можно рассматривать как процедуру прослеживания всех возможных последовательностей тактов недетерминированного правого анализатора для данной грамматики. Однако так же, как в случае исходящего разбора, надо избегать ситуаций, в которых число возможных последовательностей тактов бесконечно.

Одна такая ловушка оказывается тогда, когда грамматика содержит циклы, т. е. выводы вида $A \Rightarrow^+ A$ для некоторого нетерминала A . Число частичных деревьев может быть в этом случае бесконечным, так что грамматики с циклами мы исключим из рассмотрения. Трудности создаются также e -правилами, так как тогда можно проделать произвольное число сверток, при которых пустая цепочка «свертывается» к нетерминалу. Восходящий разбор можно расширить так, чтобы он охватывал грамматики с e -правилами, но пока для простоты мы предпочтем обходиться без e -правил.

Алгоритм 4.2. Восходящий разбор с возвратами.

Вход. КС-грамматика $G = (N, \Sigma, P, S)$ без циклов и e -правил (все ее правила занумерованы от 1 до p) и входная цепочка $w = a_1 a_2 \dots a_n$ ($n \geq 1$).

Выход. Один обращенный правый разбор, если он существует, и слово «ошибка» в противном случае.

Метод.

(1) Произвольным образом упорядочить правила.

(2) Алгоритм будет изложен в терминах 4-компонентных конфигураций, подобных тем, что использовались в алгоритме 4.1. В конфигурации (s, i, α, β)

- (а) s представляет состояние алгоритма,
- (б) i представляет текущую позицию входного указателя (предполагается, что $(n+1)$ -м входным символом служит правый концевой маркер $\$$),
- (в) α — содержимое магазина $L1$ (верх которого расположен справа)
- (г) β — содержимое магазина $L2$ (верх которого расположен слева).

Как и раньше, алгоритм может находиться в одном из трех состояний q , b или t . В магазине $L1$ будет храниться цепочка терминалов и нетерминалов, из которой выводится часть входной цепочки, расположенная слева от входного указателя. В магазине $L2$ будет храниться история переносов и сверток, необходимых для получения из входной цепочки содержимого магазина $L1$.

(3) Начальная конфигурация алгоритма — $(q, 1\$, e)$.

(4) Сам алгоритм работает так. Начинаем с попытки применить шаг 1.

Шаг 1. Попытка свертки

$$(q, i, \alpha\beta, \gamma) \vdash (q, i, \alpha A, j\gamma)$$

при условии, что $A \rightarrow \beta$ — правило из P с номером j и β — первая правая часть в линейном упорядочении, определенном в (1), которая является суффиксом цепочки $\alpha\beta$. Номер этого правила записывается в $L2$. Если шаг 1 применим, повторить его. В противном случае перейти к шагу 2.

Шаг 2. Перенос

$$(q, i, \alpha, \gamma) \vdash (q, i+1, \alpha a_i, s\gamma)$$

при условии, что $i \neq n+1$. Перейти к шагу 1.

Если $i = n+1$, перейти к шагу 3.

При выполнении шага 2 i -й входной символ переносится в верхнюю часть магазина $L1$, позиция входного указателя увеличивается и в магазин $L2$ записывается s , чтобы указать, что сделан перенос.

Шаг 3. Допускание

$$(q, n+1, \$S, \gamma) \vdash (t, n+1, \$S, \gamma)$$

Выдать $h(\gamma)$, где h — гомоморфизм, определенный равенствами $h(s) = e$, $h(j) = j$ для всех номеров правил, $h(\gamma)$ — обращенный правый разбор цепочки w . После этого остановиться.

Если шаг 3 неприменим, перейти к шагу 4.

Шаг 4. Переход в состояние возврата

$$(q, n+1, \alpha, \gamma) \vdash (b, n+1, \alpha, \gamma)$$

при условии, что $\alpha \neq \$S$. Перейти к шагу 5.

Шаг 5. Возврат

$$(a) \quad (b, i, \alpha A, j\gamma) \vdash (q, i, \alpha' B, k\gamma)$$

если $A \rightarrow \beta$ — правило из P с номером j , а следующим правилом в упорядочении (1), правая часть которого является суффиксом цепочки $\alpha\beta$, является правило $B \rightarrow \beta'$ с номером k . (Заметим, что $\alpha\beta = \alpha'\beta'$.) Перейти к шагу 1. (Здесь происходит возврат к предыдущей свертке и делается попытка свертки с помощью следующей альтернативы.)

$$(b) \quad (b, n+1, \alpha A, j\gamma) \vdash (b, n+1, \alpha\beta, \gamma)$$

если $A \rightarrow \beta$ — правило из P с номером j и для цепочки $\alpha\beta$ не остается никакой другой свертки. Перейти к шагу 5. (Если других сверток не существует, надо «взять назад» данную свертку и продолжать возврат, оставляя входной указатель на позиции $n+1$.)

$$(b) \quad (b, i, \alpha A, j\gamma) \vdash (q, i+1, \alpha\beta, s\gamma)$$

если $i \neq n+1$, $A \rightarrow \beta$ — правило из P с номером j и для $\alpha\beta$ не остается никакой другой свертки. Здесь символ $a = a_i$ перено-

сится в магазин $L1$, а символ s поступает в $L2$. Перейти к шагу 1.

(Мы вернулись к предыдущей свертке и, так как других сверток нет, попробуем сделать перенос.)

$$(g) \quad (b, i, aa, sv) \vdash (b, i-1, a, \gamma)$$

если наверху магазина $L2$ находится символ переноса s . (Здесь в позиции i исчерпаны все альтернативы и надо «взять назад» операцию переноса. Входной указатель сдвигается влево, терминальный символ устраняется из $L1$, а символ переноса s — из $L2$). Если этот шаг невыполним, объявить об ошибке. \square

Пример 4.4. Применим описанный алгоритм восходящего разбора к грамматике G с правилами

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

Если наверху магазина $L1$ появится $E + T$, то сначала попытаемся сделать свертку, используя $E \rightarrow E + T$, а потом — используя $E \rightarrow T$. Если же появится $T * F$, то сначала попробуем $T \rightarrow T * F$, а потом $T \rightarrow F$. Для входа $a * a$ восходящий алгоритм пройдет через конфигурации

$$\begin{aligned} (q, 1, \$, e) &\vdash (q, 2, \$a, s) \\ &\vdash (q, 2, \$F, 5s) \\ &\vdash (q, 2, \$T, 45s) \\ &\vdash (q, 2, \$E, 245s) \\ &\vdash (q, 3, \$E^*, s245s) \\ &\vdash (q, 4, \$E^* a, ss245s) \\ &\vdash (q, 4, \$E^* F, 5ss245s) \\ &\vdash (q, 4, \$E^* T, 45ss245s) \\ &\vdash (q, 4, \$E^* E, 245ss245s) \\ &\vdash (b, 4, \$E^* E, 245ss245s) \\ &\vdash (b, 4, \$E^* T, 45ss245s) \\ &\vdash (b, 4, \$E^* F, 5ss245s) \\ &\vdash (b, 4, \$E^* a, ss245s) \\ &\vdash (b, 3, \$E^*, s245s) \\ &\vdash (b, 2, \$E, 245s) \\ &\vdash (q, 3, \$T^*, s45s) \\ &\vdash (q, 4, \$T^* a, ss45s) \\ &\vdash (q, 4, \$T^* F, 5ss45s) \\ &\vdash (q, 4, \$T, 35ss45s) \\ &\vdash (q, 4, \$E, 235ss45s) \\ &\vdash (t, 4, \$E, 235ss45s) \quad \square \end{aligned}$$

Корректность алгоритма 4.2 можно доказать способом, аналогичным тому, которым было показано, что нисходящий алгоритм работает правильно. Мы дадим здесь лишь набросок доказательства, оставляя большую часть деталей в качестве упражнения.

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Будем называть π частичным правым разбором, совместимым с w , если существуют такие $\alpha \in (N \cup \Sigma)^*$ и префикс x цепочки w , что $\alpha \Rightarrow_{\pi R} x$.

Лемма 4.5. Пусть G — КС-грамматика без циклов и без е-правил. Тогда найдется такая константа c , что число частичных правых разборов, совместимых с входной цепочкой длины n , не превосходит c^n .

Доказательство. Упражнение. \square

Теорема 4.4. Алгоритм 4.2 правильно находит правый разбор цепочки w , если он существует, и сигнализирует об ошибке в противном случае.

Доказательство. По лемме 4.5 число частичных правых разборов, совместимых с входной цепочкой, конечно. В качестве упражнения предлагаем показать, что пока алгоритм 4.2 не найдет разбор входной цепочки, он перебирает в естественном порядке все частичные правые разборы. А именно, каждый частичный правый разбор кодируется последовательностью индексов правил и символов переноса (s), и алгоритм 4.2 рассматривает все такие кодирующие последовательности в лексикографическом порядке. Этот лексикографический порядок определяется порядком символов, в котором s находится на последнем месте, а упорядочение индексов правил задается шагом 1 алгоритма 4.2. Заметим, что не каждая последовательность таких символов будет кодом совместимого частичного правого разбора. \square

Так же, как для алгоритма 4.1, можно показать, что длины магазинных списков в конфигурациях алгоритма 4.2 линейно зависят от длины входной цепочки.

Теорема 4.5. Пусть для каждого символа из магазинных цепочек, участвующих в конфигурациях алгоритма 4.2, требуется только одна ячейка, и пусть число элементарных операций, необходимых для вычисления одного шага алгоритма 4.2, ограничено. Тогда для входной цепочки длины n алгоритму 4.2 требуется емкость памяти $c_1 n$ и время c_2^n , где c_1 и c_2 — некоторые константы.

Доказательство. Упражнение. \square

Известно несколько модификаций основного алгоритма восходящего разбора, ускоряющих его работу.

(1) Можно добавить „заглядывание вперед“ так, что если обнаруживается, что следующие k символов справа от входного указателя не могут следовать за A ни в какой правовыводимой цепочке, то в этот момент не надо делать свертку, соответствующую A -правилу.

(2) Можно попытаться упорядочить сыртки так, чтобы наиболее вероятные из них делались первыми.

(3) Можно добавить информацию, позволяющую определять, приведут ли некоторые свертки к успеху. Например, если первая свертка использует правило $A \rightarrow a_1 \dots a_k$, где a_1 — первый входной символ, и мы знаем, что нет такой цепочки $y \in \Sigma^*$, что $S \Rightarrow^* A y$, то эту свертку можно сразу исключить. Вообще мы хотим быть уверены в том, что если $\$a$ — содержимое магазина $L1$, то a — префикс правовыводимой цепочки. Хотя проверить это, вообще говоря, сложно, некоторые понятия (такие, как предшествование, обсуждаемое в гл. 5), помогают исключить многие цепочки a , которые могли бы появиться в $L1$.

(4) Можно модифицировать алгоритм так, чтобы ускорить возвраты. Например, можно записать информацию, позволяющую сразу восстановить предыдущую конфигурацию, в которой была сделана свертка.

Некоторые из этих соображений изучаются в упр. 4.1.12—4.1.14 и 4.1.25. Замечания по поводу обнаружения и исправления ошибок, относящиеся к нисходящему возвратному алгоритму, применимы и к восходящему алгоритму.

УПРАЖНЕНИЯ

4.1.1. Пусть G определяется правилами

$$\begin{aligned} S &\rightarrow AS | a \\ A &\rightarrow bSA | b \end{aligned}$$

Какую последовательность шагов сделает алгоритм 4.1, если альтернативы рассматриваются в том порядке, как они записаны, а входной цепочкой является

- (a) $ba?$
- (б) $baba?$

Какие будут последовательности шагов, если альтернативы рассматриваются в обратном порядке?

4.1.2. Пусть G состоит из правил

$$\begin{aligned} S &\rightarrow SA | A \\ A &\rightarrow aA | b \end{aligned}$$

Какую последовательность шагов сделает алгоритм 4.2, если более длинная альтернатива считается первой, а входной цепочкой является

- (a) $ab?$
- (б) $abab?$

Что будет, если первой считать более короткую альтернативу?

4.1.3. Покажите, что каждая КС-грамматика без циклов, не порождающая пустой цепочки, правопокрывается грамматикой, для которой работает алгоритм 4.2, но может не левопокрываться грамматикой, для которой работает алгоритм 4.1.

*4.1.4. Покажите, что рекуррентные соотношения

$$\begin{aligned} D(1) &= 1 \\ D(d) &= (D(d-1)^2) + 1 \end{aligned}$$

определяют функцию $D(d) = \lceil k^{d^k} \rceil$, где k — некоторое вещественное число, а $\lceil x \rceil$ — наименьшее целое число $\geq x$. Здесь $k = 1,502837\dots$

4.1.5. Дополните доказательство следствия 2 леммы 4.4.

4.1.6. Модифицируйте алгоритм 4.1 так, чтобы можно было отказаться от использования альтернативы, если из левовыводимой цепочки, получающейся после ее применения, нельзя вывести k очередных входных символов, где k — фиксированное число.

4.1.7. Модифицируйте алгоритм 4.1 так, чтобы он работал для произвольной грамматики, наложив ограничения на рост длин магазинов $L1$ и $L2$.

**4.1.8. Дайте необходимое и достаточное условие, которому должна удовлетворять входная грамматика для того, чтобы алгоритм 4.1 никогда не оказывался в состоянии возврата.

4.1.9. Докажите лемму 4.5.

4.1.10. Докажите теорему 4.5.

4.1.11. Модифицируйте алгоритм 4.2 так, чтобы он работал для произвольной грамматики, наложив ограничения на длины магазинов $L1$ и $L2$.

4.1.12. Модифицируйте алгоритм 4.2 так, чтобы он работал быстрее, введя в него проверку того, что частичный правый разбор вместе с частью входной цепочки, расположенной справа от указателя, не содержит ни одной из цепочек длины k , которые не могут быть частью правовыводимой цепочки.

4.1.13. Модифицируйте алгоритмы 4.1 и 4.2 так, чтобы они могли возвращаться к любой специально выделенной конфигу-

рации с помощью конечного числа разумно определенных элементарных операций.

****4.1.14.** Найдите необходимое и достаточное условие, которому должна удовлетворять грамматика, чтобы алгоритм 4.2 работал на ней без возвратов. То же для алгоритма, модифицированного, как в упр. 4.1.12.

4.1.15. Найдите грамматику без циклов и без *e*-правил, для которой алгоритм 4.2 тратит экспоненциальное время.

4.1.16. Улучшите границу, указанную в лемме 4.4, для грамматик, не содержащих *e*-правил.

4.1.17. Покажите, что если в грамматике G , не содержащей бесполезных символов, есть либо цикл, либо *e*-правило, то алгоритм 4.2 не будет останавливаться на цепочках, не принадлежащих языку $L(G)$.

Определение. Опишем в общих чертах язык программирования, на котором можно записывать недетерминированные алгоритмы. Назовем этот язык НДФ (недетерминированный Фортран), потому что он состоит из операторов, подобных операторам Фортрана, плюс оператор CHOICE (n_1, \dots, n_k), где $k \geq 2$ и n_1, \dots, n_k — номера операторов.

Чтобы определить смысл программ в языке НДФ, мы постулируем существование интерпретатора, способного выполнять любое конечное число программ в круговую (т. е. работать с каждой программой по очереди в течение фиксированного числа машинных операций). Смысл обычных операторов Фортрана предполагается известным. Однако если выполняется оператор CHOICE (n_1, \dots, n_k), то интерпретатор изготавливает k копий программы и всей ее области данных (т. е. текущих значений переменных). Управление передается оператору n_i в i -й копии программы ($1 \leq i \leq k$). Весь выход выдается на одно устройство печати и весь вход поступает с одного читающего устройства (лучше считать, что перед выполнением первого оператора CHOICE весь вход уже прочитан).

Пример 4.5. Следующая программа на языке НДФ печатает один или более раз сообщение НЕ ПРОСТОЕ, если ее входом служит не простое число, и не печатает ничего, если вход — простое число:

```
READ N
I = 1
```

C ВЗЯТЬ ЗНАЧЕНИЕ I БОЛЬШЕЕ 1
 1 I = I + 1
 CHOICE (1, 2)
 2 IF (I .EQ. N) STOP

C ОПРЕДЕЛИТЬ ЯВЛЯЕТСЯ ЛИ I
C ДЕЛИТЕЛЕМ N НЕ РАВНЫМ N
IF ((N/I) * I .NE. N) STOP
WRITE (НЕ ПРОСТОЕ)
STOP



***4.1.18.** Напишите программу на НДФ, печатающую все решения „проблемы восьми ферзей“ (выбрать на шахматной доске восемь полей так, чтобы никакие два из них не лежали на одной горизонтали, вертикали или диагонали).

***4.1.19.** Напишите программы на НДФ, моделирующие левый и правый анализаторы.

Было бы хорошо, если бы существовал алгоритм, определяющий для данной программы на НДФ, будет ли она на каком-нибудь входе работать бесконечно. К сожалению, эта проблема неразрешима ни для Фортрана, ни для какого другого достаточно мощного языка программирования. Однако на этот вопрос можно ответить, если предположить, что ветвлением в программе (связанным с операторами IF и GOTO, а не с CHOICE) управляют не значения переменных программы, а некий „демон“, который пытается заставить программу работать бесконечно. Назовем программу на НДФ останавливающейся, если ни для какого входа не существует последовательности ветвей и недетерминированных выборов, заставляющей какую-нибудь копию программы выполнить больше операторов, чем некоторое фиксированное число, причем это число зависит от числа входных перфокарт, предоставленных для данных. (Предполагается, что если программа пытается читать, а доступные для этого данные отсутствуют, она останавливается.)

***4.1.20.** Постройте алгоритм, определяющий, останавливается ли программа на НДФ при условии, что переменные циклов DO никогда не уменьшаются¹⁾.

***4.1.21.** Дайте алгоритм, который по останавливающейся программе на НДФ строит эквивалентную ей программу на Алголе. Под „эквивалентной программой“ мы должны понимать программу на Алголе, выдающую результаты в одном из тех порядков, в которых их может выдать программа на НДФ (определенный порядок для программы на НДФ не задан). Алгол предпочтительнее Фортрану из-за того, что здесь очень удобно применить рекурсию.

¹⁾ Так как для Фортрана без оператора DO любое нетривиальное свойство программ неразрешимо, возникает сомнение, можно ли неясное определение останавливающейся НДФ-программы истолковать так, чтобы это упражнение имело решение.— Прим. перев.

4.1.22. Постройте по КС-грамматике $G = (N, \Sigma, P, S)$ такую КС-грамматику G' , что $L(G') = \Sigma^*$ и если $S \pi \Rightarrow_G w$, то $S \pi \Rightarrow_{G'} w$.

По грамматике можно построить МП-преобразователь (верх его магазина будет расположен слева), который ведет себя как недетерминированный анализатор по левому участку для этой грамматики. Этот анализатор будет использовать в качестве магазинных символов нетерминалы, терминалы и специальные символы вида $[A, B]$, где A и B — нетерминалы.

Появляющиеся в магазине терминалы и нетерминалы служат „целями“, которые должны распознаваться сверху вниз. В символе $[A, B]$ нетерминал A — текущая цель, которую нужно распознать, а B — нетерминал, только что распознанный снизу вверх. По КС-грамматике $G = (N, \Sigma, P, S)$ построим МП-преобразователь $M = (\{q\}, \Sigma, N \times N \cup N \cup \Sigma, \Delta, \delta, q, S, \emptyset)$, который будет анализатором по левому участку для G . Здесь $\Delta = \{1, 2, \dots, p\}$ — множество номеров правил, а δ определяется так:

- (1) Допустим, что $A \rightarrow \alpha$ — правило из P с номером i .
 - (а) Если α имеет вид $B\beta$, где $B \in N$, то $\delta(q, e, [C, B])$ содержит $(q, \beta[C, A], i)$ для всех $C \in N$. Здесь предполагается, что левый участок B уже распознан снизу вверх, так что символы цепочки β становятся целями, которые нужно распознать сверху вниз. Как только цепочка β будет распознана, будет распознан и нетерминал A .
 - (б) Если α не начинается нетерминалом, то $\delta(q, e, C)$ содержит $(q, \alpha[C, A], i)$ для всех $C \in N$. Здесь нетерминал A будет распознан, как только будет распознана цепочка α .
- (2) $\delta(q, e, [A, A])$ содержит (q, e, e) для всех $A \in N$. Здесь вхождение цели A , которой мы занимались, уже распознано. Если это вхождение A не является левым участком, то $[A, A]$ устраняется из магазина и это означает, что данное вхождение A было той целью, которую мы искали.
- (3) $\delta(q, a, a) = \{(q, e, e)\}$ для всех $a \in \Sigma$. Здесь текущей целью служит терминальный символ, совпадающий с текущим входным символом. Будучи распознанной, эта цель устраивается.

МП-преобразователь M определяет перевод $\{(w, \pi) | w \in L(G)\}$ и π — разбор цепочки w по левому участку).

Пример 4.6. Рассмотрим КС-грамматику $G = (N, \Sigma, P, S)$ с правилами

- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow F \uparrow T$
- (4) $T \rightarrow F$

- (5) $F \rightarrow (E)$
- (6) $F \rightarrow a$

Недетерминированным анализатором по левому участку для грамматики G будет МП-преобразователь

$$M = (\{q\}, \Sigma, N \times N \cup N \cup \Sigma, \{1, 2, \dots, 6\}, \delta, q, E, \emptyset)$$

где δ следующим образом определяется для всех $A \in N$:

- (1) (а) $\delta(q, e, [A, E])$ содержит $(q, +T[A, E], 1)$,
 (б) $\delta(q, e, [A, T])$ содержит $(q, [A, E], 2)$,
 (в) $\delta(q, e, [A, F])$ содержит $(q, \uparrow T[A, T], 3)$ и $(q, [A, T], 4)$,
 (г) $\delta(q, e, A) = \{(q, (E)[A, F], 5), (q, a[A, F], 6)\}$.
- (2) $\delta(q, e, [A, A])$ содержит (q, e, e) .
- (3) $\delta(q, a, a) = \{(q, e, e)\}$ для всех $a \in \Sigma$.

Устроим разбор входной цепочки $a \uparrow a + a$ с помощью M . Дерево вывода этой цепочки показано на рис. 4.7. Так как

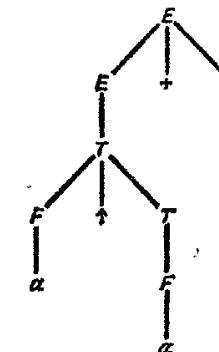


Рис. 4.7. Дерево вывода цепочки $a \uparrow a + a$,

МП-преобразователь M имеет только одно состояние, оно в дальнейшем не указывается. M начинает работу в конфигурации

$$(a \uparrow a + a, E, e)$$

Поскольку второе правило из (1г) применимо (как и первое), M может перейти в конфигурацию

$$(a \uparrow a + a, a[E, F], 6)$$

Здесь с помощью правила 6 порожден левый участок a . Символ a сравнивается затем с текущим входным символом, и это дает

$$(\uparrow a + a, [E, F], 6)$$

Теперь можно воспользоваться первым правилом из (1в) и получить

$$(\uparrow a + a, \uparrow T[E, T], 63)$$

Здесь левый участок правила $T \rightarrow F \uparrow T$ будет распознан, как только найдутся \uparrow и T . После этого можно перейти в конфигурации

$$\begin{aligned} (a+a, T[E, T], 63) &\vdash (a+a, a[T, F][E, T], 636) \\ &\vdash (+a, [T, F][E, T], 636) \\ &\vdash (+a, [T, T][E, T], 6364) \end{aligned}$$

В данный момент T служит текущей целью и найдено вхождение T , которое не является левым участком. Поэтому, применяя (2), можно стереть эту цель и получить

$$(+a, [E, T], 6364)$$

Продолжая в том же духе, получаем последовательность конфигураций

$$\begin{aligned} (+a, [E, E], 63642) &\vdash (+a, +T[E, E], 636421) \\ &\vdash (a, T[E, E], 636421) \\ &\vdash (a, a[T, F][E, E], 6364216) \\ &\vdash (e, [T, F][E, E], 6364216) \\ &\vdash (e, [T, T][E, E], 63642164) \\ &\vdash (e, [E, E], 63642164) \\ &\vdash (e, e, 63642164) \quad \square \end{aligned}$$

***4.1.23.** Покажите, что описанная выше конструкция дает для КС-грамматики G недетерминированный анализатор по левому участку.

4.1.24. Постройте алгоритм разбора по левому участку с возвратами.

Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, не содержащая правил с правыми частями длины 0 и 1. (Такая грамматика существует для каждого КС-языка $L \subseteq \{\omega | \omega \in \Sigma^* \text{ и } |\omega| \geq 2\}$). Для G можно построить недетерминированный правый анализатор типа „перенос—свертка“, у которого каждый символ магазина является парой вида (X, Q) , где $X \in N \cup \Sigma \cup \{\$\}$ ($\$$ — правый концевой маркер магазина), а Q — множество правил из P , причем для каждой правой части правила указаны все возможные префиксы, которые могли быть распознаны к данному моменту разбора (это делается с помощью точек, поставленных между некоторыми символами правых частей). Перед символом X_i в правиле $A \rightarrow X_1 X_2 \dots X_n$ точка ставится тогда и только тогда, когда $X_1 \dots X_{i-1}$ — суффикс цепочки символов грамматики, находящейся в магазине.

Шаг переноса можно сделать, если текущий входной символ служит продолжением некоторого правила. В частности, его можно сделать всегда, когда $A \rightarrow \alpha$ принадлежит P , а текущий входной символ принадлежит $\text{FIRST}_1(\alpha)$.

Шаг свертки можно сделать, если достигнут конец правой части правила. Допустим, что таким правилом является $A \rightarrow \alpha$. Чтобы сделать свертку, удалим из верхней части магазина $|\alpha|$ символов. Если теперь верхним символом магазина стал (X, Q) , пишем в магазин (A, Q') , где Q' вычисляется по Q в предположении, что A распознан, т. е. Q' образуется из Q переносом через A всех точек, стоящих непосредственно слева от A , и добавлением точек на левых концах правых частей, если их там еще не было.

Пример 4.7. Рассмотрим грамматику $S \rightarrow Sc|ab$ и входную цепочку abc . Вначале в магазине анализатора находится символ $(\$, Q_0)$, где $Q_0 = S \rightarrow \cdot Sc \cdot ab$. Затем можно перенести первый входной символ и записать в магазин (a, Q_1) , где $Q_1 = S \rightarrow \cdot Sc \cdot a \cdot b$. Здесь либо можно находиться в начале правил $S \rightarrow Sc|ab$, либо можно считать, что прочитан первый символ a правила $S \rightarrow ab$. Пересядя следующий входной символ b , запишем в магазин (b, Q_2) , где $Q_2 = S \rightarrow \cdot Sc \cdot ab$. Затем можно сделать свертку, используя правило $S \rightarrow ab$. Магазин будет теперь содержать $(\$, Q_0)(S, Q_3)$, где $Q_3 = S \rightarrow \cdot Sc \cdot ab$. \square

Домёлки предложил реализовать этот алгоритм с помощью двоичной матрицы M , представляющей правила, и двоичного вектора V , представляющего возможные позиции в каждом правиле. В описанном выше алгоритме можно вместо Q брать вектор V . Каждый новый вектор в магазине легко вычисляется по M и текущему значению вектора V с помощью простых по-разрядных двоичных операций.

4.1.25. Используйте алгоритм Домёлки для определения возможных сверток в алгоритме 4.2.

Замечания по литературе

Многие ранние компиляторы компиляторов и синтаксически управляемые компиляторы использовали недетерминированные алгоритмы разбора. Варианты методов нисходящего разбора с возвратами использовались в компиляторе компиляторов Брукера и Морриса [Розен, 1967г] и в системе построения компиляторов МЕТА [Шорре, 1964]. В системе символьного программирования COGENT недетерминированный нисходящий анализатор моделируется параллельным выполнением всех допустимых последовательностей тактов [Рейнольдс, 1965]. Методы нисходящего разбора с возвратами применялись также для синтаксического анализа естественных языков [Куно и Эттингер, 1962].

Одним из самых ранних опубликованных алгоритмов разбора был алгоритм Айронаса [1961], работавший по существу методом разбора по левому участку. Обзоры ранних методов разбора сделаны Флойдом [1964б], Читэмом и Сэттли [1964] и Гриффитсом и Петриком [1965].

Ангер [1968] описывает нисходящий алгоритм, в котором для уменьшения возвратов употребляются первые и последние символы, выводимые из нетерминала. Недетерминированные алгоритмы рассматриваются в работе Флойда [1967г].

Одна реализация алгоритма Домёлки описана Хекстом и Робертсон [1970]¹⁾. В обзорной статье Коэна и Готлиба [1970] описывается использование в алгоритмах разбора (с возвратами и без них) представлений КС-грамматик с помощью списочных структур.

4.2. ТАБЛИЧНЫЕ МЕТОДЫ СИНТАКСИЧЕСКОГО АНАЛИЗА

Мы изучим два метода синтаксического анализа, работающие для всех контекстно-свободных грамматик: алгоритм Кока—Янгера—Касами и алгоритм Эрли. Каждый из них требует времени n^3 и емкости n^2 , но последнему алгоритму для однозначных грамматик достаточно времени n^2 . Кроме того, можно добиться, чтобы алгоритм Эрли тратил линейные времена и емкость для большинства грамматик, которые можно анализировать за линейное время методами, излагаемыми в последующих главах.

4.2.1. Алгоритм Кока — Янгера — Касами

В последнем разделе мы обнаружили, что нисходящие и восходящие алгоритмы с возвратами могут затрачивать на разбор экспоненциальное время. Здесь мы изложим метод, для которого гарантируется, что он выполнит ту же работу для произвольной грамматики за время, пропорциональное кубу длины входной цепочки. Это по существу метод „динамического программирования“, и мы включили его сюда из-за его простоты. Сомнительно, однако, что он найдет практическое применение, поскольку

(1) время n^3 слишком велико, чтобы его можно было позволить потратить на разбор,

(2) используемая емкость памяти пропорциональна квадрату длины входа,

(3) метод разд. 4.2.2 (алгоритм Эрли) во всех отношениях так же хорош, как этот, а для многих грамматик даже лучше.

Метод работает следующим образом. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика без e -правил в нормальной форме Хомского. Простое обобщение алгоритма работает и для грамматик, не находящихся в этой нормальной форме; мы оставим это обобщение читателю. Так как КС-грамматика без циклов лево- и правопотокивается КС-грамматикой в нормальной форме Хомского, то это обобщение не так уж важно.

Пусть $w = a_1 a_2 \dots a_n$ — входная цепочка, которую нужно разобрать согласно грамматике G . Предполагается, что $a_i \in \Sigma$ для $1 \leq i \leq n$. Суть алгоритма состоит в построении треугольной таблицы разбора T , элементы которой обозначим t_{ij} , где $1 \leq i \leq n$

¹⁾ Уместно упомянуть работы самого Домёлки [1964, 1965]. — Прим. перев.

и $1 \leq j \leq n - i + 1$. Значениями переменных t_{ij} будут подмножества множества N . Нетерминал A будет принадлежать t_{ij} тогда и только тогда, когда $A \Rightarrow^+ a_i a_{i+1} \dots a_{i+j-1}$, т. е. когда из A выводятся j входных символов, начиная с позиции i . В частности, входная цепочка w принадлежит $L(G)$ тогда и только тогда, когда $S \in t_{1n}$.

Таким образом, чтобы выяснить, принадлежит ли w языку $L(G)$, вычислим для w таблицу разбора T и посмотрим, принадлежит ли S ее элементу t_{1n} . Затем, если нужен один (или все) разбор цепочки w , его можно построить с помощью таблицы разбора. Для этой цели можно использовать алгоритм 4.4.

Сначала мы приведем алгоритм, вычисляющий таблицу разбора, а затем алгоритм, строящий разборы по этой таблице.

Алгоритм 4.3. Алгоритм разбора Кока—Янгера—Касами.

Вход. КС-грамматика $G = (N, \Sigma, P, S)$ в нормальной форме Хомского без e -правил и входная цепочка $w = a_1 a_2 \dots a_n \in \Sigma^*$.

Выход. Таблица разбора T для цепочки w , такая, что $A \in t_{ij}$ тогда и только тогда, когда $A \Rightarrow^+ a_i a_{i+1} \dots a_{i+j-1}$.

Метод.

(1) Положить $t_{ii} = \{A \mid A \rightarrow a_i\}$ принадлежит P для каждого i . После этого шага из $A \in t_{ii}$ следует, очевидно, $A \Rightarrow^+ a_i$.

(2) Допустим, что уже вычислены $t_{ij'}$ для всех $1 \leq i \leq n$ и всех $1 \leq j' < j$. Положить

$$t_{ij} = \{A \mid \text{для некоторого } 1 \leq k < j \text{ правило } A \rightarrow BC \text{ принадлежит } P, B \in t_{ik} \text{ и } C \in t_{i+k, j-k}\}$$

Так как $1 \leq k < j$, то k и $j - k$ меньше j . Таким образом, t_{ik} и $t_{i+k, j-k}$ вычисляются раньше, чем t_{ij} . После этого шага из $A \in t_{ij}$ следует

$$A \Rightarrow BC \Rightarrow^+ a_i \dots a_{i+k-1} C \Rightarrow^+ a_i \dots a_{i+k-1} a_{i+k} \dots a_{i+j-1}$$

(3) Повторять шаг (2) до тех пор, пока не станут известны t_{ij} для всех $1 \leq i \leq n$ и $1 \leq j \leq n - i + 1$. \square

Пример 4.8. Рассмотрим грамматику G в нормальной форме Хомского с правилами

$$\begin{aligned} S &\rightarrow AA \mid AS \mid b \\ A &\rightarrow SA \mid AS \mid a \end{aligned}$$

¹⁾ Заметим, что мы не обсуждаем в деталях, как это сделать. Очевидно, что соответствующее вычисление можно выполнить на вычислительной машине. Когда речь пойдет о временной сложности алгоритма 4.3, будут даны детали этого шага, обеспечивающие его эффективное выполнение.

Пусть $abaab$ — входная цепочка. Таблица разбора T , получающаяся в результате работы алгоритма 4.3, показана на рис. 4.8. После шага (1) $t_{11} = \{A\}$, так как $A \rightarrow a$ принадлежит P и $a_1 = a$. На шаге (2) в t_{32} добавляем S , так как $S \rightarrow AA$ принадлежит P и A принадлежит t_{31} и t_{41} . Заметим вообще, что, как видно

		A, S		
	4	A, S	A, S	
	3	A, S	S	A, S
	2	A, S	A	S
$j \uparrow$	1	A	S	A
$i \uparrow$	1	2	3	4
		5		

Рис. 4.8. Таблица разбора T .

из рисунка, t_{ij} для $i > 1$ можно вычислить, обследовав нетерминалы в следующих парах элементов таблицы разбора:

$$(t_{11}, t_{i+1, j-1}), (t_{12}, t_{i+2, j-2}), \dots, (t_{i, j-1}, t_{i+j-1, 1})$$

Тогда, если $B \in t_{ik}$ и $C \in t_{i+k, j-k}$ для некоторого $1 \leq k < j$ и $A \rightarrow BC \in P$, добавляем A к t_{ij} . Это значит, что мы одновременно движемся вверх по i -му столбцу и вниз по диагонали, спускающейся вправо от ячейки t_{ij} , обозревая нетерминалы, расположенные в проходимых таким образом парах ячеек.

Так как $S \in t_{15}$, то $abaab \in L(G)$. \square

Теорема 4.6. Если алгоритм 4.3 применяется к грамматике G в нормальной форме Хомского и входной цепочке $a_1 \dots a_n$, то по окончании его работы A принадлежит t_{ij} тогда и только тогда, когда $A \Rightarrow^+ a_i \dots a_{i+j-1}$.

Доказательство. Доказательство проводится индукцией по j ; мы оставляем его в качестве упражнения. Наиболее трудный шаг содержится в доказательстве достаточности условия; здесь нужно заметить, что если $j > 1$ и $A \Rightarrow^+ a_i \dots a_{i+j-1}$, то найдутся такие нетерминалы B и C и число k , что $A \rightarrow BC \in P$, $B \Rightarrow^+ a_i \dots a_{i+k-1}$ и $C \Rightarrow^+ a_{i+k} \dots a_{i+j-1}$. \square

Покажем, что алгоритм 4.3 можно выполнить на машине с произвольным доступом к памяти за n^3 подходящим образом определенных элементарных операций. Предположим, что в нашем распоряжении несколько целых переменных, одна из кото-

рых — длина n входной цепочки. Элементарной операцией будем считать каждую из следующих:

(1) Присваивание переменной значения другой переменной или константы, а также суммы или разности значений двух переменных или констант,

(2) проверка равенства значений двух переменных,

(3) обследование и/или изменение значения переменной t_{ij} , если i и j — текущие значения двух целых переменных или констант,

(4) обследование i -го входного символа a_i , если i — значение некоторой переменной.

Заметим, что операция (3) имеет ограниченный объем, если грамматика заранее известна. Если грамматика становится более сложной, то объем памяти, необходимой для хранения переменной t_{ij} , и время, необходимое для ее обследования, возрастают, если рассматривать разумные шаги более элементарной природы. Однако здесь мы интересуемся только зависимостью времени от длины входной цепочки. Читателю предоставляется самому определить более элементарные шаги, которыми можно заменить (3), и найти функциональную зависимость времени их вычисления от числа нетерминалов и правил грамматики.

Соглашение. Запись $f(n) = 0(g(n))$ означает, что существует такая константа k , что $f(n) \leq kg(n)$ для всех $n \geq 1$. Таким образом, когда мы говорим, что алгоритм 4.3 выполняет свою работу за время $O(n^3)$, мы подразумеваем, что существует такая константа k , что для входной цепочки длины n тратится не более kn^3 элементарных операций.

Теорема 4.7. Алгоритму 4.3 для вычисления всех t_{ij} требуется $O(n^3)$ элементарных операций указанного выше типа.

Доказательство. Чтобы вычислить t_{11} для всех i , надо положить $i=1$ (операция (1)), затем несколько раз полагать $t_{11} = \{A \mid A \rightarrow a_i \in P\}$ (операции (3) и (4)), проверять, справедливо ли равенство $i=n$ (операция (2)), и, если нет, увеличивать i на 1 (операция (1)). Общее число выполняемых при этом операций равно $O(n)$.

Далее, чтобы вычислить t_{ij} , нужно выполнить следующие шаги:

(1) Положить $j=1$.

(2) Проверить, справедливо ли равенство $j=n$. Если нет, увеличить j на 1 и выполнить $\text{line}(j)$ — процедуру, которая будет определена ниже.

(3) Повторять шаг (2), пока не будет $j=n$.

Не считая операций, требующихся для $\text{line}(j)$, эта подпрограмма тратит $2n - 2$ элементарных операций. Общее число элементарных операций, требующихся алгоритму 4.3, равно, таким образом, $0(n) + \sum_{j=2}^n l(j)$, где $l(j)$ — число элементарных операций, используемых процедурой $\text{line}(j)$. Мы покажем, что $l(j) = O(n^2)$, и потому общее число операций равно $O(n^3)$.

Процедура $\text{line}(j)$ вычисляет все элементы t_{ij} , для которых $1 \leq i < n - j$. Она включает процедуру вычисления t_{ij} , описанную в примере 4.8, и определяется так (предполагаем, что вначале все t_{ij} имеют значение \emptyset):

- (1) Положить $i = 1$ и $j' = n - j + 1$.
- (2) Положить $k = 1$.
- (3) Положить $k' = i + k$ и $j'' = j - k$.
- (4) Обследовать t_{ik} и $t_{k'j''}$. Положить

$$t_{ij} = t_{ij} \cup \{A \mid A \rightarrow BC \in P, B \in t_{ik}, C \in t_{k'j''}\}$$

- (5) Увеличить k на 1.
- (6) Если $k = j$, перейти к шагу (7). Иначе перейти к шагу (3).
- (7) Если $i = j'$, остановиться. Иначе сделать шаг (8).
- (8) Увеличить i на 1 и перейти к шагу (2).

Заметим, что написанная программа содержит внутренний цикл (3)–(6) и внешний цикл (2)–(8). Внутренний цикл выполняется $j - 1$ раз (для значений переменной k от 1 до $j - 1$) всегда, когда программа попадает в него. В конце цикла t_{ij} принимает значение, предписываемое алгоритмом 4.3. Сам цикл состоит из семи элементарных операций, так что каждый раз, когда программа попадает в него, она затрачивает $O(j)$ элементарных операций.

Во внешний цикл программа входит $n - j + 1$ раз и каждый раз при этом тратится $O(j)$ элементарных операций. Так как $j \leq n$, каждое вычисление процедуры $\text{line}(j)$ требует $O(n^2)$ операций.

Так как $\text{line}(j)$ вычисляется n раз, то общее число элементарных операций, выполняемых алгоритмом, равно, таким образом, $O(n^3)$. \square

Теперь опишем, как по таблице разбора найти левый разбор. Метод излагается в виде алгоритма 4.4.

Алгоритм 4.4. Нахождение левого разбора по таблице разбора.

Вход. КС-грамматика $G = (N, \Sigma, P, S)$ в нормальной форме Хомского с правилами, занумерованными от 1 до p , входная

цепочка $w = a_1 a_2 \dots a_n$ и таблица разбора T , построенная для цепочки w алгоритмом 4.3.

Выход. Левый разбор цепочки w или сигнал „ошибка“.

Метод. Опишем рекурсивную процедуру $\text{gen}(i, j, A)$, порождающую левый разбор, соответствующий выводу $A \Rightarrow^+ a_i a_{i+1} \dots a_{i+j-1}$:

(1) Если $j = 1$ и $A \rightarrow a_i$ — правило из P с номером m , выдать номер m .

(2) Пусть $j > 1$ и k — наименьшее из целых чисел от 1 до $j - 1$, для которых существуют $B \in t_{ik}$, $C \in t_{i+k, j-k}$ и правило $A \rightarrow BC$ из P с номером, скажем, m . (Может оказаться несколько таких правил. Произвольно выберем одно из них, например с наименьшим m .) Тогда выдать номер m и выполнить $\text{gen}(i, k, B)$, а затем $\text{gen}(i+k, j-k, C)$.

Алгоритм 4.4 заключается в выполнении $\text{gen}(1, n, S)$ при условии, что $S \in t_{1n}$. Если $S \notin t_{1n}$, выдать сигнал „ошибка“. \square

Расширим понятие элементарной операции, включив в него запись номера правила. Тогда можно доказать следующий результат.

Теорема 4.8. Для входной цепочки $a_1 \dots a_n$ алгоритм 4.4 окончает работу, выдав некоторый левый разбор этой цепочки, если он существует. Число элементарных шагов, затрачиваемых алгоритмом 4.4, равно $O(n^2)$.

Доказательство. Индукция по порядку вызовов процедуры gen показывает, что если вызывается $\text{gen}(i, j, A)$, то $A \in t_{ij}$. Отсюда легко вывести, что алгоритм 4.4 дает левый разбор.

Чтобы показать, что алгоритм 4.4 заканчивает работу за время $O(n^2)$, докажем индукцией по j , что для всех j вызов $\text{gen}(i, j, A)$ расходует не более $c_1 j^2$ шагов, где c_1 — некоторая константа. Базис, $j = 1$, тривиален, так как шаг (1) алгоритма 4.4 использует одну элементарную операцию.

Для доказательства шага индукции заметим, что вызов $\text{gen}(i, j, A)$ для $j > 1$ приводит к выполнению шага (2). Читатель может проверить, что найдется такая константа c_2 , что, не считая вызовов, шаг (2) расходует не более $c_2 j$ элементарных операций. Если вызываются $\text{gen}(i, k, B)$ и $\text{gen}(i+k, j-k, C)$, то по предположению индукции на вызов $\text{gen}(i, j, A)$ тратится не более $c_1 k^2 + c_1 (j-k)^2 + c_2 j$ шагов. Приведем это выражение к виду $c_1 (j^2 + 2k^2 - 2kj) + c_2 j$. Так как $1 \leq k < j$ и $j \geq 2$, то $2k^2 - 2kj \leq 2 - 2j \leq -j$. Таким образом, если в предположении индукции взять $c_1 = c_2$, получим $c_1 k^2 + c_1 (j-k)^2 + c_2 j \leq c_1 j^2$. Поскольку мы вправе это сделать, теорема доказана. \square

Пример 4.9. Пусть G — грамматика с правилами

- (1) $S \rightarrow AA$
- (2) $S \rightarrow AS$
- (3) $S \rightarrow b$
- (4) $A \rightarrow SA$
- (5) $A \rightarrow AS$
- (6) $A \rightarrow a$

Пусть $w = abaab$ — входная цепочка. Таблица разбора цепочки приведена в примере 4.8.

Так как $S \in t_{15}$, то $w \in L(G)$. Чтобы найти левый разбор цепочки $abaab$, вызовем процедуру $\text{gen}(1, 5, S)$. При этом обнаружится, что A принадлежит t_{11} и t_{21} , и существует правило $S \rightarrow AA$. Тогда будет выдано число 1 (номер правила $S \rightarrow AA$) и затем вызваны $\text{gen}(1, 1, A)$ и $\text{gen}(2, 4, A)$. Вызов $\text{gen}(1, 1, A)$ даст правило номер 6. Так как $S \in t_{21}$, $A \in t_{33}$ и $A \rightarrow SA$ — четвертое правило, то $\text{gen}(2, 4, A)$ выдаст 4 и вызовет $\text{gen}(2, 1, S)$, а затем $\text{gen}(3, 3, A)$.

Продолжая в том же духе, получим левый разбор 164356263.

Заметим, что G — неоднозначная грамматика; в самом деле, цепочка $abaab$ имеет более одного левого разбора. В общем случае нельзя по таблице разбора получить все разборы входной цепочки за менее чем экспоненциальное время, так как у нее может быть экспоненциальное число разборов. \square

Заметим, что алгоритм 4.4 можно сделать более быстрым, если при построении таблицы разбора помещать указатели в те элементы, которые приводят к появлению новых элементов (см. упр. 4.2.21).

4.2.2. Алгоритм Эрли

В этом разделе мы изложим метод синтаксического анализа, позволяющий для произвольной КС-грамматики разобрать входную цепочку за время $O(n^3)$, использовав при этом емкость памяти $O(n^2)$, где n — длина входной цепочки. Кроме того, если грамматика однозначная, то время квадратичное, и для большинства грамматик языков программирования алгоритм можно модифицировать так, чтобы время и емкость стали линейными функциями от длины входной цепочки (упр. 4.2.18). Сначала мы неформально опишем основной алгоритм, а потом покажем, что вычисление можно организовать так, что получатся указанные выше границы сложности.

Основная идея алгоритма состоит в следующем. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика и $w = a_1 a_2 \dots a_n$ — входная цепочка из Σ^* . Объект вида $[A \rightarrow X_1 X_2 \dots X_k \cdot X_{k+1} \dots X_m, i]$ на-

зовем *ситуацией*, относящейся к цепочке w , если $A \rightarrow X_1 \dots X_m$ — правило из P и $0 \leq i \leq n$. Точка между X_k и X_{k+1} является метасимволом, не принадлежащим ни N , ни Σ . Число k может быть любым целым числом от нуля (в этом случае точка — первый символ) до m (в этом случае она — последний символ)¹⁾.

Для каждого $0 \leq j \leq n$ мы построим такой список ситуаций I_j , что $[A \rightarrow \alpha \cdot \beta, i]$ принадлежит I_j для $0 \leq i \leq j$ тогда и только тогда, когда для некоторых γ и δ существуют выводы $S \Rightarrow^* \gamma A \delta$, $\gamma \Rightarrow^* a_1 \dots a_i$ и $\alpha \Rightarrow^* a_{i+1} \dots a_j$. Таким образом, между второй компонентой ситуации и номером списка, в котором она появляется, заключена часть входной цепочки, выводимая из α . Другие условия, налагаемые на ситуацию, просто гарантируют возможность применения правила $A \rightarrow \alpha \beta$ в выводе некоторой входной цепочки, совпадающей с w до позиции j .

Последовательность списков I_0, I_1, \dots, I_n будем называть *списком разбора* для входной цепочки w . Заметим, что w принадлежит $L(G)$ тогда и только тогда, когда в I_n есть ситуация вида $[S \rightarrow \alpha \cdot, 0]$.

Опишем алгоритм, который по произвольной данной грамматике порождает список разбора любой входной цепочки.

Алгоритм 4.5. Алгоритм Эрли.

Вход. КС-грамматика $G = (N, \Sigma, P, S)$ и входная цепочка $w = a_1 a_2 \dots a_n$ из Σ^* .

Выход. Список разбора I_0, I_1, \dots, I_n для цепочки w .

Метод. Сначала строим I_0 :

(1) Если $S \rightarrow \alpha$ — правило из P , включить в $[S \rightarrow \alpha, 0]$ в I_0 .

Далее выполнять шаги (2) и (3) до тех пор, пока в I_0 можно включать новые ситуации.

(2) Если $[B \rightarrow \gamma \cdot, 0]$ принадлежит I_0 ²⁾, включить в I_0 ситуацию $[A \rightarrow \alpha B \cdot \beta, 0]$ для всех $[A \rightarrow \alpha \cdot B \beta, 0]$, уже принадлежащих I_0 .

(3) Допустим, что $[A \rightarrow \alpha \cdot B \beta, 0]$ принадлежит I_0 . Для каждого правила из P вида $B \rightarrow \gamma$ включить в I_0 ситуацию $[B \rightarrow \gamma \cdot, 0]$ (если она еще не там).

После того как построены I_0, I_1, \dots, I_{j-1} , строится I_j :

(4) Для каждой ситуации $[B \rightarrow \alpha \cdot \beta, i]$ из I_{j-1} , для которой $a = a_j$, включить в I_j ситуацию $[B \rightarrow \alpha \cdot \beta, i]$.

Далее выполнять шаги (5) и (6), пока в I_j можно включать новые ситуации.

(5) Пусть $[A \rightarrow \alpha \cdot, i]$ принадлежит I_j . Искать в I_i ситуации вида $[B \rightarrow \alpha \cdot \beta, k]$. Для каждой из них включить в I_j ситуацию $[B \rightarrow \alpha \cdot \beta, k]$.

¹⁾ Если правило имеет вид $A \rightarrow e$, то ситуация будет такой: $[A \rightarrow \cdot, i]$.

²⁾ Заметим, что γ может быть e . Так начинается применение правила (2).

(6) Пусть $[A \rightarrow \alpha \cdot B\beta, i]$ принадлежит I_j . Для каждого $B \rightarrow \gamma$ из P включить в I_j ситуацию $[B \rightarrow \cdot \gamma, j]$.

Заметим, что рассмотрение ситуации, в которой справа от точки стоит терминал, на шагах (2), (3), (5) и (6) не дает новых ситуаций.

Итак, алгоритм состоит в построении I_j для $0 \leq j \leq n$. \square

Пример 4.10. Рассмотрим грамматику G с правилами

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

и входную цепочку $(a + a) * a$. На шаге (1) включаем в I_0 ситуации $[E \rightarrow \cdot T + E, 0]$ и $[E \rightarrow \cdot T, 0]$. Рассмотрение этих ситуаций приводит к включению в I_0 посредством шага (3) ситуаций $[T \rightarrow \cdot F * T, 0]$ и $[T \rightarrow \cdot F, 0]$. Продолжая этот процесс, добавляем $[F \rightarrow \cdot (E), 0]$ и $[F \rightarrow \cdot a, 0]$. Другие ситуации уже нельзя включить в I_0 .

Теперь построим I_1 . По правилу (4) включаем $[F \rightarrow (\cdot E), 0]$, так как $a_1 = \cdot$. Тогда правило (6) позволяет включить $[E \rightarrow \cdot T + E, 1]$, $[E \rightarrow \cdot T, 1]$, $[T \rightarrow \cdot F * T, 1]$, $[T \rightarrow \cdot F, 1]$, $[F \rightarrow \cdot (E), 1]$ и $[F \rightarrow \cdot a, 1]$. Теперь в I_1 уже нельзя включить новые ситуации.

Чтобы построить I_2 , заметим, что $a_2 = a$ и по правилу (4) ситуацию $[F \rightarrow a \cdot, 1]$ нужно включить в I_2 . Затем по правилу (5) рассмотрим эту ситуацию, перейдем к списку I_1 и поищем в нем ситуации, в которых F стоит непосредственно справа от точки. Две такие ситуации найдутся, и мы включим $[T \rightarrow F * T, 1]$ и $[T \rightarrow F \cdot, 1]$ в I_2 . Рассмотрение первой из них не дает ничего, а вторая заставляет обратиться к I_1 в поисках ситуаций, содержащих $\cdot T$. Еще две ситуации $[E \rightarrow T \cdot + E, 1]$ и $[E \rightarrow T \cdot, 1]$ включаются в I_2 . Снова первая не дает ничего, а вторая приводит к включению $[F \rightarrow (E \cdot), 0]$ в I_2 . Теперь никакие ситуации нельзя включить в I_2 , так что список завершен.

Содержимое всех списков приведено на рис. 4.9.

Так как ситуация $[E \rightarrow T \cdot, 0]$ включена в последний список, входная цепочка $(a + a) * a$ принадлежит $L(G)$. \square

При анализе алгоритма Эрли мы будем действовать по следующему плану. Сначала докажем корректность упомянутой ранее неформальной интерпретации ситуаций. Затем покажем, что если грамматика G однозначная, то при разумном понятии элементарной операции время выполнения алгоритма квадратично зависит от длины входной цепочки. Наконец, покажем, как по списку разбора построить разбор и фактически как это сделать за квадратичное время.

I_0	I_1	I_2
$[E \rightarrow \cdot T + E, 0]$	$[F \rightarrow (\cdot E), 0]$	$[F \rightarrow a \cdot, 1]$
$[E \rightarrow \cdot T, 0]$	$[E \rightarrow \cdot T + E, 1]$	$[T \rightarrow F * T, 1]$
$[T \rightarrow \cdot F * T, 0]$	$[E \rightarrow \cdot T, 1]$	$[T \rightarrow F \cdot, 1]$
$[T \rightarrow \cdot F, 0]$	$[T \rightarrow \cdot F * T, 1]$	$[E \rightarrow T \cdot + E, 1]$
$[F \rightarrow \cdot (E), 0]$	$[T \rightarrow \cdot F, 1]$	$[E \rightarrow T \cdot, 1]$
$[F \rightarrow \cdot a, 0]$	$[F \rightarrow \cdot (E), 1]$	$[F \rightarrow (E \cdot), 0]$
	$[F \rightarrow \cdot a, 1]$	
I_3	I_4	I_5
	$[F \rightarrow a \cdot, 3]$	$[F \rightarrow (E \cdot), 0]$
	$[E \rightarrow T \cdot + E, 3]$	$[T \rightarrow F * T, 0]$
	$[E \rightarrow \cdot T, 3]$	$[T \rightarrow F \cdot, 0]$
	$[T \rightarrow \cdot F * T, 3]$	$[E \rightarrow T \cdot + E, 0]$
	$[T \rightarrow \cdot F, 3]$	$[E \rightarrow T \cdot, 0]$
	$[F \rightarrow \cdot (E), 3]$	
	$[F \rightarrow \cdot a, 3]$	
I_6	I_7	
$[T \rightarrow F * T, 0]$	$[F \rightarrow a \cdot, 6]$	
$[T \rightarrow \cdot F * T, 6]$	$[T \rightarrow F * T, 6]$	
$[T \rightarrow \cdot F, 6]$	$[T \rightarrow F \cdot, 6]$	
$[F \rightarrow \cdot (E), 6]$	$[T \rightarrow F * T \cdot, 0]$	
$[F \rightarrow \cdot a, 6]$	$[E \rightarrow T \cdot + E, 0]$	
	$[E \rightarrow T \cdot, 0]$	

Рис. 4.9. Списки разбора для примера 4.10.

Теорема 4.9. В списке разбора, построенном алгоритмом 4.5, $[A \rightarrow \alpha \cdot \beta, i]$ принадлежит I_j тогда и только тогда, когда $\alpha \Rightarrow^* a_{i+1} \dots a_j$ и существуют такие цепочки γ и δ , что $S \Rightarrow^* \gamma A \delta$ и $\gamma \Rightarrow^* a_1 \dots a_i$.

Доказательство. Необходимость. Эту часть теоремы докажем индукцией по числу ситуаций, которые включаются в I_0, I_1, \dots, I_j до того, как в I_j включается $[A \rightarrow \alpha \cdot \beta, i]$. Для доказательства базиса (здесь это будет I_0) заметим, что $\alpha \Rightarrow^* e$ для каждой ситуации, включенной в I_0 , так что $S \Rightarrow^* \gamma A \delta$ при $\gamma = e$.

Для доказательства шага индукции допустим, что список I_0 уже построен и предположение индукции верно для всех ситуаций, включенных в списки I_i при $i \leq j$. Пусть $[A \rightarrow \alpha \cdot \beta, i]$ включается в I_j по правилу (4). Тогда $\alpha = \alpha' a_j$ и $[A \rightarrow \alpha' \cdot a_j \beta, i]$ входит в I_{j-1} . По предположению индукции $\alpha' \Rightarrow^* a_{i+1} \dots a_{j-1}$ и существуют такие цепочки γ' и δ' , что $S \Rightarrow^* \gamma' A \delta'$ и $\gamma' \Rightarrow^* a_1 \dots a_i$. Отсюда следует, что $\alpha = \alpha' a_j \Rightarrow^* a_{i+1} \dots a_j$, и нужное утверждение выполняется при $\gamma = \gamma'$ и $\delta = \delta'$.

Пусть теперь $[A \rightarrow \alpha \cdot \beta, i]$ включается по правилу (5). Тогда $\alpha = \alpha' B$ для некоторого $B \in N$ и $[A \rightarrow \alpha' \cdot B \beta, i]$ входит в I_k для некоторого k . Кроме того, $[B \rightarrow \eta \cdot, k]$ входит в I_j для некоторой цепочки $\eta \in (N \cup \Sigma)^*$. По предположению индукции $\eta \Rightarrow^* a_{k+1} \dots a_j$ и $\alpha' \Rightarrow^* a_{i+1} \dots a_k$. Поэтому $\alpha = \alpha' B \Rightarrow^* a_{i+1} \dots a_j$. В силу того же предположения существуют такие γ' и δ' , что $S \Rightarrow^* \gamma' A \delta'$ и $\gamma' \Rightarrow^* a_1 \dots a_i$. Остальная часть нужного нам утверждения выполняется опять при $\gamma = \gamma'$ и $\delta = \delta'$.

В последнем случае, когда $[A \rightarrow \alpha \cdot \beta, i]$ включается по правилу (6), $\alpha = e$ и $i = j$. Элементарную проверку нужного утверждения предоставляем читателю, так что первую часть теоремы считаем доказанной.

Достаточность. Эта часть состоит в доказательстве утверждения

(4.2.1) если $S \Rightarrow^* \gamma A \delta$, $\gamma \Rightarrow^* a_1 \dots a_i$, $A \rightarrow \alpha \beta$ принадлежит P и $\alpha \Rightarrow^* a_{i+1} \dots a_j$, то $[A \rightarrow \alpha \cdot \beta, i]$ включается в список I_j

Утверждение (4.2.1) надо доказать для всевозможных значений входящих в него параметров. Каждый набор параметров состоит из цепочек α, β, γ и δ , нетерминала A и чисел i и j , так как S и $a_1 \dots a_n$ фиксированы. Обозначим такой набор $[\alpha, \beta, \gamma, \delta, A, i, j]$. Заключение, которое нужно получить для этого набора, состоит в том, что $[A \rightarrow \alpha \cdot \beta, j]$ включается в I_j . Заметим, что в этом утверждении γ и δ не фигурируют явно.

Введем понятие ранга набора параметров и проведем доказательство индукцией по рангу. Ранг набора $\mathcal{I} = [\alpha, \beta, \gamma, \delta, A, i, j]$ вычисляется следующим образом:

Пусть $\tau_1(\mathcal{I})$ — длина кратчайшего вывода $S \Rightarrow^* \gamma A \delta$, $\tau_2(\mathcal{I})$ — длина кратчайшего вывода $\gamma \Rightarrow^* a_1 \dots a_i$, $\tau_3(\mathcal{I})$ — длина кратчайшего вывода $\alpha \Rightarrow^* a_{i+1} \dots a_j$. Тогда ранг набора \mathcal{I} равен $\tau_1(\mathcal{I}) + 2[j + \tau_2(\mathcal{I}) + \tau_3(\mathcal{I})]$.

Теперь докажем (4.2.1). Если ранг набора $\mathcal{I} = [\alpha, \beta, \gamma, \delta, A, i, j]$ равен 0, то $\tau_1(\mathcal{I}) = \tau_2(\mathcal{I}) = \tau_3(\mathcal{I}) = j = 0$. Отсюда $\alpha = \gamma = \delta = e$ и $A = S$. Тогда нужно показать, что $[S \rightarrow \cdot \beta, 0]$ входит в I_0 . Однако это сразу следует из правила (1), так как правило $S \rightarrow \beta$ должно принадлежать P .

Для доказательства шага индукции предположим, что набор параметров \mathcal{I} для утверждения (4.2.1) имеет ранг $r > 0$ и (4.2.1) верно для наборов с меньшими рангами. Рассмотрим отдельно три случая, связанные с тем, что α может оканчиваться терминалом, нетерминалом и быть пустой цепочкой.

Случай 1: $\alpha = \alpha' a$ для некоторого $a \in \Sigma$. Так как $\alpha \Rightarrow^* a_{i+1} \dots a_j$, то $a = a_j$. Рассмотрим набор $\mathcal{I}' = [\alpha', a_j \beta, \gamma, \delta, A, i, j-1]$. Так как $A \rightarrow \alpha' a_j \beta$ принадлежит P , то \mathcal{I}' служит набором параметров для (4.2.1) и его ранг, как легко видеть, равен $r - 2$. Отсюда следует, что $[A \rightarrow \alpha' \cdot a_j \beta, i]$ входит в I_{j-1} . По правилу (4) ситуация $[A \rightarrow \alpha \cdot \beta, i]$ будет помещена в I_j .

Случай 2: $\alpha = \alpha' B$ для некоторого $B \in N$. Существует такое число $i \leq k \leq j$, что $\alpha' \Rightarrow^* a_{i+1} \dots a_k$ и $B \Rightarrow^* a_{k+1} \dots a_j$. Исследуя набор $\mathcal{I}' = [\alpha', B \beta, \gamma, \delta, A, i, k]$ меньшего ранга, заключаем, что $[A \rightarrow \alpha' \cdot B \beta, i]$ входит в I_k . Пусть $B \Rightarrow \eta$ — первый шаг в кратчайшем выводе $B \Rightarrow^* a_{k+1} \dots a_j$. Возьмем набор $\mathcal{I}'' = [\eta, e, \gamma \alpha', \beta \delta, B, k, j]$. Так как $S \Rightarrow^* \gamma A \delta \Rightarrow \gamma \alpha' B \beta \delta$, то $\tau_1(\mathcal{I}'') \leq \tau_1(\mathcal{I}) + 1$. Пусть n_1 — минимальное число шагов вывода $\alpha' \Rightarrow^* a_{i+1} \dots a_k$ и n_2 — минимальное число шагов вывода $B \Rightarrow^* a_{k+1} \dots a_j$. Тогда $\tau_3(\mathcal{I}) = n_1 + n_2$. Так как $B \Rightarrow \eta \Rightarrow^* a_{k+1} \dots a_j$, то $\tau_3(\mathcal{I}'') = n_2 - 1$. Легко видеть, что $\tau_2(\mathcal{I}'') = \tau_2(\mathcal{I}) + n_1$. Следовательно, $\tau_2(\mathcal{I}'') + \tau_3(\mathcal{I}'') = \tau_2(\mathcal{I}) + n_1 + n_2 - 1 = \tau_2(\mathcal{I}) + \tau_3(\mathcal{I}) - 1$. Таким образом, $\tau_1(\mathcal{I}'') + 2[j + \tau_2(\mathcal{I}'') + \tau_3(\mathcal{I}'')]$ меньше r . По предположению индукции для \mathcal{I}'' заключаем, что $[B \rightarrow \eta \cdot, k]$ входит в I_j , и так как $[A \rightarrow \alpha' \cdot B \beta, i]$ входит в I_k , то по правилу (2) или (5) $[A \rightarrow \alpha \cdot \beta, i]$ включается в I_j .

Случай 3: $\alpha = e$. В этом случае можно считать, что $i = j$ и $\tau_3(\mathcal{I}) = 0$. Так как $r > 0$, то длина вывода $S \Rightarrow^* \gamma A \delta$ больше нуля. Если бы она равнялась нулю, то было бы $\tau_1(\mathcal{I}) = 0$, а тогда $\gamma = e$, так что $\tau_2(\mathcal{I}) = i = 0$. Поскольку в этом случае, как уже отмечалось, $i = j$ и $\tau_3(\mathcal{I}) = 0$, то было бы $r = 0$.

Итак, можно найти $B \in N$ и такие цепочки $\gamma', \gamma'', \delta'$ и δ'' из $(N \cup \Sigma)^*$, что $S \Rightarrow^* \gamma' B \delta' \Rightarrow \gamma'' \gamma' A \delta'' \delta'$, где $B \rightarrow \gamma'' A \delta''$ принадлежит P , $\gamma = \gamma' \gamma''$, $\delta = \delta'' \delta'$ и $\gamma' B \delta' — результат предпоследнего шага кратчайшего вывода $S \Rightarrow^* \gamma A \delta$. Возьмем набор $\mathcal{I}' = [\gamma'', A \delta'', \gamma', \delta', B, k, j]$, где k — такое число, что $\gamma' \Rightarrow^* a_1 \dots a_k$ и $\gamma'' \Rightarrow^* a_{k+1} \dots a_j$. Пусть наименьшие длины указанных выводов равны$

соответственно n_1 и n_2 . Тогда $\tau_2(\mathcal{I}') = n_1$, $\tau_3(\mathcal{I}') = n_2$ и $\tau_2(\mathcal{I}) = n_1 + n_2$. Мы уже убедились, что $\tau_3(\mathcal{I}) = 0$, а B , γ' и δ' выбраны так, что $\tau_1(\mathcal{I}') = \tau_1(\mathcal{I}) - 1$. Поэтому ранг набора \mathcal{I}' равен $r - 1$. Таким образом, $[B \rightarrow \gamma' \cdot A\delta', k]$ входит в I_j . По правилу (6) или (3) ситуация $[A \rightarrow \cdot \beta, j]$ будет включена в I_j . \square

Заметим, что в качестве частного случая теоремы 4.9 мы получаем, что $[S \rightarrow \alpha \cdot, 0]$ входит в I_n тогда и только тогда, когда $S \rightarrow \alpha$ принадлежит P и $\alpha \Rightarrow^* a_1 \dots a_n$, т. е. $a_1 \dots a_n$ принадлежит $L(G)$ тогда и только тогда, когда $[S \rightarrow \alpha \cdot, 0]$ для некоторой цепочки α входит в I_n .

Теперь исследуем вопрос о времени работы алгоритма 4.5. Читателю предоставляем доказать, что вообще для разбора любой цепочки длины n в произвольной заданной грамматике достаточно $O(n^3)$ подходящим образом определенных элементарных шагов. Мы докажем сейчас, что если грамматика однозначная, то достаточно $O(n^2)$ шагов.

Лемма 4.6. Пусть $G = (N, \Sigma, P, S)$ — однозначная КС-грамматика и $a_1 \dots a_n$ — цепочка из Σ^* . Тогда при выполнении алгоритма 4.5 мы пытаемся включить $[A \rightarrow \alpha \cdot \beta, i]$ в I_j не более одного раза, если $\alpha \neq e$.

Доказательство. Ситуацию $[A \rightarrow \alpha \cdot \beta, i]$ можно включить в I_j только на шагах (2), (4), или (5). Если она включается на шаге (4), то последний символ цепочки α — терминал, а если на шагах (2) или (5), то — нетерминал. В первом случае результат очевиден. Во втором случае допустим, что $[A \rightarrow \alpha' B \cdot \beta, i]$ включается в I_j , когда рассматриваются две различные ситуации $[B \rightarrow \gamma \cdot, k]$ и $[B \rightarrow \delta \cdot, l]$. Тогда ситуация $[A \rightarrow \alpha' B \beta, i]$ должна оказаться одновременно в I_k и в I_l (возможно $k = l$).

Допустим, что $k \neq l$. По теореме 4.9 существуют такие θ_1 , θ_2 , θ_3 и θ_4 , что $S \Rightarrow^* \theta_1 A \theta_2 \Rightarrow \theta_1 \alpha' B \theta_2 \Rightarrow^* a_1 \dots a_n$ и $S \Rightarrow^* \theta_3 A \theta_4 \Rightarrow \theta_3 \alpha' B \theta_4 \Rightarrow^* a_1 \dots a_n$. Но в первом выводе $\theta_1 \alpha' \Rightarrow^* a_1 \dots a_k$, а во втором $\theta_3 \alpha' \Rightarrow^* a_1 \dots a_l$. Тогда для цепочки $a_1 \dots a_n$ существуют два разных дерева вывода, в которых $a_{l+1} \dots a_j$ выводится из $\alpha' B$ двумя разными способами.

Пусть $k = l$. Тогда должно быть $\gamma \neq \delta$. Снова для цепочки $a_1 \dots a_n$ легко найти два различных дерева вывода. Детали доказательства оставляем в качестве упражнения. \square

Изучим теперь сложность алгоритма 4.5. Определение „элементарной операции“ этого алгоритма предоставляем читателю. Решающий момент в доказательстве того, что алгоритм 4.5 имеет квадратичную временную сложность, состоит не в том, как определить „элементарные операции“, — подойдет любое разумное множество основных операций, применяемых при обработке списков. Главное здесь связано с организацией „бухгалтерии“ для

учета всех затрат времени. Грамматика G предполагается фиксированной, так что обычные операции с ее правилами и символами можно считать элементарными. Как и в предыдущем разделе, вопрос о зависимости времени от „объема“ грамматики оставляем в качестве упражнения.

Шаг (1) для I_0 можно, очевидно, проделать за фиксированное число элементарных операций. Шаг (3) для I_0 и шаг (6) в общем случае можно проделать за ограниченное число элементарных операций каждый раз, когда рассматривается очередная ситуация, при условии, что мы следим за ситуациями $[A \rightarrow \alpha \cdot \beta, i]$, уже включенными в I_j . Так как грамматика G фиксирована, эту информацию можно хранить для каждого i в конечной таблице. Тогда нет необходимости просматривать весь список I_j , чтобы узнать, находятся ли уже в нем эти ситуации.

На шагах (2), (4) и (5) включение ситуаций в I_j произойдет в том случае, если удастся отыскать в некотором списке I_i ($i \leq j$) все ситуации, в которых нужный символ расположен справа от точки, причем этот символ — терминальный на шаге (4) и нетерминальный на шагах (2) и (5). Таким образом, для каждой ситуации из списка мы должны устроить две связи.

Первая из них указывает следующую ситуацию списка. Эта связь позволяет по очереди рассматривать каждую ситуацию. Вторая указывает следующую ситуацию с тем же символом, расположенным справа от точки. Эта связь позволяет эффективно просматривать список на шагах (2), (4) и (5).

Общая стратегия заключается в том, чтобы при включении новых ситуаций каждую ситуацию из списка рассматривать только один раз. Однако сразу после включения в I_j ситуации вида $[A \rightarrow \alpha \cdot B \beta, i]$ мы справляемся в конечной таблице, заготовленной для I_j , есть ли в I_j ситуации вида $[B \rightarrow \gamma \cdot, j]$. Если да, включаем в I_j также и $[A \rightarrow \alpha B \cdot \beta, i]$.

Заметим, что в качестве первых компонент может появиться фиксированное число цепочек, скажем k . Поэтому в I_j может появиться не более $k(j+1)$ ситуаций. Если мы покажем, что алгоритм 4.5 расходует на каждую ситуацию из списка фиксированное количество времени, скажем c , то этим будет доказано, что общие затраты времени составляют $O(n^2)$, так как

$$c \sum_{i=0}^n k(j+1) = \frac{1}{2} ck(n+1)(n+2) \leq c'n^2$$

для некоторой константы c' .

„Бухгалтерский трюк“ состоит в следующем. Время расходуется на ситуацию при разных обстоятельствах — и тогда, когда она рассматривается, и тогда, когда она включается в список. Максимальное количество затрачиваемого времени в обоих слу-

чаях фиксировано. Фиксированное время расходуется также на список в целом.

Представляем читателю показать, что I_0 можно построить за фиксированное время. Мы рассмотрим ситуации в списке I_j для $j > 0$. На шаге (4) исследуется a_j и предыдущий список. Для каждой ситуации из I_{j-1} с символом a_j , расположенным справа от точки, в I_j включается некоторая ситуация. Так как можно исследовать только те ситуации из I_{j-1} , которые удовлетворяют этому условию, то на каждую включаемую ситуацию тратится лишь фиксированное время, а кроме того, фиксированное время надо потратить на список I_j как таковой для исследования a_j и нахождения первой ситуации в I_{j-1} , содержащей $\cdot a_j$.

Теперь рассмотрим каждую ситуацию из I_j и затраты времени на нее в случаях, когда применяется шаг (5) или (6). Шаг (6) можно выполнить за фиксированное время, так как надо только исследовать таблицу, связанную с I_j , которая говорит о том, включены ли уже все $[A \rightarrow \alpha, j]$ для соответствующего A . За фиксированное время таблица исследуется, и, если необходимо, в I_j включается фиксированное число ситуаций. Этим исчерпывается все время, затрачиваемое на рассматриваемую ситуацию.

Если применяется шаг (5), то в некотором списке I_k для $k \leq j$ надо просмотреть все ситуации, содержащие $\cdot B$ для некоторого конкретного B . Каждый раз, когда обнаруживается такая ситуация, в список I_j включается другая ситуация, и это время относится к включаемой ситуации, а не к рассматриваемой!

Чтобы показать, что время, затрачиваемое на любую ситуацию из любого списка, ограничено сверху фиксированным числом, достаточно заметить, что по лемме 4.6 для однозначной грамматики предпринимается лишь одна попытка включить ситуацию в список. Отсюда следует также, что на шаге (5) не надо тратить время на проверку того, появилась ли уже в списке соответствующая ситуация.

Теорема 4.10. Если исходная грамматика однозначна, то алгоритм 4.5 можно выполнить для входной цепочки длины n за $O(n^2)$ разумно определенных элементарных операций.

Доказательство. Формализацию приведенного выше рассуждения и понятия элементарной операции оставляем в качестве упражнения. \square

Теорема 4.11. В случае произвольной КС-грамматики алгоритм 4.5 можно выполнить для входной цепочки длины n за $O(n^3)$ разумно определенных элементарных операций.

Доказательство. Упражнение. \square

Заключительная часть анализа алгоритма Эрли касается метода построения разбора по законченному списку разбора. Для

этого мы опишем алгоритм 4.6, порождающий по такому списку правый разбор. Правый разбор выбран потому, что в этом случае алгоритм несколько упрощается. Небольшое изменение алгоритма позволит находить левый разбор.

Будем предполагать для простоты, что исходная грамматика не содержит циклов. Если в грамматике есть циклы, то некоторые входные цепочки могут иметь сколь угодно много разборов. Однако алгоритм 4.6 можно модифицировать так, чтобы приспособить его к грамматикам с циклами (упр. 4.2.23).

Следует отметить, что, как и в случае алгоритма 4.4, можно упростить алгоритм 4.6, снабдив указателями каждую ситуацию, включаемую в список алгоритмом 4.5. Они указывают одну или две ситуации, которые привели к включению этой ситуации в список.

Алгоритм 4.6. Построение правого разбора по списку разбора.

Вход. КС-грамматика $G = (N, \Sigma, P, S)$ без циклов, правила которой занумерованы числами от 1 до p , входная цепочка $w = a_1 \dots a_n$ и список разбора I_0, I_1, \dots, I_n для цепочки w .

Выход. Правый разбор π цепочки w или сообщение об ошибке.

Метод. Если в I_n нет ситуации вида $[S \rightarrow \alpha \cdot, 0]$, то $w \notin L(G)$. Поэтому надо выдать сигнал „ошибка“ и остановиться. В противном случае положить $\pi = e$ и выполнить процедуру $R([S \rightarrow \alpha \cdot, 0], n)$.

Процедура $R([A \rightarrow \beta \cdot, i], j)$ определяется так:

(1) Если i — номер правила $A \rightarrow \beta$, то пусть значением переменной π будет цепочка, начинающаяся номером i , за которым следует предыдущее значение π . (Мы предполагаем, что π — глобальная переменная.)

(2) Если $\beta = X_1 X_2 \dots X_m$, положить $k = m$ и $l = j$.

(3) (а) Если $X_k \in \Sigma$, вычесть 1 из k и из l .

(б) Если $X_k \in N$, найти в I_l ситуацию $[X_k \rightarrow \gamma \cdot, r]$ для некоторого r , для которого $[A \rightarrow X_1 X_2 \dots X_{k-1} \cdot X_k \dots X_m, i]$ входит в I_r . Затем выполнить $R([X_k \rightarrow \gamma \cdot, r], l)$, вычесть 1 из k и положить $l = r$.

(4) Повторять шаг (3), пока не станет $k = 0$. После этого остановиться. \square

Работа алгоритма 4.6 состоит в прослеживании правого вывода входной цепочки с помощью списка разбора, позволяющего определять правила, примененные в ходе вывода. Вызов процедуры R с аргументами $[A \rightarrow \beta \cdot, i]$ и j добавляет к левому концу текущего частичного разбора номер правила $A \rightarrow \beta$. Если $\beta = v_0 B_1 v_1 B_2 v_2 \dots B_s v_s$, где B_1, \dots, B_s — все вхождения нетер-

миналов в цепочку β , то процедура R определяет первое правило, использованное при развертке $B_t (1 \leq t \leq s)$, скажем $B_t \rightarrow \beta_t$, и позицию во входной цепочке w , непосредственно предшествующую первому терминальному символу, выводимому из B_t . Затем делаются рекурсивные вызовы процедуры R в следующем порядке:

$$\begin{aligned} R([B_s \rightarrow \beta_s \cdot, i_s], j_s) \\ R([B_{s-1} \rightarrow \beta_{s-1} \cdot, i_{s-1}], j_{s-1}) \\ \vdots \\ R([B_1 \rightarrow \beta_1 \cdot, i_1], j_1) \end{aligned}$$

где

- (1) $j_s = j - |v_s|$,
- (2) $j_q = i_{q+1} - |v_q|$ для $1 \leq q < s$.

Пример 4.11. Применим алгоритм 4.6 к списку разбора из примера 4.10, чтобы получить правый разбор входной цепочки $(a+a)*a$. Вначале можно выполнить $R([E \rightarrow T \cdot, 0], 7)$. На шаге (1) переменная π принимает значение 2 (номер правила $E \rightarrow T$). Затем полагаем $k=1$ и $l=7$ и выполняем шаг (3б). В I_1 , находим ситуацию $[T \rightarrow F * T \cdot, 0]$, а в I_0 — ситуацию $[E \rightarrow \cdot T, 0]$. Таким образом, будет выполнена процедура $R([T \rightarrow F * T \cdot, 0], 7)$, в результате чего к π будет добавлен слева номер 3 и станет $\pi=32$. Согласно этому вызову процедуры R , на шаге (2) надо положить $k=3$ и $l=7$.

Затем выполняется шаг (3б) для $k=3$. Находим $[T \rightarrow F \cdot, 6]$ в I_0 и $[T \rightarrow F * \cdot T, 0]$ в I_6 и вызываем $R([T \rightarrow F \cdot, 6], 7)$. После завершения этого вызова положим $k=2$ и $l=6$. На шаге (3а) придется рассмотреть $*$ и положить $k=1$ и $l=5$. Далее находим $[F \rightarrow (E) \cdot, 0]$ в I_5 и $[T \rightarrow \cdot F * T, 0]$ в I_0 и, следовательно, вызываем $R([F \rightarrow (E) \cdot, 0], 5)$.

Продолжая в том же духе, получаем правый разбор 64642156432.

Вызовы процедуры R показаны на рис. 4.10, где они наложены на дерево вывода цепочки $(a+a)*a$. \square

Теорема 4.12. Алгоритм 4.6 правильно находит правый разбор цепочки $a_1 \dots a_n$, если таковой существует, и его можно выполнить за время $O(n^2)$.

Доказательство. Прямой индукцией по порядку вызовов процедуры R можно показать, что порождается правый разбор. Эту часть доказательства мы оставляем в качестве упражнения.

По аналогии с доказательством теоремы 4.10 можно показать, что на вызов $R([A \rightarrow \beta \cdot, i], j)$ расходуется время $O((j-i)^2)$, если сначала показать, что шаг (3б) требует $O(j-i)$ элементарных операций. Для этого нужно предварительно обрабатывать списки так, чтобы для рассмотрения всех ситуаций из I_k , вторая компонента которых равна l , требовалось фиксированное

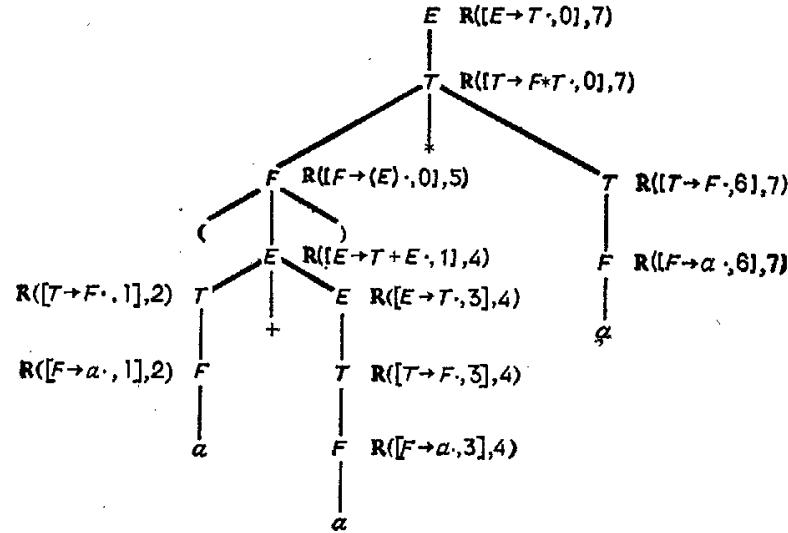


Рис. 4.10. Диаграмма выполнения алгоритма 4.6.

время. Иными словами, в каждом списке нужно связать ситуацию с общей второй компонентой и ввести заголовок, указывающий первый элемент списка. Эту обработку можно очевидным образом проделать за время $O(n^2)$.

Затем на шаге (3б) исследуются ситуации списка I_i со второй компонентой $r=l, l-1, \dots, i$, пока не обнаружится нужная ситуация вида $[X_k \rightarrow \gamma \cdot, r]$. Проверка того, что получена нужная ситуация, занимает фиксированное время, так как все ситуации со второй компонентой i можно найти в списке I_r за фиксированное время. Общее количество времени, потраченного на шаге (3б), пропорционально, таким образом, $j-i$. \square

УПРАЖНЕНИЯ

4.2.1. Пусть G определяется правилами $S \rightarrow AS|b, A \rightarrow SA|a$. С помощью алгоритма 4.3 постройте таблицы разбора следующих цепочек:

- (а) $bbaab$,
- (б) $ababab$,
- (в) $aabb$.

4.2.2. С помощью алгоритма 4.4 получите левые разборы для тех цепочек из упр. 4.2.1, которые принадлежат языку $L(G)$.

4.2.3. С помощью алгоритма 4.5 постройте по грамматике G из упр. 4.2.1 списки разбора для цепочек из этого же упражнения.

4.2.4. С помощью алгоритма 4.6 постройте правые разборы для тех цепочек из упр. 4.2.1, которые принадлежат языку $L(G)$.

4.2.5. Пусть грамматика G задана правилами $S \rightarrow SS \mid a$. С помощью алгоритма 4.5 постройте несколько списков из последовательности I_0, I_1, \dots для входной цепочки $aa \dots$. Сколько элементарных операций требуется для вычисления I_i ?

4.2.6. Докажите теорему 4.6.

4.2.7. Докажите, что алгоритм 4.4 порождает левые разборы.

4.2.8. Дополните доказательство теоремы 4.9 (необходимость условия).

4.2.9. Покажите, что для произвольной КС-грамматики алгоритм Эрли заканчивает работу за время $O(n^3)$.

4.2.10. Дополните доказательство леммы 4.6.

4.2.11. Для теорем 4.10—4.12 опишите разумное множество элементарных операций.

4.2.12. Докажите теорему 4.10.

4.2.13. Докажите теорему 4.11.

4.2.14. Покажите, что алгоритм 4.6 порождает правые разборы.

4.2.15. Модифицируйте алгоритм 4.3 так, чтобы он работал для КС-грамматик не в нормальной форме Хомского. Указание: Элемент t_{ij} должен содержать не только нетерминалы A , из которых выводится $a_i \dots a_{i+j-1}$, но также и некоторые подцепочки правых частей правил, из которых выводится $a_i \dots a_{i+j-1}$.

***4.2.16.** Покажите, что для линейных грамматик можно модифицировать алгоритм 4.3 так, чтобы он заканчивал работу за время $O(n^2)$.

***4.2.17.** Можно модифицировать алгоритм 4.3, используя „заглядывание вперед“ на $k \geq 0$ символов. По данной грамматике G и входной цепочке $w = a_1 a_2 \dots a_n$ будем строить таблицу разбора T так, что t_{ij} содержит A тогда и только тогда, когда

$$(1) S \Rightarrow^* aAa,$$

$$(2) A \Rightarrow^+ a_i \dots a_{i+j-1},$$

$$(3) a_{i+j} a_{i+j+1} \dots a_{i+j+k-1} = \text{FIRST}_k(x).$$

Таким образом, A помещается в ячейку t_{ij} при условии, что k входных символов, расположенных справа от a_{i+j-1} , могут законно появиться после A в некоторой выводимой цепочке. Алгоритм 4.3 „заглядывает вперед“ на 0 символов. Модифицируйте алгоритм 4.3 так, чтобы он заглядывал вперед на $k \geq 1$ символов. Какова временная сложность такого алгоритма?

***4.2.18.** Можно и алгоритм Эрли модифицировать так, чтобы он использовал заглядывание вперед. При этом ситуации будут иметь вид $[A \rightarrow \alpha \cdot \beta, i, u]$, где u — „увиденная впереди“ цепочка длины k . Эта ситуация включается в список I_j , только если существует вывод $S \Rightarrow^* \gamma Auv$, где $\gamma \Rightarrow^* a_1 \dots a_i$, $a \Rightarrow^* a_{i+1} \dots a_f$ и $\text{FIRST}_k(\beta u)$ содержит $a_{j+1} \dots a_{j+k}$. Доведите до конца модификацию алгоритма Эрли, включающую заглядывание вперед, и затем исследуйте временную сложность полученного алгоритма.

4.2.19. Модифицируйте алгоритм 4.4 так, чтобы он порождал правые разборы.

4.2.20. Модифицируйте алгоритм 4.6 так, чтобы он порождал левые разборы.

4.2.21. Покажите, что можно модифицировать алгоритм 4.4 так, чтобы он порождал разбор за линейное время, если при построении таблицы разбора каждый нетерминал $A \in t_{ij}$ снабдить указателями, дающими те нетерминалы $B \in t_{ik}$ и $C \in t_{i+k, j-k}$, которые привели к тому, что на шаге (2) алгоритма 4.3 нетерминал A был помещен в ячейку t_{ij} .

4.2.22. Покажите, что если модифицировать алгоритм 4.5, включив в него для каждой ситуации указатели, дающие те ситуации, которые привели к тому, что данная ситуация была включена в список, то правый (или левый) разбор можно получить по списку разбора за линейное время.

4.2.23. Модифицируйте алгоритм 4.6 так, чтобы он работал для произвольных КС-грамматик (включая грамматики с циклами). Указание: Введите в списки указатели, как в упр. 4.2.22.

4.2.24. Каково максимальное число ситуаций, которые могут появиться в списке I_j в ходе работы алгоритма 4.5?

***4.2.25.** Говорят, что грамматика G имеет *конечную степень неоднозначности*, если существует такая константа k , что любая цепочка $w \in L(G)$ имеет не более k различных левых разборов. Покажите, что алгоритм Эрли тратит время $O(n^2)$ для любых грамматик, имеющих конечную степень неоднозначности.

Открытые проблемы

Мало что известно о том, какое на самом деле необходимо время для синтаксического анализа произвольной КС-грамматики.

Фактически нет хорошей верхней оценки времени, требуемого для распознавания цепочек из $L(G)$ для произвольной КС-грамматики G , не говоря уже о разборе. Поэтому предлагаем следующие открытые проблемы и области исследования.

4.2.26. Существует ли верхняя граница, меньшая чем $O(n^3)$, для времени распознавания произвольного КС-языка на некоторой разумной модели машины с произвольным доступом к памяти или на многоленточной машине Тьюринга¹⁾?

4.2.27. Существует ли лучшая верхняя граница, чем $O(n^2)$, для времени распознавания однозначных КС-языков?

Проблемы для исследования

4.2.28. Найдите КС-язык, который нельзя распознать за время $f(n)$ на машине с произвольным доступом к памяти или машине Тьюринга (последнее сделать легче), где $f(n)$ растет быстрее n , т. е. $\lim_{n \rightarrow \infty} (n/f(n)) = 0$. Можете ли Вы указать КС-язык, который, по-видимому, требует времени для распознавания больше, чем $O(n^2)$ (даже если Вы не умеете это доказать)?

4.2.29. Найдите широкий класс КС-грамматик, для которых разбор с помощью алгоритма Эрли возможен за линейное время. Найдите широкий класс неоднозначных грамматик, для которых разбор с помощью алгоритма Эрли возможен за время $O(n^2)$. (Отметим, что в первом из этих классов содержатся все детерминированные языки.)

Упражнения на программирование

4.2.30. Для одной из грамматик, данных в приложении, постройте синтаксический анализатор, использующий алгоритм Эрли.

4.2.31. Постройте программу, которая воспринимает в качестве входа произвольную КС-грамматику и выдает для нее анализатор, использующий алгоритм Эрли.

Замечания по литературе

Алгоритм 4.3 был независимо изобретен несколькими авторами. В книге Хейса [1967] излагается одна из версий этого алгоритма, которая приписывается там Дж. Коку.

Янгер [1967] с помощью алгоритма 4.3 показал, что временная сложность проблемы принадлежности для КС-языков не больше $O(n^3)$. Аналогичный алгоритм приведен в работе Касами [1965]. Алгоритм 4.5 описан в диссертации Эрли [1968]. В работе Касами и Тории [1969] сообщается об алгоритме, анализирующем однозначные КС-грамматики за время $O(n^2)$.

¹⁾ Прямо в таком виде эта проблема решена в работе Валнанта [1975], где доказана верхняя оценка $O(n^{2.81})$. Неизвестно, однако, можно ли еще понизить эту оценку. (См. также [Грэхем и др., 1976].) — *Прим. перев.*

ОДНОПРОХОДНЫЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ БЕЗ ВОЗВРАТОВ

5

В гл. 4 излагались алгоритмы с возвратами, позволяющие моделировать недетерминированные левые и правые анализаторы для широкого класса контексто-свободных грамматик. Однако оказалось, что в некоторых случаях такое моделирование с точки зрения затрат времени слишком расточительно. В этой главе мы рассмотрим классы КС-грамматик, для которых можно построить эффективные анализаторы, расходящиеся на обработку входной цепочки длины n время $c_1 n$ и память $c_2 n$, где c_1 и c_2 — малые константы.

За эту эффективность приходится расплачиваться тем, что ни один из классов грамматик, для которых можно строить такие эффективные анализаторы, не порождает все КС-языки. Однако похоже, что эти ограниченные классы грамматик адекватно отражают все синтаксические черты языков программирования, обычно определяемых с помощью КС-грамматик.

Излагаемые в этой главе алгоритмы разбора характеризуются тем, что входная цепочка считывается один раз слева направо и процесс разбора полностью детерминирован¹⁾. Фактически мы просто ограничиваем класс КС-грамматик так, чтобы для грамматик из этого класса всегда можно было построить детерминированный левый или правый анализатор.

Классы грамматик, рассматриваемые в данной главе, содержат

(1) LL(k)-грамматики, для которых левый анализатор работает детерминированно, если позволить ему принимать во внимание k входных символов, расположенных справа от текущей входной позиции²⁾;

(2) LR(k)-грамматики, для которых правый анализатор работает детерминированно, если позволить ему „видеть“ k входных символов, расположенных вслед за текущей входной позицией;

¹⁾ Существенно еще и то, что рабочая память представленных здесь алгоритмов — это один магазин, а не, скажем, лента машины Тьюринга или два магазина. — *Прим. перев.*

²⁾ Это не приводит к расширению понятия ДМП-преобразователя, так как k „увиденных впереди“ символов хранятся в конечной памяти управляющего устройства.

(3) грамматики предшествования, для которых правый анализатор может находить основу правовыводимой цепочки, учитывая только некоторые отношения между парами ее смежных символов.

5.1. LL(k)-ГРАММАТИКИ

В этом разделе будет изучен самый широкий „естественный“ класс левоанализируемых грамматик, а именно класс LL(k)-грамматик.

5.1.1. Определение LL(k)-грамматики

Для начала предположим, что $G = (N, \Sigma, P, S)$ — однозначная грамматика и $w = a_1 a_2 \dots a_n$ — цепочка из $L(G)$. Тогда существует единственная последовательность левовыводимых цепочек $\alpha_0, \alpha_1, \dots, \alpha_m$, для которой $S = \alpha_0, \alpha_i p_i \Rightarrow \alpha_{i+1}$ при $0 \leq i < m$ и $\alpha_m = w$. Последовательность $p_0 p_1 \dots p_{m-1}$ — левый разбор цепочки w .

Допустим, что мы хотим найти этот левый разбор, просматривая w один раз слева направо. Можно попытаться сделать это, строя последовательность левовыводимых цепочек $\alpha_0, \alpha_1, \dots, \alpha_m$. Если $\alpha_i = a_1 \dots a_j A \beta$, то к данному моменту анализа мы уже прочли первые j входных символов и сравнили их с первыми j символами цепочки α_i . Было бы желательно определить α_{i+1} , зная только $a_1 \dots a_j$ (часть входной цепочки, считанную к данному моменту), несколько следующих входных символов ($a_{j+1} a_{j+2} \dots a_{j+k}$ для некоторого фиксированного k) и нетерминал A . Если эти три фактора однозначно определяют, какое правило надо применить для развертки нетерминала A , то α_{i+1} точно определяется по α_i и k входным символам $a_{j+1} a_{j+2} \dots a_{j+k}$.

Грамматика, в которой каждый левый вывод обладает этим свойством, называется LL(k)-грамматикой. Мы увидим, что для каждой LL(k)-грамматики можно построить детерминированный левый анализатор, работающий линейное время. Сначала дадим несколько определений.

Определение. Пусть $\alpha = x\beta$ — такая левовыводимая цепочка в грамматике $G = (N, \Sigma, P, S)$, что $x \in \Sigma^*$, а β либо начинается нетерминалом, либо пустая цепочка. Будем называть x *законченной частью* цепочки α , а β — *незаконченной частью*. Границу между x и β будем называть *рубежом*.

Пример 5.1. Пусть $\alpha = abacAaB$. Тогда $abac$ — законченная часть цепочки α , AaB — незаконченная часть. Если $\alpha = abc$, то abc — законченная часть и e — незаконченная часть, а рубежом для них служит правый конец цепочки. \square

Идею, лежащую в основе понятия LL(k)-грамматики, интуитивно можно объяснить так: если мы строим левый вывод $S \Rightarrow_l^* w$ и уже построили $S \Rightarrow_l \alpha_1 \Rightarrow_l \alpha_2 \Rightarrow_l \dots \Rightarrow_l \alpha_i$ так, что $\alpha_i \Rightarrow_l^* w$, то можно построить α_{i+1} , т. е. сделать очередной шаг вывода, видя только законченную часть цепочки α_i и „еще немножко“, а именно следующие k входных символов цепочки w . (Заметим, что законченная часть цепочки α_i является префиксом цепочки w .) Важно отметить, что если мы не видим всей цепочки w , когда строится α_{i+1} , то фактически не знаем, какая терминальная цепочка в конечном счете выводится из S . Таким образом, из условия,

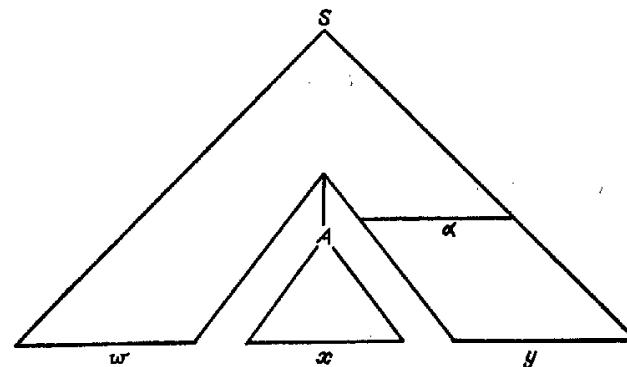


Рис. 5.1. Дерево вывода цепочки wxy .

налагаемого на LL(k)-грамматику, вытекает, что α_{i+1} по существу не зависит (не считая следующих k терминальных символов) от того, что выводится из незаконченной части цепочки α_i .

В терминах деревьев этот процесс выглядит так: дерево вывода цепочки wxy в LL(k)-грамматике строится, начиная с корня, детерминировано сверху вниз. А именно, если уже построено частичное дерево вывода с короной $wA\alpha$, то по w и первым k символам цепочки xy можно сказать, какое правило применить к A . Набросок завершенного дерева показан на рис. 5.1.

Напомним, что в гл. 4 для КС-грамматики $\hat{G} = (N, \Sigma, P, S)$ была определена функция $\text{FIRST}_k^G(\alpha)$ (где k — целое число и $\alpha \in (N \cup \Sigma)^*$), равная $\{\omega \in \Sigma^* \mid \text{либо } |\omega| < k \text{ и } \alpha \Rightarrow_G^* \omega, \text{ либо } |\omega| = k \text{ и } \alpha \Rightarrow_G^* \omega x \text{ для некоторого } x\}$. Мы будем опускать k и G , если это не вызовет недоразумений, и писать просто FIRST.

Если α состоит только из терминалов, то $\text{FIRST}_k(\alpha) = \{\omega\}$, где ω — это первые k символов цепочки α при $|\alpha| \geq k$ и $\omega = \alpha$ при $|\alpha| < k$. В этом случае будем писать $\text{FIRST}_k(\alpha) = \omega$, а не $\{\omega\}$. Значение $\text{FIRST}_k^G(\alpha)$ легко находится по заданной грамматике G . Соответствующий алгоритм мы опишем в разд. 5.1.6.

Определение. КС-грамматика $G = (N, \Sigma, P, S)$ называется $LL(k)$ -грамматикой для некоторого фиксированного k , если из существования двух левых выводов

$$(1) S \Rightarrow_i^* wA\alpha \Rightarrow_i^* w\beta\alpha \Rightarrow_i^* wx$$

и

$$(2) S \Rightarrow_i^* wA\alpha \Rightarrow_i^* w\gamma\alpha \Rightarrow_i^* wy,$$

для которых $\text{FIRST}_k(x) = \text{FIRST}_k(y)$, вытекает, что $\beta = \gamma$.

Говоря менее формально, G будет $LL(k)$ -грамматикой, если для данной цепочки $wA\alpha \in (N \cup \Sigma)^*$ и первых k символов (если они есть), выводящихся из $A\alpha$, существует не более одного правила, которое можно применить к A , чтобы получить вывод какой-нибудь терминальной цепочки, начинающейся с w и продолжающейся упомянутыми k терминалами.

Грамматика называется LL -грамматикой, если она $LL(k)$ -грамматика для некоторого k .

Пример 5.2. Пусть G_1 состоит из правил $S \rightarrow aAS \mid b, A \rightarrow a \mid bSA$. Интуитивно G_1 является $LL(1)$ -грамматикой, потому что, коль скоро дан самый левый нетерминал C в левовыводимой цепочке и следующий входной символ c , существует не более одного правила, применимого к C и приводящего к терминальной цепочке, начинающейся символом c . Переходя к определению $LL(1)$ -грамматики, мы видим, что если $S \Rightarrow_i^* wS\alpha \Rightarrow_i^* w\beta\alpha \Rightarrow_i^* wx$ и $S \Rightarrow_i^* wS\alpha \Rightarrow_i^* w\gamma\alpha \Rightarrow_i^* wy$ и цепочки x и y начинаются одним и тем же символом, то должно быть $\beta = \gamma$. В данном случае если x и y начинаются символом a , то в выводе участвовало правило $S \rightarrow aAS$ и $\beta = \gamma = aAS$. Альтернатива $S \rightarrow b$ здесь невозможна. С другой стороны, если x и y начинаются с b , то должно применяться правило $S \rightarrow b$ и $\beta = \gamma = b$. Заметим, что случай $x = y = e$ невозможен, так как из S -в грамматике G_1 не выводится e .

Когда рассматриваются два вывода $S \Rightarrow_i^* wA\alpha \Rightarrow_i^* w\beta\alpha \Rightarrow_i^* wx$ и $S \Rightarrow_i^* wA\alpha \Rightarrow_i^* w\gamma\alpha \Rightarrow_i^* wy$, рассуждение аналогично. \square

Грамматика G_1 служит примером так называемой простой $LL(1)$ -грамматики (или разделенной грамматики).

Определение. КС-грамматика $G = (N, \Sigma, P, S)$ без e -правил называется простой $LL(1)$ -грамматикой (или разделенной грамматикой), если для каждого $A \in N$ все его альтернативы начинаются различными терминальными символами.

Таким образом, в простой $LL(1)$ -грамматике для данной пары (A, a) , где $A \in N$ и $a \in \Sigma$, существует не более одного правила вида $A \rightarrow a\alpha$.

Пример 5.3. Рассмотрим более сложный случай — грамматику G_2 , определяемую правилами $S \rightarrow e \mid abA, A \rightarrow Saa \mid b$. Покажем, что G_2 — это $LL(2)$ -грамматика. Для этого докажем, что если $wB\alpha$ — любая левовыводимая цепочка грамматики G_2 и $wx \in L(G)$, то в G_2 найдется не более одного правила $B \rightarrow \beta$, для которого $\text{FIRST}_2(\beta\alpha)$ содержит $\text{FIRST}_2(x)$. Допустим, что $S \Rightarrow_i^* wS\alpha \Rightarrow_i^* w\beta\alpha \Rightarrow_i^* wx$ и $S \Rightarrow_i^* wS\alpha \Rightarrow_i^* w\gamma\alpha \Rightarrow_i^* wy$, где первые два символа цепочки x (если они есть) совпадают с первыми двумя символами цепочки y . Так как G_2 — линейная грамматика, то $\alpha \in (a + b)^*$. На самом деле можно сказать больше: либо $w = \alpha = e$, либо последним участвовало в выводе $S \Rightarrow_i^* wS\alpha$ правило $A \rightarrow Saa$. (Другим способом S в левовыводимой цепочке появиться не может.) Таким образом, либо $\alpha = e$, либо α начинается с aa .

Допустим, что при переходе от $wS\alpha$ к $w\beta\alpha$ применялось правило $S \rightarrow e$. Тогда $\beta = e$ и x — либо e , либо начинается с aa . Аналогично, если при переходе от $wS\alpha$ к $w\gamma\alpha$ применялось правило $S \rightarrow e$, то $\gamma = e$ и y — либо e , либо начинается с aa . Если при переходе от $wS\alpha$ к $w\beta\alpha$ применялось $S \rightarrow abA$, то $\beta = abA$ и x начинается с ab . Аналогично, если при переходе от $wS\alpha$ к $w\gamma\alpha$ применялось $S \rightarrow abA$, то $\gamma = abA$ и y начинается с ab . Итак, нет иных возможностей, кроме $x = y = e$, x и y начинаются с aa , x и y начинаются с ab . Из любого другого условия, которое можно наложить на первые два символа цепочек x и y , следует, что либо один, либо оба вывода невозможны. В первых двух из рассмотренных выше случаев в обоих выводах применяется правило $S \rightarrow e$ и $\beta = \gamma = e$. В третьем случае должно применяться $S \rightarrow abA$ и $\beta = \gamma = abB$.

В качестве упражнения докажите, что ситуация, при которой справа от рубежа рассматриваемой левовыводимой цепочки стоит символ A , не противоречит определению $LL(2)$ -грамматики. Прочертите также, что G_2 не является $LL(1)$ -грамматикой. \square

Пример 5.4. Рассмотрим грамматику $G_3 = (\{S, A, B\}, \{0, 1, a, b\}, P_3, S)$, где P_3 состоит из правил $S \rightarrow A \mid B, A \rightarrow aAb^0 \mid 0, B \rightarrow aBbb \mid 1$. Здесь $L(G_3) = \{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$. G_3 не является $LL(k)$ -грамматикой ни для какого k . Интуитивно, если мы начинаем с чтения достаточно длинной цепочки, состоящей из символов a , то не знаем, какое из правил $S \rightarrow A$ и $S \rightarrow B$ было применено первым, пока не встретим 0 или 1 .

Обращаясь к точному определению $LL(k)$ -грамматики, положим $w = \alpha = e$, $\beta = A$, $\gamma = B$, $x = a^k 0 b^k$ и $y = a^k 1 b^{2k}$. Тогда выводы

$$S \Rightarrow_i^* S \Rightarrow_i^* A \Rightarrow_i^* a^k 0 b^k$$

$$S \Rightarrow_i^* S \Rightarrow_i^* B \Rightarrow_i^* a^k 1 b^{2k}$$

соответствуют выводам (1) и (2) определения. Первые k символов цепочек x и y совпадают. Однако заключение $\beta = \gamma$ ложно.

Так как k здесь выбрано произвольно, то G_3 не является LL-грамматикой. В гл. 8 мы увидим, что для языка $L(G_3)$ не существует LL(k)-грамматики.

5.1.2. Предсказывающие алгоритмы разбора

Покажем, что разбор для LL(k)-грамматики очень удобно осуществить с помощью так называемого k -предсказывающего алгоритма разбора. k -предсказывающий алгоритм \mathcal{A} для КС-грамматики $G = (N, \Sigma, P, S)$ использует входную ленту, магазин и

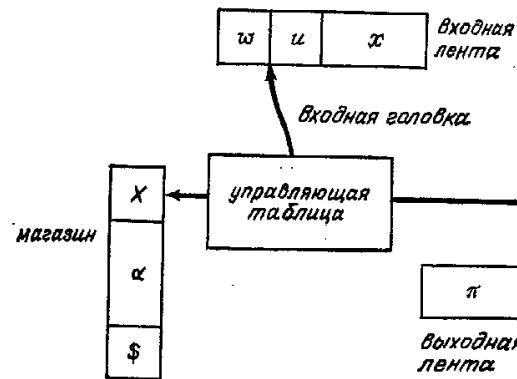


Рис. 5.2. Предсказывающий алгоритм разбора.

выходную ленту (рис. 5.2). Этот алгоритм пытается проследить левый вывод цепочки, записанной на его входной ленте.

При чтении анализируемой цепочки, находящейся на входной ленте, входная головка может „заглядывать вперед“ на k очередных символов (отсюда число k в названии k -предсказывающего алгоритма). Эту цепочку из k символов, увиденную впереди входной головкой, будем называть *аванцепочкой*¹). На рис. 5.2 аванцепочкой служит подцепочка u входной цепочки wux .

Магазин содержит цепочку $X\alpha\$$, где $X\alpha$ — цепочка магазинных символов, $\$$ — специальный символ, применяемый в качестве маркера дна магазина, и X — верхний символ магазина. Алфавит магазинных символов (без $\$$) будем обозначать Γ .

Выходная лента содержит цепочку π , состоящую из номеров правил.

Конфигурацию предсказывающего алгоритма разбора будем представлять в виде тройки $(x, X\alpha, \pi)$, где

(1) x — неиспользованная часть первоначальной входной цепочки,

¹) Термин образован по аналогии со словами „авансцена“, „авантгард“ и т. п. (в оригинале lookahead string). — Прим. перев.

- (2) $X\alpha$ — цепочка в магазине (X — верхний символ),
- (3) π — цепочка на выходной ленте.

Например, на рис. 5.2 изображена конфигурация $(ux, X\alpha\$, \pi)$.

Работой k -предсказывающего алгоритма \mathcal{A} руководит *управляющая таблица* M , задающая отображение множества $(\Gamma \cup \{\$\}) \times \Sigma^*$ в множество, в которое входят

(1) (β, i) , где $\beta \in \Gamma^*$, а i — номер правила (предполагается, что β будет либо правой частью i -го правила, либо некоторым его представлением),

- (2) **выброс**¹⁾,
- (3) **допуск**,
- (4) **ошибка**.

Алгоритм анализирует входную цепочку, проделывая последовательность тактов, очень похожих на такты преобразователя с магазинной памятью. На каждом такте сначала определяются аванцепочка u и верхний символ магазина X . Затем для определения того, что действительно надо делать, рассматривается элемент $M(X, u)$ управляющей таблицы. Как и следовало ожидать, такты предсказывающего алгоритма мы опишем в терминах отношения \vdash , определенного на множестве конфигураций. Пусть $u = \text{FIRST}_k(x)$.

(1) $(x, X\alpha, \pi) \vdash (x, \beta\alpha, \pi i)$, если $M(X, u) = (\beta, i)$. Здесь верхний символ магазина X заменяется цепочкой $\beta \in \Gamma^*$ и к выходу добавляется номер правила i . Входная головка не сдвигается.

(2) $(x, \alpha\$, \pi) \vdash (x', \alpha, \pi)$, если $M(a, u) = \text{выброс}$ и $x = ax'$. Когда верхний символ магазина совпадает с текущим входным символом (первым символом аванцепочки), он выбрасывается из магазина и входная головка сдвигается на один символ вправо.

(3) Если алгоритм достигает конфигурации $(e, \$, \pi)$, работа прекращается и выходная цепочка π называется *разбором* первоначальной входной цепочки. Будем предполагать, что всегда $M(\$, e) = \text{допуск}$, и конфигурацию $(e, \$, \pi)$ будем называть *допускающей*.

(4) Если алгоритм достигает конфигурации $(x, X\alpha, \pi)$ и $M(X, u) = \text{ошибка}$, то разбор прекращается и выдается сообщение об ошибке. Этую конфигурацию $(x, X\alpha, \pi)$ назовем *ошибочной*.

Конфигурация $(w, X_0\$, e)$, где $w \in \Sigma^*$ — анализируемая цепочка, а X_0 — выделенный начальный символ, называется *начальной*. Если $(w, X_0\$, e) \vdash^* (e, \$, \pi)$, то будем писать $\mathcal{A}(w) = \pi$ и называть π выходом алгоритма \mathcal{A} для входа w . Если из конфигурации $(w, X_0\$, e)$ допускающая конфигурация не достигается, то будем говорить, что значение $\mathcal{A}(w)$ не определено. Переводом, опреде-

¹) В оригинале *pop*. — Прим. перев.

ляемым алгоритмом \mathcal{A} , называется множество $\tau(\mathcal{A}) = \{(w, \pi) \mid \mathcal{A}(w) = \pi\}$.

Будем называть k -предсказывающий алгоритм разбора \mathcal{A} для КС-грамматики G *корректным* в том случае, когда

- (1) $L(G) = \{w \mid \mathcal{A}(w)$ определено},
- (2) если $\mathcal{A}(w) = \pi$, то π —левый разбор цепочки w .

Если работой k -предсказывающего алгоритма \mathcal{A} руководит управляющая таблица M и он корректен для КС-грамматики G , то M будем называть *корректной управляющей таблицей* для G .

Аванцепочка

	<i>a</i>	<i>b</i>	<i>в</i>
<i>S</i>	<i>aAS, 1</i>	<i>b, 2</i>	ошибка
<i>A</i>	<i>a, 3</i>	<i>bSA, 4</i>	ошибка
<i>a</i>	выброс	ошибка	ошибка
<i>b</i>	ошибка	выброс	ошибка
<i>\$</i>	ошибка	ошибка	допуск

Рис. 5.3. Управляющая таблица алгоритма \mathcal{A} .

Пример 5.5. Построим 1-предсказывающий алгоритм разбора \mathcal{A} для простой LL(1)-грамматики G_1 из примера 5.2. Прежде всего занумеруем правила грамматики G_1 :

- (1) $S \rightarrow aAS$
- (2) $S \rightarrow b$
- (3) $A \rightarrow a$
- (4) $A \rightarrow bSA$

Управляющая таблица алгоритма \mathcal{A} показана на рис. 5.3.

С помощью этой таблицы алгоритм \mathcal{A} так проанализирует входную цепочку $abbab$:

$$\begin{aligned}
 (abbab, S\$, e) &\vdash (abbab, aAS\$, 1) \\
 &\vdash (bbab, AS\$, 1) \\
 &\vdash (bbab, bSAS\$, 14) \\
 &\vdash (bab, SAS\$, 14) \\
 &\vdash (bab, bAS\$, 142) \\
 &\vdash (ab, AS\$, 142) \\
 &\vdash (ab, aS\$, 1423) \\
 &\vdash (b, S\$, 1423) \\
 &\vdash (b, b\$, 14232) \\
 &\vdash (e, \$, 14232)
 \end{aligned}$$

На первом такте $M(S, a) = (aAS, 1)$, так что верхний символ магазина S заменяется на aAS и номер правила 1 записывается на выходной ленте. На следующем такте $M(a, a) =$ *выброс*, так что a выбрасывается из магазина и входная головка сдвигается на одну позицию вправо.

Продолжая в том же духе, получаем допускающую конфигурацию $(e, \$, 14232)$. Очевидно, что 14232—левый разбор цепочки $abbab$ и, более того, \mathcal{A} —корректный 1-предсказывающий алгоритм разбора для грамматики G_1 . \square

k-предсказывающий алгоритм разбора для КС-грамматики G можно моделировать на детерминированном МП-преобразователе с концевым маркером на входной ленте. Так как входная головка МП-преобразователя может обозревать только один входной символ, аванцепочка должна храниться в копечной памяти управляющего устройства. Остальные детали моделирования очевидны.

Теорема 5.1. Пусть \mathcal{A} —*k*-предсказывающий алгоритм разбора для КС-грамматики G . Тогда существует такой детерминированный МП-преобразователь T , что $\tau(T) = \{(w\$, \pi) \mid \mathcal{A}(w) = \pi\}$.

Доказательство. Упражнение. \square

Следствие. Пусть \mathcal{A} —корректный *k*-предсказывающий алгоритм разбора для КС-грамматики G . Тогда для G существует детерминированный левый анализатор.

Пример 5.6. Построим детерминированный левый анализатор P_l для 1-предсказывающего алгоритма из примера 5.5. Так как грамматика простая, получится небольшой ДМП-преобразователь, на каждом такте сдвигающий входную головку вправо. Левый анализатор будет использовать $\$$ в качестве маркера правого конца входной ленты, а также маркера дна магазина.

Пусть $P_l = \{ (q_0, q, \text{допуск}), \{a, b, \$\}, \{S, A, a, b, \$\}, \delta, q_0, \$, \{\text{допуск}\} \}$, где δ определяется равенствами

$$\begin{aligned}
 \delta(q_0, e, \$) &= (q, S\$, e) \\
 \delta(q, a, S) &= (q, AS, 1) \\
 \delta(q, b, S) &= (q, e, 2) \\
 \delta(q, a, A) &= (q, e, 3) \\
 \delta(q, b, A) &= (q, SA, 4) \\
 \delta(q, \$, \$) &= (\text{допуск}, e, e)
 \end{aligned}$$

Легко видеть, что $(w\$, \pi) \in \tau(P_l)$ тогда и только тогда, когда $\mathcal{A}(w) = \pi$. \square

5.1.3. Следствия определения LL(k)-грамматики

Покажем, что для каждой LL(k)-грамматики можно механически построить корректный k -предсказывающий алгоритм разбора. Так как ядром предсказывающего алгоритма является управляющая таблица, надо показать, как строить по грамматике такую таблицу. Начнем с некоторых следствий определения LL(k)-грамматики.

В определении LL(k)-грамматики утверждается, что для данной левовыводимой цепочки $wA\alpha$ цепочка w и непосредственно следующие за ней k входных символов однозначно определяют, какое применить правило для развертки нетерминала A . Поэтому на первый взгляд может показаться, что для определения нужного правила надо помнить всю цепочку w . Однако это не так. Докажем теорему, очень важную для понимания LL(k)-грамматик:

Теорема 5.2. КС-грамматика $G = (N, \Sigma, P, S)$ является LL(k)-грамматикой тогда и только тогда, когда для двух различных правил $A \rightarrow \beta$ и $A \rightarrow \gamma$ из P пересечение $\text{FIRST}_k(\beta\alpha) \cap \text{FIRST}_k(\gamma\alpha)$ пусто при всех таких $wA\alpha$, что $S \Rightarrow_i^* wA\alpha$.

Доказательство. Необходимость. Допустим, что w, A, α, β и γ удовлетворяют условиям теоремы, а $\text{FIRST}_k(\beta\alpha) \cap \text{FIRST}_k(\gamma\alpha)$ содержит x . Тогда по определению FIRST для некоторых y и z найдутся выводы $S \Rightarrow_i^* wA\alpha \Rightarrow_i w\beta\alpha \Rightarrow_i^* wxy$ и $S \Rightarrow_i^* wA\alpha \Rightarrow_i w\gamma\alpha \Rightarrow_i^* wxz$. (Заметим, что здесь мы использовали тот факт, что N не содержит бесполезных нетерминалов, как это предполагается для всех рассматриваемых грамматик.) Если $|x| < k$, то $y = z = e$. Так как $\beta \neq \gamma$, то G не LL(k)-грамматика.

Достаточность. Допустим, что G не LL(k)-грамматика. Тогда найдутся такие два вывода $S \Rightarrow_i^* wA\alpha \Rightarrow_i w\beta\alpha \Rightarrow_i^* wxy$ и $S \Rightarrow_i^* wA\alpha \Rightarrow_i w\gamma\alpha \Rightarrow_i^* wxz$, что цепочки x и y совпадают в первых k позициях, но $\beta \neq \gamma$. Поэтому $A \rightarrow \beta$ и $A \rightarrow \gamma$ — различные правила из P и каждое из множеств $\text{FIRST}_k(\beta\alpha)$ и $\text{FIRST}_k(\gamma\alpha)$ содержит цепочку $\text{FIRST}_k(x)$, совпадающую с цепочкой $\text{FIRST}_k(y)$. \square

Дадим несколько приложений теоремы 5.2 к LL(1)-грамматикам. Пусть КС-грамматика $G = (N, \Sigma, P, S)$ не содержит e -правил, и нам надо узнать, является ли она LL(1)-грамматикой. Из теоремы 5.2 следует, что G будет LL(1)-грамматикой тогда и только тогда, когда для всех $A \in N$ каждое множество A -правил $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ из P таково, что множества $\text{FIRST}_1(\alpha_1), \text{FIRST}_1(\alpha_2), \dots, \text{FIRST}_1(\alpha_n)$ попарно не пересекаются. (Отсутствие e -правил здесь существенно.)

Пример 5.7. Грамматика G , состоящая из двух правил $S \rightarrow aS | a$, не будет LL(1)-грамматикой, так как $\text{FIRST}_1(aS) = \text{FIRST}_1(a) = a$. Интуитивно это можно объяснить так: видя при разборе цепочки,

начинающейся символом a , только этот первый символ, мы не знаем, какое из правил $S \rightarrow aS$ или $S \rightarrow a$ надо применить к S . С другой стороны, G — это LL(2)-грамматика. В самом деле, в обозначениях теоремы 5.2, если $S \Rightarrow_i^* wA\alpha$, то $A = S$ и $\alpha = e$. Так как для S даны только два указанных правила, то $\beta = aS$ и $\gamma = a$. Поскольку $\text{FIRST}_2(aS) = aa$ и $\text{FIRST}_2(a) = a$, то по теореме 5.2 G будет LL(2)-грамматикой. \square

Рассмотрим LL(1)-грамматики, содержащие e -правила. Здесь удобно ввести функцию FOLLOW $_k^G$.

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Определим FOLLOW $_k^G(\beta)$, где k — целое число и $\beta \in (N \cup \Sigma)^*$, как множество $\{w \mid S \Rightarrow_i^* \alpha\beta\} \cup \{w \in \text{FIRST}_k^G(\gamma) \mid \gamma \in \text{FIRST}_k^G(\beta)\}$. Как обычно, будем опускать k и G там, где понятно, о чём идет речь.

Итак, FOLLOW $_1^G(A)$ — это множество терминальных символов, которые могут встречаться непосредственно справа от A (т. е. следовать за A) в каких-нибудь выводимых цепочках, причем если αA — выводимая цепочка, то e тоже принадлежит FOLLOW $_1^G(A)$.

Можно очевидным образом расширить определение функций FIRST и FOLLOW так, чтобы их аргументами были не только отдельные цепочки, но и множества цепочек. Иначе говоря, если $G = (N, \Sigma, P, S)$ — КС-грамматика и $L \subseteq (N \cup \Sigma)^*$, то

$\text{FIRST}_k^G(L) = \{w \mid w \in \text{FIRST}_k^G(\alpha) \text{ для некоторой цепочки } \alpha \in L\}$
 $\text{FOLLOW}_k^G(L) = \{w \mid w \in \text{FOLLOW}_k^G(\alpha) \text{ для некоторой цепочки } \alpha \in L\}$

Для LL(1)-грамматик справедливо следующее важное утверждение.

Теорема 5.3. КС-грамматика $G = (N, \Sigma, P, S)$ является LL(1)-грамматикой тогда и только тогда, когда для двух различных правил $A \rightarrow \beta$ и $A \rightarrow \gamma$ пересечение $\text{FIRST}_1(\beta \text{ FOLLOW}_1(A)) \cap \text{FIRST}_1(\gamma \text{ FOLLOW}_1(A))$ пусто при всех $A \in N$.

Доказательство. Упражнение. \square

Таким образом, G является LL(1)-грамматикой тогда и только тогда, когда для каждого множества A -правил $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$

(1) множества $\text{FIRST}_1(\alpha_1), \text{FIRST}_1(\alpha_2), \dots, \text{FIRST}_1(\alpha_n)$ попарно не пересекаются,

(2) если $\alpha_i \Rightarrow^* e$, то $\text{FIRST}_1(\alpha_j) \cap \text{FOLLOW}_1(A) = \emptyset$ для $1 \leq j \leq n$, $i \neq j$.

Это просто переформулировка теоремы 5.3. Может показаться, что теорема 5.3 непосредственно обобщается на LL(k)-грамматики. Мы хотим предостеречь читателя, что это не так. КС-

грамматика G , для которой

- (5.1.1) если $A \rightarrow \beta$ и $A \rightarrow \gamma$ —различные A -правила, то
 $\text{FIRST}_k(\beta \text{ FOLLOW}_k(A)) \cap \text{FIRST}_k(\gamma \text{ FOLLOW}_k(A)) = \emptyset$

называется *сильно LL(k)-грамматикой*. Каждая LL(1)-грамматика—сильно LL(1)-грамматика, но, как показывает следующий пример, для $k > 1$ существуют LL(k)-грамматики, не являющиеся сильно LL(k)-грамматиками.

Пример 5.8. Пусть грамматика G определена правилами

$$\begin{aligned} S &\rightarrow aAaa \mid bAba \\ A &\rightarrow b \mid e \end{aligned}$$

С помощью теоремы 5.2 можно убедиться, что это LL(2)-грамматика. Возьмем вывод $S \Rightarrow aAaa$. Заметим, что $\text{FIRST}_2(baa) \cap \text{FIRST}_2(aa) = \emptyset$. В обозначениях теоремы 5.2 $\alpha = aa$, $\beta = b$ и $\gamma = e$. Аналогично, если $S \Rightarrow bAba$, то $\text{FIRST}_2(bba) \cap \text{FIRST}_2(ba) = \emptyset$. Так как все выводы в грамматике G имеют длину 2, по теореме 5.2 G будет LL(2)-грамматикой. Но $\text{FOLLOW}_2(A) = \{aa, ba\}$, так что

$$\text{FIRST}_2(b \text{ FOLLOW}_2(A)) \cap \text{FIRST}_2(\text{FOLLOW}_2(A)) = \{ba\}$$

Условие (5.1.1) нарушено, и, значит, LL(2)-грамматика G не является сильно LL(2)-грамматикой. \square

Одно из важных следствий определения LL(k)-грамматики состоит в том, что леворекурсивная грамматика не может быть LL(k)-грамматикой ни для какого k (упр. 5.1.1).

Пример 5.9. Пусть грамматика G определяется двумя правилами $S \rightarrow Sa \mid b$. Возьмем, как и в теореме 5.2, вывод $S \Rightarrow^i Sa^i$, где $i \geq 0$, $A = S$, $\alpha = e$, $\beta = Sa$ и $\gamma = b$. Тогда для $i \geq k$

$$\text{FIRST}_k(Sa^i) \cap \text{FIRST}_k(ba^i) = ba^{k-1}$$

Таким образом, G не может быть LL(k)-грамматикой ни для какого k . \square

Важно также заметить, что каждая LL(k)-грамматика однозначна (упр. 5.1.3). Поэтому, если дана неоднозначная грамматика, сразу можно сказать, что она не LL(k)-грамматика.

В гл. 8 мы покажем, что для многих детерминированных КС-языков не существует LL(k)-грамматик. Один из таких языков—язык $\{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$. Кроме того, неразрешима проблема распознавания, существует ли для данной КС-грамматики G , не являющейся LL(k)-грамматикой (k фиксировано), эквивалентная ей LL(k)-грамматика. Но несмотря на эти неприятности, есть все же ситуации, в которых к данной не LL(1)-грамматике можно применить преобразования, позво-

ляющие превратить ее в эквивалентную LL(1)-грамматику. Приведем здесь два таких преобразования.

Первое—устранение левой рекурсии. Проиллюстрируем этот прием на примере.

Пример 5.10. Пусть G —грамматика $S \rightarrow Sa \mid b$, которая, как мы видели в примере 5.9, не является LL-грамматикой. Эти два правила можно заменить тремя правилами

$$\begin{aligned} S &\rightarrow bS' \\ S' &\rightarrow aS' \mid e \end{aligned}$$

получив при этом эквивалентную грамматику G' . С помощью теоремы 5.3 легко показать, что G' —LL(1)-грамматика. \square

Другое полезное преобразование—левая факторизация (или вынесение левого множителя). Этот прием тоже проиллюстрируем на примере.

Пример 5.11. Рассмотрим LL(2)-грамматику G с двумя правилами $S \rightarrow aS \mid a$. В этих двух правилах “вынесем влево за скобки” символ a , записав их в виде $S \rightarrow a(S \mid e)$. Иными словами, мы считаем что операция конкатенации дистрибутивна относительно операции выбора альтернативы (обозначаемой вертикальной чертой). Затем заменим эти правила на

$$\begin{aligned} S &\rightarrow aA \\ A &\rightarrow S \mid e \end{aligned}$$

получив тем самым эквивалентную LL(1)-грамматику. \square

В общем случае процесс левой факторизации заключается в замене правил $A \rightarrow \alpha\beta_1 \mid \dots \mid \alpha\beta_n$ правилами $A \rightarrow \alpha A'$ и $A' \rightarrow \beta_1 \mid \dots \mid \beta_n$.

5.1.4. Разбор для LL(1)-грамматик

Ядром k -предсказывающего алгоритма разбора является управляющая таблица M . В этом и следующем разделах мы покажем, как для каждой LL(k)-грамматики G построить корректную управляющую таблицу, и таким образом докажем, что для нее возможен левый разбор с помощью k -предсказывающего алгоритма. Сначала исследуем важный частный случай, когда G —LL(1)-грамматика.

Алгоритм 5.1. Управляющая таблица для LL(1)-грамматики.

Вход. LL(1)-грамматика $G = (N, \Sigma, P, S)$.

Выход. Корректная управляющая таблица M для грамматики G .

Метод. Будем считать, что $\$$ — маркер дна магазина. Таблица M определяется на множестве $(N \cup \Sigma \cup \{\$\}) \times (\Sigma \cup \{e\})$ следующим образом:

(1) Если $A \rightarrow \alpha$ — правило из P с номером i , то $M(A, a) = (\alpha, i)$ для всех $a \neq e$, принадлежащих $\text{FIRST}_1(\alpha)$. Если $e \in \text{FIRST}_1(\alpha)$, то $M(A, b) = (\alpha, i)$ для всех $b \in \text{FOLLOW}_1(A)$.

(2) $M(a, a) = \text{выброс}$ для всех $a \in \Sigma$.

(3) $M(\$, e) = \text{допуск}$.

(4) В остальных случаях $M(X, a) = \text{ошибка}$ для $X \in N \cup \Sigma \cup \{\$\}$ и $a \in \Sigma \cup \{e\}$. \square

Прежде чем показать, что алгоритм 5.1 дает для грамматики G корректную управляющую таблицу, рассмотрим пример.

Пример 5.12. Посмотрим, как строится управляющая таблица для грамматики G с правилами

- | | |
|---------------------------|---------------------------|
| (1) $E \rightarrow TE'$ | (2) $E' \rightarrow +TE'$ |
| (3) $E' \rightarrow e$ | (4) $T \rightarrow FT'$ |
| (5) $T' \rightarrow *FT'$ | (6) $T' \rightarrow e$ |
| (7) $F \rightarrow (E)$ | (8) $F \rightarrow a$ |

С помощью теоремы 5.3 можно проверить, что G — LL(1)-грамматика. На самом деле внимательный читатель заметит, что грамматика G получена из G_0 в результате устранения ле-

	α	$($	$)$	$+$	$*$	ϵ
E	$TE', 1$	$TE', 1$				
E'			$e, 3$	$+TE', 2$		$e, 3$
T	$FT', 4$	$FT', 4$				
T'			$e, 6$	$e, 6$	$*FT', 5$	$e, 6$
F	$a, 8$	$(E), 7$				
α	выброс					
$($		выброс				
$)$			выброс			
$+$				выброс		
$*$					выброс	
$\$$						допуск

Рис. 5.4. Управляющая таблица для грамматики G .

вой рекурсии, как в примере 5.10. Кстати, G_0 не является LL-грамматикой.

На шаге (1) алгоритма 5.1 найдем элементы строки таблицы для нетерминала E . Здесь $\text{FIRST}_1[TE'] = \{(, a\}$, так что $M[E, ()] = [TE', 1]$ и $M[E, a] = [TE', 1]$. Все остальные элементы этой строки — ошибки. Вычислим теперь строку для нетерминала E' . Заметим, что $\text{FIRST}_1[+TE'] = +$, так что $M[E', +] = [+TE', 2]$. Так как есть правило $E' \rightarrow e$, мы должны вычислить $\text{FOLLOW}_1[E'] = \{e, \}\}$. Таким образом, $M[E', e] = M[E', \}] = [e, 3]$. Каждый из остальных элементов строки для E' — ошибка. Продолжая в том же духе, получим управляющую таблицу для G , показанную на рис. 5.4. Ячейки, в которых должна стоять ошибка, оставлены пустыми.

1-предсказывающий алгоритм разбора с помощью этой таблицы проанализирует входную цепочку $(a * a)$ так:

- $[(a * a), E\$, e] \vdash [(a * a), TE\$, 1]$
- $\vdash [(a * a), FT'E\$, 14]$
- $\vdash [(a * a), (E) T'E\$, 147]$
- $\vdash [a * a], E) T'E\$, 147]$
- $\vdash [a * a], TE') T'E\$, 1471]$
- $\vdash [a * a], FT'E') T'E\$, 14714]$
- $\vdash [a * a], aT'E') T'E\$, 147148]$
- $\vdash [* a], T'E') T'E\$, 147148]$
- $\vdash [* a], *FT'E') T'E\$, 1471485]$
- $\vdash [a], FT'E') T'E\$, 1471485]$
- $\vdash [a], aT'E') T'E\$, 14714858]$
- $\vdash [(), T'E') T'E\$, 14714858]$
- $\vdash [(), E') T'E\$, 147148586]$
- $\vdash [(),) T'E\$, 1471485863]$
- $\vdash [e, T'E', 1471485863]$
- $\vdash [e, E\$, 14714858636]$
- $\vdash [e, \$, 147148586363]$ \square

Теорема 5.4. Алгоритм 5.1 строит корректную управляющую таблицу для LL(1)-грамматики G .

Доказательство. Заметим сначала, что если G — LL(1)-грамматика, то на шаге (1) алгоритма 5.1 для каждого элемента $M(A, a)$ управляющей таблицы определяется не более одного значения. Это замечание — просто переформулировка теоремы 5.3.

Далее, прямой индукцией по числу шагов 1-предсказывающего алгоритма \mathcal{A} с управляющей таблицей M можно показать, что если $(xy, S\$, e) \vdash^* (y, \alpha\$, \pi)$, то $S \xrightarrow{\pi} x\alpha$. Другой индукцией

(по числу шагов левого вывода) можно доказать обратное утверждение, а именно: если $S \xrightarrow{\pi} x\alpha$, где α — незаконченная часть цепочки $x\alpha$, и $\text{FIRST}_1(y) \subseteq \text{FIRST}_1(\alpha)$, то $(xy, S\$, e) \vdash^*(y, \alpha\$, \pi)$. Из всего этого следует, что $(w, S\$, e) \vdash^*(e, \$, \pi)$ тогда и только тогда, когда $S \xrightarrow{\pi} w$. Таким образом, \mathcal{A} — корректный алгоритм разбора для грамматики G , а M — корректная управляющая таблица. \square

5.1.5. Разбор для LL(k)-грамматик

Построим теперь управляющую таблицу для произвольной LL(k)-грамматики $G = (N, \Sigma, P, S)$, где $k \geq 1$. Если G — сильно LL(k)-грамматика, то можно применить алгоритм 5.1 с аванцепочками, содержащими до k символов. Однако ситуация несколько усложняется, если G не является сильно LL(k)-грамматикой. В 1-предсказывающем алгоритме разбора для LL(1)-грамматики в магазин помещались только символы из $\Sigma \cup N$ и оказывалось, что для однозначного определения очередного применяемого правила достаточно знать нетерминальный символ наверху магазина и текущий входной символ. Но если G не является сильно LL(k)-грамматикой, то нетерминального символа и аванцепочки не всегда достаточно для однозначного определения очередного правила.

Возьмем, например, LL(2)-грамматику

$$\begin{aligned} S &\rightarrow aAaa \mid bAba \\ A &\rightarrow b \mid e \end{aligned}$$

из примера 5.8. Если даны нетерминал A и аванцепочка ba , то не известно, какое из правил $A \rightarrow b$ и $A \rightarrow e$ надо применить.

Неопределенности такого рода можно, однако, разрешить, связав с каждым нетерминалом и частью левовыводимой цепочки, которая может появиться справа от него, специальный символ, называемый LL(k)-таблицей (не надо смешивать ее с управляющей таблицей). По данной аванцепочке LL(k)-таблица будет однозначно определять, какое применить правило на первом шаге левого вывода в грамматике G .

Определение. Пусть Σ — некоторый алфавит. Если L_1 и L_2 — подмножества Σ^* , то положим

$$\begin{aligned} L_1 \oplus_k L_2 = \{w &| \text{ для некоторых } x \in L_1 \text{ и } y \in L_2 \\ &\text{либо } w = xy, \text{ если } |xy| \leq k, \\ &\text{либо } w \text{ состоит из первых } k \text{ символов} \\ &\text{цепочки } xy\} \end{aligned}$$

Пример 5.13. Пусть $L_1 = \{e, abb\}$ и $L_2 = \{b, bab\}$. Тогда $L_1 \oplus_2 L_2 = \{b, ba, ab\}$. \square

Лемма 5.1. Для любой КС-грамматики $G = (N, \Sigma, P, S)$ и любых $\alpha, \beta \in (N \cup \Sigma)^*$

$$\text{FIRST}_k^G(\alpha\beta) = \text{FIRST}_k^G(\alpha) \oplus_k \text{FIRST}_k^G(\beta)$$

Доказательство. Упражнение. \square

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Для каждого $A \in N$ и $L \subseteq \Sigma^{*k}$ определим LL(k)-таблицу $T_{A,L}$, соответствующую A и L , как функцию, значением которой для данной аванцепочки $u \in \Sigma^{*k}$ служит либо символ ошибки, либо A -правило и конечный список подмножеств множества Σ^{*k} , а именно

(1) $T_{A,L}(u) = \text{ошибка}$, если в P нет такого правила $A \rightarrow \alpha$, что $\text{FIRST}_k^G(\alpha) \oplus_k L$ содержит u ;

(2) $T_{A,L}(u) = (A \rightarrow \alpha, \langle Y_1, Y_2, \dots, Y_m \rangle)$, если $A \rightarrow \alpha$ — единственное правило из P , для которого $\text{FIRST}_k^G(\alpha) \oplus_k L$ содержит u ; если

$$\alpha = x_0 B_1 x_1 B_2 x_2 \dots B_m x_m \quad (m \geq 0)$$

где $B_i \in N$ и $x_i \in \Sigma^*$, то $Y_i = \text{FIRST}_k^G(x_i B_{i+1} x_{i+1} \dots B_m x_m) \oplus_k L$ (назовем Y_i локальным множеством для B_i ; если $m=0$, то $T_{A,L}(u) = (A \rightarrow \alpha, \emptyset)$);

(3) $T_{A,L}(u)$ не определено, если в множестве $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ найдутся два (или более) таких правила $A \rightarrow \alpha_i$ и $A \rightarrow \alpha_j$, что

$$u \in (\text{FIRST}_k^G(\alpha_i) \oplus_k L) \cap (\text{FIRST}_k^G(\alpha_j) \oplus_k L)$$

(Эту ситуацию иногда называют конфликтом между правилами $A \rightarrow \alpha_i$ и $A \rightarrow \alpha_j$. Если G — LL(k)-грамматика, то таких конфликтов не возникает.)

Интуитивно это означает, что если $T_{A,L}(u) = \text{ошибка}$, то в G невозможен вывод вида $Ax \Rightarrow^+ uv$ для $x \in L$ и $v \in \Sigma^*$. Если $T_{A,L}(u) = (A \rightarrow \alpha, \langle Y_1, Y_2, \dots, Y_m \rangle)$, то найдется в точности одно правило $A \rightarrow \alpha$, которое можно применить на первом шаге вывода $Ax \Rightarrow^+ uv$ для $x \in L$ и $v \in \Sigma^*$. Каждое множество Y_i дает всевозможные префиксы длины, не большей k , терминальных цепочек, которые могут следовать за цепочкой, выводимой из B_i , когда в выводе вида $Ax \Rightarrow_l ax \Rightarrow^* uv$, где $x \in L$, применяется правило $A \rightarrow \alpha$, где $\alpha = x_0 B_1 x_1 B_2 x_2 \dots B_m x_m$.

По теореме 5.2 $G = (N, \Sigma, P, S)$ не является LL(k)-грамматикой тогда и только тогда, когда существует такая цепочка $\alpha \in (N \cup \Sigma)^*$, что

$$(1) S \xrightarrow{l} wA\alpha,$$

(2) $\text{FIRST}_k^G(\beta\alpha) \cap \text{FIRST}_k^G(\gamma\alpha) \neq \emptyset$ для некоторых $\beta \neq \gamma$, для которых $A \rightarrow \beta$ и $A \rightarrow \gamma$ принадлежат P .

В силу леммы 5.1 условие (2) можно перефразировать так:

(2') если $L = \text{FIRST}_k^G(\alpha)$, то пересечение $\text{FIRST}_k^G(\beta) \oplus_k L$ и $\text{FIRST}_k^G(\gamma) \oplus_k L$ непусто.

Следовательно, если G — LL(k)-грамматика и есть вывод $S \Rightarrow^* wA\alpha \Rightarrow^* wx$, то $T_{A,L}(u)$ будет однозначно определять, какое правило применить к A , если $u = \text{FIRST}_k(x)$ и $L = \text{FIRST}_k(\alpha)$.

Пример 5.14. Рассмотрим LL(2)-грамматику

$$\begin{aligned} S &\rightarrow aAaa \mid bAb \\ A &\rightarrow b \mid e \end{aligned}$$

Вычислим LL(2)-таблицу $T_{S,\{e\}}$, которую обозначим T_0 . Для правила $S \rightarrow aAaa$ найдем $\text{FIRST}_2(aAaa) \oplus_2 \{e\} = \{aa, ab\}$. Для $S \rightarrow bAb$ вычислим $\text{FIRST}_2(bAb) \oplus_2 \{e\} = \{bb\}$. Таким образом, $T_0(aa) = (S \rightarrow aAaa, Y)$, где $Y = \text{FIRST}_2(aa) \oplus_2 \{e\} = \{aa\}$ — локальное множество для A и aa — цепочка, расположенная справа от A в правиле $S \rightarrow aAaa$. Продолжая в том же духе, получаем $T_0(bb) = (S \rightarrow bAb, Y)$, где $Y = \text{FIRST}_2(bb) \oplus_2 \{e\} = \{bb\}$. Для каждой цепочки $u \in (a+b)^{*2}$, не показанной в этой таблице, $T_0(u) = \text{ошибка}$. \square

Таблица 5.1
 T_0

u	Правило	Множества
aa	$S \rightarrow aAaa$	{aa}
ab	$S \rightarrow aAaa$	{aa}
bb	$S \rightarrow bAb$	{bb}

Теперь изложим алгоритм для LL(k)-грамматики G , вычисляющий LL(k)-таблицы, необходимые для построения управляющей таблицы. Следует заметить, что если G — LL(1)-грамматика, то этот алгоритм может выдать более чем по одной таблице на нетерминал. Однако анализаторы, строящиеся алгоритмами 5.1 и 5.2, очень похожи. На входных цепочках, принадлежащих языку, определяемому грамматикой, они, конечно, работают одинаково. На других входных цепочках после того, как анализатор алгоритма 5.2 обнаружит ошибку, анализатор алгоритма 5.1 сделает, возможно, еще несколько шагов.

Алгоритм 5.2. Построение LL(k)-таблиц.

Вход. LL(k)-грамматика $G = (N, \Sigma, P, S)$.

Выход. Множество \mathcal{T} LL(k)-таблиц, необходимых для построения управляющей таблицы для G .

Метод.

- (1) Построить LL(k)-таблицу T_0 , соответствующую S и $\{e\}$.
- (2) Положить вначале $\mathcal{T} = \{T_0\}$.
- (3) Для каждой LL(k)-таблицы $T \in \mathcal{T}$, содержащей элемент $T(u) = (A \rightarrow x_0B_1x_1B_2x_2 \dots B_mx_m, \langle Y_1, Y_2, \dots, Y_m \rangle)$

$$T(u) = (A \rightarrow x_0B_1x_1B_2x_2 \dots B_mx_m, \langle Y_1, Y_2, \dots, Y_m \rangle)$$

включить в \mathcal{T} LL(k)-таблицу T_{B_i,Y_i} для $1 \leq i \leq m$, если T_{B_i,Y_i} еще не входит в \mathcal{T} .

(4) Повторять шаг (3) до тех пор, пока в \mathcal{T} можно включать новые таблицы. \square

Пример 5.15. Построим соответствующее множество LL(2)-таблиц для грамматики

$$\begin{aligned} S &\rightarrow aAaa \mid bAb \\ A &\rightarrow b \mid e \end{aligned}$$

Начнем с $\mathcal{T} = \{T_{S,\{e\}}\}$. Так как $T_{S,\{e\}}(aa) = (S \rightarrow aAaa, \{aa\})$, то в \mathcal{T} надо включить $T_{A,\{aa\}}$. Аналогично, так как $T_0(bb) = (S \rightarrow bAb, \{bb\})$, то в \mathcal{T} нужно также включить $T_{A,\{bb\}}$. (Элементы LL(2)-таблиц $T_{A,\{aa\}}$ и $T_{A,\{bb\}}$, отличные от значения ошибки, приведены в табл. 5.2.) Сейчас $\mathcal{T} = \{T_{S,\{e\}}, T_{A,\{aa\}}, T_{A,\{bb\}}\}$, и алгоритм 5.2 уже не может включить в \mathcal{T} новых таблиц, так что эти три LL(2)-таблицы образуют множество, соответствующее грамматике G . \square

Дадим алгоритм, которым можно построить корректную управляющую таблицу для LL(k)-грамматики G по соответствующему ей множеству LL(k)-таблиц. Управляемый этой таблицей k -предсказывающий алгоритм будет в качестве магазинных символов употреблять вместо нетерминалов LL(k)-таблицы.

Таблица 5.2

$T_{A,\{aa\}}$			$T_{A,\{ba\}}$		
u	Правило	Множества	u	Правило	Множества
ba	$A \rightarrow b$	—	ba	$A \rightarrow e$	—
aa	$A \rightarrow e$	—	bb	$A \rightarrow b$	—

Алгоритм 5.3. Управляющая таблица для LL(k)-грамматики.

Вход. LL(k)-грамматика $G = (N, \Sigma, P, S)$ и соответствующее ей множество \mathcal{T} LL(k)-таблиц.

Выход. Корректная управляющая таблица M для G .

Метод. M определяется на множестве $(\mathcal{T} \cup \Sigma \cup \{\$\}) \times \Sigma^{*k}$ следующим образом:

(1) Если $A \rightarrow x_0B_1x_1B_2x_2 \dots B_mx_m$ — правило из P с номером i и $T_{A,L} \in \mathcal{T}$, то для всех u , для которых

$$T_{A,L}(u) = (A \rightarrow x_0B_1x_1B_2x_2 \dots B_mx_m, \langle Y_1, Y_2, \dots, Y_m \rangle)$$

полагаем $M(T_{A,L}, u) = (x_0T_{B_1,Y_1}x_1T_{B_2,Y_2}x_2 \dots T_{B_m,Y_m}x_m, i)$.

- (2) $M(a, av) = \text{выброс}$ для всех $v \in \Sigma^{*(k-1)}$.
- (3) $M(\$, e) = \text{допуск}$.
- (4) В остальных случаях $M(X, u) = \text{ошибка}$.
- (5) $T_{S, \{\epsilon\}}$ — начальная таблица. \square

Пример 5.16. Построим управляющую таблицу для LL(2)-грамматики

- (1) $S \rightarrow aAaa$
- (2) $S \rightarrow bAb$
- (3) $A \rightarrow b$
- (4) $A \rightarrow e$

используя соответствующее ей множество LL(2)-таблиц, найденное в примере 5.15. Алгоритм 5.3 выдаст управляющую таблицу,

	aa	ab	a	ba	bb	b	e
T_0	$aT_1aa, 1$	$aT_1aa, 1$			$0T_2ba, 2$		
T_1	$\epsilon, 4$			$b, 3$			
T_2				$\epsilon, 4$	$b, 3$		
α	выброс	выброс	выброс				
b				выброс	выброс	выброс	
$\$$							допуск

Рис. 5.5. Управляющая таблица.

изображенную на рис. 5.5, где $T_0 = T_{S, \{\epsilon\}}$, $T_1 = T_{A, \{aa\}}$ и $T_2 = T_{A, \{ba\}}$. Подразумевается, что в пустых ячейках **ошибка**.

Для входной цепочки bba 2-предсказывающий алгоритм проделает такую последовательность тиков:

$$\begin{aligned}
 (bba, T_0\$, e) &\vdash (bba, bT_2ba\$, 2) \\
 &\vdash (ba, T_2ba\$, 2) \\
 &\vdash (ba, ba\$, 24) \\
 &\vdash (a, a\$, 24) \\
 &\vdash (e, \$, 24) \quad \square
 \end{aligned}$$

Теорема 5.5. Алгоритм 5.3 строит для LL(k)-грамматики $G = (\mathcal{N}, \Sigma, P, S)$ корректную таблицу, управляющую работой соответствующего k-предсказывающего алгоритма разбора.

Доказательство. Доказательство аналогично доказательству теоремы 5.4. При построении LL(k)-таблиц для произ-

вольной КС-грамматики могут возникать конфликты, упомянутые при определении таблицы $T_{A, L}$ (пункт (3)). Но если $G = \text{LL}(k)$ -грамматика, то конфликтов не будет, так как если $A \rightarrow \beta$ и $A \rightarrow \gamma$ принадлежат P и $S \Rightarrow_i^* wA\alpha$, то

$$(FIRST_k(\beta) \oplus_k FIRST_k(\alpha)) \cap (FIRST_k(\gamma) \oplus_k FIRST_k(\alpha)) = \emptyset$$

При построении LL(k)-таблиц, соответствующих грамматике G , таблица $T_{A, L}$ вычисляется только тогда, когда $S \Rightarrow_i^* wA\alpha$ и $L = FIRST_k(\alpha)$ для некоторой цепочки α , т. е. L — локальное множество для A . Поэтому если $u \in L$, то существует не более одного правила $A \rightarrow \beta$, для которого $u \in FIRST_k(\beta) \oplus_k L$.

Зададим гомоморфизм h на множестве $\mathcal{T} \cup \Sigma$ равенствами $h(a) = a$ для всех $a \in \Sigma$ и $h(T) = A$, если T — LL(k)-таблица, соответствующая A и L .

Заметим, что у каждой таблицы из \mathcal{T} есть хотя бы один элемент, содержащий правило, и A однозначно определяется таблицей T .

Теперь докажем, что

- (5.1.2) $S \Rightarrow x\alpha$ тогда и только тогда, когда найдется такая цепочка $\alpha' \in (\mathcal{T} \cup \Sigma)^*$, что $h(\alpha') = \alpha$ и $(xy, T_0\$, e) \vdash^* (y, \alpha'\$, \pi)$ для всех y , для которых $\alpha' \Rightarrow^* y$, где T_0 — LL(k)-таблица, соответствующая S и $\{\epsilon\}$.

Достаточность. Из способа построения управляющей таблицы следует, что когда выдается номер i правила $A \rightarrow \beta$, алгоритм разбора заменяет таблицу T , для которой $h(T) = A$, такой цепочкой β' , что $h(\beta') = \beta$. Таким образом, эту часть утверждения (5.1.2) можно доказать прямой индукцией по числу тиков работы алгоритма разбора.

Необходимость. Здесь мы покажем, что

- (5.1.3) если $A \Rightarrow_i^* x$, то алгоритм разбора \vdash проделает последовательность тиков $(xy, T, e) \vdash^* (y, e, \pi)$ для любой LL(k)-таблицы T , соответствующей A и L , где $L = FIRST_k(\alpha)$ для некоторой цепочки α , для которой $S \Rightarrow_i^* wA\alpha$, и $y \in L$.

Доказательство проведем индукцией по длине цепочки π . Если $A \Rightarrow_i^* a_1a_2\dots a_n$, то

$$(a_1a_2\dots a_ny, T, e) \vdash (a_1\dots a_ny, a_1\dots a_n, i)$$

поскольку $T(u) = (A \rightarrow a_1a_2\dots a_n, \emptyset)$ для всех $u \in FIRST_k(a_1a_2\dots a_n) \oplus_k L$. Тогда $(a_1\dots a_ny, a_1\dots a_n, i) \vdash^n (y, e, i)$. Теперь предположим, что утверждение (5.1.3) верно для всех левых выводов длины, не большей l , и пусть $A \Rightarrow_i^* x_0B_1x_1B_2x_2\dots B_mx_m$ и $B_j \pi_j \Rightarrow y_j$, где $|\pi_j| < l$. Тогда $(x_0y_1x_1\dots y_mx_my, T, e) \vdash^* (x_0y_1x_1\dots y_mx_my, x_0T_1x_1\dots T_mx_m, i)$, так как $T(u) = (A \rightarrow x_0B_1x_1\dots B_mx_m, \emptyset)$.

$\langle Y_1, \dots, Y_m \rangle$ для всех $u \in \text{FIRST}_k(x_0B_1x_1\dots B_mx_m) \oplus_k L$. Каждая таблица T_j — это LL(k)-таблица, соответствующая B_j и Y_j ($1 \leq j \leq m$), так что предположение индукции выполняется для каждой последовательности тактов вида

$$(y_jx_j\dots y_mx_my, T_j, e) \vdash^* (x_j\dots y_mx_my, e, \pi_j)$$

Вставим такты, на которых из магазина выбрасываются x_j ; тогда $(x_0y_1x_1y_2x_2\dots y_mx_my, T, e)$

$$\begin{aligned} &\vdash (x_0y_1x_1y_2x_2\dots y_mx_my, x_0T_1x_1T_2x_2\dots T_mx_m, t) \\ &\vdash^* (y_1x_1y_2x_2\dots y_mx_my, T_1x_1T_2x_2\dots T_mx_m, i) \\ &\vdash^* (x_1y_2x_2\dots y_mx_my, x_1T_2x_2\dots T_mx_m, i\pi_1) \\ &\vdash^* (y_2x_2\dots y_mx_my, T_2x_2\dots T_mx_m, i\pi_1) \\ &\cdot \\ &\cdot \\ &\cdot \\ &\vdash^* (y, e, i\pi_1\pi_2\dots\pi_m) \end{aligned}$$

Из утверждения (5.1.3), в частности, вытекает, что если $S_\pi \Rightarrow w$, то $(w, T_0\$, e) \vdash^* (e, \$, \pi)$. \square

Пример 5.17. Рассмотрим LL(2)-грамматику G_2 с правилами

- (1) $S \rightarrow e$
- (2) $S \rightarrow abA$
- (3) $A \rightarrow Saa$
- (4) $A \rightarrow b$

Построим сначала соответствующие LL(2)-таблицы. Начнем с $T_0 = T_{S, \{e\}}$ (см. табл. 5.3). Затем по T_0 найдем $T_1 = T_{A, \{e\}}$, потом $T_2 = T_{S, \{aa\}}$ и, наконец, $T_3 = T_{A, \{aa\}}$. По этим LL(2)-таблицам получим управляющую таблицу, показанную на рис. 5.6.

2-предсказывающий алгоритм, управляемый этой таблицей, разберет входную цепочку $abaa$, проделав такую последовательность тактов:

$$\begin{aligned} (abaa, T_0\$, e) &\vdash (abaa, abT_1\$, 2) \\ &\vdash (baa, bT_1\$, 2) \\ &\vdash (aa, T_1\$, 2) \\ &\vdash (aa, T_2aa\$, 23) \\ &\vdash (aa, aa\$, 231) \\ &\vdash (a, a\$, 231) \\ &\vdash (e, \$, 231) \quad \square \end{aligned}$$

В заключение этого раздела покажем, что k -предсказывающий алгоритм разбирает каждую входную цепочку за линейное время.

Таблица 5.3

T_0			T_2		
u	Правило	Множества	u	Правило	Множества
e	$S \rightarrow e$	$\overline{\{e\}}$	aa	$S \rightarrow e$	$\overline{\{aa\}}$
ab	$S \rightarrow abA$	$\{e\}$	ab	$S \rightarrow abA$	$\{aa\}$
T_1			T_3		
u	Правило	Множества	u	Правило	Множества
b	$A \rightarrow b$	$\overline{\{b\}}$	aa	$A \rightarrow Saa$	$\{aa\}$
aa	$A \rightarrow Saa$	$\{aa\}$	ab	$A \rightarrow Saa$	$\{aa\}$
ab	$A \rightarrow Saa$	$\{aa\}$	ba	$A \rightarrow b$	$\overline{\{b\}}$

Теорема 5.6. Число шагов, выполняемых k -предсказывающим алгоритмом с управляющей таблицей, построенной алгоритмом 5.3 по LL(k)-грамматике G , линейно зависит от n , где n — длина входной цепочки.

Доказательство. LL(k)-грамматика G не может быть леворекурсивной. По лемме 4.1 максимальное число шагов в выводе вида $A \Rightarrow^* B\alpha$ меньше некоторой константы c . Таким образом, максимальное число шагов, которое может потратить k -предсказывающий алгоритм \mathcal{A} на обработку одного входного символа вплоть до операции выброса, сдвигающей с этого сим-

	aa	ab	a	ba	bb	b	e
T_0	$abT_1, 2$						$e, 1$
T_1	$T_2aa, 3$	$T_2aa, 3$				$b, 4$	
T_2	$e, 1$	$abT_3, 2$					
T_3	$T_2aa, 3$	$T_2aa, 3$		$b, 4$			
a	выброс	выброс	выброс				
b				выброс	выброс	выброс	
$\$$							допуск

Рис. 5.6. Управляющая таблица для грамматики G_2 .

вала входную головку, не превосходит c . Следовательно, при обработке входной цепочки длины n алгоритм \mathcal{A} может проделать не более $O(n)$ шагов. \square

5.1.6. Проверка LL(k)-условия

По отношению к произвольной данной грамматике G возникает ряд естественных вопросов. Во-первых, является ли G LL(k)-грамматикой для данного k ? Во-вторых, является ли G LL-грамматикой, т. е. существует ли такое k , что G — LL(k)-грамматика? И наконец, так как для LL(1)-грамматики левые разборы строятся особенно просто, естественно спросить: если G не LL(1)-грамматика, то существует ли эквивалентная ей LL(1)-грамматика G' , для которой $L(G') = L(G)$?

К сожалению, только для первого вопроса есть отвечающий на него алгоритм. Можно показать, что вторая и третья проблемы алгоритмически неразрешимы.

В этом разделе мы опишем алгоритм, проверяющий, является ли G LL(k)-грамматикой для заданного значения k . Если $k=1$, можно воспользоваться теоремой 5.3. Для произвольного k можно применить теорему 5.2. Здесь мы рассмотрим общий случай. По существу надо просто показать, что алгоритм 5.3 успешно строит управляющую таблицу только тогда, когда G — LL(k)-грамматика.

Напомним, что $G=(N, \Sigma, P, S)$ не является LL(k)-граммикой тогда и только тогда, когда для некоторой цепочки $\alpha \in (N \cup \Sigma)^*$

- (1) $S \Rightarrow_i^* wA\alpha$,
- (2) $L = \text{FIRST}_k(\alpha)$,

(3) $(\text{FIRST}_k(\beta) \oplus_k L) \cap (\text{FIRST}_k(\gamma) \oplus_k L) \neq \emptyset$ для некоторых $\beta \neq \gamma$, для которых $A \rightarrow \beta$ и $A \rightarrow \gamma$ — правила из P .

Алгоритм 5.4. Проверка LL(k)-условия.

Вход. КС-грамматика $G=(N, \Sigma, P, S)$ и целое число k .

Выход. „Да“, если G — LL(k)-грамматика, и „нет“ в противном случае.

Метод.

(1) Для каждого нетерминала $A \in N$, имеющего две или более альтернативы, вычислить $\sigma(A) = \{L \mid L \subseteq \Sigma^{*k}\}$ и существует такая цепочка α , что $S \Rightarrow_i^* wA\alpha$ и $L = \text{FIRST}_k(\alpha)\}$. (В дальнейшем будет дан алгоритм вычисления $\sigma(A)$.)

(2) Если $A \rightarrow \beta$ и $A \rightarrow \gamma$ — различные A -правила, то для каждого $L \in \sigma(A)$ вычислить $f(L) = (\text{FIRST}_k(\beta) \oplus_k L) \cap (\text{FIRST}_k(\gamma) \oplus_k L)$. Если $f(L) \neq \emptyset$, остановиться и выдать „нет“. Если $f(L) = \emptyset$ для

всех $L \in \sigma(A)$, повторить шаг (2) для всех различных пар A -правил.

(3) Повторить шаги (1) и (2) для всех нетерминалов из N .

(4) Выдать „да“, если нарушений LL(k)-условия не обнаружено. \square

Чтобы реализовать алгоритм 5.4, нужно уметь вычислять $\text{FIRST}_k(\beta)$ для любой КС-грамматики $G=(N, \Sigma, P, S)$ и всех $\beta \in (N \cup \Sigma)^*$. Во-вторых, нужно уметь находить множества $\sigma(A)$. Сейчас мы дадим соответствующие алгоритмы.

Алгоритм 5.5. Вычисление функции $\text{FIRST}_k(\beta)$.

Вход. КС-грамматика $G=(N, \Sigma, P, S)$ и цепочка $\beta = X_1X_2 \dots \dots X_n \in (N \cup \Sigma)^*$.

Выход. $\text{FIRST}_k(\beta)$.

Метод. Так как по лемме 5.1

$$\text{FIRST}_k(\beta) = \text{FIRST}_k(X_1) \oplus_k \text{FIRST}_k(X_2) \oplus_k \dots \oplus_k \text{FIRST}_k(X_n)$$

то достаточно показать, как найти $\text{FIRST}_k(X)$ для $X \in N$. Если $X \in \Sigma \cup \{e\}$, то очевидно, что $\text{FIRST}_k(X) = \{X\}$.

Определим множества $F_i(X)$ для каждого $X \in N \cup \Sigma$ и возрастающих значений $i \geq 0$:

- (1) $F_i(a) = \{a\}$ для всех $a \in \Sigma$ и $i \geq 0$.
- (2) $F_0(A) = \{x \mid x \in \Sigma^{*k}$ и существует правило $A \rightarrow x\alpha$ из P , для которого либо $|x|=k$, либо $|x| < k$ и $\alpha=e\}$.

(3) Допустим, что F_0, F_1, \dots, F_{i-1} уже определены для всех $A \in N$. Тогда

$$F_i(A) = F_{i-1}(A) \cup \{x \mid A \rightarrow Y_1 \dots Y_n \text{ принадлежит } P \text{ и } x \in F_{i-1}(Y_1) \oplus_k F_{i-1}(Y_2) \oplus_k \dots \oplus_k F_{i-1}(Y_n)\}$$

(4) Так как $F_{i-1}(A) \subseteq F_i(A) \subseteq \Sigma^{*k}$ для всех A и i , то в конце концов мы дойдем до такого i , что $F_{i-1}(A) = F_i(A)$ для всех $A \in N$. Тогда положим $\text{FIRST}_k(A) = F_i(A)$ для этого значения i . \square

Пример 5.18. Построим множества $F_i(X)$ для $k=1$ и грамматики G с правилами

$$\begin{aligned} S &\rightarrow BA \\ A &\rightarrow +BA \mid e \\ B &\rightarrow DC \\ C &\rightarrow *DC \mid e \\ D &\rightarrow (S) \mid a \end{aligned}$$

Вначале

$$\begin{aligned} F_0(S) &= F_0(B) = \emptyset \\ F_0(A) &= \{+, e\} \\ F_0(C) &= \{*, e\} \\ F_0(D) &= \{(, a\} \end{aligned}$$

Затем $F_1(B) = \{(, a\}$ и $F_1(X) = F_0(X)$ для всех остальных X . Далее $F_2(S) = \{(, a\}$ и $F_2(X) = F_1(X)$ для всех остальных X . Так как $F_3(X) = F_2(X)$ для всех X , то

$$\text{FIRST}(S) = \text{FIRST}(B) = \text{FIRST}(D) = \{(, a\}$$

$$\text{FIRST}(A) = \{-, e\}$$

$$\text{FIRST}(C) = \{*, e\} \quad \square$$

Теорема 5.7. Алгоритм 5.5 правильно вычисляет $\text{FIRST}_k(A)$.

Доказательство. Заметим, что если $F_{i-1}(X) = F_i(X)$ для всех $X \in N \cup \Sigma$, то $F_i(X) = F_j(X)$ для всех $j > i$ и всех X . Таким образом, мы должны доказать, что $x \in \text{FIRST}_k(A)$ тогда и только тогда, когда $x \in F_j(A)$ для некоторого j .

Достаточность. Индукцией по j покажем, что $F_j(A) \subseteq \text{FIRST}_k(A)$. Базис, $j=0$, тривиален. Рассмотрим фиксированное значение j и допустим, что наше утверждение верно для всех меньших значений j .

Если $x \in F_j(A)$, то либо $x \in F_{j-1}(A)$ (и в этом случае сразу получается нужный результат), либо в P можно найти такое правило $A \rightarrow Y_1 \dots Y_n$, что $x_p \in F_{j-1}(Y_p)$ для $1 \leq p \leq n$ и $x = \text{FIRST}_k(x_1 \dots x_n)$. По предположению индукции $x_p \in \text{FIRST}_k(Y_p)$. Таким образом, для каждого p существует такой вывод $Y_p \Rightarrow^* y_p$, что $x_p = \text{FIRST}_k(y_p)$. Следовательно, $A \Rightarrow^* y_1 \dots y_n$. Теперь надо показать, что $x = \text{FIRST}_k(y_1 \dots y_n)$, и заключить отсюда, что $x \in \text{FIRST}_k(A)$.

Случай 1: $|x_1 \dots x_n| < k$. Тогда $y_p = x_p$ для каждого p , и $x = y_1 \dots y_n$. Так как в этом случае $y_1 \dots y_n \in \text{FIRST}_k(A)$, то $x \in \text{FIRST}_k(A)$.

Случай 2: $|x_1 \dots x_s| < k$ для некоторого $s \geq 0$, но $|x_1 \dots x_{s+1}| \geq k$. Тогда $y_p = x_p$ для $1 \leq p \leq s$ и x состоит из первых k символов цепочки $x_1 \dots x_{s+1}$. Так как x_{s+1} — префикс цепочки y_{s+1} , то x — префикс цепочки $y_1 \dots y_{s+1}$, а значит, и цепочки $y_1 \dots y_n$. Итак, $x \in \text{FIRST}_k(A)$.

Необходимость. Пусть $x \in \text{FIRST}_k(A)$. Тогда $A \Rightarrow^* y$ для некоторого r и $x = \text{FIRST}_k(y)$. Докажем индукцией по r , что $x \in F_r(A)$. Базис, $r=1$, тривиален, так как $x \in F_0(A)$. (На самом деле предположение индукции можно было бы несколько усилить, но здесь это не к чему.)

Зафиксируем r и предположим, что наше утверждение верно для меньших значений r . Тогда

$$A \Rightarrow Y_1 \dots Y_n \Rightarrow^{r-1} y$$

где $y = y_1 \dots y_n$ и $Y_p \Rightarrow^r y_p$ для $1 \leq p \leq n$. Очевидно, $r_p < r$. По предположению индукции $x_p \in F_{r-1}(Y_p)$, где $x_p = \text{FIRST}_k(y_p)$. Таким образом, $\text{FIRST}_k(x_1 \dots x_n) = x \in F_r(A)$. \square

Следующий алгоритм представляет собой метод вычисления для заданной грамматики $G = (N, \Sigma, P, S)$ тех множеств L из Σ^{*k} , для которых существуют такие w, A и α , что $S \Rightarrow^*_i wA\alpha$ и $\text{FIRST}_k(\alpha) = L$.

Алгоритм 5.6. Вычисление $\sigma(A)$.

Вход. КС-грамматика $G = (N, \Sigma, P, S)$.

Выход. $\sigma(A) = \{L \mid L \subseteq \Sigma^{*k} \text{ и } S \Rightarrow^*_i wA\alpha, \text{ FIRST}_k(\alpha) = L \text{ для некоторых } w \text{ и } \alpha\}$.

Метод. Будем вычислять для всех A и B из N множества $\sigma(A, B) = \{L \mid L \subseteq \Sigma^{*k} \text{ и } A \Rightarrow^*_i xB\alpha, L = \text{FIRST}_k(\alpha) \text{ для некоторых } x \text{ и } \alpha\}$. С этой целью будем строить множества $\sigma_i(A, B)$ для $i = 0, 1, \dots$:

(1) Положим $\sigma_0(A, B) = \{L \mid L \subseteq \Sigma^{*k}, A \rightarrow \beta B\alpha \text{ принадлежит } P \text{ и } L = \text{FIRST}_k(\alpha)\}$.

(2) Допустим, что $\sigma_{i-1}(A, B)$ уже построены для всех A и B . Определим $\sigma_i(A, B)$ следующим образом:

(a) Если $L \in \sigma_{i-1}(A, B)$, то поместим L в $\sigma_i(A, B)$.

(b) Если в P есть правило $A \rightarrow X_1 \dots X_n$, то в $\sigma_i(A, B)$ поместим такой язык L , для которого найдутся такие $1 \leq j \leq n$ и $L' \in \sigma_{i-1}(X_j, B)$, что $L = L' \bigoplus_k \text{FIRST}_k(X_{j+1}) \bigoplus_k \dots \bigoplus_k \text{FIRST}_k(X_n)$.

(3) В тот момент, когда $\sigma_i(A, B) = \sigma_{i-1}(A, B)$ для некоторого i и для всех A, B , положим $\sigma(A, B) = \sigma_i(A, B)$. Так как $\sigma_{i-1}(A, B) \subseteq \sigma_i(A, B) \subseteq \mathcal{P}(\Sigma^{*k})$ для всех i , то такое i должно найтись.

(4) Нужное нам множество $\sigma(A)$ есть $\sigma(S, A)$. \square

Теорема 5.8. L принадлежит множеству $\sigma(S, A)$ построенному алгоритмом 5.6, тогда и только тогда, когда $S \Rightarrow^*_i wA\alpha$ и $L = \text{FIRST}_k(\alpha)$ для некоторых $w \in \Sigma^*$ и $\alpha \in (N \cup \Sigma)^*$.

Доказательство. Доказательство аналогично доказательству предыдущей теоремы; оставляем его в качестве упражнения. \square

Пример 5.19. Проверим, выполняется ли LL(1)-условие для грамматики G с правилами

$$S \rightarrow AS | e$$

$$A \rightarrow aA | b$$

Начнем с вычисления $\text{FIRST}_1(S) = \{e, a, b\}$ и $\text{FIRST}_1(A) = \{a, b\}$. Затем нужно вычислить $\sigma(S) = \sigma(S, S)$ и $\sigma(A) = \sigma(S, A)$. На шаге (1) алгоритма 5.4 получаем

$$\begin{array}{ll} \sigma_0(S, S) = \{\{e\}\} & \sigma_0(S, A) = \{\{e, a, b\}\} \\ \sigma_0(A, S) = \emptyset & \sigma_0(A, A) = \{\{e\}\} \end{array}$$

На шаге (2) к этим множествам добавить нечего. Например, так как есть правило $S \rightarrow AS$ и $\sigma_0(A, A)$ содержит $\{e\}$, то на шаге (2б) надо добавить к $\sigma_1(S, A)$ множество $L = \{e\} \oplus_1 \text{FIRST}_1(S) = \{e, a, b\}$. Но $\sigma_1(S, A)$ уже содержит множество $\{e, a, b\}$, так как его содержит $\sigma_0(S, A)$.

Таким образом, $\sigma(A) = \{\{e, a, b\}\}$ и $\sigma(S) = \{\{e\}\}$. Чтобы проверить, что G — LL(1)-грамматика, нужно убедиться в том, что $(\text{FIRST}_1(AS) \oplus_1 \{e\}) \cap (\text{FIRST}_1(e) \oplus_1 \{e\}) = \emptyset$. (Это делается для двух S -правил и единственного элемента $\{e\}$ множества $\sigma(S, S)$.) Так как

$$\text{FIRST}_1(AS) = \text{FIRST}_1(A) \oplus_1 \text{FIRST}_1(S) = \{a, b\}$$

и $\text{FIRST}_1(e) = \{e\}$, то действительно $\{a, b\} \cap \{e\} = \emptyset$.

Для двух A -правил нужно показать, что

$$(\text{FIRST}_1(aA) \oplus_1 \{e, a, b\}) \cap (\text{FIRST}_1(bA) \oplus_1 \{e, a, b\}) = \emptyset$$

Это сводится к проверке равенства $\{a\} \cap \{b\} = \emptyset$, которое, очевидно, верно. Итак, G — это LL(1)-грамматика. \square

УПРАЖНЕНИЯ

5.1.1. Покажите, что если грамматика G леворекурсивна, то она не LL-грамматика.

5.1.2. Покажите, что если грамматика G содержит два правила $A \rightarrow \alpha\alpha \mid \alpha\beta$, где $\alpha \neq \beta$, то она не может быть LL(1)-грамматикой.

5.1.3. Покажите, что каждая LL-грамматика однозначна.

5.1.4. Покажите, что каждая грамматика, удовлетворяющая условию (5.1.1), является LL(k)-грамматикой.

5.1.5. Покажите, что грамматика с правилами

$$\begin{aligned} S &\rightarrow aAaB \mid bAbB \\ A &\rightarrow a \mid ab \\ B &\rightarrow aB \mid a \end{aligned}$$

является LL(3)-грамматикой, но не LL(2)-грамматикой.

5.1.6. Постройте LL(3)-таблицы для грамматики из упр. 5.1.5.

5.1.7. Постройте детерминированный левый анализатор для грамматики из примера 5.17.

5.1.8. Дайте алгоритм вычисления $\text{FOLLOW}_k^G(A)$ для нетерминала A .

***5.1.9.** Покажите, что каждое регулярное множество порождается LL(1)-грамматикой.

5.1.10. Покажите, что $G = (N, \Sigma, P, S)$ является LL(1)-граммикой тогда и только тогда, когда для каждого множества A -правил $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$

(1) $\text{FIRST}_1^G(\alpha_i) \cap \text{FIRST}_1^G(\alpha_j) = \emptyset$ для $i \neq j$,

(2) если $\alpha_i \Rightarrow^* e$, то $\text{FIRST}_1^G(\alpha_j) \cap \text{FOLLOW}_1^G(A) = \emptyset$ для $1 \leq i \leq n$, $i \neq j$ (обратите внимание, что e может выводиться не более чем из одной цепочки α_i).

****5.1.11.** Докажите неразрешимость проблемы: существует ли такое число k , что КС-грамматика G является LL(k)-граммикой? (В противоположность этому, если произвольное число k фиксировано, узнать, является ли G LL(k)-граммикой для этого определенного значения k , можно.)

***5.1.12.** Докажите неразрешимость проблемы: порождает ли КС-грамматика G LL-язык?

***5.1.13.** LL(k)-граммику часто определяют так. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Если $S \Rightarrow^* wAx$ для некоторых $w, x \in \Sigma^*$ и $A \in N$, то для каждой цепочки $y \in \Sigma^{*k}$ существует не более одного такого правила $A \rightarrow \alpha$, что $y \in \text{FIRST}_k(\alpha x)$. Покажите, что это определение эквивалентно определению в разд. 5.1.1.

5.1.14. Дополните доказательство теоремы 5.4.

5.1.15. Докажите лемму 5.1.

5.1.16. Докажите теорему 5.8.

****5.1.17.** Покажите, что если L является LL(k)-языком, то он определяется LL(k)-граммикой в нормальной форме Хомского.

5.1.18. Покажите, что LL(0)-язык содержит не более одной цепочки.

5.1.19. Покажите, что грамматика G с правилами $S \rightarrow aaSbb \mid a \mid e$ является LL(2)-граммикой. Найдите эквивалентную ей LL(1)-граммику.

****5.1.20.** Покажите, что $\{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$ не LL-язык.

Упражнения 5.1.21—5.1.24 будут решены в гл. 8. Возможно, читатель захочет сделать их сейчас.

****5.1.21.** Докажите разрешимость проблемы: для двух LL(k)-граммик G_1 и G_2 выяснить, верно ли, что $L(G_1) = L(G_2)$.

***5.1.22.** Покажите, что для каждого $k \geq 0$ существуют LL($k+1$)-языки, которые не являются LL(k)-языками.

****5.1.23.** Покажите, что каждый $LL(k)$ -язык определяется $LL(k+1)$ -грамматикой без e -правил.

****5.1.24.** Покажите, что каждый $LL(k)$ -язык определяется $LL(k+1)$ -грамматикой в нормальной форме Грейбах.

***5.1.25.** Допустим, что $A \rightarrow \alpha\beta | \alpha\gamma$ — такие два правила грамматики G , что из α не выводится e , а β и γ начинаются разными символами. Покажите, что G не $LL(1)$ -грамматика. При каких условиях замена этих правил правилами

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta | \gamma \end{aligned}$$

преобразует грамматику G в эквивалентную $LL(1)$ -грамматику?

5.1.26. Покажите, что если $G = (N, \Sigma, P, S)$ является $LL(k)$ -грамматикой, то для каждого $A \in N$ грамматика G_A , получающаяся из грамматики (N, Σ, P, A) удалением всех бесполезных правил и символов, также является $LL(k)$ -грамматикой.

Существует класс грамматик, называемых LC -грамматиками, которые, подобно LL -грамматикам, можно анализировать с помощью детерминированного МП-преобразователя, читающего вход

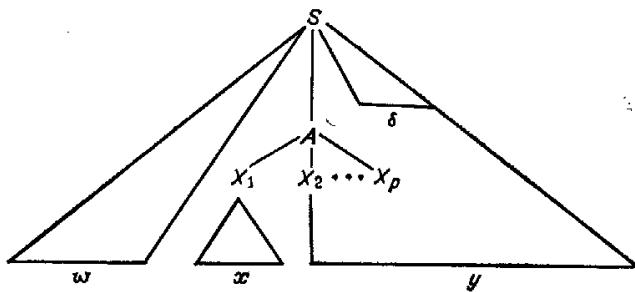


Рис. 5.7. Разбор по левому участку.

слева направо, но делающего разбор по левому участку. Содержательно грамматика $G = (N, \Sigma, P, S)$ является $LC(k)$ -грамматикой, если, зная левый вывод $S \Rightarrow^* wA\delta$, можно однозначно определить, что к A должно применяться правило $A \rightarrow X_1 \dots X_p$, когда уже известна часть входной цепочки, выведенная из X_1 (символ X_1 называется левым участком правила $A \rightarrow X_1 \dots X_p$), и следующие k входных символов. В формальном определении, если X_1 — терминал, можно посмотреть только на следующие $k-1$ символов. Это ограничение налагается ради простоты формулировки одной интересной теоремы, приведенной в упр. 5.1.33. На рис. 5.7 отражено, что правило $A \rightarrow X_1 \dots X_p$ распознается по цепочке wx и первым k символам (или $k-1$ символам, если

$X_1 \in \Sigma$) цепочки y . Заметим, что если бы грамматика G была $LL(k)$ -грамматикой, то можно было бы распознать это правило «быстрее», а именно сразу после того, как прочитаны w и $FIRST_k(xy)$.

В определении $LC(k)$ -грамматики будут участвовать выводы следующего типа:

Определение. Пусть G — КС-грамматика. Будем писать $S \Rightarrow_{lc}^* wA\delta$, если $S \Rightarrow^* wA\delta$ и нетерминал A не является левым участком того правила, благодаря которому он в ходе этого вывода оказался в левово выводимой цепочке $wA\delta$.

Например, для грамматики G_0 неверно, что $E \Rightarrow_{lc}^* E + T$, так как E появляется в $E + T$ в качестве левого участка правила $E \rightarrow E + T$. С другой стороны, верно, что $E \Rightarrow_{lc}^* a + T$, так как T не является левым участком правила $E \rightarrow E + T$, благодаря которому символ T в ходе вывода $E \Rightarrow E + T \Rightarrow a + T$ оказался в цепочке $a + T$.

Определение. КС-грамматика $C = (N, \Sigma, P, S)$ называется $LC(k)$ -грамматикой, если она удовлетворяет таким условиям: Допустим, что $S \Rightarrow_{lc}^* wA\delta$. Тогда для каждой цепочки $u \in \Sigma^{*k}$ и вывода $A \Rightarrow^* Bu$ существует не более одного такого правила $B \rightarrow \alpha$, что

- (1) (a) если $\alpha = C\beta$, где $C \in N$, то $u \in FIRST_k(\beta\gamma\delta)$,
- (б) если, кроме того, $C = A$, то $u \notin FIRST_k(\delta)$,
- (2) если α начинается терминалом, то $u \in FIRST_k(\alpha\gamma\delta)$.

Условие (1a) гарантирует, что правило $B \rightarrow C\beta$, которое нужно применить, определяется однозначно, как только мы увидим терминальную цепочку w , выведенную из C (левого участка), и цепочку $FIRST_k(\beta\gamma\delta)$ (аванцепочку).

Условие (1b) гарантирует, что если нетерминал A леворекурсивный (в LC -грамматике это возможно), то после того, как обнаружено его вхождение, можно сказать, является ли оно левым участком правила $B \rightarrow A\gamma$ или это вхождение A в левово выводимую цепочку $wA\delta$.

Условие (2) говорит о том, что $FIRST_k(\alpha\gamma\delta)$ однозначно определяет правило $B \rightarrow \alpha$, которое при разборе по левому участку нужно применить сразу после того, как в поле зрения оказалась цепочка wB (если цепочка α пуста или начинается терминальным символом).

Для каждой $LC(k)$ -грамматики G можно построить детерминированный алгоритм разбора по левому участку, который анализирует входную цепочку, распознавая левый участок применимого правила восходящим методом, а остальную часть этого правила — сверху вниз.

Теперь покажем в общих чертах, как по LC(1)-грамматике строить соответствующий анализатор. Пусть $G = (N, \Sigma, P, S)$ — LC(1)-грамматика. Построим по ней такой анализатор по левому участку \mathcal{A} , что $\tau(\mathcal{A}) = \{(x, \pi) \mid x \in L(G)\}$ и π — разбор по левому участку цепочки x . Анализатор \mathcal{A} использует входную ленту, магазин и выходную ленту так, как это делает k -предсказывающий анализатор.

Множеством магазинных символов служит $\Gamma = N \cup \Sigma \cup (N \times N) \cup \{\$\}$. Вначале магазин содержит $S\$$ (причем S — верхний символ). Одни нетерминальный или терминальный символ, расположенный наверху магазина, можно интерпретировать как очередную цель, которую нужно распознать. Если верхним символом магазина является пара нетерминалов вида $[A, B]$, то первую компоненту A можно рассматривать как текущую цель, которую нужно распознать, а вторую компоненту B — как левый участок, только что распознанный.

Для удобства построим таблицу T , управляющую разбором по левому участку. Она отображает множество $\Gamma \times (\Sigma \cup \{e\})$ в $(\Gamma^* \times (P \cup \{e\})) \cup \{\text{выброс, допуск, ошибка}\}$. Эта управляющая таблица похожа на таблицу, управляющую 1-предсказывающим алгоритмом разбора для LL(1)-грамматик. Конфигурацией анализатора \mathcal{A} будет тройка $(w, X\alpha, \pi)$, где w — необработанная часть входной цепочки, $X\alpha$ — содержимое магазина ($X \in \Gamma$ — верхний символ) и π — часть выходной цепочки, образовавшаяся к данному моменту. Если $T(X, a) = (\beta, i)$, где $X \in N \cup (N \times N)$, то будем писать $(aw, X\alpha, \pi) \vdash (\beta, a)$. Если $T(a, a) = \text{выброс}$, то пишем $(aw, a\alpha, \pi) \vdash (w, \alpha, \pi)$. Будем говорить, что π — разбор (по левому участку) цепочки x , если $(x, S\$, e) \vdash^*(e, \$, \pi)$.

Пусть $G = (N, \Sigma, P, S)$ — LC(1)-грамматика. Таблица T строится по G следующим образом:

- (1) Допустим, что $B \rightarrow \alpha$ — правило из P с номером i .
 - (а) Если $\alpha = C\beta$, где C — нетерминал, то $T([A, C], a) = (\beta [A, B], i)$ для всех $A \in N$ и $a \in \text{FIRST}_1(\beta\gamma\delta)$, таких, что $S \Rightarrow_{lc}^* wA\delta$ и $A \Rightarrow^* B\gamma$. Здесь \mathcal{A} распознает левые участки снизу вверх. Заметим, что A — либо S , либо левый участок некоторого правила, так что в какой-то момент разбора A будет целью.
 - (б) Если α не начинается нетерминалом, то $T(A, a) = (\alpha [A, B], i)$ для всех $A \in N$ и $a \in \text{FIRST}_1(\alpha\gamma\delta)$, таких, что $S \Rightarrow_{lc}^* wA\delta$ и $A \Rightarrow^* B\gamma$.
- (2) $T([A, A], a) = (e, e)$ для всех $A \in N$ и $a \in \text{FIRST}_1(\delta)$, таких, что $S \Rightarrow_{lc}^* wA\delta$.
- (3) $T(a, a) = \text{выброс}$ для всех $a \in \Sigma$.
- (4) $T(\$, e) = \text{допуск}$.
- (5) В остальных случаях $T(X, a) = \text{ошибка}$.

Пример 5.20. Рассмотрим грамматику G с правилами

- | | |
|---------------------------------------|-----------------------|
| (1) $S \rightarrow S + A$ | (2) $S \rightarrow A$ |
| (3) $A \rightarrow A * B$ | (4) $A \rightarrow B$ |
| (5) $B \rightarrow \langle S \rangle$ | (6) $B \rightarrow a$ |

G является LC(1)-грамматикой, причем это фактически грамматика G_0 , слегка измененная. Таблица, управляющая разбором по левому участку для грамматики G , приведена на рис. 5.8.

		входной символ					
		α	$<$	$>$	$+$	$*$	$\$$
магазинный символ							
S	$a[S, B], 6$	$\langle S \rangle [S, B], 5$					
A	$a[A, B], 6$	$\langle S \rangle [A, B], 5$					
B	$a[B, B], 6$	$\langle S \rangle [B, B], 5$					
$[S, S]$		e, e	$+A [S, S], 1$				e, e
$[S, A]$			$[S, S], 2$	$[S, S], 2$	$*B [S, A], 3$	$[S, S], 2$	
$[S, B]$			$[S, A], 4$	$[S, A], 4$	$[S, A], 4$	$[S, A], 4$	
$[A, A]$		e, e	e, e	$*B [A, A], 3$	e, e		
$[A, B]$			$[A, A], 4$	$[A, A], 4$	$[A, A], 4$	$[A, A], 4$	
$[B, B]$		e, e	e, e	e, e	e, e	e, e	
a	выброс						
$<$	выброс						
$>$		выброс					
$+$			выброс				
$*$				выброс			
$\$$					выброс		допуск

Рис. 5.8. Таблица, управляющая разбором по левому участку для грамматики G .

Анализатор, управляемый этой таблицей, для входной цепочки $\langle a * a \rangle$ проделает такую последовательность тиков:

- $$\begin{aligned}
 & (\langle a * a \rangle, S\$, e) \vdash \langle a * a \rangle, \langle S \rangle [S, B]\$, 5 \\
 & \quad \vdash (a * a), S\langle S, B] \$, 5 \\
 & \quad \vdash (a * a), a [S, B] \langle S, B] \$, 56 \\
 & \quad \vdash (* a), [S, B] \langle S, B] \$, 56 \\
 & \quad \vdash (* a), [S, A] \langle S, B] \$, 564 \\
 & \quad \vdash (* a), * B [S, A] \langle S, B] \$, 5643
 \end{aligned}$$

$\vdash \langle a \rangle, B[S, A] \triangleright [S, B] \$, 5643)$
 $\vdash \langle a \rangle, a[B, B][S, A] \triangleright [S, B] \$, 56436)$
 $\vdash \langle \rangle, [B, B][S, A] \triangleright [S, B] \$, 56436)$
 $\vdash \langle \rangle, [S, A] \triangleright [S, B] \$, 56436)$
 $\vdash \langle \rangle, [S, S] \triangleright [S, B] \$, 564362)$
 $\vdash \langle \rangle, > [S, B] \$, 564362)$
 $\vdash (e, [S, B] \$, 564362)$
 $\vdash (e, [S, A] \$, 5643624)$
 $\vdash (e, [S, S] \$, 56436242)$
 $\vdash (e, \$, 56436242)$

Читатель может легко проверить, что 56436242—действительно разбор по левому участку цепочки $\langle a * a \rangle$. \square

*5.1.27. Покажите, что грамматика с правилами

$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow aAb \mid 0 \\ B &\rightarrow aBbb \mid 1 \end{aligned}$$

не является LC(k)-грамматикой ни для какого k .

*5.1.28. Покажите, что грамматика

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow P \uparrow F \mid P \\ P &\rightarrow (E) \mid a \end{aligned}$$

является LC(1)-грамматикой.

5.1.29. Постройте анализатор по левому участку для грамматики из упр. 5.1.28.

**5.1.30. Дайте алгоритм, проверяющий, является ли произвольная грамматика LC(1)-грамматикой.

**5.1.31. Покажите, что каждая LL(k)-грамматика является LC(k)-грамматикой.

5.1.32. Приведите пример LC(1)-грамматики, которая не является LL-грамматикой.

**5.1.33. Покажите, что если к грамматике G применяется алгоритм 2.14 для преобразования ее к нормальной форме Грейбах, то в результате получится грамматика, которая будет LL(k)-грамматикой тогда и только тогда, когда G —LC(k)-грамматика. Следовательно, класс LC-языков совпадает с классом LL-языков.

*5.1.34. Дайте алгоритм построения анализатора по левому участку для произвольной LC(k)-грамматики.

Проблема для исследования

5.1.35. Найдите преобразования, позволяющие превращать не LL(k)-грамматики в эквивалентные LL(1)-грамматики.

Упражнения на программирование

5.1.36. Напишите программу, которая в качестве входа воспринимает произвольную КС-грамматику G и строит для нее таблицу, управляющую 1-предсказывающим анализатором, если G —LL(1)-грамматика.

5.1.37. Напишите программу, которая по входу, состоящему из управляющей таблицы и цепочки, делает с помощью данной таблицы разбор данной входной цепочки.

5.1.38. Преобразуйте одну из грамматик, описанных в приложении, в эквивалентную LL(1)-грамматику. Затем постройте для этой грамматики LL(1)-анализатор.

5.1.39. Напишите программу, проверяющую, является ли данная грамматика LL(1)-грамматикой.

Пусть M —управляющая таблица для LL(1)-грамматики G . Допустим, что мы анализируем входную цепочку и анализатор достиг конфигурации $(ax, X\alpha, \pi)$. Если $M(X, a) = \text{ошибка}$, то нам хотелось бы объявить о том, что в данной позиции входной цепочки встретилась ошибка, и перейти к процедуре исправления ошибок, изменяющей содержимое магазина и входной ленты так, чтобы можно было нормально продолжать разбор. Среди стратегий исправления ошибок возможны такие:

- (1) УстраниТЬ a и попытаться продолжать разбор.
 - (2) Заменить a таким символом b , что $M(X, b) \neq \text{ошибка}$, и продолжать разбор.
 - (3) Вставить перед символом a во входной цепочке такой символ b , что $M(X, b) \neq \text{ошибка}$, и продолжать разбор. Этим приемом надо пользоваться осторожно, так как легко впасть в бесконечный цикл.
 - (4) Читать далее входную цепочку, пока не обнаружится некоторый выделенный входной символ b . Выбрасывать из магазина символы до тех пор, пока не обнаружится такой символ X , что $X \Rightarrow^* b\beta$ для некоторой цепочки β . Затем продолжить нормальный разбор.
- Можно также для каждой пары (X, a) , для которой $M(X, a) = \text{ошибка}$, перечислить несколько возможных приемов исправления ошибки, начиная с наиболее обещающих из них. Вполне возможно, что в некоторых ситуациях наиболее разум-

ный образ действий заключается во вставке символа, тогда как в других случаях более вероятно, что к успеху приведет устранение или изменение символа.

5.1.40. Разработайте алгоритм исправления ошибок для LL(1)-анализатора, построенного в упр. 5.1.38.

Замечания по литературе

LL(k)-грамматики были впервые определены Льюисом и Стирнзом [1968]. В ранней версии их работы эти грамматики назывались TD(k)-грамматиками (по первым буквам слов top-down — сверху вниз). Простые LL(1)-грамматики впервые исследованы Коренъяком и Хопкрофтом [1966], которые называли их S-грамматиками¹⁾.

Теория LL(k)-грамматик была в значительной степени развита Розенкранцем и Стирнзом [1970], в их статье можно найти решения упр. 5.1.21—5.1.24. LL(k)-грамматики и другие варианты грамматик, ориентированных на детерминированный разбор, рассматривались Кнутом [1967], Курки-Суони [1969], Вудом [1969a, 1970] и Чуликом [1968]²⁾.

Льюис, Стирнз и Розенкранц построили компиляторы для Алгола и Фортрана, в которых на этапе синтаксического анализа используется LL(1)-анализатор. Подробности о компиляторе для Алгола можно найти в статье Льюиса и Розенкранца [1971], в ней также содержится LL(1)-грамматика для Алгола 60.

LC(k)-грамматики были впервые определены Розенкранцем и Льюисом [1970], в их статье можно найти ключ к упр. 5.1.27—5.1.34.

ДОПОЛНЕНИЕ О МЕТОДАХ РАЗБОРА „ПО ТЕКУЩЕМУ СИМВОЛУ“

B. N. Агафонов

В разд. 2.5.3 был описан недетерминированный предсказывающий алгоритм разбора для произвольных КС-грамматик. Введение понятия LL(1)-грамматики — это один из возможных способов наложить на грамматику такие условия, чтобы шаг предсказания очередного правила вывода можно было сделать детерминированным, используя для этого только текущий вход-

¹⁾ А также независимо от этих авторов Фуксманом [1968], назвавшим эти грамматики *разделенными*. — Прим. перев.

²⁾ Классы грамматик, ориентированные на безвозвратный исходящий анализ и обобщающие класс LL(k)-грамматик, изучаются в работах Яжабека и Кравчика [1975], Нийхольта [1976], Испомняцкой [1976], Анисимова [1974а, б], Никитченко и Шкильняк [1975] и Шевченко [1974]. В последних четырех работах проблема разбора рассматривается в рамках некоторой общей схемы управления анализом, предложенной Редько [1970]. Особого внимания заслуживают методы разбора „по текущему символу“, к которым относится алгоритм анализа для LL(1)-грамматик. По-видимому, первый метод такого рода разработан Конвэем [1953]. Затем (кроме работ Коренъяка и Хопкрофта [1966], Льюиса и Стирнса [1968]) следует упомянуть работы Фостера [1968, 1970], Вуда [1969], Вельбицкого и Ющенко [1970], Вельбицкого [1973], Фуксмана ([1976] и [Основы разработки трансляторов, 1974]), Комора [1972], Ахо и др. [1975], Кауфмана [1976] (см. дополнение переводчика к этому разделу). — Прим. перев.

ной символ. Рассмотрим несколько иной подход, который является естественным обобщением алгоритма разбора, соответствующего разделенной (т. е. простой LL(1)-) грамматике. Для разделенной грамматики шаг предсказания максимально прост: если a — текущий входной символ, A — верхний символ магазина и в грамматике есть правило $A \rightarrow a\alpha$, то в магазине надо заменить A на α и сдвинуть входную головку. Идея обобщения такова. К A -правилам разделенной грамматики разрешается добавить e -правило $A \rightarrow e$, а на шаге предсказания для текущих символов a и A разрешается применить правило $A \rightarrow e$ (т. е. удалить A из магазина), если в грамматике нет правила вида $A \rightarrow a\alpha$. Классы грамматик, для которых работает такой метод разбора, изучались в работах Фуксмана ([Основы разработки трансляторов, 1974] и [1976]), Комора [1972], Кауфмана [1976], Ахо и др. [1975].

Определение. КС-грамматика $G = (N, \Sigma, P, S)$ называется грамматикой в слабой нормальной форме Грейбах, если каждое правило из P имеет вид $A \rightarrow a\alpha$, где $\alpha \in \Sigma$, $a \in N^*$, или $A \rightarrow e$. Если к тому же у нее правые части правил вида $A \rightarrow a\alpha$ и $A \rightarrow b\beta$ начинаются разными символами, т. е. $a \neq b$, то она называется псевдоразделенной.

Для псевдоразделенной грамматики $G = (N, \Sigma, P, S)$ определим соответствующий простой МП-распознаватель $M_G = (\Sigma, N, \delta, S)$, у которого $\Sigma \cup \{\$\}$ — входной алфавит ($\$\notin \Sigma$), N — магазинный алфавит, S — начальный символ магазина и δ — функция, отображающая $(\Sigma \cup \{\$\}) \times N$ в $N^* \times \{0, 1\}$ (она описывает один тakt его работы). Функция δ задается равенствами

(1) $\delta(a, A) = (\alpha, 1)$ тогда и только тогда, когда правило $A \rightarrow a\alpha$ принадлежит P ;

(2) $\delta(b, A) = (e, 0)$ тогда и только тогда, когда $A \rightarrow e$ принадлежит P и в P нет правила вида $A \rightarrow b\beta$.

В парах $(\alpha, 1)$ и $(e, 0)$ вторая компонента указывает соответственно на наличие сдвига входной головки и его отсутствие. Для всех $w \in \Sigma^*$ и $\gamma \in N^*$ положим $(aw\$, A\gamma) \vdash (w\$, \alpha\gamma)$ в случае (1) и $(bw\$, A\gamma) \vdash (bw\$, \gamma)$ в случае (2). Кроме того, если в равенстве (2) $b = \$$, положим $(\$, A\gamma) \vdash (\$, \gamma)$. Языком $L(M_G)$, распознаваемым M_G , назовем множество $\{w \in \Sigma^* \mid (w\$, S) \vdash^* (\$, e)\}$.

Заметим, что если $w \in L(M_G)$, то для цепочки w вычисление в M_G соответствует ее левому выводу в G и $w \in L(G)$, т. е. $L(M_G) \subseteq L(G)$. Но не обязательно $L(M_G) = L(G)$.

Пример 1. Пусть G_1 — псевдоразделенная грамматика с правилами

$$\begin{aligned} S &\rightarrow aSA \mid e \\ A &\rightarrow a \mid b \end{aligned}$$

Тогда $M_G = (\{a, b\}, \{S, A\}, \delta, S)$, где

$$\begin{aligned}\delta(a, S) &= (SA, 1) \\ \delta(b, S) &= (e, 0) \\ \delta(\$, S) &= (e, 0) \\ \delta(a, A) &= (e, 1) \\ \delta(b, A) &= (e, 1)\end{aligned}$$

Легко убедиться, что $L(G) = \{a^n x \mid n \geq 0, x \in \{a, b\}^*, |x| = n\}$ и $L(M_G) = \{e\} \cup \{a^n by \mid n \geq 1, y \in \{a, b\}^*, |y| = n - 1\}$. Для цепочки $aa\$$ распознаватель M_G сделает последовательность тактов

$$(aa\$, S) \vdash (a\$, SA) \vdash (\$, SAA) \vdash (\$, AA)$$

Сравним эту последовательность с левым выводом цепочки aa в грамматике G :

$$S \Rightarrow_t aSA \Rightarrow_t aA \Rightarrow_t aa$$

Если $L(M_G) = L(G)$, то распознаватель M_G распознает в точности язык $L(G)$, т. е. он, так сказать, адекватен грамматике G и, если снабдить его выходом, станет анализатором для G , будет строить левые выводы всех цепочек из $L(G)$. \square

Определение. Псевдоразделенная грамматика G называется **слаборазделенной**, если $L(G) = L(M_G)$ для соответствующего простого МП-распознавателя M_G .

Пример 2. Грамматика G_2 с правилами

$$\begin{aligned}S &\rightarrow aSA \mid e \\ A &\rightarrow b \mid e\end{aligned}$$

слаборазделенная и $L(G_2) = \{a^n b^k \mid 0 \leq k \leq n\}$. \square

Упр. 1. Для произвольной КС-грамматики G постройте эквивалентную ей грамматику G' в слабой нормальной форме Грейбах.

Упр. 2. Для простого МП-распознавателя M_G , соответствующего псевдоразделенной грамматике G , постройте такой ДМП-автомат M , что $L_e(M) = L(M_G) \$$.

****Упр. 3.** Докажите, что не существует алгоритма, распознавающего по произвольной КС-грамматике, является ли она слаборазделенной.

Определение. 1-слаборазделенной грамматикой называется псевдоразделенная грамматика $G = (N, \Sigma, P, S)$, для которой выполняется следующее условие: если P содержит правила $A \rightarrow e$ и $A \rightarrow aa$, то не существует такого $B \in N$ (в том числе и $B = A$), что P содержит правило $B \rightarrow ab$, $S \Rightarrow_t^* wA\gamma B\delta$ и $\gamma \Rightarrow^* e$. Если $B = A$ допускается, грамматика G называется 2-слаборазделенной.

Упр. 4. Покажите, что 2-слаборазделенная грамматика является слаборазделенной.

Упр. 5. Покажите, что 1-слаборазделенная грамматика является LL(1)-грамматикой.

Упр. 6. Покажите, что для каждой LL(1)-грамматики можно построить эквивалентную ей 1-слаборазделенную грамматику.

Упр. 7. Покажите, что существует 2-слаборазделенная грамматика, которая не является 1-слаборазделенной. **Указание.** Рассмотрите грамматику G_2 из примера 2.

Упр. 8. Постройте алгоритмы, проверяющие, является ли КС-грамматика G

- (а) 1-слаборазделенной,
- (б) 2-слаборазделенной.

****Упр. 9.** Покажите, что

- (а) существует детерминированный язык, который не является слаборазделенным,
- (б) существует слаборазделенный язык, который не является 2-слаборазделенным,
- (в) существует 2-слаборазделенный язык, который не является 1-слаборазделенным (т. е. LL(1)-языком в силу упр. 5 и 6).

В работах Вельбицкого и Ющенко [1970] и Вельбицкого [1973] определены два метаязыка, предназначенные для описания детерминированных распознавателей, использующих один или несколько магазинов (они названы соответственно CM- и R-метаязыком, причем второй язык является некоторым уточнением и развитием первого). Ограничивааясь случаем одного магазина, мы определим здесь метаязык (назовем его CMR-метаязыком), охватывающий основные черты этих формализмов.

Программа распознавателя, записанная на CMR-метаязыке, состоит из команд, роль которых аналогична роли команд ДМП-автомата (если $\delta(q, a, Z) = (q', \gamma)$ интерпретировать как команду) или операторов метаязыка Флойда — Эванса (разд. 5.4.4). Задаются конечный алфавит состояний K (он же будет магазинным алфавитом), входной, или терминальный, алфавит Σ , и каждая команда записывается в виде

$$k \sim a \xrightarrow{W_i(\gamma)} X$$

где $k \in K$, $a \in \Sigma \cup \{e\}$, $\gamma \in K^*$, $X \in K \cup \{\mu\}$ (причем $\mu \notin K$) и W_i — обозначение операции, выполняемой командой над конфигурацией распознавателя. Конфигурация имеет вид $(k, w\$, \alpha)$, где $k \in K$ — состояние распознавателя, $w \in \Sigma^*$ — необработанная часть

входной цепочки, $\alpha \in K^*$ — цепочка в магазине (будем считать, что верхний символ расположен слева), $\$$ — правый концевой маркер ($\$ \notin \Sigma \cup K$).

Существует два типа операций, обозначаемых W_1 и W_2 , и соответственно два типа команд. Команда первого типа в случае $X = k' \in K$ применима к конфигурации $(k, aw\$, \alpha)$; в результате она дает конфигурацию $(k', w\$, \gamma\alpha)$, т. е. к содержимому магазина приписывается γ (или ничего, если $\gamma = e$) и, если $a \in \Sigma$, сдвигается входная головка. В случае $X = \mu$ она применима к конфигурации $(k, aw\$, k'\alpha)$, если $\gamma = e$, и $(k, aw\$, \alpha)$, если $\gamma = k''\gamma'$. Для первой конфигурации она дает $(k', w\$, \alpha)$, т. е. из магазина извлекается очередное состояние. Для второй конфигурации она дает $(k'', w\$, \gamma'\alpha)$, т. е. очередным состоянием становится первый символ цепочки γ , а остальная ее часть помещается в магазин.

Команда второго типа имеет вид

$$k \sim a \xrightarrow{W_2(r)} k'$$

где $r \in K$ и $k' \in K$. Она применима к конфигурации $(k, aw\$, r\alpha)$ и дает $(k', w\$, \alpha)$, т. е. она применима, если r — верхний символ магазина, и в этом случае она удаляет его из магазина. Заметим, что применимость команды второго типа зависит от содержимого магазина, а первого типа — нет.

Символ, стоящий слева от знака \sim , образует левую часть команды. Команду с левой частью k назовем k -командой. Последовательность k -команд

$$k \sim a_1 \xrightarrow{W_{l_1}(\gamma_1)} X_1, k \sim a_2 \xrightarrow{W_{l_2}(\gamma_2)} X_2, \dots, k \sim a_n \xrightarrow{W_{l_n}(\gamma_n)} X_n$$

будем называть k -комплектом, если из $a_i \in \Sigma$ и $i < j$ следует,

что $a_i \in \Sigma$. Будем записывать его в виде $k \sim a_1 \xrightarrow{W_{l_1}(\gamma_1)} X_1 | \dots | a_n \xrightarrow{W_{l_n}(\gamma_n)} X_n$. Заметим, что команды k -комплекта упорядочены так, чтобы команды с $a_i \in \Sigma$ предшествовали командам с $a_i = e$.

Определение. CMR-распознавателем назовем пятерку $M = (K, \Sigma, R, k_0, f)$, где K — алфавит состояний (он же магазинный алфавит), Σ — входной, или терминальный, алфавит, не пересекающийся с K , R — конечное множество k -комплектов (не более одного для каждого k), $k_0 \in K$ — начальное состояние, $f \in K$ — заключительное состояние.

Если первая из команд k -комплекта распознавателя M , применимая к конфигурации $(k, w\$, \gamma)$, дает $(k', w'\$, \gamma')$, будем писать $(k, w\$, \gamma) \vdash_M (k', w'\$, \gamma')$.

Языком, определяемым распознавателем M , назовем множество

$$L(M) = \{w \in \Sigma^* \mid (k_0, w\$, f) \vdash_M^* (f, \$, e)\}$$

Пример 3. Рассмотрим CMR-распознаватель $M = (\{k_0, k_1, f\}, \{a, b\}, R, k_0, f)$, в котором R состоит из комплектов

$$\begin{aligned} k_0 \sim a &\xrightarrow{W_1(k_1)} k_0 | e \xrightarrow{W_1(e)} \mu \\ k_1 \sim b &\xrightarrow{W_1(e)} \mu | e \xrightarrow{W_1(e)} \mu \end{aligned}$$

Для входной цепочки $aab\$$ распознаватель M сделает такую последовательность тактов:

$$\begin{aligned} (k_0, aab\$, f) &\vdash (k_0, ab\$, k_1 f) \\ &\vdash (k_0, b\$, k_1 k_1 f) \\ &\vdash (k_1, b\$, k_1 f) \\ &\vdash (k_1, \$, f) \\ &\vdash (f, \$, e) \quad \square \end{aligned}$$

Определение. CMR-распознаватель назовем *слабым*, если все его команды первого типа.

Для слабого CMR-распознавателя $M = (K, \Sigma, R, k_0, f)$ определим соответствующую КС-грамматику $G_M = (N, \Sigma, P, S)$, положив $N = K$, $S = k_0$ и для каждой команды из R вида $k \sim a \xrightarrow{W_1(\gamma)} k'$ включив в P правило $k \rightarrow ak'\gamma$, а для команды вида $k \sim a \xrightarrow{W_1(\gamma)} \mu$ — правило $k \rightarrow a\gamma$.

Пример 4. Распознаватель M из примера 3 слабый и ему соответствует КС-грамматика

$$\begin{aligned} k_0 &\rightarrow ak_0k_1 | e \\ k_1 &\rightarrow b | e \end{aligned}$$

Она получается из грамматики примера 2 очевидным переименованием нетерминалов. \square

Легко видеть, что если $w \in L(M)$, то каждый шаг слабого CMR-распознавателя M соответствует шагу левого вывода цепочки w в грамматике G_M , на котором применяется соответствующее правило. Следовательно, $L(M) \equiv L(G_M)$. Но не обязательно $L(M) = L(G_M)$.

Слабый CMR-распознаватель M назовем *адекватным*, если $L(M) = L(G_M)$.

Упр. 10. Покажите, что если в каждом k -комплекте слабого CMR-распознавателя M отбросить для каждого $a \in \Sigma \cup \{e\}$ все правила вида $k \sim a \xrightarrow{W_1(\gamma)} X$, кроме первого, то получится распознаватель M' , эквивалентный M .

Упр. 11. Покажите, что если у слабого CMR-распознавателя M команды вида $k \sim e \xrightarrow{W_1(\gamma)} X$ удовлетворяют условиям $\gamma = e$ и $X = \mu$, то

(а) соответствующая КС-грамматика G_M псевдоразделенная,
 (б) M эквивалентен простому МП-распознавателю $M' = M_{G_M}$,
 соответствующему грамматике G_M , и, значит, M адекватен тогда
 и только тогда, когда G_M — слаборазделенная грамматика. Ука-
 зание. Покажите, что $(k, w\$, \gamma) \vdash_M (k', w'\$, \gamma')$ тогда и только
 тогда, когда $(w\$, k\gamma) \vdash_{M'} (w'\$, k'\gamma')$.

Упр. 12. Для произвольного слабого CMR-распознавателя M постройте эквивалентный ему слабый CMR-распознаватель M' , команды которого удовлетворяют условиям упр. 11.

Упр. 13. Покажите, что для произвольного CMR-распозна-
 вателя можно построить эквивалентный ему ДМП-автомат, и
 обратно (т. е. CMR-распознаватели определяют в точности класс
 детерминированных КС-языков).

Рассмотрим теперь так называемый диаграммер Конвэя [1963] —
 МП-распознаватель, управляющее устройство которого задается
 конечным множеством D „синтаксических диаграмм“. Содержа-
 тельно, диаграмма $d \in D$ определяет некоторый „синтаксический
 тип“, т. е. множество цепочек L_d , распознаваемых диаграммером
 при „прохождении“ диаграммы d . Связь между d и L_d в диаграм-
 мере аналогична связи между нетерминалом A и множеством L_A
 цепочек, выводимых из A , в КС-грамматике. Чтобы подчеркнуть
 эту связь, определим вначале синтаксические диаграммы част-
 ного вида, непосредственно связанные с КС-правилами.

Определение. КС-диаграммой d_A , соответствующей множеству
 A -правил

$$A \rightarrow X_{11} \dots X_{1n_1} | X_{21} \dots X_{2n_2} | \dots | X_{k1} \dots X_{kn_k}$$

назовем помеченный ориентированный граф, изображенный на
 рис. Д.1. В этой диаграмме вершина 1 начальная, вершина j —
 заключительная, а в нумерации вершин важно лишь то, что
 она взаимно однозначна. Нетерминал A считается именем ди-
 аграмм d_A .

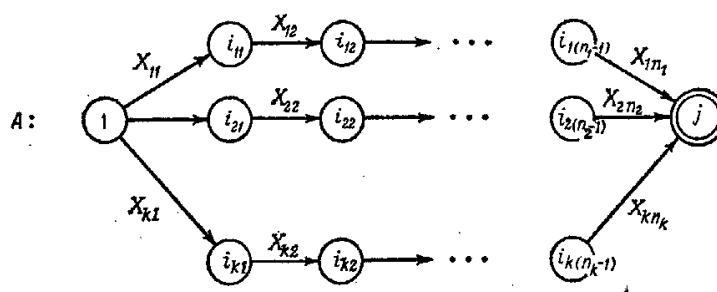


Рис. Д.1. КС-диаграмма.

КС-диаграммом, соответствующим КС-грамматике $G = (N, \Sigma, P, S)$, назовем четверку $M_G = (\Sigma, N, D, d_S)$, где $D = \{d_A \mid A \in N\}$ и d_S — начальная диаграмма.

Пример 5. КС-диаграмм $M_{G_2} = (\{a, b\}, \{S, A\}, D, d_S)$, соответствующий КС-грамматике G_2 примера 2, состоит из двух КС-диаграмм, изображенных на рис. Д.2. □

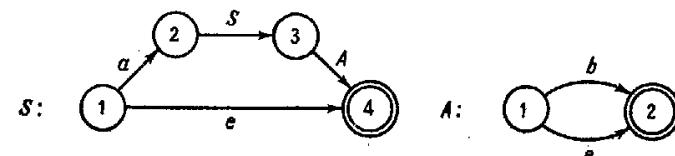


Рис. Д.2. КС-диаграмм.

В общем случае *синтаксической диаграммой* (С-диаграммой) над парой непересекающихся алфавитов Σ и Δ назовем конечный помеченный ориентированный граф, вершины которого занумерованы и выделены одна начальная вершина (с номером 1) и одна или несколько заключительных вершин. Каждая дуга помечена символами либо $a \in \Sigma \cup \{e\}$, либо $d \in \Delta$, либо (d, i) (содержательно, i — номер заключительной вершины диаграммы, обозначенной d и имеющей несколько заключительных вершин). В начальную вершину дуги не входят, из заключительной — не выходят.

Диаграммером назовем четверку $M = (\Sigma, \Delta, D, d_0)$, где Σ — конечный входной алфавит, Δ — конечный диаграммный алфавит (его символы служат именами или обозначениями диаграмм), D — конечное множество С-диаграмм над алфавитами Σ и Δ (причем $\Sigma \cap \Delta = \emptyset$ и между Δ и D установлено взаимно однозначное соответствие, именующее диаграммы), d_0 — начальная диаграмма с одной заключительной вершиной.

Состоянием диаграммера M назовем пару (d, i) , где $d \in D$ и i — номер некоторой вершины диаграммы d , т. е. состояние — это по существу вершина диаграммы, обозначаемая как пара. Множество состояний Q является также магазинным алфавитом диаграммера M (как и у CMR-распознавателя). Конфигурация диаграммера имеет вид $((d, i), w\$, \gamma\$)$, где $(d, i) \in Q$ — текущее состояние, $w \in \Sigma^*$ — необработанная часть входной цепочки, $\gamma \in Q^*$ — магазинная цепочка (верхний символ слева), $\$$ — концевой маркер входа и магазина ($\$ \notin \Sigma \cup Q$).

Определим один торт работы диаграммера M , т. е. отношение — на его конфигурациях. Пусть дана конфигурация $((d, i), a\omega\$, \gamma\$)$, где $a \in \Sigma$ либо $a = w = e$.

(а) Если из вершины (d, i) выходит дуга, помеченная символом $a \in \Sigma$ и входящая в вершину (d, j) , то

$$((d, i), aw\$, \gamma\$) \vdash ((d, j), w\$, \gamma\$)$$

т. е. делается переход из (d, i) в (d, j) по дуге, помеченной a , и сдвигается входная головка.

(б) Если условие в (а) не выполнено, но из (d, i) выходит дуга, помеченная именем диаграммы d' или парой (d', j) , то

$$((d, i), aw\$, \gamma\$) \vdash ((d', 1), aw\$, (d, i) \gamma\$)$$

т. е. состоянием становится начальная вершина диаграммы d' , а в магазине запоминается вершина (d, i) , к которой надо вернуться позднее, когда будет пройдена диаграмма d' .

(в) Если не выполнены условия ни в (а), ни в (б), но из (d, i) выходит дуга, помеченная e и ведущая в (d, j) , то

$$((d, i), aw\$, \gamma\$) \vdash ((d, j), aw\$, \gamma\$)$$

т. е. „пустая“ дуга проходится в последнюю очередь без изменения входа и магазина.

(г) Если (d, i) —заключительная вершина, $\gamma = (d', j) \gamma'$ и дуга с меткой (d, i) ведет из (d', j) в (d', k) , то

$$((d, i), aw\$, (d', j) \gamma') \vdash ((d', k), aw\$, \gamma')$$

т. е. информация о следующем состоянии извлекается из магазина (как у СМР-распознавателя). В этом случае диаграмма d уже пройдена, и надо перейти в (d', k) по дуге, соответствующей заключительной вершине (d, i) .

Заметим, что для текущего входного символа a и текущей вершины (d, i) выбор выходящей из нее дуги играет роль выполнения очередной команды распознавателя. Пункты (а)–(в) показывают, что диаграммер, так же как простой МП-распознаватель и СМР-распознаватель, пытается сначала выполнить команду, помеченную символом a , и в последнюю очередь—команду, помеченную e . Отметим еще, что диаграммер является, вообще говоря, недетерминированным распознавателем, так как может случиться, что из (d, i) можно выйти по разным дугам, т. е. выполнить разные команды.

Пусть (d, i) —заключительная вершина диаграммы d . Язык, распознаваемым диаграммером M при прохождении диаграммы d из ее начальной вершины в вершину (d, i) , назовем множество

$$L_{(d, i)} = \{w \in \Sigma^* \mid ((d, 1), w\$, \$) \vdash^* ((d, i), \$, \$)\}$$

Если d имеет одну заключительную вершину i , то вместо $L_{(d, i)}$ будем писать L_d . Язык, распознаваемый диаграммером M ,—это множество $L(M) = L_d$.

Упр. 14. Покажите, что

(а) если $M_G = (\Sigma, N, D, d)$ —КС-диаграммер, соответствующий КС-грамматике $G = (N, \Sigma, P, S)$, то $L_d \subseteq L_A = \{w \mid A \Rightarrow^* G w\}$ и, в частности, $L(M_G) \subseteq L(G)$.

(б) если G —псевдоразделенная грамматика и M'_G —соответствующий ей простой МП-распознаватель, то $L(M_G) = L(M'_G)$ (и, значит, если грамматика G слаборазделенная, то $L(M_G) = L(G)$).

Упр. 15. Для произвольной КС-грамматики постройте эквивалентный ей КС-диаграммер.

Упр. 16. Для произвольного диаграммера постройте эквивалентный ему (недетерминированный) МП-автомат.

Из упр. 15 и 16 следует, что диаграммеры распознают в точности класс КС-языков.

Наложим на диаграммер ограничение, аналогичное LL(1)-условию или условию разделенности для КС-грамматики и позволяющее сделать детерминированным выбор дуги на каждом шаге.

С каждой меткой X , помечающей дугу диаграммера M , связем соответствующее ей „множество первых символов“ $F(X)$, определяемое так:

$$(1) F(a) = \{a\} \text{ для } a \in \Sigma \cup \{e\},$$

(2) если d обозначает диаграмму с одной заключительной вершиной, то

$$F(d) = \{a \mid (a \in \Sigma \text{ и } aw \in L_d) \text{ или } (a = e \text{ и } e \in L_d)\}$$

$$(3) F((d, i)) = \{a \mid (a \in \Sigma \text{ и } aw \in L_{(d, i)}) \text{ или } (a = e \text{ и } e \in L_{(d, i)})\}.$$

Назовем диаграммер M *разделенным*, если для каждой вершины (d, i) , из которой выходят n дуг, помеченных метками X_1, \dots, X_n , множества $F(X_1), \dots, F(X_n)$ попарно не пересекаются. Для разделенного диаграммера M доопределим отношение \vdash : для текущего символа a и вершины (состояния) (d, i) выбирается выходящая из нее дуга с меткой X , для которой $a \in F(X)$, а если таковой нет, то дуга, для которой $e \in F(X)$. Заметим, что выбрать можно только одну дугу.

Упр. 17. Покажите, что если КС-диаграммер M_G , соответствующий КС-грамматике G , разделенный, то G эквивалентна псевдоразделенной грамматике G' и $L(M_G) = L(M'_G)$ для соответствующего G' простого МП-распознавателя M'_G (ср. с упр. 14(б)).

В работе [Ломет, 1973] изучаются детерминированные диаграммеры следующего специального вида.

Определение. Диаграммер $M = (\Sigma, \Delta, D, d_0)$ назовем *L-диаграммером*, если в каждой его диаграмме все дуги, выходящие

из начальной вершины, помечены разными входными символами, а все дуги, выходящие из незаключительной вершины, помечены парами $(d, i_1), (d, i_2), \dots, (d, i_k)$ для $k \geq 1$, обозначающими заключительные вершины некоторой диаграммы d .

Очевидно, что L-диаграммер является детерминированным распознавателем.

****Упр. 18.** Покажите, что для каждого детерминированного КС-языка существует распознавающий его L-диаграммер.

Рассмотренные до сих пор детерминированные нисходящие анализаторы с магазинной памятью строят только один начальный отрезок левого вывода, совместимый с обработанной частью входной цепочки. В [Алгол 68] (гл. 1) описан анализатор (назовем его *синхронным*), который пытается синхронно следить за несколькими левыми выводами¹).

Определение. Синхронным распознавателем M_G , соответствующим КС-грамматике $G = (N, \Sigma, P, S)$, назовем пятерку (Σ, N, D, d_S, Q) , где (Σ, N, D, d_S) — КС-диаграммер, соответствующий грамматике G , а Q — множество состояний, элементами которого служат множества вершин диаграмм из D . Множество Q является также магазинным алфавитом распознавателя M_G .

Конфигурация распознавателя M_G имеет вид $(q, w\$, \gamma\$)$, где $q \in Q$, $w \in \Sigma^*$, $\gamma \in Q^*$, $\$ \notin \Sigma \cup Q$. Далее в пунктах (1), (2) и (3) определяется один такт работы синхронного распознавателя. Эти пункты аналогичны соответственно пунктам (а), (б) и (в, г) в определении КС-диаграммера. Однако существенное различие состоит в том, что, во-первых, состояниями здесь служат множества вершин, во-вторых, даже из одной вершины возможен одновременный переход за один такт по нескольким дугам и, в-третьих, синхронный распознаватель сразу определяется как детерминированный автомат, тогда как диаграммер становится детерминированным, вообще говоря, при дополнительных условиях (например, если выполнено условие разделенности).

Пусть дана конфигурация $(q, aw\$, \gamma\$)$, где $a \in \Sigma$, либо $a = w = e$.

(1) Если множество q содержит вершину, из которой выходит дуга, помеченная $a \in \Sigma$, то

$$(q, aw\$, \gamma\$) \vdash (q', w\$, \gamma\$)$$

где $q' = \{(d, j) |$ дуга, выходящая из $(d, i) \in q$ и помеченная a , ведет в $(d, j)\}$.

¹⁾ L-диаграммер, упоминаемый в упр. 18, тоже может следить за несколькими выводами в подходящей LR-грамматике, но делает это менее „прямым“ способом.

(2) Если условие в (1) не выполнено, но q содержит вершину, из которой выходит дуга, помеченная нетерминалом, то

$$(q, aw\$, \gamma\$) \vdash (q', aw\$, q''\gamma\$)$$

где $q' = \{(d_A, 1) |$ из $(d, i) \in q$ выходит дуга, помеченная $A\}$ и $q'' = \{(d, i) |$ из $(d, i) \in q$ выходит дуга, помеченная нетерминалом $\}$.

(3) Если не выполнены условия ни в (1), ни в (2), но q содержит заключительную вершину или вершину, из которой выходит дуга, помеченная e (обозначим через \tilde{q} множество этих вершин), то

$$(q, aw\$, \gamma\$) \vdash (q', aw\$, \gamma'\$)$$

где $\gamma = q''\gamma'$ и $q' = \{(d, j) |$ из $(d, i) \in q''$ в (d, j) ведет дуга, помеченная A и q содержит вершину диаграммы $d_A\}$.

Языком, распознаваемым синхронным распознавателем M_G , назовем множество

$$L(M_G) = \{\omega \in \Sigma^* | (\{(d_S, 1)\}, w\$, \$) \vdash^* (\{(d_S, k)\}, \$, \$)\}$$

где (d_S, k) — заключительная вершина начальной диаграммы d_S .

Пример 6. Рассмотрим синхронный распознаватель M_{G_3} , соответствующий КС-грамматике G_3 с правилами

$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow aAb \mid e \\ B &\rightarrow aBc \mid e \end{aligned}$$

Его КС-диаграммы изображены на рис. Д.3. Для входной цепочки $ab\$$ распознаватель M_{G_3} сделает такую последовательность тактов:

$$\begin{aligned} (\{(d_S, 1)\}, ab\$, \$) &\vdash (\{(d_A, 1), (d_B, 1)\}, ab\$, \$) \\ &\vdash (\{(d_A, 2), (d_B, 2)\}, b\$, \{(d_S, 1)\}\$) \\ &\vdash (\{(d_A, 1), (d_B, 1)\}, b\$, \{(d_A, 2), (d_B, 2)\}\{(d_S, 1)\}\$) \\ &\vdash (\{(d_A, 3), (d_B, 3)\}, b\$, \{(d_S, 1)\}\$) \\ &\vdash (\{(d_A, 4)\}, \$, \{(d_S, 1)\}\$) \\ &\vdash (\{(d_S, 2)\}, \$, \$) \end{aligned}$$

Нетрудно убедиться, что $L(M_{G_3}) = L(G_3) = \{a^n b^n | n \geq 0\} \cup \{a^n c^n | n \geq 0\}$.

Распознаватель M_G назовем *адекватным* (грамматике G), если $L(M_G) = L(G)$.

Упр. 19. Покажите, что если синхронный распознаватель M_G адекватен грамматике G , не содержащей бесполезных нетерминалов, то G не леворекурсивна.

Упр. 20. Покажите, что если G —псевдоразделенная грамматика, то соответствующие простой МП-распознаватель M_G и синхронный распознаватель M'_G эквивалентны (и, значит, если G —слаборазделенная грамматика, то M'_G адекватен G).

Из упр. 20 и 3 следует, что проблема распознавания адекватности синхронного распознавателя неразрешима. Заметим еще,

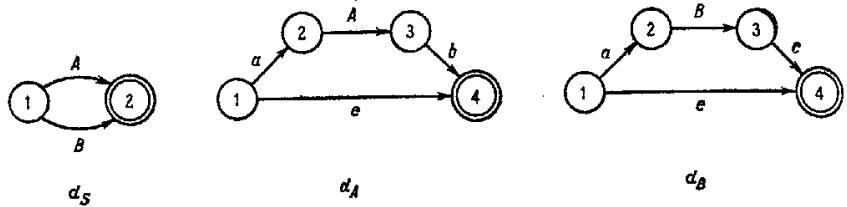


Рис. Д.3. КС-диаграммы синхронного распознавателя.

что переход от синхронного распознавателя к соответствующему анализатору нетривиален: для получения разбора может понадобиться еще один проход (об этом см. в [Алгол 68]).

5.2. ДЕТЕРМИНИРОВАННЫЙ ВОСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

В предыдущем разделе мы рассмотрели класс грамматик, для которых возможен детерминированный нисходящий синтаксический анализ с чтением входной цепочки слева направо. Существует аналогичный класс грамматик, для которых тоже возможен детерминированный анализ с чтением цепочки слева направо, но анализ восходящий. Они называются LR-грамматиками и их изложение будет во многом параллельно изложению LL-грамматик предыдущего раздела.

5.2.1. Разбор с помощью детерминированного алгоритма типа «перенос — свертка»

В гл. 4 было показано, что восходящий разбор можно провести, чередуя свертки и переосы, с помощью двух магазинов. Разбор типа „перенос — свертка“ состоит в переносе входных символов в магазин, пока в его верхней части не окажется основа. В этот момент делается свертка основы. Если не попадается ошибок, то процесс повторяется до тех пор, пока вся входная цепочка не будет прочитана, а в магазине останется один начальный символ грамматики. В гл. 4 мы изложили работающий именно таким образом алгоритм с возвратами, который мог вначале неправильно выбирать свертки для некоторых

основ, но в конце концов делал правильные выборы. В этом разделе будет изучен большой класс грамматик, для которых всегда возможен детерминированный безвозвратный анализ такого типа. Это LR(k)-грамматики, образующие наиболее широкий класс грамматик, анализируемых естественным образом снизу вверх с помощью детерминированного преобразователя с магазинной памятью. Буква L в названии LR(k)-грамматик указывает на то, что входная цепочка читается слева (left) направо, R — на то, что выдается правый (right) разбор, а k — число входных символов, на которые алгоритм „заглядывает вперед“.

В дальнейшем мы исследуем разные подклассы LR(k)-грамматик, в том числе грамматики предшествования и грамматики ограниченного правого контекста.

Пусть αx — правовыводимая цепочка какой-то грамматики, причем α — либо пустая цепочка, либо оканчивается нетерминальным символом. Тогда α называется *открытой частью* цепочки αx , а x — *замкнутой частью*. Границу между α и x назовем *рубежом*. Эти определения замкнутой и открытой части правовыводимой цепочки не следует путать с аналогичными определениями *законченной* и *незаконченной* части левовыводимой цепочки.

Алгоритм разбора типа „перенос — свертка“ можно рассматривать как программу расширенного детерминированного МП-преобразователя, который проводит разбор снизу вверх. Для данной входной цепочки w МП-преобразователь моделирует в обратном порядке ее правый вывод. Допустим, что

$$S = \alpha_0 \Rightarrow_r \alpha_1 \Rightarrow_r \dots \Rightarrow_r \alpha_m = w$$

— правый вывод цепочки w . Каждая правовыводимая цепочка α_i распределяется в памяти ДМП-преобразователя так, что ее открытая часть хранится в магазине, а замкнутая — на входной ленте справа от головки. Например, если $\alpha_i = \alpha A x$, то αA будет в магазине (причем A — верхний символ), а x — это еще не прочитанная часть первоначальной входной цепочки.

Допустим, что $\alpha_{i-1} = \gamma B z$ и на шаге $\alpha_{i-1} \Rightarrow_r \alpha_i$ применено правило $B \rightarrow \beta y$, причем $\gamma\beta = \alpha A$ и $yz = x$. Имея в магазине цепочку αA , МП-преобразователь переиесет несколько (возможно, ни одного) символов с левого конца цепочки x в магазин, пока не обнаружит правый конец основы цепочки α_i . К этому моменту в магазине будет перенесена цепочка y .

Затем МП-преобразователь должен локализовать левый конец основы. Как только он это сделает, он заменит основу (здесь ею будет цепочка βy), занимающую верхнюю часть магазина, подходящим нетерминалом (здесь нетерминалом B) и выдаст номер правила $B \rightarrow \beta y$. Теперь в магазине окажется γB ,

$a z$ образует непрочитанную часть входа. Эти цепочки являются соответственно открытой и замкнутой частями правовыводимой цепочки α_{i-1} .

Заметим, что основа цепочки αAx никогда не может лежать целиком внутри α , хотя она может быть полностью внутри x , т. е. α_{i-1} может иметь вид αAx_1Bx_2 и к ней можно применить правило вида $B \rightarrow y$, где $x_1yx_2 = x$, чтобы получить α_i . Так как цепочка x_1 может быть очень длинной, то может потребоваться много операций переноса, прежде чем α_i будет сведена к α_{i-1} .

Итак, в ходе разбора алгоритм типа „перенос—свертка“ должен принимать решения трех типов. Во-первых, перед каждым шагом надо определить, перенести ли в магазин входной символ или делать свертку. Это решение фактически состоит в выяснении того, где в правовыводимой цепочке находится правый конец основы.

Второе и третье решения надо принять после того, как локализован правый конец основы. Как только становится известным, что в верхней части магазина образовалась основа, нужно локализовать в магазине ее левый конец. Затем, когда основа уже выделена, надо найти подходящий нетерминал, которым следует ее заменить.

Грамматика, в которой никакие два различные правила не имеют одинаковых правых частей, называется *детерминированной* (или *однозначно обратимой* или просто *обратимой*). Нетрудно показать, что каждый контекстно-свободный язык порождается по крайней мере одной обратимой КС-грамматикой.

Если грамматика обратимая, то выделенную в правовыводимой цепочке основу можно заменить в точности одним нетерминалом. Однако многие полезные грамматики не обратимы, так что в общем случае нам нужен какой-то механизм для определения того, каким нетерминалом заменить основу.

Пример 5.21. Рассмотрим грамматику G с правилами

- (1) $S \rightarrow SaSb$
- (2) $S \rightarrow e$

и правый вывод

$$S \Rightarrow SaSb \Rightarrow SaSaSbb \Rightarrow SaSabb \Rightarrow Saabb \Rightarrow aabb$$

Разберем цепочку $aabb$, используя магазин и применяя операции переноса и свертки. Символ $\$$ будет служить концевым маркером входной цепочки и дна магазина.

Работу алгоритма разбора типа „перенос—свертка“ опишем в терминах конфигураций, представляющих собой тройки вида $(\alpha X, x, \pi)$, где

- (1) αX — содержимое магазина, причем X — верхний символ;
- (2) x — непрочитанная часть входной цепочки;
- (3) π — выход к данному моменту.

Эти конфигурации имеют тот же вид, что и конфигурации расширенного МП-преобразователя, только в них опущено состояние и магазин предшествует входу. В разд. 5.3.1 мы дадим формальное описание алгоритма типа „перенос—свертка“.

Вначале алгоритм находится в конфигурации $(\$, aabb\$, e)$. Ему надо догадаться, что основой правовыводимой цепочки $aabb$ является входящая в нее на левом конце пустая цепочка e и что эту основу надо свернуть к S . Отложим пока описание самого механизма, обеспечивающего распознавание основы. Итак, алгоритм должен перейти в конфигурацию $(\$S, aabb\$, 2)$. После этого он перенесет в магазин входной символ, перейдя в конфигурацию $(\$Sa, abb\$, 2)$. Затем догадается, что наверху магазина находится основа e и сделает свертку, перейдя в конфигурацию $(\$SaS, abb\$, 22)$. Продолжая в том же духе, алгоритм сделает такую последовательность тактов:

$$\begin{aligned} (\$, aabb\$, e) &\vdash (\$S, aabb\$, 2) \\ &\vdash (\$Sa, abb\$, 2) \\ &\vdash (\$SaS, abb\$, 22) \\ &\vdash (\$SaSa, bb\$, 22) \\ &\vdash (\$SaSaS, bb\$, 222) \\ &\vdash (\$SaSaSb, b\$, 222) \\ &\vdash (\$SaS, b\$, 2221) \\ &\vdash (\$SaSb, \$, 2221) \\ &\vdash (\$, \$, 22211) \\ &\vdash \text{допуск } \square \end{aligned}$$

5.2.2. LR(k)-грамматики

В этом разделе мы опишем широкий класс грамматик, для которых всегда можно построить детерминированные правые анализаторы. Он состоит из LR(k)-грамматик.

Говоря неформально, грамматика будет LR(k)-грамматикой, если для произвольного правого вывода $S = \alpha_0 \Rightarrow_r \alpha_1 \Rightarrow_r \alpha_2 \Rightarrow_r \dots \Rightarrow_r \alpha_m = z$ в каждой правовыводимой цепочке α_i , читая ее слева направо, можно выделить основу и определить, каким нетерминалом надо ее заменить, дойдя при этом не более чем до k -го символа, расположенного справа от правого конца этой основы.

Допустим, что $\alpha_{i-1} = \alpha Aw$ и $\alpha_i = \alpha\beta w$, где β — основа цепочки α_i . Допустим далее, что $\alpha\beta = X_1X_2\dots X_r$. Если КС-грамматика является LR(k)-грамматикой, можно быть уверенным в следующем:

(1) Зная $X_1 X_2 \dots X_j$ и первые k символов цепочки $X_{j+1} \dots X_\omega$, можно не сомневаться, что правый конец основы не будет достигнут до тех пор, пока не станет $j = r$ ¹⁾.

(2) Зная $\alpha\beta$ и не более k символов цепочки ω , всегда можно определить, что β — основа и ее надо свернуть к A .

(3) Если $\alpha_{i-1} = S$, то можно уверенно сигнализировать о том, что входную цепочку следует допустить.

Заметим, что, проходя по последовательности $\alpha_m, \alpha_{m-1}, \dots, \alpha_0$, мы вначале видим только цепочку $\text{FIRST}_k(\alpha_m) = \text{FIRST}_k(z)$. На каждом шаге цепочка символов, на которые мы „заглядываем вперед“ (аванцепочка), состоит из k или менее терминальных символов.

Теперь перейдем к определению $\text{LR}(k)$ -грамматики, но прежде введем простое понятие пополненной грамматики.

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Пополненной грамматикой, полученной из G , будем называть грамматику $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$. Другими словами, это просто G с новым начальным правилом $S' \rightarrow S$, где S' — новый начальный символ, не принадлежащий N . Будем считать, что $S \rightarrow S'$ — нулевое правило грамматики G' , а остальные правила занумерованы числами 1, 2, …, r . Это начальное правило добавляется для того, чтобы свертку, в которой используется нулевое правило, можно было интерпретировать как сигнал о том, что цепочка допускается.

Дадим точное определение $\text{LR}(k)$ -грамматики.

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика и $G' = (N', \Sigma, P', S')$ — полученная из нее пополненная грамматика. Будем называть G $\text{LR}(k)$ -грамматикой для $k \geq 0$, если из условий

- (1) $S' \Rightarrow_{G'}^*, \alpha A w \Rightarrow_{G'} \alpha \beta w$,
- (2) $S' \Rightarrow_{G'}^*, \gamma B x \Rightarrow_{G'} \alpha \beta y$,
- (3) $\text{FIRST}_k(w) = \text{FIRST}_k(y)$

следует, что $\alpha \beta y = \gamma B x$ (т. е. $\alpha = \gamma$, $A = B$ и $x = y$).

Грамматика G называется LR -грамматикой, если она $\text{LR}(k)$ -грамматика для некоторого $k \geq 0$.

Интуитивно это определение говорит о том, что если $\alpha\beta\omega$ и $\alpha\beta y$ — правовыводимые цепочки пополненной грамматики, у которых $\text{FIRST}_k(w) = \text{FIRST}_k(y)$, и $A \rightarrow \beta$ — последнее правило, использованное в правом выводе цепочки $\alpha\beta\omega$, то правило $A \rightarrow \beta$ должно использоваться также в правом разборе при свертке

¹⁾ Утверждение не очень четко сформулировано и, если его понимать буквально, для $\text{LR}(k)$ -грамматики неверное. — Прим. ред.

$\alpha\beta y$ к $\alpha A y$. Так как A дает β независимо от w , то $\text{LR}(k)$ -условие говорит о том, что в $\text{FIRST}_k(w)$ содержится информация, достаточная для определения того, что $\alpha\beta$ за один шаг выводится из αA . Поэтому никогда не может возникнуть сомнений относительно того, как свернуть очередную правовыводимую цепочку пополненной грамматики. Кроме того, работая с $\text{LR}(k)$ -грамматикой, мы всегда знаем, допустить ли данную входную цепочку или продолжать разбор. Если начальный символ S не встречается в правых частях правил, то в определении $\text{LR}(k)$ -грамматики вместо S' можно взять S , а именно $G = (N, \Sigma, P, S)$ будет $\text{LR}(k)$ -грамматикой, если из трех условий

- (1) $S \Rightarrow^* \alpha A w \Rightarrow, \alpha \beta w$,
- (2) $S \Rightarrow^*, \gamma B x \Rightarrow, \alpha \beta y$,
- (3) $\text{FIRST}_k(w) = \text{FIRST}_k(y)$

следует, что $\alpha \beta y = \gamma B x$.

Мы не всегда можем пользоваться этим определением, потому что если начальный символ встречается в правой части некоторого правила, то нельзя сказать, достигнут ли конец входной цепочки и ее нужно допустить, или надо продолжать разбор.

Пример 5.22. Рассмотрим грамматику G с правилами
 $S \rightarrow Sa | a$

Если игнорировать ограничение, касающееся появления начального символа в правых частях правил, и пользоваться вторым определением, то G придется считать $\text{LR}(0)$ -грамматикой.

Однако, согласно правильному определению, G не $\text{LR}(0)$ -грамматика, так как из трех условий

- (1) $S' \Rightarrow_{G'}^0 S' \Rightarrow_{G'} S$,
- (2) $S' \Rightarrow_{G'} S \Rightarrow_{G'} Sa$,
- (3) $\text{FIRST}_0(e) = \text{FIRST}_0(a) = e$

не следует, что $S'a = S$. Применяя определение к этой ситуации, имеем $\alpha = e$, $\beta = S$, $w = e$, $\gamma = e$, $A = S'$, $B = S$, $x = e$ и $y = 0$. Проблема здесь заключается в том, что нельзя установить, является ли S основой правовыводимой цепочки Sa , не видя символа, стоящего после S (т. е. наблюдая „нулевое“ количество символов). В соответствии с интуицией G не должна быть $\text{LR}(0)$ -грамматикой и она не будет ею, если пользоваться первым определением. Это определение мы и будем использовать на протяжении всей книги. □

В данном разделе мы покажем, что для каждой $\text{LR}(k)$ -грамматики $G = (N, \Sigma, P, S)$ можно построить детерминированный правый анализатор, который ведет себя следующим образом.

Прежде всего, этот анализатор строится по дополненной грамматике G' . Ведет он себя во многом так же, как анализатор типа „перенос—свертка“, введенный в примере 5.21, за исключением того, что после каждого символа грамматики в магазин будет записываться специальный информационный символ, называемый $LR(k)$ -таблицей. Эти $LR(k)$ -таблицы помогут определять,

	Действие			Переход		
	<i>a</i>	<i>b</i>	<i>e</i>	<i>S</i>	α	<i>b</i>
T_0	2	<i>X</i>	2	T_1	<i>X</i>	<i>X</i>
T_1	<i>S</i>	<i>X</i>	<i>A</i>	<i>X</i>	T_2	<i>X</i>
T_2	2	2	<i>X</i>	T_3	<i>X</i>	<i>X</i>
T_3	<i>S</i>	<i>S</i>	<i>X</i>	<i>X</i>	T_4	T_5
T_4	2	2	<i>X</i>	T_6	<i>X</i>	<i>X</i>
T_5	1	<i>X</i>	1	<i>X</i>	<i>X</i>	<i>X</i>
T_6	<i>S</i>	<i>S</i>	<i>X</i>	<i>X</i>	T_4	T_7
T_7	1	1	<i>X</i>	<i>X</i>	<i>X</i>	<i>X</i>

Рис. 5.9. $LR(1)$ -анализатор для грамматики G (i —свертка, при которой применено i -е правило, S —перенос, A —допуск, X —ошибка).

что надо делать на очередном шаге—свертку или перенос, и в случае свертки—какое правило при этом применить.

По-видимому, наилучший способ описать поведение $LR(k)$ -анализатора—это рассмотреть подходящий пример.

Возьмем грамматику G из примера 5.21, которая, как легко проверить, является $LR(1)$ -грамматикой. Полненная грамматика G' состоит из правил

- (0) $S' \rightarrow S$
- (1) $S \rightarrow SaSb$
- (2) $S \rightarrow e$

$LR(1)$ -анализатор для грамматики G приведен на рис. 5.9.

$LR(k)$ -анализатор для КС-грамматики G —это не что иное, как множество строк большой таблицы, каждая строка которой называется $LR(k)$ -таблицей. Одна строка, здесь это T_0 , выделяется в качестве начальной $LR(k)$ -таблицы. Каждая $LR(k)$ -таблица состоит из двух функций—функции действия f и функции переходов g :

(1) Аргументом функции действия f служит цепочка $u \in \Sigma^{*k}$ (она называется аванцепочкой), а соответствующее значение $f(u)$ —один из символов „действий“: перенос, свертка i , ошибка или допуск.

(2) Аргументом функции переходов g служит символ $X \in N \cup \Sigma$, а соответствующее значение $g(X)$ —либо имя некоторой $LR(k)$ -таблицы, либо ошибка.

Сейчас мы не будем объяснять, как построить такой анализатор, а отложим это до разд. 5.2.3 и 5.2.4.

LR -анализатор ведет себя так же, как алгоритм типа „перенос—свертка“, используя в процессе работы магазин, входную и выходную ленты. Вначале магазин содержит начальную $LR(k)$ -таблицу T_0 и ничего больше. На входной ленте находится анализируемая цепочка, а выходная лента вначале пустая. Если предположить, что надо разобрать входную цепочку $aabb$, то начальной конфигурацией анализатора будет $(T_0, aabb, e)$. Затем разбор будет осуществляться согласно следующему алгоритму.

Алгоритм 5.7. $LR(k)$ -алгоритм разбора.

Вход. Множество \mathcal{T} $LR(k)$ -таблиц для $LR(k)$ -грамматики $G = (N, \Sigma, P, S)$ с таблицей $T_0 \in \mathcal{T}$, выделенной в качестве начальной, и входная цепочка $z \in \Sigma^*$, которую надо разобрать.

Выход. Если $z \in L(G)$, то правый разбор цепочки z в грамматике G , в противном случае сигнал об ошибке.

Метод. Выполнять шаги (1) и (2) до тех пор, пока не будет допущена входная цепочка или не встретится сигнал об ошибке. В случае допуска цепочка на выходной ленте представляет собой правый разбор цепочки z .

(1) Определяется аванцепочка u , состоящая из k очередных входных символов (или менее чем k символов, если обрабатывается конец входной цепочки).

(2) Функция действия f таблицы, расположенной наверху магазина, применяется к аванцепочке u .

(а) Если $f(u) = \text{перенос}$, то следующий входной символ, скажем a , переносится со входа в магазин. К a применяется функция переходов g верхней таблицы магазина и определяется новая таблица, которую надо поместить наверху магазина. После этого вернуться к шагу (1). Если следующего входного символа нет или значение $g(a)$ не определено, остановиться и выдать сигнал об ошибке.

(б) Если $f(u) = \text{свертка } i$ и $A \rightarrow \alpha$ —правило с номером i , то из верхней части магазина устраняются $2|\alpha|$ символов¹⁾ и на

¹⁾ Если $\alpha = X_1 \dots X_r$, то в данный момент верхняя часть магазина имеет вид $T_0 X_1 T_1 X_2 \dots X_r T_r$. Устранив из магазина $2|\alpha|$ символов, мы устраним основу вместе с промежуточными LR -таблицами.

выходной ленте записывается номер правила i . Наверху магазина оказывается тогда новая таблица T' , и ее функция переходов применяется к A для определения следующей таблицы, которую надо поместить наверху магазина. Помещаем A и эту новую таблицу наверху магазина и переходим к шагу (1).

- (в) Если $f(u) = \text{ошибка}$, разбор прекращается (на практике надо перейти к процедуре исправления ошибок).
 (г) Если $f(u) = \text{допуск}$, остановиться и объявить цепочку, записанную на выходной ленте, правым разбором первоначальной входной цепочки. \square

Пример 5.23. Пусть алгоритм 5.7 начинает работу в начальной конфигурации $(T_0, aabb, e)$. Будем пользоваться $LR(1)$ -таблицами, приведенными на рис. 5.9. Здесь авантцепочкой служит a . Значением функции действия таблицы T_0 на цепочке a будет свертка 2, где 2 — номер правила $S \rightarrow e$. На шаге (2б) надо устранить из магазина $2|e|=0$ символов и выдать номер 2. Верхней таблицей в магазине все еще остается T_0 . Так как функция переходов таблицы T_0 на аргументе S принимает значение T_1 , то наверху магазина помещаем ST_1 — это приводит к конфигурации $(T_0ST_1, aabb, 2)$.

Пройдем еще раз через этот цикл. Авантцепочкой по-прежнему служит a . Значением функции действия таблицы T_1 на a будет **перенос**, так что символ a переносим со входа в магазин. Функция переходов таблицы T_1 на аргументе a принимает значение T_2 , так что после этого шага получится конфигурация $(T_0ST_1aT_2, abb, 2)$.

Продолжая в том же духе, LR -анализатор сделает такую последовательность тиков:

$$\begin{aligned} (T_0, aabb, e) &\vdash (T_0ST_1, aabb, 2) \\ &\vdash (T_0ST_1aT_2, abb, 2) \\ &\vdash (T_0ST_1aT_2ST_3, abb, 22) \\ &\vdash (T_0ST_1aT_2ST_3aT_4, bb, 22) \\ &\vdash (T_0ST_1aT_2ST_3aT_4ST_5, bb, 222) \\ &\vdash (T_0ST_1aT_2ST_3aT_4ST_5bT_6, b, 222) \\ &\vdash (T_0ST_1aT_2ST_3, b, 2221) \\ &\vdash (T_0ST_1aT_2ST_3bT_5, e, 2221) \\ &\vdash (T_0ST_1, e, 2221) \end{aligned}$$

Заметим, что эта последовательность шагов по существу совпадает с той, что была в примере 5.21, а $LR(1)$ -таблицы как раз поясняют способ, которым в том примере принимались решения. \square

В данном разделе будут разработаны необходимые алгоритмы, обеспечивающие автоматическое построение LR -анализатора указанного вида для каждой LR -грамматики. Мы увидим, что G является $LR(k)$ -грамматикой тогда и только тогда, когда для

нее можно построить $LR(k)$ -анализатор. Но сначала вернемся к основному определению $LR(k)$ -грамматики и изучим некоторые его следствия.

Чтобы доказать, что для $LR(k)$ -грамматики алгоритм 5.7 правильно осуществляет разбор, надо значительно развить теорию $LR(k)$ -грамматик. Убедимся сначала в том, что из определения $LR(k)$ -грамматики фактически вытекают наши интуитивные представления о том, какой должна быть детерминированно правовоизводимая грамматика. Пусть дана такая правовоизводимая цепочка $\alpha\beta\omega$ пополненной $LR(k)$ -грамматики, что $\alpha A\omega \Rightarrow^* \alpha\beta\omega$. Покажем, что после чтения $\alpha\beta$ и $\text{FIRST}_k(\omega)$ не может быть никаких недоразумений по поводу

- (1) позиций правого конца основы,
- (2) позиций левого конца основы,
- (3) свертки, которую нужно сделать для данной выделенной основы.

(1) Допустим, что существует другая правовоизводимая цепочка $\alpha\beta y$, для которой $\text{FIRST}_k(y) = \text{FIRST}_k(\omega)$, но y можно записать в виде $y_1y_2y_3$, где $B \rightarrow y_2$ — правило и $\alpha\beta y_1B y_3$ — правовоизводимая цепочка, т. е. $S' \Rightarrow^* \alpha\beta y_1B y_3 \Rightarrow^* \alpha\beta y_1y_2y_3$. Такой случай явно исключается определением $LR(k)$ -грамматики. Это станет очевидным, если положить в этом определении $x = y_3$ и $\gamma B = \alpha\beta y_1B$. Правый конец основы мог бы оказаться левее конца цепочки β , т. е. могла бы найтись другая правовоизводимая цепочка $\gamma_1B\gamma y$, для которой $B \rightarrow \gamma_2$ — правило, $\text{FIRST}_k(y) = \text{FIRST}_k(\omega)$ и $\gamma_1B\gamma y \Rightarrow^* \gamma_1\gamma_2\gamma y = \alpha\beta y$. Этот случай тоже исключается, если в определении $LR(k)$ -грамматики положить $\gamma B = \gamma_1B$ и $x = \gamma y$.

(2) Допустим теперь, что положение правого конца основы правовоизводимой цепочки известно, но есть еще сомнения относительно левого конца. Иначе говоря, допустим, что $\alpha A\omega$ и $\alpha' A'\omega$ — такие правовоизводимые цепочки, что $\text{FIRST}_k(\omega) = \text{FIRST}_k(y)$ и $\alpha A\omega \Rightarrow^* \alpha\beta\omega$, $\alpha' A'\omega \Rightarrow^* \alpha'\beta'\omega = \alpha\beta\omega$. Однако в $LR(k)$ -определении требуется, чтобы было $\alpha A = \alpha' A'$, так что $\beta = \beta'$ и $A = A'$. Поэтому левый конец основы определяется однозначно.

(3) Недоразумений типа (3) тоже не может быть, потому что, как уже отмечалось, $A = A'$. Таким образом, нетерминал, которым надо заменить основу, всегда определяется однозначно.

Дадим теперь несколько примеров LR - и не LR -грамматик.

Пример 5.24. Пусть G_1 — праволинейная грамматика с правилами

$$\begin{aligned} S &\rightarrow C \mid D \\ C &\rightarrow aC \mid b \\ D &\rightarrow aD \mid c \end{aligned}$$

Покажем, что G_1 — $LR(1)$ -грамматика¹⁾.

¹⁾ На самом деле это $LR(0)$ -грамматика.

Каждый (правый) вывод в дополненной грамматике G'_1 имеет вид
 $S' \Rightarrow S \Rightarrow C \Rightarrow^* a^i C \Rightarrow a^i b$ для $i \geq 0$

или

$S' \Rightarrow S \Rightarrow D \Rightarrow^* a^i D \Rightarrow a^i c$ для $i \geq 0$

Вернемся к определению LR(1)-грамматики и допустим, что есть выводы $S' \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha \beta w$ и $S' \Rightarrow_r^* \gamma B x \Rightarrow_r \alpha \beta y$. Тогда, так как G'_1 — праволинейная грамматика, должно быть $w = x = e$. Если $\text{FIRST}_1(w) = \text{FIRST}_1(y)$, то $y = e$. Надо теперь показать, что $\alpha A = \gamma B$, т. е. $\alpha = \gamma$ и $A = B$. Пусть $B \rightarrow \delta$ — правило, примененное при переходе от $\gamma B x$ к $\alpha \beta y$. Нужно рассмотреть следующие три случая:

Случай 1: $A = S'$ (т. е. вывод $S' \Rightarrow_r^* \alpha A w$ тривиален). Тогда $\alpha = e$ и $\beta = S$. Из вида выводов в G'_1 следует, что правовыводимую цепочку S можно вывести лишь одним способом; поэтому $\gamma = e$ и $B = S'$, что и требовалось доказать.

Случай 2: $A = C$. Тогда β — либо aC , либо b . В первом случае должно быть $B = C$, так как только в правилах для C и S правые части оканчиваются символом C . Если $B = S$, то из вида выводов в G'_1 следует, что $\gamma = e$. Тогда $\gamma B \neq \alpha \beta$. Таким образом, можно заключить, что $B = C$, $\delta = aC$ и $\gamma = \alpha$. Во втором случае ($\beta = b$) должно быть $B = C$, потому что только для C существует правило, оканчивающееся символом b . Отсюда снова непосредственно следует, что $\gamma = \alpha$ и $B = A$.

Случай 3: $A = D$. Этот случай симметричен случаю 2.

Заметим, что G_1 не является LL-грамматикой. \square

Пример 5.25. Пусть G_2 — леволинейная грамматика с правилами

$$\begin{array}{l} S \rightarrow Ab \mid Bc \\ A \rightarrow Aa \mid e \\ B \rightarrow Ba \mid e \end{array}$$

Заметим, что $L(G_2) = L(G_1)$, где G_1 — грамматика из предыдущего примера. Однако G_2 не является LR(k)-грамматикой ни для какого k .

Допустим, что G_2 — LR(k)-грамматика. Рассмотрим два правых вывода в дополненной грамматике G'_2 :

$$S' \Rightarrow_r S \Rightarrow_r^* A a^k b \Rightarrow_r a^k b$$

и

$$S' \Rightarrow_r S \Rightarrow_r^* B a^k c \Rightarrow_r a^k c$$

Эти два вывода удовлетворяют условию из определения LR(k)-грамматики при $\alpha = e$, $\beta = e$, $w = a^k b$, $\gamma = e$ и $y = a^k c$. Но так как

заключение неверно, т. е. $A \neq B$, то G_2 — не LR(k)-грамматика. Более того, так как LR(k)-условие нарушается для всех k , то G_2 — не LR-грамматика. \square

Грамматика из примера 5.25 не обратима, и хотя мы знаем, где в правовыводимой цепочке расположена основа, но, если вне правого конца основы разрешается просмотреть только ограниченное число терминальных символов, мы не всегда знаем, свернуть ли первую основу, которая является пустой цепочкой, к A или к B .

Пример 5.26. Ситуация, в которой нельзя однозначно определить положение основы, встречается в грамматике G_3 с правилами

$$\begin{array}{l} S \rightarrow AB \\ A \rightarrow a \\ B \rightarrow CD \mid aE \\ C \rightarrow ab \\ D \rightarrow bb \\ E \rightarrow bba \end{array}$$

G_3 — не LR(1)-грамматика. Это можно усмотреть, взяв два правых вывода в дополненной грамматике

$$S' \Rightarrow S \Rightarrow AB \Rightarrow ACD \Rightarrow ACbb \Rightarrow Aabb$$

и

$$S' \Rightarrow S \Rightarrow AB \Rightarrow AaE \Rightarrow Aabba$$

Если известен только первый символ цепочки w , нельзя определить, где в правовыводимой цепочке $Aabb$ находится правый конец основы — между b и w (когда $w = bb$) или справа от Aab (когда $w = ba$). Заметим, однако, что G_3 — LR(2)-грамматика. \square

Можно дать неформальное, но наглядное определение LR(k)-грамматики в терминах деревьев выводов. Грамматика G будет LR(k)-грамматикой, если, просмотрев только часть кроны дерева вывода в этой грамматике, расположенную слева от данной внутренней вершины, и часть кроны, выведенную из нее, а также следующие k терминальных символов, можно установить, какое правило было применено к этой вершине. Например, рассмотрев цепочки uv и $\text{FIRST}_k(w)$ на рис. 5.10, можно точно определить, какое правило было применено к вершине A . В отличие от этого определение LL(k)-грамматики говорит о том, что правило, примененное к A , можно определить по цепочкам u и $\text{FIRST}_k(vw)$.

Грамматика G_2 из примера 5.25 не была $LR(k)$ -грамматикой, потому что по первым k символам a нельзя было определить, какое из правил $A \rightarrow e$ и $B \rightarrow e$ применялось для вывода пустой цепочки в начале всей входной цепочки. Это нельзя решить,

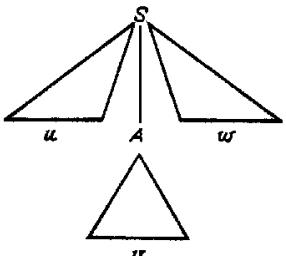


Рис. 5.10. Дерево разбора.

пока не будет виден последний входной символ, b или c . В гл. 8 мы попытаемся сделать аргументацию такого типа более строгой, но, хотя интуитивно идея понятна, ее довольно трудно формализовать.

5.2.3. Следствия определения $LR(k)$ -грамматики

Теперь займемся теорией, необходимой для конструирования $LR(k)$ -анализаторов.

Определение. Допустим, что $S \Rightarrow^* \alpha Aw \Rightarrow \alpha bw$ — правый вывод в грамматике G . Назовем цепочку γ *активным префиксом* грамматики G , если γ — префикс цепочки αb , т. е. γ — префикс некоторой правовыводимой цепочки, не выходящий за правый конец ее основы.

Ядро $LR(k)$ -анализатора составляют таблицы. Они аналогичны LL-таблицам для LL-грамматик, которые по данной авантцепочке подсказывали нам, какое правило надо применить следующим. Для $LR(k)$ -грамматики каждая таблица соответствует некоторому активному префиксу. Таблица, соответствующая активному префиксу γ , для данной авантцепочки, состоящей из k символов, сообщает о том, достигнут ли правый конец основы. Если да, то она сообщает также, какова эта основа и какое правило надо применить для ее свертки.

Здесь возникает несколько проблем. Так как цепочка γ может быть сколь угодно длинной, то неясно, можно ли обойтись конечным множеством таблиц. $LR(k)$ -условие говорит о том, что основу правовыводимой цепочки можно определить однозначно, если известен весь отрезок этой цепочки слева от основы, а также k

очередных входных символов. Поэтому не очевидно, что основу всегда можно определить, располагая только фиксированным количеством информации о цепочке, предшествующей основе. Кроме того, если $S \Rightarrow^* \alpha Aw \Rightarrow_r \alpha bw$ и можно ответить на вопрос: „Кончается ли некоторый правый вывод цепочки αbw правилом r ?“, то еще неясно, можно ли вычислить таблицы, соответствующие αA , по таблицам, соответствующим αb , способом, „реализуемым“ на МП-преобразователе (или, быть может, каким-нибудь другим удобным способом). Поэтому таблицы должны содержать достаточно информации, чтобы по таблице, соответствующей αb , можно было вычислить таблицу для αA , если $\alpha Aw \Rightarrow_r \alpha bw$. Итак, мы приходим к следующему определению.

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Будем называть $[A \rightarrow \beta_1 \cdot \beta_2, u]$ *LR(k)-ситуацией* (для данных k и G), но эти параметры не будут явно указываться, когда это не может привести к недоразумениям), если $A \rightarrow \beta_1 \beta_2$ — правило из P и $u \in \Sigma^k$. Будем говорить, что $LR(k)$ -ситуация $[A \rightarrow \beta_1 \cdot \beta_2, u]$ *допустима* для активного префикса $\alpha \beta_1$, если существует такой вывод $S \Rightarrow^* \alpha Aw \Rightarrow_r \alpha \beta_1 \beta_2 w$, что $u = FIRST_k(w)$.

Заметим, что может быть $\beta_1 = e$ и для каждого активного префикса найдется хотя бы одна допустимая для него $LR(k)$ -ситуация.

Пример 5.27. Рассмотрим грамматику G_1 из примера 5.24. Ситуация $[C \rightarrow a \cdot C, e]$ допустима для aaa , так как существует вывод $S \Rightarrow^* aaC \Rightarrow_r aaaC$, т. е. в этом примере $\alpha = aa$ и $w = e$. □

Обратите внимание на сходство данного здесь определения ситуации с тем, что встречалось раньше при описании алгоритма Эрли. Между этими двумя понятиями возникает интересная связь, если алгоритм Эрли применяется к $LR(k)$ -грамматике (см. упр. 5.2.16).

Представление о $LR(k)$ -ситуациях, соответствующих активным префиксам грамматики, дает ключ к пониманию того, как работает детерминированный правый анализатор для $LR(k)$ -грамматики. В некотором смысле основной интерес для нас представляют $LR(k)$ -ситуации вида $[A \rightarrow \beta \cdot, u]$, в которых точка находится на правом конце правила. Эти ситуации указывают, какие правила можно применить для свертки правовыводимых цепочек. В основе $LR(k)$ -разбора лежат следующие определение и теорема:

Определение. Определим функцию $EFF_k^G(\alpha)$ так (далее мы опускаем k и/или G , если ясно, о чём речь):

- (1) если α начинается терминалом, то $EFF_k(\alpha) = FIRST_k(\alpha)$,
- (2) если α начинается нетерминалом, то $EFF_k(\alpha) = \{w \mid w \in FIRST_k(\alpha) \text{ и существует вывод } \alpha \Rightarrow^* \beta \Rightarrow_r wx, \text{ где } \beta \neq Awx \text{ для } A \in N\}$.

Таким образом, $\text{EFF}_k(\alpha)$ включает все элементы множества $\text{FIRST}_k(\alpha)$, при выводе которых нетерминал, стоящий на левом конце цепочки, не заменяется пустой цепочкой e (иначе говоря, в правом выводе из цепочки α , начинающейся нетерминалом, на последнем шаге не должно применяться e -правило).

Пример 5.28. Рассмотрим грамматику G с правилами

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow Ba \mid e \\ B &\rightarrow Cb \mid C \\ C &\rightarrow c \mid e \end{aligned}$$

Здесь $\text{FIRST}_2(S) = \{e, a, b, c, ab, ac, ba, ca, cb\}$, $\text{EFF}_2(S) = \{ca, cb\}$. \square

Напомним, что в гл. 4 излагался восходящий алгоритм разбора, который не работал для грамматик, содержащих e -правила. В случае $LR(k)$ -разбора e -правила в грамматике разрешаются, но надо быть осторожным при „свертке“ пустой цепочки к нетерминалу.

Мы увидим, что функция EFF помогает правильно устанавливать, когда пустая цепочка является основой, к которой надо применить свертку. Сначала, однако, введем слегка видоизмененное определение $LR(k)$ -грамматики. Два вывода, входящие в это определение, фактически играют взаимозаменяемые роли, и, следовательно, не теряя общности, можно считать, что основа цепочки во втором выводе простирается по крайней мере так же далеко вправо, как в первом.

Лемма 5.2. Если $G = (N, \Sigma, P, S')$ — пополненная грамматика, которая не является $LR(k)$ -грамматикой, то существуют выводы $S' \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha \beta w$ и $S' \Rightarrow_r^* \gamma B x \Rightarrow_r \gamma \delta x = \alpha \beta y$, где $\text{FIRST}_k(w) = \text{FIRST}_k(y)$ и $|\gamma \delta| \geq |\alpha \beta|$, но $\gamma \delta x \neq \alpha \beta y$.

Доказательство. Согласно $LR(k)$ -определению, можно найти выводы, удовлетворяющие всем нужным условиям, за исключением, быть может, условия $|\gamma \delta| \geq |\alpha \beta|$. Допустим поэтому, что $|\gamma \delta| < |\alpha \beta|$. Покажем, что $LR(k)$ -условие нарушается для другого примера, в котором $\gamma \delta$ играет роль $\alpha \beta$ в первоначальном условии.

Так как дано, что $\gamma \delta x = \alpha \beta y$ и $|\gamma \delta| < |\alpha \beta|$, то для некоторой цепочки $z \in \Sigma^+$ можно записать $\alpha \beta = \gamma \delta z$. Таким образом, есть выводы

$$S' \Rightarrow_r^* \gamma B x \Rightarrow_r \gamma \delta x$$

и

$$S' \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha \beta w = \gamma \delta z w$$

Далее, цепочка z была определена так, что $x = zy$. Так как $\text{FIRST}_k(w) = \text{FIRST}_k(y)$, то $\text{FIRST}_k(x) = \text{FIRST}_k(zy)$. $LR(k)$ -условие, если оно выполняется, говорит, что $\alpha A w = \gamma B z w$. Поэтому „отцепляя“ от обеих сторон равенства w , получаем $\alpha A = \gamma B z$, а „прицепляя“ в новом равенстве y , получаем $\gamma B z y = \alpha A y$. Но $z y = x$, и, значит, мы показали, что $\alpha A y = \gamma B x$, а это с самого начала предполагалось ложным. Если сопоставить два приведенных выше вывода с $LR(k)$ -условием, окажется, что они удовлетворяют условиям леммы (разумеется, надо подходящим образом переименовать цепочки). \square

Метод $LR(k)$ -разбора основан на следующей теореме.

Теорема 5.9. Грамматика $G = (N, \Sigma, P, S)$ является $LR(k)$ -грамматикой тогда и только тогда, когда для каждой цепочки $u \in \Sigma^{*k}$ выполняется такое условие. Пусть $\alpha \beta$ — активный префикс правовыводимой цепочки $\alpha \beta w$ пополненной грамматики G' . Если $LR(k)$ -ситуация $[A \rightarrow \beta \cdot, u]$ допустима для $\alpha \beta$, то нет другой $LR(k)$ -ситуации $[A_1 \rightarrow \beta_1 \cdot \beta_2, v]$, допустимой для $\alpha \beta$, причем $v \in \text{EFF}_k(\beta_2 w)$. (Заметим, что β_2 может быть пустой цепочкой e .)

Доказательство. Необходимость. Пусть $[A \rightarrow \beta \cdot, u]$ и $[A_1 \rightarrow \beta_1 \cdot \beta_2, v]$ — две различные ситуации, допустимые для $\alpha \beta$, т. е. в пополненной грамматике существуют выводы

$$\begin{aligned} S' &\Rightarrow_r^* \alpha A w \Rightarrow_r \alpha \beta w \\ S' &\Rightarrow_r^* \alpha_1 A_1 x \Rightarrow_r \alpha_1 \beta_1 \beta_2 x \end{aligned}$$

причем $\text{FIRST}_k(w) = u$, $\text{FIRST}_k(x) = v$ и $\alpha \beta = \alpha_1 \beta_1$. Кроме того, $\beta_2 x \Rightarrow_r^* \gamma y$ для некоторой цепочки y и в этом (возможно, пустом) выводе нетерминал на левом конце никогда не заменяется пустой цепочкой.

Мы утверждаем, что G не может быть $LR(k)$ -грамматикой. Чтобы доказать это, исследуем три случая: (1) $\beta_2 = e$, (2) $\beta_2 \in \Sigma^+$ и (3) β_2 содержит нетерминал.

Случай 1: Если $\beta_2 = e$, то $u = v$ и выводы имеют вид

$$\begin{aligned} S' &\Rightarrow_r^* \alpha A w \Rightarrow_r \alpha \beta w \\ S' &\Rightarrow_r^* \alpha_1 A_1 x \Rightarrow_r \alpha_1 \beta_1 x \end{aligned}$$

где $\text{FIRST}_k(w) = \text{FIRST}_k(x) = u = v$. Так как рассматриваемые $LR(k)$ -ситуации различны, то либо $A \neq A_1$, либо $\beta \neq \beta_1$. В любом случае нарушается $LR(k)$ -условие.

Случай 2: Если $\beta_2 = z$ для некоторой цепочки $z \in \Sigma^+$, то

$$\begin{aligned} S' &\Rightarrow_r^* \alpha A w \Rightarrow_r \alpha \beta w \\ S' &\Rightarrow_r^* \alpha_1 A_1 x \Rightarrow_r \alpha_1 \beta_1 zx \end{aligned}$$

где $\alpha \beta = \alpha_1 \beta_1$ и $\text{FIRST}_k(zx) = u$. Но тогда G — не $LR(k)$ -грамматика, так как $\alpha A zx$ не может совпадать с $\alpha_1 A_1 x$, если $z \in \Sigma^+$.

Случай 3: Допустим, что β_2 содержит хотя бы один нетерминальный символ. Тогда $\beta_2 \Rightarrow^* u_1 Bu_3 \Rightarrow_r u_1 u_2 u_3$, где $u_1 u_2 \neq e$, так как в этом выводе нетерминал на левом конце не заменяется на e . Итак, имеем два вывода

$$S' \Rightarrow_r \alpha Aw \Rightarrow_r \alpha \beta w$$

и

$$\begin{aligned} S' &\Rightarrow^* \alpha_1 A_1 x \Rightarrow_r \alpha_1 \beta_1 \beta_2 x \\ &\Rightarrow^* \alpha_1 \beta_1 u_1 Bu_3 x \Rightarrow_r \alpha_1 \beta_1 u_1 u_2 u_3 x \end{aligned}$$

в которых $\alpha_1 \beta_1 = \alpha \beta$ и $u_1 u_2 u_3 x = uy$. В определении LR(k)-грамматики требуется, чтобы $\alpha A u_1 u_2 u_3 x = \alpha_1 \beta_1 u_1 Bu_3 x$, т. е. $\alpha A u_1 u_2 = \alpha_1 \beta_1 u_1 B$. Подставляя $\alpha \beta$ вместо $\alpha_1 \beta_1$, получаем $A u_1 u_2 = \beta u_1 B$. Но это невозможно, так как $u_1 u_2 \neq e$.

Заметим, что именно здесь используется условие, что $u \in \text{EFF}_k(\beta_2 v)$. Если бы в формулировке теоремы мы заменили EFF на FIRST , то цепочка $u_1 u_2$ могла бы быть пустой, и тогда $\alpha A u_1 u_2 u_3 x = \alpha_1 \beta_1 u_1 Bu_3 x$ (если $u_1 u_2 = e$ и $\beta = e$).

Достаточность. Допустим, что G — не LR(k)-грамматика. Тогда в пополненной грамматике есть два вывода

$$(5.2.1) \quad S' \Rightarrow_r^* \alpha Aw \Rightarrow_r \alpha \beta w$$

и

$$(5.2.2) \quad S' \Rightarrow_r^* \gamma Bx \Rightarrow_r \gamma \delta x = \alpha \beta y$$

для которых $\text{FIRST}_k(w) = \text{FIRST}_k(y) = u$, но $\alpha Ay \neq \gamma Bx$. Эти выводы можно выбрать такими, что цепочка $\alpha \beta$ будет настолько короткой, насколько это возможно.

В силу леммы 5.2 можно считать, что $|\gamma \delta| \geq |\alpha \beta|$. Пусть $\alpha_1 A_1 y_1$ — последняя цепочка в выводе

$$S' \Rightarrow_r^* \gamma Bx$$

такая, что длина ее открытой части не больше $|\alpha \beta| + 1$, т. е. $|\alpha_1 A_1| < |\alpha \beta| + 1$. Тогда вывод (5.2.2) можно записать в виде

$$(5.2.3) \quad S' \Rightarrow_r^* \alpha_1 A_1 y_1 \Rightarrow_r \alpha_1 \beta_1 \beta_2 y_1 \Rightarrow_r^* \alpha_1 \beta_1 y$$

где $\alpha_1 \beta_1 = \alpha \beta$. В силу нашего выбора цепочки $\alpha_1 A_1 y_1$ получаем $|\alpha_1| \leq |\alpha \beta| \leq |\gamma \delta|$ и, кроме того, на последнем шаге вывода $\beta_2 y_1 \Rightarrow_r^* y$ не участвует правило $B \rightarrow e$. Если бы это правило применялось последним, то $\alpha_1 A_1 y_1$ не была бы последней цепочкой вывода $S' \Rightarrow_r^* \gamma Bx$, у которой длина открытой части не превосходит $|\alpha \beta| + 1$. Таким образом, $u = \text{FIRST}_k(y)$ содержится в $\text{EFF}_k(\beta_2 y_1)$, и можно заключить, что ситуация $[A_1 \rightarrow \beta_1 \cdot \beta_2, u]$ допустима для $\alpha \beta$, где $v = \text{FIRST}_k(y_1)$.

Из существования вывода (5.2.1) вытекает, что ситуация $[A \rightarrow \beta \cdot, u]$ тоже допустима для $\alpha \beta$, так что осталось доказать, что $A_1 \rightarrow \beta_1 \cdot \beta_2$ отличается от $A \rightarrow \beta \cdot$. Итак, пусть $A_1 \rightarrow \beta_1 \cdot \beta_2$ совпадает с $A \rightarrow \beta \cdot$. Тогда вывод (5.2.3) имеет вид

$$S' \Rightarrow_r^* \alpha_1 A y \Rightarrow_r \alpha_1 \beta y$$

где $\alpha_1 \beta = \alpha \beta$. Поэтому $\alpha_1 = \alpha$ и $\alpha A y = \alpha B x$, что противоречит предположению о том, что G не LR(k)-грамматика. \square

Чтобы построить детерминированный правый анализатор для LR(k)-грамматики, надо знать, как для всех активных префиксов найти все допустимые LR(k)-ситуации.

Определение. Пусть G — КС-грамматика и γ — ее активный префикс. Определим $V_k^G(\gamma)$ как множество LR(k)-ситуаций (относительно k и G), допустимых для γ . Как и раньше, будем опускать индексы k и G , если понятно, о чём идет речь. Назовем множество $\mathcal{S} = \{A \mid A = V_k^G(\gamma)\}$ для некоторого активного префикса γ грамматики G системой множеств допустимых LR(k)-ситуаций, соответствующей грамматике G .

Теперь приведем алгоритм построения множества LR(k)-ситуаций, допустимых для произвольной заданной цепочки, а затем алгоритм построения системы всех таких множеств для любой грамматики G .

Алгоритм 5.8. Построение множества $V_k^G(\gamma)$.

Вход. КС-грамматика $G = (N, \Sigma, P, S)$ и $\gamma \in (N \cup \Sigma)^*$.

Выход. $V_k^G(\gamma)$.

Метод. Если $\gamma = X_1 X_2 \dots X_n$, то для того, чтобы построить $V_k(\gamma)$, построим $V_k(e)$, $V_k(X_1)$, $V_k(X_1 X_2)$, ..., $V_k(X_1 X_2 \dots X_n)$.

(1) Построим $V_k(e)$ так:

- (а) Если $S \rightarrow \alpha \in P$, включить $[S \rightarrow \cdot \alpha, e]$ в $V_k(e)$.
- (б) Если ситуация $[A \rightarrow \cdot B \alpha, u]$ уже включена в $V_k(e)$ и $B \rightarrow \beta \in P$, то для каждой цепочки $x \in \text{FIRST}_k(\alpha u)$ включить ситуацию $[B \rightarrow \cdot \beta, x]$ в $V_k(e)$ при условии, что там ее еще нет.
- (в) Повторять шаг (б) до тех пор, пока в $V_k(e)$ нельзя будет включить новую ситуацию.

(2) Допустим, что мы уже построили $V_k(X_1 X_2 \dots X_{i-1})$ для $i \leq n$. Построим $V_k(X_1 X_2 \dots X_i)$:

- (а) Если $[A \rightarrow \alpha \cdot X_i \beta, u] \in V_k(X_1 \dots X_{i-1})$, включить $[A \rightarrow \alpha \cdot X_i \beta, u]$ в $V_k(X_1 \dots X_i)$.
- (б) Если ситуация $[A \rightarrow \alpha \cdot B \beta, u]$ уже включена в $V_k(X_1 \dots X_i)$ и $B \rightarrow \delta \in P$, включить в $V_k(X_1 \dots X_i)$ ситуацию $[B \rightarrow \cdot \delta, x]$ для каждой цепочки $x \in \text{FIRST}_k(\beta u)$ при условии, что там ее еще нет.

(в) Повторять шаг (2б) до тех пор, пока в $V_k(X_1 \dots X_i)$ нельзя будет включить новую ситуацию. \square

Определение. Повторные применения шагов (1б) и (2б) алгоритма 5.8 к некоторому множеству ситуаций будем называть операцией **замыкания** этого множества.

Определим на множествах ситуаций грамматики $G = (N, \Sigma, P, S)$ функцию GOTO. Если \mathcal{A} — такое множество ситуаций, что $\mathcal{A} = V_k^G(\gamma)$ для некоторой цепочки $\gamma \in (N \cup \Sigma)^*$ и $X \in (N \cup \Sigma)^*$, то значением $\text{GOTO}(\mathcal{A}, X)$ будет такое множество \mathcal{A}' , что $\mathcal{A}' = V_k^G(\gamma X)$. На шаге (2) алгоритма 5.8 вычисляется

$$V_k(X_1 X_2 \dots X_i) = \text{GOTO}(V_k(X_1 X_2 \dots X_{i-1}), X_i)$$

Заметим, что шаг (2) на самом деле зависит от множества $V_k(X_1 \dots X_{i-1})$, а не от $X_1 \dots X_{i-1}$.

Пример 5.29. Построим множества $V_1(e)$, $V_1(S)$ и $V_1(Sa)$ для дополненной грамматики

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow SaSb \\ S &\rightarrow e \end{aligned}$$

(Заметим, однако, что алгоритм 5.8 не требует, чтобы грамматика была дополненной.) Сначала вычислим $V(e)$ (шаг (1) алгоритма 5.8). На шаге (1а) в $V(e)$ включим $[S' \rightarrow \cdot S, e]$. На шаге (1б) в $V(e)$ включим $[S \rightarrow \cdot SaSb, e]$ и $[S \rightarrow \cdot, e]$. Так как $[S \rightarrow \cdot SaSb, e]$ принадлежит теперь $V(e)$, то надо включить в $V(e)$ также и $[S \rightarrow \cdot SaSb, x]$, $[S \rightarrow \cdot, x]$ для всех $x \in \text{FIRST}_1(aSb) = \{a\}$. Таким образом, $V(e)$ содержит ситуации

$$\begin{aligned} [S' \rightarrow \cdot S, e] \\ [S \rightarrow \cdot SaSb, e | a] \\ [S \rightarrow \cdot, e | a] \end{aligned}$$

Здесь мы пишем сокращенно $[A \rightarrow \alpha \cdot \beta, x_1 | x_2 | \dots | x_n]$ вместо $[A \rightarrow \alpha \cdot \beta, x_1], [A \rightarrow \alpha \cdot \beta, x_2], \dots, [A \rightarrow \alpha \cdot \beta, x_n]$. Чтобы получить $V(S)$, вычислим $\text{GOTO}(V(e), S)$. На шаге (2а) в $V(S)$ включаются три ситуации: $[S' \rightarrow S \cdot, e]$ и $[S \rightarrow S \cdot aSb, e | a]$. Вычисление замыкания не дает новых ситуаций, так что $V(S)$ состоит из ситуаций

$$\begin{aligned} [S' \rightarrow S \cdot, e] \\ [S \rightarrow S \cdot aSb, e | a] \end{aligned}$$

$V(Sa)$ вычисляется как значение $\text{GOTO}(V(S), a)$ и состоит из шести ситуаций

$$\begin{aligned} [S \rightarrow Sa \cdot Sb, e | a] \\ [S \rightarrow \cdot SaSb, a | b] \\ [S \rightarrow \cdot, a | b] \quad \square \end{aligned}$$

Теперь покажем, что алгоритм 5.8 правильно вычисляет $V_k(\gamma)$.

Теорема 5.10. $\text{LR}(k)$ -ситуация принадлежит множеству $V_k(\gamma)$ после шага (2) алгоритма 5.8 тогда и только тогда, когда она допустима для γ .

Доказательство. Достаточность. Представляем читателю доказать, что алгоритм 5.8 заканчивает работу и правильно вычисляет $V_k(e)$. Мы докажем, что если все допустимые для $X_1 \dots X_{i-1}$ ситуации и только они содержатся в $V_k(X_1 X_2 \dots X_{i-1})$, то все допустимые для $X_1 \dots X_i$ ситуации и только они содержатся в $V_k(X_1 \dots X_i)$.

Предположим, что ситуация $[A \rightarrow \beta_1 \cdot \beta_2, u]$ допустима для $X_1 \dots X_i$. Тогда существует такой вывод $S \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha \beta_1 \beta_2 w$, что $\alpha \beta_1 = X_1 X_2 \dots X_i$ и $u = \text{FIRST}_k(w)$. Надо рассмотреть два случая.

Пусть $\beta_1 = \beta'_1 X_i$. Тогда ситуация $[A \rightarrow \beta'_1 \cdot X_i \beta_2, u]$ допустима для $X_1 \dots X_{i-1}$ и по предположению индукции принадлежит $V_k(X_1 \dots X_{i-1})$. На шаге (2а) алгоритма 5.8 $[A \rightarrow \beta'_1 X_i \cdot \beta_2, u]$ включается в $V_k(X_1 \dots X_i)$.

Допустим, что $\beta_1 = e$, так что в этом случае $\alpha = X_1 \dots X_i$. Так как $S \Rightarrow_r^* \alpha A w$ — правый вывод, то в нем найдется промежуточный шаг, на котором появился последний символ X_i цепочки α . Поэтому можно писать $S \Rightarrow_r^* \alpha' \gamma Y \Rightarrow_r \alpha' \gamma X_i \delta y \Rightarrow_r^* \alpha A w$, где $\alpha' \gamma = X_1 \dots X_{i-1}$, и на каждом шаге вывода $\alpha' \gamma X_i \delta y \Rightarrow_r^* \alpha A w$ правило применяется к нетерминалу, стоящему справа от явно указанного символа X_i . Тогда ситуация $[B \rightarrow \gamma \cdot X_i \delta, v]$, где $v = \text{FIRST}_k(y)$, допустима для $X_1 \dots X_{i-1}$ и по предположению индукции принадлежит $V_k(X_1 \dots X_{i-1})$. Следовательно, на шаге (2а) алгоритма 5.8 $[B \rightarrow \gamma X_i \cdot \delta, v]$ включается в $V_k(X_1 \dots X_i)$. Так как $\delta y \Rightarrow_r^* A w$, то можно найти такие нетерминалы D_1, D_2, \dots, D_m и цепочки $\theta_2, \dots, \theta_n \in (N \cup \Sigma)^*$, что δ начинается нетерминалом D_1 , $A = D_m$ и в P для каждого $1 \leq i \leq m$ есть правило $D_i \rightarrow D_{i+1} \theta_{i+1}$. Тогда в результате одного из применений шага (2б) ситуация $[A \rightarrow \cdot \beta_2, u]$ будет включена в $V_k(X_1 \dots X_i)$. Читателю предоставляем доказать, что вторая компонента u этой ситуации удовлетворяет нужному условию.

Необходимость. Предположим, что ситуация $[A \rightarrow \beta_1 \cdot \beta_2, u]$ включена в $V_k(X_1 \dots X_i)$. Индукцией по числу ситуаций, ранее включенных в $V_k(X_1 \dots X_i)$, докажем, что она допустима для $X_1 \dots X_i$.

Базис, когда в $V_k(X_1 \dots X_i)$ еще нет ситуаций, доказывается просто. В этом случае $[A \rightarrow \beta_1 \cdot \beta_2, u]$ включается в $V_k(X_1 \dots X_i)$ на шаге (2а), так что $\beta_1 = \beta'_1 X_i$ и $[A \rightarrow \beta'_1 \cdot X_i \beta_2, u]$ содержитя в $V_k(X_1 \dots X_{i-1})$. Таким образом, $S \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha \beta'_1 X_i \beta_2 w$ и $\alpha \beta'_1 = X_1 \dots X_{i-1}$. Следовательно, ситуация $[A \rightarrow \beta_1 \cdot \beta_2, u]$ допустима для $X_1 \dots X_i$.

В доказательстве шага индукции рассуждение то же, что и

для базиса, если ситуация $[A \rightarrow \beta_1 \cdot \beta_2, u]$ включается в $V_k(X_1 \dots X_i)$ на шаге (2a). Если она включается на шаге (2б), то $\beta_1 = e$ и найдется ситуация $[B \rightarrow \gamma \cdot A\delta, v]$, ранее включенная в $V_k(X_1 \dots X_i)$, причем $v \in \text{FIRST}_k(\delta v)$. По предположению индукции ситуация $[B \rightarrow \gamma \cdot A\delta, v]$ допустима для $X_1 \dots X_i$, так что существует вывод $S \Rightarrow^* \alpha' By \Rightarrow^* \alpha' \gamma A\delta y$, где $\alpha' \gamma = X_1 \dots X_i$. Тогда

$$S \Rightarrow^* X_1 \dots X_i A\delta y \Rightarrow^* X_1 \dots X_i A z \Rightarrow^* X_1 \dots X_i \beta_2 z$$

где $u = \text{FIRST}_k(z)$. Следовательно, ситуация $[A \rightarrow \cdot \beta_2, u]$ допустима для $X_1 \dots X_i$. \square

Алгоритм 5.8 дает метод построения множества LR(k)-ситуаций, допустимых для произвольного активного префикса. При построении правого анализатора для LR(k)-грамматики G нас интересуют такие множества для всевозможных активных префиксов грамматики G , а именно система множеств допустимых ситуаций для G . Так как грамматика содержит конечное множество правил, то число множеств ситуаций тоже конечно, но часто бывает очень большим. Если γ — активный префикс приводимой цепочки γw , то, как будет показано, $V_k(\gamma)$ содержит всю информацию о γ , необходимую для продолжения разбора цепочки γw .

Приведем алгоритм, обеспечивающий систематический метод вычисления множеств LR(k)-ситуаций для G .

Алгоритм 5.9. Система множеств допустимых LR(k)-ситуаций для G .

Вход. КС-грамматика $G = (N, \Sigma, P, S)$ и целое число k .

Выход. $\mathcal{S} = \{\mathcal{A} \mid \mathcal{A} = V_k(\gamma)\}$ и γ — активный префикс грамматики G .

Метод. Вначале система \mathcal{S} пуста.

(1) Поместить $V_k(e)$ в \mathcal{S} . Множество $V_k(e)$ вначале „не отмечено“.

(2) Если множество ситуаций \mathcal{A} системы \mathcal{S} не отмечено, отметить \mathcal{A} , вычислив для каждого $X \in (N \cup \Sigma)$ значение $\text{GOTO}(\mathcal{A}, X)$ (можно с помощью алгоритма 5.8). Если множество $\mathcal{A}' = \text{GOTO}(\mathcal{A}, X)$ не пусто и еще не включено в \mathcal{S} , то включить его в \mathcal{S} в качестве неотмеченного множества ситуаций.

(3) Повторять шаг (2) до тех пор, пока все множества ситуаций системы \mathcal{S} не будут отмечены. \square

Определение. Если G — КС-грамматика, то систему множеств допустимых LR(k)-ситуаций для пополненной грамматики будем называть *канонической системой* множеств LR(k)-ситуаций для G .

Заметим, что множество $\text{GOTO}(\mathcal{A}, S')$ никогда не потребуется вычислять, так как оно всегда будет пусто.

Пример 5.30. Вычислим каноническую систему множеств LR(1)-ситуаций для грамматики G , если соответствующая ей пополненная грамматика состоит из правил

$$S' \rightarrow S$$

$$S \rightarrow SaSb$$

$$S \rightarrow e$$

Начнем с вычисления $\mathcal{A}_0 = V(e)$. (Это сделано в примере 5.29.)

$$\begin{aligned}\mathcal{A}_0: \quad & [S' \rightarrow \cdot S, e] \\ & [S \rightarrow \cdot SaSb, e | a] \\ & [S \rightarrow \cdot, e | a]\end{aligned}$$

Затем вычислим $\text{GOTO}(\mathcal{A}_0, X)$ для всех $X \in \{S, a, b\}$. Обозначим $\text{GOTO}(\mathcal{A}_0, S)$ через \mathcal{A}_1 . Тогда

$$\begin{aligned}\mathcal{A}_1: \quad & [S' \rightarrow S \cdot, e] \\ & [S \rightarrow S \cdot aSb, e | a]\end{aligned}$$

Оба множества $\text{GOTO}(\mathcal{A}_0, a)$ и $\text{GOTO}(\mathcal{A}_0, b)$ пусты, так как ни a , ни b не являются активными префиксами грамматики G . Далее надо вычислить $\text{GOTO}(\mathcal{A}_1, X)$ для $X \in \{S, a, b\}$. Множества $\text{GOTO}(\mathcal{A}_1, S)$ и $\text{GOTO}(\mathcal{A}_1, b)$ пусты, а $\text{GOTO}(\mathcal{A}_1, a) = \mathcal{A}_2$, где

$$\begin{aligned}\mathcal{A}_2: \quad & [S \rightarrow Sa \cdot Sb, e | a] \\ & [S \rightarrow \cdot SaSb, a | b] \\ & [S \rightarrow \cdot, a | b]\end{aligned}$$

Продолжая в том же духе, получаем множества

$$\begin{aligned}\mathcal{A}_3: \quad & [S \rightarrow SaS \cdot b, e | a] \\ & [S \rightarrow S \cdot aSb, a | b] \\ \mathcal{A}_4: \quad & [S \rightarrow Sa \cdot Sb, a | b] \\ & [S \rightarrow \cdot SaSb, a | b] \\ & [S \rightarrow \cdot, a | b] \\ \mathcal{A}_5: \quad & [S \rightarrow SaSb \cdot, e | a] \\ \mathcal{A}_6: \quad & [S \rightarrow SaS \cdot b, a | b] \\ & [S \rightarrow S \cdot aSb, a | b] \\ \mathcal{A}_7: \quad & [S \rightarrow SaSb \cdot, a | b]\end{aligned}$$

Функция GOTO изображена в виде табл. 5.4. Заметим, что множество $\text{GOTO}(\mathcal{A}, X)$ будет всегда пусто, если в каждой ситуации из \mathcal{A} точка расположена на правом конце правила. Здесь примерами таких множеств служат \mathcal{A}_5 и \mathcal{A}_7 .

Обращаем внимание читателя на сходство между функцией GOTO в табл. 5.4 и функцией переходов LR(1)-анализатора для грамматики G , приведенной на рис. 5.9. \square

Теорема 5.11. Алгоритм 5.9 правильно вычисляет систему \mathcal{S} .

Доказательство. По теореме 5.8 достаточно доказать, что множество ситуаций \mathcal{A} помещается в \mathcal{S} тогда и только тогда, когда существует вывод $S \Rightarrow^* \alpha Aw \Rightarrow^* \alpha\beta w$, где γ — префикс цепочки $\alpha\beta$ и $\mathcal{A} = V_k(\gamma)$. Необходимость условия доказы-

Таблица 5.4

	Символ грамматики		
	S	a	b
Множество ситуаций			
\mathcal{A}_0	\mathcal{A}_1		
\mathcal{A}_1	$\overline{\mathcal{A}_2}$		
\mathcal{A}_2	$\overline{\mathcal{A}_3}$		
\mathcal{A}_3	$\overline{\mathcal{A}_4}$		
\mathcal{A}_4	$\overline{\mathcal{A}_5}$		
\mathcal{A}_5	$\overline{\mathcal{A}_6}$		
\mathcal{A}_6	$\overline{\mathcal{A}_7}$		
\mathcal{A}_7	$\overline{\mathcal{A}_1}$		

вается прямой индукцией по порядку, в котором множества ситуаций помещаются в \mathcal{S} . Достаточность доказывается не менее очевидной индукцией по длине цепочки γ . И то и другое мы оставляем читателю в качестве упражнений. \square

5.2.4. Проверка LR(k)-условия

Может оказаться, что нам надо знать, является ли интересующая нас грамматика LR(k)-грамматикой для данного k . Алгоритм, решающий эту проблему, можно построить на основе теоремы 5.9 и алгоритма 5.9.

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика и k — целое число. Множество \mathcal{A} LR(k)-ситуаций для грамматики G назовем *непротиворечивым*, если оно не содержит двух различных элементов вида $[A \xrightarrow{\beta_1} \beta_2, u]$ и $[B \xrightarrow{\beta_2} \beta_1, v]$, где $u \in \text{EFF}_k(\beta_2 v)$, причем β_2 может быть пустой цепочкой.

Алгоритм 5.10. Проверка LR(k)-условия.

Вход. КС-грамматика $G = (N, \Sigma, P, S)$ и целое число $k \geq 0$.

Выход. „Да“, если G — LR(k)-грамматика, и „нет“ в противном случае.

Метод.

(1) С помощью алгоритма 5.9 вычислить каноническую систему \mathcal{S} множеств LR(k)-ситуаций для грамматики G .

(2) Рассмотреть каждое множество LR(k)-ситуаций из \mathcal{S} и установить, непротиворечиво ли оно.

(3) Если все множества из \mathcal{S} непротиворечивы, выдать ответ „да“. В противном случае выдать „нет“, т. е. объявить, что G не LR(k)-грамматика для данного k . \square

Утверждение о корректности алгоритма 5.10 — это просто переформулировка теоремы 5.9.

Пример 5.31. Проверим, является ли грамматика из примера 5.30 LR(1)-грамматикой. Имеем $\mathcal{S} = \{\mathcal{A}_0, \dots, \mathcal{A}_7\}$. Достаточно проверить лишь множества LR(1)-ситуаций, содержащие точку на правом конце правила. Такими множествами оказываются $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_4, \mathcal{A}_5$ и \mathcal{A}_7 .

Рассмотрим \mathcal{A}_0 . Для ситуаций $[S' \rightarrow \cdot S, e]$ и $[S \rightarrow \cdot Sa Sb, e | a]$ из \mathcal{A}_0 множества $\text{EFF}(S)$ и $\text{EFF}(Sa)$ пусты, поэтому они совместимы с ситуацией $[S \rightarrow \cdot, e | a]$, т. е. \mathcal{A}_0 непротиворечиво.

Рассмотрим \mathcal{A}_1 . Здесь $\text{EFF}(aSb) = \text{EFF}(aSba) = a$, но a не является аванцепткой в ситуации $[S' \rightarrow S \cdot, e]$. Следовательно, \mathcal{A}_1 непротиворечиво.

Множества \mathcal{A}_2 и \mathcal{A}_4 непротиворечивы, потому что $\text{EFF}(Sbx)$ и $\text{EFF}(Sa Sbx)$ пусты для всех x . Множества \mathcal{A}_5 и \mathcal{A}_7 очевидно непротиворечивы.

Таким образом, все множества системы \mathcal{S} непротиворечивы и, значит, G — LR(1)-грамматика. \square

5.2.5. Детерминированные правые анализаторы для LR(k)-грамматик

В этом разделе мы неформально опишем, как по LR(k)-грамматике построить детерминированный расширенный МП-преобразователь, заглядывающий на k символов вперед, который работает как правый анализатор для этой грамматики. На описанный ранее МП-преобразователь можно смотреть как на алгоритм разбора типа „перенос — свертка“, решающий на основании состояния верхнего символа магазина и аванцепткой, делать ему перенос или свертку, а в последнем случае — какую именно свертку.

Чтобы помочь анализатору принять правильное решение, в нужных ячейках магазина будут находиться „LR(k)-таблицы“, содержащие необходимую для разбора информацию, извлеченную из соответствующего множества ситуаций. В частности, если α — префикс магазинной цепочки (верхняя часть ее расположена справа), то таблица, приписываемая самому правому символу цепочки α , получается из множества ситуаций $V_k(\alpha)$. Следовательно, построение правого анализатора состоит по существу

в нахождении LR(k)-таблиц, соответствующих множествам ситуаций.

Определение. Пусть G — КС-грамматика и \mathcal{S} — система множеств LR(k)-ситуаций для G . LR(k)-таблицей $T(\mathcal{A})$, соответствующей множеству ситуаций \mathcal{A} из \mathcal{S} , назовем пару функций (f, g) . Функцию f назовем *функцией действия*, g — *функцией переходов*. И определим их следующим образом:

(1) f отображает Σ^{*k} в множество {ошибка, перенос, допуск} ∪ {свертка i | i — номер правила из P , $i \geq 1$ }, причем

(а) $f(u) = \text{перенос}$, если $[A \rightarrow \beta_1 \cdot \beta_2, v]$ содержится в \mathcal{A} , $\beta_2 \neq e$ и $u \in \text{EFF}_k(\beta_2 v)$,

(б) $f(u) = \text{свертка } i$, если $[A \rightarrow \beta \cdot, u]$ содержится в \mathcal{A} и $A \rightarrow \beta$ — правило из P с номером i , $i \geq 1$,

(в) $f(e) = \text{допуск}$, если $[S' \rightarrow S \cdot, e]$ содержится в \mathcal{A} ,

(г) $f(u) = \text{ошибка}$ в остальных случаях.

(2) g определяет очередную применяемую таблицу, она используется сразу после переноса и свертки. Формально, g отображает $N \cup \Sigma$ в множество таблиц и сигнал ошибки. Значением $g(X)$ является таблица, соответствующая множеству $\text{GOTO}(\mathcal{A}, X)$. Если $\text{GOTO}(\mathcal{A}, X)$ пусто, то $g(X) = \text{ошибка}$.

Следует подчеркнуть, что если G — LR(k)-грамматика и \mathcal{S} — каноническая система множеств LR(k)-ситуаций для G , то по теореме 5.9 между действиями, определяемыми по правилам (1a) — (1b), не возникает конфликтов.

Будем говорить, что таблица $T(\mathcal{A})$ соответствует активному префиксу γ грамматики G , если $\mathcal{A} = V_k(\gamma)$.

Определение. Канонической системой LR(k)-таблиц для LR(k)-грамматики G назовем пару (\mathcal{T}, T_0) , где \mathcal{T} — множество LR(k)-таблиц, соответствующее канонической системе множеств LR(k)-ситуаций для G , а T_0 — LR(k)-таблица, соответствующая множеству $V_k^G(e)$.

Обычно канонический LR(k)-анализатор представляется в виде таблицы, каждая строка которой является LR(k)-таблицей.

Если LR(k)-алгоритм разбора, описанный ранее как алгоритм 5.7, использует каноническую систему LR(k)-таблиц, будем называть его *каноническим LR(k)-алгоритмом разбора* или, короче, *каноническим LR(k)-анализатором*.

Итак, опишем процесс построения по LR(k)-грамматике канонической системы LR(k)-таблиц.

Алгоритм 5.11. Построение по LR(k)-грамматике канонической системы LR(k)-таблиц.

Вход. LR(k)-грамматика $G = (N, \Sigma, P, S)$.

Выход. Каноническая система LR(k)-таблиц для G .

Метод.

(1) Построить дополненную грамматику

$$G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$$

где $S' \rightarrow S$ — нулевое правило.

(2) По G' построить каноническую систему \mathcal{S} множеств допустимых LR(k)-ситуаций для G .

(3) Взять $\mathcal{T} = \{T \mid T = T(\mathcal{A})$ для некоторого $\mathcal{A} \in \mathcal{S}\}$ в качестве множества LR(k)-таблиц. Положить $T_0 = T(\mathcal{A}_0)$, где $\mathcal{A}_0 = V_k^G(e)$. □

Пример 5.32. Построим каноническую систему LR(1)-таблиц для грамматики G , которой соответствует дополненная грамматика

$$(0) \quad S' \rightarrow S$$

$$(1) \quad S \rightarrow SaSb$$

$$(2) \quad S \rightarrow e$$

Каноническая система \mathcal{S} множеств LR(1)-ситуаций для G приведена в примере 5.30. По \mathcal{S} построим систему LR(k)-таблиц.

Вычислим таблицу $T_0 = (f_0, g_0)$, соответствующую \mathcal{A}_0 . Так как $k = 1$, то авантючками могут быть только a , b и e . Поскольку \mathcal{A}_0 содержит ситуации $[S \rightarrow \cdot, e | a]$, то $f_0(e) = f_0(a) = \text{свертка 2}$. Из остальных ситуаций, принадлежащих \mathcal{A}_0 , находим, что $f_0(b) = \text{ошибка}$ (так как $\text{EFF}(S\alpha)$ пусто). Чтобы вычислять функцию переходов g_0 , заметим, что $\text{GOTO}(\mathcal{A}_0, S) = \mathcal{A}_1$ и $\text{GOTO}(\mathcal{A}_0, X) = \emptyset$ в остальных случаях. Если T_1 обозначает $T(\mathcal{A}_1)$, то $g_0(S) = T_1$ и $g_0(X) = \text{ошибка}$ для остальных X . Вычисление таблицы T_0 закончено. Она имеет вид

			f_0			g_0		
			a	b	e	S		
			—	—	—	a	b	e
T_0	2	X	2	X	2	T_1	X	X

Здесь 2 обозначает свертку, в которой применено правило 2, а X — символ ошибки.

Найдем теперь элементы таблицы $T_1 = (f_1, g_1)$. Так как $[S' \rightarrow S \cdot, e] \in \mathcal{A}_1$, то $f_1(e) = \text{допуск}$, и так как $[S \rightarrow S \cdot aSb, e | a] \in \mathcal{A}_1$, то $f_1(a) = \text{перенос}$. Заметим, что авантючки последних ситуаций здесь не играют роли. Далее, $f_1(b) = \text{ошибка}$, и, так как $\text{GOTO}(\mathcal{A}_1, a) = \mathcal{A}_2$, полагаем $g_1(a) = T_2$, где $T_2 = T(\mathcal{A}_2)$. □

Продолжая в том же духе, получаем систему LR(1)-таблиц, приведенную на рис. 5.9. □

В гл. 7 мы изложим другие методы построения LR(k)-анализаторов по грамматикам. Эти методы часто дают анализаторы меньшего объема, чем канонический LR(k)-анализатор. Однако

последний обладает рядом замечательных свойств, которые будут служить эталоном для сравнения с другими LR(k)-анализаторами. Отметим несколько таких свойств.

(1) Простой индукцией по числу сделанных шагов можно показать, что каждая таблица, находящаяся в магазине, соответствует цепочке символов грамматики, расположенной слева от нее. Поэтому, как только первые k символов необработанной части входной цепочки оказываются такими, что нет суффикса, который вместе с обработанной частью давал бы цепочку, принадлежащую $L(G)$, анализатор сразу сообщает об ошибке. В каждый момент его работы цепочка символов грамматики, хранящаяся в магазине, должна быть активным префиксом грамматики. Поэтому LR(k)-анализатор объявляет об ошибке при первой же возможности в ходе считывания входной цепочки слева направо.

(2) Пусть $T_j = (f_j, g_j)$. Если $f_j(u) = \text{переис}$ и анализатор находится в конфигурации

$$(5.2.4) \quad (T_0 X_1 T_1 X_2 T_2 \dots X_j T_j, x, \pi)$$

то найдется ситуация $[B \rightarrow \beta_1 \cdot \beta_2, v]$, допустимая для $X_1 X_2 \dots X_j$, причем $u \in \text{EFF}(\beta_2 v)$. Поэтому если $S' \Rightarrow^* X_1 X_2 \dots X_j u y$ для некоторой цепочки $y \in \Sigma^*$, то по теореме 5.9 правый конец основы цепочки $X_1 X_2 \dots X_j u y$ должен быть где-то справа от символа X_j .

(3) Если в конфигурации (5.2.4) $f_j(u) = \text{свертка } i$ и $A \rightarrow Y_1 Y_2 \dots Y_r$ — правило с номером i , то цепочка $X_{j-r+1} X_{j-r+2} \dots X_j$ в магазине должна совпадать с $Y_1 \dots Y_r$, так как множество ситуаций, по которому построена таблица T_j , содержит ситуацию $[A \rightarrow Y_1 Y_2 \dots Y_r \cdot, u]$. Таким образом, на шаге свертки не обязательно рассматривать символы верхней части магазина, надо только выбросить из магазина $2r$ символов.

(4) Если $f_j(u) = \text{допуск}$, то $u = e$. Содержимое магазина в этот момент имеет вид $T_0 S T$, где T — LR(k)-таблица, соответствующая множеству ситуаций, в которое входит $[S' \rightarrow S \cdot, e]$.

(5) Можно построить ДМП-преобразователь с концевым маркером, реализующий канонический LR(k)-алгоритм разбора. Действительно, так как аванцепочку можно хранить в конечной памяти управляющего устройства ДМП-преобразователя, то ясно, как построить расширенный ДМП-преобразователь, реализующий алгоритм 5.7 (LR(k)-алгоритм разбора).

Доказательство этих фактов оставляем в качестве упражнений. По существу это переформулировки определений допустимой ситуации и LR(k)-таблицы. Таким образом, справедлива следующая теорема.

Теорема 5.12. Канонический LR(k)-алгоритм разбора правильно находит правый разбор входной цепочки, если он существует, и объявляет об ошибке в противном случае.

Доказательство. На основании сформулированных выше свойств индукцией по числу шагов, сделанных алгоритмом разбора, можно показать, что если α — цепочка символов грамматики, находящихся в магазине, и x — необработанная часть входной цепочки, включающая аванцепочку, то $\alpha x \Rightarrow_\pi w$, где w — первоначальная входная цепочка и π^R — текущая выходная цепочка. В качестве частного случая получаем, что если алгоритм допускает цепочку w и выдает на выходе π^R , то $S \Rightarrow_\pi w$. \square

Из LR(1)-условия непосредственно следует однозначность LR-грамматики. Пусть даны два различных правых вывода

$$S \Rightarrow_r \alpha_1 \Rightarrow_r \dots \Rightarrow_r \alpha_n \Rightarrow_r w \text{ и } S \Rightarrow_r \beta_1 \Rightarrow_r \dots \Rightarrow_r \beta_m \Rightarrow_r w.$$

Рассмотрим наименьшее из чисел i , для которых $\alpha_{n-i} \neq \beta_{m-i}$. Сразу получаем, что LR(k)-условие нарушается для любого k . Детали оставляем в качестве упражнения.

Итак, канонический LR(k)-алгоритм разбора для LR(k)-грамматики G выдает правый разбор входной цепочки w тогда и только тогда, когда $w \in L(G)$. Вначале не совсем очевидно, что канонический LR(k)-анализатор выполняет свою работу за линейное время, даже если элементарными операциями считать его собственные шаги. Докажем, что это действительно так.

Теорема 5.13. Канонический LR(k)-алгоритм при разборе входной цепочки длины n делает $O(n)$ шагов.

Доказательство. Определим C -конфигурации рассматриваемого анализатора:

- (1) начальная конфигурация является C -конфигурацией,
- (2) конфигурация, непосредственно следующая за шагом переноса, является C -конфигурацией,
- (3) конфигурация, непосредственно следующая за сверткой, которая делает магазин короче, чем в предыдущей C -конфигурации, является C -конфигурацией.

При разборе входной цепочки длины n анализатор может оказаться не более чем в $2n$ C -конфигурациях. Назовем *характеристикой* C -конфигурации число, равное сумме числа символов грамматики в магазине и удвоенного числа необработанных входных символов. Если C_1 и C_2 — последовательные C -конфигурации, то характеристика конфигурации C_1 по крайней мере на единицу больше характеристики конфигурации C_2 . Так как характеристика начальной конфигурации равна $2n$, то количество всех C -конфигураций не превосходит $2n$.

Теперь достаточно показать, что найдется такая константа c , что между двумя последовательными C -конфигурациями анализатор может делать не более c шагов. Для этого рассмотрим

ДМП-автомат, работающий с магазином так же, как LR(k)-анализатор. Из доказательства теоремы 2.22. извлечем такую константу c , что если ДМП-автомат за c шагов не сделает переноса входного символа и не уменьшил объем магазина, то он зацикливается. Следовательно, то же произойдет и с анализатором.

Но, как мы заметили раньше, если обработанную часть входной цепочки нельзя продолжить до цепочки, принадлежащей $L(G)$, анализатор сигнализирует об ошибке. Поэтому зацикливание анализатора означало бы, что найдется цепочка из $L(G)$, имеющая сколь угодно длинные правые выводы. Это противоречит однозначности LR(k)-грамматики. Таким образом можно заключить, что анализатор не зацикливается и нужная константа существует. \square

5.2.6. Реализация LL(k)- и LR(k)-анализаторов

На первый взгляд кажется, что при реализации LL(k)- и LR(k)-анализаторов придется помещать в магазин большие таблицы. На самом деле этого можно избежать следующим образом:

(1) Поместить в память по одному экземпляру каждой таблицы, а в магазине заменить сами таблицы указателями на их место в памяти.

(2) Так как в LL(k)- и LR(k)-таблицах есть ссылки на другие таблицы, вместо имен таблиц можно использовать указатели.

Заметим также, что наличие в магазине символов грамматики излишне и на практике их можно туда не записывать.

УПРАЖНЕНИЯ

5.2.1. Определите, какие из следующих грамматик являются LR(1)-грамматиками:

- (а) G_0 ,
- (б) $S \rightarrow AB, A \rightarrow 0A1|e, B \rightarrow 1B|1,$
- (в) $S \rightarrow 0S1|A, A \rightarrow 1A|1,$
- (г) $S \rightarrow S+A|A, A \rightarrow (S)|a(S)|a.$

Последняя из этих грамматик порождает скобочные выражения с операцией + и идентификаторами, обозначенными символом a , возможно с индексом.

5.2.2. Какие из грамматик упр. 5.2.1 являются LR(0)-грамматиками?

5.2.3. Постройте системы LR(1)-таблиц для тех грамматик из упр. 5.2.1, которые являются LR(1)-грамматиками. Не забудьте сначала пополнить эти грамматики.

5.2.4. Напишите последовательность шагов, которую сделает LR(1)-анализатор для грамматики G_0 при входе $(a+a)^*(a+(a+a)*a)$.

***5.2.5.** Докажите или опровергните каждое из следующих утверждений:

(а) Каждая праволинейная грамматика является LL-грамматикой.

(б) Каждая праволинейная грамматика является LR-грамматикой.

(в) Каждая регулярная грамматика является LL-грамматикой.

(г) Каждая регулярная грамматика является LR-грамматикой.

(д) Каждое регулярное множество определяется LL(1)-граммикой.

(е) Каждое регулярное множество определяется LR(1)-граммикой.

(ж) Каждое регулярное множество определяется LR(0)-граммикой.

5.2.6. Покажите, что каждая LR-грамматика однозначная.

***5.2.7.** Для $G = (N, \Sigma, P, S)$ определим $G_R = (N, \Sigma, P_R, S)$, где P_R — это множество P , правые части правил которого записаны в обратном порядке, т. е. $P_R = \{A \rightarrow \alpha^R \mid A \rightarrow \alpha \in P\}$. Приведите пример, показывающий, что G_R не обязательно будет LR(k)-грамматикой, даже если G такова.

***5.2.8.** Пусть $G = (N, \Sigma, P, S)$ — произвольная КС-грамматика. Для $i \in \Sigma^*$ и номера правила i определим множество $R_k^G(i, u) = \{\alpha\beta u \mid S \Rightarrow^* \alpha A w \Rightarrow^* \alpha\beta w\}$, где $u = \text{FIRST}_k(w)$ и $A \rightarrow \beta$ — i -е правило. Покажите, что $R_k^G(i, u)$ — регулярное множество.

5.2.9. Постройте новый алгоритм разбора для LR(k)-грамматик, который для всех i и u следит за состояниями конечного автомата, распознающего $R_k^G(i, u)$.

***5.2.10.** Покажите, что G является LR(k)-грамматикой тогда и только тогда, когда для всех α, β, u и v соотношения $\alpha \in R_k^G(i, u)$ и $\alpha\beta \in R_k^G(j, v)$ влечут за собой $\beta = e$ и $i = j$.

***5.2.11.** Покажите, что G является LR(k)-грамматикой тогда и только тогда, когда она однозначна и для всех w, x, y и z из Σ^* выполнение четырех условий $S \Rightarrow^* wAy$, $A \Rightarrow^* x$, $S \Rightarrow^* wxz$ и $\text{FIRST}_k(y) = \text{FIRST}_k(z)$ влечет за собой $S \Rightarrow^* wAz$.

***5.2.12.** Докажите неразрешимость проблемы: является ли КС-грамматика LR(k)-грамматикой для какого-нибудь k ?

****5.2.13.** Докажите неразрешимость проблемы: является ли LR(k)-грамматика LL-грамматикой?

5.2.14. Докажите разрешимость проблемы: является ли $LR(k)$ -грамматика $LL(k)$ -грамматикой для того же значения k .

***5.2.15.** Покажите, что каждый КС-язык, не содержащий пустой цепочки e , порождается обратимой КС-грамматикой без e -правил.

***5.2.16.** Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика и $w = a_1 \dots a_n$ — цепочка из Σ^n . Допустим, что, применяя к G алгоритм Эрли, мы обнаружим в списке I_i ситуацию $[A \rightarrow \alpha \cdot \beta, j]$ (в смысле алгоритма Эрли). Покажите, что существует такой вывод $S \Rightarrow^* \gamma x$, что ситуация $[A \rightarrow \alpha \cdot \beta, u]$ (в смысле $LR(k)$ -алгоритма) допустима для γ , причем $u = FIRST_k(x)$ и $\gamma \Rightarrow^* a_1 \dots a_i$.

***5.2.17.** Докажите утверждение, обратное утверждению в упр. 5.2.16.

***5.2.18.** С помощью упр. 5.2.16 покажите, что если G — $LR(k)$ -грамматика, то алгоритм Эрли, заглядывающий на k символов вперед (см. упр. 4.2.17), тратит линейное время и емкость памяти.

5.2.19. Пусть $X \in N \cup \Sigma$. Покажите, что $EFF_k(X\alpha) = EFF_k(X) \oplus FIRST_k(\alpha)$.

5.2.20. Используя упр. 5.2.19, дайте эффективный алгоритм вычисления $EFF(\alpha)$ для любой цепочки α .

5.2.21. Доведите до формальных деталей доказательство того, что в случаях 1 и 3 теоремы 5.9 нарушается $LR(k)$ -условие.

5.2.22. Дополните доказательство теоремы 5.11.

5.2.23. Докажите корректность алгоритма 5.10.

5.2.24. Докажите корректность алгоритма 5.11, обосновав каждое из замечаний, следующих за описанием этого алгоритма.

В гл. 8 будут доказаны разные теоремы, касающиеся LR -грамматик. Читатель, возможно, захочет уже сейчас испытать себя на некоторых из них (ур. 5.2.25—5.2.28).

****5.2.25.** Докажите, что каждая $LL(k)$ -грамматика является $LR(k)$ -грамматикой.

****5.2.26.** Докажите, что каждый детерминированный КС-язык определяется $LR(1)$ -грамматикой.

***5.2.27.** Покажите, что существуют (детерминированно) проверяющие грамматики, которые не являются LR -грамматиками.

***5.2.28.** Покажите, что существуют LR -языки, которые не являются LL -языками.

****5.2.29.** Докажите, что каждая $LC(k)$ -грамматика является $LR(k)$ -грамматикой.

****5.2.30.** Каково максимальное число множеств допустимых ситуаций $LR(k)$ -грамматики, рассматриваемое как функция от числа символов грамматики, количества правил и длины самого длинного правила?

***5.2.31.** Назовем $LR(k)$ -ситуацию *существенной*, если точка находится в ней не на левом конце (т. е. она появляется на шаге (2a) алгоритма 5.8). Покажите, что если отвлечься от множества ситуаций, соответствующего e и от „сверток“ пустой цепочки, то в определении $LR(k)$ -таблицы, соответствующей множеству ситуаций, можно ограничиться существенными ситуациями и при этом таблица не изменится.

***5.2.32.** Покажите, что действием $LR(1)$ -таблицы для символа a будет *перенос* тогда и только тогда, когда в некоторой ситуации из множества, по которому построена таблица, a находится непосредственно справа от точки.

Упражнения на программирование

5.2.33. Напишите программу, проверяющую, является ли произвольная грамматика $LR(1)$ -грамматикой. Оцените время и емкость памяти, требуемые Вашей программой, как функции от объема входной грамматики.

5.2.34. Напишите программу, использующую для разбора входных цепочек $LR(1)$ -таблицу, такую, как на рис. 5.9.

5.2.35. Напишите программу, которая строит $LR(1)$ -анализатор для $LR(1)$ -грамматики.

5.2.36. Постройте $LR(1)$ -анализатор для какой-нибудь небольшой грамматики.

***5.2.37.** Напишите программу, проверяющую, образует ли произвольная система $LR(1)$ -таблиц корректный анализатор для данной КС-грамматики.

Допустим, что $LR(1)$ -анализатор находится в конфигурации $(\alpha T, ax, \pi)$ и таблица T ассоциирует с символом a операцию *ошибка*. Как и при LL -разборе, нам хотелось бы в этот момент объявить об ошибке и перейти к процедуре исправления ошибок, которая должна модифицировать входную цепочку и/или содержимое магазина так, чтобы можно было продолжить $LR(1)$ -анализ. Как и в случае LL -анализа, можно устранить входной символ, изменить его или вставить другой входной символ в зависимости от того, какая стратегия кажется наиболее подходящей в рассматриваемой ситуации.

Лейниус [1970] описывает более изощренную стратегию, в которой участвуют LR(1)-таблицы, хранящиеся в магазине.

5.2.38. Напишите LR(1)-грамматику для какого-нибудь простого языка. Разработайте процедуру исправления ошибок, которую можно было бы использовать вместе с LR(1)-анализатором для этой грамматики. Оцените эффективность Вашей процедуры.

Замечания по литературе

LR(k)-грамматики впервые были определены Кнутом [1965]. Метод построения LR-анализатора, описанный в этом разделе, дает, к сожалению, очень большие по объему анализаторы для грамматик, представляющих практический интерес. В гл. 7 мы изучим технику, предложенную Де Ремером [1969], Кореняком [1969] и Ахо и Ульманом [1971], которая часто позволяет строить LR-анализаторы меньшего объема.

Идея LR(k)-грамматики была распространена Уолтерсом [1970] на контекстно-зависимые грамматики¹⁾.

Ответы к упр. 5.2.8—5.2.10 можно найти в книге Хопкрофта и Ульмана [1969]. Упр. 5.2.12 взято из работы Кнута [1965]²⁾.

5.3. ГРАММАТИКИ ПРЕДШЕСТВОВАНИЯ

Среди грамматик, анализируемыми алгоритмами типа „перенос—свертка“, можно выделять различные подклассы LR(k)-грамматик. В этом разделе мы точно определим алгоритм разбора типа „перенос—свертка“ и рассмотрим класс грамматик предшествования, которые анализируются легко реализуемым алгоритмом этого типа.

5.3.1. Формальное определение алгоритма типа „перенос—свертка“

Определение. Пусть $G = (N, \Sigma, P, S)$ —КС-грамматика, правила которой занумерованы числами от 1 до p . Алгоритмом типа „перенос—свертка“ для грамматики G назовем пару функций $\mathcal{A} = (f, g)$ ³⁾, где f называется функцией переноса, а g —функцией свертки. Функции f и g определяются так:

(1) f отображает $V^* \times (\Sigma \cup \{\$\})^*$ в множество {перенос, свертка, ошибка, допуск}, где $V = N \cup \Sigma \cup \{\$\}$, а $\$$ —новый символ, концевой маркер.

¹⁾ В этой связи см. также работы Гончаровой [1975] и Шумея и Зониса [1975].—Прим. перев.

²⁾ Сложность проверки LR(k)-условия изучается в интересной работе Хайта и др. [1975].—Прим. перев.

³⁾ Это не те функции, которые ассоциируются с LR (k)-таблицей.

(2) g отображает $V^* \times (\Sigma \cup \{\$\})^*$ в множество {1, 2, …, p , ошибка} при условии, что если $g(\alpha, w) = i$, то правая часть i -го правила является суффиксом цепочки α .

Алгоритм типа „перенос—свертка“ использует входную ленту, читаемую слева направо, и магазин. На основе того, что находится в магазине и осталось не обработанным на входе, функция f решает, перенести ли текущий входной символ в магазин или вызвать процедуру свертки. Если принимается последнее из этих решений, то функция g решает, какую сделать свертку.

Работу алгоритма типа „перенос—свертка“ можно рассматривать в терминах конфигураций, т. е. троек вида

$$(\$X_1\dots X_m, a_1\dots a_n\$, p_1\dots p_r)$$

где

(1) $\$X_1\dots X_m$ —содержимое магазина, причем X_m —верхний символ, $X_i \in N \cup \Sigma$ и $\$$ служит маркером дна магазина,

(2) $a_1\dots a_n$ —оставшаяся необработанной часть первоначальной входной цепочки, a_1 —текущий входной символ, $\$$ служит правым концевым маркером для входа,

(3) $p_1\dots p_r$ —цепочка номеров правил, примененных при свертке первоначальной входной цепочки к цепочке $X_1\dots X_m a_1\dots a_n$.

Один шаг алгоритма \mathcal{A} можно описать с помощью двух отношений $\vdash^s_{\mathcal{A}}$ и $\vdash^r_{\mathcal{A}}$, определенных на конфигурациях (индекс \mathcal{A} в этих обозначениях будем опускать всюду, где можно):

(1) если $f(\alpha, aw) = \text{перенос}$, то $(\alpha, aw, \pi) \vdash^s (\alpha a, w, \pi)$ для $\alpha \in V^*$, $w \in (\Sigma \cup \{\$\})^*$ и $\pi \in \{1, \dots, p\}^*$,

(2) если $f(\alpha\beta, w) = \text{свертка}$, $g(\alpha\beta, w) = i$ и $A \rightarrow \beta$ —правило с номером i , то $(\alpha\beta, w, \pi) \vdash^r (\alpha A, w, \pi i)$,

(3) если $f(\alpha, w) = \text{допуск}$, то $(\alpha, w, \pi) \vdash^s \text{допуск}$,

(4) в остальных случаях $(\alpha, w, \pi) \vdash^s \text{ошибка}$.

Определим отношение \vdash как объединение отношений \vdash^s и \vdash^r . Отношения \vdash^+ и \vdash^* определяются, как обычно.

Для $w \in \Sigma^*$ положим $\mathcal{A}(w) = \pi$, если $(\$, w\$, e) \vdash^* (\$, \$, \pi) \vdash \text{допуск}$, и $\mathcal{A}(w) = \text{ошибка}$, если такого π нет.

Будем называть алгоритм \mathcal{A} корректным для грамматики G , если

(1) $L(G) = \{w \mid \mathcal{A}(w) \neq \text{ошибка}\}$,

(2) π —правильный разбор цепочки w , если $\mathcal{A}(w) = \pi$.

Пример 5.33. Построим алгоритм типа „перенос—свертка“ $\mathcal{A} = (f, g)$ для грамматики G с правилами

(1) $S \rightarrow SaSb$

(2) $S \rightarrow e$

Функцию переноса f зададим так: для всех $\alpha \in V^*$ и $x \in (\Sigma \cup \{\$\})^*$

- (1) $f(\alpha S, cx) = \text{перенос}$, если $c \in \{a, b\}$,
- (2) $f(\alpha c, dx) = \text{свертка}$, если $c \in \{a, b\}$ и $d \in \{a, b\}$,
- (3) $f(\$, ax) = \text{свертка}$,
- (4) $f(\$, bx) = \text{ошибка}$,
- (5) $f(\alpha X, \$) = \text{ошибка}$ для $X \in \{S, a\}$,
- (6) $f(\alpha b, \$) = \text{свертка}$,
- (7) $f(\$S, \$) = \text{допуск}$,
- (8) $f(\$, \$) = \text{ошибка}$.

Функцию свертки g зададим так: для всех $\alpha \in V^*$ и $x \in (\Sigma \cup \{\$\})^*$

- (1) $g(\$, ax) = 2$,
- (2) $g(\alpha a, cx) = 2$ для $c \in \{a, b\}$,
- (3) $g(\$SaSb, cx) = 1$ для $c \in \{a, \$\}$,
- (4) $g(\alpha aSaSb, cx) = 1$ для $c \in \{a, b\}$,
- (5) $g(\alpha, x) = \text{ошибка}$ в остальных случаях.

Разберем с помощью алгоритма \mathcal{A} входную цепочку $aabb$. Он приступает к работе в начальной конфигурации

$(\$, aabb\$, e)$

Первый шаг определяется значением $f(\$, aabb\$)$, которое, как видно из определения функции f , равно **свертке**. О том, какую делать свертку, говорит значение $g(\$, aabb\$)$, равное, очевидно, 2. Поэтому первым шагом будет свертка

$(\$, aabb\$, e) \vdash^r (\$, aabb\$, 2)$

Следующий шаг определяется значением $f(\$, aabb\$) = \text{перенос}$. Таким образом, это шаг переноса

$(\$, aabb\$, 2) \vdash^s (\$, abb\$, 2)$

Продолжая в том же духе, алгоритм \mathcal{A} сделает такую последовательность шагов:

$(\$, aabb\$, e) \vdash^r (\$, aabb\$, 2)$
 $\vdash^s (\$, abb\$, 2)$
 $\vdash^r (\$, abb\$, 2)$
 $\vdash^s (\$, bb\$, 2)$
 $\vdash^r (\$, bb\$, 2)$
 $\vdash^s (\$, b\$, 2)$
 $\vdash^r (\$, b\$, 2)$
 $\vdash^s (\$, \$, 2)$
 $\vdash^r (\$, \$, 2)$
 $\vdash^s \text{допуск}$

Таким образом, $\mathcal{A}(aabb) = 22211$. Ясно, что 22211 — правый разбор цепочки $aabb$. \square

На практике мы не хотим на каждом шаге рассматривать всю цепочку в магазине и всю необработанную часть входной цепочки, чтобы выяснить, каким должен быть очередной шаг алгоритма разбора. Обычно нам хочется, чтобы функция переноса зависела только от небольшого числа верхних символов магазина и от небольшого числа следующих входных символов. Аналогично хотелось бы, чтобы функция свертки зависела только от символов магазина, расположенных в нем не ниже, чем на один или два символа от сворачиваемой основы, и от одного или двух следующих входных символов.

Заметим, что в предыдущем примере функция f зависит фактически только от верхнего символа магазина и следующего входного символа, а функции g требуется только один символ ниже основы и один следующий входной символ.

$LR(k)$ -анализатор, построенный в предыдущем разделе, можно рассматривать как алгоритм типа „перенос—свертка“, в котором магазинный алфавит дополнен $LR(k)$ -таблицами. Если трактовать его так, то можно заметить, что его функция переноса зависит только от верхнего символа магазина (текущей $LR(k)$ -таблицы) и следующих k входных символов. Функция свертки зависит только от таблицы, расположенной в магазине непосредственно ниже основы, и не зависит от следующих входных символов. Однако, согласно данному нами формальному определению, надо исследовать все содержимое магазина, чтобы выяснить, какова верхняя таблица. Алгоритмам, излагаемым в этом разделе, требуется только та информация, которая хранится „почти“ на самом верху магазина. Поэтому мы принимаем такое соглашение:

Соглашение. Если f и g — функции алгоритма разбора типа „перенос—свертка“ и значение $f(\alpha, w)$ определено, то $f(\beta\alpha, wx) = f(\alpha, w)$ для всех β и x , если не оговорено противное. Аналогичное утверждение касается функции g .

5.3.2. Грамматики простого предшествования

Простейший класс алгоритмов типа „перенос—свертка“ основан на так называемых „отношениях предшествования“. Для грамматики предшествования границы основы правовыводимой цепочки можно определить с помощью некоторых отношений (предшествования) между символами, входящими в эту цепочку.

Техника разбора, ориентированная на отношения предшествования, использовалась при построении анализаторов для языков программирования одной из первых, и с тех пор в литературе появилось несколько вариантов грамматик предшествования. Мы рассмотрим основанный на отношениях предшествования детерминированный анализ слева направо, производящий правый

разбор. При этом будут введены грамматики предшествования следующих типов:

- (1) простого предшествования,
- (2) расширенного предшествования,
- (3) слабого предшествования,
- (4) смешанной стратегии предшествования,
- (5) операторного предшествования.

Ключ к пониманию метода разбора по отношениям предшествования дает определение отношения \triangleright между символами грамматики, которое при просмотре слева направо правовыводимой цепочки $\alpha\beta\omega$, где β — основа, впервые оказывается выполненным для последнего символа цепочки β и первого символа цепочки ω .

Если для разбора применяется алгоритм типа „перенос — свертка“, то всякий раз, когда между верхним символом магазина и первым из необработанных входных символов выполняется отношение \triangleright , принимается решение о свертке; в противном случае делается перенос.

Таким образом, с помощью отношения \triangleright локализуется правый конец основы правовыводимой цепочки. Локализация левого конца основы и определение нужной свертки осуществляется разными способами в зависимости от используемого типа предшествования.

Техника анализа, основанная на так называемом „простом предшествовании“, использует для выделения основы правовыводимой цепочки $\alpha\beta\omega$ три отношения предшествования: \triangleleft , \equiv и \triangleright . Если β — основа, то между всеми смежными символами цепочки α выполняется либо отношение \triangleleft , либо \equiv , между последним символом цепочки α и первым символом цепочки β выполняется \triangleleft , между смежными символами основы — отношение \equiv и между последним символом цепочки β и первым символом цепочки ω — отношение \triangleright .

Итак, основу правовыводимой цепочки грамматики простого предшествования можно выделить, просматривая эту цепочку слева направо до тех пор, пока впервые не встретится отношение \triangleright . Для нахождения левого конца основы надо возвращаться назад, пока не встретится отношение \triangleleft . Цепочка, заключенная между \triangleleft и \triangleright , будет основой. Если грамматика предполагается обратимой, то основу можно однозначно свернуть. Этот процесс продолжается до тех пор, пока входная цепочка не свернется к начальному символу (либо пока дальнейшие свертки окажутся невозможными).

Определение. Отношения предшествования Вирта—Вебера \triangleleft , \equiv и \triangleright для КС-грамматики $G = (N, \Sigma, P, S)$ определяются на множестве $N \cup \Sigma$ следующим образом:

- (1) $X \triangleleft Y$, если в P есть такое правило $A \rightarrow \alpha X B \beta$, что $B \Rightarrow^+ Y \gamma$;
- (2) $X \equiv Y$, если в P есть правило $A \rightarrow \alpha X Y \beta$;
- (3) отношение \triangleright определяется на $(N \cup \Sigma) \times \Sigma$, так как непосредственно справа от основы в правовыводимой цепочке может быть только терминальный символ; полагаем $X \triangleright a$, если в P есть правило $A \rightarrow \alpha B Y \beta$, $B \Rightarrow^+ \gamma X$ и $Y \Rightarrow^* a\delta$ (заметим, что $Y = a$, если $Y \Rightarrow^* a\delta$).

Анализируя входную цепочку методом предшествования, удобно добавлять к ней левый и правый концевые маркеры. В качестве такого концевого маркера возьмем символ $\$$ и будем считать, что $S \triangleleft X$ для всех X , для которых $S \Rightarrow^+ X\alpha$, и $Y \triangleright \$$ для всех Y , для которых $S \Rightarrow^+ \alpha Y$.

Вычислять отношения предшествования Вирта—Вебера не трудно. Предоставляем читателю самому разработать соответствующий алгоритм или воспользоваться приведенным в разд. 5.3.3 алгоритмом, вычисляющим расширенные отношения предшествования.

Определение. КС-грамматика $G = (N, \Sigma, P, S)$ называется *грамматикой предшествования*, если она приведенная¹⁾, не содержит *e*-правил и для любой пары символов из $N \cup \Sigma$ выполняется не более одного отношения предшествования Вирта—Вебера. Обратимая грамматика предшествования называется *грамматикой простого предшествования*.

Как обычно, язык, порождаемый грамматикой (простого) предшествования, назовем языком (*простого*) предшествования.

Пример 5.34. Пусть G состоит из правил

$$S \rightarrow aSSb | c$$

Отношения предшествования для грамматики G вместе с дополнительными отношениями для концевых маркеров показаны в виде матрицы предшествования на рис. 5.11. Каждая ячейка матрицы содержит отношение предшествования между символом, помечаящим соответствующую строку, и символом, помечаящим столбец. Пустая ячейка интерпретируется как *ошибка*.

Опишем теперь систематический подход к построению отношений предшествования. Отношение \equiv вычисляется легко: просматриваются правые части правил и обнаруживается, что $a \equiv S$, $S \equiv S$ и $S \equiv b$.

Чтобы вычислить отношение \triangleleft , будем искать в правых частях правил пары смежных символов XC . Тогда X находится

¹⁾ Напомним, что КС-грамматика G называется приведенной, если в ней нет выводов вида $A \Rightarrow^+ A$, бесполезных символов и *e*-правил, за исключением, быть может, $S \rightarrow e$, причем в этом случае S не встречается в правых частях правил.

в отношении \lessdot с каждым самым левым символом цепочки, не-
триуально выводимой из C . (Читателю предоставляем разрабо-
тать алгоритм для нахождения всех таких символов.) В нашем
примере будут рассмотрены пары aS и SS . В обоих случаях

S	a	b	c	$\$$
\vdash	\lessdot	\vdash	\lessdot	
\vdash	\lessdot		\lessdot	
b	\gt	\gt	\gt	\gt
c	\gt	\gt	\gt	\gt
$\$$		\lessdot		\lessdot

Рис. 5.11. Отношения предшествования.

роль C играет S , причем из S выводятся цепочки, начинаю-
щиеся символом a или c . Поэтому $X \lessdot Y$, где $X \in \{a, S\}$ и
 $Y \in \{a, c\}$.

Чтобы вычислить \gt , снова ищем в правых частях правил пары смежных символов, на этот раз вида CX . Затем ищем сим-
волы Y , которые могут оказаться на правом конце цепочки,
выводимой из C за один или более шагов, и терминалы d , кото-
рые могут находиться в начале цепочки, выводимой из X за
нуль или более шагов. Если X — терминал, то единственная
возможность: $X = d$. В данном примере парами такого вида
будут SS и Sb . Поэтому $Y \gt d$, где $Y \in \{b, c\}$ и $d \in \{a, c\}$ для
первого правила, $d = b$ — для второго.

Подчеркнем, что отношения \vdash , \lessdot и \gt не обладают свой-
ствами, которые обычно приписываются отношениям $=$, \lessdot и \gt ,
заданным на вещественных числах, целых и т. п. Например,
 \vdash обычно не является отношением эквивалентности, \lessdot и \gt ,
вообще говоря, не транзитивны, но могут быть симметричными
или рефлексивными.

Так как в каждой ячейке матрицы на рис. 5.11 записано не
более одного отношения предшествования, то G — грамматика
предшествования. Кроме того, правые части всех правил в G
различны, так что G — грамматика простого предшествования,
а $L(G)$ — язык простого предшествования.

Рассмотрим правовыводимую цепочку $\$accb\$$ грамматики G ,
ограниченную концевыми маркерами. Имеем $\$ \lessdot a$, $a \lessdot c$ и $c \gt c$.
Основной цепочки $accb$ служит первый символ c , а отношения
предшествования как раз и выделяют эту основу. \square

Всю существенную информацию, содержащуюся в матрице
предшествования $n \times n$, часто можно представить в виде двух

n -мерных векторов. Такие представления матриц предшествова-
ния будут изучаться в разд. 7.1.

Теорема 5.14, сформулированная ниже, показывает, что отно-
шение \lessdot встречается в начале основы правовыводимой цепочки,
отношение \vdash — между смежными символами основы, а отно-
шение \gt — на правом конце основы. Это верно для любой
грамматики без e -правил, но только в грамматиках предшест-
вования любые два смежных символа активного префикса пра-
вовыводимой цепочки связаны не более чем одним отношением
предшествования.

Сначала выведем одно следствие из существования отноше-
ния предшествования для двух данных символов.

Лемма 5.3. Пусть $G = (N, \Sigma, P, S)$ — приведенная КС-грамма-
тика без e -правил.

(1) Если $X \lessdot A$ или $X \vdash A$ и $A \rightarrow Ya$ содержится в P , то
 $X \lessdot Y$.

(2) Если $A \lessdot a$, $A \vdash a$ или $A \gt a$ и $A \rightarrow aY$ содержится
в P , то $Y \gt a$.

Доказательство. Мы докажем (2), а (1) оставим в качес-
тве упражнения. Если $A \lessdot a$, то существует такая правая часть
 $\beta_1 A \beta_2$, что $B \Rightarrow^+ a\gamma$ для некоторой цепочки γ . Так как $A \Rightarrow^+ aY$,
то отсюда сразу получаем $Y \gt a$. Если $A \vdash a$, то существует
правая часть $\beta_1 A \beta_2$. Поскольку $a \Rightarrow^* a$ и $A \Rightarrow^+ aY$, отсюда снова
получаем $Y \gt a$. Если $A \gt a$, то существует правая часть $\beta_1 BX\beta_2$,
где $B \Rightarrow^+ \gamma A$ и $X \Rightarrow^* ab$ для некоторых γ и b . Так как $B \Rightarrow^+ \gamma a Y$,
то опять получаем нужное заключение. \square

Теорема 5.14. Пусть $G = (N, \Sigma, P, S)$ — приведенная КС-грам-
матика без e -правил. Если

$$\begin{aligned} \$\$ &\Rightarrow_r^n X_p X_{p-1} \dots X_{k+1} A a_1 \dots a_q \\ &\Rightarrow_r X_p X_{p-1} \dots X_{k+1} X_k \dots X_1 a_1 \dots a_q \end{aligned}$$

то

- (1) $X_{i+1} \lessdot X_i$ или $X_{i+1} \vdash X_i$ для $p < i < k$,
- (2) $X_{k+1} \lessdot X_k$,
- (3) $X_{i+1} \vdash X_i$ для $k > i \geq 1$,
- (4) $X_1 \gt a_1$.

Доказательство. Доказательство проведем индукцией по n . Для $n=0$ имеем $\$\$ \Rightarrow_r \$ X_k \dots X_1 \$$. По определению отно-
шений предшествования $\$ \lessdot X_k$, $X_{i+1} \vdash X_i$ для $k > i \geq 1$ и
 $X_1 \gt \$$. Заметим, что $X_k \dots X_1$ не может быть пустой цепочкой,
так как по условию в G нет e -правил.

Для доказательства шага индукции предположим, что теорема верна для n . Рассмотрим вывод

$$\begin{aligned} \$S\$ &\Rightarrow_r^n X_p \dots X_{k+1} A a_1 \dots a_q \\ &\Rightarrow_r X_p \dots X_{k+1} X_k \dots X_1 a_1 \dots a_q \\ &\Rightarrow_r X_p \dots X_{j+1} Y_r \dots Y_1 X_{j-i} \dots X_1 a_1 \dots a_q \end{aligned}$$

в котором на последнем шаге X_i заменяется цепочкой $Y_r \dots Y_1$. Поэтому X_{j-1}, \dots, X_1 — терминалы, причем случай $j=1$ не исключается.

По предположению индукции $X_{j+1} \lessdot X_j$ или $X_{j+1} \equiv X_j$. Поэтому $X_{j+1} \lessdot Y_r$ в силу леммы 5.3(1). К тому же X_j находится в одном из трех отношений с символом справа от него (которым может быть a_1). Таким образом, $Y_1 \succ X_{j-1}$ либо $Y_1 \succ a_1$ при $j=1$. Так как $Y_r \dots Y_1$ — правая часть правила, то $Y_r \equiv Y_{r-1} \equiv \dots \equiv Y_1$. Наконец, $X_{i+1} \lessdot X_i$ или $X_{i+1} \equiv X_i$ для $r < i < j$ по предположению индукции. Теорема доказана. \square

Следствие 1. Если G — грамматика предшествования, то утверждение (1) теоремы 5.14 можно усилить, добавив слова „точно одно из отношений \lessdot или \equiv “. Утверждения (1) — (4) можно усилить, добавив слова „и не выполняются никакие другие отношения“.

Доказательство. Непосредственно следует из определения грамматики предшествования. \square

Следствие 2. Каждая грамматика простого предшествования однозначна.

Доказательство. Надо лишь заметить, что для любой отличной от S правовыводимой цепочки β предыдущая правовыводимая цепочка α , для которой $\alpha \Rightarrow \beta$, определяется однозначно. Из следствия 1 вытекает, что основу цепочки β можно однозначно определить, просматривая эту цепочку (ограниченную концевыми маркерами) слева направо, пока впервые не обнаружится отношение \succ , а затем возвращаясь назад до ближайшего \lessdot . Основа лежит между ними. Так как грамматика простого предшествования обратима, то нетерминал, к которому надо свернуть основу, находится однозначно. Таким образом, α определяется по β однозначно. \square

Заметим, что так как мы оперируем с приведенными грамматиками, нетрудно доказать, что этот и последующие алгоритмы трятся на разбор линейное время. Доказательства оставляем в качестве упражнений.

Теперь опишем, как по грамматике простого предшествования строить детерминированный правый анализатор.

Алгоритм 5.12. Построение алгоритма типа „перенос — свертка“ для грамматик простого предшествования.

Вход. Грамматика простого предшествования $G = (N, \Sigma, P, S)$, в которой правила занумерованы числами от 1 до p .

Выход. $\mathcal{A} = (f, g)$, алгоритм разбора типа „перенос — свертка“ для грамматики G .

Метод.

(1) Алгоритм \mathcal{A} использует символ $\$$ в качестве маркера dna магазина и правого концевого маркера входной цепочки.

(2) Функция переноса f зависит только от верхнего символа магазина и самого левого символа необработанной части входной цепочки. Поэтому зададим f только на $(N \cup \Sigma \cup \{\$\}) \times (\Sigma \cup \{\$\})$, за исключением одного случая (правило (в)):

(а) $f(X, a) = \text{перенос}$, если $X \lessdot a$ или $X \equiv a$,

(б) $f(X, a) = \text{свертка}$, если $X \succ a$,

(в) $f(\$, \$) = \text{допуск}^1$,

(г) $f(X, a) = \text{ошибка}$ в остальных случаях.

(Правила реализуются с помощью матрицы предшествования.)

(3) Функция свертки g зависит только от верхней части магазинной цепочки, включающей основу и еще один символ ниже нее. Оставшаяся необработанной часть входной цепочки и g не влияет. Поэтому зададим g только на $(N \cup \Sigma \cup \{\$\})^*$:

(а) $g(X_{k+1} X_k X_{k-1} \dots X_1, e) = i$, если $X_{k+1} \lessdot X_k, X_{k+1} \equiv X_j$ для $k > j \geq 1$ и $A \rightarrow X_k X_{k-1} \dots X_1$ — правило с номером i .

(Заметим, что функция g участвует только тогда, когда $X_1 \succ a$, где a — текущий входной символ.)

(б) $g(\alpha, e) = \text{ошибка}$ в остальных случаях. \square

Пример 5.35. Построим алгоритм разбора типа „перенос — свертка“ $\mathcal{A} = (f, g)$ для грамматики G с правилами

(1) $S \rightarrow aSSb$

(2) $S \rightarrow c$

Отношения предшествования для грамматики G приведены на рис. 5.11. Функцию переноса f можно вычислить с помощью матрицы предшествования. Функция свертки g определяется так:

(1) $g(XaSSb) = 1$, если $X \in \{S, a, \$\}$,

(2) $g(Xc) = 2$, если $X \in \{S, a, \$\}$.

(3) $g(\alpha) = \text{ошибка}$ в остальных случаях.

¹) Заметим, что это правило имеет приоритет над правилами (2a) и (2b), когда $X = S$ и $a = \$$.

Для входной цепочки $accb$ алгоритм \mathcal{A} сделает такую последовательность шагов:

$$\begin{aligned} (\$, accb\$, e) &\vdash^s (\$a, ccb\$, e) \\ &\vdash^s (\$ac, cb\$, e) \\ &\vdash^r (\$aS, cb\$, 2) \\ &\vdash^s (\$aSc, b\$, 2) \\ &\vdash^r (\$aSS, b\$, 22) \\ &\vdash^s (\$aSSb, \$, 22) \\ &\vdash^r (\$S, \$, 221) \end{aligned}$$

В конфигурации $(\$ac, cb\$, e)$, например, будет $f(c, c) = \text{свертка}$ и $g(ac, e) = 2$. Поэтому

$$(\$ac, cb\$, e) \vdash (\$aS, cb\$, 2)$$

Посмотрим, как ведет себя \mathcal{A} на цепочке acb , не принадлежащей языку $L(G)$. Для этой цепочки алгоритм \mathcal{A} сделает последовательность шагов

$$\begin{aligned} (\$, acb\$, e) &\vdash^s (\$a, cb\$, e) \\ &\vdash^s (\$ac, b\$, e) \\ &\vdash^r (\$aS, b\$, 2) \\ &\vdash^s (\$bSb, \$, 2) \\ &\vdash \text{ошибка} \end{aligned}$$

В конфигурации $(\$aSb, \$, 2)$ будет $f(b, \$) = \text{свертка}$. Так как $\$ \ll a$ и $a \leq S \leq b$, то свертку можно сделать только, если aSb — правая часть какого-нибудь правила. Однако такого правила нет, и поэтому $g(aSb, e) = \text{ошибка}$.

На практике мы могли бы завести список „ошибочных правил“, и всякий раз, когда с помощью функции g обнаруживается ошибка, можно было бы справиться в этом списке, нельзя ли сделать свертку с помощью ошибочного правила. Другие приемы исправления ошибок, ориентированные на разбор методом предшествования, указаны в замечаниях по литературе в конце этого раздела. \square

Теорема 5.15. Алгоритм 5.12 строит корректный алгоритм разбора типа „перенос—свертка“ для грамматики простого предшествования.

Доказательство. Теорема непосредственно следует из теоремы 5.14, свойства обратимости и конструкции алгоритма 5.12. Детали доказательства оставляем в качестве упражнения. \square

Интересно изучить классы языков, порождаемых грамматиками предшествования и грамматиками простого предшествования. Оказывается, каждый КС-язык, не содержащий пустой цепочки e , определяется грамматикой предшествования, но не каждый такой КС-язык определяется грамматикой простого предшествования. Кроме того, для каждого КС-языка, не содержащего e , можно найти обратимую КС-грамматику без e -правил. Таким образом, если от грамматик требуется, чтобы они были обратимыми и грамматиками предшествования, то это уменьшает их порождающую способность. Каждая грамматика простого предшествования является LR(1)-грамматикой, но LR(1)-язык $\{a0^i1^i \mid i \geq 1\} \cup \{b0^i1^{2i} \mid i \geq 1\}$ не определяется никакой грамматикой простого предшествования (это будет показано в разд. 8.3).

5.3.3. Грамматики расширенного предшествования

Отношения предшествования Вирта—Бебера, определенные для пар символов, можно расширить на пары цепочек. Определим расширенные отношения предшествования — между цепочками из m символов и цепочками из n символов. Наше определение ориентировано на некоторый подразумеваемый алгоритм разбора типа „перенос—свертка“.

Для понимания мотивов введения расширенного предшествования вспомним две роли, которые играют отношения предшествования в алгоритме типа „перенос—свертка“.

(1) Пусть αX — это m верхних символов магазина (причем X — самый верхний) и aw — следующие n входных символов. Если $\alpha X \ll aw$ или $\alpha X \leq aw$, то a надо перенести в магазин. Если $\alpha X \gg aw$, то надо сделать свертку.

(2) Допустим, что $X_p \dots X_2 X_1$ — цепочка в магазине и $a_1 \dots a_q$ — необработанная часть входной цепочки в тот момент, когда вызывается процедура свертки (т. е. $X_m \dots X_1 \gg a_1 \dots a_n$). Если основой является $X_k \dots X_1$, то нам хочется, чтобы было

$$X_{m+j} X_{m+j-1} \dots X_{j+1} \ll X_j X_{j-i} \dots X_1 a_1 \dots a_{n-j}$$

для $k > j \geq 1$ и

$$X_{m+k} \dots X_{k+1} \ll X_k \dots X_1 a_1 \dots a_{n-k}$$

Таким образом, процедура разбора для обратимой грамматики расширенного предшествования аналогична процедуре, соответствующей грамматике простого предшествования Вирта—Бебера, за исключением того, что отношение предшествования между символами X и Y определяется цепочками αX и βY , где

¹⁾ Предполагаем, что $X_r X_{r-1} \dots X_1 a_1 \dots a_{n-r} = X_r X_{r-1} \dots X_{r-n+1}$, если $r \geq n$.

α состоит из $m - 1$ символов, расположенных слева от X , а β — из $n - 1$ символов, расположенных справа от Y .

Символы переносятся в магазин до тех пор, пока между цепочкой наверху магазина и оставшейся необработанной частью входной цепочки не встретится отношение \gg . Тогда возвращаемся по магазину назад, проходя отношения \equiv , пока впервые не встретится \ll . Между \ll и \gg лежит осиова.

Эти замечания мотивируют следующее определение.

Определение. Пусть $G = (N, \Sigma, P, S)$ — приведенная КС-грамматика без e -правил. Определим отношения (m, n) -предшествования \ll , \equiv и \gg на множестве $(N \cup \Sigma \cup \{\$\})^m \times (N \cup \Sigma \cup \{\$\})^n$. Пусть

$$\begin{aligned} \$^m S \$^n &\Rightarrow^* X_p X_{p-1} \dots X_{k+1} A a_1 \dots a_q \\ &\Rightarrow, X_p X_{p-1} \dots X_{k+1} X_k \dots X_1 a_1 \dots a_q \end{aligned}$$

— произвольный правый вывод. Тогда

(1) $\alpha \ll \beta$, если α состоит из последних m символов цепочки $X_p X_{p-1} \dots X_{k+1}$ и либо

(a) β состоит из первых n символов цепочки $X_k \dots X_1 a_1 \dots a_q$, либо

(b) X_k — терминал и $\beta \in \text{FIRST}_n(X_k \dots X_1 a_1 \dots a_q)$;

(2) $\alpha \equiv \beta$ для всех таких $k > j \geq 1$, что α состоит из последних m символов цепочки $X_p X_{p-1} \dots X_{j+1}$ и либо

(a) β состоит из первых n символов цепочки $X_j X_{j-1} \dots X_1 a_1 \dots a_q$, либо

(b) X_j — терминал и $\beta \in \text{FIRST}_n(X_j X_{j-1} \dots X_1 a_1 \dots a_q)$;

(3) $X_m X_{m-1} \dots X_1 \gg a_1 \dots a_n$.

Будем говорить, что G — грамматика (m, n) -предшествования, если G — приведенная КС-грамматика без e -правил и отношения \ll , \equiv и \gg попарно не пересекаются. Из леммы 5.3 с очевидностью следует, что G — грамматика предшествования тогда и только тогда, когда она — грамматика $(1, 1)$ -предшествования. Детали, связанные с концевыми маркерами, легко восстанавливаются. При $n = 1$ условия (1б) и (2б) не дают ничего нового.

Заметим также, что если пересечение отношений \ll и \equiv непусто только за счет условий (1б) и (2б), все равно для такой грамматики расширенного предшествования найдется алгоритм разбора типа „перенос — свертка“. Можно было бы дать более сложное, но менее ограничительное определение, но мы предпочитаем оставить читателю в виде упражнения разработку такого класса грамматик.

Изложим теперь алгоритм, вычисляющий отношения расширенного предшествования. Его, очевидно, можно применять и для вычисления отношений предшествования Вирта — Вебера.

Алгоритм 5.13. Построение отношений (m, n) -предшествования.

Вход. Приведенная КС-грамматика $G = (N, \Sigma, P, S)$ без e -правил.

Выход. Отношения (m, n) -предшествования \ll , \equiv и \gg для грамматики G .

Метод. Начнем с построения множества \mathcal{S} всех подцепочек длины $m+n$, которые могут встретиться в такой цепочке $\alpha\beta\gamma$, что $\$^m S \$^n \Rightarrow^* \alpha A w \Rightarrow, \alpha\beta\gamma$ и $w = \text{FIRST}_n(\gamma)$. Это осуществляется так:

(1) Положим $\mathcal{S} = \{\$^m S \$^{n-1}, \$^{m-1} S \$^n\}$. Эти две цепочки в \mathcal{S} считаются „нерассмотренными“.

(2) Если δ — нерассмотренная цепочка из \mathcal{S} , „рассмотрим“ ее, выполнив следующие две операции.

(a) Если δ не имеет вида αAx , где $|x| \leq n$, то ничего не делать.

(b) Если $\delta = \alpha Ax$, $|x| \leq n$ и $A \in N$, то добавить к \mathcal{S} , если их еще нет там, цепочки γ , для которых в P найдется такое правило $A \rightarrow \beta$, что γ — подцепочка длины $m+n$ цепочки $\alpha\beta\gamma$. Заметим, что так как G — приведенная грамматика, то $|\alpha\beta\gamma| \geq m+n$. Добавленные к \mathcal{S} цепочки считаются нерассмотренными.

(3) Повторять шаг (2), пока в \mathcal{S} не останется нерассмотренных цепочек.

По множеству \mathcal{S} построим отношения \ll , \equiv и \gg :

(4) Для каждой цепочки $\alpha\beta\gamma$ из \mathcal{S} , для которой $|\alpha| = m$, и для каждого правила $A \rightarrow \beta$ из P положим $\alpha \ll \delta$, где δ — первые n символов цепочки $\beta\gamma$ либо β начинается терминалом и $\delta \in \text{FIRST}_n(\beta\gamma)$.

(5) Для каждой цепочки $\alpha\beta\gamma$ из \mathcal{S} , для которой $|\alpha| = m$, и для каждого правила $A \rightarrow \beta_1 XY\beta_2$ из P положим $\delta_1 \equiv \delta_2$, где δ_1 — последние m символов цепочки $\alpha\beta_1 X$, а δ_2 — первые n символов цепочки $Y\beta_2\gamma$, либо Y — терминал, а $\delta_2 = Yw$ для некоторой цепочки $w \in \text{FIRST}_{n-1}(\beta_2\gamma)$.

(6) Для каждой цепочки αAw из \mathcal{S} , для которой $|w| = n$, и для каждого правила $A \rightarrow \beta$ из P положим $\delta \gg w$, где δ — последние m символов цепочки $\alpha\beta$. \square

Пример 5.36. Рассмотрим грамматику G с правилами

$$S \rightarrow 0S11|011$$

Отношения $(1, 1)$ -предшествования для нее приведены на рис. 5.12. Так как $1 \gg 1$ и $1 \equiv 1$, то G не является грамматикой $(1, 1)$ -предшествования.

С помощью алгоритма 5.13 вычислим для G отношения $(2, 1)$ -предшествования. Начнем с вычисления \mathcal{S} . Вначале

$\mathcal{S} = \{\$\$, \$\$S\}$. Рассмотрим $\$S$, добавляя к \mathcal{S} цепочки $\$0S$, $0S1$, $S11$, $11\$$ (это все подцепочки длины 3 цепочки $\$0S11\$$) и $\$01$, 011 (подцепочки длины 3 цепочки $\$011\$$). Рассмотрение цепочки $\$\S приводит к добавлению $\$\0 , рассмотрение $\$0S$ добавляет $\$00$, $00S$ и 001 , рассмотрение $0S1$ добавляет 111 , и,

s	0	1	$\$$
s		\doteq	
0	\doteq	\lessdot	\doteq
1		$\doteq, >$	$>$
$\$$		\lessdot	

Рис. 5.12. Отношения $(1,1)$ -предшествования.

наконец, рассмотрение $00S$ добавляет 000 . Вот все элементы множества \mathcal{S} .

Чтобы построить \lessdot , рассмотрим цепочки из \mathcal{S} с S на правом конце. Получим $\$\$ \lessdot 0$, $\$0 \lessdot 0$ и $00 \lessdot 0$. Для построения \doteq снова рассмотрим такие цепочки и найдем

$$\$0 \doteq S, 0S \doteq 1, S1 \doteq 1, \$0 \doteq 1, 01 \doteq 1, 00 \doteq S \text{ и } 00 \doteq 1.$$

Чтобы построить \gg , рассмотрим цепочки из \mathcal{S} , у которых S в середине. Из $\$S\$$ получится $11 \gg \$$, а из $0S1$ получится $11 \gg 1$.

Отношения $(2,1)$ -предшествования для грамматики G приведены на рис. 5.13. Цепочки длины 2, не принадлежащие области определения отношений \doteq , \lessdot и \gg , здесь опущены.

s	0	1	$\$$
$\$\$$		\lessdot	
$\$0$	\doteq	\lessdot	\doteq
$0S$		\doteq	
00	\doteq	\lessdot	
01		\doteq	
$S1$		\doteq	
11		\gg	\gg

Рис. 5.13. Отношения $(2,1)$ -предшествования.

Так как конфликтов между отношениями $(2,1)$ -предшествования нет, то G — грамматика $(2,1)$ -предшествования. \square

Теорема 5.16. Алгоритм 5.13 правильно вычисляет отношения \lessdot , \doteq и \gg .

Доказательство. Сначала покажем, что правильно определяется множество \mathcal{S} , т. е. что $\gamma \in \mathcal{S}$ тогда и только тогда, когда $|\gamma| = m+n$ и γ — подцепочка цепочки $\alpha\beta u$, где $\$^mS^n \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha\beta w$ и $u = \text{FIRST}_n(w)$.

Необходимость. Доказательство проводится индукцией по порядку, в котором цепочки добавляются к множеству \mathcal{S} . Базис, когда в \mathcal{S} содержатся только первые два элемента, получается сразу. Для доказательства шага индукции допустим, что γ добавляется к \mathcal{S} потому, что $\alpha Ax \in \mathcal{S}$ и $A \rightarrow \beta \in P$, т. е. γ — подцепочка цепочки $\alpha\beta x$. Так как αAx содержится в \mathcal{S} , то по предположению индукции существует вывод $\$^mS^n \Rightarrow_r^* \alpha' A' w \Rightarrow_r \alpha'\beta' uv$, где $u = \text{FIRST}_n(w)$ и $\alpha'\beta'u$ можно записать в виде $\delta_1 \alpha A x \delta_2$ для некоторых δ_1 и δ_2 из $(N \cup \{\$\})^*$. Так как G — приведенная грамматика, то найдется такая цепочка $y \in (\Sigma \cup \{\$\})^*$, что $\delta_2 \Rightarrow_r^* y$. Таким образом, $\$^mS^n \Rightarrow_r^* \delta_1 \alpha A x y v \Rightarrow_r \delta_1 \alpha \beta x y v$. Так как γ — подцепочка длины $m+n$ цепочки $\alpha\beta x$, то она, конечно, является подцепочкой цепочки $\alpha\beta z$, где $z = \text{FIRST}_n(xyv)$.

Достаточность. Индукцией по k можно показать, что если $\$^mS^n \Rightarrow_r^k \alpha A w \Rightarrow_r \alpha\beta w$, то каждая подцепочка длины $m+n$ цепочки $\alpha\beta u$ содержится в \mathcal{S} , где $u = \text{FIRST}_n(w)$.

Из определения отношений \lessdot , \doteq и \gg непосредственно следует, что они правильно вычисляются на шагах (4)–(6). \square

Докажем теорему, которая лежит в основе анализатора типа „перенос—свертка“ для обратимых грамматик (m, n) -предшествования, аналогичного алгоритму 5.12.

Теорема 5.17. Пусть $G = (N, \Sigma, P, S)$ — произвольная приведенная КС-грамматика, m и n — целые числа и

$$(5.3.1) \quad \begin{aligned} \$^mS^n &\Rightarrow_r^* X_p X_{p-1} \dots X_{k+1} A a_1 \dots a_q \\ &\Rightarrow_r X_p X_{p-1} \dots X_{k+1} X_k \dots X_1 a_1 \dots a_q \end{aligned}$$

(1) Пусть $p-m \geq j > k$, α — последние m символов цепочки $X_p X_{p-1} \dots X_{j+1}$ и β — первые n символов цепочки $X_j X_{j-1} \dots X_1 a_1 \dots a_q$. Тогда если $\beta \in (\Sigma \cup \{\$\})^*$, то либо $\alpha \lessdot \beta$, либо $\alpha \doteq \beta$.

(2) $X_{m+k} X_{m+k-1} \dots X_{k+1} \lessdot \beta$, где β состоит из первых n символов цепочки $X_k \dots X_1 a_1 \dots a_q$.

(3) Пусть $k > j \geq 1$, α —последние m символов цепочки $X_p X_{p-1} \dots X_{j+1}$ и β —первые n символов цепочки $X_j X_{j-1} \dots X_1 a_1 \dots a_p$. Тогда $\alpha \sqsupseteq \beta$.

$$(4) \quad X_m X_{m-1} \dots X_1 \gg a_1 \dots a_n.$$

Доказательство. Утверждения (2)–(4) непосредственно следуют из определений. Чтобы доказать (1), заметим, что так как $j > k$, β состоит не только из символов \$. Поэтому вывод (5.3.1) можно записать в виде

$$\begin{aligned} (5.3.2) \quad \$^m S \$^n &\Rightarrow_r^i \gamma B w \\ &\Rightarrow_r \gamma \delta_1 \delta_2 w \\ &\Rightarrow_r^* X_p X_{p-1} \dots X_{k+1} A a_1 \dots a_q \\ &\Rightarrow_r X_p X_{p-1} \dots X_{k+1} X_k \dots X_1 a_1 \dots a_q \end{aligned}$$

где i —наибольшее из чисел, для которых $\delta_2 \neq e$ в правиле $B \rightarrow \delta_1 \delta_2$, из B выводятся X_{j+1} и X_j и $\gamma \delta_1 = X_p X_{p-1} \dots X_{j+1}$.

Если первый символ цепочки δ_2 терминальный, скажем $\delta_2 = a\delta_3$, то по правилу (2б) определения отношения \sqsubseteq имеем $X_{j+m} X_{j+m-1} \dots X_{j+1} \sqsubseteq \beta$, где $\beta = ax$ и $x \in \text{FIRST}_{n-1}(\delta_3 w)$.

Если первый символ цепочки δ_2 нетерминальный, положим $\delta_2 = C\delta_3$. Так как по предположению X_j —терминал, то из C после нескольких шагов вывода (5.3.2) должна получиться цепочка $D\epsilon$ для некоторых $D \in N$ и $\epsilon \in (N \cup \Sigma)^*$. Далее, D заменяется цепочкой $X_j\theta$, и по правилу (2б) отсюда следует требующееся отношение \lhd . \square

Следствие. Если в теореме 5.17 G —грамматика (m, n) -предшествования, то теорему можно усилить, добавив к каждому из утверждений (1)–(4) слова о том, что между соответствующими цепочками не выполняются никакие другие отношения. \square

Алгоритм разбора типа „перенос-свертка“ для обратимых грамматик расширенного предшествования полностью аналогичен алгоритму 5.12 для грамматик простого предшествования, так что мы только вкратце скажем о нем. Первые n необработанных входных символов можно хранить наверху магазина. Если такими символами являются $a_1 \dots a_n$ и непосредственно под ними в магазине расположена цепочка $X_m \dots X_1$, причем $X_m \dots X_1 \sqsubseteq a_1 \dots a_n$ или $X_m \dots X_1 \lhd a_1 \dots a_n$, то надо сделать перенос. Если же $X_m \dots X_1 \gg a_1 \dots a_n$, то делается свертка. Утверждение (1) теоремы 5.17 гарантирует, что один из первых двух случаев произойдет всегда, когда основа лежит вправо от X_1 . В силу утверждения (4) этой теоремы правый конец основы достигается тогда и только тогда, когда происходит третий случай.

Чтобы сделать свертку, надо так же, как в алгоритме 5.12, вернуться назад, проходя через отношения \sqsubseteq , в поисках отношения \lhd . Из утверждений (2) и (3) теоремы 5.17 вытекает, что основа будет выделена правильно.

5.3.4. Грамматики слабого предшествования

Многие естественно встречающиеся грамматики не являются грамматиками простого предшествования, и попытки найти для данного языка грамматику простого предшествования часто приводят к довольно неуклюжим грамматикам. Можно расширить класс грамматик, анализируемых методом предшествования, ослабив ограничение, что отношения \lhd и \sqsubseteq не должны пересекаться.

Отношение \gg по-прежнему будем использовать для локализации правого конца основы. Тогда для локализации левого конца основы можно использовать правые части правил, подыскив среди них такую, которая совпадает с символами, стоящими непосредственно слева от правого конца основы. Это не намного более трудная работа, чем разбор методом простого предшествования. В ходе разбора для грамматики простого предшествования после выделения основы все равно требуется определить, какое именно правило применить для свертки, так что эти символы так или иначе придется рассматривать.

Для того чтобы эта схема разбора работала, надо уметь определить, какое правило применить в том случае, когда правая часть одногого правила является суффиксом правой части другого правила. Например, пусть $\alpha\beta\gamma\omega$ —правовыводимая цепочка, в которой правый конец основы лежит между γ и ω . Если $A \rightarrow \gamma$ и $B \rightarrow \beta\gamma$ —два разных правила, то не ясно, какое из них нужно применить для свертки.

Мы ограничимся применением самого длинного из применимых правил. Класс грамматик, для которых такое решение оказывается правильным, образуют грамматики слабого предшествования.

Определение. Пусть $G = (N, \Sigma, P, S)$ —приведенная КС-грамматика без e -правил. Назовем G *грамматикой слабого предшествования*, если

(1) отношение \gg не пересекается с объединением отношений \lhd и \sqsubseteq ,

(2) для $A \rightarrow \alpha X \beta$ и $B \rightarrow \beta$ из P , где $X \in N \cup \Sigma$, не выполняется ни $X \lhd B$, ни $X \sqsubseteq B$.

Пример 5.37. Примером грамматики слабого предшествования может служить грамматика G^1) с правилами

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

Матрица предшествования для G приведена на рис. 5.14.

Заметим, что конфликты возникают только между отношениями \ll и \equiv , так что условие (1) определения грамматики слабого предшествования удовлетворяется. Чтобы убедиться, что

	E	T	F	α	()	$+$	$*$	$$$
E							\doteq	\doteq	
T							$>$	$>$	\doteq
F							$>$	$>$	$>$
α							$>$	$>$	$>$
)							$>$	$>$	$>$
(\ll, \doteq	$<$	$<$	\ll	$<$	$<$			
+	\ll, \doteq	$<$	\ll	\ll					
*		\doteq	$<$	$<$					
\$	$<$	$<$	$<$	$<$	$<$	$<$			

Рис. 5.14. Матрица предшествования.

условие (2) тоже не нарушается, рассмотрим сначала три правила $E \rightarrow E + T$, $E \rightarrow + T$ и $E \rightarrow T$ ²⁾. Из матрицы предшествования видно, что между E и E , а также между $+$ и E (т. е. когда $+$ рассматривается как левый аргумент отношения) нет никакого отношения предшествования. Таким образом, эти три правила не нарушают условия (2). Остальные правила, в которых одна правая часть служит суффиксом другой,— это $T \rightarrow T * F$ и $T \rightarrow F$. Так как между $*$ и T нет отношения предшествования,

¹⁾ Очевидно, что грамматика G тесно связана с нашей любимой грамматикой G_0 . В самом деле, язык $L(G)$ — это $L(G_0)$ с лишними унарными знаками $+$, как, например, в цепочке $+a * (+a + a)$. Грамматика G_0 — тоже обратимая грамматика слабого предшествования, но не грамматика простого предшествования.

²⁾ Случайно эти три правила имеют одну и ту же левую часть.

то и здесь условие (2) не нарушено. Итак, G — грамматика слабого предшествования.

Хотя G не является грамматикой простого предшествования, она порождает язык простого предшествования. Позже мы увидим, что это верно всегда, т. е. каждая обратимая грамматика слабого предшествования порождает язык простого предшествования. \square

Убедимся теперь в том, что в правовыводимой цепочки грамматики слабого предшествования основой всегда будет самая длинная из правых частей применимых правил.

Лемма 5.4. Пусть $G = (N, \Sigma, P, S)$ — грамматика слабого предшествования и P содержит правило $B \rightarrow \beta$. Пусть $\$S\$ \Rightarrow^* \gamma Cw \Rightarrow^* \delta X\beta w$. Если существует правило $A \rightarrow \alpha X\beta$, то последним в этом выводе применялось не правило $B \rightarrow \beta$.

Доказательство. Допустим, что, напротив, $C = B$ и $\gamma = \delta X$. Применив теорему 5.14 к выводу $S \Rightarrow^* \gamma Cw$, получим $X \ll B$ или $X \equiv B$. Это следует из того, что основа цепочки γCw оканчивается где-то правее символа C , и, значит, C — один из символов X ; теоремы 5.14. Но тогда сразу нарушается условие слабого предшествования. \square

Лемма 5.5. Пусть G — такая же грамматика, как в лемме 5.4, и пусть она к тому же обратимая. Если правила вида $A \rightarrow \alpha X\beta$ нет, то в выводе $\$S\$ \Rightarrow^* \gamma Cw \Rightarrow^* \delta X\beta w$ должно быть $C = B$ и $\gamma = \delta X$ (т. е. последним применяется правило $B \rightarrow \beta$).

Доказательство. Очевидно, что на последнем шаге заменяется символ C . Левый конец основы цепочки $\delta X\beta w$ не может быть левее символа X , так как нет правила $A \rightarrow \alpha X\beta$. Если основа оканчивается где-то правее первого символа цепочки β , то нарушается лемма 5.4, в которой $B \rightarrow \beta$ играет роль правила $A \rightarrow \alpha X\beta$. Поэтому основа— цепочка β , и нужный результат следует из обратимости грамматики G . \square

Итак, сущность алгоритма разбора для обратимых грамматик слабого предшествования заключается в следующем. Мы просматриваем правовыводимую цепочку (ограниченную концевыми маркерами) слева направо до тех пор, пока впервые не встретим отношение \gg . Это отношение указывает правый конец основы. Затем по одному рассматриваем символы слева от \gg . Допустим, что есть правило $B \rightarrow \beta$ и слева от \gg оказывается цепочка $X\beta$. Если правила вида $A \rightarrow \alpha X\beta$ нет, то по лемме 5.5 β — основа. Если такое правило есть, то по лемме 5.4 правило $B \rightarrow \beta$ неприменимо. Следовательно, решение о том, свертывать ли β , можно принять, рассмотрев только один символ слева от β .

Таким образом, для каждой обратимой грамматики слабого предшествования можно построить алгоритм разбора типа „перенос—свертка“.

Алгоритм 5.14. Построение алгоритма разбора типа „перенос—свертка“ для грамматик слабого предшествования.

Вход. Обратимая грамматика слабого предшествования $G = (N, \Sigma, P, S)$, правила которой занумерованы числами от 1 до p .

Выход. $\mathcal{A} = (f, g)$, алгоритм разбора типа „перенос—свертка“ для грамматики G .

Метод. Построение аналогочно тому, которое было в алгоритме 5.12. Функция переноса f определяется прямо по отношениям предшествования:

- (1) $f(X, a) = \text{перенос}$, если $X \lessdot a$ или $X \equiv a$,
- (2) $f(X, a) = \text{свертка}$, если $X \gg a$,
- (3) $f(\$, \$) = \text{допуск}$,
- (3) $f(X, a) = \text{ошибка}$ в остальных случаях.

Функция свертки g определяется так, чтобы при свертке применялось самое длинное из применимых правил:

- (4) $g(X\beta) = i$, если $B \rightarrow \beta$ — правило из P с номером i и в P нет правила вида $A \rightarrow \alpha X\beta$,
- (5) $g(\alpha) = \text{ошибка}$ в остальных случаях. \square

Теорема 5.18. Алгоритм 5.14 строит корректный алгоритм разбора типа „перенос—свертка“ для грамматики G .

Доказательство. Теорема непосредственно следует из лемм 5.4 и 5.5, определения обратимой грамматики слабого предшествования и конструкции самого алгоритма \mathcal{A} . \square

С помощью некоторых преобразований можно устранить из грамматики имеющиеся в ней конфликты между отношениями предшествования. Приведем здесь несколько полезных преобразований такого рода, которые часто позволяют переделать грамматику, не обладающую нужными свойствами предшествования, в эквивалентную грамматику $(1,1)$ -предшествования или слабого предшествования.

Допустим, что в грамматике есть конфликт вида $X \equiv Y$ и $X \gg Y$. Так как $X \equiv Y$, то в правой части одного или нескольких правил встречается подцепочка XY . Если в таком правиле заменить X новым нетерминалом A , то $X \equiv Y$ исчезнет и тем самым конфликт будет устранен. Чтобы сохранить эквивалентность, добавим к грамматике правило $A \rightarrow X$. Если X не является правой частью другого правила, то сохранится и обратимость грамматики.

Пример 5.38. Рассмотрим грамматику G с правилами

$$S \rightarrow 0S11|011$$

В примере 5.36 мы видели, что G не грамматика простого предшествования, потому что $1 \equiv 1$ и $1 \gg 1$. Однако, если в каждое правило вместо первой единицы поставить новый нетерминал A

S	A	0	1	$\$$
\vdash	\doteq	\lessdot		
			\doteq	
\doteq	\doteq	\lessdot	\lessdot	
			\gg	\gg
		\lessdot		

Рис. 5.15. Отношения предшествования для грамматики G' .

и добавить правило $A \rightarrow 1$, то получится грамматика простого предшествования G' с правилами

$$\begin{aligned} S &\rightarrow 0SA1|0A1 \\ A &\rightarrow 1 \end{aligned}$$

Отношения предшествования для этой грамматики приведены на рис. 5.15. \square

Аналогичными преобразованиями можно устраниТЬ конфликты вида $X \lessdot Y$, $X \gg Y$ (а также вида $X \lessdot Y$, $X \equiv Y$, если желательно простое предшествование).

В тех случаях, когда эти преобразования нарушают обратимость, можно попытаться устраниТЬ конфликт, удаляя правила, как в лемме 2.14.

Пример 5.39. Рассмотрим грамматику G с правилами

$$\begin{aligned} E &\rightarrow E + T | T \\ T &\rightarrow T * F | F \\ F &\rightarrow a | (E) | a(L) \\ L &\rightarrow L, E | E \end{aligned}$$

В этой грамматике из L выводятся списки выражений, а переменные могут иметь индексы, представляющие собой произвольные последовательности выражений.

G не является грамматикой слабого предшествования, так как $E \equiv$ и $E \gg$. Можно было бы устраниТЬ этот конфликт, заменив E в правиле $F \rightarrow (E)$ на E' и добавив правило $E' \rightarrow E$.

Но тогда оказалось бы два правила с правой частью E . Если же устраниТЬ из G правило $F \rightarrow a(L)$, сделав подстановку вместо L , как в лемме 2.14, то получится эквивалентная грамматика G с правилами

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow a \mid (E) \mid a(L, E) \mid a(E) \\ L &\rightarrow L, E \mid E \end{aligned}$$

Так как L больше не встречается слева от $)$, то в этой грамматике уже нет $E \triangleright$). Легко проверить, что G' — грамматика слабого предшествования. \square

Небольшое обобщение этих приемов позволит преобразовать каждую обратимую грамматику слабого предшествования в эквивалентную грамматику простого предшествования. Следовательно, по своей порождающей способности обратимые грамматики слабого предшествования не превосходят грамматики простого предшествования, хотя, как мы видели на примере 5.37, среди них есть такие, которые не являются грамматиками простого предшествования.

Теорема 5.19. Язык определяется обратимой грамматикой слабого предшествования тогда и только тогда, когда он является языком простого предшествования.

Доказательство. Достаточность. Пусть $G = (N, \Sigma, P, S)$ — грамматика простого предшествования. Очевидно, что условие (1) определения грамматики слабого предшествования удовлетворяется. Допустим, что условие (2) нарушено, т. е. в P есть такие правила $A \rightarrow \alpha X Y \beta$ и $B \rightarrow Y \beta$, что либо $X \triangleleft B$, либо $X \underline{\equiv} B$. Тогда по лемме 5.3 $X \triangleleft Y$. Но $X \underline{\equiv} Y$ из-за правила $A \rightarrow \alpha X Y \beta$. Такая ситуация невозможна, поскольку G — грамматика предшествования.

Необходимость. Пусть $G = (N, \Sigma, P, S)$ — обратимая грамматика слабого предшествования. Построим грамматику простого предшествования $G' = (N', \Sigma, P', S)$, для которой $L(G') = L(G)$.

(1) Пусть N' — это N плюс новые символы вида $[\alpha]$, где цепочка $\alpha \neq e$ такова, что в P есть правило вида $A \rightarrow \beta \alpha$.

(2) Пусть P' состоит из правил

- (а) $[X] \rightarrow X$ для каждого $[X] \in N'$, для которого $X \in N \cup \Sigma$,
- (б) $[X\alpha] \rightarrow X[\alpha]$ для каждого $[X\alpha] \in N'$, для которого $X \in N \cup \Sigma$ и $\alpha \neq e$,
- (в) $A \rightarrow [\alpha]$ для каждого $A \rightarrow \alpha$ из P .

Покажем, что отношения \triangleleft , $\underline{\equiv}$ и \triangleright для грамматики G' попарно не пересекаются. Концевой маркер не может вызвать конфликта, поэтому рассмотрим X и Y из $N' \cup \Sigma$. Заметим, что

(1) если $X \triangleleft Y$, то $X \in N \cup \Sigma$;

(2) если $X \underline{\equiv} Y$, то $X \in N \cup \Sigma$ и $Y \in N' - N$, так как только пункт (2б) построения грамматики G' дает правила, у которых правая часть содержит больше одного символа;

(3) если $X \triangleright Y$, то $X \in N' \cup \Sigma$ и $Y \in \Sigma$.

Докажем, что пересечение каждой пары отношений пусто.

Первая пара: $\underline{\equiv} \cap \triangleright = \emptyset$. Если $X \underline{\equiv} Y$, то $Y \in N' - N$. Если $X \triangleright Y$, то $Y \in \Sigma$. Ясно, что $\underline{\equiv} \cap \triangleright = \emptyset$.

Вторая пара: $\triangleleft \cap \triangleright = \emptyset$. Допустим, что $X \triangleleft Y$ и $X \triangleright Y$. Тогда $X \in N \cup \Sigma$ и $Y \in \Sigma$. Так как $X \triangleleft Y$ в грамматике G' , то в P' есть правило вида $[X\alpha_1] \rightarrow X[\alpha_1]$, причем $[\alpha_1] \Rightarrow_G^+ Y\alpha_2$ для некоторой цепочки $\alpha_2 \in (N \cup \Sigma)^*$. Но цепочка $X\alpha_1$ должна быть суффиксом некоторого правила $A \rightarrow \alpha_s X\alpha_1$ из P . Тогда $\alpha_1 \Rightarrow_G^+ Y\alpha_2$ для некоторой цепочки $\alpha'_2 \in (N \cup \Sigma)^*$. Таким образом, для G имеем $X \underline{\equiv} Y$ или $X \triangleleft Y$.

Рассмотрим теперь $X \triangleright Y$ для грамматики G' . В P' должно быть такое правило $[B\beta_1] \rightarrow B[\beta_1]$, что $B \Rightarrow_G^+ \beta_2 X$ и $[\beta_1] \Rightarrow_G^+ Y\beta_3$ для некоторых $\beta_2 \in (N \cup \Sigma)^*$ и $\beta_3 \in (N' \cup \Sigma)^*$. В грамматике G цепочка $B\beta_1$ является суффиксом некоторого правила $C \rightarrow \gamma B\beta_1$ из P . Кроме того, $B \Rightarrow_G^+ \beta_2 X$ и $\beta_1 \Rightarrow_G^+ Y\beta_3$ для некоторой цепочки $\beta_3 \in (N \cup \Sigma)^*$. Таким образом, $X \triangleright Y$ для грамматики G .

Мы показали, что если $X \triangleleft Y$ и $X \triangleright Y$ для G' , то для G либо $X \triangleleft Y$ и $X \triangleright Y$, либо $X \underline{\equiv} Y$ и $X \triangleright Y$. В любом случае получается противоречие с предположением о том, что G — грамматика слабого предшествования. Таким образом, $\triangleleft \cap \triangleright = \emptyset$ для грамматики G' .

Третья пара: $\triangleleft \cap \underline{\equiv} = \emptyset$. Предположим, что $X \triangleleft [Y\alpha]$ и $X \underline{\equiv} [Y\alpha]$ для некоторых $X \in N \cup \Sigma$ и $[Y\alpha] \in N' - N$. Тогда в P' есть такие правила $[X\beta] \rightarrow X[\beta]$ и $B \rightarrow [Y\alpha]$, что $[\beta] \Rightarrow_G^+ B\gamma\beta \Rightarrow_G^+ [Y\alpha]$ для некоторых $\gamma \in (N \cup \Sigma)^*$ и $A, B \in N$. Отсюда следует, что в P есть такие правила $C \rightarrow \delta X\beta$ и $B \rightarrow Y\alpha$, что $A \Rightarrow_G^+ B\gamma$ для некоторой цепочки $\gamma \in (N \cup \Sigma)^*$. Таким образом, для G получается $X \triangleleft B$ или $X \underline{\equiv} B$. (Последнее может произойти тогда и только тогда, когда $B = A$.)

Теперь рассмотрим $X \underline{\equiv} [Y\alpha]$. В P' есть правило $[XY\alpha] \rightarrow X[Y\alpha]$, и потому в P есть правило $D \rightarrow \varepsilon XY\alpha$.

Следовательно, если $X \triangleleft [Y\alpha]$ и $X \underline{\equiv} [Y\alpha]$ для грамматики G' , то в P найдутся два правила вида $B \rightarrow Y\alpha$ и $D \rightarrow \varepsilon XY\alpha$ и для G будет $X \triangleleft B$ или $X \underline{\equiv} B$, что нарушает условие (2) определения грамматики слабого предшествования.

Вид правил множества P' позволяет сразу заключить, что грамматика G' обратима, если обратима G . Следовательно, $L(G') = L(G)$. Мы оставляем это в качестве упражнения. \square

Следствие. Каждая обратимая грамматика слабого предшествования однозначна.

Доказательство. Если бы у какой-нибудь цепочки оказалось два разных правых вывода в грамматике G теоремы 5.19, то можно было бы построить разные правые выводы той же цепочки в грамматике G' . \square

Построение, приведенное в доказательстве теоремы 5.19, больше подходит для целей теории, чем в качестве практического инструмента. На практике можно применить гораздо менее изнурительный подход. Мы дадим простой алгоритм преобразования обратимой грамматики слабого предшествования в эквивалентную грамматику простого предшествования. В качестве упражнения предлагаем доказать, что этот алгоритм работает корректно.

Алгоритм 5.15. Переход от обратимого слабого предшествования к простому предшествованию.

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

Выход. Грамматика простого предшествования G' , для которой $L(G') = L(G)$.

Метод.

(1) Допустим, что в $N \cup \Sigma$ нашлись такие X и Y , что $X \sqsubseteq Y$ и $X \lhd Y$ для грамматики G . Устраним из P каждое правило вида $A \rightarrow \alpha X Y \beta$ и заменим его на $A \rightarrow \alpha X [Y \beta]$, где $[Y \beta]$ — новый нетерминал.

(2) Для каждого символа $[Y \beta]$, введенного на шаге (1), заменим правило вида $B \rightarrow Y \beta$ на $B \rightarrow [Y \beta]$ и добавим $[Y \beta] \rightarrow Y \beta$.

(3) Повторим шаг (1) столько раз, сколько он окажется применимым. Если он больше неприменим, остановимся. Результатом будет грамматика G' . \square

Пример 5.40. Пусть G — грамматика из примера 5.37. С помощью алгоритма 5.15 получаем грамматику G' с правилами

$$\begin{aligned} E &\rightarrow E + [T] | + [T] | [T] \\ T &\rightarrow T * F | F \\ F &\rightarrow ([E]) | a \\ [T] &\rightarrow T \\ [E] &\rightarrow E \end{aligned}$$

Шаг (1) применяется к парам $X = ($, $Y = E$ и $X = +$, $Y = T$. Отношения предшествования для грамматики G' приведены на рис. 5.16. \square

E	T	F	$[T]$	$[F]$	α	$($	$)$	$+$	$*$	$$$
E						\doteq	\doteq			\sim
T						$>$	$>$	\doteq	$>$	
F						$>$	$>$	$>$	$>$	
$[T]$						$>$	$>$			$>$
$[E]$						$>$	$>$	$>$	$>$	
α						$>$	$>$	$>$	$>$	
$)$						$>$	$>$	$>$	$>$	
$($	\lhd	\lhd	\lhd	\lhd	\doteq	\lhd	\lhd		\lhd	
$+$	\lhd	\lhd	\doteq			\lhd	\lhd			
$*$		\doteq				$<$	\lhd			
$$$	\lhd	\lhd	\lhd	\lhd		$<$	$<$	$<$	$<$	$($

Рис. 5.16. Матрица простого предшествования.

УПРАЖНЕНИЯ

5.3.1. Какие из следующих грамматик являются грамматиками простого предшествования?

- (а) G_0
- (б) $S \rightarrow \text{если } E, \text{ то } S \text{ иначе } S | a$
 $E \rightarrow E \text{ или } b | b$
- (в) $S \rightarrow AS | A$
 $A \rightarrow (S) | ()$
- (г) $S \rightarrow SA | A$
 $A \rightarrow (S) | ()$

5.3.2. Какие из грамматик упр. 5.3.1 являются грамматиками слабого предшествования?

5.3.3. Какие из грамматик упр. 5.3.1 являются грамматиками (2,1)-предшествования?

5.3.4. Приведите примеры грамматик предшествования, для которых

- (а) отношение \equiv не рефлексивно, не симметрично и не транзитивно;
 (б) отношение \lessdot не иррефлексивно и не транзитивно;
 (в) отношение \gg не иррефлексивно и не транзитивно.

***5.3.5.** Покажите, что каждое регулярное множество определяется грамматикой простого предшествования. Указание: Убедитесь в том, что Ваша грамматика обратима.

***5.3.6.** Покажите, что каждая обратимая грамматика (m, n) -предшествования является LR-грамматикой.

***5.3.7.** Покажите, что каждая грамматика слабого предшествования является LR-грамматикой.

5.3.8. Докажите, что алгоритм 5.12 правильно производит правый разбор.

5.3.9. Докажите, что G —грамматика предшествования тогда и только тогда, когда она—грамматика $(1,1)$ -предшествования.

5.3.10. Докажите лемму 5.3(1).

5.3.11. Докажите, что алгоритм 5.14 правильно производит правый разбор.

5.3.12. Постройте алгоритм правого разбора для обратимых грамматик (m, n) -предшествования.

5.3.13. Докажите следствие теоремы 5.17.

***5.3.14.** Покажите, что алгоритм 5.15 строит грамматику простого предшествования, эквивалентную исходной грамматике.

5.3.15. Для тех грамматик из упр. 5.3.1, которые являются грамматиками слабого предшествования, постройте эквивалентные грамматики простого предшествования.

***5.3.16.** Докажите, что язык $L = \{a0^n1^n \mid n \geq 1\} \cup \{b0^n1^{2n} \mid n \geq 1\}$ не является языком простого предшествования. Указание: Подумайте о том, как вел бы себя на цепочках вида $a0^n1^n$ и $b0^n1^{2n}$ правый анализатор, построенный алгоритмом 5.12, если бы язык L определялся грамматикой простого предшествования.

***5.3.17.** Постройте грамматику $(2,1)$ -предшествования для языка из упр. 5.3.16.

***5.3.18.** Постройте грамматику простого предшествования для языка $\{0^na1^n \mid n \geq 1\} \cup \{0^nb1^{2n} \mid n \geq 1\}$.

***5.3.19.** Покажите, что каждую КС-грамматику без e -правил можно преобразовать в эквивалентную грамматику $(1,1)$ -предшествования.

5.3.20. Для КС-грамматики $G = (N, \Sigma, P, S)$ определим отношения λ , μ и ρ :

- (1) $A\lambda X$, если в P есть правило $A \rightarrow X\alpha$, где α —какая-нибудь цепочка;

- (2) $X\mu Y$, если в P есть правило $A \rightarrow \alpha XY\beta$ для некоторых α и β ; кроме того, $S\mu S$ и SpS ;

- (3) $X\rho A$, если в P есть правило $A \rightarrow \alpha X$, где α —какая-нибудь цепочка.

Покажите, что эти отношения и отношения предшествования Вирта—Бебера связаны следующим образом:

- (а) $\lessdot = \mu\lambda^+$,
 (б) $\equiv = U \{(S, S), (S, \$)\} = \mu$,
 (в) $\gg = \rho^+ \mu\lambda^* \cap ((N \cup \Sigma) \times \Sigma)$

(\vdash обозначает транзитивное, а $*$ —рефлексивное и транзитивное замыкание).

****5.3.21.** Докажите неразрешимость проблемы: является ли данная грамматика грамматикой расширенного предшествования (т. е. грамматикой (m, n) -предшествования для некоторых m и n)?

***5.3.22.** Докажите, что если G —грамматика слабого предшествования, то G —грамматика расширенного предшествования (для некоторых m и n).

5.3.23. Покажите, что $a \in FOLLOW_1(A)$ тогда и только тогда, когда $A \lessdot a$, $A \equiv a$ или $A \gg a$.

5.3.24. Обобщите лемму 5.3 на грамматики расширенного предшествования.

5.3.25. Предположим, что условия расширенного предшествования ослаблены так, что допускается конфликт $\alpha \lessdot w$ и $\alpha \equiv w$, если он порождается только пунктами (1б) и (2б). Разработайте алгоритм разбора типа „перенос—свертка“ для грамматик, удовлетворяющих этому ослабленному определению.

Проблема для исследования

5.3.26. Найдите преобразования, позволяющие превращать грамматики в грамматики простого или слабого предшествования.

Открытая проблема

5.3.27. Порождается ли каждый язык простого предшествования грамматикой простого предшествования, у которой начальный символ не встречается в правых частях правил? Хорошо, если бы это было так, ибо в противном случае мы могли бы пытаться сделать свертку, когда в магазине $\$S$, а на входе S .

Упражнения на программирование

5.3.28. Напишите программу, строящую отношения предшествования Вирта—Вебера для КС-грамматики G . Примените Вашу программу к грамматике языка ПЛ 360, данной в приложении.

5.3.29. Напишите программу, строящую для входной КС-грамматики G алгоритм разбора типа „перенос—свертка“, если G —грамматика простого предшествования. Примените Вашу программу к построению анализатора для языка ПЛ 360.

5.3.30. Напишите программу, проверяющую, является ли грамматика обратимой грамматикой слабого предшествования.

5.3.31. Напишите программу, строящую алгоритм разбора типа „перенос—свертка“ для обратимых грамматик слабого предшествования.

Замечания по литературе

Разбор методом „перенос—свертка“ появился впервые в работе Флойда [1961]. В трактовке этого метода мы следовали статье Ахо и др. [1972]. Грамматики простого предшествования были определены Виртом и Вебером [1966] и независимо Пэрром [1964]. Метод простого предшествования использовался в компиляторах для нескольких языков, включая Euler [Вирт и Вебер, 1966], Алгол W [Баузер и др., 1968] и ПЛ 360 [Вирт, 1968]. Фишер [1969] доказал, что каждый КС-язык, не содержащий e , порождается (не обязательно обратимой) грамматикой предшествования (упр. 5.3.19).

Расширенное предшествование было предложено Виртом и Вебером. Грэй [1969] указал, что некоторые из ранних определений расширенного предшествования некорректны. При $m+n > 3$ расширенное (m, n) -предшествование, по-видимому, принесет на практике мало пользы из-за больших затрат памяти. Методы сокращения объема таблицы анализатора расширенного предшествования изучал Мак-Киман [1966]. Грэхем [1970] доказала интересную теорему о том, что каждый детерминированный язык определяется обратимой грамматикой $(2, 1)$ -предшествования.

Грамматики слабого предшествования введены Ихбия и Морзе [1970]. Теорема 5.19 взята из статьи Ахо и др. [1972].

Для анализаторов типа „перенос—свертка“ возможны несколько схем исправления ошибок. В ходе разбора этого типа можно объявить об ошибке как в фазе переноса, так и в фазе свертки. Если об ошибке сообщает функция переноса, то можно попытаться делать изменения, вставки или удаления, как при LL- или LR-разборе. Если ошибка встречается при свертке, то можно использовать заранее составленный список ошибочных правил, применяя его к верхней части магазина.

Техника исправления ошибок для грамматик простого предшествования рассматривалась Виртом [1968] и Лейниусом [1970¹]. Чтобы усилить способность анализатора простого предшествования обнаруживать ошибки, Лейниус предложил, в частности, проверять после того, как сделана свертка, выполнение ли допустимое отношение предшествования между символами X и A , где A —новый нетерминал на верху магазина, а X —символ, расположенный сразу под A .

¹) См. также [Ахо и Ульман, 1972], [Грэхем и Роудз, 1975]. — Прим. перев

5.4. ДРУГИЕ КЛАССЫ ГРАММАТИК, АНАЛИЗИРУЕМЫХ МЕТОДОМ «ПЕРЕНОС — СВЕРТКА»

Рассмотрим еще несколько подклассов LR-грамматик, для которых существуют алгоритмы синтаксического анализа типа „перенос—свертка“. Это грамматики ограниченного правого контекста, грамматики смешанного предшествования и грамматики операторного предшествования. Мы рассмотрим также язык Флойда—Эванса, который по существу является языком программирования, предназначенный для записи алгоритмов детерминированного разбора.

5.4.1. Грамматики ограниченного правого контекста

Хотелось бы расширить изученный выше класс грамматик слабого предшествования, ослабив требование обратимости. Совсем устраниТЬ это требование нельзя, так как экономный алгоритм разбора для класса всех грамматик предшествования не известен. Но многие грамматики можно анализировать, используя слабое предшествование для локализации правого конца основы, а затем, если грамматика не является обратимой, с помощью локального контекста найти левый конец основы и определить, каким нетерминалом ее заменить.

Широкий класс грамматик, анализируемых подобным способом, образуют грамматики (m, n) -ограниченного правого контекста (ОПК). Неформально $G = (N, \Sigma, P, S)$ будет (m, n) -ОПК-грамматикой, если для каждого правого вывода

$$S' \Rightarrow^* \alpha Aw \Rightarrow, \alpha \beta w$$

в пополненной грамматике $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$ основу β и правило $A \rightarrow \beta$, применяемое для свертки основы в цепочке $\alpha \beta w$, можно однозначно определить так:

(1) Цепочка $\alpha \beta w$ просматривается слева направо, пока не будет прочитана основа.

(2) Решение о том, является ли β основой цепочки $\alpha \beta w$, где βw —префикс этой цепочки, принимается только по цепочке β , т. е. символам слева от β и w символам справа от β .

(3) В качестве основы из кандидатур, предложенных в (2), выбирается самая левая подцепочка, расположенная справа от самого правого нетерминала цепочки $\alpha \beta w$ или включающая его.

Для удобства обозначенний будем добавлять к каждой правово-выводимой цепочке m символов $\$$ слева и n символов $\$$ справа. Тогда можно быть уверенными в том, что в расширенной таким образом правово-выводимой цепочке всегда найдутся по крайней мере m символов слева от основы и n символов справа.

Определение. $G = (N, \Sigma, P, S)$ называется *грамматикой (m, n) -ограниченного правого контекста* (ОПК), если из условий

- (1) $\$^m S' \$^n \Rightarrow_{G'}^* \alpha A w \Rightarrow_{G'}^* \alpha \beta w$ и
 - (2) $\$^m S' \$^n \Rightarrow_{G'}^* \gamma B x \Rightarrow_{G'}^* \gamma \delta x = \alpha' \beta y$ — правые выводы в дополненной грамматике $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$,
 - (3) $|x| \leq |y|$,
 - (4) последние m символов цепочек α и α' , а также первые n символов цепочек w и y совпадают
- вытекает, что $\alpha' A y = \gamma B x$, т. е. $\alpha' = \gamma$, $A = B$ и $y = x$.

Грамматика называется ОПК-грамматикой, если она является (m, n) -ОПК-грамматикой для некоторых m и n .

Если мы считаем вывод (2) „настоящим“, а (1) рассматриваем как возможную причину недоразумения, то условие (3) гарантирует, что левее „настоящей“ основы δ не встретится подцепочки, которую можно было бы принять за основу (β , окруженнюю последними m символами цепочки α и первыми n символами цепочки w). Следовательно, в качестве основы можно выбирать самую левую цепочку, которая „выглядит как основа“. Условие (4) гарантирует, что решение о том, является ли цепочка основой, можно принять, привлекая лишь m символов левого контекста и n символов правого контекста.

Как и для LR(k)-грамматик, дополненная грамматика включается в определение только на тот случай, когда S встречается в правой части какого-нибудь правила. Например, грамматика G с двумя правилами

$$S \rightarrow Sa | a$$

без оговорки относительно дополненной грамматики была бы $(1, 0)$ -ОПК-грамматикой. Но так же, как в примере 5.22, не заглянув на один символ вперед, нельзя решить, как поступить с S в правовыводимой цепочке Sa . Поэтому мы не хотим считать G $(1, 0)$ -ОПК-грамматикой.

В дальнейшем мы докажем, что каждая (m, k) -ОПК-грамматика является LR(k)-грамматикой. Но не для каждой LR(0)-грамматики найдутся такие числа m и n , чтобы она была (m, n) -ОПК-грамматикой, потому что LR-определение позволяет, говоря неформально, использовать в ходе разбора для принятия решения весь кусок правовыводимой цепочки слева от основы, а ОПК-определение ограничивает нас m символами. Разумеется, оба определения ограничивают использование части цепочки, расположенной справа от основы.

Пример 5.41. Грамматика G_1 с правилами

$$\begin{aligned} S &\rightarrow aA c \\ A &\rightarrow Abb | b \end{aligned}$$

является $(1, 0)$ -ОПК-грамматикой. Правовыводимые цепочки (отличные от S' и S) имеют вид $aAb^{2n}c$ или $ab^{2n+1}c$ для $n \geq 0$. Основами могут быть только aAc , Abb и b , так что основу любой правовыводимой цепочки можно однозначно определить, проанализировав эту цепочку слева направо, пока не встретится цепочка aAc или Abb либо символ b , слева от которого стоит a . Заметим, что ни один из символов b в цепочке Abb не может быть основой, поскольку слева от него стоит A или b .

С другой стороны, грамматика G_2 с правилами

$$\begin{aligned} S &\rightarrow aAc \\ A &\rightarrow bAb | b \end{aligned}$$

порождает тот же язык, но она даже не LR-грамматика. \square

Пример 5.42. Грамматика G с правилами

$$\begin{aligned} S &\rightarrow aA | bB \\ A &\rightarrow 0A | 1 \\ B &\rightarrow 0B | 1 \end{aligned}$$

является LR(0)-, но не ОПК-грамматикой, так как в правовыводимых цепочках $a0^n1$ и $b0^n1$ основой служит 1 , но для выяснения того, какое из правил $A \rightarrow 1$ и $B \rightarrow 1$ применить для свертки основы, недостаточно знать ограниченное число символов, расположенных непосредственно слева от 1 .

Для формального доказательства рассмотрим выводы

$$\$^m S' \$^n \Rightarrow_r^* \$^m a 0^n 1 \$^n \Rightarrow_r \$^m a 0^n 1 \n$

и

$$\$^m S' \$^n \Rightarrow_r^* \$^m b 0^n 1 \$^n \Rightarrow_r \$^m b 0^n 1 \n$

По определению ОПК-грамматики $\alpha = \$^m a 0^n$, $\alpha' = \gamma = \$^m b 0^n$, $\beta = \delta = 1$ и $y = w = x = \n . Тогда α и α' оканчиваются одними и теми же m символами 0^n , w и y начинаются n символами $\n и $|x| \leq |y|$, но $\alpha' A y \neq \gamma B x$ (A и B соответствуют в ОПК-определении сами себе).

Грамматика с правилами

$$\begin{aligned} S &\rightarrow aA | bA \\ A &\rightarrow 0A | 1 \end{aligned}$$

порождает тот же язык и является $(0, 0)$ -ОПК-грамматикой. \square

Условие (3) в определении ОПК-грамматики может сначала показаться лишним. Но именно оно гарантирует, что если в правовыводимой цепочке $\alpha' \beta y$ самой левой подцепочки, совпадающей с правой частью некоторого правила $A \rightarrow \beta$, является β и

она имеет в $\alpha'\beta y$ корректные правый и левый контексты, то получающаяся после свертки цепочки $\alpha'Ay$ будет правовыводимой.

ОПК-грамматики связаны с некоторыми из исследованных в этой главе классов грамматик. Как уже упоминалось, они образуют подмножество LR-грамматик. Кроме того, это грамматики расширенного предшествования, и каждая обратимая грамматика (m, n) -предшествования является ОПК-грамматикой. Класс $(1, 1)$ -ОПК-грамматик содержит все обратимые грамматики слабого предшествования. Докажем спачала последнее утверждение.

Теорема 5.20. Если $G = (N, \Sigma, P, S)$ — обратимая грамматика слабого предшествования, то она является $(1, 1)$ -ОПК-грамматикой.

Доказательство. Допустим, что $(1, 1)$ -ОПК-условие нарушается, т. е. существуют два вывода

$$\begin{aligned} \$\$ \Rightarrow^* \alpha Aw &\Rightarrow_r \alpha\beta w \\ \$\$ \Rightarrow^* \gamma Bx &\Rightarrow_r \gamma\delta x = \alpha'\beta y \end{aligned}$$

в которых α и α' оканчиваются одним и тем же символом, w и y начинаются одним и тем же символом и $|x| \leq |y|$, но $\gamma\delta x \neq \alpha'Ay$. Так как G — грамматика слабого предшествования, то, применяя теорему 5.14 к цепочке $\gamma\delta x$, находим, что отношение \Rightarrow впервые встречается между δ и x . Применяя эту теорему к $\alpha\beta w$, находим \Rightarrow между β и w , а так как w и y начинаются одним и тем же символом, то \Rightarrow оказывается между β и y . Поэтому $|\alpha'\beta| \geq |\gamma\delta|$. Так как дано, что $|x| \leq |y|$, то должно быть $\alpha'\beta = \gamma\delta$ и $x = y$.

Если мы сможем показать, что $\beta = \delta$, то этим докажем, что $\alpha' = \gamma$. Но из обратности следует, что $A = B$, и, значит, мы получим противоречие с предположением, что $\gamma\delta x \neq \alpha'Ay$.

Если $\beta \neq \delta$, то одна из этих цепочек должна быть суффиксом другой. Чтобы показать, что $\beta = \delta$, исследуем отдельно каждый из возможных случаев.

Случай 1: $\beta = \varepsilon X\delta$ для некоторых ε и X . Так как X — последний символ цепочки γ , то, применяя теорему 5.14 к правовыводимой цепочке $\gamma\delta x$, имеем $X \ll B$ или $X \sqsubseteq B$. Это нарушает условие слабого предшествования.

Случай 2: $\beta = \varepsilon X\delta$ для некоторых ε и X . Этот случай симметричен только что рассмотренному.

Итак, $\beta = \delta$ и G — $(1, 1)$ -ОПК-грамматика. \square

Теорема 5.21. Каждая (m, k) -ОПК-грамматика является $LR(k)$ -грамматикой.

Доказательство. Пусть $G = (N, \Sigma, P, S)$ является (m, k) -ОПК-грамматикой, но не $LR(k)$ -грамматикой. Тогда по лемме 5.2

в дополненной грамматике G' существуют выводы

$$\begin{aligned} S' &\Rightarrow^* \alpha Aw \Rightarrow_r \alpha\beta w \\ S' &\Rightarrow^* \gamma Bx \Rightarrow_r \gamma\delta x = \alpha'y \end{aligned}$$

где $|\gamma\delta| \geq |\alpha\beta|$ и $FIRST_k(y) = FIRST_k(w)$, но $\gamma\delta x \neq \alpha'Ay$. Если окружить все цепочки символами $\$$ и положить $\alpha' = \alpha$, то (m, k) -ОПК-условие будет нарушено. \square

Следствие. Каждая ОПК-грамматика однозначна. \square

Теперь перейдем к построению алгоритма разбора типа „перенос — свертка“ для ОПК-грамматик и обсудим эффективную реализацию этого алгоритма. Допустим, что алгоритм указанного типа используется для анализа ОПК-грамматики G и что он находится в конфигурации (α, w, π) . Тогда можно задать множества \mathcal{H} и \mathcal{N} , которые скажут нам, появилась ли основа правовыводимой цепочки αw наверху магазина (т. е. образует ли она суффикс цепочки α) или же правый конец основы расположен где-то в w (и тогда надо делать перенос). Если основа в магазине, то эти множества скажут, какова она и какое правило применить для ее свертки.

Определение. Пусть $G = (N, \Sigma, P, S)$ — ОПК-грамматика. Для $A \in N$ определим $\mathcal{H}_{m,n}^G(A)$ как множество таких троек (α, β, x) , что $|\alpha|=m$, $|x|=n$ и в дополненной грамматике есть вывод

$$\$^m S' \$^n \Rightarrow^* \gamma\alpha Axy \Rightarrow_r \gamma\alpha\beta xy$$

Множество $\mathcal{N}_{m,n}^G$ пусть состоит из таких пар (α, x) , что

(1) либо $|\alpha|=m+l$, где l — длина самой длинной правой части правил из P , либо $|\alpha| < m+l$ и α начинается с $\$$;

(2) $|x|=n$;

(3) существует вывод $\$^m S' \$^n \Rightarrow^* \beta Ay \Rightarrow_r \beta\gamma y$, где αx — подцепочка цепочки $\beta\gamma y$, расположенная так, что α лежит внутри цепочки $\beta\gamma$ и не содержит ее последнего символа.

В очевидных случаях мы будем опускать индексы G , m и n в обозначениях $\mathcal{H}(A)$ и \mathcal{N} .

Идея состоит в том, что появление подцепочки $\alpha\beta x$ в ходе просмотра слева направо правовыводимой цепочки должно указывать, что β — основа и ее надо свернуть к A , если $(\alpha, \beta, x) \in \mathcal{H}(A)$. Появление такой цепочки αx , что $(\alpha, x) \in \mathcal{N}$, указывает на то, что основа еще не получена, но, возможно, расположена справа от α . Лемма 5.6 подтверждает, что так оно и есть.

Лемма 5.6. Грамматика $G = (N, \Sigma, P, S)$ является (m, n) -ОПК-грамматикой тогда и только тогда, когда выполняются следующие условия:

(1) Пусть $A \rightarrow \beta$ и $B \rightarrow \delta$ — разные правила. Тогда если $(\alpha, \beta, x) \in \mathcal{H}_{m,n}(A)$ и $(\gamma, \delta, x) \in \mathcal{H}_{m,n}(B)$, то $\alpha\beta$ — не суффикс цепочки $\gamma\delta$, и обратно.

(2) Для всех $A \in N$ из $(\alpha, \beta, x) \in \mathcal{H}_{m,n}(A)$ следует, что $(\theta\alpha\beta, x)$ не содержится в $\mathcal{N}_{m,n}$ ни для какой цепочки θ .

Доказательство. Достаточность. Допустим, что G — не (m, n) -ОПК-грамматика. Тогда в дополненной грамматике G' найдутся выводы

$$\begin{aligned} \$^m S' \$^n &\Rightarrow_r^* \alpha A w \Rightarrow_r \alpha \beta w \\ \$^m S' \$^n &\Rightarrow_r^* \gamma B x \Rightarrow_r \gamma \delta x = \alpha' \beta y \end{aligned}$$

где α и α' совпадают в последних m позициях, w и y совпадают в первых n позициях и $|x| \leq |y|$, но $\gamma B x \neq \alpha' A y$. Пусть ε состоит из последних m символов цепочки α , а z из первых n символов цепочки w . Тогда $(\varepsilon, \beta, z) \in \mathcal{H}(A)$. Если $x \neq y$ и $|x| \leq |y|$, то должно быть $(\theta\beta, z) \in \mathcal{N}$ для некоторой цепочки θ , так что условие (2) нарушается. Если $x = y$, то $(\eta, \beta, z) \in \mathcal{H}(B)$, где η состоит из последних m символов цепочки γ . Если $A \rightarrow \beta$ и $B \rightarrow \delta$ — это одно и то же правило, то при $x = y$ мы вопреки предположению получаем $\gamma B x = \alpha' A y$. Но так как одна из цепочек $\eta\beta$ и $\varepsilon\beta$ является суффиксом другой, то, если правила $A \rightarrow \beta$ и $B \rightarrow \delta$ различны, нарушается условие (1).

Необходимость. Если нарушается условие (1) или (2), легко доказать, что нарушается (m, n) -ОПК-условие. Эту часть доказательства оставляем в качестве упражнения. \square

Теперь можно перейти к описанию алгоритма разбора типа „перенос—свертка“ для ОПК-грамматик. Так как для этого нужно знать множества $\mathcal{H}(A)$ и \mathcal{N} , займемся сначала их вычислением.

Алгоритм 5.16. Построение множеств $\mathcal{H}_{m,n}(A)$ и $\mathcal{N}_{m,n}$.

Вход. Приведенная грамматика $G = (N, \Sigma, P, S)$.

Выход. Множества $\mathcal{H}_{m,n}(A)$ для $A \in N$ и $\mathcal{N}_{m,n}$.

Метод.

(1) Пусть l — длина самой длинной правой части правила. Вычислим множество \mathcal{S} таких цепочек γ , что

- (а) $|\gamma| = m + n + l$ или $|\gamma| < m + n + l$ и γ начинается с $\n ;
- (б) γ — подцепочка цепочки $\alpha\beta w$, где $\alpha\beta w$ — правовыводимая цепочка с основой β и $w = \text{FIRST}_n(w)$;
- (в) γ содержит хотя бы один нетерминал.

Здесь можно применить метод, подобный тому, что применялся в первой части алгоритма 5.13.

(2) Для $A \in N$ пусть $\mathcal{H}(A)$ содержит каждую тройку (α, β, x) , для которой в \mathcal{S} пайдется цепочка $\gamma\alpha A xy$, в P — правило $A \rightarrow \beta$ и $|\alpha| = m$, $|x| = n$.

(3) Пусть \mathcal{N} содержит каждую пару (α, x) , для которой в \mathcal{S} найдется цепочка $\gamma\beta y$, в P — правило $B \rightarrow \delta$, αx — подцепочка цепочки $\gamma\beta y$, α внутри цепочки $\gamma\beta$, за исключением ее последнего символа. Требуется также, чтобы $|x| = n$ и $|\alpha| = m + l$, где l — самая длинная правая часть правила, либо α начинается с $\n и $|\alpha| < m + l$. \square

Теорема 5.22. Алгоритм 5.6 правильно вычисляет множества $\mathcal{H}(A)$ и \mathcal{N} .

Доказательство. Упражнение. \square

Алгоритм 5.17. Построение алгоритма типа „перенос—свертка“ для ОПК-грамматик.

Вход. (m, n) -ОПК-грамматика $G = (N, \Sigma, P, S)$, дополненная до $G' = (N', \Sigma, P, S')$.

Выход. Алгоритм $\mathcal{A} = (f, g)$ типа „перенос—свертка“ для грамматики G .

Метод. Функции f и g зададим так:

- (1) $f(\alpha, w) = \text{перенос}$, если $(\alpha, w) \in \mathcal{H}_{m,n}$,
- (2) $f(\alpha, w) = \text{свертка}$, если $\alpha = \alpha_1 \alpha_2$ и $(\alpha_1, \alpha_2, w) \in \mathcal{H}_{m,n}(A)$ для некоторого A , но не верно, что $A = S'$, $\alpha_1 = \$$ и $\alpha_2 = S$,
- (3) $f(\$^m S, \$^n) = \text{допуск}$,
- (4) $f(\alpha, w) = \text{ошибка}$ в остальных случаях,
- (5) $g(\alpha, w) = i$, если $\alpha = \alpha_1 \alpha_2$, $(\alpha_1, \alpha_2, w) \in \mathcal{H}(A)$ и $A \rightarrow \alpha_2$ — правило с номером i ,
- (6) $g(\alpha, w) = \text{ошибка}$ в остальных случаях. \square

Теорема 5.23. Алгоритм 5.17 строит корректный алгоритм разбора типа „перенос—свертка“ для грамматики G .

Доказательство. В силу леммы 5.6 при определении функций f и g не возникает недоразумений. По определению множества $\mathcal{H}(A)$ всякий раз, когда делается свертка, свертываемая цепочка α_2 служит основой некоторой цепочки $\beta\alpha_1\alpha_2wz$. Если бы она оказалась основой какой-нибудь другой цепочки $\beta'\alpha_1\alpha_2wz'$, то нарушилось бы ОПК-условие. Единственный трудный момент — доказательство того, что выполняется условие (3) определения ОПК-грамматики: $|x| \leq |y|$. Нетрудно показать, что в качестве выводов в ОПК-определении можно взять выводы цепочек $\beta\alpha_1\alpha_2wz$ и $\beta'\alpha_1\alpha_2wz'$ (в любом порядке), так что условие (3) выполняется. \square

В алгоритме $\mathcal{A} = (f, g)$, построенном алгоритмом 5.17, функции f и g зависят, очевидно, только от ограничений части цепочек, которые могут быть их аргументами, хотя в различные моменты приходится рассматривать подцепочки различной длины. Приведем реализацию обеих функций f и g в виде дерева реше-

ний. Сначала для данных цепочек — α в магазине и x на входе — можно пойти по ветви, соответствующей первым n символам цепочки x . Потом для каждой такой цепочки из n символов можно начать просматривать α в обратном направлении, на каждом шаге решая, продолжать ли просмотр дальше или объявить об ошибке, переносе или свертке. Если объявлена свертка, то у нас достаточно информации, чтобы точно сказать, какое применить правило, так что в дерево решений можно включить и функцию g . Для принятия решений можно также использовать обобщенный алгоритм Домёлки (см. упражнения разд. 4.1).

Пример 5.43. Рассмотрим грамматику G с правилами

- (0) $S' \rightarrow S$
- (1) $S \rightarrow 0A$
- (2) $S \rightarrow 1S$
- (3) $A \rightarrow 0A$
- (4) $A \rightarrow 1$

Это $(1,0)$ -ОПК-грамматика. Для вычисления множеств $\mathcal{H}(A)$, $\mathcal{H}(S)$ и \mathcal{N} нам нужно множество цепочек длины 3 или меньше, которые могут встречаться в активных префиксах правовыводимых цепочек и содержать нетерминал. Это множество состоит из цепочек $\$S'$, $\$S$, $\$0A$, $\$1S$, $00A$, $11S$, $10A$ и их подцепочек.

Теперь вычисляем

$$\begin{aligned}\mathcal{H}_{1,0}(S') &= \{(\$, S, e)\} \\ \mathcal{H}_{1,0}(S) &= \{(\$, 0A, e), (\$, 1S, e), (1, 0A, e), (1, 1S, e)\} \\ \mathcal{H}_{1,0}(A) &= \{(0, 0A, e), (0, 1, e)\}\end{aligned}$$

Множество \mathcal{N} состоит из пар (α, e) , где α — любая из цепочек $\$$, $\$0$, $\$00$, $\$000$, $\$1$, $\$11$, $\$10$, 100 и 110 . Функции f и g приведены на рис. 5.17. Под „окончанием цепочки α “ подразумевается ее кратчайший суффикс, необходимый для определения значений $f(\alpha, e)$ и $g(\alpha, e)$.

Реализация f и g в виде дерева решений приведена на рис. 5.18. Так как $n=0$, то ветвление, соответствующее цепочкам длины n , отсутствует. Вершины с метками A и S , лежащие ниже уровня 1, опущены, так как все они, конечно, имеют выход ошибки. \square

5.4.2. ГРАММАТИКИ СМЕШАННОЙ СТРАТЕГИИ ПРЕДШЕСТВОВАНИЯ

К сожалению, реализация алгоритма $\mathcal{A} = (f, g)$, построенного алгоритмом 5.17, требует довольно больших затрат памяти для хранения функций f и g . Можно определить класс грамматик,

анализируемых методом „перенос — свертка“ с меньшими затратами, если использовать предшествование для локализации правого конца основы, а левый конец основы и нетерминал, которым надо ее заменить, определять с помощью локального контекста.

Окончание цепочки α	$f(\alpha, e)$	$g(\alpha, e)$
$\$0A$	свертка	1
$10A$	свертка	1
$00A$	свертка	3
$\$1S$	свертка	2
$11S$	свертка	2
01	свертка	4
00	перенос	
$\$0$	перенос	
$\$10$	перенос	
110	перенос	
$\$1$	перенос	
$\$11$	перенос	
111	перенос	
$\$$	перенос	
$\$S$	допуск	

Рис. 5.17. Функции переноса и свертки.

Пример 5.44. Рассмотрим (необратимую) грамматику слабого предшествования G с правилами

$$\begin{aligned}S &\rightarrow aA \mid bB \\ A &\rightarrow CA1 \mid C1 \\ B &\rightarrow DBE1 \mid DE1 \\ C &\rightarrow 0 \\ D &\rightarrow 0 \\ E &\rightarrow 1\end{aligned}$$

G порождает язык $\{a0^n1^n \mid n \geq 1\} \cup \{b0^n1^{2n} \mid n \geq 1\}$, который, как будет показано в гл. 8, не является языком простого предшествования. Отношения предшествования для грамматики G даны на рис. 5.19. Заметим, что G — необратимая грамматика, потому что 0 — правая часть двух правил: $C \rightarrow 0$ и $D \rightarrow 0$. Однако $\mathcal{H}_{1,0}(C) = \{(a, 0, e), (C, 0, e)\}$ и $\mathcal{H}_{1,0}(D) = \{(b, 0, e), (D, 0, e)\}$. Поэтому если 0 выделен в качестве основы правовыводимой цепочки, то символ, расположенный непосредственно слева от 0 , определяет, свернуть ли 0 к C или к D . А именно, первое из этих правил выбирается, если это символ a или C , а второе — если это b или D . \square

Пример 5.44 подсказывает следующее определение.

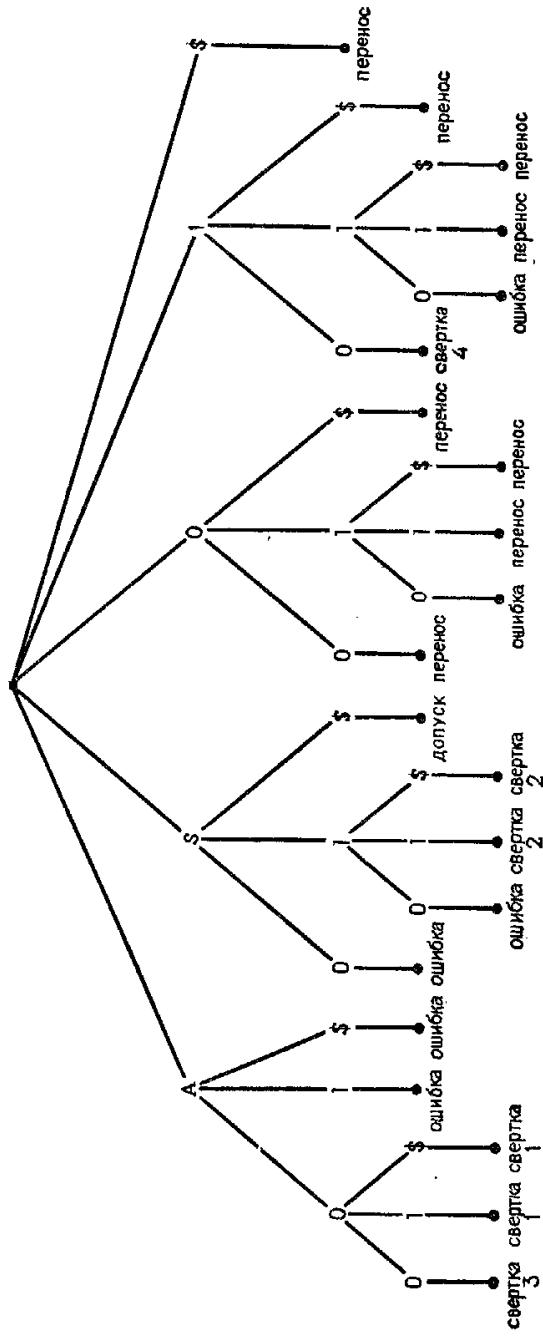


Рис. 5.18. Дерево решений.

Определение. Пусть $G = (N, \Sigma, P, S)$ — приведенная КС-грамматика без e -правил. Назовем G грамматикой $(p, q; m, n)$ -смешанной стратегии предшествования (ССП), если выполняются такие условия:

(1) Отношение (p, q) -предшествования \triangleright не пересекается с объединением отношений (p, q) -предшествования \triangleleft и \equiv .

(2) Если $A \rightarrow \alpha\beta$ и $B \rightarrow \beta$ — разные правила из P , то утверждения

- $\mathcal{H}_{m, n}(A)$ содержит $(\gamma, \alpha\beta, x)$,
- $\mathcal{H}_{m, n}(B)$ содержит (δ, β, x) ,
- δ состоит из последних m символов цепочки $\gamma\alpha$

не могут быть верными одновременно.

	<i>S</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>a</i>	<i>b</i>	<i>0</i>	<i>1</i>	<i>\$</i>
<i>S</i>	—	—	—	—	—	—	—	—	—	—	—
<i>A</i>	—	—	—	—	—	—	—	—	—	—	—
<i>B</i>	—	—	—	—	—	—	—	—	—	—	—
<i>C</i>	—	—	—	—	—	—	—	—	—	—	—
<i>D</i>	—	—	—	—	—	—	—	—	—	—	—
<i>E</i>	—	—	—	—	—	—	—	—	—	—	—
<i>a</i>	—	—	—	—	—	—	—	—	—	—	—
<i>b</i>	—	—	—	—	—	—	—	—	—	—	—
<i>0</i>	—	—	—	—	—	—	—	—	—	—	—
<i>1</i>	—	—	—	—	—	—	—	—	—	—	—
<i>\$</i>	—	—	—	—	—	—	—	—	—	—	—

Рис. 5.19. Отношения предшествования для грамматики G .

(1,1; 1,0)-ССП-грамматику будем называть *простой ССП-грамматикой*. Грамматика из примера 5.44 — простая ССП-грамматика. На самом деле каждая обратимая грамматика слабого предшествования является простой ССП-грамматикой.

Пусть $l(A) = \{X \mid X \triangleleft A \text{ или } X \equiv A\}$. Для простой ССП-грамматики приведенное выше условие (2) сводится к следующему:

- Если $A \rightarrow \beta'X\beta$ и $B \rightarrow \beta$ содержатся в P , то $X \notin l(B)$.
- Если $A \rightarrow \beta$ и $B \rightarrow \beta$ содержатся в P и $A \neq B$, то $l(A)$ и $l(B)$ не пересекаются.

Условия (1) и (2а)—это условия слабого предшествования. Таким образом, на простую ССП-грамматику можно смотреть как па (возможно, не обратимую) грамматику, в которой одного символа левого контекста достаточно, чтобы сделать правильный выбор из правил с одинаковыми правыми частями (условие (2б)). В этом и заключается смешанная стратегия разбора, использующая предшествование, для ССП-грамматик.

Алгоритм 5.18. Построение алгоритма разбора для ССП-грамматик.

Вход. $(p, q; m, n)$ -ССП-грамматика $G = (N, \Sigma, P, S)$ с занумерованными правилами.

Выход. Алгоритм $\mathcal{A} = (f, g)$ типа „перенос—свертка“ для грамматики G .

Метод.

- (1) Пусть $|\alpha| = p$ и $|x| = q$. Тогда
 - (a) $f(\alpha, x) =$ перенос, если $\alpha \lessdot x$ или $\alpha \equiv x$,
 - (б) $f(\alpha, x) =$ свертка, если $\alpha \gg x$.
- (2) $f(\$, \$) =$ допуск.
- (3) $f(\gamma, w) =$ ошибка в остальных случаях.
- (4) Пусть $\mathcal{H}_{m, n}(A)$ содержит (α, β, x) и $A \rightarrow \beta$ —правило с номером i . Тогда $g(\alpha\beta, x) = i$.
- (5) $g(\gamma, w) =$ ошибка в остальных случаях. \square

Теорема 5.24. Алгоритм 5.18 строит алгоритм $\mathcal{A} = (f, g)$, корректный для грамматики G .

Доказательство. Упражнение. Достаточно доказать, что каждая ССП-грамматика является ОПК-грамматикой, а затем показать, что функции f и g , определяемые алгоритмом 5.18, согласуются с функциями, определяемыми алгоритмом 5.17. \square

5.4.3. Грамматики операторного предшествования

Существует эффективная процедура разбора для класса грамматик, называемых грамматиками операторного предшествования. Разбор методом операторного предшествования прост для реализации. Он использовался во многих компиляторах.

Определение. Операторной грамматикой называется приведенная КС-грамматика без e -правил, в которой правые части правил не содержат смежных нетерминалов.

Для операторной грамматики отношения предшествования можно задать на множестве терминалов плюс символ $\$$, игнорируя нетерминалы. Пусть $G = (N, \Sigma, P, S)$ —операторная грам-

матика и $\$\!$ —новый символ. Зададим отношения операторного предшествования на множестве $\Sigma \cup \{\$\}$:

- (1) $a \equiv b$, если $A \rightarrow \alpha a \gamma \beta \in P$ и $\gamma \in N \cup \{e\}$,
- (2) $a \lessdot b$, если $A \rightarrow \alpha a B \beta \in P$ и $B \Rightarrow^+ \gamma b \delta$, где $\gamma \in N \cup \{e\}$,
- (3) $a \gg b$, если $A \rightarrow \alpha B \beta \in P$ и $B \Rightarrow^+ \delta a \gamma$, где $\gamma \in N \cup \{e\}$,
- (4) $\$ \lessdot a$, если $S \Rightarrow^+ \gamma a \alpha$ и $\gamma \in N \cup \{e\}$,
- (5) $a \gg \$$, если $S \Rightarrow^+ \alpha a \gamma$ и $\gamma \in N \cup \{e\}$.

Операторная грамматика G называется грамматикой операторного предшествования, если между любыми двумя терминальными символами выполняется не более одного отношения операторного предшествования.

Пример 5.45. Грамматика G_0 —классический пример грамматики операторного предшествования

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

Отношения операторного предшествования для этой грамматики приведены на рис. 5.20. \square

Для грамматики операторного предшествования можно эффективно находить „остовные“ разборы. Идея синтаксического ана-

		(α	*	+)	$\$$
)					
α							
*		\lessdot	\lessdot	\gg	\gg	\gg	\gg
+		\lessdot	\lessdot	\lessdot	\gg	\gg	\gg
()		\lessdot	\lessdot	\lessdot	\lessdot	\neq	
$\$$		\lessdot	\lessdot	\lessdot	\lessdot		

Рис. 5.20. Отношения операторного предшествования для грамматики G_0 .

лизи та же, что и для грамматик простого предшествования. Легко убедиться в справедливости следующей теоремы.

Теорема 5.25. Пусть $G = (N, \Sigma, P, S)$ —операторная грамматика и $\$S\$ \Rightarrow^* \alpha A w \Rightarrow^* \alpha \beta w$. Тогда

- (1) отношение операторного предшествования \lessdot или \equiv выполняется между последовательными терминалами (и символом $\$$) цепочки α ;

(2) отношение \triangleleft выполняется между самым правым терминалом цепочки α и самым левым терминалом цепочки β ;

(3) отношение \sqsubseteq выполняется между последовательными терминалами цепочки β ;

(4) отношение \triangleright выполняется между самым правым терминалом цепочки β и первым символом цепочки w .

Доказательство. Упражнение. \square

Следствие. Если G —грамматика операторного предшествования, то к каждому из утверждений (1)–(4) теоремы 5.25 можно добавить слова „и никакие другие отношения не выполняются“.

Доказательство. Вытекает непосредственно из определения грамматики операторного предшествования. \square

Итак, с помощью алгоритма разбора типа „перенос—свертка“ легко выделить терминальные символы, входящие в основу. Однако возникают проблемы в связи с нетерминальными символами, поскольку на них не определены отношения операторного предшествования. Тем не менее тот факт, что мы располагаем операторной грамматикой, позволяет строить „остовный“ правый разбор.

Пример 5.46. Разберем цепочку $(a+a)*a$ в соответствии с отношениями операторного предшествования для грамматики G_0 , приведенными на рис. 5.20. Мы не будем заботиться о нетерминалах, а просто заменим каждый из них символом E . Тогда нам не надо беспокоиться о том, свернуть ли F к T или T к F (хотя в данном случае с такими проблемами можно было бы справиться, выйдя за пределы метода операторного предшествования). Можно эффективно сделать разбор в соответствии с грамматикой G с правилами

- (1) $E \rightarrow E + E$
- (3) $E \rightarrow E * E$
- (5) $E \rightarrow (E)$
- (6) $E \rightarrow a$

полученной из G_0 заменой всех нетерминалов символом E и устранением всех цепных правил. (Заметим, что в операторной грамматике правилом, не содержащим терминалов в правой части, может быть только целное правило.)

Грамматика G , очевидно, не однозначная, но отношения операторного предшествования гарантируют единственность искомого разбора. Алгоритм разбора \mathcal{A} для грамматики G задается определяемыми ниже функциями f и g . Цепочки, которые служат аргументами этих функций, будут состоять только из тер-

миналов грамматики G_0 и символов $\$$ и E . Далее, γ обозначает E или пустую цепочку, b и c —терминалы или $\$$.

$$(1) f(b\gamma, c) = \begin{cases} \text{перенос, если } b \triangleleft c \text{ или } b \sqsubseteq c, \\ \text{свертка, если } b \triangleright c, \\ \text{допуск, если } b = \$, \gamma = E \text{ и } c = \$, \\ \text{ошибка в остальных случаях.} \end{cases}$$

$$(2) g(b\gamma a, x) = \begin{cases} 6, & \text{если } b \triangleleft a, \\ g(bE * E, x) = 3, & \text{если } b \triangleleft *, \\ g(bE + E, x) = 1, & \text{если } b \triangleleft +, \\ g(b\gamma(E), x) = 5, & \text{если } b \triangleleft (, \\ g(\alpha, x) = \text{ошибка в остальных случаях.} \end{cases}$$

Таким образом, для входной цепочки $(a+a)*a$ алгоритм \mathcal{A} сделает такую последовательность шагов:

$$\begin{aligned} [\$, (a+a)*a \$, e] &\vdash^s [\$, (a+a)*a \$, e] \\ &\vdash^s [\$ (a, + a)*a \$, e] \\ &\vdash^r [\$ (E, + a)*a \$, 6] \\ &\vdash^s [\$ (E + , a)*a \$, 6] \\ &\vdash^s [\$ (E + a,)*a \$, 6] \\ &\vdash^r [\$ (E + E,)*a \$, 66] \\ &\vdash^r [\$ (E,)*a \$, 661] \\ &\vdash^s [\$ (E), *a \$, 661] \\ &\vdash^r [\$ E, *a \$, 6615] \\ &\vdash^s [\$ E *, a \$, 6615] \\ &\vdash^s [\$ E * a, \$, 6615] \\ &\vdash^r [\$ E * E, \$, 66156] \\ &\vdash^r [\$ E, \$, 661563] \\ &\vdash \text{допуск} \end{aligned}$$

Можно убедиться в том, что 661563—действительно оставшийся правый разбор цепочки $(a+a)*a$ в грамматике G . Этот оставшийся разбор цепочки $(a+a)*a$ можно представить в виде дерева на рис. 5.21. \square

Заметим, что можно было бы дополнить дерево оставшегося разбора на рис. 5.21 так, чтобы получилось соответствующее дерево в грамматике G_0 . Но на практике в этом часто нет необходимости. Цель построения дерева—трансляция, а естественный перевод нетерминала E , T или F грамматики G_0 —это программа машины, вычисляющая выводимое из него выражение. Поэтому если применяется правило $E \rightarrow T$ или $T \rightarrow F$, то перевод правой части будет скорее всего тот же, что и левой части.

Пример 5.46 представляет собой частный случай метода, работающего для многих грамматик, особенно для тех, которые определяют языки, являющиеся множествами арифметических выражений. Этот метод включает построение новой грамматики, получаемой из старой заменой всех нетерминалов одним нетерминалом и устранением цепных правил. Для грамматики операторного предшествования всегда можно с помощью алгоритма

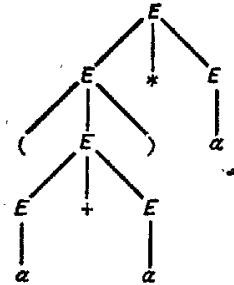


Рис. 5.21. Дерево оственного разбора.

типа „перенос—свертка“ найти один разбор для каждой входной цепочки. Очень часто для целей трансляции достаточно новой грамматики и соответствующего ей анализатора. В таких ситуациях разбор методом операторного предшествования особенно прост и эффективен.

Определение. Пусть $G = (N, \Sigma, P, S)$ — операторная грамматика. Остовной грамматикой для G назовем грамматику $G_s = (\{S\}, \Sigma, P', S)$, содержащую каждое правило $S \rightarrow X_1 \dots X_m$, для которого в P найдется такое правило $A \rightarrow Y_1 \dots Y_m$, что для $1 \leq i \leq m$

- (1) $X_i = Y_i$, если $Y_i \in \Sigma$,
- (2) $X_i = S$, если $Y_i \in N$.

Но в P' не должно быть правила $S \rightarrow S$.

Обращаем внимание читателя на то, что $L(G) \equiv L(G_s)$ и, вообще говоря, $L(G_s)$ может содержать цепочки, не принадлежащие $L(G)$. Теперь опишем алгоритм типа „перенос—свертка“ для грамматик операторного предшествования.

Алгоритм 5.19. Построение анализатора, использующего операторное предшествование.

Вход. Грамматика операторного предшествования $G = (N, \Sigma, P, S)$.

Выход. Алгоритм разбора $\mathcal{A} = (f, g)$ типа „перенос—свертка“ для грамматики G_s .

Метод. Пусть β обозначает S или e .

- (1) $f(a\beta, b) =$ перенос, если $a \lessdot b$ или $a \equiv b$.
- (2) $f(a\beta, b) =$ свертка, если $a \gg b$.
- (3) $f(\$S, \$) =$ допуск.
- (4) $f(\alpha, w) =$ ошибка в остальных случаях.
- (5) $g(a\beta b\gamma, w) = i$, если
 - (a) $\beta = S$ или e ,
 - (b) $a \lessdot b$,
 - (b) отношение \equiv выполняется между последовательными терминальными символами цепочки γ , если они существуют,
 - (g) $S \rightarrow \beta b\gamma$ — правило с номером i грамматики G_s .
- (6) $g(\alpha, w) =$ ошибка в остальных случаях. \square

Применение алгоритма 5.19 к грамматике G продемонстрировано на примере 5.46. Для доказательства корректности алгоритма 5.19 нам понадобятся две леммы.

Лемма 5.7. Если α — правовыводимая цепочка операторной грамматики, то α не содержит смежных нетерминалов.

Доказательство. Элементарная индукция по длине вывода цепочки α . \square

Лемма 5.8. Если α — правовыводимая цепочка операторной грамматики, то символ, расположенный непосредственно слева от основы, не может быть нетерминалом.

Доказательство. Если бы этот символ был нетерминалом, то правовыводимая цепочка, к которой свертывается α , содержала бы два смежных нетерминалов. \square

Теорема 5.26. Алгоритм $\mathcal{A} = (f, g)$, построенный алгоритмом 5.19, правильно выдает оставные разборы всех цепочек языка $L(G)$.

Доказательство. В силу следствия из теоремы 5.25 первое встреченное отношение \gg и предыдущее \lessdot правильно выделяют основу. Лемма 5.7 оправдывает условие, что β может быть только S или e (а не любой цепочкой из S^*). Лемма 5.8 объясняет, почему в пункте (5) β включается в основу. \square

5.4.4. Язык Флойда — Эванса

То, чем мы сейчас займемся, не новый алгоритм разбора, а скорее язык, на котором можно описывать детерминированные (безвозвратные) алгоритмы нисходящего и восходящего анализов. Он называется языком Флойда — Эванса. С помощью этого синтаксического метаязыка было реализовано несколько компи-

ляторов. Этот язык не совсем правильно называют еще языком продукций или правил, так как операторы языка не связаны с определенными правилами грамматики. Программа, написанная на языке Флойда—Эванса, определяет алгоритм разбора, принимающий решения под воздействием управляющего устройства с конечной памятью¹⁾.

Анализатор, записанный на языке Флойда—Эванса, представляет собой список операторов определенного вида. Каждый оператор имеет метку, и эти метки можно рассматривать как состояния управляющего устройства. Предполагается, что различные операторы не могут иметь одну и ту же метку. Операторы работают над входной цепочкой и магазином и приводят к построению правого разбора. Текущее состояние процесса анализа можно представить в виде конфигурации

$$(q, \$X_m \dots X_1, a_1 \dots a_n \$, \pi)$$

где

- (1) q — метка текущего активного оператора;
- (2) $X_m \dots X_1$ — содержимое магазина, причем X_1 — верхний символ ($\$$ — маркер дна магазина);
- (3) $a_1 \dots a_n$ — необработанная часть входной цепочки ($\$$ используется также в качестве правого концевого маркера входной ленты);
- (4) π — готовая к данному моменту часть выхода анализа-тора; ожидается, что полным выходом будет правый разбор входной цепочки для некоторой КС-грамматики.

Оператор языка Флойда—Эванса имеет вид

$$\langle \text{метка} \rangle: \alpha | a \rightarrow \beta | \langle \text{действие} \rangle * \langle \text{следующая метка} \rangle$$

причем метасимволов \rightarrow и $*$ может не быть.

Допустим, что анализатор находится в конфигурации

$$(L_1, \gamma\alpha, ax, \pi)$$

и оператор L_1 имеет вид

$$L_1: \alpha | a \rightarrow \beta | \text{выдача } s * L_2$$

¹⁾ Можно было бы добавить, что это обстоятельство не дает окончательного обобщения понятия алгоритма типа „перенос—свертка“. LR(k)-алгоритм тоже использует управляющее устройство с конечной памятью, а дополнительную информацию хранит в магазине. Самой общей моделью алгоритма этого типа следовало бы считать ДМП-преобразователь. Однако в разд. 3.4 мы видели, что ДМП-преобразователь вовсе не обязан проводить разбор входной цепочки, делая свертки в соответствии с грамматикой, для анализа которой он предназначен, как это делал LR(k)-алгоритм и алгоритмы из разд. 5.3 и 5.4.

Этот оператор говорит о том, что если наверху магазина расположена цепочка α и a — текущий входной символ, то надо заменить α на β , выдать цепочку s , сдвинуть входную головку на один символ вправо (на это указывает наличие символа $*$) и перейти к следующему оператору L_2 . Анализатор перейдет, таким образом, в конфигурацию $(L_2, \gamma\beta, x, \pi s)$. Возможно, что $a = e$. В этом случае текущий входной символ игнорируется, но головка сдвигается с него, если есть символ $*$.

Если оператор L_1 нельзя применить из-за того, что α не совпадает с верхней частью магазина или a не является текущим входным символом, то делается попытка применить оператор, непосредственно следующий в списке операторов за L_1 .

Обе метки у оператора тоже не обязательны (хотя для того, чтобы обозначать операторы в конфигурациях, предполагается, что они имеют имена). Если отсутствует символ \rightarrow , то магазин не меняется, и поэтому нет смысла указывать β . Если опущен символ $*$, то входная головка не сдвигается. Если опущена \langle следующая метка \rangle , то после данного оператора применяется следующий в списке.

Другие возможные действия — **допуск** и **ошибка**. Если место, предназначенное для действия, пусто, то не надо делать ничего, кроме распознавания входного символа и замены в магазине.

Первоначально анализатор находится в конфигурации $(L, \$, w\$, e)$, где w — анализируемая входная цепочка и L — выделенный оператор. Затем последовательно проверяются операторы, пока среди них не найдется применимый. После выполнения всех действий, предписываемых этим оператором, управление передается оператору, указанному меткой.

Анализатор продолжает работать до тех пор, пока не выполнится действие **ошибка** или **допуск**. Выход принимается во внимание только в случае допуска.

Мы покажем, как с помощью языка Флойда—Эванса описываются алгоритмы типа „перенос—свертка“, но обращаем внимание читателя на то, что этот язык можно использовать и для реализации исходящих алгоритмов разбора. Итак, предполагается, что можно писать $\alpha = \alpha_1 \alpha_2 \alpha_3$ и $\beta = \alpha_1 A \alpha_3$ или $\beta = \alpha_1 A \alpha_3 a$, если есть $*$. При этом $A \rightarrow \alpha_2$ — правило грамматики, для которой предпринимается разбор, а выход s — номер этого правила.

Язык Флойда—Эванса можно модифицировать так, чтобы действиями стали некоторые „семантические процедуры“. Тогда вместо того, чтобы выдавать разбор, анализатор будет выполнять синтаксически управляемую трансляцию, вычисляющую перевод нетерминала A в терминах переводов составных частей цепочки α_2 . Система такого рода описана Фельдманом [1966].

Пример 5.47. Запишем на языке Флойда—Эванса анализатор для грамматики G_6 с правилами

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

Символ $\#$ играет роль представителя любого символа. Предполагается, что с обеих сторон стрелки он обозначает один и тот же символ. Начальным оператором служит $L11$.

$$\begin{array}{lll} L0: & (\mid \# \rightarrow (\# & * L0 \\ L1: & a \mid \# \rightarrow F \# & \text{выдача } 6 * \\ L2: & T * F \# \mid \rightarrow T \# & \text{выдача } 3 \quad L4 \\ L3: & F \# \mid \rightarrow T \# & \text{выдача } 4 \\ L4: & T * \mid \# \rightarrow T * \# & * L0 \\ L5: & E + T \# \mid \rightarrow E \# & \text{выдача } 1 \quad L7 \\ L6: & T \# \mid \rightarrow E \# & \text{выдача } 2 \\ L7: & E + \mid \# \rightarrow E + \# & * L0 \\ L8: & (E) \mid \# \rightarrow F \# & \text{выдача } 5 * L2 \\ L9: & \$E\$ \mid \rightarrow & \text{допуск} \\ L10: & \$ \mid \rightarrow & \text{ошибка} \\ L11: & \mid \# \rightarrow \# & * L0 \end{array}$$

Для входной цепочки $(a+a)*a$ анализатор пройдет такую последовательность конфигураций:

$$\begin{aligned} [L11, \$, (a+a)*a\$, e] \vdash & [L0, \$(), a+a)*a\$, e] \\ \vdash & [L0, \$a, +a)*a\$, e] \\ \vdash & [L1, \$a, +a)*a\$, e] \\ \vdash & [L2, \$F+, a)*a\$, 6] \\ \vdash & [L3, \$F+, a)*a\$, 6] \\ \vdash & [L4, \$T+, a)*a\$, 64] \\ \vdash & [L5, \$T+, a)*a\$, 64] \\ \vdash & [L6, \$T+, a)*a\$, 64] \\ \vdash & [L7, \$E+, a)*a\$, 642] \\ \vdash & [L0, \$E+a,)*a\$, 642] \\ \vdash & [L1, \$E+a,)*a\$, 642] \\ \vdash & [L2, \$E+F,)*a\$, 6426] \\ \vdash & [L3, \$E+F,)*a\$, 6426] \\ \vdash & [L4, \$E+T,)*a\$, 64264] \\ \vdash & [L5, \$E+T,)*a\$, 64264] \\ \vdash & [L7, \$E,)*a\$, 642641] \end{aligned}$$

$$\begin{aligned} \vdash & [L8, \$E,)*a\$, 642641] \\ \vdash & [L2, \$F*, a\$, 6426415] \\ \vdash & [L3, \$F*, a\$, 6426415] \\ \vdash & [L4, \$T*, a\$, 64264154] \\ \vdash & [L0, \$T*a, \$, 64264154] \\ \vdash & [L1, \$T*a, \$, 64264154] \\ \vdash & [L2, \$T*F\$, e, 642641546] \\ \vdash & [L4, \$T\$, e, 6426415463] \\ \vdash & [L5, \$T\$, e, 6426415463] \\ \vdash & [L6, \$T\$, e, 6426415463] \\ \vdash & [L7, \$E\$, e, 64264154632] \\ \vdash & [L8, \$E\$, e, 64264154632] \\ \vdash & [L9, \$E\$, e, 64264154632] \\ \vdash & \text{допуск } \square \end{aligned}$$

Заметим, что анализатор Флойда—Эванса можно промоделировать на ДМП-преобразователе. Поэтому любой анализатор Флойда—Эванса распознает только детерминированный КС-язык. Но распознаваемый им язык может не совпадать с языком, определяемым грамматикой, для которой он должен строить разборы, поскольку передача управления операторами может привести к тому, что некоторые свертки не будут реализованы¹⁾.

Пример 5.48. Пусть G состоит из правил

$$\begin{array}{l} (1) \quad S \rightarrow aS \\ (2) \quad S \rightarrow bS \\ (3) \quad S \rightarrow a \end{array}$$

$L(G) = (a+b)^*a$. Приведем последовательность операторов, представляющую собой анализатор, который в соответствии с грамматикой G разбирает цепочки из языка b^*a , но не допускает других цепочек:

$$\begin{array}{lll} L0: & |\# \rightarrow \# & * \\ L1: & a \mid \rightarrow S & \text{выдача } 3 \quad L4 \\ L2: & b \mid \rightarrow & L0 \\ L3: & \$ \mid \rightarrow & \text{ошибка} \\ L4: & aS \mid \rightarrow S & \text{выдача } 1 \quad L4 \\ L5: & bS \mid \rightarrow S & \text{выдача } 2 \quad L4 \\ L6: & \$S \mid \$ \rightarrow \$S\$ \mid \text{допуск} & * \\ L7: & \mid \rightarrow & \text{ошибка} \end{array}$$

¹⁾ Обсуждаемая здесь ситуация аналогична той, что рассматривалась в дополнении к разд. 5.1: анализатор Флойда—Эванса M_G , построенный по грамматике G , может быть не адекватен G , т. е. $L(M_G) \neq L(G)$. — Прим. перев.

Для входной цепочки ba анализатор сделает такую последовательность шагов:

$$\begin{aligned} [L0, \$, ba\$, e] &\vdash [L1, \$b, a\$, e] \\ &\vdash [L2, \$b, a\$, e] \\ &\vdash [L0, \$b, a\$, e] \\ &\vdash [L1, \$ba, \$, e] \\ &\vdash [L4, \$bS, \$, 3] \\ &\vdash [L5, \$bS, \$, 3] \\ &\vdash [L4, \$S, \$, 32] \\ &\vdash [L5, \$S, \$, 32] \\ &\vdash [L6, \$S, \$, 32] \end{aligned}$$

Цепочка ba допускается после выполнения оператора $L6$. Последовательность шагов для цепочки aa такова:

$$\begin{aligned} [L0, \$, aa\$, e] &\vdash [L1, \$a, a\$, e] \\ &\vdash [L4, \$S, a\$, 3] \\ &\vdash [L5, \$S, a\$, 3] \\ &\vdash [L6, \$S, a\$, 3] \\ &\vdash [L7, \$S, a\$, 3] \end{aligned}$$

Оператор $L7$ объявляет об ошибке, хотя $aa \in L(G)$.

В примере 5.48 ничего мистического нет. Программа анализатора, написанная на языке Флойда—Эванса, не так тесно связана с грамматикой, для которой она производит разбор, как другие алгоритмы этой главы со своими грамматиками. \square

В гл. 7 мы дадим алгоритм, механически порождающий анализатор, написанный на языке Флойда—Эванса, для обратимой грамматики слабого предшествования.

5.4.5. Резюме

Диаграмма на рис. 5.22 демонстрирует иерархию грамматик, рассмотренных в этой главе. Все включения на этой диаграмме собственные. В качестве упражнений предлагаем доказать включения, которые мы не доказывали в этой главе. Включение класса LL-грамматик в класс LR-грамматик будет доказано в гл. 8.

Что касается классов языков, порождаемых грамматиками этих классов, то получены следующие результаты:

(1) Каждый из классов грамматик (а)–(з) порождает в точности класс детерминированных контекстно-свободных языков:

- | | |
|--------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| (а) LR,
(в) ОПК,
(д) ССП,
(ж) обратимые | (б) LR(1),
(г) (1,1)-ОПК,
(е) простые ССП,
(з) анализируемые по
Флойду—Эвансу.
(2,1)-предшествования, |
|--------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|

(2) LL-языки образуют собственный подкласс детерминированных КС-языков.

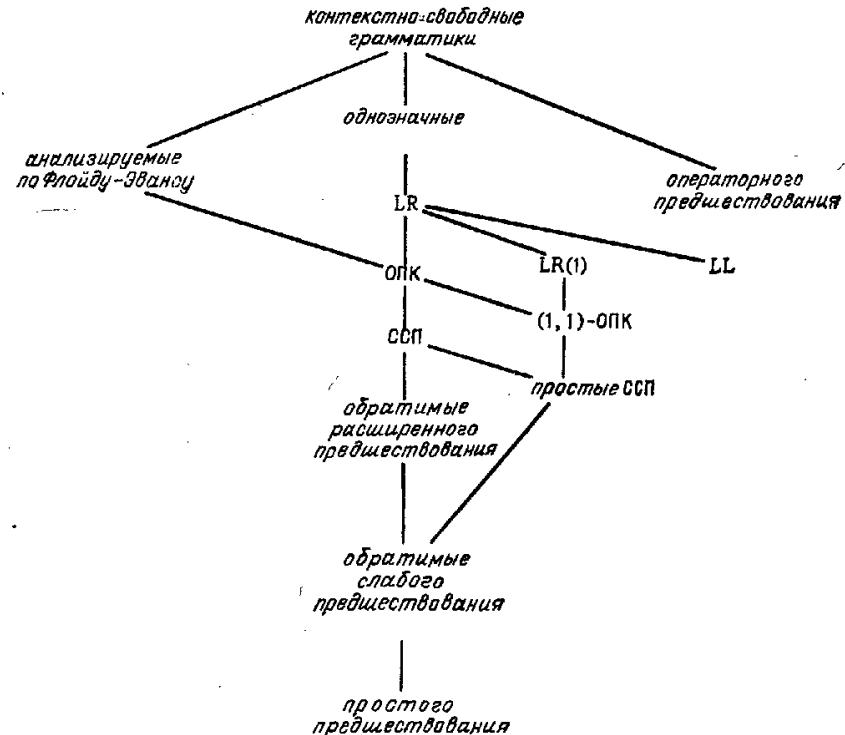


Рис. 5.22. Иерархия грамматик.

(3) Обратимые грамматики слабого предшествования порождают в точности класс языков простого предшествования, который

- (а) является собственным подклассом детерминированных КС-языков и
- (б) несравним с классом LL-языков.

(4) Класс языков, порождаемых грамматиками операторного предшествования, тот же, что и для обратимых грамматик этого типа. Он является собственным подклассом языков простого предшествования.

Многие из перечисленных результатов о классах языков можно будет найти в гл. 8.

Читатель может, конечно, спросить, какой из этих классов грамматик больше всего подходит для описания языков программирования и какой метод синтаксического анализа наилучший. Однозначного ответа на такой вопрос нет. Желание использовать простейший класс грамматик часто может потребовать каких-то манипуляций с данной грамматикой, необходимых для ее преобразования в грамматику из этого класса. При этом нередко грамматика становится неестественной и неудобной для использования ее в схеме синтаксически управляемой трансляции.

Очень привлекательны для практического применения LL(1)-грамматики. Для каждой такой грамматики можно построить компактный быстрый анализатор, производящий левый разбор, что имеет преимущества с точки зрения трансляции. Однако здесь есть и неудобства. LL(1)-грамматика для данного языка может оказаться неестественной и ее трудно построить. Кроме того, как будет показано в гл. 8, не каждый детерминированный КС-язык определяется LL-грамматикой, а тем более LL(1)-грамматикой.

Метод операторного предшествования применялся в нескольких компиляторах, он легко реализуется и работает очень эффективно. Грамматики простого предшествования тоже легко анализируются, но чтобы получить такую грамматику для данного языка, часто требуется добавить много целевых правил вида $A \rightarrow X$ для устранения конфликтов отношений предшествования. Кроме того, существует много детерминированных КС-языков, которые нельзя определить ни грамматиками простого предшествования, ни обратимыми грамматиками слабого предшествования.

Метод LR(1)-анализа, изложенный в этой главе, очень близок к описанному в оригинальной работе Кнута. Получающиеся при этом анализаторы могут быть очень большими. В гл. 7 будет описана техника построения LR(1)-анализаторов, которые по своим размерам и скорости работы могут для широкого спектра языков программирования конкурировать с анализаторами, работающими методом предшествования. Некоторые эмпирические данные приведены в статье Лалонда и др. [1971]. Так как LR(1)-грамматики образуют широкий класс грамматик, методы LR(1)-анализа представляются весьма привлекательными.

Наконец, подчеркнем, что в конкретных приложениях часто

можно улучшить реализацию любой техники разбора. В гл. 7 мы опишем методы, с помощью которых можно уменьшить объем и увеличить скорость анализаторов.

УПРАЖНЕНИЯ

5.4.1. Постройте алгоритм разбора типа „перенос—свертка“ для (1,0)-ОПК-грамматики G_i из примера 5.41.

5.4.2. Какие из следующих грамматик являются (1, 1)-ОПК-грамматиками?

- (а) $S \rightarrow aA \mid B$
 $A \rightarrow 0A1 \mid a$
 $B \rightarrow 0B1 \mid b$
- (б) $S \rightarrow aA \mid bB$
 $A \rightarrow 0A1 \mid 01$
 $B \rightarrow 0B1 \mid 01$
- (в) $E \rightarrow E + T \mid E - T \mid T$
 $T \rightarrow T * F \mid T / F \mid F$
 $F \rightarrow (E) \mid -E \mid a$

Определение. Приведенная КС-грамматика $G = (N, \Sigma, P, S)$ называется *граммикой (m, n) -ограниченного контекста (OK)*, если из условий

- (1) $\$^m S' \$^n \Rightarrow^* \alpha_1 A_1 \gamma_1 \Rightarrow \alpha_1 \beta_1 \gamma_1$ в дополненной грамматике,
 - (2) $\$^m S' \$^n \Rightarrow^* \alpha_2 A_2 \gamma_2 \Rightarrow \alpha_2 \beta_2 \gamma_2 = \alpha_3 \beta_1 \gamma_3$ в дополненной грамматике,
 - (3) последние m символов цепочек α_1 и α_3 , а также первые n символов цепочек γ_1 и γ_3 совпадают
- вытекает, что $\alpha_3 A_1 \gamma_3 = \alpha_2 A_2 \gamma_2$.

5.4.3. Покажите, что каждая (m, n) -OK-грамматика является (m, n) -ОПК-грамматикой.

5.4.4. Разработайте алгоритм разбора типа „перенос—свертка“ для OK-грамматик.

5.4.5. Дайте пример ОПК-грамматики, которая не является OK-грамматикой.

5.4.6. Докажите, что каждая обратимая грамматика расширенного предшествования является ОПК-грамматикой.

5.4.7. Покажите, что каждая ОПК-грамматика является грамматикой расширенного предшествования (не обязательно, разумеется, обратимой).

5.4.8. Для грамматик из упр. 5.4.2, которые являются (1, 1)-ОПК-грамматиками, постройте алгоритмы разбора типа „перенос—свертка“ и их реализацию в виде дерева решений.

5.4.9. Докажите необходимость условия из леммы 5.6.

5.4.10. Докажите теорему 5.22.

5.4.11. Какие из следующих грамматик являются простыми ССП-грамматиками?

(а) G_0
 $S \rightarrow A \mid B$
 $A \rightarrow 0A1 \mid 01$
 $B \rightarrow 2B1 \mid 1$

(б) $S \rightarrow A \mid B$
 $A \rightarrow 0A1 \mid 01$
 $B \rightarrow 0B1 \mid 1$

(г) $S \rightarrow A \mid B$
 $A \rightarrow 0A1 \mid 01$
 $B \rightarrow 01B1 \mid 01$

5.4.12. Докажите, что каждая обратимая грамматика слабого предшествования является простой ССП-грамматикой.

5.4.13. Является ли каждая $(m, n; m, n)$ -СПП-грамматика (m, n) -ОПК-грамматикой?

5.4.14. Докажите теорему 5.24.

5.4.15. Являются ли следующие грамматики грамматиками операторного предшествования?

(а) Грамматика из упр. 5.4.2(б).

(б) $S \rightarrow \text{если } B \text{ то } S \text{ иначе } S$

$S \rightarrow \text{если } B \text{ то } S$

$S \rightarrow s$

$B \rightarrow B \text{ или } b$

$B \rightarrow b$

(в) $S \rightarrow \text{если } B \text{ то } S_i \text{ иначе } S$

$S \rightarrow \text{если } B \text{ то } S$

$S_i \rightarrow \text{если } B \text{ то } S_i \text{ иначе } S_i$

$S \rightarrow s$

$S_i \rightarrow s$

$B \rightarrow B \text{ или } b$

$B \rightarrow b$

В пунктах (б) и (в) подразумевается, что **если**, **то**, **иначе**, **или**, **b**, **s** — терминальные символы.

5.4.16. Для грамматик из упр. 5.4.15 постройте оставные грамматики.

5.4.17. Постройте алгоритмы разбора $\mathcal{A} = (f, g)$ для грамматик из упр. 5.4.15, которые являются грамматиками операторного предшествования.

5.4.18. Докажите теорему 5.25.

5.4.19. Покажите, что для каждой грамматики операторного предшествования G соответствующая оставная грамматика G_s обратима.

*5.4.20. Покажите, что каждый язык операторного предшествования определяется грамматикой операторного предшествования без цепных правил.

*5.4.21. Покажите, что каждый язык операторного предшествования определяется обратимой грамматикой операторного предшествования.

5.4.22. Для грамматик из упр. 5.4.2 напишите анализаторы на языке Флойда — Эванса.

5.4.23. Покажите, что для каждой ОПК-грамматики существует анализатор, написанный на языке Флойда — Эванса.

5.4.24. Покажите, что для каждой $LL(k)$ -грамматики существует анализатор на языке Флойда — Эванса (порождающий левые разборы).

**5.4.25. Докажите неразрешимость следующих проблем: является ли КС-грамматика

(а) ОПК-грамматикой?

(б) ОК-грамматикой?

(в) ССП-грамматикой?

5.4.26. Докажите, что $G = (N, \Sigma, P, S)$ — простая ССП-грамматика тогда и только тогда, когда она грамматика слабого предшествования и если $A \rightarrow \alpha$ и $B \rightarrow \alpha$ содержатся в P , причем $A \neq B$, то $I(A) \cap I(B) = \emptyset$.

*5.4.27. Допустим, что мы ослабили определение операторной грамматики, не требуя, чтобы она была приведенной и не содержала e -правил. Покажите, что L — язык операторного предшествования в смысле нового определения тогда и только тогда, когда $L - \{e\}$ — язык операторного предшествования в смысле старого определения.

5.4.28. Расширьте алгоритм Домёлки, упоминаемый в упражнениях разд. 4.1, так, чтобы его можно было использовать для анализа ОПК-грамматик.

Определение. Можно обобщить идею операторного предшествования так, чтобы в разборе участвовали не только все терминалы, но и некоторые нетерминалы. Пусть $G = (N, \Sigma, P, S)$ — приведенная КС-грамматика без e -правил и T — подмножество множества $N \cup \Sigma$, причем $\Sigma \subseteq T$. Пусть $V = N \cup \Sigma$. Грамматику G назовем **T -канонической**, если

(1) для каждой правой части правила, скажем $\alpha X\beta$, из $X \notin T$ следует $Y \in \Sigma$;

(2) из $A \in T$ и $A \Rightarrow^* \alpha$ следует, что α содержит символ из T .

Таким образом, Σ -каноническая грамматика — это то же что операторная грамматика. Для T -канонической грамматики G будем называть множеством ее знаменательных символов.

Зададим для T -канонической грамматики G отношения T -канонического предшествования на $T \cup \{\$\}$:

(1) Если в P есть правило $A \rightarrow \alpha X\beta Y\gamma$, X и Y содержатся в T и β — либо e , либо содержитя в $V - T$, то $X \sqsubseteq Y$.

(2) Если $A \rightarrow \alpha X\beta \in P$ и $B \Rightarrow^* \gamma Y\delta$, где X и Y содержатся в T и γ — либо e , либо содержитя в $V - T$, то $X \lhd Y$.

(3) Пусть $A \rightarrow \alpha_1 B \alpha_2 Z \alpha_3 \in P$, где α_2 — либо e , либо символ из $V - T$. Допустим, что $Z \Rightarrow^* \beta_1 a \beta_2$, где β_1 — либо e , либо содержитя в $V - T$ и $a \in \Sigma$. (Заметим, что по определению T -канонической грамматики этот вывод должен быть тривиальным, т. е. состоять из нуля шагов, если $\alpha_2 \neq e$.) Допустим также, что существует вывод

$$\begin{aligned} B \Rightarrow \gamma_1 C_1 \delta_1 &\Rightarrow \gamma_1 \gamma_2 C_2 \delta_2 \delta_1 \Rightarrow \dots \Rightarrow \gamma_1 \dots \gamma_{k-1} C_{k-1} \delta_{k-1} \dots \delta_1 \\ &\Rightarrow \gamma_1 \dots \gamma_k X \delta_k \dots \delta_1 \end{aligned}$$

где на каждом шаге заменяется символ C_i , все δ_j принадлежат $\{e\} \cup V - T$ и $X \in T$. Тогда $X \triangleright a$.

(4) Если $S \Rightarrow^* \alpha X\beta$ и $\alpha \in \{e\} \cup V - T$, то $\$ \lhd X$. Если $\beta \in \{e\} \cup V - T$, то $X \triangleright \$$.

Заметим, что при $T = \Sigma$ получается определение отношений операторного предшествования, а при $T = V$ — отношений Вирта—Бебера.

Пример 5.49. Рассмотрим грамматику G_0 с множеством знаменательных символов $\Delta = \{F, a, (,), +, *\}$. Найдем, что (\sqsubseteq) , так как (E) — правая часть правила и E не знаменательный символ. Далее, $+ \lhd *$, так как существуют правая часть $E + T$ и вывод $T \Rightarrow^* T * F$, и T не знаменательный символ. Кроме того $+ \triangleright +$, так как существуют правая часть $E + T$ и вывод $E \Rightarrow E + T$, и T не знаменательный символ. Отношения Δ -канонического предшествования для G_0 приведены на рис. 5.23. \square

5.4.29. Найдите все множества знаменательных символов для грамматики G_0 .

5.4.30. Покажите, что Σ — множество знаменательных символов для $G = (N, \Sigma, P, S)$ тогда и только тогда, когда G — грамматика операторного предшествования. Покажите, что $N \cup \Sigma$ —

множество знаменательных символов тогда и только тогда, когда G — грамматика предшествования.

Определение. Пусть $G = (N, \Sigma, P, S)$ — T -каноническая грамматика. T -остовная грамматика для G получается заменой в правилах из P всех вхождений символов из $V - T$ новым символом S_0 и устранением правила $S_0 \rightarrow S_0$, если оно при этом появилось.

	a	$+$	$*$	$()$	F	$\$$
a	\sqsubseteq	\triangleright	\triangleright		\triangleright	\triangleright
$+$	\lhd	\triangleright	\lhd	\lhd	\triangleright	\lhd
$*$	\lhd		\lhd		\doteq	
$()$	\lhd	\lhd	\lhd	\doteq	\lhd	
$)$		\triangleright	\triangleright	\triangleright		\triangleright
F		\triangleright	\triangleright	\triangleright		\triangleright
$\$$	\lhd	\lhd	\lhd	\lhd	\lhd	

Рис. 5.23. Отношения Δ -канонического предшествования.

Пример 5.50. Пусть Δ то же, что и в примере 5.49. Тогда Δ -остовной грамматикой для G_0 будет

$$\begin{aligned} S_0 &\rightarrow S_0 + S_0 | S_0 * F | F \\ F &\rightarrow (S_0) | a \end{aligned} \quad \square$$

5.4.31. Разработайте алгоритм разбора типа „перенос—свертка“ для грамматик T -канонического предшествования, имеющих обратимые T -остовные грамматики. При этом, разумеется, должны вырабатываться разборы, соответствующие остановной грамматике.

Проблема для исследования

5.4.32. Разработайте преобразования, с помощью которых можно получать ОПК-грамматики, грамматики простого или операторного предшествования.

Упражнения на программирование

5.4.33. Напишите программу, проверяющую, является ли данная грамматика грамматикой операторного предшествования.

5.4.34. Напишите программу, строящую соответствующий анализатор для грамматики операторного предшествования.

5.4.35. Найдите грамматику операторного предшествования для одного из языков, приведенных в приложении, и затем постройте для этой грамматики соответствующий анализатор.

5.4.36. Напишите программу, строящую соответствующий анализатор для (1, 1)-ОПК-грамматики.

5.4.37. Напишите программу, строящую соответствующий анализатор для простой ССП-грамматики.

5.4.38. Определите язык программирования, в основе которого лежал бы язык Флойда — Эванса. Постройте для него компилятор.

Замечания по литературе

Различные методы разбора, ориентированные на предшествование, применялись в самых первых компиляторах. Формализация понятия операторного предшествования принадлежит Флойду [1963]. Методы, использующие ограниченный контекст и ограниченный правый контекст, были определены тоже в начале 1960-х годов. Большинство ранних результатов, касающихся разбора методом ограниченного контекста и его вариантов, содержится в работах Эйкеля и др. [1963], Флойда [1963, 1964а, 1964б], Грэхема [1964], Айронса [1964] и Поля [1962].

Приведение в этом разделе определение ОПК-грамматики эквивалентно тому, которое дал Флойд [1964а]. Локс [1970] разработал алгоритм построения анализаторов для некоторых классов ОПК-грамматик. Вайз [1971] распространил алгоритм Домёлки на ОПК-грамматики.

Смешанная стратегия предшествования была введена Маккиманом и применена им с соавторами [1970] в качестве основы системы построения компиляторов XPL.

Язык Флойда — Эванса — это предложение Эвансом [1964] модификация языка, первоначально введенного Флойдом [1961]. Фельдман [1966] использовал его в качестве основы системы построения компиляторов, названной Языком Формальной Семантики (FSL). При этом среди действий, входящих в операторы языка, могут быть общие семантические процедуры.

T-каноническое предшествование было введено Грэмом и Харрисоном [1969]. Пример 5.47 взят из книги Хопгуда [1969]¹⁾.

¹⁾ Трахтенгерд и Шумей [1971] обобщили идею предшествования на не КС-грамматики. Трубчининов [1976] обобщил понятие *T*-канонического предшествования. Интересный класс „строго детерминированных“ грамматик ввели Харрисон и Хавел [1973, 1974]. Эти грамматики порождают класс LR(0)-языков, но алгоритм разбора для них не восходящий и не нисходящий, а обладает чертами обоих методов. — Прим. перев.

6

АЛГОРИТМЫ РАЗБОРА С ОГРАНИЧЕННЫМИ ВОЗВРАТАМИ

В настоящей главе мы изложим несколько алгоритмов синтаксического анализа, которые, подобно общим нисходящим и восходящим алгоритмам разд. 4.1, могут содержать возвраты. Однако на возвраты, встречающиеся в этих алгоритмах, налагаются ограничения. Поэтому они оказываются экономнее алгоритмов гл. 4. Тем не менее в случаях, когда достаточно безвозвратного детерминированного алгоритма, ими лучше не пользоваться.

В разд. 6.1 будут рассмотрены два языка высокого уровня, на которых можно записывать нисходящие алгоритмы разбора с ограниченными возвратами. Эти языки, называемые ЯНРОВ и ОЯНРОВ, позволяют задавать распознаватели для всех детерминированных КС-языков с концевыми маркерами и благодаря возможности ограниченных возвратов даже некоторые не КС-языки, но (вероятно) не все КС-языки. Затем дадим метод построения для широкого класса КС-грамматик восходящих алгоритмов разбора, ориентированных на предшествование и допускающих ограниченные возвраты.

6.1. НИСХОДЯЩИЙ РАЗБОР С ОГРАНИЧЕННЫМИ ВОЗВРАТАМИ

В данном разделе определяются два формализма, предназначенные для описания алгоритмов с ограниченными возвратами, которые строят дерево разбора сверху вниз, перебирая все альтернативы каждого нетерминала до тех пор, пока не найдется альтернатива, из которой выводится префикс необработанной части входной цепочки. Как только такая альтернатива обнаружена, другие альтернативы уже не проверяются. Конечно, полученный таким способом префикс может оказаться « ошибочным », и тогда, поскольку возврата не будет, попытка разбора окончится неудачей. К счастью, в практических ситуациях это обстоятельство редко создает серьезные проблемы, если альтер-

нативы упорядочены так, что первыми испытываются более длинные из них.

Мы рассмотрим взаимоотношения между этими формализмами, их реализацию, а также кратко обсудим их как механизмы, определяющие классы языков. Будет показано, что классы определяемых ими языков отличаются от класса КС-языков.

6.1.1. Язык нисходящего разбора с ограниченными возвратами

Возьмем общий нисходящий алгоритм разбора из разд. 4.1. Допустим, что мы решили вывести некоторую цепочку из нетерминала A и что $\alpha_1, \alpha_2, \dots, \alpha_n$ — его альтернативы. Предположим, что в каком-то выводе входной цепочки из A выводится некоторый префикс x еще не обработанной части этой цепочки и этот вывод начинается шагом $A \Rightarrow_i \alpha_m$ ($1 \leq m \leq n$), причем вывода, который начинался бы шагом $A \Rightarrow_j \alpha_j$ для $j < m$, не существует.

Нисходящий алгоритм гл. 4 испытывает альтернативы $\alpha_1, \alpha_2, \dots, \alpha_n$ по порядку. После неудачи, связанной с альтернативой α_j для $j < m$, восстанавливается положение входного указателя и делается новая попытка, уже для α_{j+1} . Эта новая попытка предпринимается независимо от того, выводится из α_j терминальная цепочка, являющаяся префиксом необработанной части входной цепочки, или нет.

Изложим технику разбора, в которой нетерминалы трактуются как процедуры, сообщающие о том, обнаружена или нет подходящая цепочка. Чтобы проиллюстрировать этот подход, допустим, что $a_1 \dots a_n$ — входная цепочка и уже построен частичный левый разбор, для которого первые $i - 1$ символов порождаемой им цепочки совпали с соответствующими символами входной цепочки. Если далее нужно «развернуть» нетерминал A , то его можно «вызвать» как процедуру с входной позицией i в качестве аргумента. Если из A выводится терминальная цепочка, являющаяся префиксом цепочки $a_1 a_2 \dots a_n$, то говорят, что для входной позиции i нетерминал A успешен, а в противном случае, что он неудачен для позиции i .

Эти процедуры могут вызывать друг друга рекурсивно. Если нетерминал A вызван подобным образом, то он сам может вызывать нетерминалы, содержащиеся в его первой альтернативе α_1 . Если какой-то из этих вызовов закончится неудачно, то входной указатель возвращается туда, где он находился, когда впервые был вызван A , после этого A вызывает альтернативу α_2 и т. д. Если вызов α_j приводит к совпадению с символами $a_1 a_2 \dots a_k$, то A возвращается к вызвавшей его процедуре и передвигает входной указатель вперед на позицию $k + 1$.

Различие между таким алгоритмом и алгоритмом 4.1 состоит в том, что если последний потерпит неудачу при попытке найти полный разбор, в котором из α_j выводится $a_1 \dots a_k$, то он вернется назад, станет испытывать выводы, начинаяющиеся правилами $A \rightarrow \alpha_{j+1}$, $A \rightarrow \alpha_{j+2}$ и т. д., и при этом, возможно, получит другой префикс цепочки $a_1 \dots a_n$, выводимый из A . Наш новый алгоритм так не делает. Если уж он обнаружил, что из α_j выводится префикс входной цепочки и что полученный после этого вывод неудачен, т. е. не дает цепочки, совпадающей с входной цепочкой, то он возвращается к процедуре, вызвавшей A , и сообщает о неудаче. Алгоритм будет вести себя так, как будто из A не выводится никакой префикс цепочки $a_1 \dots a_n$. Таким образом, этот алгоритм может не обнаружить некоторые разборы и даже может распознать не тот язык, который определяет лежащая в его основе КС-грамматика¹⁾. Мы же будем, следовательно, связывать такой алгоритм с конкретной КС-грамматикой, а будем трактовать его сам по себе как формализм, предназначенный для определения и синтаксического анализа языка.

Возьмем конкретный пример. Если правила

$$\begin{aligned} S &\rightarrow Ac \\ A &\rightarrow a \mid ab \end{aligned}$$

рассматриваются в указанном здесь порядке, то алгоритм с ограниченными возвратами не распознает цепочку abc . Нетерминал S , вызванный для входной позиции 1, вызовет A для входной позиции 1. Применяя первую альтернативу, A сообщит об успехе и передвинет входной указатель на позицию 2. Однако символ c не совпадает со вторым входным символом, и потому S сообщает о своей неудаче на позиции 1. Так как нетерминал A сообщил об успехе своего первого вызова, он уже не будет вызываться для проверки второй альтернативы. Заметим, что этой трудности можно избежать, записав A -альтернативы в обратном порядке

$$A \rightarrow ab \mid a$$

Теперь изложим язык нисходящего разбора с ограниченными возвратами (сокращенно ЯНРОВ), который можно использовать для описания процедур разбора указанного типа. Оператором (или правилом) этого языка называется цепочка одного из видов

$$A \rightarrow BC/D$$

или

$$A \rightarrow a$$

¹⁾ Здесь снова возникает проблема адекватности распознавателя M_G соответствующей грамматике G , упоминавшейся в дополнении к разд. 5.1 и в разд. 5.4.4. — Прим. перев.

где A, B, C и D —нетерминалы, а a —терминал, пустая цепочка или специальный символ f (обозначающий неудачу).

Определение. ЯНРОВ-программой P называется четверка (N, Σ, R, S) , где

(1) N и Σ —конечные непересекающиеся множества нетерминалов и терминалов,

(2) R —последовательность ЯНРОВ-операторов, содержащая для каждого $A \in N$ не более одного оператора с левой частью A (слева от стрелки),

(3) $S \in N$ —начальный символ.

ЯНРОВ-программу можно связать с грамматикой в специальной нормальной форме. Оператор вида $A \rightarrow BC/D$ соответствует двум правилам $A \rightarrow BC$ и $A \rightarrow D$, из которых первое всегда испытывается первым. Оператор вида $A \rightarrow a$ соответствует правилу $A \rightarrow a$, когда $a \in \Sigma$ или $a = e$. Если $a = f$, то нетерминал A имеет специальный смысл, который мы разъясним позже.

ЯНРОВ-программу можно описать иначе как множество процедур (нетерминалов), вызываемых рекурсивно для некоторых входных цепочек в качестве аргументов. Выходом, или результатом, вызова нетерминала будет либо **неудача** (ни один префикс входной цепочки не подходит данному нетерминалу), либо **успех** (который префикс входной цепочки подходит ему).

Результатом вызова оператора вида $A \rightarrow BC/D$ для входной цепочки w будет следующая последовательность вызовов процедур:

(1) Сначала A вызывает B для входной цепочки w . Если $w = xx'$ и цепочка x подходит нетерминалу B , то B сообщает об **успехе**, т. е. результатом вызова B будет **успех**. После этого A вызывает C для цепочки x' .

(a) Если $x' = yz$ и y подходит C , то C сообщает об **успехе**. Тогда A тоже сообщает об **успехе** и о том, что ему подходит префикс xy цепочки w .

(b) Если никакой префикс цепочки x' не подходит C , то C сообщает о **неудаче**. Тогда A вызывает D для цепочки w . Заметим, что в этом случае успех вызова B аннулируется.

(2) Если A вызывает B для цепочки w и оказывается, что никакой префикс цепочки w не подходит B , то B сообщает о **неудаче**. Тогда A вызывает D для цепочки w .

(3) Если D был вызван для $w = uv$ и цепочка u подходит D , то D сообщает об **успехе**. Тогда A тоже сообщает об **успехе** и о том, что ему подходит префикс u цепочки w .

(4) Если D был вызван для цепочки w и никакой префикс цепочки w ему не подходит, то D сообщает о **неудаче**. Тогда A тоже сообщает о **неудаче**.

Заметим, что D вызывается, когда не оба вызова B и C оказались успешными. В дальнейшем мы исследуем другую систему разбора, в которой D вызывается только тогда, когда вызов B оканчивается неудачей. Заметим, что если B и C привели к успеху, то альтернатива D не вызывается. Эта особенность отличает ЯНРОВ-программу от общего алгоритма нисходящего разбора из гл. 4.

С операторами вида $A \rightarrow a$, $A = e$ и $A = f$ обращаются так:

(1) Если есть правило $A \rightarrow a$, в котором $a \in \Sigma$, и A вызывается для цепочки, начинающейся символом a , то цепочка a подходит A , и A сообщает об **успехе**. В противном случае A сообщает о **неудаче**.

(2) Если есть правило $A \rightarrow e$, то вызов A всегда успешен и пустая цепочка всегда подходит A .

(3) Если есть правило $A \rightarrow f$, то вызов A всегда оканчивается неудачей.

Определим формально, как нетерминал «действует на входную цепочку».

Определение. Пусть $P = (\Sigma, T, R, S)$ —ЯНРОВ-программа. Зададим отношения \Rightarrow^P между нетерминалами и парами вида $(x \uparrow y, r)$, где x и y —цепочки из Σ^* , а r —либо s (обозначает **успех**), либо f (обозначает **неудачу**). Метасимвол \uparrow используется для указания позиции текущего входного символа. Индекс P будем опускать всюду, где можно.

(1) Если $A \rightarrow e \in R$, то $A \Rightarrow^P (\uparrow w, s)$ для всех $w \in \Sigma^*$.

(2) Если $A \rightarrow f \in R$, то $A \Rightarrow^P (\uparrow w, f)$ для всех $w \in \Sigma^*$.

(3) Если $A \rightarrow a \in R$ и $a \in \Sigma$, то

- (a) $A \Rightarrow^P (a \uparrow x, s)$ для всех $x \in \Sigma^*$;
- (b) $A \Rightarrow^P (\uparrow y, f)$ для всех $y \in \Sigma^*$ (включая e), не начинающихся с a .

(4) Пусть $A \rightarrow BC/D \in R$. Тогда

(a) $A \Rightarrow^{m+n+1} (xy \uparrow z, s)$, если

(i) $B \Rightarrow^m (x \uparrow y, s)$,

(ii) $C \Rightarrow^n (y \uparrow z, s)$;

(b) $A \Rightarrow^i (u \uparrow v, s)$ для $i = m + n + 1$, если

(i) $B \Rightarrow^m (x \uparrow y, s)$,

(ii) $C \Rightarrow^n (\uparrow y, f)$,

(iii) $D \Rightarrow^p (u \uparrow v, s)$, где $uv = xy$;

(c) $A \Rightarrow^i (\uparrow xy, f)$ для $i = m + n + p + 1$, если

(i) $B \Rightarrow^m (x \uparrow y, s)$,

(ii) $C \Rightarrow^n (\uparrow y, f)$,

(iii) $D \Rightarrow^p (\uparrow xy, f)$;

- (г) $A \Rightarrow^{m+n+1} (x \uparrow y, s)$, если
 - (i) $B \Rightarrow^m (\uparrow xy, f)$,
 - (ii) $D \Rightarrow^n (x \uparrow y, s)$;
- (д) $A \Rightarrow^{m+n+1} (\uparrow x, f)$, если
 - (i) $B \Rightarrow^m (\uparrow x, f)$,
 - (ii) $D \Rightarrow^n (\uparrow x, f)$.

(5) Отношение \Rightarrow^n выполняется только в случаях, оговоренных в (1)–(4).

Пункт (4а) учитывает тот случай, когда B и C оба приводят к успеху. В (4б) и (4в) B успешен, но C неудачен. В последних четырех случаях вызывается D , и это оканчивается либо успехом, либо неудачей. Заметим, что число у стрелки указывает количество «вызовов», сделанных до того, как получен результат. Еще отметим, что если $A \Rightarrow^n (x \uparrow y, f)$, то $x = e$, т. е. в случае неудачного вызова входной указатель переставляется туда, где он находился перед вызовом.

Положим $A \Rightarrow^f (x \uparrow y, r)$ тогда и только тогда, когда $A \Rightarrow^n (x \uparrow y, r)$ для некоторого $n \geq 1$. Языком, определяемым программой P , назовем множество $L(P) = \{w \mid S \Rightarrow^f (w \uparrow, s) \text{ и } w \in \Sigma^*\}$.

Пример 6.1. Пусть $P = (\{S, A, B, C\}, \{a, b\}, R, S)$ — ЯНРОВ-программа, где R состоит из операторов

$$\begin{aligned} S &\rightarrow AB/C \\ A &\rightarrow a \\ B &\rightarrow CB/A \\ C &\rightarrow b \end{aligned}$$

Исследуем поведение программы P на входной цепочке aba , используя введенные выше обозначения. Вначале, благодаря правилу $S \rightarrow AB/C$, S вызывает A для цепочки aba . A распознает первый входной символ и сообщает об успехе. Учитывая (3) предыдущего определения, можно писать $A \Rightarrow^1 (a \uparrow ba, s)$. Далее S вызывает B для цепочки ba . Так как для B есть правило $B \rightarrow CB/A$, нужно заняться поведением C на цепочке ba . Оказывается, что b подходит C , так что C сообщает об успехе. В соответствии с (3) пишем $C \Rightarrow^1 (b \uparrow a, s)$.

Затем B рекурсивно вызывает себя для цепочки a . Однако C терпит неудачу на a , и потому $C \Rightarrow^1 (\uparrow a, f)$. Тогда B вызывает A для цепочки a . Так как a подходит A , то $A \Rightarrow^1 (a \uparrow, s)$. Так как вызов A успешен, то второй вызов B тоже успешен. В соответствии с (4г) пишем $B \Rightarrow^g (a \uparrow, s)$.

Возвращаясь к первому вызову B для цепочки ba , замечаем, что, так как вызовы C и B успешны, этот вызов B тоже успешен, и в соответствии с (4а) пишем $B \Rightarrow^g (ba \uparrow, s)$.

Возвращаясь теперь к вызову S , видим, что оба вызова A и B успешны. Таким образом, цепочка aba подходит S , и S сообщает об успехе. Учитывая (4а), пишем $S \Rightarrow^? (aba \uparrow, s)$. Поэтому $aba \in L(P)$.

Нетрудно показать, что $L(P) = ab^*a + b$. \square

Каждая ЯНРОВ-программа обладает одним важным свойством, а именно: ее результат для данного входа однозначен. Это доказывает следующая лемма.

Лемма 6.1. Пусть $P = (N, \Sigma, R, S)$ — ЯНРОВ-программа, для которой $A \Rightarrow^{n_1} (x_1 \uparrow y_1, r_1)$ и $A \Rightarrow^{n_2} (x_2 \uparrow y_2, r_2)$, где $A \in N$, $x_1y_1 = x_2y_2 = w \in \Sigma^*$. Тогда $x_1 = x_2$, $y_1 = y_2$ и $r_1 = r_2$.

Доказательство. Доказательство проводится простой индукцией по меньшему из чисел n_1 и n_2 . Без потери общности будем считать, что $n_1 \leq n_2$.

Базис. $n_1 = 1$. Применяется одно из правил $A \rightarrow a$, $A \rightarrow e$ или $A \rightarrow f$. Отсюда сразу получаем нужное утверждение.

Шаг индукции. Допустим, что утверждение верно для $n < n_1$, причем $n_1 > 1$. Пусть для A есть правило $A \rightarrow BC/D$. Допустим, что $A \Rightarrow^{n_i} (x_i \uparrow y_i, r_i)$ для $i = 1$ и $i = 2$ получено в силу (4) из $B \Rightarrow^{m_i} (u_i \uparrow v_i, t_i)$ и (возможно) $C \Rightarrow^{k_i} (u'_i \uparrow v'_i, t'_i)$ и/или $D \Rightarrow^{l_i} (u''_i \uparrow v''_i, t''_i)$. Тогда $m_i < n_1$, так что, применяя предположение индукции, заключаем, что $u_i = u_2$, $v_i = v_2$ и $t_i = t_2$. Рассмотрим отдельно два случая, зависящие от значения t_i .

Случай 1: $t_i = t_2 = f$. Так как $t_i < n_1$, то $u''_i = u''_2$, $v''_i = v''_2$ и $t''_i = t''_2$. Поскольку в данном случае $x_i = u''_i$, $y_i = v''_i$ и $r_i = t_i$ для $i = 1$ и $i = 2$, получаем нужный результат.

Случай 2: $t_i = t_2 = s$. Тогда $u'_i v'_i = v_i$ для $i = 1$ и $i = 2$. Так как $k_i < n_1$, то $u'_i = u'_2$, $v'_i = v'_2$ и $t'_i = t'_2$. Если $t'_i = s$, то $x_i = u'_i$, $y_i = v'_i$ и $r_i = s$ для $i = 1$ и $i = 2$; нужное заключение, таким образом, получено. При $t'_i = f$ рассуждаем для u'_i и v'_i также, как в случае 1. \square

Отметим, что ЯНРОВ-программа не обязана давать результат для каждой входной цепочки. Например, каждая программа, содержащая правило $S \rightarrow SS/S$, где S — начальный символ, не распознает ни одной цепочки (точнее, для нее отношение \Rightarrow^+ пусто).

Представление ЯНРОВ-программ, которым мы пользовались до сих пор, выбрано для удобства изложения. На практике желательно иметь правила более общего вида. Для этого введем **расширенные ЯНРОВ-правила** и определим их смысл в терминах прежних правил:

(1) Правило $A \rightarrow BC$ заменяет два правила $A \rightarrow BC/D$ и $D \rightarrow f$, где D — и новый символ.

(2) Правило $A \rightarrow B/C$ заменяет правила $A \rightarrow BD/C$ и $D \rightarrow e$.

(3) Правило $A \rightarrow B$ заменяет правила $A \rightarrow BC$ и $C \rightarrow e$.

(4) Правило $A \rightarrow A_1A_2 \dots A_n$ для $n > 2$ заменяет множество правил $A \rightarrow A_1B_1, B_1 \rightarrow A_2B_2, \dots, B_{n-3} \rightarrow A_{n-2}B_{n-2}, B_{n-2} \rightarrow A_{n-1}A_n$.

(5) Правило $A \rightarrow \alpha_1/\alpha_2/\dots/\alpha_n$, где α_i — цепочка нетерминалов, заменяет множество правил $A \rightarrow B_1/C_1, C_1 \rightarrow B_2/C_2, \dots, C_{n-3} \rightarrow B_{n-2}/C_{n-2}, C_{n-2} \rightarrow B_{n-1}/B_n$ и $B_1 \rightarrow \alpha_1, B_2 \rightarrow \alpha_2, \dots, B_n \rightarrow \alpha_n$. Если $n = 2$, это множество состоит из правил $A \rightarrow B_1/B_2, B_1 \rightarrow \alpha_1$ и $B_2 \rightarrow \alpha_2$. Для $1 \leq i \leq n$ в случае $|\alpha_i| = 1$ можно положить $B_i = \alpha_i$ и исключить правило $B_i \rightarrow \alpha_i$.

(6) Правило $A \rightarrow \alpha_1/\alpha_2/\dots/\alpha_n$, где α_i — цепочка нетерминалов и терминалов, заменяет множество правил $A \rightarrow \alpha'_1/\alpha'_2/\dots/\alpha'_n$ и $X_a \rightarrow a$ для каждого терминала a , где α'_i — цепочка α_i , в которой каждый терминал a заменен нетерминалом X_a .

Начиная с этого момента мы будем допускать включение в ЯНРОВ-программы расширенных правил указанного типа. Приведенные выше определения позволяют механически строить эквивалентную ЯНРОВ-программу, удовлетворяющую первоначальному определению.

Эти расширенные правила имеют естественный смысл. Например, если для A есть правило $A \rightarrow Y_1Y_2 \dots Y_n$, то вызов A успешен тогда и только тогда, когда вызов Y_1 успешен в той входной позиции, где был сделан вызов A , вызов Y_2 успешен там, где закончился вызов Y_1 , вызов Y_3 успешен там, где закончился вызов Y_2 , и т. д.

Аналогично, если для A есть правило $A \rightarrow \alpha_1/\alpha_2/\dots/\alpha_n$, то вызов A успешен тогда и только тогда, когда α_1 достигает успеха там, где вызван A ; либо когда α_1 приводит к неудаче, но α_2 достигает успеха там, где вызван A , и т. д.

Пример 6.2. Рассмотрим расширенную ЯНРОВ-программу $P = (\{E, F, T\}, \{a, (), +, *\}, R, S)$, где R состоит из правил

$$\begin{aligned} E &\rightarrow T+E/T \\ T &\rightarrow F*T/F \\ F &\rightarrow (E)/a \end{aligned}$$

Читателю предлагаем убедиться в том, что $L(P) = L(G_0)$, где G_0 — наша стандартная грамматика для арифметических выражений.

Чтобы привести P к стандартной форме, сначала, применяя (6), введем новые нетерминалы $X_a, X_(), X_+, X_*$. Правила станут

таким:

$$\begin{aligned} E &\rightarrow TX_+E/T \\ T &\rightarrow FX_*T/F \\ F &\rightarrow X_aEX_a/X_a \\ X_a &\rightarrow a \\ X_() &\rightarrow () \\ X_+ &\rightarrow + \\ X_* &\rightarrow * \end{aligned}$$

Согласно (5), первое правило заменяется правилами $E \rightarrow B_1/T$ и $B_1 \rightarrow TX_+E$. Учитывая (4), заменяем $B_1 \rightarrow TX_+E$ правилами $B_1 \rightarrow TB_2$ и $B_2 \rightarrow X_+E$. Затем заменяем их на $B_1 \rightarrow TB_2/D, B_2 \rightarrow X_aE/D$ и $D \rightarrow f$. Правило $E \rightarrow B_1/T$ заменяется на $E \rightarrow B_1C/T$ и $C \rightarrow e$. В результате получим множество правил

$$\begin{array}{lll} X_a \rightarrow a & D \rightarrow f & B_4 \rightarrow X_aT/D \\ X_() \rightarrow () & E \rightarrow B_1C/T & F \rightarrow B_5C/X_a \\ X_+ \rightarrow + & B_1 \rightarrow TB_2/D & B_5 \rightarrow X_aB_6/D \\ X_* \rightarrow * & B_2 \rightarrow X_aE/D & B_6 \rightarrow EX_a/D \\ C \rightarrow e & T \rightarrow B_3C/F & \\ & B_3 \rightarrow FB_4/D & \end{array}$$

Для простоты мы отождествили с нетерминалом C каждый нетерминал X , для которого получается правило $X \rightarrow e$, и с нетерминалом D каждый Y , для которого $Y \rightarrow f$. \square

Всякий раз, когда ЯНРОВ-программа распознает цепочку w , можно сверху вниз построить „дерево разбора“ этой цепочки, прослеживая последовательность ЯНРОВ-операторов, выполняемых при распознавании w . Внутренние вершины этого дерева соответствуют нетерминалам, вызываемым в ходе выполнения программы и сообщающим об успехе.

Алгоритм 6.1. Дерево разбора, соответствующее выполнению ЯНРОВ-программы.

Вход. ЯНРОВ-программа $P = (N, \Sigma, P, S)$ и такая цепочка $w \in \Sigma^*$, что $S \Rightarrow^n (w \uparrow, s)$.

Выход. Дерево разбора цепочки w .

Метод. Ядро алгоритма образует рекурсивная процедура **стройдерево**, которая по утверждению вида $A \Rightarrow^m (x \uparrow y, s)$ (это ее аргумент) строит дерево с корнем, помеченным A , и кроной x . Вначале эта процедура вызывается для утверждения $S \Rightarrow^n (w \uparrow, s)$.

Процедура стройдерево: Пусть $A \Rightarrow^m (x \uparrow y; s)$ — вход процедуры.

(1) Если для A есть правило $A \rightarrow a$ или $A \rightarrow \bar{e}$, то построить вершину с меткой A и для нее одного прямого потомка, помеченного a или \bar{e} соответственно. Остановиться.

(2) Если для A есть правило $A \rightarrow BC/D$ и x можно представить в виде $x = x_1 x_2$, так что $B \Rightarrow^{m_1} (x_1 \uparrow x_2 y, s)$ и $C \Rightarrow^{m_2} (x_2 \uparrow y, s)$, то построить вершину, помеченную A . Выполнить процедуру стройдерево для аргумента $B \Rightarrow^{m_1} (x_1 \uparrow x_2 y, s)$, а затем для аргумента $C \Rightarrow^{m_2} (x_2 \uparrow y, s)$. Присоединить получившиеся при этом деревья к вершине, помеченной A , так, чтобы корень дерева, получившегося в результате первого вызова, стал левым прямым потомком этой вершины, а корень второго дерева — правым. Остановиться.

(3) Если для A есть правило $A \rightarrow BC/D$, но (2) не выполняется, то должно быть $D \Rightarrow^{m_3} (x \uparrow y, s)$. Вызвать процедуру стройдерево для этого аргумента и сделать корень получившегося дерева единственным потомком уже построенной вершины с меткой A . Остановиться. \square

Заметим, что процедура стройдерево вызывает себя рекурсивно только для меньших значений m , так что алгоритм 6.1 должен в конце концов остановиться.

Пример 6.3. С помощью алгоритма 6.1 построим дерево разбора, порождаемое ЯНРОВ-программой P из примера 6.1 для входной цепочки aba .

Вначале вызовем процедуру стройдерево для утверждения $S \Rightarrow^r (aba \uparrow, s)$ в качестве ее аргумента. Правило $S \rightarrow AB/C$ приводит к успеху, потому что A и B успешно распознают цепочки a и ba соответственно. Затем вызовем эту процедуру дважды: сначала для аргумента $A \Rightarrow^1 (a \uparrow, s)$, а потом для аргумента $B \Rightarrow^5 (ba \uparrow, s)$. Таким образом, построение дерева начнется так, как показано на рис. 6.1, а. Вызов A для a сразу приводит к успеху, поэтому к вершине A присоединяется один прямой потомок, помеченный a . B приводит к успеху, потому что есть правило $B \rightarrow CB/A$ и нетерминалы C и B приводят к успеху для a и b соответственно. Таким образом, предыдущее дерево вырастет до дерева, показанного на рис. 6.1, в.

Сразу приводит к успеху для b , так что вершина, помеченная C , получит одного потомка, помеченного b . Вызов B успешен для a , поскольку успешен вызов A , так что B получает прямого потомка, помеченного A , а он в свою очередь — потомка, помеченного a . Все дерево показано на рис. 6.1, в. \square

Заметим, что если множество ЯНРОВ-правил трактуется как программа разбора, то всякий раз, когда нетерминал сообщает

об успехе, для распознанной им части входной цепочки можно построить соответствующий перевод (который позднее можно аннулировать), причем используются переводы „потомков“ этого нетерминала в смысле только что описанного дерева разбора.

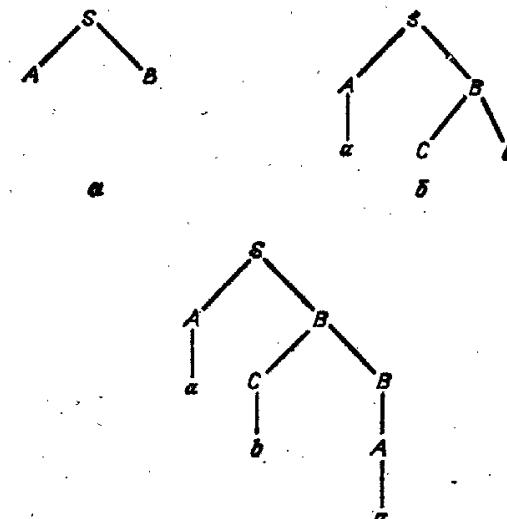


Рис. 6.1. Построение дерева разбора с помощью ЯНРОВ-программы.

Этот метод трансляции подобен синтаксически управляемой трансляции для контекстно-свободных грамматик. О нем еще будет идти речь в гл. 9.

Корректность алгоритма 6.1 нельзя „доказать“, потому что, как уже говорилось, он сам служит конструктивным определением дерева разбора, порождаемого ЯНРОВ-программой. Однако легко показать, что короной этого дерева является входная цепочка. Индукцией по числу вызовов процедуры стройдерево можно показать, что если она вызывается для аргумента $A \Rightarrow^m (x \uparrow y, s)$, то результатом будет дерево с короной x .

Некоторые из успешных результатов будут аннулированы. Иными словами, когда $A \Rightarrow^m (xy \uparrow z, s)$ есть правило $A \rightarrow BC/D$, может оказаться, что $B \Rightarrow^+ (x \uparrow yz, s)$, но $C \Rightarrow^+ (\uparrow yz, f)$. Тогда соотношение $B \Rightarrow^+ (x \uparrow yz, s)$ и все успешные шаги, с помощью которых оно было построено, на самом деле не включаются в определение дерева разбора входной цепочки (не считая самого отрицательного результата). Эти успешные шаги не отражены в дереве разбора, как и вызов процедуры стройдерево для аргу-

мента $B \Rightarrow^m (x \uparrow yz, s)$. Но в него включаются все успешные шаги, которые в конечном счете приводят к успешному распознаванию входной цепочки.

6.1.2. ЯНРОВ и детерминированные КС-языки

Может оказаться довольно трудным установить, какой язык определяет ЯНРОВ-программа. Чтобы дать представление об определяющей силе ЯНРОВ-программ, докажем, что каждый детерминированный КС-язык с концевым маркером распознается некоторой ЯНРОВ-программой. Кроме того, существует тесная связь между деревьями разбора, порождаемыми этой программой, и деревьями выводов в „естественной“ КС-грамматике для этого языка, которая строится по соответствующему ДМП-автомату так, как описано в лемме 2.26. Лемма 6.2 позволяет упростить представление ДМП-автомата, используемое в этом разделе.

Лемма 6.2. Если $L = L_e(M_1)$ для ДМП-автомата M_1 , то $L = L_e(M)$ для ДМП-автомата M , который на каждом своем шаге может увеличить длину магазина не более чем на 1.

Доказательство. Доказать это утверждение было предложено еще в упр. 2.5.3. Говоря неформально, надо вместо шага, заменяющего Z цепочкой $X_1 \dots X_k$ для $k > 2$, сделать шаги, заменяющие Z на $Y_k X_k$, Y_k на $Y_{k-1} X_{k-1}$, ..., Y_4 на $Y_3 X_3$ и Y_3 на $X_1 X_2$. Все Y_i — это новые магазинные символы, причем верх магазина расположены слева. \square

Лемма 6.3. Если L — детерминированный КС-язык и $\$$ — новый символ, то $L\$ = L_e(M)$ для некоторого ДМП-автомата M .

Доказательство. Доказательство получается простой модификацией доказательства леммы 2.22. Пусть $L = L(M_1)$. Тогда M моделирует M_1 , храня в конечной памяти своего управляющего устройства очередной входной символ. Если M_1 переходит в заключительное состояние и $\$$ — очередной входной символ, то M стирает содержимое магазина. Так как никаких других шагов не требуется и они невозможны, то M — детерминированный МП-автомат. \square

Теорема 6.1. Пусть $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ — ДМП-автомат и $L = L_e(M)$. Тогда существует такая ЯНРОВ-программа P , что $L = L(P)$.

Доказательство. Допустим, что M удовлетворяет лемме 6.2. Построим такую расширенную ЯНРОВ-программу $P = (N, \Sigma, R, S)$, что $L(P) = L_e(M)$. Множество нетерминалов N состоит из

- (1) символов S ,
- (2) символов вида $[qZp]$, где $q, p \in Q$ и $Z \in \Gamma$,
- (3) символов вида $[qZp]_a$, где q, Z и p те же, что и в (2), и $a \in \Sigma$.

Идея состоит в том, что вызов нетерминала $[qZp]$ успешен и приводит к распознаванию цепочки w тогда и только тогда, когда $(q, w, Z) \vdash_M^*(p, e, e)$, а при всех других условиях, включая случай $(q, w, Z) \vdash_M^*(p', e, e)$ для некоторого $p' \neq p$, вызов $[qZp]$ оканчивается неудачей. Вызов нетерминала $[qZp]_a$ распознает цепочку w тогда и только тогда, когда $(q, aw, Z) \vdash_M^*(p, e, e)$. Правила (операторы) программы P определяются так:

- (1) Для S есть правило $S \rightarrow [q_0 Z_0 q_0]/[q_0 Z_0 q_1]/\dots/[q_0 Z_0 q_k]$, где q_0, q_1, \dots, q_k — все состояния множества Q .
- (2) Если $\delta(q, e, Z) = (p, e)$, то для $[qZp]$ есть правило $[qZp] \rightarrow e$, и для каждого $p' \neq p$ есть правило $[qZp'] \rightarrow f$.
- (3) Если $\delta(q, e, Z) = (p, X)$, то для каждого $[qZr]$, где $r \in Q$, есть правило $[qZr] \rightarrow [pXr]$.
- (4) Если $\delta(q, e, Z) = (p, XY)$, то для каждого $[qZr]$, где $r \in Q$, есть правило $[qZr] \rightarrow [pXq_0][q_0 Yr]/[pXq_1][q_1 Yr]/\dots/[pXq_k][q_k Yr]$, где q_0, q_1, \dots, q_k — все состояния множества Q .
- (5) Если $\delta(q, e, Z) = \emptyset$, то пусть a_1, \dots, a_l — символы из Σ , для которых $\delta(q, a, Z) \neq \emptyset$. Тогда для каждого $[q, Z, r]$, где $r \in Q$, есть правило $[qZr] \rightarrow a_1[qZr]_{a_1}/a_2[qZr]_{a_2}/\dots/a_l[qZr]_{a_l}$. Если $l = 0$, то правило имеет вид $[qZr] \rightarrow f$.
- (6) Если $\delta(q, a, Z) = (p, e)$ для $a \in \Sigma$, то есть правило $[qZp]_a \rightarrow e$ и для $p' \neq p$ есть правило $[qZp']_a \rightarrow f$.
- (7) Если $\delta(q, a, Z) = (p, X)$, то для каждого $r \in Q$ есть правило $[qZr]_a \rightarrow [pXr]$.
- (8) Если $\delta(q, a, Z) = (p, XY)$, то для каждого $r \in Q$ есть правило $[qZr]_a \rightarrow [pXq_0][q_0 Yr]/[pXq_1][q_1 Yr]/\dots/[pXq_k][q_k Yr]$.

Заметим, что так как автомат M детерминированный, то эти определения непротиворечивы; для каждого нетерминала из N есть не более одного правила. Докажем утверждения:

- (6.1.1) $[qZp] \Rightarrow^+ (w \uparrow x, s)$ для любой цепочки x тогда и только тогда, когда $(q, wx, Z) \vdash_M^+ (p, x, e)$.
- (6.1.2) Если $(q, wx, Z) \vdash_M^+ (p, x, e)$, то $[qZp'] \Rightarrow^+ (\uparrow wx, f)$ для всех $p' \neq p$.

Докажем одновременно (6.1.2) и достаточность условия в (6.1.1) индукцией по числу шагов, сделанных автомата M при переходе из конфигурации (q, wx, Z) в конфигурацию (p, x, e) . Если сделан один шаг, то $\delta(q, a, Z) = (p, e)$, где a — это либо e , либо первый символ цепочки wx . В соответствии с (2) или (5) и (6) $[qZp] \Rightarrow^+ (a \uparrow y, s)$, где $ay = wx$, и $[qZp'] \Rightarrow^+ (\uparrow wx, f)$ для всех $p' \neq p$. Базис доказан.

Предположим, что нужный результат верен для любого числа шагов, меньшего числа шагов, потребовавшегося при переходе из конфигурации (q, wx, Z) в (p, x, e) . Пусть $w = aw'$ для некоторого $a \in \Sigma \cup \{e\}$. Тогда надо рассмотреть два случая.

Случай 1: Первый шаг — это $(q, wx, Z) \vdash (q', w'x, X)$ для некоторого $X \in \Gamma$. По предположению индукции $[q'Xp] \Rightarrow^+ (w' \uparrow x, s)$ и $[qZp'] \Rightarrow^+ (\uparrow w'x, f)$ для $p' \neq p$. Поэтому в силу (3) и (5), (7) $[qXp] \Rightarrow^+ (w \uparrow x, s)$ и $[qZp'] \Rightarrow^+ (\uparrow wx, f)$ для всех $p' \neq p$: Для строгого доказательства этих утверждений нужно сначала преобразовать расширенные правила программы P в правила исходного типа.

Случай 2: Для некоторых $X, Y \in \Gamma$, предполагая, что $w' = yw$, имеем $(q, wx, Z) \vdash (q', w'x, XY) \vdash^+ (q'', w'x, Y) \vdash^+ (p, x, e)$, где между конфигурациями $(q', w'x, XY)$ и $(q'', w'x, Y)$ магазин всегда содержит по крайней мере два символа. По предположению индукции $[q'Xq''] \Rightarrow^+ (y \uparrow w'x, s)$ и $[q''Yp] \Rightarrow^+ (w' \uparrow x, s)$. Кроме того, если $p' \neq q''$, то $[q'Xp'] \Rightarrow^+ (\uparrow w'x, f)$. Допустим сначала, что $a = e$. Если взглянуть на (4) и воспользоваться определением расширенных ЯНРОВ-операторов, то окажется, что каждая последовательность вида $[q'Yp'][p'Yp]$ приводит к неудаче при $p' \neq q''$. Однако, $[q'Xq''][q''Yp]$ приводит к успеху, и потому, как и хотелось, $[qZp] \Rightarrow^+ (w \uparrow x, s)$.

Далее заметим, что если $p' \neq p$, то $[q'Yp'] \Rightarrow^+ (\uparrow w'x, f)$, и потому все пары $[q'Yp'][p'Yp]$ приводят к неудаче. (Если $p'' \neq q''$, то к неудаче приводит $[q'Xp'']$, а если $p'' = q''$, то к неудаче приводит $[p'Yp']$.) Таким образом, $[qZp'] \Rightarrow^+ (\uparrow wx, f)$ для $p' \neq p$. Случай $a \in \Sigma$ исследуется аналогично с помощью (5) и (8).

Теперь нужно доказать необходимость условия в утверждении (6.1.1). Если $[qZp] \Rightarrow^+ (w \uparrow x, s)$, то $[qZp] \Rightarrow^n (w \uparrow x, s)$ для некоторого n^1 . Докажем наше утверждение индукцией по n . Если $n = 1$, то нужно применить (2) и результат получится элементарно. Допустим, что он верен для $n < n_0$, и пусть $[qZp] \Rightarrow^n (w \uparrow x, s)$.

Случай 1: Для $[qZp]$ есть правило $[qZp] \rightarrow [q'Xp]$. Тогда $\delta(q, e, Z) = (q', X)$ и $[q'Xp] \Rightarrow^{n_1} (w \uparrow x, s)$, где $n_1 < n_0$. По предположению индукции $(q', wx, X) \vdash^+ (p, x, e)$. Поэтому $(q, wx, Z) \vdash^+ (p, x, e)$.

Случай 2: Для $[qZp]$ есть правило $[qZp] \rightarrow [q'Xq_1][q_0Yp]/\dots/[q'Xq_k][q_kYp]$. Тогда $w = w'w''$ и для некоторого p' имеем $[q'Xp'] \Rightarrow^{n_1} (w' \uparrow w''x, s)$ и $[p'Yp] \Rightarrow^{n_2} (w'' \uparrow x, s)$, где n_1 и n_2 меньше n_0 . По предположению $(q', w'w''x, XY) \vdash^+ (p', w''x, Y) \vdash^+ (p, x, e)$. В силу (4) $\delta(q, e, Z) = (q', XY)$. Поэтому $(q, wx, Z) \vdash^+ (p, x, e)$.

1) Счет шагов должен вестись после перехода от расширенных правил к правилам первоначального вида.

Случай 3: Правило для $[qZp]$ определяется в (5), т. е. $\delta(q, e, Z) = \emptyset$. Тогда не может быть $w = e$, и потому пусть $w = aw'$. Если для нетерминала $[qZp]_a$ есть правило $[qZp]_a \rightarrow e$, то мы знаем, что $\delta(q, a, Z) = (p, e)$ и, следовательно, $w' = e$, $w = a$ и $(q, w, Z) \vdash (p, e, e)$. Ситуации, в которых правило для $[qZp]_a$ определяется в (7) или (8), рассматриваются аналогично случаям 1 и 2 соответственно. Соответствующие рассуждения мы опускаем.

Чтобы завершить доказательство теоремы, заметим, что $S \Rightarrow^+ (w \uparrow, s)$ тогда и только тогда, когда $[q_0Z_0p] \Rightarrow^+ (w \uparrow, s)$ для некоторого p . Из утверждения (6.1.1) следует, что $[q_0Z_0p] \Rightarrow^+ (w \uparrow, s)$ тогда и только тогда, когда $(q_0, w, Z_0) \vdash^+ (p, e, e)$. Таким образом, $L(P) = L_e(M)$. \square

Следствие. Если L — детерминированный КС-язык и $\$\text{—новый символ}$, то $L\$$ — язык, распознаваемый ЯНРОВ-программой.

Доказательство. Получается из леммы 6.3. \square

6.1.3. Обобщенный ЯНРОВ

Заметим, что если в ЯНРОВ-программе есть оператор $A \rightarrow BC/D$, то D вызывается, если либо B , либо C терпит неудачу. Не существует способа передавать управление по-разному, когда B терпит неудачу и когда B приводит к успеху, а C — к неудаче. Чтобы преодолеть этот недостаток, определим другой язык разбора, который назовем ОЯНРОВ, т. е. обобщенный язык нисходящего разбора с ограниченными возвратами. Программа, написанная на этом языке, представляет собой последовательность операторов видов

- (1) $A \rightarrow B[C, D]$
- (2) $A \rightarrow a$
- (3) $A \rightarrow e$
- (4) $A \rightarrow f$

Интуитивный смысл оператора $A \rightarrow B[C, D]$ заключается в том, что если вызывается нетерминал A , то он вызывает B . Если B приводит к успеху, то вызывается C . Если B терпит неудачу, то D вызывается в той входной позиции, где был вызван A . Результатом вызова A является результат вызова C или D смотря по тому, какой из этих нетерминалов был на самом деле вызван. Заметим, что это соглашение отличается от того, которое было принято для оператора $A \rightarrow BC/D$, где D вызывается тогда, когда B приводит к успеху, но C терпит неудачу.

Операторы типов (2)–(4) имеют тот же смысл, что и в ЯНРОВ-программе.

Определение. ОЯНРОВ-программой назовем четверку $P = (N, \Sigma, R, S)$, где N, Σ те же, что и в ЯНРОВ-программе, а R — список правил (операторов) вида $A \rightarrow B[C, D]$, $A \rightarrow a$ и $A \rightarrow f$. Здесь A, B, C и D принадлежат N , $a \in \Sigma \cup \{e\}$ и f — метасимвол, означающий неудачу, как и в ЯНРОВ-программе. Для каждого A есть не более одного правила, в котором A стоит слева от стрелки.

Определим для ОЯНРОВ-программы отношения \Rightarrow^n :

(1) Если для A есть правило $A \rightarrow a$, где $a \in \Sigma \cup \{e\}$, то $A \Rightarrow^1(a \uparrow w, s)$ для всех $w \in \Sigma^*$ и $A \Rightarrow^1(\uparrow w, f)$ для всех $w \in \Sigma^*$, у которых нет префикса a .

(2) Если для A есть правило $A \rightarrow f$, то $A \Rightarrow^1(\uparrow w, f)$ для всех $w \in \Sigma^*$.

(3) Если для A есть правило $A \rightarrow B[C, D]$, то

- (a) из $B \Rightarrow^m(w \uparrow xy, s)$ и $C \Rightarrow^n(x \uparrow y, s)$ следует $A \Rightarrow^{m+n+1}(wx \uparrow y, s)$,
- (б) из $B \Rightarrow^m(w \uparrow x, s)$ и $C \Rightarrow^n(\uparrow x, f)$ следует $A \Rightarrow^{m+n+1}(\uparrow wx, f)$,
- (в) из $B \Rightarrow^m(\uparrow wx, f)$ и $D \Rightarrow^n(w \uparrow x, s)$ следует $A \Rightarrow^{m+n+1}(w \uparrow x, s)$,
- (г) из $B \Rightarrow^m(\uparrow w, f)$ и $D \Rightarrow^n(\uparrow w, f)$ следует $A \Rightarrow^{m+n+1}(\uparrow w, f)$.

Будем писать $A \Rightarrow^+(x \uparrow y, r)$, если $A \Rightarrow^n(x \uparrow y, r)$ для некоторого $n \geq 1$. Языком, определяемым программой P , назовем язык $L(P) = \{w \mid S \Rightarrow^+(w \uparrow, s)\}$.

Пример 6.4. Пусть P — это ОЯНРОВ-программа с правилами

$$\begin{aligned} S &\rightarrow A[C, E] \\ C &\rightarrow S[B, E] \\ A &\rightarrow a \\ B &\rightarrow b \\ E &\rightarrow e \end{aligned}$$

Мы утверждаем, что P распознает язык $\{a^n b^n \mid n \geq 0\}$. Можно показать индукцией по n , что $S \Rightarrow^+(a^n b^n \uparrow x, s)$ и $C \Rightarrow^+(a^n b^{n+1} \uparrow x, s)$ для всех x и n . Например, для входной цепочки $aabb$

$$\begin{aligned} A &\Rightarrow^1(\uparrow bb, f) \\ E &\Rightarrow^1(\uparrow bb, s) \\ S &\Rightarrow^3(\uparrow bb, s) \\ B &\Rightarrow^1(b \uparrow b, s) \\ C &\Rightarrow^5(b \uparrow b, s) \\ A &\Rightarrow^1(a \uparrow bb, s) \\ S &\Rightarrow^2(ab \uparrow b, s) \\ B &\Rightarrow^1(b \uparrow, s) \\ C &\Rightarrow^0(ab \uparrow, s) \\ A &\Rightarrow^1(a \uparrow abb, s) \\ S &\Rightarrow^{11}(aabbb \uparrow, s) \quad \square \end{aligned}$$

Приведем теперь пример ОЯНРОВ-программы, определяющей не контекстно-свободный язык.

Пример 6.5. Построим ОЯНРОВ-программу, распознающую язык $\{0^n 1^n 2^n \mid n \geq 1\}$. Из примера 6.4 мы знаем, как написать правила, проверяющие, начинается ли в данной цепочке с некоторого места цепочка $0^n 1^n$ или $1^n 2^n$. Наша стратегия будет состоять в том, чтобы сначала проверить, что входная цепочка имеет префикс вида $0^m 1^n 2$ для $m \geq 0$. Если нет, то мы сделаем так, чтобы входная цепочка не была допущена. Если да, то произойдет неудачный промежуточный вызов, который заставит рассмотреть входную цепочку сначала. Затем проверим, имеет ли она вид $0^l 1^j 2^j$. Таким образом, входная цепочка выдержит оба эти испытания тогда и только тогда, когда она имеет вид $0^n 1^n 2^n$ для $n \geq 1$.

Нам нужны нетерминалы, распознающие один терминал или сразу приводящие к успеху или неудаче. С них начнем список правил:

- (1) $X \rightarrow 0$
- (2) $Y \rightarrow 1$
- (3) $Z \rightarrow 2$
- (4) $E \rightarrow e$
- (5) $F \rightarrow f$

Программу из примера 6.4 приспособим для распознавания цепочки $0^n 1^n 2$ нетерминалом S_1 . Соответствующие правила таковы:

- (6) $S_1 \rightarrow A[Z, Z]$
- (7) $A \rightarrow X[B, E]$
- (8) $B \rightarrow A[Y, E]$

Правила (7), (8), (1), (2) и (4) точно соответствуют аналогичным правилам примера 6.4. Правило (6) гарантирует, что S_1 распознает цепочку, состоящую из цепочки, распознаваемой нетерминалом A (это будет $0^m 1^m$), за которой следует символ 2. Заметим, что A всегда приводит к успеху, так что правило для S_1 могло бы иметь вид $S_1 \rightarrow A[Z, W]$ для любого W .

Далее, надо написать правила, распознающие цепочку вида 0^n , за которой следует $1^j 2^j$ для некоторого j . Для этого достаточно взять

- (9) $S_2 \rightarrow X[S_2, C]$
- (10) $C \rightarrow Y[D, E]$
- (11) $D \rightarrow C[Z, E]$

Правила (10), (11), (2), (3) и (4) соответствуют аналогичным правилам примера 6.4, причем C распознает $1^j 2^j$. Правило для S_2 работает так: пока на входе идет префикс, состоящий из ну-

лей, S_1 распознает его и продолжает далее вызывать себя; когда X терпит неудачу, т. е. входной указатель моновал нули, вызывается нетерминал C , распознающий префикс вида $1/2^*$. Заметим, что C всегда приводит к успеху, поэтому и S_2 тоже.

Теперь нужно соединить программы для S_1 и S_2 . Сначала введем нетерминал S_3 , который сам не распознает никакой входной цепочки, т. е. никогда не меняет положение входного указателя, но приводит к успеху или неудаче тогда, когда S_1 приводит соответственно к неудаче или успеху. Правило для S_3 таково:

$$(12) \quad S_3 \rightarrow S_1[F, E]$$

Заметим, что если S_1 приводит к успеху, то S_3 вызывает нетерминал F , который сообщает о неудаче и возвращает входной указатель туда, где был вызван S_1 . Если S_1 терпит неудачу, то S_3 вызывает нетерминал E , который сообщает об успехе. Таким образом, S_3 в любом случае не „потребляет“ входную цепочку. Теперь введем начальный символ S с правилом

$$(13) \quad S \rightarrow S_3[F, S_2]$$

Если S_1 приводит к успеху, то S_3 сообщает о неудаче, и S_3 вызывается в начале входной цепочки. Поэтому S приводит к успеху тогда, когда S_1 и S_2 приводят к успеху для одной и той же входной цепочки. Если S_1 приводит к неудаче, то S_3 сообщает об успехе, так что S приводит к неудаче. Если S_1 сообщает об успехе, а S_2 — о неудаче, то S тоже сообщает о неудаче. Таким образом, эта программа распознает язык $\{0^n 1^n 2^n \mid n \geq 1\}$, представляющий собой пересечение множеств, распознаваемых нетерминалами S_1 и S_2 . Следовательно, существуют не контекстно-свободные языки, определяемые ОЯНРОВ-программами. То же, кстати, верно и для ЯНРОВ-программ (см. упр. 6.1.1). \square

Изучим некоторые свойства ОЯНРОВ-программ. Справедлива лемма, аналогичная лемме 6.1:

Лемма 6.4. Пусть $P = (N, \Sigma, R, S)$ — ОЯНРОВ-программа. Если $A \Rightarrow^+ (x \uparrow y, r_1)$ и $A \Rightarrow^+ (u \uparrow v, r_2)$, где $xy = uv$, то $x = u$, $y = v$ и $r_1 = r_2$.

Доказательство. Упражнение. \square

Теперь докажем две теоремы об ОЯНРОВ-программах. Первая говорит о том, что класс языков, определяемых ЯНРОВ-программами, содержится в классе языков, определяемых ОЯНРОВ-программами. Вторая утверждает, что каждый язык, определяемый ОЯНРОВ-программой, распознается за линейное время подходящей машиной с произвольным доступом к памяти.

Теорема 6.2. Каждый язык, определяемый ЯНРОВ-программой, определяется ОЯНРОВ-программой.

Доказательство. Пусть $L = L(P)$ для ЯНРОВ-программы $P = (N, \Sigma, R, S)$. Возьмем ОЯНРОВ-программу $P' = (N', \Sigma, R', S)$, где R' определяется так:

- (1) Если правило $A \rightarrow e$ принадлежит R , то оно включается в R' .
- (2) Если правило $A \rightarrow a$ принадлежит R , то оно включается в R' .
- (3) Если правило $A \rightarrow f$ принадлежит R , то оно включается в R' .
- (4) Вводятся новые нетерминалы E, F и в R' включаются правила $E \rightarrow e$ и $F \rightarrow f$. (Заметим, что другие нетерминалы с такими же правилами можно отождествить с этими.)
- (5) Если $A \rightarrow BC/D \in R$, то в R' включаются правила

$$\begin{aligned} A &\rightarrow A'[E, D] \\ A' &\rightarrow B[C, F] \end{aligned}$$

где A' — новый нетерминал.

Пусть N' — это N вместе со всеми новыми нетерминалами, введенными при построении R' .

Элементарное наблюдение показывает, что если B и C приводят к успеху, то A' приводит к успеху, а в противном случае A' сообщает о неудаче. Таким образом, A приводит к успеху тогда и только тогда, когда A' приводит к успеху (т. е. B и C сообщают об успехе) или A' терпит неудачу (т. е. B терпит неудачу, или B приводит к успеху, а C — к неудаче), а D приводит к успеху. Легко также проверить, что нетерминалы B , C и D вызываются программой P' в тех же точках входной цепочки, в которых они вызываются программой P . Так как R' непосредственно моделирует каждое правило из R , то $S \Rightarrow^+ (w \uparrow s)$ тогда и только тогда, когда $S \Rightarrow^+ (w \uparrow s)$, и, значит, $L(P) = L(P')$. \square

По-видимому, ОЯНРОВ-программы могут в смысле распознавания делать больше, чем ЯНРОВ-программы. Например, легко написать ОЯНРОВ-программу, моделирующую оператор вида

$$A \rightarrow BC/(D_1, D_2)$$

в котором D_1 должен вызываться тогда, когда B терпит неудачу, а D_2 — тогда, когда B приводит к успеху и C — к неудаче. Но вопрос о том, образуют ли языки, определяемые ЯНРОВ-программами, собственный подкласс класса языков, определяемых ОЯНРОВ-программами, остается открытым.

Так же, как и предыдущий язык, можно сделать ОЯНРОВ более выразительным, введя расширенные операторы (см. упр. 6.1.12). Например, каждый расширенный ЯНРОВ-оператор можно рассматривать как расширенную форму ОЯНРОВ-оператора.

6.1.4. Временная сложность ОЯНРОВ-языков

Главный результат этого раздела состоит в том, что успешный процесс распознавания входной цепочки ОЯНРОВ-программой (а следовательно, и ЯНРОВ-программой) можно промоделировать за линейное время на автомате, подобном машине с произвольным доступом к памяти. Соответствующий алгоритм напоминает как алгоритм Кока—Янгера—Касами, так и алгоритм Эрли из разд. 4.2, но входную цепочку он обрабатывает в обратном направлении..

Алгоритм 6.2. Распознавание ОЯНРОВ-языков за линейное время.

Вход. ОЯНРОВ-программа $P = (N, \Sigma, R, S)$, где $N = \{A_1, A_2, \dots, A_k\}$ и $S = A_1$ и входная цепочка $w = a_1 a_2 \dots a_n$ из Σ^* . Предполагается, что $a_{n+1} = \$$ — правый концевой маркер.

Выход. Матрица $[t_{ij}]$ размера $k \times (n+1)$. Каждый ее элемент либо не определен, либо это целое число m ($0 \leq m \leq n$), либо символ неудачи f . Если $t_{ij} = m$, то $A_i \Rightarrow^+ (a_j a_{j+1} \dots a_{j+m-1} \uparrow a_{j+m} \dots a_n, s)$. Если $t_{ij} = f$, то $A_i \Rightarrow^+ (\uparrow a_j \dots a_n, f)$. В остальных случаях t_{ij} не определен.

Метод. Матрицу $[t_{ij}]$ будем вычислять следующим образом. Вначале все ее элементы не определены.

(1) Делать шаги (2)–(4) для $j = n+1, n, \dots, 1$.

(2) Для каждого $1 \leq i \leq k$ если в R есть правило $A_i \rightarrow f$, положить $t_{ij} = f$; если в R есть правило $A_i \rightarrow e$, положить $t_{ij} = 0$; если $A_i \rightarrow a_j$, положить $t_{ij} = 1$ и, если $A \rightarrow b$ для $b \neq a_j$, положить $t_{ij} = f$. (Так как $a_{n+1} \notin \Sigma$, то $A \rightarrow a_{n+1} \notin R$.)

(3) Повторять шаг (4) для $i = 1, 2, \dots, k$, пока t_{ij} не перестанут меняться.

(4) Пусть правило для A_i имеет вид $A_i \rightarrow A_p [A_q, A_r]$ и элемент t_{ij} еще не определен:

- (а) если $t_{pj} = f$ и $t_{rj} = x$, положить $t_{ij} = x$, где x — число или f ,
- (б) если $t_{pj} = m_1$ и $t_{q(j+m_1)} = m_2 \neq f$, положить $t_{ij} = m_1 + m_2$,
- (в) если $t_{pj} = m_1$ и $t_{q(j+m_1)} = f$, положить $t_{ij} = f$.

Во всех остальных случаях с t_{ij} ничего не делать. \square

Теорема 6.3. Алгоритм 6.2 правильно определяет t_{ij} .

Доказательство. Мы утверждаем, что после выполнения алгоритма 6.2 для входной цепочки $w = a_1 \dots a_n$ будет $t_{ij} = f$ тогда и только тогда, когда $A_i \Rightarrow^+ (\uparrow a_j \dots a_n, f)$, и $t_{ij} = m$ тогда и только тогда, когда $A_i \Rightarrow^+ (a_j \dots a_{j+m-1} \uparrow a_{j+m} \dots a_n, s)$. Прямая индукция по порядку, в котором вычисляются элементы t_{ij} , показывает, что когда элемент получает значение, оно именно то, о котором сказано выше. Обратно, индукция по i показывает, что если $A_i \Rightarrow^l (\uparrow a_j \dots a_n, f)$ или $A_i \Rightarrow^l (a_j \dots a_{j+m-1} \uparrow a_{j+m} \dots a_n, s)$, то t_{ij} получает значение f или m соответственно. Элемент t_{ij} остается неопределенным, если вызов A_i в позиции j не заканчивается. Детали доказательства оставляем в качестве упражнения. \square

Заметим, что $a_1 \dots a_n \in L(P)$ тогда и только тогда, когда $t_{11} = n$.

Теорема 6.4. Для каждой ОЯНРОВ-программы существует такая константа c , что для входной цепочки длины $n \geq 1$ алгоритм 6.2 тратит не более c элементарных шагов, причем эти элементарные шаги того же типа, что и в алгоритме 4.3.

Доказательство. Суть доказательства в том, чтобы заметить, что на шаге (3) мы проходим по всем нетерминалам не более k раз для любого данного j . \square

В заключение отметим, что по матрице, вычисленной алгоритмом 6.2, можно для допускаемых входных цепочек строить древовидные синтаксические структуры, подобные тем, что строились алгоритмом 6.1 для ЯНРОВ-программ. Кроме того, алгоритм 6.2 можно модифицировать так, чтобы он работал (распознавал и анализировал) для обычной, а не для обобщенной ЯНРОВ-программы.

Пример 6.6. Пусть $P = (N, \Sigma, R_1, E)$, где $N = \{E, E_+, T, T_*, F, F', X, Y, P, M, A, L, R\}$, $\Sigma = \{a, (,), +, *\}$ и R_1 состоит из правил

- (1) $E \rightarrow T [E_+, X]$
- (2) $E_+ \rightarrow P [E, Y]$
- (3) $T \rightarrow F [T_*, X]$
- (4) $T_* \rightarrow M [T, Y]$
- (5) $F \rightarrow L [F', A]$
- (6) $F' \rightarrow E [R, X]$
- (7) $X \rightarrow f$
- (8) $Y \rightarrow e$
- (9) $P \rightarrow +$
- (10) $M \rightarrow *$
- (11) $A \rightarrow a$
- (12) $L \rightarrow ($
- (13) $R \rightarrow)$

Эта ОЯНРОВ-программа предназначена для распознавания обычных арифметических выражений с операциями + и *, т. е. языка $L(G_0)$. Нетерминал E распознает выражение, состоящее из термов (т. е. цепочек, распознаваемых нетерминалом T), разделенных знаками +. Нетерминал E_+ распознает выражение, в котором первый терм опущен. Таким образом, правило (2) говорит о том, что E_+ распознает знак + (распознаваемый P),

	($a + a$)	*	a	\$
E	7	3	f	1	f	f
E_+	0	0	2	0	0	0
T	7	1	f	1	f	f
T_+	0	0	0	0	2	0
F	5	1	f	1	f	f
F'	f	4	f	2	f	f
X	f	f	f	f	f	f
Y	0	0	0	0	0	0
P	f	f	1	f	f	f
M	f	f	f	f	1	f
A	f	1	f	1	f	f
L	1	f	f	f	f	f
R	f	f	f	f	1	f

Рис. 6.2. Таблица распознавания, построенная алгоритмом 6.2.

за которым следует любое выражение, а если знака + нет, то — пустую цепочку (распознаваемую Y). Тогда можно так интерпретировать правило (1): выражение — это терм, за которым следует нечто, распознаваемое E_+ , а именно либо пустая цепочка, либо цепочка из чередующихся знаков + и термов, начинаящаяся с + и оканчивающаяся термом. Аналогично интерпретируются правила (3) и (4).

Правила (5) и (6) говорят о том, что множитель (цепочка, распознаваемая F) — это либо заключенное в скобки выражение, либо символ a (если скобок нет).

Пусть $(a+a)*a$ — входящая цепочка для алгоритма 6.2. Тогда он построит матрицу $[t_{ij}]$, изображенную на рис. 6.2.

Вычислим для примера элементы последнего столбца этой матрицы. Элементы, соответствующие нетерминалам P , M , A , L и R , принимают значение f , так как каждый из указанных нетерминалов распознает символ из Σ , а последний символ — это концевой маркер. X всегда дает значение f , а Y — всегда 0. Применяя шаг (3) нашего алгоритма, находим, что первый проход по нетерминалам на шаге (4) дает значения для E_+ , T_+ , F и они равны соответственно 0, 0, f . После второго прохода T получает значение f . Значения для E и F' можно вычислить на третьем проходе.

Пример менее тривиального вычисления встречается в третьем столбце. Нижние семь элементов легко получаются по правилу (2). Далее, применяя (2) и учитывая, что элемент третьего столбца, соответствующий P , равен 1, находим, что элемент четвертого столбца ($4 = 3 + 1$), соответствующий E , тоже равен 1. Таким образом, элемент третьего столбца, соответствующий E_+ , равен 2 ($= 1 + 1$). \square

6.1.5. Реализация ОЯНРОВ-программ

На практике системы синтаксического анализа, подобные ОЯНРОВ-программам, реализуются не в табличной форме, как это сделано в алгоритме 6.2, а обычно с помощью метода проб и ошибок. Мы построим сейчас автомат, «реализующий» распознавающую часть ОЯНРОВ-программ. Читателю предоставляем самому показать, как расширить этот автомат до преобразователя, выдающего последовательность успешных вызовов процедур, по которой можно построить «разбор» или перевод.

Автомат состоит из входной ленты с указателем, положение которого можно восстанавливать, управляющего устройства с тремя состояниями и магазина, в котором помещаются символы из некоторого конечного алфавита и указатели входных позиций. В работе этого автомата воплощается наше интуитивное представление озывающих друг друга процедурах (нетерминалах) и возвратах входного указателя, происходящих при каждой неудаче. Указатель возвращается туда, где он находился в тот момент, когда состоялся вызов неудачной процедуры:

Определение. Анализирующей машиной назовем шестерку $M = (Q, \Sigma, \Gamma, \delta, \text{начало}, Z_0)$, где

- (1) $Q = \{\text{успех}, \text{неудача}, \text{начало}\}$;
- (2) Σ — конечное множество входных символов;
- (3) Γ — конечное множество магазинных символов;
- (4) δ — такое отображение множества $Q \times (\Sigma \cup \{e\}) \times \Gamma$ в множество $Q \times \Gamma^{*2}$, что
 - (а) если q — успех или неудача, то $\delta(q, a, Z)$ не определено, если $a \in \Sigma$, и $\delta(q, e, Z)$, имеет вид $(\text{начало}, Y)$ для некоторого $Y \in \Gamma$;
 - (б) если $\delta(\text{начало}, a, Z)$ определено для некоторого $a \in \Sigma$, то $\delta(\text{начало}, b, Z)$ не определено для всех $b \neq a$ из $\Sigma \cup \{e\}$;
 - (в) для $a \in \Sigma$ значением $\delta(\text{начало}, a, Z)$, если оно определено, может быть только $(\text{успех}, e)$;
 - (г) $\delta(\text{начало}, e, Z)$ может иметь только вид $(\text{начало}, YZ)$ для некоторого $Y \in \Gamma$ или вид (q, e) для $q = \text{успех}$ либо $q = \text{неудача}$;

- (5) начало — начальное состояние;
 (6) $Z_0 \in \Gamma$ — начальный магазинный символ.

Машина M похожа на автомат с магазинной памятью, но между ними есть несколько важных различий. Элементы множества Γ можно представлять себе как процедуры, вызывающие друг друга или „переходящие“ одна в другую. Магазин используется для записи рекурсивных вызовов и положения входной головки в момент вызова. Состояние начало обычно служит для вызова другой процедуры, это отражается в том, что если $\delta(\text{начало}, e, Z) = (\text{начало}, YZ)$, где $Y \in \Gamma$ и Z — верхний символ магазина, то Y помещается в магазин непосредственно над Z . Состояния успех и неудача служат для перехода к другой процедуре, а не для вызова ее. Если, например, $\delta(\text{успех}, e, Z) = (\text{начало}, Y)$, то Y просто заменяет Z наверху магазина. Определим формально действия машины M .

Конфигурацией машины M назовем тройку $(q, w \uparrow x, \gamma)$, где

- (1) $q \in \{\text{успех}, \text{неудача}, \text{начало}\}$;
- (2) w и x принадлежат Σ^* , \uparrow — метасимвол, указывающий положение входной головки;
- (3) γ — содержимое магазина, имеющее вид $(Z_1, i_1) \dots (Z_m, i_m)$, где $Z_j \in \Gamma$ и i_j — целое число при $1 \leq j \leq m$. Верх магазина расположен слева. Z_i для каждого i — это вызов „процедуры“, а i_j — входной указатель.

Зададим на множестве конфигураций отношение \vdash_M (или \vdash , когда M подразумевается):

- (1) Пусть $\delta(\text{начало}, e, Z) = (\text{начало}, YZ)$ для $Y \in \Gamma$. Тогда
 $(\text{начало}, w \uparrow x, (Z, i) \gamma) \vdash (\text{начало}, w \uparrow x, (Y, j) (Z, i) \gamma)$

где $j = |w|$. Здесь происходит „вызов“ Y и регистрируются положение входной головки в момент вызова и вызываемая „процедура“ Y .

- (2) Пусть $\delta(q, e, Z) = (\text{начало}, Y)$, где $Y \in \Gamma$ и $q \in \{\text{успех}, \text{неудача}\}$. Тогда

$$(q, w \uparrow x, (Z, i) \gamma) \vdash (\text{начало}, w \uparrow x, (Y, i) \gamma)$$

Здесь Z „переходит“ в Y . Входная позиция, соответствующая Y , та же, что и позиция, соответствующая Z .

- (3) Пусть $\delta(\text{начало}, a, Z) = (q, e)$ для $a \in \Sigma \cup \{e\}$. Если $q = \text{успех}$, то
 $(\text{начало}, w \uparrow ax, (Z, i) \gamma) \vdash (\text{успех}, wa \uparrow x, \gamma)$

Если a не является префиксом цепочки x или $q = \text{неудача}$, то
 $(\text{начало}, w \uparrow x, (Z, i) \gamma) \vdash (\text{неудача}, u \uparrow v, \gamma)$

где $uv = wx$ и $|u| = i$. В последнем случае входной указатель возвращается на позицию, задаваемую указателем, находящимся наверху магазина.

Заметим, что при $\delta(\text{начало}, a, Z) = (\text{успех}, e)$ очередным состоянием анализирующей машины будет успех, если необработанная часть входной цепочки начинается символом a , и неудача в противном случае.

Пусть \vdash^+ обозначает транзитивное замыкание отношения \vdash . Языком, определяемым машиной M , назовем множество $L(M) = \{w \mid w \in \Sigma^* \text{ и } (\text{начало}, \uparrow w, (Z_0, 0)) \vdash^+ (\text{успех}, w \uparrow, e)\}$.

Пример 6.7. Пусть $M = (Q, \{a, b\}, \{Z_0, Y, A, B, E\}, \delta, \text{начало}, Z_0)$, где δ задается равенствами

- (1) $\delta(\text{начало}, e, Z_0) = (\text{начало}, YZ_0)$
- (2) $\delta(\text{успех}, e, Z_0) = (\text{начало}, Z_0)$
- (3) $\delta(\text{неудача}, e, Z_0) = (\text{начало}, E)$
- (4) $\delta(\text{начало}, e, Y) = (\text{начало}, AY)$
- (5) $\delta(\text{успех}, e, Y) = (\text{начало}, Y)$
- (6) $\delta(\text{неудача}, e, Y) = (\text{начало}, B)$
- (7) $\delta(\text{начало}, a, A) = (\text{успех}, e)$
- (8) $\delta(\text{начало}, b, B) = (\text{успех}, e)$
- (9) $\delta(\text{начало}, e, E) = (\text{успех}, e)$

M распознает цепочки, состоящие из символов a и b и оканчивающиеся символом b , но делает это довольно своеобразно. A и B распознают соответственно a и b . Когда начинает действовать символ Y , он ищет символ a и, если отыскивает его, „переходит“ в себя. Таким образом, магазин не меняется, а на входе „потребляется“ символ a . Если достигнут символ b или конец входной цепочки, то Y в состоянии неудача стирает верхнюю ячейку магазина. Иными словами, Y заменяется символом B и независимо от того, приводит B к успеху или неудаче, B в конце концов стирается.

Z_0 вызывает Y (и переходит в себя) так же, как Y вызывает A . Поэтому любая цепочка, состоящая из символов a и b и оканчивающаяся символом b , приводит к тому, что Z_0 сотрется и наступит успех. Действия машины M на входной цепочке $abaa$ образуют такую последовательность конфигураций:

- $(\text{начало}, \uparrow abaa, (Z_0, 0)) \vdash (\text{начало}, \uparrow abaa, (Y, 0) (Z_0, 0))$
- $\vdash (\text{начало}, \uparrow abaa, (A, 0) (Y, 0) (Z_0, 0))$
- $\vdash (\text{успех}, a \uparrow baa, (Y, 0) (Z_0, 0))$
- $\vdash (\text{начало}, a \uparrow baa, (Y, 0) (Z_0, 0))$
- $\vdash (\text{начало}, a \uparrow baa, (A, 1) (Y, 0) (Z_0, 0))$
- $\vdash (\text{неудача}, a \uparrow baa, (Y, 0) (Z_0, 0))$
- $\vdash (\text{начало}, a \uparrow baa, (B, 0) (Z_0, 0))$

— (успех, $ab \uparrow aa$, $(Z_0, 0)$)
— (начало, $ab \uparrow aa$, $(Z_0, 0)$)
— (начало, $ab \uparrow aa$, $(Y, 2)(Z_0, 0)$)
— (начало, $ab \uparrow aa$, $(A, 2)(Y, 2)(Z_0, 0)$)
— (успех, $aba \uparrow a$, $(Y, 2)(Z_0, 0)$)
— (начало, $aba \uparrow a$, $(Y, 2)(Z_0, 0)$)
— (начало, $aba \uparrow a$, $(A, 3)(Y, 2)(Z_0, 0)$)
— (успех, $abaa \uparrow$, $(Y, 2)(Z_0, 0)$)
— (начало, $abaa \uparrow$, $(Y, 2)(Z_0, 0)$)
— (начало, $abaa \uparrow$, $(A, 4)(Y, 2)(Z_0, 0)$)
— (неудача, $abaa \uparrow$, $(Y, 2)(Z_0, 0)$)
— (начало, $abaa \uparrow$, $(B, 2)(Z_0, 0)$)
— (неудача, $ab \uparrow aa$, $(Z_0, 0)$)
— (начало, $ab \uparrow aa$, $(E, 0)$)
— (успех, $ab \uparrow aa$, e)

Заметим, что цепочка $abaa$ не допускается, потому что конец ее не достигается на последнем шаге. Однако цепочка ab была бы допущена. Важно также отметить, что в четвертой с конца конфигурации B не „вызывается“, а заменяет Y . Поэтому наверху магазина появляется 2, а не 4, и, когда B приводит к неудаче, входная головка возвращается назад. \square

Докажем теперь, что язык определяется анализирующей машиной тогда и только тогда, когда он определяется ОЯНРОВ-программой.

Лемма 6.5. Если $L = L(M)$ для некоторой анализирующей машины $M = (Q, \Sigma, \Gamma, \delta, \text{начало}, Z_0)$, то $L = L(P)$ для некоторой ОЯНРОВ-программы P .

Доказательство. Пусть $P = (N, \Sigma, R, Z_0)$, где $N = \Gamma \cup \{X\}$ и X — новый символ. Определим R :

(1) Для X правила нет.

(2)-Если $\delta(\text{начало}, a, Z) = (q, e)$, то для Z будет правило $Z \rightarrow a$, если $q = \text{успех}$, и правило $Z \rightarrow f$, если $q = \text{неудача}$.

(3) Для других $Z \in \Gamma$ зададим Y_1, Y_2 и Y_3 так:

(а) если $\delta(\text{начало}, e, Z) = (\text{начало}, YZ)$, положим $Y_1 = Y$,

(б) если $\delta(q, e, Z) = (\text{начало}, Y)$, положим $Y_2 = Y$, если $q = \text{успех}$, и $Y_3 = Y$, если $q = \text{неудача}$,

(в) если Y_i не определен в (а) и (б), положим $Y_i = X$.

Тогда для Z введем правило $Z \rightarrow Y_1[Y_2, Y_3]$.

Покажем, что для каждого $Z \in \Gamma$

(6.1.3) $Z \Rightarrow^n (w \uparrow x, s)$ тогда и только тогда, когда $(\text{начало}, \uparrow wx, (Z, 0)) \vdash^{-n} (\text{успех}, w \uparrow x, e)$

(6.1.4) $Z \Rightarrow^n (\uparrow w, f)$ тогда и только тогда, когда $(\text{начало}, \uparrow w, (Z, 0)) \vdash^{-n} (\text{неудача}, \uparrow w, e)$

Докажем оба утверждения одновременно индукцией по числу шагов вывода программы P или вычисления машины M .

Необходимость. Базисы для (6.1.3) и (6.1.4) получаются непосредственно из определения отношения \vdash .

Чтобы доказать шаг индукции для (6.1.3), допустим, что $Z \Rightarrow^n (w \uparrow x, s)$ и утверждения (6.1.3) и (6.1.4) верны для меньших значений n . Так как мы взяли $n > 1$, то для Z есть правило $Z \rightarrow Y_1[Y_2, Y_3]$.

Случай 1: $w = w_1 w_2$, $Y_1 \Rightarrow^{n_1} (w_1 \uparrow w_2 x, s)$ и $Y_2 \Rightarrow^{n_2} (w_2 \uparrow x, s)$. Тогда n_1 и n_2 меньше n и по предположению индукции

(6.1.5) $(\text{начало}, \uparrow wx, (Y_1, 0)) \vdash^+ (\text{успех}, w_1 \uparrow w_2 x, e)$

(6.1.6) $(\text{начало}, \uparrow w_2 x, (Y_2, 0)) \vdash^+ (\text{успех}, w_2 \uparrow x, e)$

Заметим, что если слева от входной головки машины M вставить какую-нибудь цепочку, в частности w_1 , то M выполнит те же действия. Поэтому из (6.1.6) получаем

(6.1.7) $(\text{начало}, w_1 \uparrow w_2 x, (Y_2, 0)) \vdash^+ (\text{успех}, w \uparrow x, e)$

Этот вывод требует, правда, доказательства, но мы оставим его в качестве упражнения.

Из определения R мы знаем, что $\delta(\text{начало}, e, Z) = (\text{начало}, Y_1 Z)$ и $\delta(\text{успех}, e, Z) = (\text{начало}, Y_2)$. Поэтому

(6.1.8) $(\text{начало}, \uparrow wx, (Z, 0)) \vdash (\text{начало}, \uparrow wx, (Y_1, 0)(Z, 0))$

(6.1.9) $(\text{успех}, w_1 \uparrow w_2 x, (Z, 0)) \vdash (\text{начало}, w_1 \uparrow w_2 x, (Y_2, 0))$

Объединяя (6.1.9), (6.1.5), (6.1.8) и (6.1.7), получаем нужное утверждение:

$(\text{начало}, \uparrow wx, (Z, 0)) \vdash^+ (\text{успех}, w \uparrow x, e)$

Случай 2: $Y_1 \Rightarrow^{n_1} (\uparrow wx, f)$ и $Y_3 \Rightarrow^{n_2} (w \uparrow x, s)$. Доказательство аналогично доказательству в случае 1, и мы оставляем его читателю.

Теперь проведем индукцию для (6.1.4). Пусть $Z \Rightarrow^n (\uparrow w, f)$.

Случай 1: $Y_1 \Rightarrow^{n_1} (w_1 \uparrow w_2, s)$ и $Y_2 \Rightarrow^{n_2} (\uparrow w_2, f)$, где $w_1 w_2 = w$. Тогда $n_1, n_2 < n$ и из (6.1.3) и (6.1.4) получаем

(6.1.10) $(\text{начало}, \uparrow w, (Y_1, 0)) \vdash^+ (\text{успех}, w_1 \uparrow w_2, e)$

(6.1.11) $(\text{начало}, \uparrow w_2, (Y_2, 0)) \vdash^+ (\text{неудача}, \uparrow w_2, e)$

Если в (6.1.11) слева от \uparrow вставить w_1 , будет

(6.1.12) $(\text{начало}, w_1 \uparrow w_2, (Y_2, 0)) \vdash^+ (\text{неудача}, \uparrow w_1 w_2, e)$

Доказательство последнего утверждения оставляем в качестве упражнения. Надо заметить, что когда стирается $(Y_2, 0)$, входная головка должна вернуться до конца влево. Присутствие цепочки w_1 не меняет дела, потому что тогда к числам, записываемым в магазин выше $(Y_1, 0)$, добавляется $|w_1|$ (при построении последовательности шагов, соответствующей (6.1.12), по последовательности (6.1.11)) и, значит, входная головка не может выйти из цепочки w_i , не стерев $(Y_1, 0)$.

По определению Y_1 и Y_2

$$(6.1.13) \text{ (начало, } \uparrow w, (Z, 0)) \vdash \text{ (начало, } \uparrow w, (Y_1, 0) (Z, 0))$$

$$(6.1.14) \text{ (успех, } w_1 \uparrow w_2, (Z, 0)) \vdash \text{ (начало, } w_1 \uparrow w_2, (Y_2, 0))$$

Объединяя (6.1.13), (6.1.10), (6.1.14) и (6.1.12), получаем $\text{(начало, } \uparrow w, (Z, 0)) \vdash^+ \text{(неудача, } \uparrow w, e)$.

Случай 2: $Y_1 \Rightarrow^{n_1} (\uparrow w, f)$ и $Y_2 \Rightarrow^{n_2} (\uparrow w, f)$. Этот случай аналогичен предыдущему: оставляем его читателю.

Достаточность. Эта часть доказательства проводится аналогично; мы оставляем ее в качестве упражнения.

Из утверждения (6.1.3), в частности, следует, что $Z_0 \Rightarrow^+ (\uparrow w, s)$, тогда и только тогда, когда $\text{(начало, } \uparrow w, (Z_0, 0)) \vdash^+ \text{(успех, } \uparrow w, e)$, так что $L(M) = L(P)$. \square

Лемма 6.6. Если $L = L(P)$ для некоторой ОЯНРОВ-программы P , то $L = L(M)$ для анализирующей машины M .

Доказательство. Пусть $P = (N, \Sigma, R, S)$. Зададим $M = (Q, \Sigma, N, \delta, \text{ начало}, S)$, где δ определяется так:

- (1) Если в R есть правило $A \rightarrow B[C, D]$, то $\delta(\text{ начало}, e, A) = \text{(начало, } BA)$, $\delta(\text{ успех, } e, A) = \text{(начало, } C)$ и $\delta(\text{ неудача, } e, A) = \text{(начало, } D)$.
- (2) (a) Если в R есть $A \rightarrow a$, где $a \in \Sigma \cup \{e\}$, то $\delta(\text{ начало}, a, A) = \text{(успех, } e)$.
 (б) Если в R есть $A \rightarrow f$, то $\delta(\text{ начало}, e, A) = \text{(неудача, } e)$.

Простое доказательство равенства $L(M) = L(P)$ оставляем в качестве упражнения. \square

Теорема 6.5. $L = L(M)$ для некоторой анализирующей машины M тогда и только тогда, когда $L = L(P)$ для некоторой ОЯНРОВ-программы P .

Доказательство. Получается непосредственно из лемм 6.5 и 6.6. \square

УПРАЖНЕНИЯ

6.1.1. Постройте ЯНРОВ-программы, распознающие следующие языки:

(а) $L(G_0)$,

(б) множество цепочек, содержащих одинаковое число символов a и b ,

(в) $\{wcbw^R \mid w \in (a+b)^*\}$,

*(г) $\{a^{2^n} \mid n \geq 1\}$ (Указание: Рассмотрите $S \rightarrow aSa \mid aa$),

(д) какое-нибудь бесконечное подмножество Фортрана.

6.1.2. Постройте ОЯНРОВ-программы, распознающие следующие языки:

(а) языки из упр. 6.1.1,

(б) язык, порождаемый грамматикой (с начальным символом E)

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid I$$

$$I \rightarrow a \mid a(L)$$

$$L \rightarrow a \mid a, L$$

$$**(\text{в}) \quad \{a^{2^n} \mid n \geq 1\}.$$

*6.1.3. Покажите, что для каждого LL(1)-языка существует ОЯНРОВ-программа, распознающая его без возвратов, т. е. анализирующая машина, построенная в лемме 6.6, никогда не сдвигает входной указатель влево.

*6.1.4. Докажите неразрешимость следующих проблем: распознает ли ЯНРОВ-программа $P = (N, \Sigma, R, S)$

(а) пустое множество \emptyset ?

(б) язык Σ^* ?

6.1.5. Покажите, что каждая ЯНРОВ и ОЯНРОВ-программа эквивалента программе, в которой есть правило для каждого нетерминала. Указание: Покажите, что если для A правила нет, то можно добавить правило $A \rightarrow AA/A$ (или соответственно $A \rightarrow A[A, A]$), не изменив распознаваемого языка.

*6.1.6. Постройте обычную ЯНРОВ-программу, эквивалентную расширенной программе

$$S \rightarrow A/B/C$$

$$A \rightarrow a$$

$$B \rightarrow SCA$$

$$C \rightarrow c$$

Какой язык она определяет? Каковы недостатки этой программы с практической точки зрения?

6.1.7. Дайте формальное доказательство леммы 6.3.

6.1.8. Докажите лемму 6.4.

6.1.9. Дайте формальное доказательство того, что программа P из примера 6.5 определяет язык $\{0^n 1^n 2^n \mid n \geq 1\}$.

6.1.10. Дополните доказательство теоремы 6.2.

6.1.11. С помощью алгоритма 6.2 покажите, что цепочка $((a)) + a$ содержится в языке $L(P)$, где P — программа из примера 6.6.

6.1.12. ОЯНРОВ-операторы можно расширить тем же способом, что и ЯНРОВ-операторы. Например, можно разрешить ОЯНРОВ-операторы вида

$$A \rightarrow X_1 g_1 X_2 g_2 \dots X_k g_k$$

где каждый символ X_i — терминал или нетерминал, а каждый символ g_i — либо e , либо пара вида $[\alpha_i, \beta_i]$, в которой α_i и β_i — цепочки символов. Нетерминал A сообщает об успехе тогда и только тогда, когда каждая цепочка $X_i g_i$ приводят к успеху, а это определяется рекурсивно следующим образом. Цепочка $X_i [\alpha_i, \beta_i]$ приводит к успеху тогда и только тогда, когда

(1) X_i и α_i приводят к успеху

или

(2) X_i приводит к неудаче, а β_i — к успеху.

(а) Покажите, как заменить этот расширенный оператор эквивалентным множеством обычных ОЯНРОВ-операторов.

(б) Покажите, что каждый расширенный ЯНРОВ-оператор можно заменить эквивалентным множеством (расширенных) ОЯНРОВ-операторов.

6.1.13. Покажите, что существуют ЯНРОВ- и ОЯНРОВ-программы, для которых число операторов, выполняемых анализирующей машиной из леммы 6.6, экспоненциально зависит от длины входной цепочки.

6.1.14. Постройте ОЯНРОВ-программу, моделирующую действия правила $A \rightarrow BC/(D_1, D_2)$, упомянутого в конце разд. 6.1.3.

6.1.15. Найдите ОЯНРОВ-программу, определяющую язык $L(M)$, где M — анализирующая машина из примера 6.7.

6.1.16. Найдите анализирующие машины, распознающие языки из упр. 6.1.2.

Определение. ЯНРОВ- или ОЯНРОВ-программа $P = (N, \Sigma, R, S)$ приводит к неудаче с частичным распознаванием на цепочке w , если $w = uv$, причем $v \neq e$ и $S \Rightarrow^+ (u \uparrow v, s)$. Программа P вполне определена, если для каждой цепочки w из Σ^* либо $S \Rightarrow^+ (\uparrow w, f)$, либо $S \Rightarrow^+ (w \uparrow, s)$.

*6.1.17. Покажите, что если L — ЯНРОВ-язык (ОЯНРОВ-язык) и $\$$ — новый символ, то $-L\$$ определяется ЯНРОВ-программой (ОЯНРОВ-программой), не приводящей к неудаче с частичным распознаванием.

*6.1.18. Пусть L_1 определяется ЯНРОВ-программой (ОЯНРОВ-программой), а L_2 — вполне определенной программой того же типа. Покажите, что следующие языки являются ЯНРОВ-языками (ОЯНРОВ-языками):

- (а) $L_1 \cup L_2$,
- (б) \bar{L}_2 ,
- (в) $L_1 \cap L_2$,
- (г) $\bar{L}_1 - L_2$.

*6.1.19. Покажите, что каждая ОЯНРОВ-программа, не приводящая к неудаче с частичным распознаванием, эквивалентна некоторой вполне определенной ОЯНРОВ-программе. Указание: Достаточно обнаружить и устраниить „левую рекурсию“. Иными словами, если дана обычная ОЯНРОВ-программа, построим КС-грамматику, заменив правило вида $A \rightarrow B[C, D]$ правилами $A \rightarrow BC$ и $A \rightarrow D$ и сохранив правила $A \rightarrow a$ и $A \rightarrow e$. Имеется в виду „левая рекурсия“ в построенной КС-грамматике.

**6.1.20. Докажите, что проблема пустоты множества $L(P)$ для вполне определенных ЯНРОВ-программ P неразрешима. Замечание: Естественное моделирование проблемы соответствий Поста позволяет сделать упр. 6.1.4(a), но не всегда дает вполне определенную программу.

6.1.21. Дополните доказательство леммы 6.5.

6.1.22. Докажите лемму 6.6.

Открытые проблемы

6.1.23. Существует ли КС-язык, который не является ОЯНРОВ-языком?

6.1.24. Замкнут ли класс ЯНРОВ-языков относительно дополнения?

6.1.25. Эквивалента ли каждая ЯНРОВ-программа некоторой ЯНРОВ-программе? Мы выдвигаем гипотезу, что ОЯНРОВ-язык $\{\alpha^n \mid n \geq 1\}$ не является ЯНРОВ-языком.

Упражнения на программирование

6.1.27. Постройте интерпретатор для анализирующих машин. Напишите программу, которая по расширенной ОЯНРОВ-про-

граммме строит эквивалентную анализирующую машину, моделируемую далее интерпретатором.

6.1.28. На основе обобщенного или обычного языка исходящего разбора с ограниченными возвратами разработайте язык программирования, который можно использовать для задания трансляторов. Постройте компилятор для этого языка. Исходной программой для него будет описание транслятора, а объектной программой — фактический транслятор.

Замечания по литературе

ЯНРОВ — это абстракция языка, использованного Мак-Клюром [1965] в его компиляторе компиляторов ТМГ. Анализирующая машина из разд. 6.1.5 аналогична машине, рассмотренной Кнутом [1967, 1971]. Большинство излагаемых здесь теоретических результатов, связанных с ЯНРОВ и ОЯНРОВ, получено Бирманом и Ульманом [1970, 1973]. В их работах можно найти решения многих упражнений.

ОЯНРОВ — это модель семейства компиляторов компиляторов типа МЕТА [Шорре, 1964] и др.

6.2. ВОСХОДЯЩИЙ РАЗБОР С ОГРАНИЧЕННЫМИ ВОЗВРАТАМИ

Рассмотрим возможности восходящего детерминированного анализа, если допускается большая свобода в выборе средств. В частности, разрешим ограниченные возвраты на входе и не будем требовать, чтобы разбор был обязательно правым. Главный обсуждаемый здесь метод — алгоритм Колмерауэра, основанный на отношениях предшествования.

6.2.1. Неканонический разбор

Существует несколько приемов, с помощью которых можно провести детерминированный разбор для грамматик, не являющихся LR-грамматиками. Один из них заключается в том, чтобы разрешить заглядывать вперед на любое число символов, позволяя входному указателю передвигаться по входной цепочке вперед для разрешения встретившейся неопределенности. После того как решение принято, входной указатель находит обратный путь в подходящее для свертки место.

Пример 6.8. Рассмотрим грамматику G с правилами

$$\begin{aligned} S &\rightarrow Aa \mid Bb \\ A &\rightarrow 0A1 \mid 01 \\ B &\rightarrow 0B11 \mid 011 \end{aligned}$$

Эта грамматика порождает язык $\{0^n 1^n a \mid n \geq 1\} \cup \{0^n 1^{n+1} b \mid n \geq 1\}$, который не является детерминированным КС-языком. Однако разбор для грамматики G можно провести, сначала передвинув

входной указатель к концу входной цепочки, чтобы посмотреть, каков последний символ — a или b , а затем, вернувшись к началу цепочки и продолжив разбор для цепочки $0^n 1^n$ или $0^n 1^{n+1}$ соответственно. \square

Другой способ — свертывать такие составляющие выводимой цепочки, которые не являются основами.

Определение. Если $G = (N, \Sigma, P, S)$ — КС-грамматика и в ней есть вывод $S \Rightarrow^* aA\gamma \Rightarrow a\beta\gamma$, то цепочка β называется *составляющей* выводимой цепочки $a\beta\gamma$. Пусть $X_1 \dots X_k$ и $X_j \dots X_l$ — составляющие выводимой цепочки $X_1 \dots X_n$; будем говорить, что составляющая $X_i \dots X_k$ расположена левее составляющей $X_j \dots X_l$, если $i < j$ либо $i = j$ и $k < l$. Таким образом, если грамматика однозначная, то основа — самая левая составляющая правовыводимой цепочки.

Пример 6.9. Рассмотрим грамматику G с правилами

$$\begin{aligned} S &\rightarrow 0ABb \mid 0aBc \\ A &\rightarrow a \\ B &\rightarrow B1 \mid 1 \end{aligned}$$

$L(G)$ — это регулярное множество $0a1^+(b+c)$, но G — не LR-грамматика. Однако для G возможен восходящий разбор, если решение вопроса о том, является ли a составляющей, отложить до тех пор, пока не прочтем последний входной символ. Иными словами, входную цепочку вида $0a1^n$ можно свернуть к $0aB$ независимо от того, следует за ней b или c . В первом случае цепочка $0ABb$ сначала свертывается к $0ABb$, а затем к S . Во втором случае цепочка $0aBc$ сразу свертывается к S . При этом, конечно, мы не получим ни левого, ни правого разбора. \square

Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика, правила которой за- нумерованы числами от 1 до p , и пусть

$$(6.2.1) \quad S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n = w$$

— вывод цепочки w из S . Для $0 \leq i < n$ пусть $\alpha_i = \beta_i A_i \delta_i$ и $A_i \rightarrow \gamma_i$ — правило с номером p_i , которое применялось к явно указанному вхождению A_i в α_i и дало $\alpha_{i+1} = \beta_i \gamma_i \delta_i$. Шаг вывода $\alpha_i \Rightarrow \alpha_{i+1}$ можно представить парой чисел (p_i, l_i) , где $l_i = |\beta_i|$. Таким образом, вывод (6.2.1) можно представить цепочкой, состоящей из n пар чисел

$$(6.2.2) \quad (p_0, l_0) (p_1, l_1) \dots (p_{n-1}, l_{n-1})$$

Если вывод правый или левый, то вторые компоненты пар цепочки (6.2.2), указывающие позицию нетерминала, развертываемого на очередном шаге вывода, можно не писать.

Определение. Назовем цепочку пар вида (6.2.2) (обобщенным) нисходящим разбором цепочки w . Ясно, что левый разбор—частный случай нисходящего разбора. Аналогично назовем обращение цепочки вида (6.2.2), т. е. цепочку

$$(p_{n-1}, l_{n-1}) (p_{n-2}, l_{n-2}) \dots (p_1, l_1) (p_0, l_0)$$

(обобщенным) восходящим разбором цепочки w . Таким образом, правый разбор—частный случай восходящего разбора.

Если ослабить ограничение на просмотр входной цепочки, требующее, чтобы она читалась только слева направо, и допустить возвраты на входе, то детерминированный разбор становится возможным для таких грамматик, которые нельзя анализировать, просматривая вход только слева направо.

6.2.2. Анализаторы с двумя магазинами

В качестве модели некоторых алгоритмов с возвратами мы введем автомат с двумя магазинами, причем второй магазин выступает также в роли входной ленты. Детерминированная версия этого устройства родственна анализатору с двумя магазинами, который применялся в алгоритмах 4.1 и 4.2 для общего нисходящего и восходящего анализа. Мы наложим на это устройство ограничения, благодаря которым оно будет вести себя как восходящий анализатор, использующий предшествование.

Определение. Двухмагазинным (восходящим) анализатором для грамматики $G = (N, \Sigma, P, S)$ назовем конечное множество правил вида $(\alpha, \beta) \rightarrow (\gamma, \delta)$, где α, β, γ и δ —цепочки символов из $N \cup \Sigma \cup \{ \$ \}$, причем $\$$ —новый символ, концевой маркер. Каждое правило $(\alpha, \beta) \rightarrow (\gamma, \delta)$ должно иметь один из двух видов: либо

- (1) $\beta = X\delta$ для некоторого $X \in N \cup \Sigma$ и $\gamma = \alpha X$, либо
- (2) $\alpha = \gamma e$ для некоторого $e \in (N \cup \Sigma)^*$, $\delta = A\beta$ и $A \rightarrow e$ —правило из P .

В общем случае правило $(\alpha, \beta) \rightarrow (\gamma, \delta)$ говорит о том, что если α —верхняя цепочка первого магазина и β —верхняя цепочка второго магазина, то можно заменить α на γ в первом магазине и β на δ во втором. Правило типа (1) соответствует переносу в алгоритме, анализирующем методом „перенос—свертка“. Правило типа (2) соответствует шагу свертки. Существенное различие состоит в том, что символ A , являющийся левой частью применяемого правила грамматики, помещается на верх второго магазина, а не первого. Это соглашение илагает ограничения на возвраты. Можно перемещать символы из первого магазина во второй (который функционирует как входная лента), но только

в моменты сверток. Разумеется, правила типа (1) позволяют символам в любой момент перемещаться из второго магазина в первый.

Конфигурацией двухмагазинного анализатора T называется тройка (α, β, π) , где $\alpha \in \$ (N \cup \Sigma)^*$, $\beta \in (N \cup \Sigma)^* \$$, а π —цепочка пар, состоящих из целого числа и номера правила. Таким образом, π может быть частью разбора некоторой цепочки из $L(G)$. Мы пишем $(\alpha, \beta, \pi) \vdash_T (\alpha', \beta', \pi')$, если

(1) $\alpha = \alpha_1 \alpha_2$, $\beta = \beta_2 \beta_1$, $(\alpha_2, \beta_2) \rightarrow (\gamma, \delta)$ —правило анализатора T ;

(2) $\alpha' = \alpha_1 \gamma$, $\beta' = \delta \beta_1$;

(3) $\pi' = \pi$, если $(\alpha_2, \beta_2) \rightarrow (\gamma, \delta)$ —правило типа 1; если это—правило типа 2 и применяется i -е правило грамматики, то $\pi' = \pi(i, j)$, где $j = |\alpha'| - 1$ ¹⁾.

Заметим, что верх первого магазина расположен справа, а второго магазина—слева.

Обычным образом по отношению \vdash_T определим отношения \vdash_t , $\vdash_{\tilde{t}}$ и $\vdash_{\hat{t}}$. Когда можно, будем опускать индекс T .

Переводом, определяемым анализатором T , назовем множество $\tau(T) = \{(w, \pi) | (\$, w \$, e) \vdash^* (\$, \$ \$, \pi)\}$. Назовем анализатор T корректным для грамматики G , если для каждой цепочки $w \in L(G)$ найдется такой ее восходящий разбор π , что $(w, \pi) \in \tau(T)$. Можно элементарно показать, что если $(w, \pi) \in \tau(T)$, то π —восходящий разбор цепочки w .

Анализатор T назовем детерминированным, если выполнено следующее условие. Пусть $(\alpha_1, \beta_1) \rightarrow (\gamma_1, \delta_1)$ и $(\alpha_2, \beta_2) \rightarrow (\gamma_2, \delta_2)$ —такие правила, что одна из цепочек α_1, α_2 является суффиксом другой, а одна из цепочек β_1, β_2 является префиксом другой; тогда $\gamma_1 = \gamma_2$ и $\delta_1 = \delta_2$. Таким образом, для каждой конфигурации C существует не более одной такой конфигурации C' , что $C \vdash C'$.

Пример 6.10. Рассмотрим грамматику G с правилами

- (1) $S \rightarrow aSA$
- (2) $S \rightarrow bSA$
- (3) $S \rightarrow b$
- (4) $A \rightarrow a$

Эта грамматика порождает недетерминированный КС-язык $\{wba^n | w \in (a+b)^* \text{ и } n = |w|\}$.

Можно построить (недетерминированный) двухмагазинный преобразователь, который будет анализировать цепочки в соот-

¹⁾ Единица вычитается потому, что цепочка α' содержит концевой маркер.

ветствии с грамматикой G , сначала помещая всю входную цепочку в первый магазин, а затем разбирая ее по существу справа налево.

Правила анализатора T таковы:

- (1) $(e, X) \rightarrow (X, e)$ для всех $X \in \{a, b, S, A\}$ (любой символ можно перенести из второго магазина в первый),
- (2) $(a, e) \rightarrow (e, A)$ (a можно свернуть к A),
- (3) $(b, e) \rightarrow (e, S)$ (b можно свернуть к S),
- (4) $(aSA, e) \rightarrow (e, S)$,
- (5) $(bSA, e) \rightarrow (e, S)$.

(Последние два правила анализатора допускают свертки, соответствующие правилам (1) и (2) грамматики G .)

Заметим, что анализатор T недетерминированный и для каждой входной цепочки можно получить много разборов. Один из восходящих разборов цепочки $abbaa$ представляется такой последовательностью конфигураций:

$$\begin{aligned}
 (\$, abbaa\$, e) &\vdash (\$a, bbaa\$, e) \\
 &\vdash (\$ab, baa\$, e) \\
 &\vdash (\$abb, aa\$, e) \\
 &\vdash (\$ab, Saa\$, (3, 2)) \\
 &\vdash (\$abS, aa\$, (3, 2)) \\
 &\vdash (\$abSa, a\$, (3, 2)) \\
 &\vdash (\$abSaa, \$, (3, 2)) \\
 &\vdash (\$abSa, A\$, (3, 2)(4, 4)) \\
 &\vdash (\$abS, AA\$, (3, 2)(4, 4)(4, 3)) \\
 &\vdash (\$abSA, A\$, (3, 2)(4, 4)(4, 3)) \\
 &\vdash (\$a, SA\$, (3, 2)(4, 4)(4, 3)(2, 1)) \\
 &\vdash (\$aS, A\$, (3, 2)(4, 4)(4, 3)(2, 1)) \\
 &\vdash (\$aSA, \$, (3, 2)(4, 4)(4, 3)(2, 1)) \\
 &\vdash (\$, S\$, (3, 2)(4, 4)(4, 3)(2, 1)(1, 0))
 \end{aligned}$$

Цепочка $(3, 2)(4, 4)(4, 3)(2, 1)(1, 0)$ образует восходящий разбор цепочки $abbaa$, соответствующий выводу

$$S \Rightarrow aSA \Rightarrow abSAA \Rightarrow abSaA \Rightarrow abbaa \quad \square$$

Двухмагазинный анализатор имеет особенность, отличающую его от обычных алгоритмов типа „перенос—свертка“. Может оказаться, что такой анализатор работает и для неоднозначной грамматики, игнорируя некоторые из возможных в ней выводов.

Пример 6.11. Пусть G определяется правилами

$$\begin{aligned}
 S &\rightarrow A \mid B \\
 A &\rightarrow aA \mid a \\
 B &\rightarrow Ba \mid a
 \end{aligned}$$

G —неоднозначная грамматика для языка a^+ . Если игнорировать нетерминал B и B -правила, то детерминированный двухмагазин-

ный анализатор для G образуется из правил

$$\begin{aligned}
 (e, a) &\rightarrow (a, e) \\
 (a, \$) &\rightarrow (e, A\$) \\
 (a, A) &\rightarrow (aA, e) \\
 (aA, \$) &\rightarrow (e, A\$) \\
 (\$, A) &\rightarrow (\$A, e) \\
 (\$A, \$) &\rightarrow (\$, S\$) \quad \square
 \end{aligned}$$

6.2.3. Отношения предшествования Колмерауэра

Двухмагазинный анализатор можно устроить так, чтобы способ его работы был отчасти похож на разбор методом предшествования. Для этого надо предположить, что существуют три непересекающихся отношения \ll , \equiv и \gg , заданные на символах грамматики, из которых \gg означает, что надо сделать свертку, а \ll и \equiv —перенос. Когда делается свертка, \ll указывает левый конец некоторой составляющей (не обязательно основы). Следует подчеркнуть, что отношения \ll , \equiv и \gg по крайней мере временно не предполагаются связанными с правилами грамматики. Поэтому, например, может быть $X \equiv Y$, даже если X и Y не стоят рядом ни в какой правой части правила.

Определение. Пусть $G = (N, \Sigma, P, S)$ —КС-грамматика и \ll , \equiv , \gg —три непересекающихся отношения на $N \cup \Sigma \cup \{\$\}$, где $\$$ —новый символ (концевой маркер). Двухмагазинный анализатор, индуцированный отношениями \ll , \equiv и \gg , определяется правилами

(1) $(X, Y) \rightarrow (XY, e)$ тогда и только тогда, когда $X \ll Y$ или $X \equiv Y$,

(2) $(XZ_1 \dots Z_k, Y) \rightarrow (X, AY)$ тогда и только тогда, когда $Z_k \gg Y$, $Z_i \equiv Z_{i+1}$ для $1 \leq i < k$, $X \ll Z_1$ и $A \rightarrow Z_1 \dots Z_k$ —правило грамматики G .

Заметим, что если G —обратимая грамматика, то индуцированный двухмагазинный анализатор будет детерминированным, и обратно.

Пример 6.12. Пусть G —грамматика с правилами

$$\begin{aligned}
 (1) \quad S &\rightarrow aSA \\
 (2) \quad S &\rightarrow bSA \\
 (3) \quad S &\rightarrow b \\
 (4) \quad A &\rightarrow a
 \end{aligned}$$

как в примере 6.10.

Зададим отношения \ll , \equiv и \gg таблицей на рис. 6.3.

Эти отношения индуцируют двухмагазинный преобразователь, правила которого определяются так:

$$\begin{aligned} (X, Y) &\rightarrow (XY, e) \text{ для всех } X \in \{\$, a, b\}, Y \in \{a, b\} \\ (Xa, Y) &\rightarrow (X, AY) \text{ для всех } X \in \{\$, a, b\}, Y \in \{\$, A\} \\ (Xb, Y) &\rightarrow (X, SY) \text{ для всех } X \in \{\$, a, b\}, Y \in \{\$, A\} \\ (X, S) &\rightarrow (XS, e) \text{ для всех } X \in \{a, b\} \\ (S, A) &\rightarrow (SA, e) \\ (XaSA, Y) &\rightarrow (X, SY) \text{ для всех } X \in \{\$, a, b\} \text{ и } Y \in \{A, \$\} \\ (XbSA, Y) &\rightarrow (X, SY) \text{ для всех } X \in \{\$, a, b\} \text{ и } Y \in \{A, \$\} \end{aligned}$$

T допускает цепочку wba^n , у которой $|w|=n$, проделав последовательность тактов

$$\begin{aligned} (\$, wba^n\$, e) &\vdash^{2n+1} (\$wba^n, \$, e) \\ &\vdash^n (\$wb, A^n\$, (4, 2n) \dots (4, n+1)) \\ &\vdash (\$w, SA^n\$, (4, 2n) \dots (4, n+1)(3, n)) \\ &\vdash^{2n} (\$, S\$, (4, 2n) \dots (4, n+1)(3, n)(i_n, n-1) \dots (i_1, 0)) \end{aligned}$$

где $i_j \in \{1, 2\}$ для $1 \leq j \leq n$. Заметим, что на последних $3n$ тактах чередуются переносы S и A со свертками aSA либо bSA к S .

Легко проверить, что преобразователь T детерминированный. Поэтому для каждой цепочки из $L(G)$ возможна только одна

	<i>a</i>	<i>b</i>	<i>s</i>	<i>A</i>	<i>\$</i>
<i>\$</i>	<	<			
<i>a</i>	<	<	=	>	>
<i>b</i>	<	<	=	>	>
<i>s</i>				=	
<i>A</i>				>	>

Рис. 6.3. Отношения „предшествования“.

последовательность тактов. Так как все сверки преобразователя T соответствуют правилам грамматики G , то T — двухмагазинный анализатор для G . \square

В некоторых грамматиках можно так задать отношения “предшествования”, что индуцируемые ими двухмагазинные анализаторы будут одновременно детерминированными и корректными. Сейчас мы дадим соответствующее определение, а в следующем разделе опишем простой тест, позволяющий установить, обладает ли грамматика нужным анализатором.

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Назовем G грамматикой Колмерауэра, если

(1) она однозначная,

(2) приведенная,

(3) существуют непересекающиеся отношения \triangleleft , \triangleq и \triangleright на $N \cup \Sigma \cup \{\$\}$, индуцирующие двухмагазинный анализатор, корректный для G .

Эти отношения назовем *отношениями предшествования Колмерауэра*. Заметим, что из условия (3) следует, что грамматика Колмерауэра должна быть обратимой.

Пример 6.13. Отношения на рис. 6.3 являются отношениями предшествования Колмерауэра, и, значит, грамматики из примеров 6.10 и 6.12 — грамматики Колмерауэра. \square

Пример 6.14. Каждая грамматика простого предшествования является грамматикой Колмерауэра. Пусть \triangleleft , \triangleq и \triangleright — отношения предшествования Вирта — Вебера для грамматики $G = (N, \Sigma, P, S)$. Если G — грамматика простого предшествования, то она по определению приведенная и однозначная. Индуцируемый ею двухмагазинный анализатор действует почти как анализатор типа “перенос — свертка” для грамматик предшествования.

Однако, когда делается свертка правовыводимой цепочки $a\$A\$$ к цепочке $aA\$$, мы затачиваем $\$a$ в первый магазин, а $A\$$ во второй, тогда как при разборе методом предшествования $\$a\$$ находится в магазине, а $\$$ — на входе. Если X — последний символ цепочки $\$a$, то по теореме 5.14 либо $X \triangleleft A$, либо $X \triangleq A$. Поэтому на очередном шаге двухмагазинный анализатор должен перенести A в первый магазин. Затем до следующей свертки двухмагазинный анализатор работает как анализатор простого предшествования.

Заметим, что если отношения предшествования Колмерауэра являются отношениями Вирта — Вебера, то индуцированный двухмагазинный анализатор дает правые разборы. Однако в общем случае этого ожидать нельзя. \square

6.2.4. Проверка условий предшествования Колмерауэра

Дадим теперь условие, необходимое и достаточное для того, чтобы однозначная приведенная грамматика была грамматикой Колмерауэра. Оно включает три отношения, определяемые ниже. Следует напомнить, что проблема выяснения однозначности КС-грамматики неразрешима и что, как мы видели в примере 6.12, существуют неоднозначные грамматики, обладающие детерминированными двухмагазинными анализаторами. Таким образом,

нельзя установить, является ли произвольная КС-грамматика грамматикой Колмерауэра, не зная заранее, что она однозначная.

Определение. Пусть $G = (N, \Sigma, P, S)$ — КС-грамматика. Определим три новых отношения λ („слева“), μ („смежность“) и ρ („справа“) на $N \cup \Sigma$. Для всех $X, Y \in N \cup \Sigma$ и $A \in N$ положим

- (1) $A\lambda Y$, если в P есть правило $A \rightarrow Y\alpha$,
- (2) $X\mu Y$, если в P есть правило $A \rightarrow \alpha XY\beta$,
- (3) $X\rho A$, если в P есть правило $A \rightarrow \alpha X$.

Как обычно, для каждого отношения R обозначим $R^+ = \bigcup_{i=1}^{\infty} R^i$

и $R^* = \bigcup_{i=0}^{\infty} R^i$. Напомним, что R^+ и R^* удобно вычислять с помощью алгоритма 0.2.

Заметим, что отношения предшествования Вирта—Вебера \triangleleft , \equiv и \triangleright на $N \cup \Sigma$ можно определить через λ , μ и ρ :

- (1) $\triangleleft = \mu\lambda^+$,
- (2) $\equiv = \mu$,
- (3) $\triangleright = \rho^+\mu\lambda^* \cap (N \cup \Sigma) \times \Sigma$.

Остальную часть этого раздела посвятим доказательству того, что однозначная приведенная КС-грамматика G обладает отношениями предшествования Колмерауэра тогда и только тогда, когда

- (1) $\rho^+\mu \cap \mu\lambda^* = \emptyset$,
- (2) $\mu \cap \rho^*\mu\lambda^+ = \emptyset$.

Пример 6.15. Рассмотрим предыдущую грамматику

$$\begin{aligned} S &\rightarrow aSA \mid bSA \mid b \\ A &\rightarrow a \end{aligned}$$

Для нее

$$\lambda = \{(S, a), (S, b), (A, a)\}$$

$$\mu = \{(a, S), (S, A), (b, S)\}$$

$$\rho = \{(A, S), (b, S), (a, A)\}$$

$$\rho^+\mu = \{(A, A), (b, A), (a, A)\}$$

$$\mu\lambda^* = \{(a, S), (S, A), (b, S), (a, a), (a, b), (S, a), (b, a), (b, b)\}$$

$$\rho^*\mu\lambda^+ = \{(a, a), (a, b), (b, a), (b, b), (S, a), (A, a)\}$$

Так как $\rho^+\mu \cap \mu\lambda^* = \emptyset$ и $\rho^*\mu\lambda^+ \cap \mu = \emptyset$, эта грамматика обладает отношениями предшествования Колмерауэра; мы их уже видели на рис. 4.3. \square

Покажем теперь, что если грамматика G содержит такие символы X и Y , что $X\mu Y$ и $X\rho^*\mu\lambda^+ Y$, то она не может быть

грамматикой Колмерауэра. Здесь X и Y не обязаны быть различными.

Лемма 6.7. Пусть $G = (N, \Sigma, P, S)$ — грамматика Колмерауэра с отношениями предшествования Колмерауэра \triangleleft , \equiv и \triangleright . Если $X\mu Y$, то $X \equiv Y$.

Доказательство. Так как грамматика G приведенная, существует вывод, в котором применяется правило $A \rightarrow \alpha XY\beta$. При разборе цепочки $w \in L(G)$, в выводе которой встречается это правило, цепочка $\alpha XY\beta$ в какой-то момент окажется на верху первого магазина и свернется к A . Это может произойти, только если $X \equiv Y$. \square

Лемма 6.8. Пусть $G = (N, \Sigma, P, S)$ — такая КС-грамматика, что $X\rho^*\mu\lambda^+ Y$ и $X\mu Y$ для некоторых X и Y из $N \cup \Sigma$. Тогда G не является грамматикой Колмерауэра.

Доказательство. Предположим, что G — грамматика Колмерауэра. Пусть \triangleleft , \equiv и \triangleright — отношения Колмерауэра для этой грамматики и T — индуцированный ими двухмагазинный анализатор. Так как G предполагается приведенной, найдутся такие x и y из Σ^* , что $X \Rightarrow^* x$ и $Y \Rightarrow^* y$. Так как $X\mu Y$, существуют правило $A \rightarrow \alpha XY\beta$ и такие цепочки w_1, w_2, w_3 и w_4 из Σ^* , что $S \Rightarrow^* w_1 Aw_4 \Rightarrow^* w_1 \alpha XY\beta w_4 \Rightarrow^* w_1 w_2 XYw_3 w_4 \Rightarrow^* w_1 w_2 xyw_3 w_4$. Так как $X\rho^*\mu\lambda^+ Y$, существует такое правило $B \rightarrow \gamma ZC\delta$, что $Z \Rightarrow^* \gamma' X$, $C \Rightarrow^* Y\delta'$, и для некоторых z_1, z_2, z_3 и z_4

$$\begin{aligned} S &\Rightarrow^* z_1 B z_4 \Rightarrow z_1 \gamma Z C \delta z_4 \Rightarrow^* z_1 \gamma' \gamma' X Y \delta' \delta z_4 \\ &\Rightarrow^* z_1 z_2 X Y z_3 z_4 \Rightarrow^* z_1 z_2 x y z_3 z_4 \end{aligned}$$

По лемме 6.7 можно считать, что $X \equiv Y$. Посмотрим, как анализатор T обрабатывает две цепочки $u = w_1 w_2 x y w_3 w_4$ и $v = z_1 z_2 x y z_3 z_4$. В частности, проследим за цепочками, к которым свертываются x и y в каждом разборе, и выясним, появляются эти цепочки в первом магазине, во втором или же они как-то разделены по магазинам. Пусть $\theta_1, \theta_2, \dots$ — последовательность цепочек, к которым свертывается xy в цепочке u , ψ_1, ψ_2, \dots — аналогичная последовательность для цепочки v . Мы знаем, что находится такое число j , что $\theta_i = XY$, так как в силу предположенной однозначности грамматики X и Y при свертке цепочки u должны свертываться одновременно. Если $\psi_i = \theta_i$ для $1 \leq i \leq j$, то в тот момент, когда Y подвергается свертке при обработке цепочки v , стоящий слева X тоже будет свернут, так как $X \equiv Y$. Но этого не может быть, так как $C \Rightarrow^* Y\delta'$ для некоторого δ' в выводе цепочки v ¹⁾.

¹⁾ Заметим, что здесь X и Y обозначают определенные входления этих символов в выводы, т. е. конкретные вершины дерева вывода. Мы надеемся, будет ясно, что именно имеется в виду.

Поэтому предположим, что для какого-то наименьшего $1 < i \leq j$ либо $\theta_i \neq \psi_i$, либо ψ_i не существует (потому что следующая свертка символа из ψ_i затрагивает также символ вне ψ_i). Мы знаем, что если $i > 2$, то точка, делящая цепочку на две части, расположенные в разных магазинах (назовем ее точкой разделения по магазинам), занимает одну и ту же позицию в θ_{i-1} и в ψ_{i-1} после того, как θ_{i-1} и ψ_{i-1} построены в результате свертки. Следовательно, если точка разделения по магазинам вышла из θ_{i-1} до того, как построена цепочка θ_i , то она вышла и из ψ_{i-1} , причем в том же направлении. Принимая во внимание случай $i = 2$, в котором $\theta_1 = \psi_1 = xy$, получаем, что перед построением θ_i и ψ_i точка разделения по магазинам находится либо

(1) внутри θ_i и ψ_i (и в обеих цепочках в одном и том же месте), т. е. цепочки θ_{i-1} и ψ_{i-1} захватывают оба магазина, либо

(2) справа от θ_{i-1} и ψ_{i-1} , т. е. обе цепочки находятся в первом магазине.

Заметим, что не может быть так, чтобы точка разделения находилась слева от θ_{i-1} и ψ_{i-1} и все же на следующем такте эти цепочки изменились. Кроме того, число тактов между появлением θ_{i-1} и θ_i не может быть таким же, как между появлением ψ_{i-1} и ψ_i . Нас не заботит время, которое точка разделения проводит вне этих подцепочек: цепочки могут изменяться только тогда, когда точка разделения возвращается в них.

Отсюда следует, что так как $\theta_i \neq \psi_i$, первая свертка, затрагивающая символ из ψ_{i-1} , должна также затрагивать хотя бы один символ вне ψ_{i-1} , т. е. ψ_i на самом деле не существует, ибо по определению цепочек $\theta_1, \theta_2, \dots$ свертка θ_{i-1} к θ_i затрагивает только символы из θ_{i-1} . Если бы очередная свертка, затрагивающая ψ_{i-1} , происходила целиком внутри ψ_{i-1} , то, согласно сделанным выше замечаниям (1) и (2), цепочка ψ_{i-1} в результате должна была бы свернуться к θ_i .

Исследуем отдельно случаи, зависящие от того, была ли в θ_{i-1} свернута к X подцепочка x и/или к Y подцепочка y .

Случай 1: Сделаны обе свертки. Это невозможно, так как мы взяли $i \leq j$.

Случай 2: Подцепочка y не была свернута к Y , x была свернута к X . Здесь свертка цепочки ψ_{i-1} затрагивает символы в ψ_{i-1} и вне ψ_{i-1} . Следовательно, точка разделения по магазинам находится внутри θ_{i-1} и ψ_{i-1} , и свертывается некоторый префикс обеих цепочек. Таким образом, для входной цепочки u анализатор T свертывает X раньше Y . Так как, по предположению, T — корректный анализатор, то u имеет два разных дерева разбора, и, значит, грамматика G неоднозначна. На самом деле она однозначна, так что этот случай тоже можно отбросить.

Случай 3: Подцепочка x не была свернута к X , а y была свернута к Y . Тогда $\theta_{i-1} = \theta Y$ для некоторой цепочки θ . Надо рассмотреть положение точки разделения по магазинам в двух подслучаях:

(а) Если точка разделения лежит внутри θ_{i-1} и, значит, внутри ψ_{i-1} , то ψ_i может отличаться от θ_i только тогда, когда при свертке цепочки θ_{i-1} свертывается ее префикс, а символ слева от θ_{i-1} находится в отношении \ll с самым левым символом цепочки θ_{i-1} . Однако для ψ_{i-1} выполняется отношение \sqsubseteq , так что происходит другая свертка. Но тогда некоторые символы из ψ_{i-1} , которые уже должны быть свернуты к X , свертываются вместе с некоторыми символами вне ψ_{i-1} . Эта возможность исключается тем же рассуждением, что в случае 2.

(б) Если точка разделения лежит справа от θ_{i-1} , то префикс θ может оказаться на верху первого магазина, только если Y подвергается свертке, так как единственный способ уменьшить длину первого магазина — свернуть его верхние символы. Но тогда при обработке u к тому моменту, когда x свертывается к X , Y уже будет свернут. Но X и Y должны свертываться одновременно в единственном дереве вывода цепочки u . Итак, в этом случае мы тоже вынуждены заключить, что либо T не корректный анализатор, либо G не однозначная грамматика.

Случай 4: x не свертывается к X и y не свертывается к Y . Здесь надо применить одно из рассуждений, использованных в случаях 2 и 3.

Таким образом, мы исключили все возможности и можем сделать вывод, что для грамматики Колмерауэра множество $\mu P^+ \mu \lambda^+$ должно быть пустым. \square

Покажем, что если в КС-грамматике G есть символы X и Y , для которых $X P^+ \mu Y$ и $X \mu \lambda^* Y$, то она не может быть грамматикой Колмерауэра.

Лемма 6.9. Пусть $G = (N, \Sigma, P, S)$ — такая КС-грамматика, что $X P^+ \mu Y$ и $X \mu \lambda^* Y$ для некоторых X и Y из $N \cup \Sigma$. Тогда G не является грамматикой Колмерауэра.

Доказательство. Доказательство, аналогичное, к сожалению, доказательству леммы 6.8, оставляется в качестве упражнения. Так как $X P^+ \mu Y$, в P найдется такое правило $A \rightarrow \alpha Z Y \beta$, что $Z \Rightarrow^+ \alpha' X$. Так как $X \mu \lambda^* Y$, в P найдется такое правило $B \rightarrow \gamma X C \delta$, что $C \Rightarrow^* Y \delta'$. Поскольку грамматика G приведенная, можно найти такие цепочки u и v из языка $L(G)$, что каждый вывод цепочки u включает правило $A \rightarrow \alpha Z Y \beta$ и вывод цепочки $\alpha' X$ из Z , а каждый вывод цепочки v включает правило $B \rightarrow \gamma X C \delta$ и вывод цепочки $Y \delta'$ из C . В любом случае из X выводится какая-то цепочка $x \in \Sigma^*$ и из Y — какая-то цепочка $y \in \Sigma^*$.

Как и в лемме 6.8, надо следить за тем, что произойдет с подцепочкой xy цепочек u и v . Для u окажется, что X придется свернуть раньше, чем Y , а при обработке v либо X и Y свертываются одновременно (если $C \Rightarrow^* Y\delta'$ — тривиальный вывод), либо Y свертывается раньше X (если $C \Rightarrow^+ Y\delta'$). С помощью рассуждений, аналогичных проведенным в лемме 6.8, можно доказать, что если цепочки, к которым свертывается xy в цепочках u и v , различны, то либо первый, либо второй вывод не может идти правильно. \square

Итак, условия $\mu \cap \rho^* \mu \lambda^+ = \emptyset$ и $\rho^+ \mu \cap \mu \lambda^* = \emptyset$ необходимы для того, чтобы G была грамматикой Колмерауэра. Переидем теперь к доказательству того, что вместе с однозначностью, приведенностью и обратимостью эти условия также и достаточны.

Лемма 6.10. Пусть $G = (N, \Sigma, P, S)$ — приведенная КС-грамматика. Тогда если $\alpha X Y \beta$ — любая выводимая цепочка этой грамматики, то $X \rho^* \mu \lambda^* Y$.

Доказательство. Элементарная индукция по длине вывода цепочки $\alpha X Y \beta$. \square

Лемма 6.11. Пусть $G = (N, \Sigma, P, S)$ — однозначная приведенная КС-грамматика, причем $\mu \cap \rho^* \mu \lambda^+ = \emptyset$ и $\rho^+ \mu \cap \mu \lambda^* = \emptyset$. Если $\alpha Y X_1 \dots X_k Z \beta$ — выводимая цепочка этой грамматики, то из условий $X_1 \mu X_2, \dots, X_{k-1} \mu X_k, Y \rho^* \mu \lambda^* X_1$ и $X_k \rho^+ \mu \lambda^* Z$ следует, что $X_1 \dots X_k$ — составляющая цепочки $\alpha Y X_1 \dots X_k Z \beta$.

Доказательство. Если бы это было не так, то в $\alpha Y X_1 \dots X_k Z \beta$ нашлась бы другая составляющая, содержащая X_1 .

Случай 1: Допустим, что все символы X_2, \dots, X_k входят в эту другую составляющую. Тогда в нее входит либо Y , либо Z , так как она не совпадает с $X_1 \dots X_k$. Пусть входит Y . Тогда $Y \mu X_1$. Но $Y \rho^* \mu \lambda^* X_i$ и потому $\mu \cap \rho^* \mu \lambda^+ \neq \emptyset$. Если входит Z , то $X_k \mu Z$. Но по условию также $X_k \rho^+ \mu \lambda^* Z$. Если λ^* содержит хотя бы одно вхождение λ , т. е. $X_k \rho^+ \mu \lambda^* Z$, то $\mu \cap \rho^* \mu \lambda^+ \neq \emptyset$. Если λ^* не содержит ни одного вхождения λ , то $X_k \rho^+ \mu Z$. Так как $X_k \mu Z$, то $X_k \mu \lambda^* Z$ и потому $\rho^+ \mu \cap \mu \lambda^* \neq \emptyset$.

Случай 2: X_i для некоторого $1 \leq i < k$ входит в эту составляющую, а X_{i+1} не входит. Пусть эта составляющая свертывается к A . Тогда, применяя лемму 6.10 к выводимой цепочке, к которой свертывается $\alpha Y X_1 \dots X_k Z \beta$, получаем $A \rho^* \mu \lambda^* X_{i+1}$ и, значит, $X_i \rho^+ \mu \lambda^* X_{i+1}$. Но уже было $X_i \mu X_{i+1}$, так что либо $\mu \cap \rho^* \mu \lambda^+ \neq \emptyset$, либо $\rho^+ \mu \cap \mu \lambda^* \neq \emptyset$ в зависимости от того, нуль или более раз входит λ в λ^* в выражении $\rho^+ \mu \lambda^*$. \square

Лемма 6.12. Пусть $G = (N, \Sigma, P, S)$ — однозначная, приведенная и обратимая КС-грамматика, для которой $\mu \cap \rho^* \mu \lambda^+ = \emptyset$ и $\rho^+ \mu \cap \mu \lambda^* = \emptyset$. Тогда G — грамматика Колмерауэра.

Доказательство. Зададим отношения предшествования Колмерауэра:

(1) $X \equiv Y$ тогда и только тогда, когда $X \mu Y$,

(2) $X \lessdot Y$ тогда и только тогда, когда $X \mu \lambda^+ Y$ либо $X = \$$, $Y \neq \$$,

(3) $X \gg Y$ тогда и только тогда, когда $X \neq \$$ и $Y = \$$ либо $X \rho^+ \mu \lambda^* Y$, но $X \mu \lambda^+ Y$ не выполняется.

Легко показать, что эти отношения не пересекаются. Если $\equiv \cap \lessdot \neq \emptyset$ или $\equiv \cap \gg \neq \emptyset$, то $\mu \cap \rho^* \mu \lambda^+ \neq \emptyset$ или $\mu \cap \rho^+ \mu \neq \emptyset$, и тогда $\mu \lambda^* \cap \rho^+ \mu \neq \emptyset$. Если $\lessdot \cap \gg \neq \emptyset$, то $X \mu \lambda^+ Y$. Но $X \mu \lambda^+ Y$ не выполняется. Ясно, что все три пересечения отношений пусты.

Допустим, что индуцированный этими отношениями двухмагазинный анализатор T содержит правило $(YX_1 \dots X_k, Z) \rightarrow (Y, AZ)$. Тогда $Y \lessdot X_1$ и потому $Y = \$$ или $Y \mu \lambda^+ X_1$. Но $X_i \equiv X_{i+1}$ для $1 \leq i < k$, так что $X_i \mu X_{i+1}$. Наконец, $X_k \gg Z$, и поэтому $Z = \$$ или $X_k \rho^+ \mu \lambda^* Z$. Если $Y \neq \$$ и $Z \neq \$$, то, согласно лемме 6.11, если хранящаяся в двух магазинах цепочка выводима в грамматике G , $X_1 \dots X_k$ будет ее составляющей. Со случаями $Y = \$$ и $Z = \$$ справиться легко, и потому можно заключить, что каждая свертка, проделанная анализатором T над выводимой цепочкой, дает выводимую цепочку.

Таким образом, достаточно показать, что, начиная работать с цепочкой w из языка $L(G)$, анализатор T будет продолжать делать свертки, пока не свернет w к S .

По лемме 6.10, если X и Y — смежные символы выводимой цепочки, то $X \rho^* \mu \lambda^* Y$. Поэтому верно какое-нибудь из соотношений $X \mu Y$, $X \rho^+ \mu Y$, $X \mu \lambda^+ Y$ или $X \rho^+ \mu \lambda^+ Y$. В любом случае X и Y находятся в одном из отношений предшествования Колмерауэра.

Индукцией по числу тактов анализатора T можно показать, что если X и Y — смежные символы в первом магазине, то $X \lessdot Y$ или $X \equiv Y$. Доказательство по существу состоит в том, что X и Y могут оказаться смежными, только если Y переносится в первый магазин в тот момент, когда X — его верхний символ. Но правила анализатора T таковы, что $X \lessdot Y$ или $X \equiv Y$.

Так как $\$$ остается во втором магазине, в нем всегда есть пара смежных символов, находящихся в отношении \gg . Таким образом, если анализатор T еще не достиг конфигурации $(\$, \$\$, \pi)$, он делает переносы до тех пор, пока верхние символы первого и второго магазинов не окажутся в отношении \gg . В этот момент становится возможной свертка, поскольку между символами первого магазина никогда не выполняется отношение \gg . T делает ее и продолжает работу. \square

Теорема 6.6. Грамматика является грамматикой Колмерауэра тогда и только тогда, когда она однозначная, приведенная, обратимая и $\mu \cap \rho^* \mu \lambda^+ = \emptyset$, $\rho^+ \mu \cap \mu \lambda^* = \emptyset$.

Доказательство. Непосредственно следует из лемм 6.8, 6.9 и 6.12. \square

	<i>a</i>	<i>b</i>	<i>s</i>	<i>A</i>	$\$$
$\$$	<	<	<	<	
<i>a</i>	<	<	\doteq	\geq	\geq
<i>b</i>	<	<	\doteq	\geq	\geq
<i>s</i>	<			\doteq	\geq
<i>A</i>	\geq			\geq	\geq

Рис. 6.4. Отношения предшествования Колмерауэра.

Пример 6.16. В примере 6.15 мы видели, что грамматика $S \rightarrow aSA | bSA | b$, $A \rightarrow a$ удовлетворяет нужным условиям. Лемма 6.12 подсказывает, что для этой грамматики отношения предшествования Колмерауэра определяются так, как показано на рис. 6.4. \square

УПРАЖНЕНИЯ

6.2.1. Какие из следующих разборов являются нисходящими для грамматики G_0 ? Какая выводится цепочка (если таковая существует)?

- (а) (1,0)(3,2)(5,4)(2,0)(4,2)(2,5)(4,0)(4,5)(6,5)(6,2)(6,0).
- (б) (2,0)(4,0)(5,0)(6,1).

6.2.2. Постройте двухмагазинный анализатор, корректный для грамматики G_0 .

6.2.3. Какие из следующих грамматик являются грамматиками Колмерауэра?

- (а) G_0
- (б) $S \rightarrow aA | bB$
 $A \rightarrow 0A1 | 01$
 $B \rightarrow 0B11 | 011$
- (в) $S \rightarrow aAB | b$
 $A \rightarrow bSB | a$
 $B \rightarrow a$

***6.2.4.** Покажите, что если грамматика G —приведенная и обратимая и $\mu \cap \rho^* \mu \lambda^+ = \emptyset$, $\rho^+ \mu \cap \mu \lambda^* = \emptyset$, то она однозначная.

6.2.5. Покажите, что каждая обратимая регулярная грамматика является грамматикой Колмерауэра.

6.2.6. Покажите, что каждая обратимая грамматика в нормальной форме Грейбах, для которой $\rho^+ \mu \cap \mu = \emptyset$, является грамматикой Колмерауэра.

6.2.7. Покажите, что двухмагазинный анализатор из примера 6.12 корректен для соответствующей грамматики.

6.2.8. С помощью теоремы 6.6 покажите, что каждая грамматика простого предшествования является грамматикой Колмерауэра.

6.2.9. Докажите лемму 6.9.

6.2.10. Докажите лемму 6.10.

6.2.11. Пусть G —грамматика Колмерауэра с отношениями предшествования Колмерауэра, для которой индуцируемый ек двухмагазинный анализатор не только анализирует каждую цепочку из $L(G)$, но также корректно анализирует каждую выводимую цепочку грамматики G . Покажите, что

- (а) $\mu \subseteq \doteq^1$,
- (б) $\mu \lambda^+ \subseteq \lessdot$,
- (в) $\rho^+ \mu \subseteq \geq$,
- (г) $\rho^+ \mu \lambda^+ \subseteq \lessdot \cup \geq$.

***6.2.12.** Пусть G —грамматика Колмерауэра и v —подмножество отношения $\rho^+ \mu \lambda^+ - \rho^+ \mu - \mu \lambda^+$. Покажите, что отношения $\doteq = \mu$, $\lessdot = \rho^* \mu \lambda^+ - v$ и $\geq = \rho^+ \mu \cup v$ —это отношения предшествования Колмерауэра, позволяющие анализировать любую выводимую цепочку грамматики G .

6.2.13. Покажите, что каждый двухмагазинный анализатор расходует на обработку цепочки длины n время $O(n)$.

***6.2.14.** Покажите, что если язык L определяется грамматикой Колмерауэра, то и L^R определяется такой грамматикой.

***6.2.15.** Является ли каждая (1,1)-ОПК-грамматика грамматикой Колмерауэра?

***6.2.16.** Покажите, что существует такой язык L , определяемый грамматикой Колмерауэра, что ни L , ни L^R не является детерминированным КС-языком. Заметьте, что использовавшийся в качестве примера язык $\{wba^n || w|=n\}$ не детерминированный, хотя его обращение—детерминированный язык.

¹⁾ Это утверждение—частный случай леммы 6.7; оно включено только для полноты.

***6.2.17.** Говорят, что двухмагазинный анализатор T распознает область определения отношения $\tau(T)$ независимо от того, связи ли как-нибудь выдаваемый им „разбор“ с входной цепочкой. Покажите, что каждый рекурсивно перечислимый язык распознается некоторым детерминированным двухмагазинным анализатором. *Указание:* Полезно сделать исходную грамматику неоднозначной.

Открытая проблема

6.2.18. Дайте характеристику класса КС-грамматик, однозначных или нет, для которых существуют детерминированные двухмагазинные анализаторы, индуцируемые непересекающимися отношениями „предшествования“.

Замечания по литературе

Грамматики Колмерауэра и теорема 6.6 впервые были опубликованы в работе Колмерауэра [1970]. Грэй и Харрисон [1969] связали эти идеи с понятием множества знаменательных символов. Коэн и Чуллик [1971] рассматривали LR(k)-схему анализа, эффективно применяющую возвраты¹⁾.

¹⁾ См. также работы Лаврова и Ордяна [1975] и Ордяна [1975]. — Прим. перев.

ПРИЛОЖЕНИЕ

В приложении даются синтаксические описания четырех языков программирования, среди них

- (1) простой язык, служащий базой расширяемого языка;
- (2) Снобол 4, язык для обработки строк;
- (3) ПЛ 360, машинный язык высокого уровня для вычислительных машин ИБМ 360;
- (4) PAL, язык, объединяющий лямбда-исчисление с операторами присваивания.

Эти языки выбраны потому, что они не похожи друг на друга. Кроме того, их синтаксические описания достаточно малы, чтобы ими можно было пользоваться в некоторых упражнениях на программирование, приведенных в нашей книге, не расходя слишком много времени (как человека, так и машины). При этом выбранные языки довольно сложны, и в них представлен целый букет проблем, с которыми приходится сталкиваться при реализации более традиционных языков программирования, таких, как Алгол, Фортран, ПЛ/I. Синтаксические описания последних можно найти в следующих источниках:

- (1) Алгол 60 в [Наур, 1963],
- (2) Алгол 68 в [Вай Вейнгаарден, 1969],
- (3) Фортран в [АНС, 1966] (см. также [АНС Подкомитет, 1971]).
- (4) ПЛ/I в отчете Венской лаборатории ИБМ, TR 25.096¹⁾.

П.1. СИНТАКСИС РАСШИРЯЕМОГО ЯЗЫКА

Опишем язык, предложенный Ливенвортом [1966] в качестве базового языка, который можно расширять с помощью синтаксических макросов. Синтаксис этого языка будет представлен

¹⁾ См. более доступную книгу: Универсальный язык программирования PL/I, изд-во „Мир“, М., 1968. — Прим. перев.

ПРИЛОЖЕНИЕ

в виде двух частей. Первая состоит из правил высокого уровня, определяющих базовый язык. Этот базовый язык можно использовать сам по себе как алгебраический язык с блочной структурой.

Вторая часть описания — множество правил, определяющих механизм расширения, который позволяет описывать новые виды операторов и функций с помощью оператора макроопределения, порождаемого правилом 37. Это правило говорит о том, что частным случаем <оператора> может быть <макроопределение>, а оно по правилам 39 и 40 может быть либо <макроопределением оператора>, либо <макроопределением функции>.

Правила 41 и 42 показывают, что каждое из этих макроопределений включает <макроструктуру> и <определение>. <Макроструктура> определяет форму новой синтаксической конструкции, а <определение> дает ассоциируемый с ней перевод. <Макроструктура> и <определение> могут быть любыми цепочками, состоящими из терминальных и нетерминальных символов, но при условии, что каждый нетерминал, встречающийся в <определении>, должен также встречаться в <макроструктуре>. (Это похоже на правило СУ-схемы, но здесь не налагается ограничений на то, сколько раз можно брать один нетерминал в транслирующем элементе.)

Мы не даем явных правил для <макроструктуры> и <определения>. На самом деле определение, говорящее о том, что каждый нетерминал из <определения> должен встречаться в <макроструктуре>, нельзя выразить с помощью контекстно-свободных правил.

Правило 37 указывает, что вместо <оператора> в выводимую цепочку можно подставить любой частный случай <макроструктуры>, определенный в некотором макроопределении оператора. Аналогично, правило 43 позволяет любой частный случай <макроструктуры>, определенный в некотором макроопределении функции, подставлять в выводимую цепочку вместо <первичного>.

Например, оператор суммирования можно определить с помощью вывода

```
<оператор> => <макроопределение>
              => <макроопределение оператора>
              => smacro <макроструктура> define <определение>
                  endmacro
```

Возьмем в качестве <макроструктуры> цепочку

```
sum <выражение>(1) with <переменная> ← <выражение>(2) to
                                         <выражение>(3)
```

Перевод этой макроструктуры можно определить, взяв в качестве ее определения цепочку

```
begin local t; local s; local r;
      t ← 0;
      <переменная> ← <выражение>(2);
      r: if <переменная> > <выражение>(3) then goto s;
          t ← t + <выражение>(1);
          <переменная> ← <переменная> + 1;
          goto r;
      s: result t
end
```

Тогда, если написан оператор

```
sum a with b ← c to d
```

то до того, как он будет проанализирован в соответствии с правилами высокого уровня, его надо перевести в цепочку

```
begin local t; local s; local r;
      t ← 0;
      b ← c;
      r: if b > d then goto s;
          t ← t + a;
          b ← b + 1;
          goto r;
      s: result t
end
```

И, наконец, нетерминалы <идентификатор>, <метка> и <костанта> — это лексические единицы (лексемы), которые мы оставляем неопределенными, а читателя просим либо рассматривать их как терминальные символы, либо определить их, как ему нравится, и добавить эти определения.

Правила высокого уровня

- 1 <программа> →
 <блок>
- 2 <блок> →
 begin <необязат локал идентифиры> <список операторов> e
- 3 <необязат локал идентифиры> →
 <необязат локал идентифиры> local <идентификатор>; | e
- 5 <список операторов> →
 <оператор> | <список операторов>; <оператор>
- 7 <оператор> →
 <переменная> ← <выражение> | goto <идентификатор> |
 if <выражение> then <оператор> | <блок> | result <выражение>
 <метка>; <оператор>

- 13 <выражение> →
 <арифмет выражение> <операция отношения> <арифмет выражение>
 | <арифмет выражение>
 15 <арифмет выражение> →
 <арифмет выражение> <опер типа слож> <терм> | <терм>
 17 <терм> →
 <терм> <опер типа умнож> <первичное> | <первичное>
 19 <первичное> →
 <переменная> | <константа> | (<выражение>) | <блок>
 23 <переменная> →
 <идентификатор> | <идентификатор> (<список выражений>)
 25 <список выражений> →
 <список выражений>, <выражение> | <выражение>
 27 <операция отношения> →
 <|≤|=|=|≠|>|≥>
 33 <опер типа слож> →
 + | -
 35 <опер типа умнож> →
 * | /

Механизм расширения

- 37 <оператор> →
 <макроопределение> | <макроструктура>
 39 <макроопределение> →
 <макроопределение оператора> | <макроопределение функции>
 41 <макроопределение оператора> →
 smacro <макроструктура> define <определение> endmacro
 42 <макроопределение функции> →
 fmacro <макроструктура> define <определение> endmacro
 43 <первичное> →
 <макроструктура>

Замечания

1. <макроструктура> и <определение> могут быть любыми цепочками терминальных и нетерминальных символов. Однако каждый нетерминал, участвующий в <определении>, должен встречаться также и в соответствующей <макроструктуре>.

2. <константа>, <идентификатор> и <метка> — лексические переменные, которые здесь не определяются.

П.2. СИНТАКСИС ОПЕРАТОРОВ ЯЗЫКА СНОБОЛ 4

Здесь мы определим синтаксическую структуру операторов Снобола 4, как это описано у Грисвoldа и др. [1971]. Синтаксическое описание состоит из двух частей. Первая часть содер-

жит контексто-свободные правила, описывающие синтаксис операторов с использованием лексических переменных, которые описываются во второй части с помощью регулярных определений, рассмотренных в гл. 3. Деление грамматики на синтаксическую и лексическую части здесь довольно произвольно, и в синтаксическом описании не отражены приоритет и ассоциативность операций. Все операции, кроме \neg , $!$ и $**$, предполагаются ассоциативными „слева направо“. Приоритет операций задается следующим соглашением:

1. &	2.	3. <пробелы>
4. @	5. + -	6. #
7. /	8. *	9. %
10. ! **	11. \$	12. \neg ?

Правила высокого уровня

- 1 <оператор> →
 <оператор присваивания> | <оператор сравнения> | <оператор замены> | <вырожденный оператор> | <оператор конца>
 6 <оператор присваивания> →
 <необязат метка> <область субъекта> <равно> <область объекта> <область перехода> <коноп>
 7 <оператор сравнения> →
 <необязат метка> <область субъекта> <область образца>
 | <область перехода> <коноп>
 8 <оператор замены> →
 <необязат метка> <область субъекта> <область образца>
 | <область объекта> <область перехода> <коноп>
 9 <вырожденный оператор> →
 <необязат метка> <область субъекта> <область перехода>
 | <необязат метка> <область перехода> <коноп>
 11 <оператор конца> →
 END <коноп> | END <пробелы> <метка> <коноп> |
 END <пробелы> END <коноп>
 14 <необязат метка> →
 <метка> | e
 16 <область субъекта> →
 <пробелы> <элемент>
 17 <равно> →
 <пробелы> =
 18 <область объекта> →
 <пробелы> <выражение>
 19 <область перехода> →
 <пробелы>: <необязат пробелы> <основной переход> | e
 21 <основной переход> →
 <переход> |

ПРИЛОЖЕНИЕ

*S <переход> <необязат пробелы> <необязат F переход> |
*F <переход> <необязат пробелы> <необязат S переход>**

24 <переход> →
 (<выражение>) | <<выражение>>

26 <необязат S переход> →
 S <переход> | e

28 <необязат F переход> →
 F <переход> | e

30 <коноп> →
 <необязат пробелы>; | <необязат пробелы> <констрок>

32 <область образца> →
 <пробелы> <выражение>

33 <элемент> →
 <необязат унарные> <основной элемент>

34 <необязат унарные> →
 <операция> <необязат унарные> | e

36 <основной элемент> →
 <идентификатор> | <литерный> | <вызов функции> | <имя> |
 (<выражение>)

41 <вызов функции> →
 <идентификатор> (<список аргум>)

42 <имя> →
 <идентификатор> <<список аргум>>

43 <список аргум> →
 <список аргум>, <выражение> | <выражение>

45 <выражение> →
 <необязат пробелы> <элемент> <необязат пробелы> |
 <необязат пробелы> <операция> <необязат пробелы> |
 <необязат пробелы>

48 <необязат пробелы> →
 <пробелы> | e

50 <операция> →
 <элемент> <бинарное> <элемент> | <элемент> <бинарное>
 <выражение>

Регулярные определения лексем

<цифра> =
 0|1|2|3|4|5|6|7|8|9

<буква> =
 A | B | C | ... | Z

<буквоцифра> =
 <буква> | <цифра>

<идентификатор> =
 <буква> (<буквоцифра> |·|_)*

<пробелы> =
 <знак пробела>*

<целое> =
 <цифра>*

<вещественное> =
 <целое>. <целое> | <целое>.

<операция> =
 ¬ | ? | \$ | . | ! | % | * | / | # | + | - | @ | || | &

<бинарное> =
 <пробелы> | <пробелы> <операция> <пробелы> |
 <пробелы> ** <пробелы>

<слитерное> =
 '(<EBCDIC литер>—') **¹⁾

<длитерное> =
 "(<EBCDIC литер>—") **

<литерное> =
 <слитерное> | <длитерное> | <целое> | <вещественное>

<метка> =
 <буквоцифра> (<EBCDIC литер>—(<знак пробела> |;))**
 —END

Лексические переменные

<знак пробела>
 <EBCDIC литер>
 <констрок>²⁾

П.3. СИНТАКСИС ПЛ 360

В данном разделе содержится синтаксическое описание ПЛ 360, машинного языка высокого уровня, разработанного Никлаусом Виртом для вычислительных машин ИБМ 360. Это описание представляет собой грамматику предшествования, взятую из работы [Вирт, 1968].

Правила высокого уровня

- 1 <регистр> →
 <идентификатор>
- 2 <идентификатор ячейки> →
 <идентификатор>

¹⁾ Знак — в этой и следующих строках является метасимволом.
²⁾ Конец строки.

ПРИЛОЖЕНИЕ

3 <идентификатор процедуры> →
 <идентификатор>
 4 <идентификатор функции> →
 <идентификатор>
 5 <ячейка> →
 <идентификатор ячейки> | <яч1> | <яч2>
 8 <яч1> →
 <яч2> <ариф опер> <число> | <яч3> <число>
 10 <яч2> →
 <яч3> <регистр>
 11 <яч3> →
 <идентификатор ячейки> (
 12 <унар опер> →
 abs | neg | neg abs
 15 <ариф опер> →
 + | - | * | / | + + | --
 21 <лог опер> →
 and | or | xor
 24 <сдвиг> →
 shla | shra | shll | shr1
 28 <присваивание регистру> →
 <регистр> := <ячейка> |
 <регистр> := <число> |
 <регистр> := <цепочка> |
 <регистр> := <регистр> |
 <регистр> := <унар опер> <ячейка> |
 <регистр> := <унар опер> <число> |
 <регистр> := <унар опер> <регистр> |
 <регистр> := @ <ячейка> |
 <присваивание регистру> <ариф опер> <ячейка> |
 <присваивание регистру> <ариф опер> <число> |
 <присваивание регистру> <ариф опер> <регистр> |
 <присваивание регистру> <лог опер> <ячейка> |
 <присваивание регистру> <лог опер> <число> |
 <присваивание регистру> <лог опер> <регистр> |
 <присваивание регистру> <сдвиг> <число> |
 <присваивание регистру> <сдвиг> <регистр>
 44 <функ1> →
 <функ2> <число> |
 <функ2> <регистр> |
 <функ2> <ячейка> |
 <функ2> <цепочка>
 48 <функ2> →
 <идентификатор функции> | <функ1>,
 50 <послед случаев> →
 case <регистр> of begin | <послед случаев> <оператор>;

52 <простой оператор> →
 <ячейка> := <регистр> | <присваивание регистру> | null |
 goto <идентификатор> <идентификатор процедуры> |
 <идентификатор функции> | <функ1> ()
 <послед случаев> end | <тело блока> end
 61 <отношение> →
 <|=|=|> | <=|=|> = | ≠ = |
 67 <не> →
 ¬
 68 <условие> →
 <регистр> <отношение> <ячейка> |
 <регистр> <отношение> <число> |
 <регистр> <отношение> <регистр> |
 <регистр> <отношение> <цепочка> |
 overflow | <отношение> |
 <ячейка> | <не> <ячейка>
 76 <состав условие> →
 <условие> | <состус или> <условие>
 78 <состус или> →
 <состав условие> and | <состав условие> or
 80 <состус то> →
 <состав условие> then
 81 <истин часть> →
 <простой оператор> else
 82 <пока> →
 while
 83 <услов делай> →
 <состав условие> do
 84 <шаг присваивания> →
 <присваивание регистру> step <число>
 85 <граница> →
 until <регистр> | until <ячейка> | until <число>
 88 <делай> →
 do
 89 <оператор*> →
 <простой оператор> |
 if <состус то> <оператор*> |
 if <состус то> <истин часть> <оператор*> |
 пока <услов делай> <оператор*> |
 for <шаг присваивания> <граница> <делай> <оператор*>
 94 <оператор> →
 <оператор*>

95 <простой тип> →
 short integer | integer | logical | real | long real | byte | character
 102 <тип> →
 <простой тип> | array <число> <простой тип>
 104 <опис1> →
 <тип> <идентификатор> | <опис2> <идентификатор>
 106 <опис2> →
 <опис7>,
 107 <опис3> →
 <опис1>=
 108 <опис4> →
 <опис3> (| <опис5>,
 110 <опис5> →
 <опис4> <число> | <опис4> <цепочка>
 112 <опис6> →
 <опис3>
 113 <опис7> →
 <опис1> | <опис6> <число> | <опис6> <цепочка> | <опис5>
 117 <опис функ1> →
 function | <опис функ7>
 119 <опис функ2> →
 <опис функ1> <идентификатор>
 120 <опис функ3> →
 <опис функ2> (|
 121 <опис функ4> →
 <опис функ3> <число>
 122 <опис функ5> →
 <опис функ4>,
 123 <опис функ6> →
 <опис функ5> <число>
 124 <опис функ7> →
 <опис функ6>
 125 <синоним опис1> →
 <тип> <идентификатор> syn |
 <простой тип> register <идентификатор> syn |
 <синоним опис3> <идентификатор> syn
 128 <синоним опис2> →
 <синоним опис1> <ячейка> |
 <синоним опис1> <число> |
 <синоним опис1> <регистр>
 131 <синоним опис3> →
 <синоним опис2>,
 132 <голова сегмента> →
 segment

133 <заголовов проц1> →
 procedure | <голова сегмента> procedure
 135 <заголовов проц2> →
 <заголовов проц1> <идентификатор>
 136 <заголовов проц3> →
 <заголовов проц2> (|
 137 <заголовов проц4> →
 <заголовов проц3> <регистр>
 138 <заголовов проц5> →
 <заголовов проц4>);
 139 <заголовов проц6> →
 <заголовов проц5>;
 140 <описание> →
 <опис7> | <опис функ7> | <синоним опис2> |
 <заголовов проц6> <оператор*> | <голова сегмента> base
 <регистр>
 145 <определение метки> →
 <идентификатор>:
 146 <голова блока> →
 begin | <голова блока> <описание>;
 148 <тело блока> →
 <голова блока> | <тело блока> <оператор>; |
 <тело блока> <определение метки>
 151 <программа> →
 <оператор>.

Лексические переменные

<идентификатор>
 <цепочка>
 <число>

П.4. СХЕМА СИНТАКСИЧЕСКИ УПРАВЛЯЕМОГО ПЕРЕВОДА ДЛЯ ЯЗЫКА PAL

Здесь мы приведем схему синтаксически управляемой трансляции для языка программирования PAL, созданного Возенкрафтом и Эвансом [1969] и включающего в себя лямбда-исчисление и операторы присваивания. Название языка составлено из первых букв слов Pedagogic Algorithmic Language (Педагогический Алгоритмический Язык).

Описываемая здесь СУ-схема отображает программы, написанные на слегка модифицированной версии языка PAL, в постфиксные польские записи. Эта СУ-схема взята из диссертации Де Ремера [1969]. Она состоит из двух частей. Первая представляет собой входную контекстно-свободную грамматику, а

именно простую LR(1)-грамматику. Вторая часть определяет семантические правила, ассоциируемые с правилами входной грамматики.

Кроме того, мы дадим регулярные определения, описывающие используемые в этой схеме лексические переменные <константа> и <переменная>.

Входная грамматика

```

1 <программа> →
    <список определ> | <выражение>
3 <список определ> →
    def <определение> <список определ> | def <определение>
5 <выражение> →
    let <определение> in <выражение> |
    fn <оспер часть>. <выражение> |
    <где выражение>
8 <где выражение> →
    <знач выражение> where <рек определение> |
    <знач выражение>
10 <знач выражение> →
    valof <команда> | <команда>
12 <команда> →
    <помеч команда>; <команда> | <помеч команда>
14 <помеч команда> →
    <переменная>: <помеч команда> | <услов команда>
16 <услов команда> →
    test <логическое> ifso <помеч команда> ifnot <помеч команда> |
    test <логическое> ifnot <помеч команда> ifso <помеч команда> |
    if <логическое> do <помеч команда> |
    unless <логическое> do <помеч команда> |
    while <логическое> do <помеч команда> |
    until <логическое> do <помеч команда> |
    <основ команда>
23 <основ команда> →
    <кортеж>:=<кортеж> | goto <комбинация> |
    res <кортеж> | <кортеж>
27 <кортеж> →
    <T1> | <T1>, <кортеж>
20 <T1> →
    <T1> aug <услов выражение> | <услов выражение>
31 <услов выражение> →
    <логическое> → <услов выражение> | <услов выражение> |
    <T2>

```

```

33 <T2> →
    $ <комбинация> | <логическое>
35 <логическое> →
    <логическое> or <конъюнкция> | <коинъюнкция>
37 <конъюнкция> →
    <конъюнкция> & <отрицание> | <отрицание>
39 <отрицание> →
    not <отношение> | <отношение>
41 <отношение> →
    <ариф выражение> <функция отнош> <ариф выражение> |
    <ариф выражение>
43 <ариф выражение> →
    <ариф выражение> + <терм> | <ариф выражение> - <терм> |
    + <терм> | - <терм> | <терм>
48 <терм> →
    <терм> * <множитель> | <терм>/<множитель> |
    <множитель>
51 <множитель> →
    <первичное> ** <множитель> | <первичное>
53 <первичное> →
    <первичное> % <переменая> <комбинация> |
    <комбинация>
55 <комбинация> →
    <комбинация> <произвольное> | <произвольное>
57 <произвольное> →
    <переменная> | <константа> | (<выражение>) |
    [<выражение>]
61 <определение> →
    <внутр определение> within <определение> |
    <внутр определение>
63 <внутр определение> →
    <внутр определение> inwhich <одноврем определение> |
    <одноврем определение>
65 <одноврем определение> →
    <рек определение> and <одноврем определение> |
    <рек определение>
67 <рек определение> →
    rec <основное определение> | <основное определение>
69 <основное определение> →
    <список переменных> = <выражение> |
    <переменая> <оспер часть> = <выражение> |
    (<определение>) | [<определение>]
73 <оспер часть> →
    <оспер часть> <основ перем> | <основ перем>

```

75 <основ перем> →
 ⟨переменная⟩ | (<список переменных⟩) | ()
 78 <список переменных⟩ →
 ⟨переменная⟩, <список переменных⟩ | ⟨переменная⟩

Правила перевода, соответствующие правилам входной грамматики

1 <программа> =
 ⟨список определ⟩ | <выражение>
 3 <список определ⟩ =
 ⟨определение⟩ <список определ⟩ def <определение⟩ lastdef
 5 <выражение⟩ =
 ⟨определение⟩ <выражение⟩ let |
 ⟨оспер часть⟩ <выражение⟩ lambda |
 ⟨где выражение⟩
 8 <где выражение⟩ =
 ⟨знач выражение⟩ <рек определение⟩ where |
 ⟨знач выражение⟩
 10 <знач выражение⟩ =
 ⟨команда⟩ valof | <команда⟩
 12 <команда⟩ =
 ⟨помеч команда⟩ <команда⟩; | <помеч команда⟩
 14 <помеч команда⟩ =
 ⟨переменная⟩ <помеч команда⟩: | <услов команда⟩
 16 <услов команда⟩ =
 ⟨логическое⟩ <помеч команда⟩ <помеч команда⟩ test-true |
 ⟨логическое⟩ <помеч команда⟩ <помеч команда⟩ test-false |
 ⟨логическое⟩ <помеч команда⟩ if |
 ⟨логическое⟩ <помеч команда⟩ unless |
 ⟨логическое⟩ <помеч команда⟩ while |
 ⟨логическое⟩ <помеч команда⟩ until |
 ⟨основная команда⟩
 23 <основная команда⟩ =
 ⟨кортеж⟩ <кортеж⟩: = | <комбинация⟩ goto |
 ⟨кортеж⟩ res | <кортеж⟩
 27 <кортеж⟩ =
 ⟨T1⟩ | <T1⟩ <кортеж⟩,
 29 <T1⟩ =
 ⟨T1⟩ <услов выражение⟩ aug | <услов выражение⟩
 31 <услов выражение⟩ =
 ⟨логическое⟩ <услов выражение⟩ <услов выражение⟩
 test-true | <T2⟩

33 <T2⟩ =
 ⟨комбинация⟩ \$ | <логическое⟩
 35 <логическое⟩ =
 ⟨логическое⟩ <конъюнкция⟩ or | <конъюнкция⟩
 37 <конъюнкция⟩ =
 ⟨конъюнкция⟩ <отрицание⟩ & | <отрицание⟩
 39 <отрицание⟩ =
 ⟨отношение⟩ not | <отношение⟩
 41 <отношение⟩ =
 ⟨ариф выражение⟩ <ариф выражение⟩ <функция отнош⟩ |
 ⟨ариф выражение⟩
 43 <ариф выражение⟩ =
 ⟨ариф выражение⟩ <терм⟩ + |
 ⟨ариф выражение⟩ <терм⟩ — |
 ⟨терм⟩ pos | <терм⟩ neg | <терм⟩
 48 <терм⟩ =
 ⟨терм⟩ <множитель⟩ * | <терм⟩ <множитель⟩ / |
 ⟨множитель⟩
 51 <множитель⟩ =
 ⟨первичное⟩ <множитель⟩ exp | <первичное⟩
 53 <первичное⟩ =
 ⟨первичное⟩ <переменная⟩ <комбинация⟩ % |
 ⟨комбинация⟩
 55 <комбинация⟩ =
 ⟨комбинация⟩ <произвольное⟩ gamma | <произвольное⟩
 57 <произвольное⟩ =
 ⟨переменная⟩ | <константа⟩ | <выражение⟩ |
 ⟨выражение⟩
 61 <определение⟩ =
 ⟨внутр определение⟩ <определение⟩ within |
 ⟨внутр определение⟩
 63 <внутр определение⟩ =
 ⟨внутр определение⟩ <одноврем определение⟩ inwhich |
 ⟨одноврем определение⟩
 65 <одноврем определение⟩ =
 ⟨рек определение⟩ <одноврем определение⟩ and |
 ⟨рек определение⟩
 67 <рек определение⟩ =
 ⟨основное определение⟩ rec | <основное определение⟩
 69 <основное определение⟩ =
 ⟨список переменных⟩ <выражение⟩ = |
 ⟨переменная⟩ <оспер часть⟩ <выражение⟩ ff |
 ⟨определение⟩ | <определение⟩
 73 <оспер часть⟩ =
 ⟨оспер часть⟩ <основ перем⟩ | <основ перем⟩

75 <основа переменных> =
 <переменная> | <список переменных> | ()

78 <список переменных> =
 <переменная> | <список переменных> v1 | <переменная>

Регулярные определения

<пропись буква> =
 A | B | C | ... | Z
<строч буква> =
 a | b | c | ... | z
<цифра> =
 0 | 1 | 2 | ... | 9
<буква> =
 <пропись буква> | <строч буква>
<буквоцифра> =
 <буква> | <цифра>
<знач истин> =
 true | false
<голова переменной> =
 <цифра>+ (<буква> | __) |
 <строч буква>+ (<пропись буква> | <цифра> | __) |
 <пропись буква> | __
<переменная> =
 <строч буква> | <голова переменной> (<буквоцифра> | __)*
<целое> =
 <цифра>+
<вещественное> =
 <цифра>+. <цифра>+
<заковыч элемент> =
 <любой знак отличный от * и '> |
 * n | * t | * b | * s | ** | *' | * k | * r
<заковыченное> =
 '<заковыч элемент>'*
<константа> =
 <целое> | <вещественное> | <заковыченное> |
 <знач истин> | e
<функция отнош> =
 gr | ge | eq | ne | ls | le

СПИСОК ЛИТЕРАТУРЫ¹⁾

- Абрахам [1965] (Abraham S.), Some questions of phrase structure grammars, I, *Comput. Linguist.*, 4, 61—70.
- Абрахам [1972] (Abrahams P. W.), A syntax directed parser for recalcitrant grammars, *Intern. J. Computer Math.*, 3:2/3, 105—115.
- Абэ и др. [1973] (Abe N., Mizumoto M., Toyoda J., Tanaka K.), Web grammars and several graphs, *J. Comput. System Sci.*, 7:1, 37—68.
- Автоматический перевод [1971], Сборник статей, изд-во „Прогресс“, М.
- Айронс [1961] (Ironson E. T.), A syntax directed compiler for ALGOL 60, *Comm. ACM*, 4:1, 51—55.
- Айронс [1963а] (Ironson E. T.), An error correcting parse algorithm, *Comm. ACM*, 6:11, 669—673.
- Айронс [1963б] (Ironson E. T.), The structure and use of the syntax directed compiler, *Ann. Rev. Autom. Program.*, 3, 207—227.
- Айронс [1964] (Ironson E. T.), Structural connections in formal languages, *Comm. ACM*, 7:2, 62—67.
- АЛГОЛ 68 [1976], Алгол 68. Методы реализации, под ред. Г. С. Цейтина, изд-во ЛГУ.
- Ангер [1968] (Unger S. H.), A global parser for context-free phrase structure grammars, *Comm. ACM*, 11:4, 240—246; 11:6, 427.
- Андерсон и др. [1973] (Anderson T., Eve G., Hornung I. I.), Efficient LR (I) parsers, *Acta Informatica*, 2:1, 12—39.
- Анисимов А. В. [1974а], Формальные грамматики, учитывающие внешние терминальные контексты, *Кибернетика*, 3, 81—88.
- Анисимов А. В. [1974б], Об аппарате управления в синтаксических анализаторах, *Кибернетика*, 6, 57—59.
- 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.
- Арбиг [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.

¹⁾ Кружочком ° отмечена литература, добавленная при переводе. — Прим. перев.

- *Ахо, Джонсон [1974] (Aho A. V., Johnson S. C.), LR-parsing, *Comput. Survey*, 6, 99—124.
- Ахо, Ульман [1969a] (Aho A. V., Ullman J. D.), Syntax directed translations and the pushdown assembler, *J. Comput. Syst. Sci.*, 3:1, 37—56.
- Ахо, Ульман [1969b] (Aho A. V., Ullman J. D.), Properties of syntax directed translations, *J. Comput. Syst. Sci.*, 3:3, 319—334.
- Ахо, Ульман [1971] (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 [1973] (Aho A. V., Ullman J. D.), Error detection in precedence parsers, *Math. Syst. Theory*, 7:2, 97—113.
- 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 сложность языков, допускаемых магазинными автоматами, в сб. „Языки и автоматы“, изд-во „Мир“, М., 1975, стр. 185—197.)
- Ахо и др. [1972] (Aho A. V., Denning P. J., Ullman J. D.), Weak and mixed strategy precedence parsing, *J. ACM*, 19:2, 225—243.
- *Ахо и др. [1975] (Aho A. V., Johnson S. C., Ullman J. D.), Deterministic parsing of ambiguous grammars, *Comm. ACM*, 18:8, 441—453.
- *Бабинов Ю. П. [1976], Два класса искустворачивающих грамматик предшествования, *ЖВМ и МФ*, 16:4, 1027—1037.
- Барнетт, Футрель [1962] (Barnett M. P., Futerelle 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.
- Бауэр и др. [1968] (Bauer H., Becker S., Graham S. L.), ALGOL W Implementation, CS98, Computer Science Dept., Stanford Univ., Stanford, Calif.
- 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.
- Лжозовский [1964] (Brzozowski J. A.), Derivatives of regular expressions, *J. ACM*, 11:4, 481—494.
- 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, 1—34 (1973).)
- 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.
- *Бородин [1973] (Borodin A.), Computational complexity: theory and practice, в сб. „Currents in the theory of computing“ под ред. Aho A., Prentice-Hall, Englewood Cliffs, N. J., 35—89.
- *Братчиков И. Л. [1975], Синтаксис языков программирования, изд-во „Наука“, М.
- Браффорт, Хиршберг [1963] (Brassfort P., Hirschberg D. (eds.)), Computer programming and formal systems, North-Holland, Amsterdam.
- Брукер, Моррис [1963] (Brooker R. A., Morris D.), The compiler-compiler, *Ann. Rev. Autom. Program.*, 3, 229—275.
- Бук [1970] (Book R. V.), Problems in formal language theory, Proc. 4th Annual Princeton Conference on Information Sciences and Systems, 253—256.
- 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.
- Вайз [1971] (Wise D. S.), Domölk's algorithm applied to generalized overlap resolvable grammars, Proc. 3rd Annual ACM Symposium on Theory of Computing, 171—184.
- *Вайз [1972] (Wise D. S.), Generalized overlap resolvable grammars and their parsers, *J. Comp. Syst. Sci.*, 6:6, 538—572.
- *Валиант [1975] (Valiant L. G.), General context-free recognition in less than cubic time, *J. Comp. Syst. Sci.*, 10:8, 308—315.
- 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.)
- Вегбрейт [1970] (Wegbreit B.), Studies in extensible programming languages, Ph. D. Thesis, Harvard Univ., Cambridge, Mass.
- *Вельбицкий И. В. [1973], Метаязык R-грамматик, *Кибернетика*, 3, 47—63.
- *Вельбицкий И. В., Ющенко Е. Л. [1970], Метаязык, ориентированный на синтаксический анализ и контроль, *Кибернетика*, 2, 50—53.
- 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.)
- *Виноград [1972] (Winograd T.), Understanding natural language, Academic Press, New York, Edinburgh Univ. Press, Edinburgh. (Русский перевод: Виноград Т., Программа, понимающая естественный язык, изд-во „Мир“, М., 1976.)
- 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 [1969b] (Wood D.), A note 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.
- *Вуд [1970] (Woods W. A.), Transition network grammars for natural language analysis, *Comm. ACM*, 13:10, 591—606. (Русский перевод: Вудс В. А., Сетевые грамматики для анализа естественных языков, *Кибернетический сборник*, новая серия, вып. 13, изд-во „Мир“, М., 1976, стр. 120—158.)
- Галлер, Перлес [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.

- *Геци [1975] (Ghezzi C.), LL(1) grammars supporting an efficient error handling, *Inform. Proces. Letters*, 3:6, 174—176.
- 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 А. [1968] (Ginzburg A.), Algebraic theory of automata, Academic Press, New York.
- Гинзбург С. [1962] (Ginsburg S.), An introduction to mathematical machine theory, Addison-Wesley, Reading, Mass.
- Гинзбург С. [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.), Deterministic context-free languages, *Inform. and Control*, 9:6, 620—648.
- Гинзбург, Грейбах [1969] (Ginsburg S., Greibach S.), Abstract families of languages, *Memoir Amer. Math. Soc.*, 87. (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.)
- *Гинзбург, Уллиан [1966] (Ginsburg S., Ullian J. S.), Ambiguity in context-free languages, *J. ACM*, 13:3, 364—368.
- *Гладкий А. В. [1965], Алгоритмическая нераспознаваемость существенной неопределенности КС-языков, *Алгебра и логика*, 4:4, 53—64.
- *Гладкий А. В. [1973], Формальные грамматики и языки, изд-во „Наука“, М.
- Глушков В. М. [1962], Синтез цифровых автоматов, Физматгиз, М.
- Гончарова Л. И. [1975], Приоритетный анализ и контекстные условия, *ЖВМ и МФ*, 15:3, 719—727.
- Грау и др. [1967] (Grau A. A., Hill U., Langmaack H.), Translation of ALGOL 60, Springer, Berlin.
- Грейбах [1965] (Greibach S.), A new normal form theorem for context-free phrase structure grammars, *J. ACM*, 12:1, 42—52.
- Грейбах, Хопкрофт [1969] (Greibach S., 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.)
- Грисвold и др. [1971] (Griswold R. E., Poage J. F., Polonsky I. P.), The SNOBOL 4 programming language (2nd ed.), Prentice-Hall, Inc., 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.)
- *Триффитс [1974] (Griffiths T. V.), LL(1) grammars and analysers. Compiler Construction, *Lecture Notes in Computer Science*, 21, 57—84.
- Гриффитс, Петрик [1965] (Griffiths T. V., Petrick S. R.), On the relative efficiencies of context-free grammar recognizers, *Comm. ACM*, 8:5, 289—300.
- *Триффитс, Петрик [1968] (Griffiths T. V., Petrick S. R.), Top-down versus bottom-up analysis, Information Processing — 68 (IFIP Congress), Booklet B, 80—85.
- Гросс, Лантен [1970] (Gross M., Lantin A.), Introduction to formal grammars, Springer, Berlin. (Русский перевод: Гросс М., Лантен А., Теория формальных грамматик, изд-во „Мир“, М., 1971.)
- *Грушевский В. В. [1972], Синтаксические структуры, в сб. „Системное и теоретическое программирование“, ВЦ СО АН ССР, Новосибирск.
- Грей [1969] (Gray J. N.), Precedence parsers for programming languages, Ph. D. Thesis, Univ. of California, Berkeley.
- Грей, 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.
- *Грей, Харрисон [1972] (Gray J. N., Harrison M. A.), On the covering and reduction problems for context-free grammars, *J. ACM*, 19:4, 675—698.
- *Грей, Харрисон [1973] (Gray J. N., Harrison M. A.), Canonical precedence schemes, *J. ACM*, 20:2, 214—234.
- Грей и др. [1967] (Gray J. N., Harrison M. A., Ibarra O.), Two way pushdown automata, *Inform. and Control*, 11:1, 30—70.
- Грэхем Р. [1964] (Graham R. M.), Bounded context translation, *Proc. AFIPS Spring Joint Computer Conference*, 25, 17—29.
- Грэхем С. [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.
- *Грэхем С. [1974] (Graham S. L.), On bounded right context languages and grammars, *SIAM J. Computing*, 3:2, 224—254.
- *Грэхем С., Роудз [1975] (Graham S. L., Rhodes E. M.), Practical syntactic error recovery, *Comm. ACM*, 18:11, 639—650.
- *Грэхем С. и др. [1976] (Graham S. L., Harrison M. A., Ruzzo W. L.), On-line context-free language recognition in less than cubic time, Proc. 8th Annual ACM Symposium on Theory of Computing, 112—120.
- 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.
- *Домёлки [1964] (Domölki B.), An algorithm for syntactic analysis, *Comput Linguist.*, 3, 29—46.
- *Домёлки Б. [1965], Алгоритмы для распознавания свойств последовательностей символов, *ЖВМ и МФ*, 5:1, 77—97.
- Дьюар и др. [1969] (Dewar R. B. K., Hochsprung R. R., Worley W. S.), The PTRAN 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.
- *Жоголев Е. А. [1965], Алгоритм выделения понятий с помощью синтаксической таблицы, *ЖВМ и МФ*, 5:4, 689—698.
- *Зыков А. А. [1969], Теория конечных графов, изд-во „Наука“, Новосибирск.
- 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.)
- Ихбия, Морзе [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.
- Камеда, Вайнер [1968] (Kameda T., Weiner P.), On the reduction of nondeterministic grammars, *Inform. and Control*, 11:1, 30—70.

- ministic 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.
- 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, Тории [1969] (Kasami T., Torii K.), A syntax analysis procedure for unambiguous context-free grammars, *J. ACM*, 16:3, 423—431.
- *Кауфман В. Ш. [1976], Синтаксический анализатор RAND, Вычисл. методы и программирование, вып. 25, 108—116.
- 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 первых сетях, в сб. „Автоматы“, ИЛ, М., 1956, стр. 15—67.)
- *Клини [1967] (Kleene S. C.), Mathematical logic, John Wiley and Sons, Inc., New York. (Русский перевод: Клини С. К., Математическая логика, изд-во „Мир“, М., 1973.)
- *Кнут [1964] (Knuth D. E.), Backus normal form vs Backus-Naur form, *Comm. ACM*, 7:12, 588—589.
- Кнут [1965] (Knuth D. E.), On the translation of languages from left to right, *Inform. and Control*, 8:6, 607—639. (Русский перевод: Кнут Д., 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.
- Кнут [1968] (Knuth D. E.), The art of computer programming. Vol. 1: Fundamental algorithms, Addison-Wesley, Reading, Mass. (Русский перевод: Кнут Д., Искусство программирования для ЭВМ, изд-во „Мир“, М., 1975.)
- *Кнут [1971] (Knuth D. E.), Top-down syntax analysis, *Acta Inform.*, 1:2, 79—110.
- Кок, Шварц [1970] (Cocke J., Schwartz J. T.), Programming languages and their compilers, Courant Institute of Mathematical Sciences, New York University, New York.
- Колмерауэр [1970] (Colmerauer A.), Total precedence relations, *J. ACM*, 17:1, 14—30.
- *Комор Т. [1972], Об одном свойстве модифицированных разделенных грамматик, *ЖВМ и МФ*, 12:6, 1612—1615.
- Конвей [1963] (Conway M. E.), Design of a separable transition-diagram compiler, *Comm. ACM*, 6:7, 396—408. (Русский перевод: Конвой М. Е., Проект делимого компилятора, основанного на диаграммах перехода, в сб. „Современное программирование“, изд-во „Сов. радио“, М., 1967, стр. 206—246.)
- Конвей, Максвелл [1963] (Conway R. W., Maxwell W. L.), CORC: the Cornell computing language, *Comm. ACM*, 6:6, 317—321.
- Конвей, Максвелл [1968] (Conway R. W., Maxwell W. L.), CUPL—an approach to introductory computing instruction, TR No. 68—4, Dept. of Computer Science, Cornell Univ., Ithaca, N. Y.
- Конвей и др. [1970] (Conway R. W. et al.), PL/C. A high performance subset of PL/I, TR70—55, Dept. of Computer Science, Cornell Univ., Ithaca, N. Y.
- *Конститинов В. И., Нуриев Р. М. [1973], Метаязык трансляции контекстно-свободных языков, Вычислительные системы, вып. 57, 74—83.
- 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, Чуллик [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. О., 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-Суони [1969] (Kurki-Suonio R.), Note on top-down languages, *BIT*, 9, 225—238.
- *Лавринцева Е. М., Ющенко Е. Л. [1972], Метод анализа программ на базе СМ-языка, *Кибернетика*, 2, 41—44.
- *Лавров С. С., Ордян А. А. [1975], Об одном расширении алгоритма Куяута для анализа бесконтекстных языков, *ЖВМ и МФ*, 15:4, 1006—1019.
- Лалонд и др. [1971] (LaLond W. R., Lee E. S., Horning J. J.), An LALR(k) parser generator, *Information Processing—71* (IFIP Congress), 153—157.
- Лафранс [1970] (LaFrance J.), Optimization of error recovery in syntax directed parsing algorithms, *ACM SIGPLAN Notices*, 5:12, 2—17.
- Лейниус [1970] (Leinius R. P.), Error detection and recovery for syntax directed compiler systems, Ph. D. Thesis, Univ. of Wisconsin, Madison.
- *Леман [1971] (Lehman D.), LR (k) grammars and deterministic languages, *Israel J. Math.*, 10:4, 526—530. (Русский перевод: Леман Д., LR (k)-грамматики и детерминированные языки, в сб. „Языки и автоматы“, изд-во „Мир“, М., 1975, стр. 43—46.)
- *Лернер, Лим [1970] (Lerner A., Lim A. L.), A note on transforming context-free grammars to Wirth-Weber precedence form, *Computer J.*, 13:2, 142—144.
- Ли [1967] (Lee J. A. N.), Anatomy of a compiler, 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.
- *Ломет [1973] (Lomet D. B.), A formalization of transition diagram systems, *J. ACM*, 20:2, 235—257.
- Лукаш, Вальк [1969] (Lucas P., Walk K.), On the formal description of PL/I, *Ann. Rev. Autom. Program.*, 6:3, 105—182.
- Льюис, Розенкрэнц [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 translation, *J. ACM*, 15:3, 464—488.

- Мак-Афи, Пресснер [1972] (McAfee J., Pressner L.), An algorithm for the design of simple precedence grammars, *J. ACM*, 19:3, 385—395.
- Мак-Илрой [1960] (McIlroy M. D.), Macro instruction extensions of compiler languages, *Comm. ACM*, 3:4, 414—420.
- Мак-Илрой [1968] (McIlroy M. D.), Coroutines, неопубликованное сообщение.
- Мак-Каллок, Питтс [1943] (McCullough W. S., Pitts E.), A logical calculus of the ideas immanent in nervous activity, *Bull. Math. Biophys.*, 5, 115—133. (Русский перевод: Маккаллок У. С., Питтс Э., Логическое исчисление идей, относящихся к первои активности, в сб. „Автоматы“, ИЛ, М., 1956, стр. 362—384.)
- Мак-Карти [1963] (McCarthy J.), A basis for the mathematical theory of computations, 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. Mathematical aspects of computer science, под. ред. Schwartz J. T., Proc. Symposium in Applied Mathematics, 19 (American Mathematical Society).
- Мак-Киман [1966] (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, Ямада [1950] (McNaughton R., Yamada H.), Regular expressions and state graphs for automata, *IRE Trans. on Electr. Comput.*, 9:1, 39—47.
- Мальцев А. И. [1965], Алгоритмы и рекурсивные функции, изд-во „Наука“, М.
- Марков А. А. [1951], Теория алгоритмов, Труды Математического института им. В. А. Стеклова, 38.
- 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.)
- Мовшович С. М. [1975], Продолжение синтаксического анализа после обнаружения ошибки, *Программирование*, 2, 11—16.
- 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.
- Моултон, Мюллер [1967] (Moulton P. G., Muller M. E.), A compiler emphasizing diagnostics, *Comm. ACM*, 10:1, 45—52.
- Мурро [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. Automata Studies, под ред. Shannon C. E., McCarthy J., Princeton University Press, Princeton, N. J. (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.
- Найр [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.)

- Непомнящая А. Ш. [1976], Об одном обобщении LL(k)-грамматик, описанном на эффективный синтаксический анализ, *Программирование*, 3, 13—21.
- Нийхольт [1976] (Nijholt A.), On the parsing of LL-regular grammars, *Lecture Notes in Computer Science*, 45, 446—452.
- Никитченко Н. С., Шкильняк С. С. [1975], Синтаксический анализ языков программирования методом развертки, *Программирование*, 8, 3—11.
- 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.)
- Ордян А. А. [1975], ОП- и ELR(1)-языки, *ЖВМ и МФ*, 15:5, 1289—1298.
- Оре [1962] (Ore O.), Theory of graphs, *Amer. Math. Soc. Colloq. Publ.*, 38. (Русский перевод: Оре О., Теория графов, изд-во „Наука“, М., 1968.)
- Основы разработки трансляторов [1974], Изд. Ростовского ун-та, Ростов-на-Дону.
- 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.
- Паул, Ангер [1968] (Paull M. C., Unger S. H.), Structural equivalence of context-free grammars, *J. Comp. Syst. Sci.*, 2:1, 427—463.
- Пейджер [1973] (Pager D.), The lane tracing algorithm for constructing LR(k) parsers, Proc. of 5th Annual Symposium on Theory of Computing, 172—181.
- Пейнтер [1970] (Painter J. A.), Effectiveness of an optimizing compiler for arithmetic expressions, *ACM SIGPLAN Notices*, 5:7, 101—126.
- Перевозчикова О. Л., Цейтлин Г. Е., Шевченко В. В. [1976], О моделях грамматик, ориентированных на двусторонний синтаксический анализ, *Кибернетика*, 3, 1—11.
- Перевозчикова О. Л., Шевченко В. В. [1976], О синтаксическом анализе и локализации ошибок, *Кибернетика*, 4, 26—33.
- Петроне [1965] (Petrone L.), Syntactic mapping of context-free languages, Information Processing—65 (IFIP Congress), 2, 590—591.
- Пол [1962] (Paul M.), A general processor for certain formal languages, Proc. ICC Symposium 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.
- Пост [1965] (Post E. L.), Absolutely unsolvable problems and relatively undecidable propositions, в сб. „The Undecidable. Basic papers in undecidable propositions, unsolvable problems, and computable functions“, под ред. Davis M., Raven Press, New York.
- Пратер [1969] (Prather R. E.), Minimal solutions of Paull—Unger problems, *Math. Syst. Theory*, 3:1, 76—85.
- Пфальц, Розенфельд [1969] (Pfaltz J. L., Rosenfeld A.), Web grammars, Proc. International Joint Conf. on Artificial Intelligence, Washington, 609—619.
- Пэр [1964] (Pair C.), Arbes, piles et compilation, *RFTI—Chiffres*, 7:3, 199—216.
- Рабин [1967] (Rabin M. O.), Mathematical theory of automata, в сб. „Mathematical aspects of computer science“, под ред. Schwartz J. T., Proc. Symp. Appl. Math., 19, 173—175.
- Рабин, Скотт [1959] (Rabin M. O., Scott D.), Finite automata and their decision problems, *IBM J. Res. Devel.*, 3, 114—125. (Русский перевод: Рабин М. О., Скотт Д., Конечные автоматы и задачи их разрешения, Кибернетический сборник, вып. 4, ИЛ, М., 1962, 56—91.)
- Редько В. Н. [1964], Об определяющей совокупности соотношений алгебры регулярных событий, *Украинский матем. журнал*, 16:1, 120—126.

- *Редько В. Н. [1969], К проблеме синтаксического анализа языков, *Кибернетика*, 1, 61—67; 3, 52—57.
- *Редько В. Н. [1970], Параметрические грамматики и проблема синтаксического анализа языков, Труды 2-й Всесоюзной конф. по программир., Засед. К, 3—19.
- Рейнольдс [1965] (Reynolds J. C.), An introduction to the COGENT programming system, Proc. ACM National Conference, 422.
- Рейнольдс, Хаскел [1970] (Reynolds J. C., Haskell R.), Grammatical coverings, неопубликованное сообщение.
- Ренделл, Рассел [1964] (Randell B., Russel L. J.), ALGOL 60 implementation, Academic Press, New York. (Русский перевод: Ренделл Б., Рассел Л., Реализация АЛГОЛа 60, изд-во „Мир“, М., 1967.)
- 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 [1967b] (Rosen S.), A compiler-building system developed by Brooker and Morris, в сб. „Programming Systems and Languages“, под ред. Rosen S., McGraw-Hill, New York, 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, М., 1970, № 16, 117—146.)
- Розенкранц, Льюис [1970] (Rosenkrantz D. J., Lewis P. M. II), Deterministic left corner parsing, IEEE Conf. Record 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 [1966a] (Salomaa A.), Two complete axiom systems for the algebra of regular events, *J. ACM*, 13:1, 158—169.
- *Саломаа [1966b], Аксиоматизация алгебры событий, реализуемых логическими сетями, Проблемы кибернетики, вып. 17, изд-во „Наука“, М., 237—246.
- Саломаа [1969a] (Salomaa A.), Theory of automata, Pergamon, Elmsford, N. Y.
- Саломаа [1969b] (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.
- *Стаевичене Л. И. [1976], Об одном алгоритме построения ограниченно-контекстных анализаторов, *ЖВМ и МФ*, 16:5, 1283—1292.
- 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, стр. 117—139.)
- *Стирнз [1971] (Stearns R. E.), Deterministic top-down parsing, Proc. 5th Annual Princeton Conference on Information Sciences and Systems, 182—186.
- Суппес [1960] (Suppes P.), Axiomatic set theory, Van Nostrand Reinhold, New York.
- Томпсон [1968] (Thompson K.), Regular expression search algorithm, *Comm. ACM*, 11:6, 419—422.

- *Трахтенброт Б. А., Барздиш Я. М. [1970], Конечные автоматы (поведение и синтез), изд-во „Наука“, М.
- *Трахтенберг Э. А., Шумей А. С. [1971], Синтаксический анализ языков, порождаемых однозначными грамматиками предшествования, *ЖВМ и МФ*, 11:4, 1005—1013.
- *Трахтенберг Э. А., Шумей А. С. [1973], Об эквивалентном преобразовании порождающих грамматик в грамматики предшествования, *ЖВМ и МФ*, 13:2, 446—455.
- *Трубчанинов Г. Г. [1976], Классы грамматик, ориентированных на синтаксический анализ методом предшествования, *Программирование*, 2, 13—18.
- 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, 544—546.
- *Ульман [1972] (Ullman J. D.), Applications of language theory to compiler design, в сб. „Currents in the theory of computing“, под ред. Aho A. V., Prentice-Hall, Englewood Cliffs, N. J., 173—218.
- 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, Грэс [1968] (Feldman J. A., Gries D.), Translator writing systems, *Comm. ACM*, 11:2, 77—113. (Русский перевод: Фельдман Дж., Грэс Д., Системы построения трансляторов, в сб. „Алгоритмы и алгоритмические языки“, вып. 5, ВЦ АН СССР, М., 1971.)
- Фишер [1968] (Fischer 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.
- Флойд [1961] (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 [1962b] (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 [1964b] (Floyd R. W.), The syntax of programming languages—a survey, *IEEE Trans. Electr. Comput.*, 8, 346—353.
- Флойд [1967a] (Floyd R. W.), Assigning meanings to programs, в сб. Mathematical aspects of computer science, под ред. Schwartz J. T., *Proc. Symp. Appl. Math.*, 19, 19—32.
- Флойд [1967b] (Floyd R. W.), Nondeterministic algorithms, *J. ACM*, 14:4, 636—644.
- *Фостер [1968] (Foster J. M.), A syntax improving device, *Computer J.*, 11:1, 31.
- *Фостер [1970] (Foster J. M.), Automatic syntactic analysis, Macdonald, London, and American Elsevier Inc., New York. (Русский перевод: Фостер Дж. М., Автоматический синтаксический анализ, изд-во „Мир“, М., 1975.)
- Фримэн [1964] (Freeman D. N.), Error correction in CORC, the Cornell computing language, *Proc. AFIPS Fall Joint Computer Conference*, 26, 15—34.

- °Фу [1974] (Fu K. S.), *Syntactical methods in pattern recognition*, Academic Press, New York.
- °Фуксман А. Л. [1968], О некоторых грамматиках для описания контекстно-свободных языков, Труды I-й Всесоюзной конференции по программированию, А 135—143.
- °Фуксман А. Л. [1976], Слаборазделенные грамматики, *ЖВМ и МФ*, 16:5, 1293—1304.
- Халмос [1960] (Halmos P. R.), *Naive set theory*, Van Nostrand Reinhold, New York.
- Халмос [1963] (Halmos P. R.), *Lectures on Boolean algebras*, Van Nostrand Reinhold, New York.
- °Хаммер [1974] (Hammer M.), A new grammatical transformation into LL(k) form, Proc. 6th Annual ACM Symposium on Theory of Computing, 266—275.
- °Хант и др. [1975] (Hunt H. B., Szymansky T. G., Ullman J. D.), On the complexity of LR(k) testing, *Comm. ACM*, 18:12, 707—716.
- 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.
- °Харрисон [1973] (Harrison M. A.), On covers and precedence analysis, *Lecture Notes in Computer Science*, 1, 2—17.
- Харрисон, Хавел [1973] (Harrison M. A., Havel I. M.), Strict deterministic grammars, *J. Comp. Syst. Sci.*, 7:3, 237—277.
- °Харрисон, Хавел [1974] (Harrison M. A., Havel I. M.), On the parsing of deterministic languages, *J. ACM*, 21:4, 525—548.
- Хартманис [1970] (Hartmanis J.), A note on one-way and two-way automata, *Math. System Theory*, 4:1, 24—28.
- Хартманис, 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.)
- Хартманис и др. [1965] (Hartmanis J., Lewis P. M. II, Stearns R. E.), Classifications of computations by time and memory requirements, Information Processing—65 (IFIP Congress), 31—35.
- Хафмен [1954] (Huffman D. A.), The synthesis of sequential switching circuits, *J. Franklin Inst.*, 257, 3—4, 161, 190, 275—303.
- Хейнс [1970] (Haines L. H.), Representation theorems for context-sensitive languages, Dept. of Electrical Engineering and Computer Sciences, Univ. of California, Berkeley.
- Хэйс [1967] (Hays D. G.), *Introduction to computational linguistics*, American Elsevier, New York.
- Хекст, Роберте [1970] (Hext J. B., Roberts P. S.), Syntax analysis by Dornolki's algorithm, *Computer J.*, 13:3, 263—271.
- Хомский [1956] (Chomsky N.), Three models for the description of language, *IEEE Trans. Inform. Theory*, 2:3, 113—124. (Русский перевод: Хомский Н., Три модели для описания языка, Кибернетический сборник, вып. 2, ИЛ, М., 1961, стр. 237—266.)
- Хомский [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.)
- Хомский [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.)
- Хомский [1959b] (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.)
- Хомский [1962] (Chomsky N.), Context-free grammars and pushdown storage, *Quarterly Progress Report*, № 65, Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Mass.
- Хомский [1963] (Chomsky N.), Formal properties of grammars, *Handbook of Mathematical Psychology*, 2, под ред. Luce R. D., Bush R. R., Galanter E., Wiley, New York. (Русский перевод: Хомский Н., Формальные свойства грамматик, Кибернетический сборник, новая серия, вып. 2, изд-во „Мир“, М., 1966, стр. 121—230.)
- Хомский [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.)
- Хомский, 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.)
- Хомский, Шютценберже [1963] (Chomsky N., Schutzenberger M. P.), The algebraic theory of context-free languages, в сб. „Computer Programming and Formal Systems“, под ред. Braffort P., Hirschberg D., North-Holland, Amsterdam. (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“, М., 1972.)
- 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.
- °Хорнинг [1974] (Hornig J. J.), LR grammars and analysers, Compiler Construction, *Lecture Notes in Computer Science*, 21, 85—108.
- °Цейтин Г. С. [1971], Алгоритм для упрощенного синтаксического анализа, Проблемы кибернетики, 24, изд-во „Наука“, М., стр. 227—242.
- °Цейтлин Г. Е., Ющенко Е. Л. [1975], Некоторые вопросы теории параметрических моделей языков и параллельный синтаксический анализ, Труды Всесоюзного симпозиума по методам реализации новых алгоритмических языков, часть 2, 61—73.
- 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 TGS-II translator-generator system, Information Processing—65 (IFIP Congress), 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, Стэндлиш [1970] (Cheatham T. E., Standish T.), Optimization aspects of compiler-compilers, *ACM SIGPLAN Notices*, 5:10, 10—17.

- Читэм, Сэтли [1964] (Cheatham T. E., Sattley K.), Syntax directed compiling, *Proc. AFIPS Spring Joint Computer Conference*, 25, 31—57.
- Чулик [1966] (Čulík K.), Well-translatable languages and ALGOL-like languages, в сб. „Formal Language Description Languages for Computer Programming“, под ред. Steel T. B., North-Holland, Amsterdam, 76—85. (Русский перевод: Чулик К., Хорошо переводимые языки и языки типа АЛГОЛ, Научно-техн. информ., сер. 2, № 3, 21—23.)
- Чулик [1968] (Čulík K., II), Contribution to deterministic top-down analysis of context-free languages, *Kybernetika*, 4:5, 422—431.
- Чулик, Кохен [1973] (Čulík K., II, Cohen R.), LR regular grammars—an extension of LR (k) grammars, *J. Comp. Syst. Sci.*, 7:1, 66—96.
- Шварц [1967] (Schwartz J. T. (ed.)), Mathematical aspects of computer science, *Proc. Symp. Appl. Math.*, 19.
- Шевченко В. В. [1974], Об одном подходе к проблеме синтаксического анализа, *Кибернетика*, 4, 30—38.
- 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. (Русский перевод: Шепердсон Дж., Сведение двусторонних автоматов к односторонним автоматам, Кибернетический сборник, вып. 4, ИЛ, М., 1962, стр. 92—98.)
- Школьник [1973] (Schkolnik M.), Labelled precedence parsing, Conf. Record ACM Symposium on Principles of Programming Languages, 33—40.
- Школьник [1974] (Schkolnik M.), The equivalence of reducing transition languages and deterministic languages, *Comm. ACM*, 17:9, 517—519.
- Шорре [1964] (Schorre D. V.), METTA II, a syntax oriented compiler writing language, *Proc. ACM National Conference*, 19, pp. 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.*, 13:4, 354—356. (Русский перевод: Штрассен Ф., Алгоритм Гаусса не оптимальен, Кибернетический сборник, новая серия, вып. 7, „Мир“, М., 1970, стр. 67—70.)
- Шумей А. С. [1975], Об использовании слабых отношений предшествования между определяющими символами в синтаксическом анализаторе, *Кибернетика*, 5, 69—76.
- Шумей А. С., Зоинис В. С. [1975], О синтаксическом анализе по однозначным грамматикам, *Программирование*, 3, 22—29.
- 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] (Evy R. J.), Applications of pushdown-store machines, *Proc. AFIPS Fall Joint Computer Conference*, 24, 215—227.
- Эйкель [1973] (Eickel J.), Methoden der syntaktischen Analyse bei formalen Sprachen, *Lecture Notes in Econom. and Math. Systems*, 78, 37—53.
- 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.
- 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., Fischer P. C.), A bibliography on computational complexity, CSRR 2028, Dept. of Applied Analysis and Computer Science, Univ. of Waterloo, Waterloo, Ontario.
- 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.)
- Эрли [1975] (Earley J.), Ambiguity and precedence in syntax description, *Acta Inform.*, 4:2, 183—192.
- 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, 104—129.
- Яжабек и Кравчик [1975] (Jazabek S., Krawczyk T.), LL-regular grammars, *Inform. Proces. Letters*, 4:2, 31—37.
- 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.)

УКАЗАТЕЛЬ ОБОЗНАЧЕНИЙ

$a \in A, a \notin A$	11
\emptyset	12
$\# A$	13
$A \subseteq B, A \subset B$	13
$\{x \mid P(x)\}$	13
$A \cup B, A \cap B, A - B, \bar{A}$	14
$\mathcal{P}(A)$	15
$A \times B$	16
R^i, R^+, R^*	18, 19
$f: A \rightarrow B$	22
A^i, A^+, A^*	24
$R_1 \circ R_2$	25
e	27
a^i	27
x^R	27
$ x $	28
L^n, L^+, L^*	29
h^{-1}	30
$P \vee Q, P \wedge Q, \sim P, P \rightarrow Q$	33
$a+b, a \cdot b, \bar{a}$ (булевы операции)	36
f_L	48
$\alpha \uparrow \beta$	49, 515, 534
$lrep(T), rrep(T)$	61
LOAD, ADD, MPY, STORE	83
$G = (N, \Sigma, P, S)$	105
$L(G)$	106, 121, 516, 527
$\Rightarrow_G, \Rightarrow, \Rightarrow^i, \Rightarrow^+, \Rightarrow^*$	106, 107, 120, 121, 515, 516, 526
$A \rightarrow \beta_1 \beta_2 \dots \beta_n$	106, 107
G_0	108
$p+q, p^*, \emptyset, e$ (регулярные выражения)	124, 125
$M = (Q, \Sigma, \delta, q_0, F)$	135
$\vdash_M, \vdash, \vdash^i, \vdash^+, \vdash^*$	135, 136, 194, 199, 255, 258, 302, 326, 327, 340—342, 379, 404, 409, 412, 415, 417—419, 453, 534, 545
$L(M)$	136, 195, 409, 412, 416, 419, 535
$q \equiv^k p, q \equiv p$	148
$D_x \alpha$	160, 161

L_1/L_2	160
$\Rightarrow_t, \Rightarrow_r$	167
$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$	193
$L_e(P)$	201
$T = (N, \Sigma, \Delta, R, S)$	248
$A \rightarrow \alpha, \beta$	249
$\Rightarrow_T, \Rightarrow_T^+, \Rightarrow_T^k, \Rightarrow_T^*$	249
$\tau(T)$	249
$\tau(M)$	255, 258, 545
$M(L), M^{-1}(L)$	257
$\tau_e(P)$	258
$\mathcal{T}, \mathcal{T}_s, \mathcal{T}_r$	273
$R^{*n}, R^{+n}, R_1 R_2$	284
$i\Rightarrow, \Rightarrow_i, \pi\Rightarrow, \Rightarrow_\pi$	297, 307
T_l^G, T_t	298
M_l^G, M_t	299
T_r^G, T_r	302
M_r^G, M_r	302
L_l^G, L_t	307
T_{lc}^G, T_{tc}	314
FIRST $_k^G(\alpha)$, FIRST (α)	337
$f(n) = 0(g(n))$	355
$[A \rightarrow \alpha \cdot \beta, i]$	359
FOLLOW $_k^G(\beta)$, FOLLOW (β)	383
$L_1 \oplus_k L_2$	388
$\Rightarrow_{tc}, \Rightarrow_{tc}^*$	403
$k - a \xrightarrow{W_i(v)} X$	411
$L_{(d,i)}, L_d$	416
$[A \rightarrow \alpha \cdot \beta, u]$	433
EFF $_k^G(\alpha)$, EFF (α)	433
$V_k^G(v), V(v)$	437
GOTO (V, X)	438
$\triangleleft, \triangleright, \triangleleft^*, \triangleright^*$	456, 457, 547, 549
$A \rightarrow BC/D$	513
$A \rightarrow B[C, D]$	525

УКАЗАТЕЛЬ ЛЕММ, ТЕОРЕМ И АЛГОРИТМОВ

Номер леммы	Стр.						
2.1	125	2.19	186	3.6	275	5.3	459
2.2	129	2.20	198	3.7	275	5.4	471
2.3	129	2.21	200	3.8	276	5.5	471
2.4	130	2.22	201	3.9	276	5.6	485
2.5	131	2.23	202	3.10	278	5.7	497
2.6	131	2.24	203	3.11	278	5.8	497
2.7	132	2.25	207	3.12	279	6.1	517
2.8	141	2.26	209	3.13	280	6.2	522
2.9	141	2.27	212	3.14	280	6.3	522
2.10	142	2.28	215	3.15	308	6.4	528
2.11	149	2.29	228	4.1	331	6.5	536
2.12	166	2.30	229	4.2	332	6.6	538
2.13	166	2.31	229	4.3	332	6.7	551
2.14	175	3.1	260	4.4	334	6.8	551
2.15	178	3.2	260	4.5	343	6.9	553
2.16	182	3.3	262	4.6	364	6.10	554
2.17	185	3.4	269	5.1	389	6.11	554
2.18	186	3.5	275	5.2	434	6.12	554
Номер теоремы	Стр.						
0.1	18	2.11	168	2.26	225	3.11	300
0.2	19	2.12	169	2.27	226	3.12	303
0.3	55	2.13	172	2.28	230	3.13	306
0.4	60	2.14	173	2.29	230	3.14	307
0.5	64	2.15	174	2.30	232	3.15	309
2.1	131	2.16	175	3.1	252	4.1	333
2.2	133	2.17	177	3.2	263	4.2	335
2.3	139	2.18	180	3.3	269	4.3	335
2.4	143	2.19	183	3.4	271	4.4	343
2.5	143	2.20	187	3.5	273	4.5	343
2.6	150	2.21	210	3.6	273	4.6	354
2.7	152	2.22	214	3.7	274	4.7	355
2.8	153	2.23	216	3.8	281	4.8	357
2.9	153	2.24	221	3.9	289	4.9	362
2.10	156	2.25	224	3.10	299	4.10	366

Номер теоремы	Стр.						
4.11	366	5.8	399	5.17	467	5.25	493
4.12	368	5.9	435	5.18	472	5.26	497
5.1	381	5.10	439	5.19	474	6.1	522
5.2	382	5.11	442	5.20	484	6.2	529
5.3	383	5.12	446	5.21	484	6.3	530
5.4	387	5.13	447	5.22	487	6.4	531
5.5	392	5.14	459	5.23	487	6.5	538
5.6	395	5.15	462	5.24	492	6.6	556
5.7	398	5.16	467				

Номер алгоритма	Стр.						
0.1	60	2.10	172	4.4	356	5.10	442
0.2	63	2.11	174	4.5	359	5.11	444
0.3	65	2.12	176	4.6	367	5.12	461
1.1	85	2.13	180	5.1	385	5.13	465
2.1	127	2.14	183	5.2	390	5.14	472
2.2	149	2.15	187	5.3	391	5.15	476
2.3	154	2.16	214	5.4	396	5.16	486
2.4	155	3.1	250	5.5	397	5.17	487
2.5	155	3.2	287	5.6	399	5.18	492
2.6	159	4.1	326	5.7	427	5.19	496
2.7	169	4.2	340	5.8	437	6.1	519
2.8	171	4.3	353	5.9	440	6.2	530
2.9	171						

- 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
 Ангер (Unger S. H.) 351
 Анисимов А. В. 408
 Арбаб (Arbib M. A.) 163
 Ахо (Aho A. V.) 123, 220, 283, 408, 409, 452, 480
- 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
 Беккер (Becker S.) 480
 Берж (Berge C.) 68
 Бжозовский (Brzozowsky J. A.) 147, 163
 Бирман (Birman A.) 542
 Блатнер (Blattner M.) 241
 Бобров (Bobrow D. G.) 102
 Бородин (Borodin A.) 52
 Брукер (Brooker R. A.) 96, 351
 Бук (Book R. V.) 123, 241
 Бут (Booth T. L.) 163
 Бэкус (Backus J. W.) 95
- Вайз (Wise D. S.) 510
 Вайнер (Weiner P.) 163
 Валиант (Valiant L. G.) 372
 Вальк (Walk K.) 74
 Вебер (Weber H.) 480
 van Вейнгаарден (van Wijngaarden A.) 75, 559
 Вегбрейт (Wegbreit B.) 75
 Вельбицкий И. В. 408, 411
 Виноград С. (Winograd S.) 48
- Виноград Т. (Winograd T.) 102
 Вирт (Wirth N.) 480, 565
 Возенкрафт (Wosencraft 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
 Гладкий А. В. 123, 241
 Гончарова Л. И. 452
 Глушков В. М. 163
 Готлиб (Gotlieb C. C.) 352
 Грин (Green M. W.) 96
 Грейбах (Greibach S.) 123, 192, 241, 305
 Грис (Gries D.) 95, 96
 Грисвold (Griswold R. E.) 562
 Гриффитс (Griffiths T. V.) 268, 351
 Гросс (Gross M.) 241
 Грэй (Gray J. N.) 220, 315, 480, 510, 558
 Грэхем Р. (Graham R. M.) 510
 Грэхем С. (Graham S. L.) 372, 480
- Денинг (Denning P. J.) 480
 Де Ремер (De Remer F. L.) 452, 569
 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
- Замельсон (Samelson K.) 510
 Зыков А. А. 68
 Зонис В. С. 452

- Ибара (Ibarra O.) 220
 Ингерман (Ingerman P. Z.) 96
 Ихбия (Ichbia J. D.) 480
- Кантор (Cantor D. G.) 241
 Камеда (Kameda T.) 163
 Касами (Kasami T.) 372
 Кауфмаи Б. III. 408, 409
 Клини (Kleene S. C.) 38, 52, 147
 Кнут (Knuth D. E.) 52, 68, 74, 408, 452, 542
 Кок (Cocke J.) 95, 96, 372
 Колмераузер (Colmerauer A.) 558
 Комор (Komor T.) 408, 409
 Конвей М. (Conway M. E.) 408, 414
 Конвей Р. (Conway R. W.) 96
 Кореняк (Korenjak A. J.) 408, 452
 Косарю (Kosarju S. R.) 163
 Коэн Д. (Cohen D. J.) 352
 Коэн Р. (Cohen R. S.) 558
 Кравчик (Krawczyk T.) 408
 Куук (Cook S. A.) 48, 220
 Кристенсен (Christensen C.) 75
 Куно (Kuno S.) 351
 Курки-Суонио (Kurki-Suonio R.) 408
- Лавров С. С. 558
 Лалонд (Lalonde W. R.) 504
 Ланти (Lentin A.) 241
 Левитт (Levitt K. N.) 96
 Лейниус (Leinius R. P.) 452, 480
 Ли Дж. (Lee J. A. N.) 96
 Ли Э. (Lee E. S.) 504
 Ливенворт (Leavenworth B. M.) 74, 559
 Лоекс (Loecks J.) 510
 Ломет (Lomet D.) 417
 Лукасевич (Lukasiewicz J.) 244
 Лукаш (Lucas P.) 74
 Льюис (Lewis P. M. II) 220, 268, 408
- Мак-Илрой (McIlroy M. D.) 74, 77
 Мак-Каллок (McCullough W. S.) 123
 Мак-Карти (McCarthy J.) 96
 Мак-Киман (McKeeman W. M.) 96, 480, 510
 Мак-Клюр (McClure R. M.) 96, 542
 Мак-Нотон (McNaughton R.) 147
 Максвелл (Maxwell W. L.) 96
 Мальцев А. И. 43, 46, 52
 Марков А. А. 42
 Мендельсон (Mendelson E.) 38
 Миллер Б. (Miller W. F.) 102
 Миллер Г. (Miller G. A.) 147

- Минский (Minsky M.) 43, 52, 122, 163
 Мицумото (Mizumoto M.) 102
 Монтанари (Montanari U. G.) 102
 Морган (Morgan H. L.) 96, 296
 Морзе (Morse S. P.) 480
 Моррис (Morris D.) 96, 351
 Моултон (Moulton P. G.) 96
 Муиро (Munro I.) 68
 Мур (Moor E. F.) 123, 162
 Мюллер (Muller M. E.) 96
- Наур (Naur P.) 74, 559
 Непомнящая А. Ш. 408
 Никитченко Н. С. 408
 Нийхольт (Nijholt A.) 408
- Огден (Ogden W.) 241
 Ордян А. А. 558
 Оре (Ore O.) 68
- Павлидис (Pavlidis T.) 102
 Парик (Parikh R. J.) 241
 Паул (Paul M. C.) 192
 Пейнтер (Painter J. A.) 96
 Перль (Perles M.) 241
 Перлис (Perlis A. J.) 74
 Петрик (Petrick S. R.) 351
 Петроне (Petrone L.) 268
 Питтс (Pitts E.) 123
 Пол (Paul M.) 510
 Полонский (Polonsky I. P.) 562
 Портер (Porter J. H.) 295
 Пост (Post E. L.) 42, 52
 Поудж (Poage J. F.) 562
 Пратер (Prather R. E.) 163
 Пфальц (Pialtz J. L.) 98, 102
 Пер (Pair C.) 480
- Рабин (Rabin M. O.) 123, 147, 162
 Райс (Rice H. G.) 192
 Рассел (Russel L. J.) 96
 Редько В. Н. 147, 408
 Рейнольдс (Reynolds J. C.) 96, 315, 351
 Ренделл (Randell B.) 96
 Робертс (Roberts P. S.) 352
 Роджерс (Rogers H.) 46
 Розен (Rosen S.) 95, 351
 Розенкранц (Rosenkrantz D. J.) 123, 192, 408
 Розенфельд (Rosenfeld A.) 98, 102
 Росс (Ross D. T.) 295
 Роудз (Rhodes E. M.) 480
 Руццо (Ruzzo W. L.) 372

- Саломаа (Salomaa A.) 147, 163, 241
 Саммет (Sammet J. E.) 42, 74
 Скотт (Scott D.) 123, 147, 162
 Стил (Steel T. B.) 74
 Стирнз (Stearns R. E.) 220, 241, 268, 408
 Стендиши (Standish T.) 96
 Суннес (Suppes P.) 13
 Сэтли (Sattley K.) 351
- Танака (Tanaka K.) 102
 Тёёда (Tojoda J.) 102
 Томпсон (Thompson K.) 163, 296
 Тории (Torii K.) 372
 Трахтенброт Б. А. 163
 Трахтенберг Э. А. 510
 Трубчанинов Г. Г. 510
 Тьюриг (Turing A. M.) 42, 43, 51, 123
- Уллиан (Ullian J. S.) 241
 Ульман (Ullman J. D.) 52, 123, 220, 241, 283, 408, 409, 452, 480, 542
 Уолтерс (Walters D. A.) 452
 Уорли (Worley W. S.) 96
 Уортман (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
 Фишер П. (Fischer P. C.) 52
 Флойд (Floyd R. W.) 68, 96, 192, 241, 351, 480, 510
 Фостер (Foster J. M.) 408
 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
 Халмуш (Halmos P. R.) 13, 38
 Хант (Hunt H. B.) 452
 Харари (Harary F.) 68
 Харрисон (Harrison M. A.) 163, 220, 315, 372, 510, 558
 Хартманис (Hartmanis J.) 52, 147, 220
 Хаскел (Hackell R.) 315
 Хаффмен (Huffman D. A.) 162
 Хейес (Haines L. H.) 123
- Хейс (Hays D. G.) 372
 Хекст (Hext J. B.) 352
 Хомский (Chomsky N.) 42, 74, 102, 123, 147, 192, 220, 241
 Хопгуд (Hoggood F. R. A.) 96, 510
 Хопкрофт (Hopcroft J. E.) 52, 123, 163, 220, 241, 408, 452
 Хорнинг (Hornung J. J.) 96, 504, 510
 Хохшпруг (Hochsprung R. R.) 96
- Цейти Г. С. 102
- Чёрч (Church A.) 38, 42, 43
 Читэм (Chearham T. E.) 74, 96, 315, 351
 Чуллик I (Čulík K.) 268
 Чуллик II (Čulík K. II) 408, 558
- Шамир (Shamir E.) 241
 Шапиро (Shapiro R. M.) 96
 Шварц (Schwartz J. T.) 95, 96
 Шевченко В. В. 408
 Шепердсон (Shepherdson J. C.) 147
 Шыманский (Szymansky T. G.) 452
 Шильяяк С. С. 408
 Шорре (Schorre D. U.) 96, 351, 542
 Шоу (Shaw A. C.) 98, 102
 Штрассен (Strassen V.) 48, 68
 Шумей А. С. 452, 510
 Шютценберже (Schutzenberger M. P.) 192, 220, 241
- Эванс (Evans A. Jr.) 510, 569
 Эви (Evey R. J.) 192, 220
 Эйкель (Eickel J.) 510
 Экли (Ackley S. I.) 295
 Элспас (Elspas B.) 96
 Энгелер (Engeler E.) 74
 Эрланд (Ireland M. I.) 52
 Эрли (Earley J.) 372
 Эттингер (Ettinger A.) 220, 351
- Ющенко Е. Л. 408, 411
- Яжабек (Jarzabek S.) 408
 Я마다 (Yamada) H. 147
 Янгер (Younger D. H.) 372

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Авицепочка (lookahead string) 378, 427 *см. также* Заглядывание вперед
Автомат (automaton) *см. также* Распознаватель, Преобразователь
 — 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 магазинной памятью (pushdown) 114, 193—220, 318
 — двусторонний (two-way) 219, 220
 — детерминированный (deterministic) 211—220, 228, 229, 237—240, 283, 384, 450, 501, 503, 522—525
 — — — дочитывающий (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 (Domolki's) 351, 352, 507, 510
 — Кока — Янгера — Касами (Cocke — Younger — Kasami) 352—358
 — Маркова 42
 — недетерминированный (nondeterministic) 320, 346, 347, 351
 — разбора, предсказывающий (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*
 — выходной (output) *см. Символ выходной*
 — состояний (of states) *см. Состояние*
Альтернатива (нетерминала) (alternate) 321
Анализ (analysis)
 — лексический (lexical) 76—79, 283—296
 — — непрямой (indirect) 78, 286—290
 — — прямой (direct) 78, 290—293
 — синтаксический (syntactic) *см. Разбор*

Анализатор (parser) *см. также* Аналisis
 — 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-схема (flow chart, d-chart) 98—101

Веер (в дереве) *см. Куст (в дереве)*
Вершина (графа) (node, vertex) 52
 — концевая *см. Лист*
Включение (множеств) (inclusion) 13, 238
Вход (input) 38 *см. также* Лента входная
Выход (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
 — инфиксное (infix) 244
 — постфиксное (postfix) 244, 245, 247, 248, 259, 264, 267, 529
 — префиксное (prefix) 244, 245, 259, 264, 267
 — расширенное регулярное (extended regular) 284—290
 — регулярное (regular) 124—131, 145, 147
Высота (вершины дерева) (height) 58, 84
Выход (output) 38, 243, 255, 258

Генерация кода (code generation) 75, 82—88, 90, 92, 93
Глубина (вершины дерева) (depth) 58
Головка входная (распознавателя) (input head) 113—115
Гомоморфизм (homomorphism) 29, 225, 236, 238, 239, 243, 244
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, 314, 340, 343, 346, 367
 — бессконтекстная *см. Грамматика контекстно-свободная*
 — входная (СУ-схемы) (input underlying) 250
 — выходная (output) 250
 — индексная (indexed) 120, 121
 — Колмерауэр (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
 — леволинейная (left linear) 145
 — леворекурсивная (left recursive) 178—181, 324, 325, 331, 332, 385, 400
 — линейная (linear) 191, 237, 268
 — LC(k) 402—408
 — LL 376
 — LL(k) 301, 373—408, 449, 450
 — LL(1) 373, 382—387, 408, 411, 539

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- Грамматика LR(k) 304, 373, 421, 423—452, 478, 482, 484, 503
 — LR(1) 424, 463, 503, 504
 — неоднозначная (ambiguous) 168, 189, 231—236, 239, 317, 546 см. также Грамматика однозначная
 — неукорачивающая см. Грамматика контекстно-зависимая и Грамматика без ϵ -правил
 — общего вида (unrestricted) 105—112, 118—120, 122
 — обратимая (unique invertible) 422, 450, 457, 503—506, 547, 557
 — 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
 — операторная (operator) 190, 492
 — операторного предшествования (operator precedence) 493—497, 503, 504, 507
 — (1,1)-ОПК ((1,1)-bounded-right-context) 484, 503
 — остаточная (skeletal) 496, 507
 — (2,1)-предшествования ((2,1)-precedence) 480, 503
 — правоанализируемая (right parsable) 304—306, 450
 — 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
 — предшествования (precedence) 456, 457, 463, 480
 — приведенная (proper) 175
 — 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
 — псевдоразделенная 409, 410, 414, 420
 — разделенная см. Грамматика простая LL(1)
 — расширенного предшествования (extended precedence) 463—469, 478—480, 484, 505
 — регулярная (regular) 145, 557
 — рекурсивная (recursive) 178
 — сильно LL(k) (strong LL(k)) 384, 388
 — слабого предшествования (weak precedence) 469—477, 479, 491, 492, 503—506
 — слаборазделенная 410, 411, 414
 — смешанной стратегии предшествования (mixed-strategy precedence) 488—492, 503, 507
 — с самовставлением (self-embedding) 240
 — T -канонического предшествования (T -canonical precedence) 507—510
 — T -остаточная (T -skeletal) 509
 — Хомского (Chomsky) см. Грамматика
 Граф (graph) 52—68
 — ациклический (ориентированный) (directed acyclic, dag) 54, 138
 — нагруженный см. Граф помеченный
 — неориентированный (undirected) 66
 — переходов (автомата) (transition) см. Диаграмма конечного автомата
 — помеченный (labelled) 53, 57
 — связный (неориентированный) (connected) 66
 — сильно связный (strongly connected) 54
 — упорядоченный (ordered) 56
 — ациклический (dag) 57
 Дерево (tree) 55, 61, 62, 71—73, 81—87, 100, 101, 319, 487—490
 — вывода (derivation) см. Дерево разбора
 — неориентированное (undirected) 66, 67
 — разбора (parse) 164—168, 205—207, 250—252, 307, 431, 432, 519—521

- Дерево 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
 Длина (length)
 — вывода (of a derivation) 107
 — цепочки (of a string) 28
 ДМП-автомат см. Автомат с магазинной памятью детерминированный
 Дополнение (множества) (complementation) 14, 216, 226, 237, 541
 Допускать (цепочку, языков) (accept) 115, 136, 195, 201
 Дуга (в графе) (arc, edge) 53

 ϵ -правило (грамматики) (ϵ -production) 111, 177, 178, 340, 402
 ϵ -такт (распознавателя) (ϵ -move) 194, 218

 Заглядывание вперед (lookahead) 337, 344, 370, 371, 373—375, 378, 402, 421, 424, 450
 Законы де Моргана (De Morgan's laws) 23
 Замкнутость (относительно операций) (closure) 152, 224—226, 257, 266
 Замыкание (отношения) (closure)
 — рефлексивное и транзитивное (reflexive and transitive) 19
 — транзитивное (transitive) 18, 62—65, 68
 Запись польская (Polish notation) см. Выражение префиксное

 Идентификатор (identifier) 76—80, 116, 286, 287, 289—293
 Иерархия Хомского (Chomsky hierarchy) 112
 Индекс
 — грамматики, языка 239, 240
 — отношения эквивалентности (index) 17
 Исправление ошибок (error correction) 75, 90—93, 96, 337, 338, 407, 446, 451, 452, 480
 Исчисление высказываний (propositional calculus) 35, 50
 Итерация (языка) (closure) 29, 225
 — маркирования (marked) 240
 — позитивная (positive) 29

 К3-грамматика см. Грамматика контексто-зависимая
 Код (code)
 — объектный (object) 75, 82, 242
 — промежуточный (intermediate) 75, 82—87
 Компилятор (compiler) 75—96, 351, 408
 — компиляторов (compiler-compiler) 96, 351
 Композиция (отношений) (composition) 25, 281
 Конкатенация (concatenation) 27, 29, 225, 238
 — 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
 — допускающая (accepting) см. Конфигурация заключительная
 — заключительная (final) 115, 135, 195, 201, 255, 258, 259, 379
 — зацикливающая (looping) 213—216
 — начальная (initial) 115, 135, 194, 326, 340, 379

Конфликт
 — отношений предшествования (precedence conflict) 472, 473
 — правил 389, 393
 Крома (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, 523, 541
 Матрица (matrix)
 — предшествования (precedence) 457—459
 — смежностей (adjacency) 62, 63
 Машина (machine)
 — анализирующая (parsing) 533—538, 540
 — 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)
 Множество (set) 11—30
 — бесконечное (infinite) 22, 26
 — вполне упорядоченное (well ordered) 24, 30
 — знаменательных символов (token) 508
 — конечное (finite) 12, 22
 — линейное (linear) 239
 — непротиворечивое LR(k)-ситуаций (consistent set of LR(k) items) 442, 443
 — полулинейное (semi-linear) 239
 — пустое (empty) 12
 — регулярное (regular) 124—163, 218, 219, 225, 236—238, 257, 266, 269, 270, 400, 478
 — 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
 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

Неоднозначность существенная (inherent) см. Язык неоднозначный
 Нетерминал (nonterminal) 105, 120, 248, 512—514

Обратимость (unique invertibility) см. Грамматика обратимая
 Обращение
 — гомоморфизма (inverse homomorphism) 30
 — цепочки или языка (reversal) 27, 144, 153
 Объединение (union) 14, 225, 229—231, 238, 541
 — маркированное (marked) 240
 ОК-грамматика см. Грамматика ограниченногоконтекста
 Операция элементарная (алгоритма) (elementary operation) 355, 357, 364—366, 447
 ОПК-грамматика см. Грамматика ограниченногоправого контекста
 Определение регуляриос (regular definition) 285, 286
 Оптимизация кода (code optimization) 75, 88—90
 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
 — антисимметричес (antisymmetric) 21
 — асимметричес (asymmetric) 20
 — иррефлексивное (irreflexive) 20
 — конгруэнтиес (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
 — симметричес (symmetric) 17
 — транзитивное (transitive) 17
 — эквивалентности (equivalence) 17, 18, 24, 149—151, 157, 158
 — — правоинвариантное (right invariant) 157, 158

Память (распознавателя) (memory) 113—115
 Пара цепочек выводимая (translation form) 246—249
 Перевод (translation) 71, 242—245, 258, 379, 380, 545
 — регулярий (regular) см. Преобразование конечное
 — синтаксически управляемый (syntax directed) 74, 83—88, 246—253, 260—282, 499, 569—574
 — простой (simple) 253, 260—263, 271—274, 282, 298, 569—574
 Пересечение (множеств) (intersection) 14, 225, 226, 230, 237, 541
 ПЛ/1 (PL/1) 74, 76, 78, 559
 ПЛ 360 (PL 360) 565—569
 Подстановка (языков) (substitution) 224, 225
 Позиция (в цепочке) (position) 220
 Покрытие (grammatiki) (cover) 309—311, 314, 315
 — левое (left) 310, 314, 315, 345
 — правое (right) 310, 314, 315, 345
 Порядок (как отношение) (order) 20, 21
 — лексикографический (lexicographic) 25, 331, 343
 — линейный (linear) 21, 25, 59, 60
 — обратный (вершин дерева) (postorder) 59
 — полный (well) 24, 30
 — прямой (вершин дерева) (preorder) 58

- Порядок частичный (partial) 20, 21, 25, 26, 59, 60
Порядок (схемы СУ-перевода) (order) 274—281
Потомок (в графе) (descendant) 55
Правило (грамматики) (production) 106, 120
— цепное (single) 173, 174, 507
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
— — 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
— — — расширенный (extended) 302—304, 421—423
Предфикс (цепочки) (prefix) 28
— активный (правовыводимой цепочки) (viable) 432, 444
Приоритет (операций) (precedence) 82, 168, 169, 264
Проблема (алгоритмическая) (problem) 43—52
— неразрешимая (undecidable) 44
— остановки (halting) 50
— принадлежности (membership) 154—156, 161, 162, 227
— 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
Программа (program)
— исходная (source) 75, 93, 242
— объектная (object) см. Код объектный
Продукция (production) см. Правило (грамматики)
Π: произведение декартово (Cartesian product) 16
Производная (регуляриного выражения) (derivative) 160, 161
Путь (в графе) (path) 54, 66

- Разбор (как синтаксический анализ) (parsing) 72, 75, 80—82, 90—93, 296—309
— восходящий (bottom-up) 205—209, 301—309, 338—344, 542—558 см. также Грамматика предшествования, LR(k), ОПК
— нисходящий (top-down) 205, 297—301, 304—309, 321—338, 499, 511—542 см. также Грамматика LL(k)
— по текущему символу 408—419
— по левому участку (left corner) 313, 314, 348—350, 403—406
— с возвратами (backtrack) 317—352, 511—558
— типа «перенос—свертка» (shift—reduce) 303, 338, 350, 351, 420—423, 426, 427
Разбор (как результат синтаксического анализа) (parse) 297, 379, 544
— 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

- 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
 Рубеж (в выводимой цепочке) (border) 374, 421
- Свертка (цепочки) (reduction)** 205, 302, 338, 339, 344 *см. также* Разбор типа «перенос—свертка»
Свойство (property)
 - префиксное (prefix) 29, 31, 239, 289
 - суффиксное (suffix) 29, 31**Связка логическая (logical connective)** 33—35
Семантика (semantics) 71—74, 243—246
Сечениe (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
 - 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)
 - Поста каноническая (Post's canonical) 42, 123
 - 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**Слово (word) *см.* Цепочка**
Сложность вычисления (временная и смкостная) (computational complexity, time complexity, space complexity) 41, 158, 159, 161, 162, 189, 333—337, 343, 355—357, 364—366, 368—372, 395, 447, 460, 530—532, 557
Сибол (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
 - достижимое (accessible) 140, 148
 - заключительное (final) 135, 194, 254, 326, 412
 - начальное (initial) 135, 194, 254, 412**Состояния неразличные (конечного автомата) (indistinguishable states)** 148

Список (list)

- магазинный (pushdown) см. Магазин
- разбора (parse) 359
- 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
- Сцепление (concatenation) см. Конкатенация

Таблица (table)

- идентификаторов (symbol) см. Таблица имен
- имён (symbol) 75, 79, 80, 92, 93, 287
- разбора (parse) 352
- управляющая разбором (parsing) 379, 385—387, 391—395, 404, 405 см. также LR(k)-таблица
- LL(k) 389—391, 394, 395
- LR(k) 426, 427, 444—446, 451
- ТАГ-система (Tag system) 42, 122
- Такт (распознавателя) (timestep) 115, 135, 194
- Тезис Чёрча—Тьюринга (Church-Turing thesis) 43
- Теорема (theorem) 32
 - Парника (Parikh's) 239, 241
- Терминал (в грамматике) (terminal) 106, 120, 514
- Точка наименьшая неподвижная (minimal fixed point) 127, 129—131, 144—146, 185, 186
- Транслайтор (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
- Уровни (вершины дерева) (level) см. Высота (вершины дерева)
- Устройство управляющее с конечной памятью (finite control) 114, 498 см. также Состояние (распознавателя)

Факторизация левая (left factoring) 385**Форма (form)**

- Бэкуса—Наура (Backus—Naur) 74
- нормальная Грайбах (Greibach normal) 182—188, 190, 274, 315, 402, 406
- слабая 409
- нормальная Хомского (Chomsky normal) 176, 177, 190, 273, 274, 310, 311, 314, 352, 401
- Фортран (FORTRAN) 70, 74, 77, 79, 236, 286, 346, 347, 559
- 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

Функция инъективная (injective) 22

- общекурсивная (total recursive) 41
- переходов (goto) 426—428, 444—446
- переходов (состояний автомата) (state transition) 135
- преобразования памяти (распознавателя) (fetch) 114
- рекурсивная (recursive) 41
- характеристическая (characteristic) 48
- частичная (partial) 22
- частично рекурсивная (partial recursive) 41
- EFF 433, 444, 450
- FIRST 337, 375, 376, 396—399
- FOLLOW 383, 479
- GOTO 438—441, 444

Цепочка (string) 27, 106

- выводимая (sentential form) 106
- левовыводимая (left sentential form) 167
- правовыводимая (right sentential form) 167, 459, 460, 469—471
- пустая (empty) 27

Цикл в графе (cycle, circuit) 54

- — грамматике см. Грамматика бес циклов

Часть (portion)

- (левовыводимой цепочки) законченная (closed) 374
- незаконченная (open) 374
- (правовыводимой цепочки) замкнутая (closed) 421
- открытая (open) 421

Эквивалентность (программ, автоматов, грамматик) (equivalence) 71, 147—150, 154—156, 162 см. также Проблема эквивалентности

Элемент перевода (translation element) 246—248

Язык (language) 28, 103, 104, 116, 136, 195 см. также Множество

- ассемблера (assembly) 82—88
- 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
- линейный (linear) 191, 237, 268
- LL 376
- LCK(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 ограничениями возвратами (ЯНРОВ) (top down parsing language, TDPL) 512—525, 528—530, 539—542
- обобщенный (ОЯНРОВ) (generalized, GTDPL) 525—542
- ограниченного контекста (bounded context) 505, 507
- правового контекста (bounded right context) 481—488, 503—507

- Язык одиозначный (*unambiguous*) 234, 240 см. также Грамматика одиозначая
— операторного предшествования (*operator precedence*) 493—497, 503, 504, 507
— (I, I)-ОПК ((I, I)-*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
— расширяемый (*extensible*) 74, 75, 559—562
— регулярный (*regular*) см. Множество регулярное
— существенно неодиозначный (*inherent ambiguous*) см. Язык неодиозначный
— Флойда—Энисса (*Floyd-Evans*) 411, 497—502, 507, 510
— характеризующий (*characterizing*) 269—273, 282
ЯНРОВ см. Язык исходящего разбора с ограниченными возвратами

ОГЛАВЛЕНИЕ

ОТ РЕДАКТОРА ПЕРЕВОДА 5

ПРЕДИСЛОВИЕ 7

O ПРЕДВАРИТЕЛЬНЫЕ МАТЕМАТИЧЕСКИЕ СВЕДЕНИЯ 11

- 0.1. Основные понятия теории множеств 11
 0.1.1. Множества 11
 0.1.2. Операции над множествами 14
 0.1.3. Отношения 16
 0.1.4. Замыкание отношений 18
 0.1.5. Отношения порядка 20
 0.1.6. Отображения 21
 Упражнения 23
- 0.2. Множества цепочек 26
 0.2.1. Цепочки 27
 0.2.2. Языки 28
 0.2.3. Операции над языками 29
 Упражнения 30
- 0.3. Некоторые понятия математической логики 31
 0.3.1. Доказательства 31
 0.3.2. Доказательство индукцией 32
 0.3.3. Логические связки 33
 Упражнения 35
 Замечания по литературе 38
- 0.4. Алгоритмы (частичные и всюду определенные) 38
 0.4.1. Частичные алгоритмы 38
 0.4.2. Алгоритмы 40
 0.4.3. Рекурсивные функции 41
 0.4.4. Задание алгоритмов 42
 0.4.5. Проблемы 43
 0.4.6. Проблема соответствий Поста 47
 Упражнения 48
 Замечания по литературе 51

ОГЛАВЛЕНИЕ

0.5. Некоторые понятия теории графов	52
0.5.1. Ориентированные графы	52
0.5.2. Ориентированные ациклические графы	54
0.5.3. Деревья	55
0.5.4. Упорядоченные графы	56
0.5.5. Индукция по ациклическому графу	58
0.5.6. Вложение частичного порядка в линейный	59
0.5.7. Представления деревьев	61
0.5.8. Пути в графе	62
Упражнения	66
Замечания по литературе	68

1 ВВЕДЕНИЕ В КОМПИЛЯЦИЮ 69

1.1. Языки программирования	69
1.1.1. Задание языков программирования	69
1.1.2. Синтаксис и семантика	71
Замечания по литературе	74
1.2. Обзор процесса компиляции	75
1.2.1. Основные части компилятора	75
1.2.2. Лексический анализ	76
1.2.3. Работа с таблицами	79
1.2.4. Синтаксический анализ	80
1.2.5. Генерация кода	82
1.2.6. Оптимизация кода	88
1.2.7. Анализ и исправление ошибок	90
1.2.8. Резюме	91
Упражнения	92
Замечания по литературе	95
1.3. Другие приложения алгоритмов разбора и перевода	96
1.3.1. Естественные языки	97
1.3.2. Структурное описание образов	98
Замечания по литературе	102

2 ЭЛЕМЕНТЫ ТЕОРИИ ЯЗЫКОВ 103

2.1. Способы определения языков	103
2.1.1. Мотивировка	104
2.1.2. Грамматики	105
2.1.3. Грамматики с ограничениями на правила	111
2.1.4. Распознаватели	112
Упражнения	116
Замечания по литературе	123
2.2. Регулярные множества, их распознавание и порождение	124
2.2.1. Регулярные множества и регулярные выражения	124
2.2.2. Регулярные множества и праволинейные грамматики	131
2.2.3. Конечные автоматы	134

2.2.4. Конечные автоматы и регулярные множества	141
2.2.5. Резюме	153
Упражнения	143
Замечания по литературе	147
2.3. Свойства регулярных множеств	147
2.3.1. Минимизация конечных автоматов	147
2.3.2. Лемма о разрастании для регулярных множеств	151
2.3.3. Свойства замкнутости регулярных множеств	152
2.3.4. Разрешимые проблемы, связанные с регулярными множествами	154
Упражнения	156
Замечания по литературе	162
2.4. Ко контексто-свободные языки	163
2.4.1. Деревья выводов	163
2.4.2. Преобразования КС-грамматик	168
2.4.3. Нормальная форма Хомского	176
2.4.4. Нормальная форма Грейбах	178
2.4.5. Другой метод преобразования к нормальной форме Грейбах	184
Упражнения	188
Замечания по литературе	192
2.5. Автоматы с магазинной памятью	192
2.5.1. Основное определение	193
2.5.2. Варианты МП-автоматов	198
2.5.3. Эквивалентность МП-автоматов и КС-грамматик	203
2.5.4. Детерминированные МП-автоматы	211
Упражнения	217
Замечания по литературе	220
2.6. Свойства контексто-свободных языков	220
2.6.1. Лемма Огдея	220
2.6.2. Свойства замкнутости класса КС-языков	227
2.6.3. Результаты о разрешимости	227
2.6.4. Свойства детерминированных КС-языков	230
2.6.5. Неоднозначность	231
Упражнения	236
Замечания по литературе	241

3 ТЕОРИЯ ПЕРЕВОДА 242

3.1. Формализмы, используемые для определения перевода	242
3.1.1. Перевод и семантика	242
3.1.2. Схемы синтаксически управляемого перевода	246
3.1.3. Конечные преобразователи	254
3.1.4. Преобразователи с магазинной памятью	258
Упражнения	264
Замечания по литературе	268

3.2. Свойства синтаксически управляемых переводов	268
3.2.1. Характеризующие изыски	269
3.2.2. Свойства простых СУ-переводов	273
3.2.3. Иерархия СУ-переводов	274
Упражнения	281
Замечания по литературе	283
3.3. Лексический анализ	283
3.3.1. Язык расширенных регулярных выражений	284
3.3.2. Непрямой лексический анализ	286
3.3.3. Прямой лексический анализ	290
3.3.4. Программное моделирование конечных преобразователей	293
Упражнения	294
Замечания по литературе	295
3.4. Синтаксический анализ	296
3.4.1. Определение разбора	296
3.4.2. Нисходящий разбор	297
3.4.3. Восходящий разбор	301
3.4.4. Сравнение нисходящего разбора с восходящим	304
3.4.5. Грамматическое покрытие	309
Упражнения	315
Замечания по литературе	

4**ОБЩИЕ МЕТОДЫ СИНТАКСИЧЕСКОГО АНАЛИЗА** 316

4.1. Синтаксический анализ с возвратами	317
4.1.1. Моделирование МП-преобразователя	317
4.1.2. Неформальное описание нисходящего разбора	321
4.1.3. Алгоритм нисходящего разбора	325
4.1.4. Временная и емкостная сложность нисходящего анализатора	333
4.1.5. Восходящий разбор	338
Упражнения	344
Замечания по литературе	351
4.2. Табличные методы синтаксического анализа	352
4.2.1. Алгоритм Кока—Яигера—Касами	352
4.2.2. Алгоритм Эрли	358
Упражнения	369
Замечания по литературе	372

5**ОДНОПРОХОДНЫЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ БЕЗ ВОЗВРАТОВ** 373

5.1. LL(k)-грамматики	374
5.1.1. Определение LL(k)-грамматики	374
5.1.2. Предсказывающие алгоритмы разбора	378
5.1.3. Следствия определения LL(k)-грамматики	382

5.1.4. Разбор для LL(1)-грамматик	387
5.1.5. Разбор для LL(k)-грамматик	388
5.1.6. Проверка LL(k)-условия	396
Упражнения	400
Замечания по литературе	408
Дополнение. О методах разбора „по текущему символу“. В. Н. Агафонов	408
5.2. Детерминированный восходящий синтаксический анализ	420
5.2.1. Разбор с помощью детерминированного алгоритма типа „перенос—свертка“	420
5.2.2. LR (k)-грамматики	423
5.2.3. Следствия определения LR (k)-грамматики	432
5.2.4. Проверка LR (k)-условия	442
5.2.5. Детерминированные правые анализаторы для LR(k)-грамматик	443
5.2.6. Реализация LL(k)- и LR(k)-анализаторов	448
Упражнения	448
Замечания по литературе	452
5.3. Грамматики предшествования	452
5.3.1. Формальное определение алгоритма типа „перенос—свертка“	452
5.3.2. Грамматики простого предшествования	455
5.3.3. Грамматики расширенного предшествования	463
5.3.4. Грамматики слабого предшествования	469
Упражнения	477
Замечания по литературе	480
5.4. Другие классы грамматик, анализируемых методом „перенос—свертка“	481
5.4.1. Грамматики ограниченного правого контекста	481
5.4.2. Грамматики смешанной стратегии предшествования	488
5.4.3. Грамматики операторного предшествования	492
5.4.4. Язык Флойда—Эванса	497
5.4.5. Резюме	502
Упражнения	505
Замечания по литературе	510

6**АЛГОРИТМЫ РАЗБОРА С ОГРАНИЧЕННЫМИ ВОЗВРАТАМИ** 511

6.1. Нисходящий разбор с ограниченными возвратами	511
6.1.1. Язык нисходящего разбора с ограниченными возвратами	512
6.1.2. ЯНРОВ и детерминированные КС-языки	522
6.1.3. Обобщенный ЯНРОВ	525
6.1.4. Временная сложность ОЯНРОВ-языков	530
6.1.5. Реализация ОЯНРОВ-программ	533
Упражнения	539
Замечания по литературе	542

6.2. Восходящий разбор с ограниченными возвратами	542
6.2.1. Неканонический разбор	542
6.2.2. Анализаторы с двумя магазинами	544
6.2.3. Отношения предшествования Колмерауэра	547
6.2.4. Проверка условий предшествования Колмерауэра	549
Упражнения	556
Замечания по литературе	558
Приложение	559
П.1. Синтаксис расширяемого языка	559
П.2. Синтаксис операторов языка Снобол 4	562
П.3. Синтаксис ПЛ 360	565
П.4. Схема синтаксически управляемого перевода для языка PAL	569
Список литературы	575
Указатель обозначений	590
Указатель лемм, теорем и алгоритмов	591
Именной указатель	593
Предметный указатель	596

УВАЖАЕМЫЙ ЧИТАТЕЛЬ!

Ваши замечания о содержании книги, ее оформлении, качестве перевода и другие просим присыпать по адресу: 129820, Москва, И-110, ГСП, 1-й Рижский пер., д. 2, издательство «Мир».