

БИБЛИОТЕЧКА
ПРОГРАММИСТА

И. Л. БРАТЧИКОВ

Синтаксис языков программирования



БИБЛИОТЕЧКА
ПРОГРАММИСТА

И. Л. БРАТЧИКОВ

СИНТАКСИС ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Под редакцией
С. С. ЛАВРОВА



ИЗДАТЕЛЬСТВО «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
МОСКВА 1975

517
Б 63
УДК 519-95

357
Б874



Синтаксис языков программирования,
И. Л. Братчиков, Серия «Библиотечка
программиста». Главная редакция
физико-математической литературы
изд-ва «Наука», М., 1975.

В настоящей книге освещаются на-
иболее существенные синтаксические осо-
бенности алгоритмических языков и ис-
пользуемые в настоящее время методы
их описания и изучения. Предполага-
ется знание читателями языка АЛГОЛ-60.
Примеры из других языков снабжаются
подробными пояснениями.

Книга рассчитана на системных про-
граммистов, а также на студентов,
аспирантов и инженеров, интересующих-
ся аналогичными вопросами.

Игорь Леонидович Братчиков

СИНТАКСИС ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

М., 1975 г., 232 стр с илл.

(Серия: «Библиотечка программиста»)

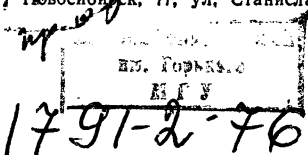
Редактор Г. Я. Пирогова
Техн. редакторы Н. В. Кошелев, Н. Я. Мирошова
Корректор Е. В. Сидоркина

Сдано в набор 1/IV 1975 г. Подписано к печати 8/IX 1975 г. Бумага 84×108¹/₃₂.
Физ. печ. л. 7,25. Условн. печ. л. 12,18. Уч.-изд. л. 12,41. Тираж 45 000 экз.
Т-13173. Заказ № 507. Цена книги 74 коп.

Издательство «Наука»

Главная редакция физико-математической литературы
117071, Москва, В-71, Ленинский проспект, 15

4-я типография изд-ва «Наука», Новосибирск, 77, ул. Станиславского, 25



Б 20204—088 63-75
053(02)-75

© Главная редакция
физико-математической литературы
издательства «Наука», 1975 г.

ОГЛАВЛЕНИЕ

Предисловие	4
Введение	7
§ 1. Язык как объект изучения средствами математики	7
§ 2. Общая классификация алгоритмических языков	12
ГЛАВА I. Порождающие грамматики	26
§ 1. Нормальные формы Бекуса	27
§ 2. Определение и общие свойства порождающих грамматик	32
§ 3. Некоторые свойства контекстно-свободных грамматик	45
§ 4. Автоматные грамматики	76
ГЛАВА II. Методы распознавания и анализа языков	85
§ 1. Машины Тьюринга как распознающие устройства	86
§ 2. Свойства конечных автоматов	104
§ 3. Автоматы с магазинной памятью	108
§ 4. Преобразователи с магазинной памятью	125
ГЛАВА III. Построение анализаторов контекстно-свободных языков по порождающим грамматикам	133
§ 1. Анализаторы предшествования	135
§ 2. $LR(k)$ -анализаторы	145
§ 3. Глобальный анализатор	161
ГЛАВА IV. Контекстные условия языков программирования	171
§ 1. Классификация контекстных условий	171
§ 2. Описание контекстных условий с помощью программных грамматик	184
§ 3. Грамматики ван Вейнгаардена	197
§ 4. Другие способы описания контекстных условий	211
Заключение	224
Литература	229

ПРЕДИСЛОВИЕ

Языки программирования или, как их часто называют, алгоритмические языки, являясь средством общения человека с электронными вычислительными машинами, начали развиваться одновременно с появлением последних, т. е. примерно три десятилетия назад. С момента своего появления языки программирования, а также методы их описания, изучения и использования интенсивно развивались и в настоящее время составляют обширную и важную область прикладной математики. Более того, можно с уверенностью сказать, что к происходящей сейчас технической революции, связанной с возникновением и развитием электронных вычислительных машин, алгоритмические языки имеют самое непосредственное отношение. Именно от них в значительной мере зависит эффективность использования вычислительных машин для решения задач как научно-исследовательской, так и хозяйственной деятельности человека.

Имеются две категории специалистов, непосредственно связанных с языками программирования.

К первой можно отнести лиц, использующих ЭВМ для решения задач, возникающих в процессе их работы. Для того, чтобы подготовить задачу к решению на ЭВМ, нужно описать алгоритм ее решения на некотором языке программирования, который либо непосредственно воспринимается машиной (в случае

программирования на машинном языке), либо доступен для ЭВМ благодаря наличию транслятора — специальной служебной программы, переводящей сообщения с этого языка на машинный язык данной ЭВМ. В последнем случае программист, изучив алгоритмический язык и используя его для описания алгоритмов решения своих задач, может даже не знать, какая именно вычислительная машина будет выполнять эти алгоритмы. В первую очередь он заинтересован в том, чтобы этот язык был легок для изучения и понимания и в то же время содержал удобные средства описания процессов решения интересующих программиста задач.

Ко второй категории относятся специалисты по математическому обеспечению ЭВМ, которых часто называют системными программистами. Одной из важных задач для них является разработка и описание удобных для пользователей языков программирования, а также методов перевода информации с этих языков на машинные языки конкретных ЭВМ. Естественно, что такая деятельность требует детального знакомства со способами описания, структурой и особенностями алгоритмических языков.

Данная книга рассчитана именно на специалистов этой второй категории и содержит сведения, необходимые для научно-исследовательской и практической работы в области развития и использования языков программирования.

Являясь новой областью математики, теория формальных языков, к которым относятся и языки программирования, еще не вполне оформилась как математическая дисциплина. В противоположность многим разделам классической математики методы изучения формальных языков разработаны недостаточно, а используемые в этих методах классы объектов не всегда удовлетворительно отражают свойства языков. Созда-

ние строгой и законченной теории формальных языков, несомненно, является одной из важнейших задач прикладной математики. В настоящей книге освещаются наиболее существенные синтаксические особенности алгоритмических языков и используемые в настоящее время методы их описания и изучения. В ней не ставится задача подробного описания каких-либо конкретных языков. Наоборот, предполагается хорошее знание читателями языка АЛГОЛ-60. Примеры из других языков снабжаются подробными пояснениями. Для лучшей ориентации в обсуждаемых вопросах желательно знакомство с машинными языками и общей структурой вычислительных машин, а также наличие у читателей элементарных сведений из математической логики.

В формировании взглядов и интересов автора в области теории формальных языков значительную роль сыграли работы А. В. Гладкого, в особенности монографии [15, 16]. Влияние этих работ можно проследить при чтении первой и отчасти второй глав книги. Автор выражает глубокую благодарность С. С. Лаврову, осуществившему титульное редактирование книги и своими советами и замечаниями оказавшему неоценимую помощь в подготовке ее к изданию, а также Х. И. Тойриху (ГДР), некоторые результаты совместной работы с которым отражены в четвертой главе, и С. Я. Фитиалову, чьи замечания при обсуждении ряда изложенных в книге вопросов оказались весьма ценными.

И. Л. Братчиков

ВВЕДЕНИЕ

§ 1. Язык как объект изучения средствами математики

В этом параграфе мы рассмотрим наиболее общие свойства языков как средства передачи и восприятия информации. Прежде всего отметим, что язык можно считать существующим только в том случае, если имеется некий пользователь этого языка, для которого данный язык понятен и который умеет его применять для обмена информацией. В случае естественных языков таких, например, как русский или английский, существуют группы людей, думающих и говорящих на этих языках, использующих их для обучения, для передачи сообщений и указаний и т. д. Иначе говоря, естественный язык служит средством общения между людьми. Хотя естественные языки являются универсальными в том смысле, что с их помощью может быть представлена любая полезная информация, они не очень удобны при необходимости выражения специальной, в частности, математической информации. Естественные языки в этом случае оказываются недостаточно строгими и формальными, они допускают нежелательные неоднозначности и разночтения. По этим причинам уже сравнительно давно были разработаны специальные искусственные языки, предназначенные для представления информации определенного, как правило, сравнительно узкого класса, свойства которых наилучшим образом соответствуют особенностям этой информации. К числу таких языков могут быть отнесены языки математических и химических формул, язык математической логики и многие другие. Оста-

ваясь средством общения между людьми, эти языки используются вместе с естественными и играют по отношению к последним подчиненную роль. Качественно новым этапом в разработке искусственных языков следует признать появление языков программирования. Принципиальной особенностью этих языков является то, что они предназначены для общения не только человека с человеком, но и человека с машиной. В качестве «пользователя» языка в данном случае наряду с человеком выступает устройство, резко отличающееся по своим возможностям передачи и восприятия информации.

При изучении любого языка необходимо знакомство с двумя его взаимосвязанными сторонами — во-первых, нужно уяснить, какую информацию можно передавать средствами языка, и во-вторых, нужно понять, каким образом при помощи данного языка передается та или иная информация. В лингвистике эти стороны языка называются, соответственно, *планом содержания* и *планом выражения*. Итак, *план содержания* — это передача средствами языка различных сведений, указаний, желаний и пр., а *план выражения* — это последовательности физических сигналов (звуков, символов), представляющие определенную информацию.

Переходя на математическую терминологию, мы можем теперь в самых общих чертах определить язык как объект, включающий в себя следующие компоненты [18]:

1) множество информации или «смыслов», характеризующее план содержания данного языка;

2) множество «текстов», т. е. последовательностей физических сигналов, которые характеризуют план выражения данного языка;

3) отображение, определенное на множестве текстов и ставящее в соответствие каждому элементу этого множества некоторые элементы множества смыслов, а также обратное отображение.

Рассмотрим указанные компоненты языка несколько подробнее. Множество смыслов для естественных языков является в значительной мере расплывчатым, плохо поддающимся формальному определению понятием. Получая осмысленное сообщение, человек реаги-

рует на него определенным образом, иначе говоря, смысловая информация, содержащаяся в этом сообщении, влияет на его последующее поведение. Множеством смыслов можно считать совокупность таких информации, воспринимаемых или передаваемых человеком.

Иначе обстоит дело в том случае, если язык предназначен для общения человека с машиной. Рассмотрим машинный язык, т. е. язык, непосредственно воспринимаемый некоторой машиной. Средства этого языка должны позволять передавать машине сообщения, вызывающие вполне определенные действия этой машины. План содержания языка должен соответствовать возможностям той машины, для общения с которой он предназначен, в том смысле, что на этом языке должны описываться любые действия, выполняемые машиной. Таким образом, множеством смыслов такого языка можно считать множество возможных операций соответствующей вычислительной машины. Множество смыслов алгоритмического языка типа АЛГОЛ-60, не ориентированного на какую-либо конкретную вычислительную машину, можно рассматривать как множество возможных действий некоторой *абстрактной* вычислительной машины. Для того, чтобы действия абстрактной машины, информация о которых передается предложениями алгоритмического языка, могли быть выполнены на одной из существующих конкретных вычислительных машин, последняя должна моделировать действия первой.

В противоположность множеству смыслов, которое по крайней мере в случае естественных языков плохо поддается формальному определению и изучению, множество текстов — это достаточно четкое понятие. Элементы этого множества — тексты или предложения языка — подчиняются как в случае естественных, так и особенно в случае искусственных алгоритмических языков строгим правилам построения. Эти правила позволяют, в частности, правильно построить желаемый текст или проверить, входит ли некоторый заданный текст в множество текстов данного языка.

Рассмотрим, наконец, соответствие между множествами смыслов и текстов. Отметим прежде всего, что полезное использование языка возможно только в том

случае, если функции, реализующие отображение множества текстов в множество смыслов и обратное отображение, эффективно вычислимы. Иначе говоря, должен существовать способ быстрого «вычисления» по любому тексту соответствующих ему смыслов и по любому смыслу соответствующих текстов. Для естественных языков это означает, что человек, пользующийся языком, может понять сообщаемую ему на этом языке информацию и может сформулировать на нем собственные мысли. К сожалению, в случае естественных языков изучение свойств функций соответствия текстов и смыслов точными математическими методами представляется чрезвычайно сложным делом. Прежде всего, мы ничего не знаем о природе этих функций за исключением того, что они должны быть вычислимыми. Пользуясь языком, человек «вычисляет» эти функции подсознательно и ничего не может сказать об алгоритме, которым он при этом пользуется. Более того, каждый пользующийся данным языком человек имеет свои собственные множества смыслов и текстов, а следовательно, и свои собственные функции отображения. Эти индивидуальные подязыки зависят от многих причин, в частности, от степени знакомства с языком, от полученного данным индивидуумом общего образования, от его специальности и пр. Значительные затруднения при изучении формальными методами вызывают такие свойства естественных языков, как существенная неоднозначность, свойственная им эмоциональная окраска, наличие идиом и фразеологических оборотов, изменчивость и многие другие.

Большинства указанных трудностей не существует в случае языков программирования. В противоположность естественным языкам, развивающимся в значительной степени стихийно, языки программирования разрабатываются сознательно и для строго определенных целей. Создатели языка программирования могут придать ему те свойства, которые кажутся им желательными как для облегчения изучения его человеком, так и для лучшего восприятия машиной сообщений на этом языке.

Итак, мы обсудили две стороны языка как средства передачи информации — его физическую, текстовую сторону и его смысловую сторону. Правила, опи-

сывающие входящие в язык тексты как цепочки некоторых символов, называются *синтаксисом*, а правила, определяющие смысловую сторону входящих в язык цепочек, т. е. приписывающие им определенные смысловые значения,— *семантикой* данного языка. В настоящей книге рассматриваются в основном математические методы изучения синтаксиса языков программирования.

Важнейшей задачей является также изучение перевода с одного языка на другой как точного алгоритмического процесса. В определенных нами выше понятиях задачей перевода с языка A на язык B является отображение множества текстов языка A в множество текстов языка B , оставляющее инвариантным смысловой образ любого текста. Отметим, что перевод с одного языка на другой не всегда возможен. Необходимым условием возможности перевода произвольного текста с языка A на язык B является совпадение их смысловых множеств, или, в крайнем случае, теоретико-множественное включение смыслового множества языка A в смысловое множество языка B . Так, бессмысленно говорить о переводе произвольных предложений русского языка на, скажем, язык нот, или программ языка АЛГОЛ-60 на машинный язык специализированной вычислительной машины.

Перевод может быть семантическим или формальным. Для того, чтобы лучше представить себе эти два способа, введем следующие обозначения:

T_A — множество текстов языка A ,

T_B — множество текстов языка B ,

S — множество смыслов языков A и B (для простоты считаем, что смысловые множества языков совпадают),

f_A — функция, отображающая T_A в S ,

\hat{f}_B — функция, отображающая S в T_B .

Функции f_A и \hat{f}_B будем считать однозначными.

Смысловой перевод, которым, по-видимому, пользуется человек, заключается в следующих действиях. Пусть имеется некоторый текст $t \in T_A$. Тогда:

а) Находится значение $f_A(t)$. Пусть $f_A(t) = s \in S$.

б) Находится значение $\hat{f}_B(s)$, которое мы обозначим через t'_B .

Итак, переводом текста $t \in T_A$ является текст $t' \in T_B$. Использование такого способа значительно осложняется в случае применения для перевода вычислительной машины. Как уже отмечалось, множество смыслов, а также функции отображения трудно описать формально, в особенности для естественных языков. Поэтому более подходящим для автоматизации представляется другой, формальный способ перевода. Этот способ заключается в явном задании алгоритма вычисления функции, являющейся суперпозицией функций f_A и \hat{f}_B . Обозначим эту суперпозицию через I_{AB} . Очевидно, она определяется на множестве T_A и принимает значения из T_B . Итак, имеем

$$I_{AB}(t) = \hat{f}_B(f_A(t)).$$

Задание функции I_{AB} позволяет исключить при переводе плохо формализуемое множество смыслов S . С другой стороны, поскольку I_{AB} определена и принимает значения на множествах текстов, легче поддающихся формальному изучению, можно предположить, что и алгоритм вычисления этой функции описывается проще. Конечно, при разработке этого алгоритма необходимо учитывать семантику соответствующих языков.

Методы формального перевода в настоящее время успешно развиваются как для естественных, так и для алгоритмических языков. Алгоритмы перевода с языков программирования на другие языки программирования (обычно машинные языки) называются трансляторами. В данной книге мы рассмотрим эти алгоритмы лишь в самых общих чертах.

§ 2. Общая классификация алгоритмических языков

Языки программирования, используемые в настоящее время для решения задач на ЭВМ, значительно отличаются друг от друга своей структурой и средствами описания алгоритмов. Различны также и методы отладки и выполнения программ, написанных на этих языках. Многообразие языков программирования достаточно хорошо иллюстрируется сравнением какого-либо машинного языка с таким алгоритмическим языком, как АЛГОЛ-60.

Каковы же основные принципы проектирования и разработки новых языков программирования? Каким требованиям должен удовлетворять язык, рассчитанный на широкое использование при решении задач на ЭВМ? Среди многих критериев качества языка наиболее существенными представляются следующие. В первую очередь, язык должен быть удобен для программиста. В частности, он должен быть легким для изучения и должен иметь средства, позволяющие с минимальными затратами времени программиста подготовить задачи к решению на вычислительных машинах. С другой стороны, язык не может разрабатываться только на основании этого требования. При разработке языка неизбежно приходится учитывать специфику ЭВМ, другими словами, язык должен быть таким, чтобы указанные в программах действия могли быть однозначно «поняты» и выполнены вычислительной машиной. Желательно при этом, чтобы машина не тратила слишком большое время на расшифровку этих действий и могла бы их выполнить достаточно эффективным способом. Итак, язык должен быть удобным как для программиста, так и для машины. К сожалению, эти требования являются в известной степени трудно совместимыми. То, что «удобно» для ЭВМ, оказывается не совсем удобным для программиста и наоборот. Между тем, при программировании конкретных задач основным может оказаться как требование «удобства» для ЭВМ, т. е. максимальное сокращение машинного времени, необходимой для решения памяти и пр., так и требование удобства для программиста — сокращение времени, необходимого для программирования, возможность описания задачи в привычных обозначениях и пр. Рассмотрим два наиболее типичных случая. Пусть мы должны составить некоторую стандартную программу, например, программу вычисления синуса или решения систем линейных алгебраических уравнений, т. е. такую программу, которая будет выполняться данной машиной много раз. Естественно, в этом случае мы можем пойти на дополнительные затраты времени программирования с тем, чтобы программа получилась оптимальной с точки зрения времени выполнения и других критериев, связанных с вычислительной машиной. Для програм-

мирования в этом случае следует выбрать машинный язык данной ЭВМ или язык, близкий к машинному, так как именно в этом языке наилучшим образом учитываются особенности той машины, для которой пишется программа.

Рассмотрим теперь ситуацию, противоположную описанной, т. е. случай, когда нам требуется составить программу, рассчитанную на однократное использование. Здесь при выборе языка программирования в первую очередь нужно учитывать его удобство для описания метода решения задачи. Выбрав подходящий с этой точки зрения язык из числа используемых на ЭВМ, предназначенной для решения, мы значительно сократим время программирования задачи. Правда, ЭВМ будет выполнять нашу программу дольше, чем в случае, если бы она была написана на машинном языке, но в данном случае это не имеет большого значения. Можно, разумеется, привести примеры, когда приходится учитывать оба критерия — желательно затратить не очень большое время на программирование, но в то же время получить достаточно экономную для реализации на ЭВМ программу.

Указанные требования, а также широта области применения универсальных ЭВМ, по сути дела, и определяют большое многообразие алгоритмических языков.

Перейдем теперь к общей классификации алгоритмических языков. Языки могут классифицироваться по различным признакам. Так, в литературе часто встречается классификация языков на основе классов задач, для описания методов решения которых предназначен тот или иной язык. Можно, например, выделить языки для задач вычислительного характера (ФОРТРАН, АЛГОЛ-60), для экономических задач (КОБОЛ), для обработки символьной информации (ЛИСП, СНОБОЛ), универсальные языки (АЛГОЛ-68, PL/1) и другие. Мы примем за основу нашей классификации другой признак — степень близости языков к тем, которые непосредственно воспринимаются вычислительными машинами. Такая классификация в определенной мере отражает также и историю развития языков программирования и методики их описания и использования.

Будем различать языки трёх уровней. К первому отнесем *машинные языки*, или, как их иногда называют, *абсолютные языки* программирования. Это, как мы уже знаем, такие языки, которые не требуют никаких дополнительных преобразований программ перед их выполнением вычислительной машиной. На машинном языке универсальной ЭВМ любой алгоритм может быть представлен в виде последовательности команд и дополнительной информации, включающей в себя исходные данные, константы и пр. Выполняя эти команды в заданном порядке, машина получает необходимые результаты. Команда и дополнительная информация представляют собой последовательности определенной длины, состоящие из нулей и единиц.

Выполняя команду, ЭВМ производит следующие два действия:

- 1) выполняет определенную командой операцию над данными,
- 2) выбирает следующую команду либо безусловно, либо способом, зависящим от данных.

В соответствии с этими действиями можно считать, что любая команда состоит из двух частей — операционной части и управляющей части. В действительности одна из этих частей во многих командах отсутствует. Управляющая часть нужна лишь в тех командах, которые нарушают естественный порядок выполнения команд (т. е. выполнение команд в порядке их расположения в памяти), — в командах перехода. В свою очередь, в командах перехода часто отсутствует операционная часть.

Рассмотрим теперь множества, элементы которых представляют преобразуемую вычислительной машиной информацию и могут использоваться в качестве компонент команд машинного языка.

- 1) Множество допустимых кодов. Как уже говорилось, любая преобразуемая ЭВМ информация представляется в виде двоичных кодов определенной длины. Допустимые коды — это такие двоичные коды, которые могут быть однозначно представлены в памяти данной машины и использованы в качестве операндов выполняемых машиной операций. Во многих ЭВМ длина допустимых кодов определяется разрядностью ячеек памяти. Так, для машины М 20

допустимым можно считать любой 45-разрядный двоичный код. В некоторых ЭВМ допустимые коды могут иметь различную длину. Например, в ЕС ЭВМ допустимые коды могут иметь длину от одного до 256 восьмиразрядных байтов. В зависимости от выполняемой операции допустимые коды могут рассматриваться как коды десятичных или двоичных чисел с фиксированной или плавающей запятой, коды элементов алфавитно-числовой информации и т. д. Поскольку множество допустимых кодов конечно для любой вычислительной машины, не все числа могут быть представлены в памяти ЭВМ точно. Длина допустимых кодов, используемых в качестве операндов арифметических операций, определяет абсолютную или относительную точность, с которой работает данная вычислительная машина.

2) Множество допустимых операций. В него входят те операции, которые может производить данная ЭВМ, выполняя ту или иную команду. Каждой операции поставлен в соответствие некоторый двоичный код определенной длины. Длина кодов допустимых операций различна для разных машин, но обычно не превышает девяти двоичных разрядов.

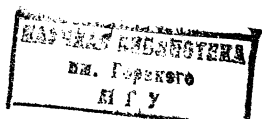
3) Множество допустимых адресов. Оно состоит из адресов ячеек памяти вычислительной машины. Каждый адрес представляет собой двоичный код некоторой постоянной длины. Количество допустимых адресов и длина соответствующих кодов зависят от емкости оперативной памяти данной машины, т. е. от числа ячеек, составляющих эту память.

Можно выделить также множество допустимых команд данного машинного языка. Каждая допустимая команда представляет собой двоичный код, составленный из кода некоторой допустимой операции и одного или нескольких допустимых адресов. Нетрудно видеть, что последовательность допустимых команд определяет работу, которую должна выполнить вычислительная машина, решая некоторую задачу, а хранящиеся в памяти допустимые коды — информацию, подвергающуюся переработке в ходе решения задачи. Существенной особенностью машинных языков является то, что один и тот же код может на разных этапах решения задачи рассматриваться машиной как допустимый код.

представляющий перерабатываемую информацию, и как допустимая команда, определяющая одну из выполняемых машиной операций.

Машинные языки обладают необходимой точностью и достаточно формализованы для адекватного представления алгоритмов решения задач. В то же время машинные языки неудобны для использования человеком. Одно из очевидных неудобств машинных языков заключается в том, что программисту приходится пользоваться двоичной системой счисления. Это, во-первых, требует перевода чисел в двоичную систему и представления их в виде допустимых кодов, а во-вторых, предполагает утомительное выписывание большого числа нулей и единиц. Впрочем, эти трудности не являются столь существенными и могут быть легко обойдены. Вместо того, чтобы вручную переводить числа, составляющие исходные данные, в двоичную систему счисления, можно написать специальную программу, которая автоматически переводила бы десятичные числа, представленные по определенным правилам в виде допустимых кодов. Далее, для представления двоичных кодов команд и нечисловой информации можно воспользоваться восьмеричной или шестнадцатеричной системами счисления. Поскольку 8 и 16 являются целыми степенями двух, возможен так называемый локальный перевод из этих систем счисления в двоичную, при котором каждая цифра переводится в двоичную систему отдельно, независимо от окружающих ее цифр. Фактически этот перевод осуществляется устройствами подготовки информации для ввода в память вычислительной машины — ленточными или карточными перфораторами.

К сожалению, машинные языки обладают и рядом значительно более существенных недостатков. Эти недостатки приводят к тому, что, подготавливая задачу для решения на ЭВМ, программист вынужден тратить большое время на выполнение ряда чисто технических действий. Для того, чтобы подробнее рассмотреть эти недостатки и наметить пути их устранения, полезно представить себе процесс подготовки задачи к решению на вычислительной машине. Подготовительная работа математика-программиста может быть разбита на ряд этапов;



1) Математическая постановка задачи. Этот предварительный, но чрезвычайно важный этап работы не требует подробных пояснений. Отметим только, что для квалифицированного выполнения этого этапа необходим достаточно высокий уровень математической подготовки, а также знакомство с прикладной стороной или, иначе говоря, со смыслом рассматриваемой задачи.

2) После того как найдена точная математическая формулировка задачи, необходимо выбрать наилучший метод ее решения. Этот этап также требует высокой квалификации, в частности хорошего знакомства с методами решения задач соответствующего класса. При выборе метода решения немаловажное значение имеет и учет особенностей той ЭВМ, на которой будет решаться задача.

3) Когда метод решения выбран и, следовательно, известен алгоритм решения задачи на вычислительной машине, нужно представить этот алгоритм в виде программы, написанной на машинном языке той ЭВМ, которая будет решать задачу. Составлением программы не обязательно должен заниматься математик, выбравший метод решения задачи. Если он сумеет достаточно хорошо описать алгоритм своему коллеге, то может передать задачу последнему. При этом программист может быть незнаком с выбранным численным методом, может не знать теоретических основ метода, но должен хорошо представлять себе последовательность операций вычислительной машины, к которой сводится выполнение соответствующего алгоритма.

Приступая к написанию программы, программист должен был бы прежде всего распределить память вычислительной машины, т.е. отвести определенные ячейки под программу, исходные данные, промежуточные и окончательные результаты вычислений. Однако распределить память до того, как написана программа, довольно затруднительно. Дело в том, что программист, как правило, не может более или менее точно оценить число команд программы, которую он собирается писать. Часто трудно также судить о количестве ячеек, необходимых для хранения результатов вычислений. Это обстоятельство приводит к тому, что программист обычно сначала пишет программу, а потом

распределяет память для команд программы и результатов вычислений. При этом программа составляется в так называемых условных адресах, т.е. вместо действительных адресов ячеек памяти используются условные обозначения этих адресов. Память, необходимая для решения задачи, представляется на этой стадии программирования в виде некоторого количества полей, т.е. связанных последовательностей ячеек. Каждое поле обозначается некоторой буквой (или идентификатором). Условным адресом первой ячейки поля обычно считается идентификатор, обозначающий это поле. Остальные ячейки поля получают адреса, представленные в виде пары значений — идентификатора поля и порядкового номера данной ячейки в поле, уменьшенного на единицу.

При программировании в условных адресах программист пишет программу, не связывая себя с конкретными адресами ячеек памяти. Когда программа написана, он определяет длину каждого употребленного поля памяти и вычисляет общее количество ячеек, требуемых для программы, данных и результатов задачи.

4) Если программа написана в условных адресах, то следующим этапом работы является распределение памяти и замена условных адресов на действительные в командах программы. При распределении памяти программист должен поставить в соответствие каждому полю памяти соответствующее количество конкретных ячеек. Иначе говоря, нужно идентификатору поля поставить в соответствие некоторый численный адрес ячейки памяти. Ячейка с этим адресом будет первой ячейкой, отведенной для данного поля. Остальные ячейки поля будут расположены непосредственно за первой, а их адреса вычисляются как суммы адреса первой ячейки данного поля и порядковых номеров соответствующих ячеек поля, уменьшенных на единицу. Выполнение этого этапа представляет собой чисто техническую работу, для которой не требуется, по существу, никакой специальной квалификации. С другой стороны, именно этот этап является источником механических ошибок, допускаемых при выписывании действительных адресов.

5) Следующим и последним этапом работы по подготовке задачи к решению на ЭВМ является отладка

программы, т. е. устранение при помощи вычислительной машины ошибок, допущенных при программировании. Все возможные ошибки можно разделить на три существенно различные группы. К первой отнесем так называемые синтаксические ошибки. В случае использования машинных языков к синтаксическим ошибкам относятся такие ошибки, как неправильное написание отдельных команд (несуществующий код операции, адрес ячейки памяти), неверно написанная константа (например, ненормализованная мантисса числа с плавающей запятой) и т. д. Важно отметить, что такие ошибки могут быть сравнительно легко обнаружены при внимательном просмотре программы без ее выполнения на ЭВМ. Ко второй группе относятся семантические ошибки, которые в большинстве случаев могут быть обнаружены лишь при выполнении программы вычислительной машиной. При этом семантические ошибки приводят к необычным и непредвиденным ситуациям при выполнении программы. К таким ситуациям относится, например, переполнение, которое возникает при делении на ноль или при умножении (сложении) двух больших чисел. Семантические ошибки часто приводят к так называемому заикливанию, т. е. к выполнению бесконечное число раз определенной группы команд. В случае семантических ошибок каждая отдельная команда записана правильно, однако неправильно определена последовательность, с которой выполняются команды, или же неправильно определены адреса операндов, участвующих в операциях, выполняемых машиной. Наконец, к третьей группе можно отнести такие ошибки, которые связаны с алгоритмом решения задачи. Мы назовем эти ошибки ошибками метода. Причина их возникновения заключается либо в неправильном выборе численного метода решения задачи, либо в неправильном программировании, при котором программа, оставаясь формально правильной, описывает не тот алгоритм, который был выбран для решения задачи. Существенной особенностью ошибок третьей группы является то, что эти ошибки невозможно найти при выполнении программы и, как правило, очень трудно — при ее просмотре. Лучшими методами их устранения являются подбор для отладки таких исходных данных, для которых

известны или могут быть легко вычислены вручную результаты решения задачи, а также сравнение результатов, полученных при решении задачи на ЭВМ по двум независимо составленным программам, представляющим, возможно, различные методы ее решения.

Отладка программ, написанных на машинном языке, является трудоемким и утомительным делом. Она часто приводит к необходимости многократного выполнения программы вычислительной машиной и даже к использованию так называемого однократного режима, при котором программист выполняет на ЭВМ отдельные команды и анализирует выводимые на панель сигнализации ЭВМ результаты выполнения этих команд. Иногда при отладке используются специальные служебные программы, печатающие результаты выполнения всех или некоторых команд отлаживаемой программы. В любом случае такая отладка связана с большим расходом как времени программиста, так и машинного времени.

Выделенные нами этапы подготовки задач к решению на ЭВМ и их характеристика показывают, что подготовительная работа на некоторых своих этапах требует участия высококвалифицированных специалистов, а на других является простой, но трудоемкой и требующей большой аккуратности обработки информации, используемой при подготовке задачи. Естественным способом сокращения времени подготовки задачи является поэтому частичная или полная автоматизация последних трех этапов подготовительной работы, т. е. частичное или полное их выполнение самой вычислительной машиной. Этот процесс, который часто называют автоматизацией программирования, связан с созданием специальных служебных программ и преследует следующие основные цели:

— сокращение времени составления программ за счет использования более удобных средств описания алгоритмов, нежели машинные языки (при этом ЭВМ сама должна переводить эти описания на машинный язык), что означает частичную автоматизацию третьего этапа подготовительной работы;

— передача вычислительной машине работы по распределению памяти и представлению программ в

действительных адресах, что означает автоматизацию четвертого этапа;

— обнаружение самой вычислительной машиной синтаксических и, по возможности, семантических ошибок и выдача о них удобной для программиста информации, что означает частичную автоматизацию пятого этапа, при которой программист при отладке будет исправлять замеченные ЭВМ ошибки, а также выявлять возможные ошибки метода.

Для достижения этих трех основных целей автоматизации программирования и были разработаны различные немашинные языки программирования, основные черты которых будут рассмотрены ниже. Эти языки, которые часто называют языками высших уровней, могут быть разбиты на две группы. К первой относятся такие языки, которые привязаны к конкретным вычислительным машинам в том смысле, что сохраняют основную структуру соответствующих машинных языков. Это так называемые *машинно-ориентированные языки* или *автокоды*, составляющие в нашей классификации языки второго уровня. Ко второй группе немашинных языков относятся такие языки, которые не привязаны к конкретным вычислительным машинам, хотя некоторые особенности ЭВМ могут определяться в этих языках заданием специальных параметров. Такие языки называются *проблемно-ориентированными языками* и являются в нашей классификации языками третьего уровня.

Рассмотрим сначала машинно-ориентированные языки. К этим языкам относятся, например, язык ассемблера ЕС ЭВМ, автокод машины М-222, язык «ЯЗ» ЭВМ «Одра-1204» и другие подобные языки. В машинно-ориентированных языках существенно расширяются множества допустимых объектов, рассмотренные нами в случае машинных языков:

1) Множество допустимых кодов расширяется за счет включения числовых кодов в восьмеричной, десятичной или шестнадцатеричной системах счисления, а также за счет включения символьных кодов (последовательностей символов некоторого фиксированного множества).

2) Множество допустимых адресов теперь включает в себя буквенно-цифровые имена типа условных

адресов, а в самом общем случае — более сложные выражения, состоящие из идентификаторов полей памяти и чисел.

3) Множество допустимых операций для машинно-ориентированных языков состоит из мнемонических кодов выполняемых машиной операций, из кодов новых, включенных в язык макроопераций, реализуемых вычислительной машиной при выполнении определенной последовательности машинных команд, а также из кодов так называемых операций для трансляции, которые указывают машине, как производить перевод (трансляцию) с автокода на машинный язык.

Составление программы на машинно-ориентированном языке несколько облегчается по сравнению с машинным языком за счет того, что, во-первых, используются мнемонические коды операций и мнемонические адреса, которые легче запомнить по сравнению с двоичными кодами, во-вторых, значительно богаче множество допустимых кодов, что позволяет программисту не тратить времени на кодирование различной обрабатываемой информации, в-третьих, наличие макроопераций позволяет сократить длину составляемой программы и, наконец, имеются удобные способы описания рабочих полей памяти и адресов ячеек этих полей. При использовании машинно-ориентированных языков распределение памяти и замена в программе условных адресов на действительные выполняются полностью автоматически во время перевода программы на машинный язык. Предусматривается также выдача информации о синтаксических и некоторых семантических ошибках.

Таким образом, появление машинно-ориентированных языков представляло собой существенный шаг в деле автоматизации программирования. В то же время машинно-ориентированные языки сохраняют наиболее существенный недостаток машинных языков — они слишком не похожи на такие традиционные математические языки, как язык алгебраических формул или язык математической логики, а предложения (команды) этих языков определяют слишком мелкие операции вычислительной машины. В связи с этим время, затрачиваемое на составление программ, остается большим, а сами программы — слишком длинными.

Нужно отметить, что машинные языки в чистом виде практически не используются при программировании для современных вычислительных машин. Обычно на машинных языках пишутся лишь программы, при помощи которых производится трансляция с автокодов на машинные языки. Во всех остальных случаях вместо машинных языков используются соответствующие машинно-ориентированные языки. Программирование на автокодах позволяет учесть все особенности конкретных вычислительных машин и составить программы столь же эффективные, как и при использовании машинных языков. В основном эти языки применяются при написании служебных программ, входящих в математическое обеспечение ЭВМ, стандартных программ, рассчитанных на многократное использование, иначе говоря, в тех случаях, когда главным требованием является машинная эффективность разрабатываемой программы.

Наиболее сложными по своей структуре и организации и интересными с точки зрения изучения и использования являются языки третьего уровня — проблемно-ориентированные языки. Определяемые в этих языках множества объектов коренным образом отличаются от объектов языков первых двух уровней. Эти объекты значительно отличаются друг от друга и в различных проблемно-ориентированных языках. В самом общем виде они могут быть разбиты на следующие четыре группы:

- 1) множество значений;
- 2) арифметические операторы, или средства языков, определяющие вычислительную работу ЭВМ;
- 3) управляющие операторы, или средства, определяющие порядок действий при решении задачи, а также режим ввода-вывода;
- 4) описания, т. е. средства описания данных, перерабатываемых при решении задачи.

Создание языков третьего уровня позволило предоставить в распоряжение программистов средства значительно более гибкой и разнообразной структуры, чем машинные языки и автокоды. Существенно и то, что применение этих языков приближает описание алгоритмов для ЭВМ к общепринятой форме записи методов решения задач соответствующих классов.

Следует подчеркнуть, что и сами языки, в свою очередь, заметно влияют на традиционные способы представления этих методов, которые становятся более точными, требующими меньшего количества пояснений и комментариев. Многие методы решения задач описываются в настоящее время непосредственно в виде процедур языка АЛГОЛ-60, и это существенно облегчает их понимание и практическую реализацию.

Языки третьего уровня обладают следующими свойствами, обусловившими их широкое использование для решения самых различных задач:

- позволяют удобно и экономно записывать формулы, определяющие вычислительную часть решения задач различных классов;

- позволяют определять и задавать структуру и форматы данных самых различных видов;

- позволяют статически, т. е. во время трансляции, и динамически, т. е. во время выполнения переведенной программы, распределять память для обрабатываемой информации;

- достаточно точны и формализованы для автоматического перевода программ на машинный язык;

- позволяют легко найти и устранить ошибки программирования;

- удобны и просты для изучения.

Дальнейшая часть книги посвящена изучению свойств именно этих, наиболее сложных и интересных алгоритмических языков.

В заключение заметим, что по мнению некоторых специалистов одним из направлений развития языков программирования в ближайшем будущем будет разработка так называемых языков четвертого уровня, включающих в себя средства для автоматического выбора метода решения задачи с помощью ЭВМ. Такие языки должны позволять описывать формулировку задачи и некоторые требования к методу решения (точность результата, примерное машинное время решения и пр.). Дальнейшая работа по подготовке задачи и ее решению на ЭВМ должна выполняться автоматически. Основная трудность на пути создания языков четвертого уровня заключается не в разработке собственно языков, а в создании программ автоматического выбора методов решения задач.

ГЛАВА I

ПОРОЖДАЮЩИЕ ГРАММАТИКИ

Рассмотрев во введении самые общие свойства и особенности различных языков программирования, мы перейдем теперь к изучению способов математического описания и исследования одной из двух характеристик языков — их синтаксиса.

Изучение математических средств описания синтаксиса естественно связано с двумя различными практическими задачами, возникающими при разработке и использовании различных языков. Первая задача заключается в составлении некоторого количества формальных правил, используя которые можно построить любую правильную фразу данного языка или, говоря точнее, любую *цепочку символов*, входящую в изучаемый язык, представленный как множество цепочек. Вторая задача в определенном смысле обратна первой. Она заключается в том, что требуется разработать правила, позволяющие определить, является ли произвольная цепочка символов правильной фразой языка. В случае утвердительного ответа на этот вопрос в большинстве случаев требуется указать отдельные конструкции, входящие в фразу, или, как обычно говорят, определить ее *синтаксическую структуру*. В случае отрицательного ответа часто полезна информация, указывающая на допущенные при написании фразы ошибки.

Хотя эти задачи взаимосвязаны, для их решения используются различные математические средства. Для задания правил порождения правильных фраз языка (первая задача) в большинстве случаев используется аппарат *порождающих грамматик*, в то время

как для задания способов распознавания произвольных цепочек символов (вторая задача), как правило, применяется аппарат распознающих автоматов. Как мы увидим ниже, порождающие грамматики являются частным классом исчислений математической логики, а распознающие автоматы в большинстве случаев можно рассматривать как машины Тьюринга с различными ограничениями.

В настоящей главе мы рассмотрим определение и те свойства порождающих грамматик, которые наиболее важны при их использовании для описания синтаксиса языков программирования. При формулировке и доказательстве ряда утверждений, проводимых в главе, использован материал монографий [15] и [16].

§ 1. Нормальные формы Бекуса

При описании структуры и правил использования некоторого языка мы всегда употребляем определенную систему обозначений, понятий и образованных из них конструкций, позволяющую представить описываемый язык при помощи известных нам объектов и отношений между ними. Эта система, подчиняясь более или менее строгим правилам образования описывающих язык конструкций, сама может считаться языком, либо уже известным, либо специально разработанным для описания другого языка. Такой язык, описывающий свойства другого языка, часто называют *метаязыком*. При описании естественных языков обычно используют те же или другие естественные языки. Так, метаязыком, описывающим русский язык, может быть тот же самый русский язык или другой разговорный язык, например английский. Такой способ описания языков вполне оправдан в случае, когда, во-первых, описание предназначено для человека, а, во-вторых, описываемый язык является таким богатым, гибким, многозначным, каким мы привыкли считать любой естественный язык. Дело, однако, существенно меняется в случае, если требуется описание разработанных для определенных целей строгих, формальных и однозначных алгоритмических языков. Описание языков программирования обычными сред

ствами оправдано только в том случае, если оно используется лишь для изучения этих языков программистами-пользователями. Если же описание предназначено для лиц, разрабатывающих трансляторы, или оно вводится в память ЭВМ для автоматического составления отдельных блоков трансляторов, то требуются специальные метаязыки, которые должны быть столь же строгими и формальными, как и описываемые с их помощью языки программирования, а также иметь, по возможности, простую структуру и внутреннюю организацию. В настоящее время используется ряд языков для описания синтаксиса алгоритмических языков (а также синтаксиса естественных языков) для дальнейшей обработки и изучения математическими средствами. Это так называемые *метасинтаксические языки*. Разрабатываются также специальные языки для описания семантики — *метасемантические языки*.

Наиболее распространенным метасинтаксическим языком являются *нормальные формы Бекуса* (или *металингвистические формулы*) — язык, специально разработанный для описания синтаксиса языка АЛГОЛ-60 и используемый для описания многих других языков программирования. Основное назначение форм Бекуса заключается в представлении в сжатом и компактном виде строго формальных и однозначных правил написания основных конструкций описываемого языка программирования.

Любое предложение языка можно рассматривать как цепочку (т. е. линейно упорядоченную последовательность) основных символов этого языка. Так, программа на языке АЛГОЛ-60 — это составленная по определенным правилам цепочка основных символов АЛГОЛа-60. В результате использования в определенном порядке форм Бекуса, описывающих синтаксис языка АЛГОЛ-60, мы можем построить любую правильную программу на этом языке ¹⁾. Из сказанного ясно, что одним из классов объектов, которые используются в формах Бекуса, должны быть основные символы описываемого языка.

¹⁾ Вообще говоря, правильная программа должна еще удовлетворять контекстным условиям. См. гл. IV,

Характерной особенностью языков программирования, так же как и естественных языков, является то, что в сложные синтаксические конструкции в качестве составных частей входят другие конструкции. Так, программа на АЛГОЛе-60 является обычно блоком, который, в свою очередь, может включать в себя один или несколько внутренних блоков. Блок также является сложной конструкцией, включающей в себя описания, операторы; последние тоже имеют составные части и т. д. Таким образом, для того чтобы написать программу, нужно знать правила написания блоков и других конструкций, входящих в программу в качестве составных частей. Вторым классом объектов, используемых в формах Бекуса, как раз и являются имена конструкций описываемого языка, или так называемые *металингвистические переменные*. Значения металингвистических переменных — это цепочки основных символов описываемого языка.

Каждая металингвистическая формула описывает правила построения некоторой конструкции языка и состоит из двух частей. В левой части находится металингвистическая переменная, обозначающая соответствующую конструкцию. Далее следует так называемая *металингвистическая связка* $::=$, имеющая смысл глагола «быть». Она соединяет левую и правую части формулы. В правой части формулы указывается один или несколько вариантов построения конструкции, определенной в левой части. Каждый вариант представляет собой цепочку, состоящую из металингвистических переменных и основных символов. Для того, чтобы построить определяемую формулой конструкцию, нужно выбрать некоторый вариант построения из правой части формулы и, используя соответствующие формулы, подставить вместо каждой металингвистической переменной некоторые цепочки основных символов. Варианты правой части формулы разделяются металингвистической связкой $|$, имеющей значение «или».

Наконец, отметим, что металингвистическая переменная обозначается словами, заключенными в угловые скобки $\langle \rangle$, которые поясняют смысл описываемой конструкции. Рассмотрим примеры форм Бекуса, взятых из описания синтаксиса языка АЛГОЛ-60 [1].

Числа определяются в языке при помощи следующих правил:

- 1) $\langle \text{число} \rangle ::= \langle \text{число без знака} \rangle | + \langle \text{число без знака} \rangle | - \langle \text{число без знака} \rangle$
- 2) $\langle \text{число без знака} \rangle ::= \langle \text{десятичное число} \rangle | \langle \text{порядок} \rangle | \langle \text{десятичное число} \rangle \langle \text{порядок} \rangle$
- 3) $\langle \text{десятичное число} \rangle ::= \langle \text{целое без знака} \rangle | \langle \text{правильная дробь} \rangle | \langle \text{целое без знака} \rangle \langle \text{правильная дробь} \rangle$
- 4) $\langle \text{порядок} \rangle ::= {}_{10} \langle \text{целое} \rangle$
- 5) $\langle \text{правильная дробь} \rangle ::= . \langle \text{целое без знака} \rangle$
- 6) $\langle \text{целое} \rangle ::= \langle \text{целое без знака} \rangle | + \langle \text{целое без знака} \rangle | - \langle \text{целое без знака} \rangle$
- 7) $\langle \text{целое без знака} \rangle ::= \langle \text{цифра} \rangle | \langle \text{целое без знака} \rangle \langle \text{цифра} \rangle$
- 8) $\langle \text{цифра} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

В приведенных формулах описываются 8 конструкций языка АЛГОЛ-60, причем для построения числа может понадобиться любая из семи остальных конструкций. Например, число $1.5_{10}-2$ выводится по формулам следующим образом:

$\langle \text{число} \rangle$, $\langle \text{число без знака} \rangle$, $\langle \text{десятичное число} \rangle$, $\langle \text{порядок} \rangle$, $\langle \text{целое без знака} \rangle$, $\langle \text{правильная дробь} \rangle$, $\langle \text{порядок} \rangle$, $\langle \text{цифра} \rangle$, $\langle \text{правильная дробь} \rangle$, $\langle \text{порядок} \rangle$, $\langle \text{правильная дробь} \rangle$, $\langle \text{порядок} \rangle$, $1. \langle \text{целое без знака} \rangle$, $\langle \text{порядок} \rangle$, $1. \langle \text{цифра} \rangle$, $\langle \text{порядок} \rangle$, $1.5 \langle \text{порядок} \rangle$, $1.5_{10} \langle \text{целое} \rangle$, $1.5_{10} - \langle \text{целое без знака} \rangle$, $1.5_{10} - \langle \text{цифра} \rangle$, $1.5_{10} - 2$.

Для вывода числа мы использовали правила в следующей последовательности (в скобках указывается порядковый номер варианта из правой части соответствующего правила): 1(1), 2(3), 3(3), 7(1), 8(2), 5, 7(1), 8(6), 4, 6(3), 7(1), 8(3).

Интересной особенностью металингвистических формул является наличие в них *рекурсий*, т. е. использование для описания некоторых конструкций самих описываемых конструкций. Рекурсия может быть явной и неявной. Явная рекурсия имеет место тогда, когда в правой части некоторой формулы используется металингвистическая переменная из левой части этой же формулы. Таким является, например, правило 7 в приведенном выше списке правил. В явно рекурсивном правиле обязательно должна содержаться

перекурсивная часть (в правиле 7 — это первый вариант из правой части), так как только в этом случае возможен вывод цепочки основных символов. Неявная рекурсия присутствует в случае, когда при выводе конструкции мы на некотором шаге используем металингвистическую переменную, обозначающую саму выводимую конструкцию. Такая ситуация может возникнуть, например, при выводе арифметического выражения, когда используются следующие правила (мы приводим их в сокращенном виде, указывая только те варианты из правых частей, которые нас интересуют):

- 1) $\langle \text{арифметическое выражение} \rangle ::= \langle \text{простое арифметическое выражение} \rangle$
- 2) $\langle \text{простое арифметическое выражение} \rangle ::= \langle \text{терм} \rangle$
- 3) $\langle \text{терм} \rangle ::= \langle \text{множитель} \rangle$
- 4) $\langle \text{множитель} \rangle ::= \langle \text{первичное выражение} \rangle$
- 5) $\langle \text{первичное выражение} \rangle ::= \langle \text{арифметическое выражение} \rangle$

Применяя для вывода арифметического выражения правила с 1) по 5), мы приходим к цепочке, в которой имеется металингвистическая переменная, обозначающая арифметическое выражение, т. е. то самое понятие, которое мы выводим. Для того, чтобы выйти из этого «заколдованного круга», мы должны применить другие, не указанные здесь варианты из правых частей формул.

Наличие рекурсий несколько затрудняет чтение и усвоение металингвистических формул, однако ниже будет доказано, что рекурсии необходимы для того, чтобы язык, описываемый формулами, был бесконечным, т. е. включал в себя бесконечное число цепочек основных символов. Так как число различных правильных программ любого алгоритмического языка бесконечно, все эти языки бесконечны, и для описания их синтаксиса приходится использовать формы Бекуса, содержащие явные и неявные рекурсии.

Для описания синтаксиса таких языков, как КОБОЛ и PL/1, используется несколько другой метасинтаксический язык. Легко показать, что этот язык эквивалентен нормальным формам Бекуса, т. е. любое правило, записанное на этом языке, может быть

однозначно представлено в виде одной или нескольких форм Бекуса и наоборот.

Ниже мы покажем, что формы Бекуса являются способом представления порождающих грамматик определенного класса, именно, контекстно-свободных порождающих грамматик.

Описанию свойств этого и других классов порождающих грамматик посвящены следующие параграфы данной главы.

§ 2. Определение и общие свойства порождающих грамматик

Нормальные формы Бекуса, рассмотренные нами в предыдущем параграфе, используются с двумя целями: во-первых, они позволяют формально представить правила построения основных конструкций языка, являясь, таким образом, метасинтаксическим языком; во-вторых, они включают в себя элементы семантики, так как в них входят осмысленные названия описываемых конструкций — последовательности слов в угловых скобках, соответствующие металингвистическим переменным. Мы несколько преобразуем нормальные формы Бекуса с целью сделать их более удобными для изучения математическими средствами и исключить ненужные нам в данном случае элементы семантики. Для преобразования выполним следующие действия над каждой металингвистической формулой, описывающей некоторую конструкцию языка:

1) Заменяем название в угловых скобках каждой конструкции языка некоторым символом так, чтобы различным конструкциям соответствовали разные символы. Эти символы будем выбирать из числа тех, которые не входят во множество основных символов описываемого языка. Множество таких символов назовем *нетерминальными* или *вспомогательными символами*.

2) Металингвистическую связку $:: =$ заменим на знак \rightarrow .

3) Каждую формулу, имеющую несколько вариантов в правой части, заменим на соответствующее число формул с одним вариантом. Тем самым отпадает

необходимость в использовании металлингвистической связи |.

4) Основные символы языка будем называть также *терминальными символами*, цепочки основных символов — *терминальными цепочками*. Образует множество терминальных символов, включив в него все используемые в правилах основные символы языка.

5) Выделим нетерминальный символ, соответствующий самой общей из описываемых конструкций, и назовём его *начальным символом*. При полном описании синтаксиса языка программирования начальный символ соответствует конструкции «программа».

Проиллюстрируем эти действия на примере нормальных форм Бекуса, описывающих конструкцию «число» языка АЛГОЛ-60 (см. § 1 настоящей главы). В данном примере будем использовать в качестве нетерминальных символов заглавные буквы русского алфавита. Обозначим используемые в формулах конструкции буквами русского алфавита следующим образом: число — *Ч*, число без знака — *Н*, десятичное число — *Д*, порядок — *П*, правильная дробь — *М*, целое — *Ф*, целое без знака — *Б*, цифра — *Ц*.

Итак, множество нетерминальных символов в нашем примере состоит из восьми символов: *Ч*, *Н*, *Д*, *П*, *М*, *Ф*, *Б*, *Ц*. Множество терминальных символов состоит из символов, используемых для написания чисел на АЛГОЛе-60. В него входят следующие символы: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., +, —, 10. Начальным символом является в данном случае символ, соответствующий конструкции «число», т. е. символ *Ч*, так как число является наиболее общей из рассматриваемых в примере конструкций. Нормальные формы Бекуса после преобразований примут следующий вид:

$Ч \rightarrow Н$	$П \rightarrow {}_{10}Ф$	$Ц \rightarrow 2$
$Ч \rightarrow +Н$	$М \rightarrow .Б$	$Ц \rightarrow 3$
$Ч \rightarrow -Н$	$Ф \rightarrow Б$	$Ц \rightarrow 4$
$Н \rightarrow Д$	$Ф \rightarrow +Б$	$Ц \rightarrow 5$
$Н \rightarrow П$	$Ф \rightarrow -Б$	$Ц \rightarrow 6$
$Н \rightarrow ДП$	$Б \rightarrow Ц$	$Ц \rightarrow 7$
$Д \rightarrow Б$	$Б \rightarrow БЦ$	$Ц \rightarrow 8$
$Д \rightarrow М$	$Ц \rightarrow 0$	$Ц \rightarrow 9$
$Д \rightarrow БМ$	$Ц \rightarrow 1$	

Вывод числа $1.5_{10}-2$ может быть теперь переписан следующим образом: $Ч, Н, ДП, БМП, ЦМП, МП, 1.БП, 1.ЦП, 1.5П, 1.5_{10}Ф, 1.5_{10}-Б, 1.5_{10}-Ц, 1.5_{10}-2$.

Множество терминальных символов, множество нетерминальных символов с выделенным в нем начальным символом и множество правил описанного вида входят в задание так называемой *формальной порождающей грамматики*. Такие грамматики часто используются для формального описания синтаксиса языков программирования, а также естественных языков. Правила, входящие в грамматику (они называются *правилами вывода*), определяют подстановки, которые можно производить при построении той или иной конструкции языка. При выполнении подстановки в цепочке, состоящей из терминальных и нетерминальных символов, выделяется последовательность символов (подцепочка), совпадающая с левой частью некоторого правила, которая и заменяется на цепочку, составляющую правую часть данного правила. *Язык, порождаемый грамматикой*, это множество терминальных цепочек, которые можно вывести из начального символа.

При описании грамматики, порождающей числа языка АЛГОЛ-60, мы использовали правила, в левых частях которых находятся нетерминальные символы (один в каждом правиле). Грамматики, имеющие правила такого вида, называются контекстно-свободными или бесконтекстными грамматиками. Именно такие грамматики обычно используются для описания синтаксиса языков программирования. Однако вид правил может быть обобщен. Можно допустить правила, в левых частях которых находятся цепочки, состоящие из терминальных и нетерминальных символов. Если не накладывать никаких ограничений на цепочки, составляющие левые и правые части правил, то мы получим формальные порождающие грамматики в наиболее общем виде.

Перейдем к формальному определению порождающих грамматик, их классификации и свойствам.

О п р е д е л е н и е. *Порождающей грамматикой* G называется четверка объектов $\langle V_T, V_A, I, S \rangle$, где

V_T — конечное множество (словарь) терминальных или основных символов;

V_A — конечное множество (словарь) нетерминальных или вспомогательных символов, $V_T \cap V_A = \emptyset$;

I — начальный символ или аксиома грамматики, $I \in V_A$;

S — конечное множество правил вида $\xi \rightarrow \eta$, где ξ и η — цепочки, состоящие из терминальных и нетерминальных символов (в дальнейшем цепочку, состоящую из символов некоторого словаря, будем называть *цепочкой над этим словарем*). Правила из S называются правилами вывода грамматики G .

Нетрудно видеть, что грамматики являются частным случаем исчислений математической логики. Объектами, к которым в исчислении, представленном порождающей грамматикой, применяются правила вывода, являются цепочки над объединением словарей терминальных и нетерминальных символов этой грамматики. Единственная аксиома такого исчисления — это цепочка, состоящая лишь из начального символа, а правила вывода представляют собой правила подстановок, входящие в множество S .

Рассмотрим некоторые понятия, связанные с порождающими грамматиками.

Определения. 1) Говорят, что цепочка ω_1 непосредственно выводима из цепочки ω_0 в грамматике $G(\omega_0 \Rightarrow_G \omega_1)$, если $\omega_0 = \xi_1 \xi \xi_2$, $\omega_1 = \xi_1 \eta \xi_2$, где ξ_1 , ξ_2 , ξ и η — некоторые цепочки над объединением словарей грамматики, и правило $\xi \rightarrow \eta$ является ее правилом вывода.

2) Говорят, что цепочка ω_n выводима из цепочки ω_0 в грамматике $G(\omega_0 \xRightarrow{*}_G \omega_n)$, если существует последовательность цепочек $\omega_0, \omega_1, \dots, \omega_n$, $n > 0$, такая, что $\omega_i \xRightarrow{*}_G \omega_{i+1}$ при $i = 0, 1, \dots, n-1$. Эта последовательность называется выводом ω_n из ω_0 в грамматике G .

Иногда вместо $\omega_0 \xRightarrow{*}_G \omega_1$ и $\omega_0 \xRightarrow{*}_G \omega_n$ будем писать просто $\omega_0 \Rightarrow \omega_1$ и $\omega_0 \Rightarrow \omega_n$.

3). Если $\omega_0 \xRightarrow{*}_G \omega_n$ и $\omega_0 = A \in V_A$, то говорят, что цепочка ω_n выводима из символа A . Терминальную цепочку, выведенную из символа, будем иногда называть *терминальным порождением* этого символа.

4) Совокупность терминальных цепочек грамматики G , выводимых в ней из начального символа, называется *языком*, *порождаемым* этой грамматикой, и обозначается через $L(G)$.

Мы уже рассматривали интерпретацию различных объектов грамматики при ее использовании для описания синтаксиса языков программирования. Напомним, что словарь терминальных символов интерпретируется как множество основных символов языка, словарь вспомогательных символов — как множество имен синтаксических конструкций, правила вывода — как правила построения конструкций языка. Начальный символ интерпретируется как программа, написанная на языке, синтаксис которого описывается данной грамматикой. Таким образом, язык, порождаемый грамматикой, есть множество синтаксически правильных программ. Так как грамматики, описывающие синтаксис большинства языков программирования, очень громоздки, мы будем в качестве примеров приводить грамматики, описывающие отдельные конструкции языков, как это уже было сделано для чисел языка АЛГОЛ-60.

Можно показать, что класс языков, порождаемых грамматиками в том общем виде, который мы определили выше, соответствует рекурсивно перечислимым множествам. Следовательно, грамматики представляют собой порождающие устройства очень общего характера. Одной из основных проблем, связанных с практическим применением грамматик, является *проблема распознавания*. Проблема распознавания *разрешима*, если существует такой алгоритм, который за конечное число шагов дает ответ на вопрос, входит ли произвольная цепочка над основным словарем некоторой грамматики в язык, порождаемый этой грамматикой. Если такой алгоритм существует, то язык называется *распознаваемым*. Если число шагов алгоритма распознавания зависит от длины цепочки и может быть оценено до выполнения алгоритма, язык называется *легко распознаваемым*. Естественно, что представляют практический интерес лишь те грамматики, которые порождают распознаваемые языки. В противном случае мы не могли бы обнаружить ошибки при отладке программ и даже не могли бы сказать,

правильна ли синтаксически та или иная программа. Поэтому практическая работа с нераспознаваемыми языками программирования невозможна. Между тем известно, что класс рекурсивно перечислимых множеств шире класса рекурсивных, т. е. распознаваемых, множеств. В связи с этим класс всех порождающих грамматик, определение которых приведено выше, не представляет интереса с точки зрения изучения языков программирования. Мы рассмотрим несколько частных классов порождающих грамматик таких, что соответствующие им языки легко распознаваемы. При определении этих классов будем накладывать различные ограничения на правила вывода. Эти ограничения будут накладываться, во-первых, на длину цепочек, входящих в левые и правые части правил, и, во-вторых, на вид этих цепочек.

Определение. *Длиной цепочки ω над некоторым словарем (обозначение $l(\omega)$) будем называть число символов, входящих в эту цепочку. Если $l(\omega) = 0$, цепочка называется пустой. Пустую цепочку будем обозначать через λ .*

Рассмотрим наиболее широкий из интересующих нас классов грамматик — класс неукорачивающих грамматик.

Определение. Грамматика называется *неукорачивающей*, если для любого ее правила вывода $\xi \rightarrow \eta$ справедливо неравенство $l(\xi) \leq l(\eta)$.

Таким образом, правая часть каждого правила вывода неукорачивающей грамматики не короче его левой части. Существенным является то, что в любом выводе в произвольной неукорачивающей грамматике длина цепочек может лишь возрастать, т. е. если имеется произвольный вывод $\omega_0, \omega_1, \dots, \omega_n$ в неукорачивающей грамматике, то для $i < k$, $0 \leq i < n$, $0 < k \leq n$, справедливо неравенство $l(\omega_i) \leq l(\omega_k)$. Это свойство позволяет доказать, что языки, порождаемые неукорачивающими грамматиками, являются легко распознаваемыми.

Теорема 1. *Язык $L(G)$, порождаемый неукорачивающей грамматикой G , легко распознаваем.*

Доказательство. Для доказательства мы построим алгоритм, который за конечное число шагов определяет, входит ли произвольная терминальная

цепочка грамматики G в язык $L(G)$. Прежде всего заметим, что простой перебор всех возможных выводов терминальных цепочек в грамматике G не подходит, так как многие грамматики порождают бесконечные языки Π , следовательно, в этих грамматиках имеется бесконечное число таких выводов. Воспользуемся тем, что G — неукорачивающая. Пусть $l(\omega) = n$, тогда в выводе ω могут встречаться лишь цепочки, длина которых не превосходит n . При поиске такого вывода мы можем ограничиться рассмотрением лишь так называемых *бесповторных выводов*, т. е. таких, в которых все цепочки различны. В самом деле, если имеется вывод с повторяющимися цепочками, то может быть построен бесповторный вывод той же заключительной цепочки. Например, в выводе $\omega_0, \dots, \omega_i, \dots, \omega_k, \dots, \omega_m$ пусть $\omega_i = \omega_k$. Исключим из него ту часть, в которой из ω_i выводится ω_k . Очевидно, получим правильный вывод той же цепочки, в котором число повторяющихся цепочек будет по крайней мере на 1 меньше, чем в первоначальном выводе. Продолжая этот процесс, мы получим за конечное число шагов бесповторный вывод этой же цепочки. Итак, вывод цепочки ω , если он существует, должен содержаться среди бесповторных выводов из начального символа, в которых встречаются цепочки длиной не более n .

Алгоритм распознавания можно теперь построить следующим образом.

1) Находятся все цепочки над $V_T \cup V_A$ данной грамматики длиной не более n .

2) Строятся все бесповторные последовательности выделенных на первом этапе цепочек, которые начинаются с цепочки I и заканчиваются цепочкой ω . При этом каждая последовательность анализируется — проверяется, является ли она правильным выводом в G . Если это так, то $\omega \in L(G)$ и алгоритм заканчивает работу. Если же ни одна из последовательностей не является правильным выводом, то, очевидно, $\omega \notin L(G)$.

Число цепочек длины не более n и число бесповторных последовательностей таких цепочек конечны и при фиксированных V_T и V_A зависят лишь от n . Поэтому язык $L(G)$ легко распознаваем. Доказательство теоремы закончено.

Хотя неукорачивающие грамматики порождают легко распознаваемые языки, они неудобны для описания синтаксиса языков по двум причинам. Во-первых, в них допускаются подстановки вместо подцепочек, содержащих терминальные символы. Поэтому естественная интерпретация нетерминальных символов как металингвистических переменных в значительной мере теряет смысл. Во-вторых, указанный при доказательстве теоремы 1 алгоритм распознавания практически неприменим из-за большого числа шагов, а предложить другой менее трудоемкий алгоритм непросто. Ниже мы рассмотрим несколько классов грамматик, более удобных для использования, в частности, один класс укорачивающих грамматик, для которых проблема распознавания разрешима. Нас будут интересовать следующие классы:

1) Грамматики *непосредственных составляющих* или контекстные грамматики (НС-грамматики). Это неукорачивающие грамматики с правилами вывода вида

$$\xi_1 A \xi_2 \longrightarrow \xi_1 \eta \xi_2,$$

где ξ_1 и ξ_2 — произвольные (возможно, пустые) цепочки над $V_T \cup V_A$, A — нетерминальный символ, η — непустая цепочка над $V_T \cup V_A$.

Каждое правило вывода НС-грамматики указывает подстановку некоторой цепочки вместо нетерминального символа. Однако возможность реализации подстановки зависит от символов, окружающих заменяемый, или, как часто говорят, от контекста, в котором находится заменяемый символ. Нетерминальные символы могут интерпретироваться как металингвистические переменные или, иными словами, конструкции языка, а вывод предложений языка может рассматриваться как информация о синтаксической структуре этих предложений. Синтаксическая структура предложений удобнее всего представляется в виде так называемого дерева вывода, которое может быть однозначно получено по выводу. Ниже мы подробно рассмотрим деревья выводов и некоторые их свойства.

Можно доказать, что класс языков, порождаемых НС-грамматиками, совпадает с классом языков,

порождаемых неукорачивающими грамматиками [16]. Иными словами, по любой неукорачивающей грамматике можно построить НС-грамматику, порождающую тот же язык. Такую грамматику будем называть *эквивалентной* исходной. Классы грамматик будем называть *эквивалентными*, если порождаемые ими классы языков совпадают. Итак, класс НС-грамматик эквивалентен классу неукорачивающих грамматик.

2) *Контекстно-свободные* или бесконтекстные грамматики (КС-грамматики). Это неукорачивающие грамматики с правилами вывода вида $A \rightarrow \eta$, где A — нетерминальный символ, а η — произвольная непустая цепочка над $V_T \cup V_A$.

Как видно из определения, любая КС-грамматика является одновременно и НС-грамматикой. Поэтому класс КС-грамматик входит в класс НС-грамматик. Можно показать, что класс КС-языков (т. е. языков, порождаемых КС-грамматиками) уже класса НС-языков. Ниже мы приведем примеры языков, являющихся НС-, но не КС-языками.

КС-грамматики играют главную роль при формальном описании и изучении синтаксиса языков программирования. Этому способствует то, что, с одной стороны, средствами КС-грамматик удастся достаточно полно описать синтаксическую структуру языков программирования (хотя некоторые синтаксические правила не описываются при помощи КС-грамматик), а с другой стороны, достаточно хорошо разработаны алгоритмы распознавания КС-языков, которые составляют основное содержание первого блока трансляторов — блока синтаксического анализа. Мы подробно рассмотрим те свойства КС-грамматик и языков, которые представляют наибольший интерес для работы с алгоритмическими языками.

3) *Укорачивающие контекстно-свободные* грамматики (УКС-грамматики). Эти грамматики имеют правила того же вида, что и КС-грамматики, но цепочка η может быть и пустой. Таким образом, этот класс грамматик не входит в класс неукорачивающих грамматик. Можно, однако, показать, что по любой УКС-грамматике можно построить почти эквивалентную ей КС-грамматику. Почти эквивалентной грамматикой мы будем называть такую, которая порождает

тот же язык, что и исходная грамматика, за исключением пустой цепочки. Таким образом, языки, порождаемые УКС-грамматиками, являются легко распознаваемыми (проверить, входит ли в УКС-язык пустая цепочка, не представляет труда).

УКС-грамматики представляют интерес потому, что именно они используются для описания синтаксиса языков программирования. Теорема о почти эквивалентности позволяет свести изучение УКС-грамматик к изучению соответствующих свойств КС-грамматик.

4) *Автоматные* грамматики (А-грамматики). Это наиболее простой класс грамматик. А-грамматики имеют правила вида $A \rightarrow bB$ и $A \rightarrow b$, где $A \in V_A$, $B \in V_A$, $b \in V_T$. Класс А-языков уже класса КС-языков. Алгоритм распознавания автоматных языков чрезвычайно прост. Автоматные грамматики используются в основном при предварительном преобразовании программ с целью их представления в форме, более удобной для дальнейшего анализа. В частности, при помощи автоматных грамматик, порождающих такие конструкции, как идентификатор и число, можно выделить их при чтении программы, записать в специальные таблицы, а в программу вставить на их места ссылки на соответствующие строки таблиц. Так как эти ссылки будут иметь стандартную длину и в дальнейшем могут восприниматься как основные символы, анализ программы значительно упростится.

При рассмотрении свойств грамматик и в примерах мы будем использовать следующие обозначения. Пусть V — некоторое множество символов. Обозначим через V^* множество всех цепочек над V (включая пустую цепочку). Нетерминальные символы грамматик будем обычно обозначать большими, а терминальные — малыми начальными буквами латинского или русского алфавитов. Если символ может быть любым, будем обозначать его одной из малых начальных букв греческого алфавита. Наконец, для обозначения терминальных цепочек будем использовать буквы x , y и z , а для любых цепочек над объединением терминального и нетерминального словарей — несколько последних букв греческого алфавита. Использование этих обозначений позволит нам при рассмотрении

конкретных грамматик в большинстве случаев указывать лишь на их правила вывода, считая, что словари терминальных и нетерминальных символов состоят из символов, встречающихся в правилах вывода.

Приведем несколько примеров грамматик различных классов. В примерах будем рассматривать как грамматики, порождающие некоторые конструкции языков программирования, так и не имеющие связи с языками программирования, но интересные с каких-либо других точек зрения грамматики.

Пример 1. Любой конечный язык, в который не входит пустая цепочка, является A -языком. Пусть $L_1 = \{x_1, x_2, \dots, x_n\}$. Выберем некоторую x_i , $1 \leq i \leq n$. Пусть $x_i = a_1 a_2 \dots a_m$. Выберем $m-1$ нетерминальных символ A_1, A_2, \dots, A_{m-1} и образуем правила вывода $I \rightarrow a_1 A_1, A_1 \rightarrow a_2 A_2, \dots, A_{m-1} \rightarrow a_m$. Аналогично образуем правила вывода и по всем другим цепочкам из L_1 , выбирая каждый раз новые нетерминальные символы. Включим в A -грамматику с начальным символом I все образованные правила. Очевидно, она порождает язык L_1 .

Пример 2. Обозначим через L_2 множество всех идентификаторов языка АЛГОЛ-60. Если использовать формы Бекуса для идентификаторов из [1], то получим порождающую грамматику со следующими правилами: $I \rightarrow \text{ИБ}, I \rightarrow \text{ИЦ}, I \rightarrow \text{Б}$. К этим правилам нужно приписать правила для подстановки вместо нетерминальных символов Б и Ц соответственно букв латинского алфавита и цифр. Легко построить также автоматную грамматику, порождающую L_2 . Эта грамматика будет содержать правила следующего вида:

$$I \rightarrow \langle \text{б} \rangle, I \rightarrow \langle \text{б} \rangle I_1, I_1 \rightarrow \langle \text{б} \rangle, I_1 \rightarrow \langle \text{ц} \rangle, I_1 \rightarrow \langle \text{б} \rangle I_1, I_1 \rightarrow \langle \text{ц} \rangle I_1.$$

При задании этой грамматики мы указали не правила вывода, а так называемые схемы правил, каждая из которых определяет некоторое количество конкретных правил. Так, чтобы получить конкретные правила из первой схемы, нужно вместо $\langle \text{б} \rangle$ подставить в схему буквы латинского алфавита. Таким образом, схема определяет 52 правила грамматики: $I \rightarrow a, \dots, I \rightarrow Z$.

Пример 3. Рассмотрим L_3 — язык простых арифметических выражений АЛГОЛа-60. Соответствующие формы Бекуса из [1] определяют следующую КС-грамматику, порождающую L_3 :

- | | |
|-----------------------------------|------------------------------|
| 1) $E \rightarrow T$, | 9) $P \rightarrow V$, |
| 2) $E \rightarrow AT$, | 10) $P \rightarrow F$, |
| 3) $E \rightarrow EAT$, | 11) $P \rightarrow (E)$, |
| 4) $T \rightarrow M$, | 12) $O \rightarrow \times$, |
| 5) $T \rightarrow TOM$, | 13) $O \rightarrow /$, |
| 6) $M \rightarrow P$, | 14) $O \rightarrow \div$, |
| 7) $M \rightarrow M \uparrow P$, | 15) $A \rightarrow +$, |
| 8) $P \rightarrow N$, | 16) $A \rightarrow -$. |

Нетерминальные символы, используемые в этой грамматике, имеют следующую интерпретацию: E — простое арифметическое выражение, T — терм, M — множитель, P — первичное выражение, N — число без знака, V — переменная, F — указатель функции, O — знак операции типа умножения, A — знак операции типа сложения.

В перечисленные выше правила грамматики не включены правила для нетерминальных символов N , V и F . Для получения этих правил также можно использовать металингвистические формулы из [1], описывающие синтаксис соответствующих конструкций. При необходимости полного задания грамматики, порождающей язык L_3 , эти правила должны быть включены во множество правил вывода грамматики.

Приведем теперь примеры грамматик, не связанных непосредственно с языками программирования.

Пример 4. Рассмотрим язык $L_4 = \{a^n b^n\} \ n=1, 2, \dots$. Здесь a^n означает цепочку, содержащую n вхождений символа a . Этот язык порождается КС-грамматикой, включающей всего два правила вывода: $I \rightarrow alb$ и $I \rightarrow ab$.

Пример 5. Расширим цепочки, входящие в язык L_4 , приписав к ним справа еще n вхождений символа a : $L_5 = \{a^n b^n a^n\}$. Интересно, что такое небольшое усложнение структуры входящих в язык цепочек выводит его из класса языков, порождаемых КС-грамматиками. Он порождается следующей неукорачивающей грамматикой:

- | | |
|--------------------------|-------------------------|
| 1) $I \rightarrow aCa,$ | 4) $aB \rightarrow Ba,$ |
| 2) $C \rightarrow aCBa,$ | 5) $bB \rightarrow bb.$ |
| 3) $C \rightarrow b,$ | |

Вывод терминальной цепочки в данной грамматике разбивается на следующие этапы:

а) Применяется правило 1).

б) Применяется $n-1$ раз правило 2). При $n=1$ этот этап вывода пропускается.

в) Применяется правило 3).

г) Если $n=1$, вывод закончен. В противном случае $(n-1) \times \times (n-2)/2$ раз применяется правило 4) и $n-1$ раз правило 5). Порядок применения этих правил может варьироваться.

Эту грамматику можно преобразовать в НС-грамматику. Для этого нужно исключить из множества правил вывода правило 4). Это в свою очередь приведет к включению в S четырех новых правил и в V_A двух новых нетерминальных символов. В качестве новых символов выберем, например, K , M . Во всех правилах вывода, кроме 4), заменим символ a на M . Вместо правила 4) включим в грамматику следующие 3 правила:

- | |
|---------------------------|
| 4.1) $MB \rightarrow KB,$ |
| 4.2) $KB \rightarrow KM,$ |
| 4.3) $KM \rightarrow BM.$ |

Включим в S также следующее новое правило:

6) $M \rightarrow a.$

Каждое из правил 4.1) — 4.3) является правилом НС-грамматики. Поскольку нетерминальный символ K не встречается в других правилах, в каждом выводе терминальной цепочки, в котором используется правило 4.1), должны применяться также и остальные правила — 4.2) и 4.3). Символ M — это так называемый двойник символа a и может заменяться лишь на него, не изменяя язык, порожаемый грамматикой. Из этих соображений следует, что мы получили НС-грамматику, эквивалентную исходной.

Рассмотренная процедура замены неукорачивающего правила группой правил НС-грамматики используется при доказательстве эквивалентности классов неукорачивающих и НС-грамматик.

Пример 6. $L_6 = \{xbx\}$, где x — произвольная цепочка над словарем $\{a_1, a_2, \dots, a_n\}$. Таким образом, в язык L_6 входят все цепочки, состоящие из двух одинаковых частей, между которыми находится разделитель b . Можно показать, что такой язык не порождается никакой КС-грамматикой. Однако порождающая его НС-грамматика очень проста:

- 1) $I \rightarrow b;$
- 2) $I \rightarrow IA_i a_i, i = 1, 2, \dots, n;$
- 3) $a_i A_j \rightarrow A_j a_i, i, j = 1, 2, \dots, n;$
- 4) $b A_i \rightarrow a_i b, i = 1, 2, \dots, n.$

Под номерами 2, 3 и 4 здесь приводятся схемы правил, определяющие, соответственно, n , n^2 и n правил вывода при выбранных в указанных пределах значениях индексов.

Пример 7. Рассмотрим теперь язык, в некотором смысле обратный предыдущему: $L_7 = \{x_1 b x_2\}$, где x_1 и x_2 — несовпадающие цепочки над словарем $\{a_1, a_2, \dots, a_n\}$. Оказывается, L_7 является КС-языком, хотя порождающая его грамматика довольно сложна [15]. Разобьем правила вывода этой грамматики на две группы. При помощи первой группы правил порождаются цепочки, у которых $l(x_1) \neq l(x_2)$:

- 1) $I \rightarrow I_1$
- 2) $I_1 \rightarrow a_i I_1 a_i, i, j = 1, \dots, n;$
- 3) $I_1 \rightarrow a_i C, i = 1, \dots, n;$
- 4) $C \rightarrow a_i C, i = 1, \dots, n;$
- 5) $C \rightarrow b;$
- 6) $I_1 \rightarrow D a_i, i = 1, \dots, n;$
- 7) $D \rightarrow D a_i, i = 1, \dots, n;$
- 8) $D \rightarrow b.$

Если $l(x_1) > l(x_2)$, используются правила с 1) по 5), если же $l(x_1) < l(x_2)$, то вместо правил 3), 4) и 5) используются соответствующие им правила 6), 7) и 8).

Вторая группа КС-грамматики применяется при порождении цепочек, у которых $l(x_1) = l(x_2)$. Однако при применении этих правил могут быть выведены и цепочки с неравными по длине частями. Вывод производится таким образом, что хотя бы в одном случае сравниваемые символы из цепочек x_1 и x_2 не совпадают. Эта группа состоит из следующих правил:

- 9) $I \rightarrow I_2;$
- 10) $I_2 \rightarrow I_2 a_i, i = 1, \dots, n;$
- 11) $I_2 \rightarrow A_i a_i, i = 1, \dots, n;$
- 12) $A_i \rightarrow a_j A_i a_k, i, j, k = 1, \dots, n;$
- 13) $A_i \rightarrow a_j B, i, j = 1, \dots, n; i \neq j;$
- 14) $B \rightarrow a_i B, i = 1, \dots, n;$
- 15) $B \rightarrow b.$

Вывод с использованием правил второй группы производится следующим образом. Сначала при помощи правил 10) порождается часть цепочки x_2 правее символа, не совпадающего с соответствующим символом цепочки x_1 . Затем применяется одно из правил 11), порождающее символ цепочки x_2 , который не будет совпадать с символом из x_1 . Далее для вывода левых частей цепочек x_1 и x_2 используются правила 12). Заметим, что длины порождаемых левых частей равны. Одно из правил 13) порождает символ цепочки x_1 , не совпадающий с соответствующим символом из x_2 . Это достигается благодаря тому, что в каждом правиле вида 13) индексы нетерминального символа в левой части и терминального в правой различны. Правила 14) применяются для вывода правой части цепочки x_1 , а правило 15) заканчивает вывод, порождая разделитель b .

Естественным обобщением языка L_7 является язык, состоящий из цепочек, которые могут содержать не две различные подцепочки x_1 и x_2 , а произвольное число таких подцепочек. Этот язык не является КС-языком. Можно показать, что он входит в класс НС-языков, но порождающая его НС-грамматика очень сложна

§ 3. Некоторые свойства контекстно-свободных грамматик

КС-грамматики являются наиболее хорошо изученным классом грамматик. Свойствам грамматик этого класса посвящен ряд монографий (см., например, [13] и [21]) и большое число журнальных публикаций. В теории КС-грамматик рассматривается целый ряд вопросов, таких, как соответствие между КС-грамматиками и автоматами с магазинной памятью, операции над КС-языками, алгоритмические проблемы. Многие публикации посвящены различным подклассам КС-грамматик, обладающим свойствами, облегчающими их изучение и использование. Мы рассмотрим, однако, лишь те черты этих грамматик, которые имеют непосредственное отношение к их использованию в теории языков программирования. Вопросы, которые будут нас интересовать в настоящем параграфе, могут быть разбиты на три группы. Во-первых, мы обсудим способы представления выводов правильных цепочек, входящих в языки, порождаемые КС-грамматиками. Этот вопрос имеет большое прикладное значение. Получение информации о выводе программы, написанной на алгоритмическом языке, является основной целью работы первого блока транслятора — блока анализа. Между тем, представление вывода в виде последовательности цепочек

крайне неэкономно, так как при таком способе многократно выписываются повторяющиеся части цепочек. Кроме того, этот способ не позволяет получить в удобном виде информацию о *синтаксических конструкциях* (терминальных цепочках, которые выведены из нетерминальных символов, используемых при выводе некоторой программы). Наилучшим способом представления вывода терминальной цепочки в некоторой КС-грамматике является так называемое синтаксическое дерево вывода. Структура и свойства синтаксических деревьев вывода будут подробно рассмотрены в настоящем параграфе.

Вторая группа вопросов, интересующих нас при рассмотрении свойств КС-грамматик, связана со способностью КС-грамматик порождать бесконечные языки. Естественно считать, что на каждом языке программирования можно написать бесконечно много различных программ. В противном случае язык программирования (по крайней мере не узко специализированный) не отвечал бы тем целям, которые ставили перед собой его разработчики. Поэтому важно знать, какими свойствами должна обладать КС-грамматика для того, чтобы она порождала бесконечный язык.

Наконец, учитывая что для описания синтаксиса языков программирования нужны укорачивающие КС-грамматики, рассмотрим вопрос о связи этих грамматик с неукорачивающими КС-грамматиками.

Перейдем теперь к изучению синтаксических деревьев вывода.

Определения. 1) Цепочку η будем называть *подцепочкой* цепочки ξ , если последняя может быть представлена в виде $\xi = \xi_1 \eta \xi_2$. Здесь ξ_1 и ξ_2 — некоторые (возможно, пустые) цепочки. Итак, любая связанная часть цепочки является ее подцепочкой.

2) Так как цепочка может включать в себя несколько одинаковых подцепочек (например, цепочка *асас* включает в себя две подцепочки *ас*), для выделения конкретной подцепочки η в цепочке ξ нужно указывать не только саму подцепочку η , но и ее положение в цепочке ξ . В связи с этим введем понятие *вхождения* подцепочки в цепочку. Будем называть вхождением подцепочки η в цепочку ξ пару объектов (η, i) , где η — данная подцепочка, а i — порядковый

номер левого символа подцепочки η в цепочке ξ . Таким образом, в приведенном выше примере можно рассматривать два различных вхождения подцепочки ac : $(ac, 1)$ и $(ac, 3)$. Заметим, что в частном случае подцепочка может состоять лишь из одного символа. Тогда говорят о вхождении символа в цепочку. Так, в цепочке $acac$ имеется два вхождения символа c : $(c, 2)$ и $(c, 4)$. Иногда вхождение подцепочки η в цепочку ξ будем указывать в виде $\xi_1\eta\xi_2$.

Пусть $\Omega = \omega_0, \omega_1, \dots, \omega_n$ — вывод в произвольной КС-грамматике. Рассмотрим две соседние цепочки вывода ω_m и ω_{m+1} . Они должны быть представимы в виде $\omega_m = \xi_1 A \xi_2$, $\omega_{m+1} = \xi_1 \eta \xi_2$, где ξ_1, ξ_2 и η — некоторые цепочки, A — нетерминальный символ и на $m+1$ -м шаге вывода Ω к выделенному вхождению символа A в цепочке ω_m применялось правило вывода $A \rightarrow \eta$. Сравнивая представления цепочек, легко заметить, что каждому вхождению какого-либо символа в ω_m , за исключением того, к которому применялось правило вывода, можно естественным образом поставить в соответствие вхождение того же символа в цепочку ω_{m+1} . Такие соответствующие друг другу вхождения символов в соседние цепочки вывода будем называть *копиями* символов в выводе. Выделенное вхождение символа A в цепочке ω_m , к которому на $m+1$ -м шаге вывода применялось правило вывода, будем называть *непосредственным предком* всех вхождений символов, образующих подцепочку η в ω_{m+1} , а эти последние вхождения — *непосредственными потомками* первого.

Приведем теперь точные определения рассматриваемых понятий. Пусть $l(\xi_1) = i$, $l(\omega_m) = j$, $l(\eta) = k$. Тогда два вхождения символа α : (α, p) в цепочку ω_m и (α, q) в цепочку ω_{m+1} называются копиями друг друга в выводе Ω , если $p \neq i+1$ и

$$q = \begin{cases} p, & \text{если } 1 \leq p \leq i, \\ p+k-1, & \text{если } i+2 \leq p \leq i. \end{cases}$$

Рассмотрим вхождения $(A, i+1)$ в цепочку ω_m и (α, q) в цепочку ω_{m+1} . Первое вхождение называется непосредственным предком второго (а второе — непосредственным потомком первого) в выводе Ω , если $i+1 \leq q \leq i+k$.

Понятия копий и непосредственных предков и потомков легко обобщить на случай несоседних цепочек вывода. Заметим, что любое вхождение α_m некоторого символа в цепочку ω_m вывода Ω однозначно определяет последовательность $\alpha_m, \alpha_{m-1}, \dots, \alpha_k$ вхождений символов в цепочки $\omega_m, \omega_{m-1}, \dots, \omega_k$ данного вывода такую, что любые два соседних вхождения последовательности являются либо копиями, либо непосредственным предком и потомком в выводе Ω . Пусть $m \geq k+1$. Тогда α_m и α_k называются копиями в выводе Ω , если все пары соседних вхождений выделенной последовательности являются копиями. Если же хотя бы одна пара соседних вхождений данной последовательности состоит из непосредственных предка и потомка, α_m называется *потомком* α_k , а α_k — *предком* α_m в выводе Ω .

Пример 1. Рассмотрим приведенную в § 2. данной главы грамматику для чисел языка АЛГОЛ-60. Пусть имеется следующий вывод в этой грамматике: $\omega_0 = Ч$, $\omega_1 = +Н$, $\omega_2 = +ДП$, $\omega_3 = +БМП$, $\omega_4 = +Б.БП$, $\omega_5 = +Б.ЦП$, $\omega_6 = +Б.6П$, $\omega_7 = +БЦ.6Ц$, $\omega_8 = +ЦЦ.6П$, $\omega_9 = +2Ц.6П$, $\omega_{10} = +25.6П$, $\omega_{11} = +25.6_{10}Ф$, $\omega_{12} = +25.6_{10} - Б$, $\omega_{13} = +25.6_{10} - Ц$, $\omega_{14} = +25.6_{10} - 2$.

Вхождение (6, 5) символа 6 в цепочку ω_{14} имеет в данном выводе следующие копии:

(6,5) в цепочках $\omega_{13}, \dots, \omega_7$, (6,4) — в ω_6 ,

и следующих предков:

(Ц, 4) — в ω_5 , (Б, 4) — в ω_4 , (М, 3) — в ω_3 ,

(Д, 2) — в ω_2 , (Н, 2) — в ω_1 и (Ч, 1) — в ω_0 .

Вхождение (Б, 2) в цепочку ω_6 имеет в ω_7 двух непосредственных потомков: (Б, 2) и (Ц, 3).

Используя введенные понятия, перейдем к определению синтаксического дерева вывода в КС-грамматике. Под *деревом* мы будем понимать конечный ориентированный граф, обладающий свойствами:

1) у него имеется одна вершина, в которую не входит ни одна дуга; эту вершину будем называть *корнем дерева*;

2) в каждую из его остальных вершин входит лишь одна дуга;

3) он не содержит контуров.

Нетрудно видеть, что из корня дерева в любую его вершину ведет ровно один путь. Назовем *уровнем* вершины дерева длину такого пути, т. е. число со-

ставляющих его дуг. Будем считать, что корень дерева имеет уровень 0. *Высотой* дерева будем называть наибольший уровень его вершин. Рассмотрим две различные вершины дерева: x_i и x_j . Если в дереве существует путь из x_i в x_j , то будем говорить, что вершина x_i предшествует вершине x_j , а x_j следует за x_i . При этом, очевидно, вершина уровня n следует ровно за одной вершиной каждого из уровней $n-1$, $n-2$, ..., 0. Наконец, вершины, не предшествующие никаким вершинам дерева, т. е. не имеющие выходящих из них дуг, назовем *заключительными*, а все остальные — *незаклучительными* вершинами дерева.

Пусть задано некоторое дерево (X, U) , где X — множество его вершин, а U — частичное отображение X в X , определяющее дуги дерева. Рассмотрим графическое изображение дерева на плоскости, удовлетворяющее следующим условиям:

а) все его вершины одного уровня расположены на одной прямой;

б) прямые, соответствующие вершинам различных уровней, параллельны и расположены в порядке возрастания уровней;

в) дуги дерева не пересекаются.

Такое изображение дерева будем называть *стандартным*. По любому дереву можно построить хотя бы одно его стандартное изображение. Начиная построение, следует выбрать на плоскости некоторую прямую, которую мы для определенности будем считать горизонтальной, и расположить на ней корень дерева. Далее дерево строится по уровням. Пусть уже построено m уровней дерева. Для построения $(m+1)$ -го уровня выберем на прямой, расположенной ниже прямой m -го уровня, k неперекрывающихся отрезков, где k — число незаклучительных вершин дерева m -го уровня. На первом слева отрезке расположим в произвольном порядке все вершины $(m+1)$ -го уровня, следующие за самой левой незаклучительной вершиной m -го уровня, и проведем соответствующие дуги. На втором слева отрезке выполним такое же построение для вершин, следующих за второй слева незаклучительной вершиной m -го уровня и т. д. После завершения процедуры получим стандартное изображение дерева.

По любому стандартному изображению дерева можно однозначно определить *отношение линейного порядка* для его вершин:

а) Пусть x_i и x_j — две различные вершины одного уровня. Примем $x_i < x_j$, если x_i расположена на прямой соответствующего уровня левее вершины x_j , и $x_i > x_j$ в противном случае.

б) Пусть x_i и x_j — вершины уровней m и n , причем $m < n$. Рассмотрим вершину y m -го уровня.

предшествующую x_j , и примем $x_i < x_j$, если $x_i \leq y$, и $x_i > x_j$ в противном случае.

На рис. 1 показано стандартное изображение дерева, в котором $x_i < x_j$ при $i < j$.

Пусть в КС-грамматике имеется некоторый вывод цепочки из языка, порождаемого этой грамматикой. По этому выводу можно однозначно построить *синтаксическое дерево вывода*, в котором незаключительные вершины

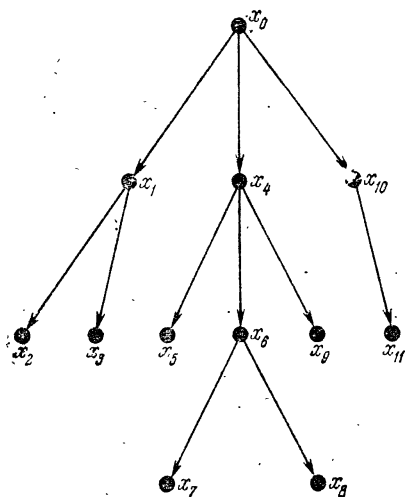


Рис. 1.

соответствуют нетерминальным символам грамматики, а заключительные — терминальным символам. Опишем процедуру построения стандартного изображения такого дерева по имеющемуся выводу.

Рассмотрим вывод $\Omega = \omega_0, \omega_1, \dots, \omega_n$ в некоторой КС-грамматике. Каждый шаг вывода определяет действия, связанные с построением соответствующего этому выводу синтаксического дерева вывода. Процедура начинается с построения корня дерева. Будем считать, что корень дерева соответствует начальному символу, и поэтому пометим его этим символом. Далее рассмотрим первый шаг вывода (переход от ω_0 к ω_1). Цепочка ω_0 состоит из одного начального символа,

и именно к нему применяется правило вывода; таким образом, ω_1 состоит из непосредственных потомков вхождения начального символа в ω_0 . Для каждого вхождения символа в ω_1 построим вершину дерева, обозначив ее соответствующим символом. Так как все построенные вершины являются вершинами одного уровня, их можно расположить на прямой таким образом, чтобы самая левая вершина соответствовала самому левому вхождению символа в ω_1 (и, следовательно, дуга, соединяющая эту вершину с корнем, была самой левой из дуг, выходящих из корня), вторая слева вершина — второму слева вхождению и т. д. На этом действия, определяемые первым шагом вывода, закончены.

Пусть уже выполнены действия, связанные с построением дерева вывода, которые соответствуют шагам вывода с первого по i -й. Рассмотрим цепочку ω_i . Определим вхождение символа, к которому на $(i+1)$ -м шаге вывода применяется правило грамматики. Найдем соответствующую этому вхождению вершину дерева. Далее, в цепочке ω_{i+1} найдем подцепочку, состоящую из непосредственных потомков данного вхождения, после чего построим для каждого символа этой подцепочки вершину дерева и соединим эти вершины дугами с вершиной, соответствующей непосредственному предку символов подцепочки. Как и на первом шаге, построение выполним так, чтобы сохранить линейный порядок расположения символов в выделенной подцепочке. Процедура построения дерева заканчивается, когда выполнены действия, определяемые последним шагом вывода.

Как уже упоминалось выше, нетерминальные символы в грамматиках, описывающих синтаксис языков программирования, соответствуют синтаксическим конструкциям этих языков. Синтаксическое дерево вывода содержит информацию о том, какие синтаксические конструкции использованы при написании некоторой программы. Эта информация представлена в дереве в достаточно удобном виде. Для ее выделения нужно рассмотреть поддерево, корень которого есть вершина дерева, соответствующая интересующей нас конструкции. Это поддерево состоит из вершин, соответствующих всем потомкам символа, которым

помечена вершина, являющаяся корнем поддерева в данном выводе. В то же время дерево вывода является весьма экономной формой представления синтаксической структуры программы, так как каждой группе копий соответствует одна вершина дерева.

Пример 2. Выводу, рассмотренному в примере 1 данного параграфа, соответствует дерево вывода, представленное на рис. 2.

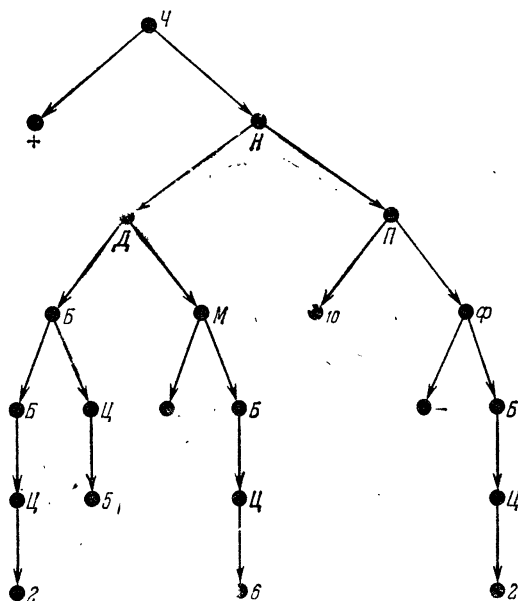


Рис. 2.

Иногда используется несколько другой, более экономный вариант дерева вывода, в котором вершины, соответствующие непосредственным потомкам одного вхождения нетерминального символа, объединяются в одну вершину и помечаются цепочкой, состоящей из этих потомков. Дуги, выходящие из объединенной вершины, соответствуют входящим в помечающую ее цепочку нетерминальным символам: первая слева дуга ведет в вершину, помеченную непосредственными потомками первого слева такого символа и т. д. Преобразованное указанным образом дерево вывода приводится на рис. 3.

Синтаксические деревья выводов представляют информацию, играющую важную роль при трансляции программ. В связи с этим большое значение имеют

способы кодирования деревьев вывода, удобные при использовании вычислительных машин. Рассмотрим один из таких способов — *линейную скобочную запись* дерева вывода, т. е. представление дерева в виде цепочки символов из множества, в которое входят терминальные и нетерминальные символы соответствующей грамматики, а также два особых символа — левая и правая скобки. Линейная скобочная запись

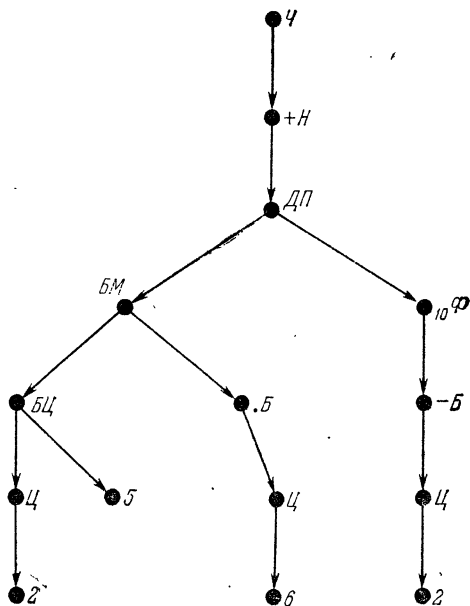


Рис. 3.

является терминальной цепочкой, вывод которой представлен соответствующим деревом, с выделенными в ней синтаксическими конструкциями. Для обозначения начала и конца каждой конструкции используются скобки. Иными словами, при помощи скобок выделяются терминальные порождения каждой группы копий нетерминальных символов данного вывода. После левой и перед правой скобкой, которые ограничивают какую-либо выделяемую подцепочку, записывается породивший ее в данном выводе нетерминальный символ.

Между деревьями выводов и их линейными скобочными записями имеется взаимно-однозначное соответствие. Рассмотрим процедуру построения цепочки, представляющей линейную скобочную запись некоторого дерева вывода. Эта процедура выполняется при помощи так называемого левого обхода дерева, при котором слева направо просматриваются все пути, ведущие от незаключительных вершин к заключительным. При выполнении процедуры просматриваются в определенной последовательности все вершины дерева и помечаются дуги, составляющие уже пройденные пути. Построение цепочки происходит путем приписывания справа символов к уже построенной части цепочки. Будем считать, что в начальный момент времени обозревается корень дерева, помеченных дуг нет, а построенная часть цепочки пуста. При этих предположениях процесс построения цепочки описывается следующим алгоритмом:

1. Если обозреваемая вершина имеет выходящие из нее дуги (т. е. вершина незаключительная), переходим к шагу 2, в противном случае к шагу 5.

2. Приписываем к цепочке левую скобку и символ, соответствующий обозреваемой вершине; переходим к шагу 3.

3. Если среди дуг, выходящих из обозреваемой вершины, есть непомеченные, помечаем самую левую из них, т. е. ведущую в наименьшую вершину, обозреваем эту вершину и переходим к шагу 1. В противном случае переходим к шагу 4.

4. Приписываем к цепочке символ, соответствующий обозреваемой вершине, и правую скобку. Если в данную вершину входит дуга, обозреваем вершину, из которой она выходит, и переходим к шагу 3. Если входящей в вершину дуги нет, т. е. обозревается корень дерева, построение линейной скобочной записи закончено.

5. Приписываем к цепочке символ, соответствующий обозреваемой вершине (так как вершина заключительная, это терминальный символ). Обозреваем вершину, из которой выходит входящая в данную вершину дуга, и переходим к шагу 3.

Из данного алгоритма видно, что разным деревьям выводов соответствуют разные линейные скобоч-

ные записи. Столь же очевидно и обратное утверждение: по любой правильной (т. е. с правильно расположенными скобками) линейной скобочной записи можно однозначно построить дерево вывода, и разным записям будут соответствовать разные деревья.

Пример 3. Линейная скобочная запись дерева вывода, приведенного в примере 2, выглядит следующим образом:

$$(4 + (H(D(B(B(C2C)B) (C5C)B) (M.(B(C6C)B)M)D) (P_{10}(\Phi - (B(C2C)B)\Phi)P)N)4).$$

Как уже отмечалось, по любому выводу в КС-грамматике соответствующее ему дерево вывода строится однозначно. Обратное утверждение в общем случае неверно. Одному синтаксическому дереву вывода могут соответствовать несколько выводов. Это происходит в том случае, если некоторая цепочка вывода содержит более чем одно вхождение нетерминального символа. Правила вывода могут быть тогда применены к любому вхождению нетерминального символа такой цепочки. Если мы изменим лишь порядок применения правил вывода к вхождениям нетерминальных символов, а сами правила будем использовать те же, то, очевидно, получим другой вывод, но заключительная терминальная цепочка вывода и ее определяемая выводом синтаксическая структура останутся неизменными. Таким выводам будет соответствовать одно и то же дерево вывода.

Рассмотрим вывод, приведенный в примере 1, и изменим его таким образом, чтобы на каждом шаге правило применялось к самому левому вхождению нетерминального символа соответствующей цепочки. Тогда получим следующий вывод того же самого числа: $4, +H, +ДП, +БМП, +БЦМП, +ЦЦМП, +2ЦМП, +25МП, +25.БП, +25.ЦП, +25.6П, +25.6_{10}\Phi, +25.6_{10}-Б, +25.6_{10}-Ц, +25.6_{10}-2$.

Синтаксическое дерево вывода для данного вывода останется таким же, что и в примере 2.

Чтобы установить взаимно-однозначное соответствие между выводами и деревьями выводов, достаточно ограничить множество выводов КС-грамматики и рассматривать не все ее возможные выводы, а лишь так называемые левосторонние или правосторонние выводы.

Определение. Вывод в КС-грамматике будем называть *левосторонним* (*правосторонним*), если правила вывода применяются к самому левому (правому) вхождению нетерминального символа каждой цепочки вывода.

Очевидно следующее свойство левосторонних и правосторонних выводов: по любому выводу в произвольной КС-грамматике можно построить эквивалентный ему (т. е. порождающий ту же цепочку) левосторонний (правосторонний) вывод. Отсюда следует, что для любой цепочки языка существует левосторонний (правосторонний) вывод этой цепочки в порождающей язык КС-грамматике. Таким образом, изучая синтаксис языка и порождающую его КС-грамматику, мы можем ограничиться рассмотрением множества левосторонних или множества правосторонних выводов.

Нетрудно показать, что между синтаксическими деревьями выводов и левосторонними выводами некоторой грамматики существует взаимно-однозначное соответствие. Докажем, например, что двум различным левосторонним выводам в произвольной КС-грамматике соответствуют различные деревья выводов. Пусть имеются левосторонние выводы: $\Omega = \omega_0, \dots, \omega_m$ и $\Omega' = \omega'_0, \dots, \omega'_n$. Поскольку выводы различны, найдется такое i , $1 \leq i \leq \min(m, n)$, что $\omega_0 = \omega'_0, \dots, \omega_{i-1} = \omega'_{i-1}$, $\omega_i \neq \omega'_i$. А это означает, что к левому вхождению нетерминального символа цепочки ω_{i-1} в первом и втором выводах применяются разные правила, т. е. вместо данного вхождения в ω_{i-1} будут вставлены различные подцепочки. При построении деревьев, соответствующих нашим выводам, на i -м шаге мы в первом случае должны построить вершины, соответствующие первой вставляемой подцепочке, а во втором случае — соответствующие второй. Поэтому поддеревья, корни которых определены как вершины, поставленные в соответствие левому вхождению нетерминального символа в цепочке ω_{i-1} , будут различны, а тогда и деревья выводов для первого и второго выводов окажутся различными. Так же просто показать, что различным деревьям выводов соответствуют различные левосторонние выводы.

Возникает вопрос: всегда ли для цепочки языка, порождаемого некоторой КС-грамматикой, существу-

ет лишь один левосторонний вывод. Следующий пример показывает, что это не так.

Пример 4. Рассмотрим КС-грамматику со следующими правилами вывода:

$$\begin{array}{ll} I \rightarrow AB, & B \rightarrow c, \\ I \rightarrow CD, & C \rightarrow a, \\ A \rightarrow ab, & D \rightarrow bc. \end{array}$$

Язык, порождаемый этой грамматикой, состоит лишь из одной цепочки *abc*. Однако в грамматике существуют два левосторонних вывода этой цепочки:

$$\Omega_1 = I, AB, abB, abc$$

и

$$\Omega_2 = I, CD, aD, abc.$$

Введем теперь понятия, соответствующие свойству неоднозначности левосторонних выводов цепочек в некоторых КС-грамматиках.

Определения. 1) КС-грамматику, в которой существует более чем один левосторонний вывод некоторой терминальной цепочки, будем называть *неоднозначной*, а язык, порождаемый такой грамматикой, — *синтаксически неоднозначным*.

2) Язык, все порождающие КС-грамматики которого неоднозначны, будем называть *существенно синтаксически неоднозначным*.

Отметим следующее тривиальное свойство неоднозначных языков: для многих из них могут быть построены КС-грамматики, не являющиеся неоднозначными. Это утверждение, в частности, справедливо для любого конечного языка. Например, язык, определенный в примере 4 как синтаксически неоднозначный, порождается грамматикой, имеющей лишь одно правило вывода: $I \rightarrow abc$. Таким образом, не все синтаксически неоднозначные языки являются существенно синтаксически неоднозначными. Можно показать, что существуют и существенно синтаксически неоднозначные языки, однако доказательство этого факта довольно громоздко.

Неоднозначность грамматик является нежелательным свойством при описании синтаксиса языков программирования. Наличие нескольких левосторонних выводов, а, следовательно, и деревьев выводов одной программы означает неоднозначность ее синтаксиче-

ской структуры и затрудняет трансляцию на машинный язык. Тем не менее, неоднозначность КС-грамматик, описывающих синтаксис языков программирования, имеет место. Рассмотрим пример неоднозначного вывода в грамматике, представляющей синтаксис языка АЛГОЛ-60.

Пример 5. Рассмотрим оператор присваивания: $y := p(x)$. Этот оператор может быть порожден из нетерминального символа, соответствующего металингвистической переменной <оператор присваивания>, при выводе любой программы на АЛГОЛе-60, содержащей операторы присваивания. Приведем правила вывода, которые можно использовать при выводе данного оператора присваивания. Эти правила получены нами из нормальных форм Беккуса, причем некоторые несущественные в данном случае металингвистические переменные опущены.

- | | | | |
|--------------------------|--------------------------|-----------------------------|-----------------------|
| 1) $P \rightarrow LA$ | 4) $L \rightarrow Pr :=$ | 7) $A \rightarrow \Phi$ | 10) $I \rightarrow x$ |
| 2) $P \rightarrow LB$ | 5) $Pe \rightarrow I$ | 8) $B \rightarrow \Phi$ | 11) $I \rightarrow y$ |
| 3) $L \rightarrow Pe :=$ | 6) $Pr \rightarrow I$ | 9) $\Phi \rightarrow Pr(I)$ | 12) $I \rightarrow r$ |

Нетерминальные символы, используемые в этих правилах, соответствуют следующим металингвистическим переменным: P — оператор присваивания, L — левая часть, A — арифметическое выражение, B — логическое выражение, Pe — простая переменная, Pr — идентификатор процедуры, I — идентификатор, Φ — указатель функции.

Один из левосторонних выводов интересующего нас оператора получается при использовании последовательно правил 1), 3), 5), 11), 7), 9), 6), 12), 10) и выглядит следующим образом: $P, LA, Pe := A, I := A, y := A, y := \Phi, y := Pr(I), y := I(I), y := r(I), y := r(x)$.

Тот же самый оператор присваивания можно получить, используя на первом шаге вывода вместо правила 1) правило 2) и на пятом — вместо правила 7) правило 8). Кроме того, на втором и третьем шагах вместо правил 3) и 5) можно использовать правила 4) и 6). Таким образом, грамматика, описывающая синтаксис языка АЛГОЛ-60, допускает 4 различных левосторонних вывода приведенного оператора присваивания.

Неоднозначность грамматик, описывающих синтаксис языков программирования, связана с тем обстоятельством, что языки программирования на самом деле не являются КС-языками. При помощи КС-грамматик можно представить не все, а лишь часть синтаксических правил языков программирования. Те же правила, которые средствами КС-грамматик не описываются, называются *контекстными условиями* и задаются обычно неформально — при помощи английского, русского и других естественных языков.

Таким образом, язык, порождаемый, например, КС-грамматикой, описывающей синтаксис языка АЛГОЛ-60, шире АЛГОЛа-60. Он включает в себя не только правильные программы АЛГОЛа-60, но и такие программы, в которых не учтены контекстные условия этого языка. Контекстные условия языков программирования рассматриваются в гл. IV, а сейчас отметим только, что при построении правильной программы на АЛГОЛе-60, т. е. такой программы, в которой учтены контекстные условия, возможен лишь один вывод оператора, приведенного в примере 5. В самом деле, многозначность вывода этого оператора возникает потому, что, во-первых, идентификатор левой части оператора может представлять как переменную, так и процедуру-функцию, и, во-вторых, в правой части оператора может находиться как арифметическое, так и логическое выражение, причем в некоторых случаях они неразличимы вне контекста, в котором находятся. Однако в каждой правильной программе роль любого идентификатора определена однозначно и зависит от описания этого идентификатора. Таким образом, идентификатор в левой части оператора присваивания представляет процедуру-функцию, если этот оператор находится в теле процедуры-функции, а идентификатор совпадает с идентификатором данной процедуры функции. В противном случае идентификатор представляет переменную. Аналогично в правильных программах обрабатываются и правые части операторов присваивания.

Итак, мы показали, что при помощи КС-грамматик описываются неоднозначные языки, включающие в себя языки программирования. Синтаксическая неоднозначность языков устраняется путем учета и проверки контекстных условий.

Для дальнейшего изучения свойств КС-грамматик нам потребуется рассмотреть ряд бинарных отношений, которые можно задать на множествах символов КС-грамматик. *Бинарным отношением* R , заданным на множестве V , называется некоторое подмножество множества $V \times V$. Если α_1 и α_2 — элементы множества V и пара $(\alpha_1, \alpha_2) \in R$, то говорят, что для данной пары выполняется отношение R . Это утверждение записывается обычно в виде $\alpha_1 R \alpha_2$.

Рассмотрим некоторые операции над бинарными отношениями, считая, что все отношения, о которых будет идти речь, заданы на одном и том же множестве. Пусть имеются отношения R_1 и R_2 . Объединением $R_1 \cup R_2$ называется бинарное отношение, полученное в результате теоретико-множественного объединения R_1 и R_2 . Произведением $R_1 R_2$ называется бинарное отношение со следующим свойством: $\alpha_1 R_1 R_2 \alpha_2$ в том и только в том случае, если найдется такой элемент α , что $\alpha_1 R_1 \alpha$ и $\alpha R_2 \alpha_2$. Транзитивным замыканием \hat{R} отношения R называется бинарное отношение со следующим свойством: $\alpha_0 \hat{R} \alpha_n$ в том и только том случае, если найдется такая последовательность элементов $\alpha_0, \alpha_1, \dots, \alpha_n, n \geq 1$, что $\alpha_i R \alpha_{i+1}$ при $0 \leq i \leq n-1$. Ясно, что $R \subseteq \hat{R}$. Пусть E — так называемое отношение равенства: $\alpha_1 E \alpha_2$ в том и только том случае, если $\alpha_1 = \alpha_2$, т. е. α_1 и α_2 — один и тот же элемент. Транзитивно-рефлексивным замыканием R^* отношения R называется бинарное отношение, вычисленное по формуле $R^* = E \cup \hat{R}$. Заметим, что для некоторых отношений может оказаться, что $R^* = \hat{R}$.

Бинарное отношение на конечном множестве можно задать квадратной матрицей порядка n , где n — число элементов этого множества. Пусть элементы множества V некоторым образом перенумерованы числами от 1 до n , например, $V = \{\alpha_1, \alpha_2, \dots, \alpha_n\}$. Тогда отношение R на V можно задать матрицей $\|\sigma_{ij}\|$, в которой $\sigma_{ij} = 1$, если $\alpha_i R \alpha_j$, и $\sigma_{ij} = 0$, если для пары (α_i, α_j) отношение R не выполнено.

Упомянутые операции над бинарными отношениями, заданными на конечном перенумерованном множестве, выражаются с помощью операций над соответствующими им матрицами. Учитывая, что эти матрицы состоят только из нулей и единиц, введем так называемую булеву арифметику на множестве из нуля и единицы. В булевой арифметике будем пользоваться лишь двумя операциями — сложением и умножением, которые задаются следующими таблицами:

$0+0=0$	$0 \cdot 0=0$
$0+1=1$	$0 \cdot 1=0$
$1+0=1$	$1 \cdot 0=0$
$1+1=1$	$1 \cdot 1=1$

Нетрудно убедиться, что объединение и произведение бинарных отношений можно найти, выполняя, соответственно, сложение и умножение задающих их матриц по обычным формулам, в которых сложение и умножение рассматриваются как операции булевой арифметики. Можно также показать, что транзитивное замыкание \hat{R} отношения R находится по формуле $\hat{R} = R \cup R^2 \cup \dots \cup R^k$, где R^2 означает произведение RR , R^3 — произведение R^2R и т. д., а k не превосходит числа элементов множества, на котором задано R . Итак, все рассмотренные операции над бинарными отношениями, заданными на конечных множествах, выполняются эффективно, т. е. за конечное число шагов.

Определим некоторые бинарные отношения на множествах нетерминальных символов КС-грамматик. Пусть имеется КС-грамматика $G = \langle V_T, V_A, I, S \rangle$. Рассмотрим некоторую пару (A_1, A_2) нетерминальных символов этой грамматики.

О п р е д е л е н и я. 1) Будем считать, что для пары (A_1, A_2) в грамматике G выполняется отношение D_N , если в S содержится правило вывода $A_1 \rightarrow A_2$. В противном случае отношение D_N для данной пары не выполняется. Если $A_1 D_N A_2$, то говорят, что символ A_2 непосредственно выводим из символа A_1 без расширения.

2) Будем считать, что для пары (A_1, A_2) в грамматике G выполняется отношение D_W , если в S содержится правило вывода вида $A_1 \rightarrow \xi_1 A_2 \xi_2$, где ξ_1 и $\xi_2 \in (V_T \cup V_A)^*$ и хотя бы одна из подцепочек ξ_1 и ξ_2 — не пуста. В противном случае отношение D_W для данной пары не выполняется. Если $A_1 D_W A_2$, то говорят, что символ A_2 непосредственно выводим из символа A_1 с расширением.

3) Обозначим объединение $D_N \cup D_W$ через D . Если $A_1 D A_2$, то говорят, что символ A_2 непосредственно выводим из символа A_1 .

Непосредственную выводимость символов не следует смешивать с определенной ранее непосредственной выводимостью цепочек. Последняя тоже может быть представлена как бинарное отношение, заданное на $(V_T \cup V_A)^*$. Однако, поскольку это множество бесконечно, такая интерпретация непосредственной выводимости цепочек нами не используется.

Отметим следующие очевидные свойства введенных бинарных отношений. Цепочка, состоящая из одного символа A_2 , выводится в грамматике G из A_1 в том и только том случае, если $A_1 \hat{D}_N A_2$. Цепочка, в которую входит символ A_2 , выводится в грамматике G из A_1 в том и только том случае, если $A_1 \hat{D} A_2$.

В дальнейшем нам понадобятся некоторые другие бинарные отношения, которые задаются на множествах $V_T \cup V_A$ КС-грамматик.

Перейдем теперь ко второму кругу вопросов, интересующих нас в связи с изучением КС-грамматик. Рассмотрим условия, при которых КС-грамматики порождают бесконечные языки. Как уже отмечалось, при представлении синтаксиса языков программирования нормальными формами Бекуса некоторые металингвистические переменные используются рекурсивно. Мы покажем, что рекурсивный характер задания синтаксиса является необходимым для того, чтобы обеспечить бесконечность соответствующих языков.

Прежде чем сформулировать необходимые и достаточные условия порождения КС-грамматиками бесконечных языков, рассмотрим ряд определений и вспомогательных утверждений. В первую очередь нас будут интересовать так называемые существенные нетерминальные символы, т. е. такие символы, которые могут применяться в выводах цепочек, входящих в порождаемые КС-грамматиками языки. Очевидно, каждый существенный символ некоторой грамматики обладает следующими двумя свойствами: во-первых, должен быть возможен вывод из начального символа цепочки, содержащей данный символ, и, во-вторых, из данного символа должна быть выводима хотя бы одна терминальная цепочка. Перейдем к точным определениям.

Определения. 1) Нетерминальный символ КС-грамматики называется *выводимым*, если он является начальным символом или существует вывод из начального символа цепочки, содержащей вхождение данного символа.

2) Нетерминальный символ КС-грамматики называется *производящим*, если существует вывод из этого символа терминальной цепочки.

(3) Нетерминальный символ КС-грамматики называется *существенным*, если он — выводимый и производящий.

4) Символы, не являющиеся выводимыми, производящими или существенными, будем называть, соответственно, *невыводимыми*, *непроизводящими*, и *несущественными*.

По определению начальный символ считается выводимым для любой грамматики. Принадлежность же остальных нетерминальных символов к выводимым и всех к производящим не является очевидной. Для того, чтобы свойство существенности можно было использовать, должен существовать *эффективный*, т. е. выполняемый за конечное число шагов алгоритм, выделяющий существенные символы из множества нетерминальных символов произвольной КС-грамматики. Этот алгоритм будет указан в следующей лемме.

Лемма 1. Существует эффективный алгоритм, выделяющий подмножество существенных символов из множества нетерминальных символов произвольной КС-грамматики.

Для доказательства этой леммы нужно указать два алгоритма, выделяющие выводимые и производящие символы. Алгоритм выделения существенных символов будет тогда заключаться в последовательном выполнении этих алгоритмов и в нахождении пересечения подмножеств выводимых и производящих символов.

Алгоритм выделения выводимых символов очевиден: символ A является выводимым в том и только в том случае, если ID^*A .

Рассмотрим алгоритм выделения производящих символов. Его можно представить в виде циклического процесса, с помощью которого находится последовательность подмножеств нетерминальных символов W_1, W_2, \dots, W_k , причем W_k содержит все производящие символы анализируемой грамматики. Включим в W_1 все такие символы, для которых в данной грамматике имеются правила вывода с терминальными цепочками в правых частях. Очевидно, все символы из W_1 — производящие. Пусть уже построены подмножества последовательности до W_i включительно и показано, что все символы из W_i — производящие. В W_{i+1} включим все символы из W_i , а также все такие символы, для

которых имеются правила вывода с правыми частями, состоящими только из терминальных символов и символов из W_i . Ясно, что в W_{i+1} будут включены только производящие символы: если $A \in W_{i+1} \setminus W_i$, то вывод из него терминальную цепочку, применим на первом шаге правило, на основании которого он был включен в W_{i+1} . Процесс построения последовательности подмножеств заканчивается, когда для некоторого k окажется, что $W_k = W_{k+1}$. При этом в W_k будут включены все производящие символы. Действительно, пусть в анализируемой грамматике имеется вывод $\omega_0, \omega_1, \dots, \omega_m$, где $\omega_0 = A$, ω_m — терминальная цепочка. Ясно, что в ω_{m-1} могут входить лишь нетерминальные символы из W_1 , в ω_{m-2} — из W_2 и т. д. Если $m > k$, то в ω_{m-k-1} могут входить лишь нетерминальные символы из W_{k+1} . Но $W_{k+1} = W_k$, поэтому во все ω_l при $0 \leq l \leq m-k-1$ могут входить лишь нетерминальные символы из этого подмножества, а тогда $A \in W_k$.

Так как при $1 \leq i \leq k-1$, $W_{i+1} \setminus W_i \neq \emptyset$, k не превосходит числа нетерминальных символов грамматики. Следовательно, процесс выделения производящих символов выполняется за конечное число шагов. Лемма доказана.

В связи с тем, что несущественные символы не могут использоваться в выводах терминальных цепочек из начального символа, представляют интерес в основном те грамматики, все нетерминальные символы которых существенны.

Определение. КС-грамматика, все нетерминальные символы которой существенны, называется *приведенной*.

По любой КС-грамматике можно построить эквивалентную ей приведенную грамматику. Для этого нужно выделить подмножество существенных символов этой грамматики и подмножество всех таких ее правил вывода, в которых встречаются лишь терминальные и существенные нетерминальные символы. Грамматика, в которую включены лишь выделенные символы и правила, будет приведенной и эквивалентной исходной.

Формально задаваемый синтаксис языков программирования, как правило, представляется приведенными КС-грамматиками. Так как несущественные сим-

волы не используются в выводах терминальных цепочек, нет смысла вводить соответствующие таким символам синтаксические конструкции. Исключение составляет язык АЛГОЛ-68. Для описания синтаксиса этого языка используется особая грамматика, которая называется грамматикой ван Вейнгаардена. В гл. IV мы покажем, что эта грамматика сводится к грамматике, имеющей бесконечное число нетерминальных символов и бесконечное число правил вывода контекстно-свободного вида.

При использовании в выводах некоторых нетерминальных символов их завершение оказывается невозможным. Такие символы являются непроеизводящими, а выводы, в которых они используются, часто называются *тупиковыми*.

Способность КС-грамматик порождать бесконечные языки тесно связана с наличием в них так называемых циклических символов.

Определение 1) *Нетерминальный символ КС-грамматики называется циклическим*, если в ней существует вывод из этого символа цепочки, содержащей вхождение данного символа. В противном случае символ называется нециклическим.

2) *КС-грамматика называется циклической*, если в ней имеется хотя бы один циклический символ. В противном случае грамматика называется нециклической.

3) *Вывод в КС-грамматике называется циклическим*, если в его цепочках найдутся два таких вхождения некоторого нетерминального символа, которые являются предком и потомком друг друга. В противном случае вывод называется нециклическим.

Из этих определений непосредственно следует, что циклические выводы возможны лишь в циклических грамматиках. Так как символ A является циклическим в том и только том случае, если $A\bar{D}A$, можно считать доказанным следующее утверждение:

Лемма 2. *Существует эффективный алгоритм выделения подмножества циклических символов произвольной КС-грамматики.*

Рассмотрим следующую очень важную лемму:

Лемма 3. *Нециклическая КС-грамматика порождает конечный язык.*

Доказательство. Пусть имеется некоторая нециклическая КС-грамматика. Обозначим через n число ее нетерминальных символов, а через m — максимальную длину правых частей ее правил вывода. Для доказательства конечности языка, порождаемого данной грамматикой, достаточно показать конечность в ней множества синтаксических деревьев выводов. Нетрудно видеть, что высота любого дерева вывода в исследуемой грамматике не превосходит n . Если бы, напротив, нашлось дерево с большей высотой, в нем существовал бы путь длины, большей чем n , а тогда на этом пути нашлись бы две вершины, помеченные одним и тем же нетерминальным символом. Следовательно, соответствующий этому дереву вывод был бы циклическим, что в случае нециклической грамматики невозможно.

Число вершин i -го уровня любого дерева нашей грамматики не превосходит m^i , и поэтому общее число его вершин меньше, чем m^{n+1} (при $m \geq 2$) или $n+2$ (при $m=1$). Таким образом, в данной грамматике можно построить лишь конечное число деревьев выводов, и следовательно, лемма доказана.

Лемма 3 позволяет сформулировать необходимое условие порождения КС-грамматиками бесконечных языков: бесконечный язык может порождаться только циклической грамматикой. Однако условие цикличности грамматики не является достаточным. Прежде всего все циклические символы грамматики могут быть несущественными, и тогда выводы цепочек языка будут в такой грамматике нециклическими. Кроме того, циклическая грамматика может порождать конечный язык, если длина правых частей некоторых ее правил равна 1. Рассмотрим следующий пример.

Пример 6. Рассмотрим грамматику, множество S которой состоит из следующих правил:

$$\begin{aligned} I &\rightarrow AB, & B &\rightarrow b, \\ A &\rightarrow C, & C &\rightarrow A. \\ A &\rightarrow a, \end{aligned}$$

Язык, порождаемый этой грамматикой, состоит из единственной цепочки ab . Тем не менее, данная грамматика — циклическая, так как в ней имеются два циклических символа: A и C ,

Приведенный пример показывает, что для получения достаточного условия порождения КС-грамматиками бесконечных языков нужно различать две разновидности циклических символов.

Определение. Циклический символ КС-грамматики будем называть *эффективным*, если в ней существует вывод из этого символа цепочки длиной больше 1, содержащей вхождение данного символа. В противном случае циклический символ будем называть *фиктивным*.

Нетрудно показать, что грамматика, все циклические символы которой фиктивные, порождает конечный язык. Очевидно, все неповторные выводы такой грамматики — нециклические. Следовательно, число неповторных выводов, порождающих цепочки языка, конечно, и язык, порождаемый грамматикой, конечен.

Как определить, имеет ли циклическая грамматика эффективные циклические символы? Ответ на этот вопрос дает следующая лемма.

Лемма 4. *Существует эффективный алгоритм выделения эффективных циклических символов произвольной КС-грамматики.*

Доказательство. Если символ A — эффективный циклический, то в грамматике должен существовать такой вывод из A цепочки, содержащей вхождение данного символа, хотя бы на одном шаге которого (возможно, на первом или последнем) применяется правило вывода с правой частью длиной больше 1. A тогда, очевидно, $AD_N^*D_W D^*A$. Лемма доказана.

Если в грамматике нет правил вывода вида $A \rightarrow B$, то выделение множества ее эффективных циклических символов существенно упрощается: очевидно, каждый ее циклический символ является эффективным. Поэтому полезно показать, что такие правила можно из грамматик исключать.

Лемма 5. *По любой КС-грамматике можно построить эквивалентную ей грамматику без правил вывода вида $A \rightarrow B$.*

Доказательство. Пусть имеется КС-грамматика $G = \langle V_T, V_A, A_1, S \rangle$, где $V_A = \{A_1, A_2, \dots, A_n\}$, и в S входят правила указанного вида. Разобьем правила вывода из S на два непересекающихся подмножества: $S = S_1 \cup S_2$, включив в S_1 все правила вида $A_i \rightarrow$

$\rightarrow A_j$, где $A_i, A_j \in V_A$, и в S_2 — все остальные правила. Определим для каждого $A_i \in V_A$ множество правил $S(A_i)$, включив в него все такие правила $A_i \rightarrow \eta$, что $A_j \rightarrow \eta \in S_2$ и $A_i \hat{D}_N A_j$. Образует грамматику $G' = \langle V_T, V_A, A_1, S' \rangle$, в которой множества терминальных и нетерминальных символов и начальный символ совпадают с соответствующими объектами грамматики G , а $S' = S_2 \cup \bigcup_{i=1}^n S(A_i)$.

Грамматика G' не содержит правил вывода вида $A_i \rightarrow A_j$. Покажем, что она эквивалентна исходной. Пусть $\Omega = \omega_0, \dots, \omega_m$ — левосторонний вывод в G цепочки из $L(G)$, причем ω_k — самая левая цепочка, к которой применяется правило из S_1 . Тогда в Ω имеются цепочки вида: $\omega_k = \xi_1 A_i \xi_2$, $\omega_l = \xi_1 A_j \xi_2$, $\omega_{l+1} = \xi_1 \eta \xi_2$, $l > k$, причем $A_i \hat{D}_N A_j$, $A_j \rightarrow \eta \in S_2$ и, следовательно, $A_i \rightarrow \eta \in S'$. Преобразуем вывод Ω в $\Omega' = \omega_0, \dots, \omega_k, \omega_{l+1}, \dots, \omega_m$. Найдем теперь в Ω_1 левую цепочку, к которой применяется правило из S_1 , и преобразуем Ω_1 в Ω_2 аналогичным образом. Продолжая процедуру преобразования, через конечное число шагов получим вывод той же цепочки, в котором применяются лишь правила из S' . Обратная процедура построения по выводу в G' эквивалентного вывода в G очевидна. Лемма доказана.

В грамматиках, описывающих синтаксис языков программирования, правила вида $A \rightarrow B$ применяются довольно широко. Во многих случаях введение таких правил позволяет облегчить семантическую интерпретацию синтаксических понятий, а также упростить их формальное описание. Наиболее характерным примером являются нормальные формы Бекуса, задающие синтаксис арифметических и логических выражений языка АЛГОЛ-60. Из форм, описывающих арифметические выражения, можно, в частности, получить следующие правила:

$\langle \text{арифметическое выражение} \rangle ::= \langle \text{простое арифметическое выражение} \rangle$

$\langle \text{простое арифметическое выражение} \rangle ::= \langle \text{терм} \rangle$

$\langle \text{терм} \rangle ::= \langle \text{множитель} \rangle$

$\langle \text{множитель} \rangle ::= \langle \text{первичное выражение} \rangle$

$\langle \text{первичное выражение} \rangle ::= \langle \text{число без знака} \rangle$

$\langle \text{первичное выражение} \rangle ::= \langle \text{переменная} \rangle$

$\langle \text{первичное выражение} \rangle ::= \langle \text{указатель функции} \rangle$

$\langle \text{переменная} \rangle ::= \langle \text{простая переменная} \rangle$

$\langle \text{переменная} \rangle ::= \langle \text{переменная с индексами} \rangle$.

Такие понятия, как «простое арифметическое выражение», «терм», «множитель», «первичное выражение», позволяют представить синтаксис арифметических выражений в форме, неявно указывающей порядок выполнения действий при вычислении значений этих выражений: для того, чтобы вычислить значение арифметического выражения, нужно найти значение одного из входящих в него простых арифметических выражений; вычисляя простое арифметическое выражение, следует найти значения всех входящих в него термов и т. д. Однако арифметическое выражение может представлять собой и отдельное простое арифметическое выражение, последнее может быть представлено термом, множителем или первичным выражением. Это и приводит к необходимости включения в грамматику, порождающую арифметические выражения, соответствующих правил вывода. Далее, поскольку числа без знака, переменные и указатели функций в арифметических выражениях взаимозаменяемы, использование объединяющего их (а также арифметических выражений в скобках) понятия «первичное выражение» позволяет значительно упростить правила вывода для множителей. В свою очередь, простые переменные и переменные с индексами также взаимозаменяемы в большинстве конструкций, поэтому введение в синтаксис объединяющего их понятия «переменная» тоже упрощает некоторые правила вывода.

Хотя, как мы показали, правила вывода вида $A \rightarrow \rightarrow B$ используются в грамматиках для языков программирования довольно часто, все циклические символы этих грамматик — эффективные. Выводимость вида $A \Rightarrow A$ не имеет, по-видимому, для языков программирования никакой разумной интерпретации и поэтому не имеет места в грамматиках, описывающих эти языки.

Перейдем к формулировке достаточного условия порождения КС-грамматиками бесконечных языков.

Л е м м а 6. Язык, порождаемый циклической приведенной КС-грамматикой, содержащей хотя бы один эффективный циклический символ, бесконечен.

Доказательство этой леммы не представляет труда. Пусть символ A приведенной грамматики — эффективный, циклический. Поскольку грамматика приведенная, в ней возможен вывод некоторой цепочки, содержащей этот символ, из начального символа:

$$I \xRightarrow{*} \xi_1 A \xi_2; \quad \xi_1, \xi_2 \in (V_T \cup V_A)^*.$$

Применим правила для вывода из A цепочки, содержащей A :

$$\xi_1 A \xi_2 \xRightarrow{*} \xi_1 \xi_3 A \xi_4 \xi_2; \quad \xi_3, \xi_4 \in (V_T \cup V_A)^*;$$

$$\max(l(\xi_3), l(\xi_4)) \geq 1.$$

Применим также правила еще $n-1$ раз:

$$\xi_1 \xi_3 A \xi_4 \xi_2 \xRightarrow{*} \xi_1 \xi_3^n A \xi_4^n \xi_2 \quad (\xi^n \text{ означает } n \text{ вхождений } \xi).$$

Используя приведенность грамматики, применим правила, заменяющие все вхождения нетерминальных символов их терминальными порождениями:

$$\xi_1 \xi_3^n A \xi_4^n \xi_2 \xRightarrow{*} T(\xi_1 \xi_3^n A \xi_4^n \xi_2),$$

где $T(\xi)$ означает такую терминальную цепочку, что

$$\xi \xRightarrow{*} T(\xi).$$

В связи с тем, что n может быть в данной схеме вывода любым целым числом, бесконечность языка, порождаемого грамматикой, доказана.

Рассмотренные нами свойства КС-грамматик позволяют сформулировать следующую теорему:

Теорема 2. Для того, чтобы приведенная КС-грамматика порождала бесконечный язык, необходимо и достаточно, чтобы она была циклической и содержала хотя бы один эффективный циклический символ.

Эффективные циклические символы грамматики, задающей синтаксис языка АЛГОЛ-60, соответствуют

таким конструкциям, как «блок», «оператор», «идентификатор» и пр. Многие другие символы хотя и не являются циклическими, но допускают циклические выводы порождаемых из них терминальных цепочек. Это символы, соответствующие конструкциям «программа», «оператор присваивания», «простая переменная» и др. Исключение составляют нетерминальные символы, задающие разбиение на группы основных символов языка. Выводы из этих нетерминальных символов могут быть лишь нециклическими. В число таких символов входят, например, символы, соответствующие понятиям «буква», «цифра», «ограничитель» и т. д.

Перейдем к вопросу о связи укорачивающих контекстно-свободных грамматик (УКС-грамматик) с КС-грамматиками. Напомним, что укорачивающими контекстно-свободными называются такие грамматики, правила вывода которых имеют вид $A \rightarrow \eta$, где A — нетерминальный символ, η — цепочка над объединением терминального и нетерминального словарей и $l(\eta) \geq 0$. Из определения следует, что любая КС-грамматика является и УКС-грамматикой. Любому выводу в УКС-грамматике можно поставить в однозначное соответствие синтаксическое дерево вывода, которое строится так же, как и в случае неукорачивающих КС-грамматик. Однако, если на некоторых шагах вывода к вхождению нетерминальных символов в цепочки применяются укорачивающие правила, то эти вхождения не имеют в данном выводе потомков. Поэтому такие шаги не определяют никаких действий процедуры построения дерева, а вершины, соответствующие указанным вхождениям, окажутся заключительными. Эта особенность деревьев выводов в УКС-грамматиках будет нами учтена ниже при доказательстве теоремы 3.

Изучение свойств УКС-грамматик интересно потому, что, если говорить строго, именно они используются для описания синтаксиса языков программирования. Укорачивающие правила применяются в тех случаях, когда описываются синтаксические конструкции языков программирования, которые включаются в программы не обязательно, а по желанию программиста или в зависимости от алгоритма решения

задачи. Типичным примером могут служить следующие конструкции, используемые в описаниях процедур АЛГОЛа-60: «совокупность формальных параметров», «список значений», «совокупность спецификаций». Все эти конструкции могут включаться в описание конкретной процедуры, но не являются обязательными. Так, например, в процедуре может не быть формальных параметров, вызываемых значениями, и тогда список значений в заголовке такой процедуры отсутствует. Использование укорачивающих правил позволяет в этих случаях упростить синтаксис и сделать его более легким и понятным для изучения. Как правило, не представляет никакого труда так изменить соответствующие правила, чтобы все они оказались неукорачивающими. Рассмотрим подробнее синтаксис описаний процедур АЛГОЛа-60.

Пример 7. Грамматика языка АЛГОЛ-60 включает в себя следующие правила, полученные из металингвистических формул, задающих описание процедур:

- | | |
|---------------------------------|---|
| 1) $З \rightarrow ПС_1; C_2C_3$ | 8) $C_2 \rightarrow \text{value } C_5;$ |
| 2) $П \rightarrow И$ | 9) $C_5 \rightarrow И$ |
| 3) $C_1 \rightarrow (C_4)$ | 10) $C_5 \rightarrow C_5, И$ |
| 4) $C_1 \rightarrow \lambda$ | 11) $C_2 \rightarrow \lambda$ |
| 5) $C_4 \rightarrow \Phi$ | 12) $C_3 \rightarrow C_6C_5;$ |
| 6) $C_4 \rightarrow C_4O\Phi$ | 13) $C_3 \rightarrow C_3C_6C_5;$ |
| 7) $\Phi \rightarrow И$ | 14) $C_3 \rightarrow \lambda$ |

В этих правилах используются следующие обозначения: $З$ — заголовок процедуры, $П$ — идентификатор процедуры, C_1 — совокупность формальных параметров, C_2 — список значений, C_3 — совокупность спецификаций, C_4 — список формальных параметров, Φ — формальный параметр, O — ограничитель параметра, C_5 — список идентификаторов, $И$ — идентификатор, C_6 — спецификация.

Укорачивающими являются правила 4), 11) и 14). Они введены для того, чтобы показать, что конструкции, соответствующие нетерминальным символам C_1 , C_2 и C_3 , могут в заголовке процедуры отсутствовать. Эту особенность синтаксиса процедур можно учесть иначе — перебором всех возможных вариантов структуры заголовка и включением в грамматику варианта правила 1) для каждого из них. Это позволит исключить из грамматики укорачивающие правила. Если формально выполнить эти действия, в грамматику для заголовка процедуры нужно будет включить следующие правила:

- | | |
|-------------------------------|----------------------------|
| 1а) $З \rightarrow ПС_1; C_2$ | 1д) $З \rightarrow П; C_2$ |
| 1б) $З \rightarrow ПС_1; C_3$ | 1е) $З \rightarrow П; C_3$ |
| 1в) $З \rightarrow ПС_1;$ | 1ж) $З \rightarrow П;$ |
| 1г) $З \rightarrow П; C_2C_3$ | |

Правила 4), 11) и 14) могут быть из грамматики исключены. Полученная грамматика — неукорачивающая и эквивалентна исходной. Учитывая, что при отсутствии формальных параметров в заголовке процедуры не может быть списка значений и совокупности спецификаций, а также правило, согласно которому все параметры, вызываемые значениями, должны быть специфицированы, мы можем исключить из грамматики правила 1а), 1г), 1д) и 1е). Полученная таким образом грамматика будет не только правильно, но даже более точно, чем исходная, описывать правила написания заголовков процедур. Однако она несколько более громоздка и менее наглядна и поэтому не используется в описании эталонного языка.

Хотя, как показывает пример, укорачивающие правила не играют существенной роли в описании языков программирования, их использование приводит к необходимости рассмотреть следующие вопросы: являются ли языки, порождаемые УКС-грамматиками, распознаваемыми, и в каком отношении находятся классы укорачивающих и неукорачивающих КС-грамматик. Ответы на эти вопросы дает следующая теорема.

Теорема 3. По любой УКС-грамматике может быть построена почти эквивалентная ей КС-грамматика.

Доказательство. Вспомним определение почти эквивалентности. Грамматика G_1 называется почти эквивалентной грамматике G_2 , если $L(G_1) = L(G_2) \setminus \{\lambda\}$, где λ — пустая цепочка. Пусть задана УКС-грамматика $G = \langle V_T, V_A, I, S \rangle$. Построим КС-грамматику $G' = \langle V_T, V_A, I, S' \rangle$, почти эквивалентную исходной.

Алгоритм построения похож на тот, который рассматривался в примере 7. Первый этап построения заключается в выделении из V_A подмножества укорачивающих символов, т. е. таких, из которых в G возможен вывод пустой цепочки. Выделение укорачивающих символов производится при помощи процедуры, аналогичной той, которая использовалась для выделения производящих символов (см. доказательство леммы 1). При этом в подмножество W_1 включаются все такие нетерминальные символы, для которых в грамматике G имеются укорачивающие правила, а при определении следующих подмножеств последовательности учитываются лишь правила вывода, не содержащие в правых частях терминальных символов.

Перейдем к построению грамматики G' . Рассмотрим подмножество неукорачивающих правил вывода исходной грамматики G :

$$S_N = \{R_i\}; i = 1, 2, \dots, n; S_N \subset S.$$

По каждому правилу $R_i \in S_N$ образуем конечное множество правил $S(R_i)$ следующим образом. Если в правую часть R_i не входят укорачивающие символы, то $S(R_i) = \{R_i\}$. В противном случае включим в $S(R_i)$ правило R_i , а также все такие неукорачивающие правила, которые получаются вычеркиванием из правой части R_i одного или нескольких вхождений укорачивающих символов. Например, для $R = A \rightarrow aBCd$, где B и C — укорачивающие, $S(R) = \{A \rightarrow aBCd, A \rightarrow aBd, A \rightarrow aCd, A \rightarrow ad\}$. R_i назовем главным правилом множества $S(R_i)$. Теперь определим грамматику G' . Множества терминальных и нетерминальных символов и начальный символ этой грамматики будем считать совпадающими с соответствующими объектами исходной грамматики G , а множество правил вывода образуем по следующей формуле: $S' = \bigcup_{i=1}^n S(R_i)$.

Грамматика G' , очевидно, неукорачивающая. Покажем ее почти эквивалентность исходной. Пусть $\Omega = \omega_0, \dots, \omega_m$ — вывод в G непустой цепочки, входящей в $L(G)$. Если в Ω не применяются укорачивающие правила, Ω является также и выводом в G' . Предположим, что укорачивающие правила в выводе Ω применяются. Построим соответствующее ему синтаксическое дерево вывода. В этом дереве должны найтись заключительные вершины, помеченные укорачивающими нетерминальными символами. Преобразуем дерево таким образом, чтобы в нем подобных вершин не было, и покажем, что после этого преобразования дерево будет представлять вывод терминальной цепочки в грамматике G' . Пусть x — одна из заключительных вершин, помеченная нетерминальным символом грамматики G . Найдем ближайшую к x предшествующую ей вершину y , из которой выходит более чем одна дуга. Такая вершина должна найтись, так как помимо x в дереве имеется по крайней мере еще одна заключительная вершина, помеченная терминаль-

ным символом, а значит, и путь, ведущий из корня в эту вершину. Рассмотрим путь из y в x . Если длина его больше чем 1, то каждая следующая за y и находящаяся на этом пути незаключительная вершина имеет ровно одну выходящую дугу и помечена некоторым укорачивающим символом грамматики G . Исключим из дерева все такие вершины, если они имеются, а также вершину x вместе с входящими в эти вершины дугами. Так как из всех исключенных вершин выходит не более одной дуги, вновь получим дерево. Число заключительных вершин, помеченных нетерминальными символами, будет в этом дереве на единицу меньше, чем в исходном. Выполняя описанную процедуру для каждой из таких вершин последовательно, получим дерево, все заключительные вершины которого помечены терминальными символами. Это дерево представляет вывод цепочки ω_m , в котором используются неукорачивающие правила грамматики G и правила, полученные из неукорачивающих правил этой грамматики исключением из правых частей некоторых вхождений укорачивающих символов. Все такие правила входят в множество S' грамматики G' .

Итак, мы показали, что $L(G) \setminus \{\lambda\} \subseteq L(G')$. Докажем обратное включение: $L(G') \subseteq L(G) \setminus \{\lambda\}$, что завершит доказательство теоремы. Пусть Ω — вывод в G' цепочки из $L(G')$. Если все правила, применяемые в Ω , — главные, Ω является также выводом в G . Предположим, что ω_i — первая цепочка, к которой применяется правило $R' = A \rightarrow \eta \in S(R_j)$, причем $R' \neq R_j$. Тогда

$$\omega_i = \xi_1 A \xi_2, \quad \omega_{i+1} = \xi_1 \eta \xi_2.$$

Рассмотрим главное правило данного множества: $R_j = A \rightarrow \xi \in S$, где ξ представляет собой цепочку η , в которую в некоторых позициях вписаны укорачивающие символы грамматики G . Применим R_j к ω_i и получим цепочку $\tilde{\omega}_{i+1} = \xi_1 \xi \xi_2$. Поскольку в G существует вывод $\xi \Rightarrow \eta$, то существует и вывод $\tilde{\omega}_{i+1} \Rightarrow \omega_{i+1}$. Заменим левую часть вывода $\Omega: \omega_0, \dots, \omega_i, \omega_{i+1}$ выводом $\omega_0, \dots, \omega_i, \tilde{\omega}_{i+1}, \dots, \omega_{i+1}$. Данный вывод является выводом в G . Рассмотрим правую часть вывода $\Omega: \omega_{i+1}, \dots, \omega_m$. Если в ней найдется цепочка, к которой

применяется правило, не включенное в S , выполним еще раз рассмотренное преобразование. Таким образом вывод в G' любой цепочки из $L(G')$ может быть преобразован в вывод этой цепочки в исходной грамматике G . Доказательство теоремы закончено.

Очевидным следствием теоремы является утверждение, что УКС-языки легко распознаваемы. Построив КС-грамматику, почти эквивалентную грамматике, порождающей УКС-язык, мы решаем вопрос распознавания для любой непустой цепочки. Для того, чтобы определить, входит ли в УКС-язык пустая цепочка, достаточно проверить, является ли начальный символ соответствующей УКС-грамматики укорачивающим, что можно сделать при помощи процедуры, описанной в доказательстве теоремы 3.

Аналогично случаю КС-грамматик можно определить укорачивающие НС-грамматики. Интересно, что теорема, аналогичная теореме 3, для них неверна. Можно показать, что по грамматике произвольного вида может быть построена эквивалентная укорачивающая НС-грамматика, и, следовательно, класс языков, порождаемых укорачивающими НС-грамматиками, совпадает с классом рекурсивно-перечислимых множеств цепочек символов.

На этом мы заканчиваем изучение свойств КС-грамматик общего вида. В следующем параграфе нас будет интересовать специальный класс КС-грамматик с правилами частного вида. В главах II и III мы еще вернемся к КС-грамматикам в связи с рассмотрением алгоритмов распознавания порождаемых ими языков.

§ 4. Автоматные грамматики

Если внимательно просмотреть грамматику, описывающую язык АЛГОЛ-60, можно заметить, что среди ее правил много таких, которые имеют более простой вид, нежели правила КС-грамматик общего вида. Характерной особенностью таких правил является то, что в их правые части либо вообще не входят нетерминальные символы, либо входит лишь один такой символ, занимающий крайнюю левую или правую позицию в цепочке. Некоторые синтаксические понятия языка хотя и не задаются правилами указанного

вида, но их легко к этому виду преобразовать. К числу таких понятий относятся, например, «идентификатор» и все понятия, представляемые идентификаторами (см. пример 2 § 2), а также «число» и связанные с ним понятия.

Пример 1. Рассмотрим приведенную в § 2 грамматику для чисел языка АЛГОЛ-60. Три правила этой грамматики не имеют интересующего нас простого вида: $B \rightarrow BЦ$, $D \rightarrow BM$ и $H \rightarrow DP$. Напомним, что здесь B означает целое без знака, $Ц$ — цифру, D — десятичное число, M — правильную дробь, H — число без знака и P — порядок.

Первое из этих правил можно рассматривать как схему, задающую десять правил с правыми частями, включающими один нетерминальный символ: $B \rightarrow B0$, $B \rightarrow B1$, ..., $B \rightarrow B9$. Второе и третье правила можно заменить группами правил простого вида. Например, десятичные числа задаются следующими правилами:

- 1) $D \rightarrow Ч_1$ 4) $Ч_2 \rightarrow ЦЧ_2$
- 2) $Ч_1 \rightarrow ЦЧ_1$ 5) $Ч_2 \rightarrow Ц$
- 3) $Ч_1 \rightarrow Ц.Ч_2$

Здесь правила 2)–5) можно рассматривать как схемы правил аналогично правилу $B \rightarrow BЦ$.

Определим классы грамматик, включающих в себя правила указанного вида.

О п р е д е л е н и я. 1) Правило вывода порождающей грамматики будем называть *заклучительным*, если оно имеет вид: $A \rightarrow x$, где A — нетерминальный символ, а x — нетерминальная цепочка.

2) Правило вывода порождающей грамматики будем называть *праволинейным (леволинейным)*, если оно имеет вид: $A \rightarrow xB$ ($A \rightarrow Bx$), где A и B — нетерминальные символы, а x — терминальная (возможно, пустая) цепочка.

3) Неукорачивающую КС-грамматику будем называть *праволинейной (леволинейной)*, если все ее правила — праволинейные (леволинейные) или заключительные.

Итак, мы определили два класса КС-грамматик простого вида. Оба эти класса определяют одно и то же множество языков. Это свойство будет нами рассмотрено позже. А сейчас покажем, что праволинейные грамматики эквивалентны автоматным.

Лемма 7. По любой праволинейной грамматике может быть построена эквивалентная ей автоматная

грамматика. По любой грамматике, включающей лишь правила вида $A \rightarrow \lambda$, а также праволинейные и заключительные правила, может быть построена почти эквивалентная автоматная грамматика.

Доказательство. Для того, чтобы построить А-грамматiku, эквивалентную произвольной праволинейной грамматике без правил вида $A \rightarrow B$, которые мы можем исключить по лемме 5, нужно каждое правило вида $A \rightarrow xB$ или $A \rightarrow x$, где $l(x) \geq 2$, заменить правилами, допустимыми при задании А-грамматик. Это можно сделать следующим образом. Пусть имеется правило $A \rightarrow a_1 a_2 \dots a_n B$, $a_i \in V_T$. Выберем $n-1$ новый нетерминальный символ: A_1, A_2, \dots, A_{n-1} , $A_i \notin V_A$, и заменим данное правило группой правил:

$$A \rightarrow a_1 A_1, A_1 \rightarrow a_2 A_2, \dots, A_{n-1} \rightarrow a_n B.$$

Если исходное правило имело вид $A \rightarrow a_1 a_2 \dots a_n$, то последнее правило группы будет несколько иным: $A_{n-1} \rightarrow a_n$. Таким способом можно исключить из S все правила, у которых длина терминальных подцепочек в правых частях больше или равна двум. При этом для каждой группы включаемых в S правил нужно выбирать свои новые нетерминальные символы. Грамматика, в множество нетерминальных символов которой включены все новые символы, а множество правил вывода преобразовано описанным способом, является автоматной и эквивалентной исходной.

Если в исходной грамматике имелись правила вида $A \rightarrow \lambda$, то по теореме 3 построим грамматiku, почти эквивалентную исходной. Эта грамматика будет праволинейной. Затем, используя описанный выше алгоритм, преобразуем праволинейную грамматiku в эквивалентную автоматную. Полученная автоматная грамматика окажется почти эквивалентной исходной.

Используя лемму 7, мы можем ограничиться изучением интересующих нас свойств А-грамматик, которые, с одной стороны, являются наиболее простым классом порождающих грамматик, а с другой — могут описывать синтаксис любого праволинейного языка. Одной из важных особенностей А-грамматик является возможность их представления в виде графов. Как мы покажем, графическое представление

А-грамматики можно рассматривать не только как удобный аппарат для порождения цепочек соответствующего автоматного языка, но и как анализатор, распознающий цепочки, принадлежащие этому языку, и вырабатывающий их синтаксическую структуру.

Для представления А-грамматики в виде графа сначала нужно сделать небольшое преобразование. Пусть имеется А-грамматика $G = \langle V_T, V_A, I, S \rangle$. Включим в ее множество нетерминальных символов еще один символ: $V'_A = V_A \cup \{K\}$, $K \notin V_A$. Множество правил S преобразуем в множество S' следующим образом. Включим в S' все правила из S вида $A \rightarrow bB$. Вместо заключительных правил, входящих в S , т.е. правил вида $A \rightarrow b$, включим в S' правила $A \rightarrow bK$. Наконец, включим в S' правило $K \rightarrow \lambda$. Образует грамматику $G' = \langle V_T, V'_A, I, S' \rangle$, которая эквивалентна исходной и имеет правила следующих двух типов: $A \rightarrow bB$ и $K \rightarrow \lambda$ (в дальнейшем грамматики с правилами указанного вида будем называть *модифицированными автоматными*). Теперь можно построить соответствующий грамматике G' (и исходной грамматике G) граф. Для этого нужно выполнить следующую процедуру:

1. Поставим в соответствие каждому нетерминальному символу из V'_A вершину графа.

2. Рассмотрим каждое правило вида $A \rightarrow bB$ и соединим вершины A и B графа дугой, направленной от A к B , пометив ее терминальным символом b .

3. Отметим в графе вершину, соответствующую начальному символу (и назовем ее начальной), а также все вершины, соответствующие символам, для которых имеются правила вида $K \rightarrow \lambda$. Эти вершины назовем заключительными.

Если граф строится для А-грамматики, то заключительная вершина всего одна — та, которая соответствует введенному при преобразовании грамматики символу K . Но описанную процедуру можно использовать для построения графа по любой модифицированной автоматной грамматике. В этом случае заключительных вершин может быть несколько. Данная процедура позволяет однозначно построить граф по исходной грамматике (с точностью до расположения вершин). Графы, представляющие автоматные и

модифицированные автоматные грамматики, обладают рядом очевидных свойств. Вывод цепочки языка в грамматике интерпретируется на графе как путь от начальной вершины к одной из заключительных, причем для получения самой цепочки нужно выписать все символы, помечающие пройденные дуги в порядке их прохождения. Если грамматика приведенная, существует путь из начальной вершины графа в любую другую вершину, а также путь из любой вершины к заключительной. Если же некоторые символы грамматики несущественные, граф может оказаться несвязным. Наконец, очевидно, что циклической грамматике соответствует граф с контурами, а если грамматика нециклическая, то не существует пути, в котором некоторая вершина проходится более одного раза. Рассмотрим примеры графических представлений автоматных грамматик.

Пример 2. Грамматика, которая порождает язык идентификаторов АЛГОЛа-60, может быть задана следующими правилами вывода, представленными в виде схем:

- | | |
|--------------------------|----------------------------|
| 1) $I \rightarrow B,$ | 4) $I_1 \rightarrow C,$ |
| 2) $I \rightarrow BI_1,$ | 5) $I_1 \rightarrow BI_1,$ |
| 3) $I_1 \rightarrow B,$ | 6) $I_1 \rightarrow CI_1.$ |

Здесь схемы 1), 2), 3) и 5) определяют по 52 правила вывода, которые получаются из соответствующих схем заменой B на любую малую или большую букву латинского алфавита. Аналогично схемы 4) и 6) определяют по 10 правил (по одному для каждой цифры). Преобразуем данную грамматику в модифицированную автоматную:

- | | |
|--------------------------|-----------------------------|
| 1) $I \rightarrow BK,$ | 5) $I_1 \rightarrow BI_1,$ |
| 2) $I \rightarrow BI_1,$ | 6) $I_1 \rightarrow CI_1,$ |
| 3) $I_1 \rightarrow BK,$ | 7) $K \rightarrow \lambda.$ |
| 4) $I_1 \rightarrow CK,$ | |

Этой грамматике соответствует граф, показанный на рис. 4. Вершина I графа — начальная, а вершина K — заключительная. Из I выходят 52 дуги в вершину I_1 , каждая из которых помечена малой или большой буквой латинского алфавита. Столько же дуг, помеченных буквами, соединяют вершины I и K . Из I_1 выходят 62 дуги в эту же вершину. Они помечены буквами и цифрами. Наконец, из I_1 в K тоже выходят 62 дуги, помеченные буквами и цифрами. Идентификатору $aZ9a$ соответствует следующий путь, представленный последовательностью вершин, между которыми в скобках указываются терминальные символы, помечающие пройденные дуги:

$$I(a)I_1(Z)I_1(9)I_1(a)K.$$

Язык идентификаторов может порождаться модифицированной автоматной грамматикой несколько более простого вида:

- 1) $I \rightarrow BI_1$, 3) $I_1 \rightarrow \zeta I_1$,
- 2) $I_1 \rightarrow BI_1$, 4) $I_1 \rightarrow \lambda$.

Этой грамматике соответствует граф, который может быть получен из приведенного на рис. 4 исключением вершины K и всех

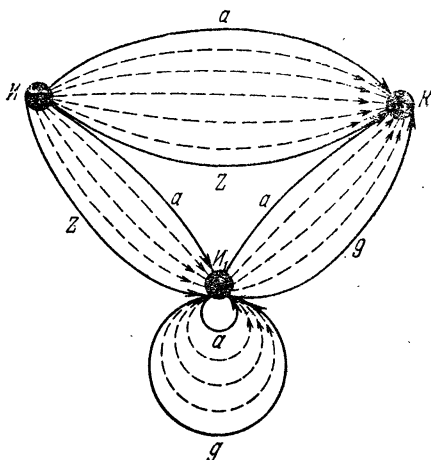


Рис. 4.

дуг, идущих к этой вершине. Заключительной вершиной преобразованного графа является I_1 . Идентификатору $aZga$ соответствует в этом графе следующий путь:

$$I(a)I_1(Z)I_1(g)I_1(a)I_1.$$

В этом случае мы несколько раз проходим заключительную вершину I_1 . Таким образом, в графе, соответствующем модифицированной автоматной грамматике общего вида, путь не обязательно заканчивается в момент попадания в заключительную вершину — при наличии для соответствующего нетерминального символа незаключительных правил он может продолжаться или заканчиваться в зависимости от порождаемой цепочки.

Графическое представление автоматных грамматик позволяет легко установить эквивалентность классов праволинейных и левوليнейных грамматик. Если имеется некоторая левوليнейная грамматика, то по ней можно построить эквивалентную грамматику с правилами вида $A \rightarrow Bb$ и $A \rightarrow b$. Для этого нужно выполнить процедуру, аналогичную описанной в лем-

ме 7. Полученную грамматику легко представить в виде графа с теми же свойствами, которые имеют место для графов, представляющих автоматные грамматики. Единственным различием является то, что при прохождении пути от начальной вершины к заключительной терминальная цепочка должна выписываться не слева направо, а, наоборот, справа налево. Преобразуем полученный граф следующим образом. Поменим местами начальную и заключительную вершины, направление всех дуг графа заменим на противоположное. После этого преобразования граф будет порождать тот же язык и представлять некоторую модифицированную автоматную грамматику с одним заключительным правилом, эквивалентную исходной. Продемонстрируем процедуру преобразования на следующем примере.

Пример 3. Нормальные формы Бекуса, представляющие идентификаторы языка АЛГОЛ-60, задают следующую грамматику:

- 1) $I \rightarrow B$, 2) $I \rightarrow IB$, 3) $I \rightarrow IC$

(здесь, как обычно, I — идентификатор, B — буква, C — цифра, и грамматика представлена схемами правил).

В правых частях правил этой грамматики нет терминальных цепочек, состоящих более чем из одного символа. Поэтому первый этап процедуры преобразования можно пропустить. Заменим первую схему правил двумя следующими: $I \rightarrow KB$, $K \rightarrow \lambda$. Теперь грамматика может представляться графом, изображенным на рис. 5. Вершина I графа — начальная, а K — заключительная.

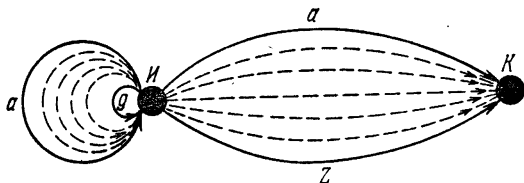


Рис. 5.

Объявим начальной вершину K , заключительной — вершину I и заменим направления дуг на противоположные. После этого получим преобразованный граф из примера 2 (с точностью до обозначений нетерминальных символов, помещающих вершины). Модифицированная грамматика, которой соответствует граф, также приведена в примере 2.

Выше мы упоминали, что графическое представление автоматных грамматик может служить удобным средством для распознавания цепочек, принадлежащих автоматным языкам. Вопросам анализа и распознавания контекстно-свободных и автоматных языков посвящены следующие две главы. Здесь мы лишь опишем процедуру распознавания цепочек с использованием графа, представляющего автоматную грамматику.

Процедура распознавания представляет собой поиск пути от начальной к заключительной вершине графа по дугам, помеченным символами, входящими в анализируемую цепочку. При этом порядок прохождения дуг должен соответствовать порядку вхождений символов в цепочку. Будем считать, что перед выполнением процедуры мы находимся в начальной вершине, а символам, входящим в цепочку, приписаны их порядковые номера слева направо. Рассмотрим дуги, выходящие из начальной вершины. Если среди них нет дуг, помеченных первым символом цепочки, процедура закончена и цепочка не входит в язык. В противном случае выбираем одну из таких дуг, отмечаем ее номером рассматриваемого символа и переходим к вершине, в которую ведет выбранная дуга. Рассматриваем следующий символ цепочки, выбираем какую-либо дугу, помеченную этим символом, отмечаем ее номером символа и попадаем в вершину, в которую входит выбранная дуга. Продолжаем процесс движения по графу. Если по исчерпанию символов цепочки мы окажемся в заключительной вершине, процедура закончена и цепочка входит в язык. Найденный путь легко использовать для получения синтаксической структуры цепочки. Может оказаться, что на i -м шаге процедуры в графе не найдется дуга, помеченная i -м символом цепочки. Тогда проходим дугу, помеченную номером $i-1$, в обратном направлении, а помечающий ее номер $i-1$ подчеркиваем. Далее рассматриваем вершину, в которой мы оказались после прохождения дуги, и $(i-1)$ -й символ цепочки. Ищем среди дуг, выходящих из этой вершины, такую, которая помечена $(i-1)$ -м символом и не отмечена подчеркнутым номером $i-1$. Если такая дуга найдется, продолжаем процедуру поиска пути. В противном

случае возвращаемся еще на один шаг назад. Если при движении назад мы попадем в начальную вершину, и дуги, с которой можно начать поиск другого возможного пути, не окажется, процедура закончена и цепочка не входит в язык. Нетрудно видеть, что, поскольку число вершин и дуг графа конечно, процедура будет в любом случае выполнена за конечное число шагов.

При изучении языков программирования может возникнуть вопрос, какие синтаксические свойства языков мешают их описанию с помощью простых и удобных для использования автоматных грамматик. Основной особенностью, не позволяющей представлять языки программирования как автоматные, является наличие в них парных символов, количество и взаимное расположение которых в программах должно выбираться в соответствии с определенными правилами. В АЛГОЛе-60 такими символами являются круглые и квадратные скобки, операторные скобки `begin` и `end` и некоторые другие символы. Рассмотрим использование круглых скобок в арифметических или логических выражениях. Порождая некоторое конкретное выражение слева направо при помощи автоматной грамматики, мы должны были бы запоминать количество открывающих скобок, для которых соответствующие закрывающие скобки еще должны быть порождены. Поэтому для каждого числа i , определяющего количество «незакрытых» открывающих скобок, нужно было бы иметь свои варианты нетерминальных символов. Так, i -й вариант определял бы, сколько нужно породить закрывающих скобок, чтобы закончить все начатые и вложенные друг в друга выражения в скобках. Так как синтаксис языка допускает любую глубину вложенности выражений в скобках, синтаксис выражений не может быть представлен автоматными грамматиками с конечным числом нетерминальных символов. Это же рассуждение справедливо для блоков, так как глубина вложенности блоков также не ограничивается, и для индексных выражений, поскольку в них могут использоваться переменные с индексами.

ГЛАВА II

МЕТОДЫ РАСПОЗНАВАНИЯ И АНАЛИЗА ЯЗЫКОВ

Для того, чтобы некоторый уже разработанный и описанный язык программирования мог практически применяться для решения задач на электронных вычислительных машинах, нужно составить транслятор, переводящий программы, написанные на этом языке, на язык той машины, которая должна использоваться для решения задач. Языки программирования обладают характерной особенностью, заключающейся в том, что семантика большинства конструкций зависит от их расположения в программах. Так, идентификатор имеет различный смысл и вызывает различные действия транслятора в зависимости от того, находится ли он в описании или в операторе, обозначает ли он процедуру или простую переменную и пр. Поэтому необходимым этапом перевода программ на машинный язык является их синтаксический анализ, определяющий синтаксическую структуру программ (например, в виде дерева вывода в порождающей грамматике), а также обнаруживающий возможные ошибки программирования. Синтаксический анализ тесно связан с проблемой распознавания. Обычно алгоритмы распознавания разрабатываются таким образом, чтобы в случае, если анализируемая цепочка входит в язык, определялась синтаксическая структура этой цепочки в терминах порождающей грамматики языка.

Нашей целью в настоящей главе будет изучение алгоритмов распознавания и синтаксического анализа языков программирования. Мы рассмотрим классы устройств, называемых конечными автоматами и автоматами с магазинной памятью, и покажем, что задаваемые с помощью этих автоматов алгоритмы позволяют решить проблему распознавания для авто-

матных и КС-языков. В гл. III мы остановимся на очень важном и интересном вопросе построения анализаторов языков по их порождающим грамматикам. Гл. IV будет посвящена контекстным условиям, т. е. способам задания и проверки тех синтаксических правил языков программирования, которые не описываются средствами КС-грамматик.

§ 1. Машины Тьюринга как распознающие устройства

Одной из наиболее известных математических моделей, осуществляющих выполнение алгоритмов, являются *машины Тьюринга*. Определение машины Тьюринга широко известно. Тем не менее мы кратко остановимся на ее основных чертах. Машину Тьюринга можно рассматривать как механическое устройство, состоящее из следующих основных частей.

1) В машине имеется потенциально неограниченная память, разбитая на отдельные линейно-упорядоченные ячейки. В каждой ячейке может быть записан символ из некоторого конечного алфавита, или же она может быть пустой. В последнем случае часто для удобства считают, что в ячейке записан особый символ, называемый *пустым*. В каждый момент времени память, обычно называемая *рабочей лентой* машины, состоит из конечного числа ячеек, однако при необходимости к ней могут быть пристроены слева или справа новые ячейки с записанным в них пустым символом. Итак, рабочая лента и информация, записанная в ней, представляются конечной цепочкой символов над словарем рабочей ленты.

2) Помимо рабочей ленты в машине Тьюринга имеется еще и другое запоминающее устройство. Это *регистр состояний* — особая память, рассчитанная на хранение одного символа. Символ, который запоминается в регистре, выбирается из конечного множества, определяющего *множество состояний* машины.

3) В каждый момент времени машина Тьюринга анализирует не всю информацию, хранящуюся на рабочей ленте, а содержимое лишь одной ячейки этой ленты. Для определения этой ячейки служит *управляющая головка*, которая всегда «указывает» на некоторую ячейку рабочей ленты.

Выполняя заданный алгоритм, машина Тьюринга последовательно производит ряд элементарных действий, причем каждое такое действие выполняется за один рабочий такт машины. Элементарные действия можно разбить на следующие три группы:

1) Машина изменяет состояние в регистре (т. е., стирая символ, хранящийся в регистре, записывает туда новый символ) и содержимое ячейки, на которую указывает управляющая головка.

2) Машина изменяет состояние и продвигает управляющую головку на одну ячейку влево.

3) Машина изменяет состояние и продвигает управляющую головку на одну ячейку вправо.

В последних двух случаях может оказаться, что до такта управляющая головка указывала на самую левую или самую правую ячейку рабочей ленты. Если требуется произвести сдвиг влево (или, соответственно, вправо), то к рабочей ленте пристраивается новая ячейка с записанным в ней пустым символом.

Машина Тьюринга может использоваться для вычисления функций, аргументы и значения которых представляются цепочками символов конечных алфавитов. При этом машина начинает работу в так называемой *начальной ситуации*, которая характеризуется следующим образом:

1) на рабочей ленте записан аргумент вычисляемой функции;

2) управляющая головка указывает на ячейку, в которой записан самый левый символ аргумента;

3) машина находится в некотором заранее выбранном состоянии, которое называется *начальным*.

Начиная работу в начальной ситуации, машина работает до тех пор, пока не окажется в некотором особом состоянии, называемом *заключительным*. Значением вычисляемой функции считается цепочка непустых символов, выписанных слева направо из рабочей ленты после окончания работы машины.

Рассмотрим теперь формальное определение машины Тьюринга и процесса ее работы.

О п р е д е л е н и е. Машиной Тьюринга называется пятерка следующих объектов:

$$T = \langle K, H, \delta, q_0, F \rangle,$$

где K — конечное множество состояний машины; H — конечное множество символов рабочей ленты, причем мы будем считать, что пересечение K и H пусто, а символы L и R не входят в множество H ; q_0 — начальное состояние, $q_0 \in K$; F — подмножество заключительных состояний, $F \subseteq K$; δ — функция, отображающая множество $K \times H$ в семейство всех подмножеств множества $K \times (H \cup \{L, R\})$.

Функция δ определяет элементарные действия машины Тьюринга. Пусть машина находится в состоянии q и читает на рабочей ленте символ a (т. е. управляющая головка указывает на ячейку, в которой записан этот символ). Тогда каждый элемент подмножества $\delta(q, a)$ определяет возможное на данном такте действие машины. Если, например, $(p, b) \in \delta(q, a)$, то машина может перейти в состояние p , заменив в соответствующей ячейке рабочей ленты символ a на b . Если $(p, L) \in \delta(q, a)$, то машина может перейти в состояние p , сдвинув управляющую головку влево. Наконец, в случае $(p, R) \in \delta(q, a)$ машина может перейти в состояние p , сдвинув управляющую головку вправо. Если $\delta(q, a)$ содержит более одного элемента, машина может выполнить любое действие, определяемое одним из элементов.

Информацию, хранимую в памяти машины Тьюринга, и положение управляющей головки удобно представлять в виде так называемой *конфигурации* машины, которая задается цепочкой символов следующего вида:

$$a_{i1}a_{i2} \dots a_{ik-1}qa_{ik} \dots a_{in},$$

где a_{ii} — символы рабочей ленты, q — состояние машины. Данная конфигурация означает, что на рабочей ленте записана цепочка $a_{i1} \dots a_{in}$, машина находится в состоянии q и управляющая головка указывает на ячейку, в которой записан символ a_{ik} . Очевидно, цепочка указанного вида может представлять конфигурацию машины Тьюринга только в том случае, если символ состояния машины не является в цепочке крайним правым символом. Пусть имеются две конфигурации α и β некоторой машины Тьюринга. Обозначение $\alpha \vdash \beta$ будем использовать тогда, когда машина может перейти от конфигурации α к кон-

фигурации β за один рабочий такт. Например, если

$$\alpha = a_{i1} \dots a_{ik-1} q a_{ik} a_{ik+1} \dots a_{in},$$

то в зависимости от функции δ конфигурация β может иметь один из следующих видов:

- а) $a_{i1} \dots a_{ik-1} p a_j a_{ik+1} \dots a_{in}$, если $(p, a_j) \in \delta(q, a_{ik})$;
- б) $a_{i1} \dots p a_{ik-1} a_{ik} a_{ik+1} \dots a_{in}$, если $(p, L) \in \delta(q, a_{ik})$;
- в) $a_{i1} \dots a_{ik-1} a_{ik} p a_{ik+1} \dots a_{in}$, если $(p, R) \in \delta(q, a_{ik})$.

Длина цепочки, представляющей конфигурацию машины, может в процессе работы увеличиваться или уменьшаться. И то, и другое происходит, когда символ состояния находится перед крайним правым или крайним левым символом рабочей ленты. Если на данном такте работы он сдвигается, соответственно, вправо или влево, то к цепочке приписывается пустой символ. Если, наоборот, символ состояния сдвигается к центру цепочки, а крайний символ рабочей ленты пуст, то этот символ исключается из цепочки.

Если машина может перейти из конфигурации α к конфигурации β за несколько тактов, используется обозначение $\alpha \vdash^* \beta$. Иными словами, пишем $\alpha \vdash^* \beta$ в том случае, если найдутся такие конфигурации $\alpha_0 = \alpha, \alpha_1, \dots, \alpha_n = \beta$, что $\alpha_i \vdash \alpha_{i+1}$ при $i = 0, 1, \dots, n-1$.

Используя понятие конфигурации, можно следующим образом представить процесс вычисления машиной Тьюринга некоторой функции.

Определение. Будем говорить, что машина Тьюринга *вычисляет* функцию $f(\alpha)$, если $q_0 \vdash^* \beta$, где $\alpha \in H^*$, $l(\alpha) \geq 1$, β — конфигурация, в которую входит одно из заключительных состояний машины. Цепочку, полученную из β вычеркиванием символа состояния и всех вхождений пустых символов, будем считать значением функции f от аргумента α .

Сделаем ряд замечаний относительно рассмотренного нами определения машины Тьюринга и процесса вычисления функций. Во-первых, машины Тьюринга можно разделить на два больших класса: *детерминированных* и *недетерминированных* машин Тьюринга. Машина Тьюринга называется детерминированной, если любая ее конфигурация однозначно

определяет следующий такт работы машины. Иными словами, в случае детерминированной машины функция δ должна быть однозначной, т. е. отображать множество $K \times H$ в множество $K \times (H \cup \{L, R\})$. Если же для некоторых конфигураций такт машины определяется неоднозначно, то машина называется недетерминированной. Теперь рассмотрим процесс работы машины при вычислении некоторой функции. Может оказаться, что при некоторых значениях аргумента вычисляемой функции машина будет работать вечно, никогда не переходя в заключительное состояние, или же перейдет в такую конфигурацию, для которой продолжение работы не определено, причем состояние машины — незаключительное¹⁾. В обоих случаях значение функции от таких аргументов можно считать неопределенным. Можно ли распознать такие цепочки, представляющие аргументы функции, вычисляемой некоторой машиной Тьюринга, для которых значения этой функции не определены? В общем случае эта проблема неразрешима. Она, однако, оказывается разрешимой для некоторых специальных классов машин Тьюринга, представляющих наибольший интерес для распознавания языков программирования. Эти классы машин Тьюринга будут нами подробно рассмотрены ниже.

Поскольку основным содержанием данной главы является изучение способов распознавания языков, наиболее интересным вариантом машин Тьюринга является такой, который позволяет определить, входит ли цепочка символов в некоторое интересующее нас множество цепочек. При таком использовании машины распознаваемому языку можно поставить в соответствие машину Тьюринга, которая вычисляет некоторую функцию, определенную как раз на тех цепочках, которые входят в язык. При этом нас пока не будут интересовать значения вычисляемой функции. Итак, если, вычисляя функцию от аргумента α , машина при удачном варианте работы перейдет в заключительное состояние, будем считать, что α входит в интересующий нас язык. В противном слу-

¹⁾ Это возможно, если функция δ данной машины — частично определенная. В некоторых случаях такие машины рассматривать удобнее, чем те, в которых функция δ полностью определена,

чае α в язык не входит. Можно показать, что указанная машина Тьюринга существует для любого рекурсивно-перечислимого множества цепочек. Однако это еще не значит, что для любого такого множества может быть решена проблема распознавания. Для того, чтобы использовать машину Тьюринга в качестве распознающего механизма, нужно не только построить такую машину, которая переходила бы в заключительное состояние в случае, если цепочка принадлежит языку, но и еще уметь определить такие цепочки, обрабатывая которые, она не может перейти в заключительное состояние. А как только что было замечено, в общем случае эта проблема не разрешима.

Рассмотрим ряд специальных классов машин Тьюринга, которые могут использоваться для распознавания языков, порождаемых грамматиками изученных нами типов. Во-первых, нас будут интересовать такие машины, рабочая лента которых ограничена длиной исходной цепочки. В начальный момент времени на рабочей ленте такой машины записана анализируемая цепочка, управляющая головка указывает на первый слева символ цепочки, и машина находится в начальном состоянии. В процессе работы машина не пристраивает к ленте новых ячеек. Будем, кроме того, считать, что за один такт работы машина может одновременно изменить состояние и символ на рабочей ленте, на который указывает управляющая головка, и сдвинуть, если нужно, головку на одну ячейку вправо или влево. Это изменение в определении работы машины вводится лишь для удобства, и, очевидно, любая такая машина может быть преобразована в эквивалентную машину Тьюринга, работающую так, как было определено раньше. Точное определение машины Тьюринга с ограниченной рабочей лентой, которая часто называется линейно-ограниченным автоматом (ЛО-автоматом), выглядит следующим образом.

О п р е д е л е н и е. *Линейно-ограниченным автоматом* называется следующая пятерка объектов:

$$B = \langle K, H, \delta, q_0, F \rangle,$$

где K — непустое множество состояний, H — непустое

стое множество символов рабочей ленты, $K \cap H = \emptyset$, δ — функция, отображающая множество $K \times H$ в семейство всех подмножеств множества $K \times H \times \{L, R, N\}$, q_0 — начальное состояние, $q_0 \in K$, F — подмножество заключительных состояний, $F \subseteq K \setminus \{q_0\}$.

Здесь символы L , R и N означают, соответственно, сдвиг головки влево, сдвиг вправо и отсутствие сдвига. Конфигурациями машины будем считать цепочки длины $l+1$, где l — длина исходной цепочки, в которые входят символы рабочей ленты и один символ состояния. Будем при этом допускать и такие цепочки, в которых символ состояния находится в крайней правой позиции. В случае, если машина пришла в такую конфигурацию, продолжение ее работы не определяется. Отношения между конфигурациями $\alpha \vdash \beta$ и $\alpha \vdash \beta$ определим аналогично случаю машины Тьюринга общего вида. Наконец, будем говорить, что ЛО-автомат допускает цепочку $\alpha \in H^*$, если $q_0 \alpha \vdash \beta f$, где $\beta \in H^*$ и $f \in F$, а множество цепочек, допускаемых ЛО-автоматом B , назовем языком, допускаемым автоматом.

Нетрудно видеть, что ЛО-автомат может использоваться для распознавания допускаемого им языка. Алгоритм, определяющий, является ли произвольная цепочка над H длины l допустимой, заключается в следующем.

Образуем все возможные конфигурации автомата длиной $l+1$. Таких конфигураций конечное число. Затем образуем из этих конфигураций все возможные последовательности так, чтобы каждая конфигурация входила в последовательность не более одного раза. Таких последовательностей также конечное число. Рассмотрим последовательности, начинающиеся с конфигурации $q_0 \alpha$, где α — анализируемая цепочка, и заканчивающиеся конфигурациями вида βf , где β — некоторая цепочка над H и $f \in F$. Если среди этих последовательностей найдется такая, которая определяет возможный вариант работы ЛО-автомата, цепочка α допускается автоматом, в противном случае она не допускается.

Следующая теорема указывает связь между ЛО-автоматами и НС-грамматиками.

Теорема 4. *Классы языков допускаемых ЛО автоматами, и НС-языков совпадают.*

Доказательство. Воспользуемся прежде всего упомянутым нами в гл. I свойством эквивалентности НС- и неукорачивающих грамматик. Так как классы языков, порождаемых НС-грамматиками и неукорачивающими грамматиками, совпадают, для доказательства теоремы достаточно показать, что совпадают и классы неукорачивающих языков и языков, допускаемых ЛО-автоматами.

Пусть имеется произвольный ЛО-автомат, имеющий n символов рабочей ленты и m состояний. Построим неукорачивающую грамматику, «имитирующую» работу автомата в обратном порядке. Для этого нужно, во-первых, каждой конфигурации автомата поставить в соответствие некоторую цепочку, состоящую из терминальных и нетерминальных символов грамматики. Пусть имеется конфигурация ЛО-автомата: $\alpha = a_1 a_2 \dots a_{k-1} q_j a_k \dots a_p$. Этой конфигурации поставим в соответствие цепочку

$$C(\alpha) = a'_1 a_2 \dots a_{k-1} a_{k+1} \dots a_p,$$

причем символ со штрихом этой цепочки и символ с двумя индексами будем считать нетерминальными, а остальные символы — терминальными символами грамматики. Символы со штрихом также могут быть снабжены двумя индексами — в случае, если цепочка соответствует конфигурации, в которой символ состояния находится в крайней левой позиции. Из указанного представления видно, что грамматика должна содержать по крайней мере $(2m+1)n$ нетерминальных символов. Правила грамматики, имитирующие работу ЛО-автомата, определим так, чтобы соотношение $C(\beta) \Rightarrow C(\alpha)$ в грамматике имело место тогда и только тогда, когда $\alpha \vdash \beta$. Теперь заметим, что после перехода ЛО-автомата в заключительное состояние на его рабочей ленте может находиться некоторая цепочка. Поэтому первым этапом работы имитирующей грамматики должно быть построение цепочек, представляющих такую конфигурацию автомата, из которой он может перейти в заключительную за 1 такт. Если имитирующая грамматика

сможет при удачном выборе варианта работы преобразовать полученную на первом этапе работы цепочку в терминальную, то последняя допускается ЛО-автоматом. Итак, мы должны построить грамматику, которая работает следующим образом.

1) Получает цепочку, представляющую некоторую конфигурацию ЛО-автомата, непосредственно предшествующую заключительной, т. е. цепочку вида $a'_1 \dots a_{p_j}$ или a'_{p_j} , причем

$$(f, a_k, R) \in \delta(q_j, a_p), f \in F.$$

2) Имитирует работу ЛО-автомата, т. е. использует правила, в левые и правые части которых входят символы с двумя индексами.

3) Завершает работу, преобразуя цепочки, представляющие начальные конфигурации автомата, в терминальные цепочки.

Перейдем к формальному определению грамматики с указанными свойствами. Пусть задан произвольный ЛО-автомат

$$B = \langle K, H, \delta, q_1, F \rangle,$$

причем $K = \{q_1, \dots, q_m\}$, $H = \{a_1, \dots, a_n\}$.

Построим по нему грамматику $G = \langle V_T, V_A, I, S \rangle$, в которой $V_T = H$, $V_A = \{a'_i\} \cup \{a'_{ij}\} \cup \{a'_{ijl}\} \cup \{I, A\}$, $i = 1, \dots, n$; $j = 1, \dots, m$. Включим в S следующие правила:

1) $I \rightarrow Aa_{ij}$, если $(q_k, a_i, R) \in \delta(q_j, a_i)$, $q_k \in F$;

$I \rightarrow a'_{ij}$ при тех же условиях;

$A \rightarrow Aa_i$ для любых a_i ;

$A \rightarrow a'_i$ для любых a'_i .

Приведенные правила используются на первом этапе вывода.

2) $a_{ij}a_k \rightarrow a_ia_{lm}$ и $a'_ia_k \rightarrow a'ia_{lm}$,

если $(q_j, a_k, L) \in \delta(q_m, a_i)$ для всех i ;

$a_ka_{ij} \rightarrow a_{lm}a_i$ и $a'_ka_{ij} \rightarrow a'_{lm}a_i$,

если $(q_j, a_k, R) \in \delta(q_m, a_i)$ для всех i ;

$a_{kj} \rightarrow a_{lm}$ и $a'_{kj} \rightarrow a'_{lm}$, если $(q_j, a_k, N) \in \delta(q_m, a_i)$.

Эти правила применяются на втором этапе вывода.

3) $a_{ii} \rightarrow a_i$ для любых a_i .

Это — группа заключительных правил, преобразующих цепочки в терминальные.

Язык, порождаемый построенной нами грамматикой, очевидно, совпадает с языком, допускаемым исходным ЛО-автоматом.

Докажем обратное утверждение: по любой неукорачивающей грамматике можно построить ЛО-автомат, допускающий язык, который порождается данной грамматикой. Работу ЛО-автомата, который будет формально определен ниже, можно разбить на три этапа.

Сначала автомат просматривает цепочку, проверяет, входят ли в нее лишь терминальные символы грамматики, и отмечает левый и правый символы цепочки. После этого начинается второй этап работы, который заключается в том, что автомат в цепочке, записанной на рабочей ленте, пытается найти подцепочки, являющиеся правыми частями правил грамматики. Если такая подцепочка найдена, автомат заменяет правую часть соответствующего правила (т. е. найденную подцепочку) на левую часть. При этом в случае, если длина левой части меньше длины правой, в оставшиеся ячейки рабочей ленты записывается некоторый «нейтральный» символ, не влияющий на работу автомата. Если анализируемая цепочка входит в язык, то должен найтись такой вариант работы ЛО-автомата, при котором эта цепочка будет преобразована в цепочку, состоящую из начального символа грамматики и последовательности нейтральных символов. В этом случае начинается третий этап работы. Автомат просматривает цепочку до крайнего правого нейтрального символа (этот символ будет отмечен) и переходит в заключительное состояние. Если первоначальная цепочка в язык не входит, автомат при ее анализе либо придет в тупиковую конфигурацию, в которой его дальнейшая работа не определена, либо перейдет в заключительное состояние в момент, когда управляющая головка будет указывать на некоторый символ цепочки. Особо анализируются цепочки длины 1.

Перейдем к формальному определению ЛО-автомата по неукорачивающей грамматике. Пусть имеется неукорачивающая грамматика $G = \langle V_T, V_A, I, S \rangle$.

Определим по грамматике G ЛО-автомат $B = \langle K, H, \delta, q_1, F \rangle$ следующим образом. Пусть множество правил грамматики содержит p правил. Перенумеруем эти правила. По каждому правилу $S_i = \xi \rightarrow \eta$, где $l(\eta) = n > 1$, определим группу состояний ЛО-автомата: $K_i = \{q_{ij}\}$, $j = 1, \dots, n-1$. Включим во множество состояний автомата также следующие состояния:

$K_0 = \{q_1, \dots, q_5, f\}$. Итак, $K = K_0 \cup \bigcup_{i=1}^p K_i$. Во мно-

жество H символов рабочей ленты включим терминальные и нетерминальные символы грамматики, а также те же символы, помеченные сверху значками $\hat{}$ или $\tilde{}$ (при помощи помеченных символов мы будем отмечать левый и правый символы анализируемой цепочки). Кроме того, введем в рассмотрение две модификации нейтрального символа: $H_0 = \{\sigma, \tilde{\sigma}\}$. Итак, $H = V_T \cup V_A \cup V_T \cup \hat{V}_A \cup \tilde{V}_T \cup \tilde{V}_A \cup H_0$.

Определим теперь функцию δ .

А) Рассмотрим сначала группу значений функции, определяющих работу автомата при первоначальном просмотре анализируемой цепочки (значения 4)–6), будут использоваться и на втором этапе работы).

1) $(q_2, \hat{a}, R) \in \delta(q_1, a)$ для любого $a \in V_T$;

2) $(q_2, a, R) \in \delta(q_2, a)$ для любого $a \in V_T$;

3) $(q_3, \tilde{a}, N) \in \delta(q_2, a)$ для любого $a \in V_T$;

4) $(q_3, \tilde{\alpha}, L) \in \delta(q_3, \tilde{\alpha})$ для любого

$\tilde{\alpha} \in \tilde{V}_T \cup \tilde{V}_A \cup \{\tilde{\sigma}\}$;

5) $(q_3, \alpha, L) \in \delta(q_3, \alpha)$ для любого

$\alpha \in V_T \cup V_A \cup \{\sigma\}$;

6) $(q_4, \hat{\alpha}, N) \in \delta(q_3, \hat{\alpha})$ для любого

$\hat{\alpha} \in \hat{V}_T \cup \hat{V}_A$.

Б) Для любого правила грамматики S_i подберем значения функции δ так, чтобы автомат осуществлял постановку в анализируемую цепочку левой части правила вместо некоторого вхождения его правой час-

ных символов грамматики бинарное отношение D_N так же, как это делалось для нетерминальных символов КС-грамматик. Очевидно, $I \xrightarrow[G]{*} a$ в том и только том случае, если $ID_N a$. Итак, включим в данную группу следующие значения δ : $(f, a, R) \in \delta(q_1, a)$ для всех a таких, что $I \xrightarrow[G]{*} a$.

Поясним работу построенного ЛО-автомата. Цепочки длиной 1 анализируются за один шаг: если ни одно из правил группы Г) не подходит, цепочка не входит в язык $L(G)$. При анализе цепочки длиной больше 1 автомат работает так, как было указано выше. Заметим, что при помощи правил группы А) проверяется, не входят ли в цепочку символы, которые не являются терминальными символами грамматики. В качестве правого символа цепочки может быть отмечен любой символ, начиная со второго слева. Если будет отмечен не самый правый символ, автомат в дальнейшем придет в тупиковую конфигурацию. На втором этапе анализа после каждой подстановки используются правила 4) — 6) группы А), и головка рабочей ленты сдвигается в позицию крайнего левого символа. Если в процессе подстановки окажется, что очередной символ анализируемой цепочки не совпадает с соответствующим символом правой части правила, автомат оказывается в тупиковой конфигурации. На третьем этапе анализа, обнаруживая начальный символ грамматики в крайней левой позиции, автомат читает цепочку до правого символа и, если все остальные символы цепочки оказываются нейтральными, переходит в заключительное состояние. Если анализируемая цепочка входит в язык $L(G)$, то, очевидно, найдется такой вариант работы ЛО-автомата, при котором он производит подстановки по правилам в последовательности, в точности обратной той, в которой правила грамматики использовались при выводе цепочки. Поэтому цепочка допускается автоматом. Наоборот, если вывода анализируемой цепочки в грамматике не существует, не существует и последовательности правил, используя которую, автомат может преобразовать цепочку к начальному символу.

Следовательно, такая цепочка не допускается автоматом. Итак, язык, допускаемый построенным нами ЛО-автоматом, совпадает с языком $L(G)$.

Доказательство теоремы закончено.

Приведем пример построения ЛО-автомата по неукорачивающей грамматике.

Пример 1. Рассмотрим грамматику из примера 3, § 2, гл. I (стр. 43), порождающую язык $L_5 = \{a^n b^n a^n\}$. Эта грамматика имеет следующие правила:

- 1) $I \rightarrow aCa$, 4) $aB \rightarrow Ba$,
- 2) $C \rightarrow aCBa$, 5) $bB \rightarrow bb$.
- 3) $C \rightarrow b$,

Определим по грамматике ЛО-автомат, используя метод, описанный при доказательстве теоремы 4. Для того, чтобы несколько упростить функцию δ , мы можем учесть следующие свойства языка и грамматики. Во первых, в цепочках языка крайними символами могут быть только символы a . Во вторых, в процессе анализа цепочки крайним левым символом может быть только a или I (это следует из того, что при выводе цепочек первым применяется правило 1-грамматики), а крайним правым — только a или нейтральный символ σ . Поэтому множество H несколько сокращается. Оно будет включать следующие символы:

$$H = \{a, \hat{a}, \tilde{a}, b, \hat{I}, B, C, \sigma, \tilde{\sigma}\}.$$

Определим теперь функцию δ , представив ее в виде таблицы, в левой колонке которой указываются аргументы, для которых значения функции определены, а в правой — одно или несколько возможных значений. Будем считать, что во множества состояний автомата войдут все состояния, входящие в строки таблицы, причем начальным состоянием будем считать q_1 , а заключительным — f . Функция δ представлена на табл. 1.

Рассмотрим работу автомата при анализе цепочки $aaabbbbaaa$, указывая лишь наиболее интересные конфигурации, в которые приходит автомат при работе:

$$\begin{aligned} q_1aaabbbbaaa &\vdash^* q_1\hat{a}aabbbbaa\hat{a} \vdash^* \hat{a}aabq_1bbbaa\hat{a} \vdash^* \\ q_1\hat{a}aabbBaa\hat{a} &\vdash^* \hat{a}aaq_1bbBaa\hat{a} \vdash^* q_1\hat{a}aabBBaa\hat{a} \vdash^* \\ \hat{a}aaq_1bBBaa\hat{a} &\vdash^* q_1\hat{a}aaCBaa\hat{a} \vdash^* \hat{a}aaCBq_1Baa\hat{a} \vdash^* \\ q_1\hat{a}aaCBaBa\hat{a} &\vdash^* \hat{a}aq_1aCBaBa\hat{a} \vdash^* q_1\hat{a}aC\sigma\sigma\sigma Ba\hat{a} \vdash^* \\ q_1\hat{a}C\sigma\sigma\sigma\sigma\sigma\sigma\hat{a} &\vdash^* q_1\hat{I}\sigma\sigma\sigma\sigma\sigma\sigma\tilde{\sigma} \vdash^* \hat{I}\sigma\sigma\sigma\sigma\sigma\sigma\tilde{\sigma}f. \end{aligned}$$

Если анализируемая цепочка не входит в язык L_5 , ЛО-автомат при любом варианте анализа перейдет в тупиковую ситуацию. Например, один из вариантов анализа цепочки aba следующий:

$$q_1aba \vdash^* q_1\hat{a}ba \vdash^* q_1\hat{I}\sigma\hat{a} \vdash^* \hat{I}\sigma q_5\hat{a} \text{ — тупик.}$$

Таблица 1

Аргументы	Значения
q_1, a	q_2, \hat{a}, R
q_2, a	$q_2, a, R; q_3, \bar{a}, N$
q_2, b	q_2, b, R
q_3, a	q_3, a, L для $a \in \{a, \bar{a}, b, B, C, \sigma, \bar{\sigma}\}$
q_3, \hat{a}	q_4, \hat{a}, N
q_3, \hat{I}	q_4, \hat{I}, N
q_4, a	$q_{21}, C, R; q_4, a, R$
q_4, \hat{a}	$q_{11}, \hat{I}, R; q_4, \hat{a}, R$
q_4, b	$q_3, C, N; q_{51}, b, R; q_4, b, R$
q_4, B	$q_{41}, a, R; q_4, B, R$
q_4, \hat{I}	q_5, \hat{I}, R
q_4, σ	q_4, σ, R
q_5, σ	q_5, σ, R
$q_5, \bar{\delta}$	$f, \bar{\sigma}, R$
q_{11}, C	q_{12}, σ, R
q_{12}, \bar{a}	$q_3, \bar{\sigma}, N$
q_{21}, C	q_{22}, σ, R
q_{22}, B	q_{23}, σ, R
q_{23}, a	q_3, σ, N
q_{41}, a	q_3, B, N
q_{51}, b	q_3, B, N
q_{ij}, σ	q_{ij}, σ, R для $q_{ij} = q_{11}, \dots, q_{51}$

Итак, мы доказали, что для любого НС-языка может быть построен распознающий ЛО-автомат. Ниже мы покажем, что полный синтаксис языков программирования типа АЛГОЛ-60 с учетом контекстных условий может задаваться НС-грамматиками, и, следовательно, эти языки являются НС-языками. Таким образом, для распознавания языка программирования можно использовать подходящий ЛО-автомат. Однако метод построения автомата, предложенный при доказательстве теоремы 4, не годится для практического применения. Как видно из примера, автомат получается слишком усложненным — распознавание сводится к перебору очень большого числа вариантов.

Определенным недостатком машин Тьюринга рассмотренного вида и, в частности, ЛО-автоматов, при использовании их для распознавания является то обстоятельство, что анализируемая цепочка преобразуется в процессе работы. Удобнее для целей распознавания определить машину Тьюринга так, чтобы она читала распознаваемую цепочку, записанную в специальной памяти — входной ленте, слева направо, не меняя эту цепочку. При этом можно считать, что в рабочей ленте в начальный момент никакая информация не содержится, а в процессе работы туда записывается некоторая цепочка, которая преобразуется по правилам, определенным для машин Тьюринга. Различные классы таких «двухленточных» машин Тьюринга обычно используются для распознавания формальных языков, в том числе языков программирования. Приведем их точное определение.

О п р е д е л е н и е. *Машиной Тьюринга с входной лентой* называется шестерка следующих объектов:

$$T' = \langle K, H, \Sigma, \delta, q_0, F \rangle,$$

где K — конечное множество состояний машины; H — конечное множество символов рабочей ленты; Σ — конечное множество символов входной ленты; q_0 — начальное состояние; F — подмножество заключительных состояний; δ — функция, отображающая множество $K \times H \times \Sigma$ в семейство всех подмножеств множества $K \times (H \cup \{L, R\})$.

Как видно из определения, машина Тьюринга с входной лентой отличается от обычной машины Тьюринга лишь наличием множества Σ и тем, что δ зависит от элементов этого множества. Работу машины Тьюринга с входной лентой можно представить следующим образом. В начальный момент времени рассматривается первый слева символ входной ленты. На рабочей ленте записан один символ — *граничный маркер*. Машина находится в начальном состоянии. На первом такте работы машина, читая первый символ входной цепочки, переходит в некоторое состояние и преобразует информацию на рабочей ленте согласно соответствующему значению функции δ . На i -м такте работы машина производит аналогичные действия, рассматривая очередной символ входной

цепочки. Допустим и такой вариант работы, при котором машина не читает очередной входной символ, а выполняет такт, определенный функцией δ , зависящий от пустого символа из Σ . Иными словами, машина между любыми двумя символами входной цепочки может при необходимости вставить пустой символ и именно этот символ учитывать на очередном такте. Считается, что машина допускает входную цепочку, если при удачном выборе варианта работы она сможет перейти в заключительное состояние, после чтения последнего (крайнего правого) символа входной цепочки.

Легко показать, что по любой классической машине Тьюринга T , распознающей некоторый язык, можно построить машину Тьюринга T' с входной лентой, распознающую тот же язык. Справедливо и обратное утверждение: по любой T' можно построить распознающую тот же язык машину T .

Нас будет интересовать еще одна разновидность машин Тьюринга — *машина с входной и выходной лентами*. Машины Тьюринга с входной лентой хотя и удобны для распознавания, но не дают возможности получить в приемлемом виде информацию об анализируемой цепочке. Между тем мы знаем, что в результате анализа программ необходимо не только получить ответ на вопрос, правильна ли анализируемая цепочка, но и найти синтаксическую структуру правильной цепочки. При использовании для распознавания машин Тьюринга с входной и выходной лентами выходная лента как раз и служит для записи синтаксической структуры. Определение такой машины подобно определению машины T' с той разницей, что на каждом такте работы машина с выходной лентой может не все записать некоторую цепочку символов. Поэтому вводится в рассмотрение еще одно конечное множество символов Δ — множество символов выходной ленты, а функция δ принимает значения из $K \times (N \cup \{L, R\}) \times \Delta^*$. Машины Тьюринга с входной и выходной лентами часто называют *преобразователями*. Выходную ленту можно использовать не только для фиксации синтаксической структуры цепочки, но и для ее преобразования. Например, можно построить преобразователь, представля-

ющий простое арифметическое выражение языка АЛГОЛ-60 в обратной польской записи.

С точки зрения теории языков программирования представляют интерес не столько машины Тьюринга общего вида, сколько некоторые классы этих машин. Мы уже рассмотрели один такой класс — ЛО-автоматы — и показали, что машины этого класса распознают НС-языки. Аналогично случаю классических машин можно определить ЛО-автоматы с входной лентой и ЛО-преобразователи. Помимо линейно-ограниченных машин мы рассмотрим еще следующие два класса машин Тьюринга:

1) *Автоматы и преобразователи с магазинной памятью*. Это машины Тьюринга с входной лентой (а в случае преобразователя и с выходной), в которых работа с рабочей лентой определяется следующим образом: управляющая головка всегда указывает на крайний правый непустой символ ленты, и за один такт на рабочей ленте возможны действия:

а) стирание крайнего правого символа (замена его на пустой символ) со сдвигом головки на одну ячейку влево:

б) стирание крайнего правого символа и запись на рабочую ленту, начиная с ячейки, в которой находился стертый символ, непустой цепочки; при этом управляющая головка устанавливается в позицию, соответствующую крайнему правому символу преобразованной цепочки.

Итак, действия в автомате с магазинной памятью определены таким образом, что такт автомата всегда зависит от символа на рабочей ленте, записанного последним. Таким образом, порядок учета символов рабочей ленты аналогичен тому, в котором используются патроны боевого автомата: патрон, вложенный в магазин автомата позже, используется раньше. Поэтому рабочая лента автомата с магазинной памятью обычно называется магазинной памятью или магазином (см. рис. 6).

Из соображений удобства за один такт работы автомата допускается запись в магазинную память нескольких символов. Легко показать, что по любому автомату с магазинной памятью может быть построен автомат, допускающий тот же язык, в котором за

один такт работы в магазинную память записывается не более одного символа.

Ниже мы покажем, что автоматы с магазинной памятью распознают класс языков, совпадающий с

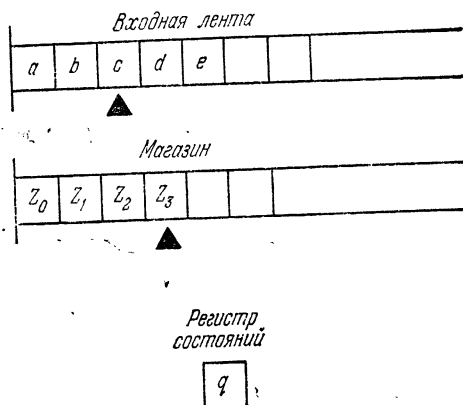


Рис. 6.

классом УКС-языков. С другой стороны, эти автоматы сравнительно просто моделируются на вычислительных машинах. Поэтому они часто используются для синтаксического анализа языков программирования.

2) Простейшим классом машин Тьюринга являются *конечные автоматы*. Конечный автомат можно рассматривать как машину Тьюринга с входной (в случае конечного преобразователя — и с выходной) лентой, но без рабочей ленты. Такт работы конечного автомата заключается в анализе очередного символа входной цепочки и переходе в другое состояние. Можно показать, и это будет сделано в следующем параграфе, что конечные автоматы распознают класс языков, порождаемых модифицированными автоматными грамматиками.

§ 2. Свойства конечных автоматов

Определение. Конечным автоматом называется упорядоченная пятерка следующих объектов:

$$A = \langle K, \Sigma, \delta, q_0, F \rangle,$$

где K — конечное множество состояний; Σ — конеч-

ное множество входных символов; δ — отображение множества $K \times \Sigma$ в множество K ; q_0 — выделенный элемент из K (начальное состояние); F — подмножество множества K (заключительные состояния).

Функция δ , которая определяет работу конечного автомата, может быть доопределена до отображения $K \times \Sigma^*$ в K , где Σ^* означает, как обычно, множество цепочек над алфавитом Σ . Пусть q — некоторое состояние автомата, а ω — цепочка над Σ . Если $\omega = \lambda$, то примем $\delta(q, \omega) = q$. Если $\omega = a_1 a_2 \dots a_n$, где a_i — входные символы, то будем считать, что

$$\delta(q, \omega) = \delta(\dots \delta(\delta(q, a_1), a_2), \dots, a_n),$$

т. е. значение $\delta(q, \omega)$ — это то состояние, в котором окажется автомат после завершения чтения входной цепочки ω , если он начинал работу, находясь в состоянии q .

Будем говорить, что конечный автомат A допускает цепочку $\omega \in \Sigma$, если $\delta(q_0, \omega) \in F$.

Языки, допускаемые конечными автоматами, часто называют *регулярными множествами*. Регулярные множества — одно из основных понятий теории конечных автоматов. Рассмотрим без доказательства некоторые интересные свойства регулярных множеств:

1) Если множества U и V регулярны, то $U \cup V$, $U \cap V$ и $U \setminus V$ также регулярны.

2) Семейство регулярных множеств в некотором алфавите Σ есть наименьшее семейство подмножеств множества Σ^* , содержащее все конечные подмножества, и замкнутое относительно операций их объединения, конкатенации (\cdot) и итерации $(*)$.

Операция *конкатенации* означает приписывание справа некоторой цепочки другой. Пусть имеются множества цепочек U и V . Тогда в множество $U \cdot V$ входят все такие цепочки $\omega = \xi_1 \xi_2$, что $\xi_1 \in U$ и $\xi_2 \in V$.

Операция *итерации* связана с конкатенацией: в итерацию множества цепочек U входят все такие цепочки, которые состоят из любого конечного числа цепочек из U . Таким образом, $U^* = \bigcup_{i=0}^{\infty} U^i$. Заме-

тим, что множество всех цепочек над алфавитом Σ ,

¹⁾ Здесь U^0 — множество $\{\lambda\}$, $U^i = U \cdot U \cdot \dots \cdot U$ (i раз).

которое мы обозначаем через Σ^* , можно понимать как итерацию множества Σ .

Перечисленные свойства регулярных множеств удобно использовать для доказательства регулярности множеств некоторых простых объектов языков программирования. Покажем, например, что множества идентификаторов и чисел языка АЛГОЛ-60 являются регулярными. Обозначив множество идентификаторов через I , множество букв через L и множество цифр через D , имеем следующую формулу: $I = L \cdot (L \cup D)^*$. Множества L и D регулярны вследствие своей конечности. Следовательно, регулярно и множество I .

Для доказательства регулярности множества чисел введем следующие обозначения: $S = \{+, -\}$, I_n — множество целых чисел без знака; N — множество, состоящее из всех чисел языка АЛГОЛ-60; если V — некоторое множество цепочек, то через $[V]$ будем обозначать любой элемент этого множества или пустую цепочку. Нетрудно проверить, что в указанных обозначениях множество N может быть представлено следующей формулой:

$$N = [S] \cdot [I_n] \cdot [\cdot \cdot I_n] \cdot [_{10} \cdot [S] \cdot I_n] \setminus S \setminus \{\lambda\},$$

где $I_n = D \cdot D^*$. Так как множества S и D конечны, а символы \cdot и $_{10}$ могут рассматриваться как одноэлементные множества, данная формула доказывает регулярность множеств целых чисел без знака и всех чисел языка АЛГОЛ-60.

Определенные нами выше конечные автоматы являются детерминированными машинами Тьюринга с входной лентой. Рассмотрим теперь класс недетерминированных конечных автоматов, который нам понадобится для доказательства совпадения классов регулярных множеств и языков, порождаемых модифицированными автоматными грамматиками. Определение недетерминированных конечных автоматов почти полностью совпадает с определением конечных автоматов, приведенным выше. Единственное различие заключается в функции δ , которая в данном случае рассматривается как отображение множества $K \times \Sigma$ в множество всех подмножеств множества K , т. е. в 2^K . Такое определение δ означает, что при работе авто-

мата $оп$, находясь в некотором состоянии и читая символ входной цепочки, может перейти в любое из нескольких состояний, входящих в соответствующее значение функции δ . Аналогично случаю детерминированных конечных автоматов функция δ может быть доопределена до отображения $K \times \Sigma^*$ в 2^K . Говорят, что недетерминированный конечный автомат допускает цепочку $\omega \in \Sigma$, если $\delta(q_0, \omega) \cap F \neq \emptyset$, т. е. найдется такой вариант анализа, при котором автомат переходит в одно из заключительных состояний после прочтения последнего символа входной цепочки ω .

В теории автоматов доказывается следующее утверждение: *классы языков, распознаваемых детерминированными и недетерминированными автоматами, совпадают.*

Перейдем к рассмотрению вопроса о соотношении конечных автоматов и модифицированных автоматных грамматик.

Теорема 5. *Класс языков, распознаваемых конечными автоматами, совпадает с классом модифицированных автоматных языков.*

Доказательство. Идея доказательства этой теоремы чрезвычайно проста. При рассмотрении модифицированных автоматных грамматик нами было показано, что каждая такая грамматика может быть представлена в виде графа, у которого вершины соответствуют нетерминальным символам, а дуги помечаются терминальными символами грамматики. Точно таким же графом может быть представлен и любой конечный автомат. Для этого достаточно поставить в соответствие каждому состоянию автомата вершину графа и затем провести дуги, помеченные входными символами, исходя из следующего правила: вершины q_1 и q_2 соединяются дугой, помеченной символом a в том и только том случае, если $q_2 \in \delta(q_1, a)$ в случае недетерминированного или $q_2 = \delta(q_1, a)$ в случае детерминированного автомата. Для завершения построения графа нужно пометить вершины, соответствующие начальному и заключительным состояниям автомата. Нетрудно видеть, что между графами указанной структуры и конечными автоматами существует взаимно однозначное (с точностью до расположения вершин графов) соответ-

вне, а процесс распознавания в точности совпадает с тем, который был описан в § 4 гл. I при рассмотрении графического представления модифицированных автоматных грамматик.

Итак, для построения по модифицированной автоматной грамматике конечного автомата, распознающего язык, порождаемый этой грамматикой, достаточно представить грамматику в виде графа и затем с помощью очевидной процедуры определить конечный автомат, представленный построенным графом. При этом множество нетерминальных символов грамматики будет играть роль множества состояний автомата, терминальные символы будут рассматриваться как входные, каждое правило грамматики вида $A \rightarrow aB$ определит B как одно из значений $\delta(A, a)$, начальный символ грамматики перейдет в начальное состояние автомата, а при наличии правила вида $A \rightarrow \lambda$ состояние A окажется заключительным.

Обратная процедура построения по конечному автомату модифицированной автоматной грамматики столь же очевидна. Поэтому доказательство теоремы можно считать законченным.

Как уже упоминалось выше, конечные автоматы могут использоваться в начальной стадии трансляции для выделения в программе некоторых простых конструкций (идентификаторов, чисел). Синтаксический анализ более сложных конструкций языков программирования и программ в целом может осуществляться с помощью автоматов с магазинной памятью, которые рассматриваются в следующем параграфе.

§ 3. Автоматы с магазинной памятью

Автоматы и преобразователи с магазинной памятью, неформальное описание которых имеется в § 1 настоящей главы, играют важную роль в теории языков программирования. Именно они или их модификации используются в большинстве практически работающих трансляторов для синтаксического анализа программ.

Рассмотрим формальное определение автоматов с магазинной памятью.

Определение. Автоматом с магазинной памятью, или МП-автоматом, называется следующая семерка объектов:

$$M = \langle K, \Sigma, H, \delta, Z_0, q_0, F \rangle,$$

где K — конечное множество состояний; Σ — конечное множество входных символов; H — конечное множество символов магазинной памяти; δ — отображение множества $K \times \Sigma \times H$ в множество $K \times H^*$, q_0 — выделенный элемент из K (начальное состояние); Z_0 — выделенный элемент из H (граничный маркер); F — подмножество заключительных состояний, $F \subseteq K$.

Функционирование МП-автомата определяется следующим образом. Пусть автомат находится в некотором состоянии q , на входной ленте записана непустая цепочка входных символов ω , а в магазинной памяти — непустая цепочка символов магазинной памяти α . Эту информацию, определяющую работу автомата, будем называть его конфигурацией и обозначать тройкой соответствующих объектов: (q, ω, α) . Говорят, что автомат переходит из конфигурации $(q_1, \omega_1, \alpha_1)$ в конфигурацию $(q_2, \omega_2, \alpha_2)$ за один такт работы, если $\omega_1 = \alpha \omega_2$, $\alpha_1 = \alpha Z$, $\alpha_2 = \alpha \gamma$, где $a \in \Sigma$, $\alpha, \gamma \in H^*$, $Z \in H$, $\delta(q_1, a, Z) = (q_2, \gamma)$. При этом удобно считать, что такт работы автомата зависит от символа, находящегося в самой левой ячейке входной ленты, и при выполнении такта цепочка во входной ленте сдвигается на одну ячейку влево, так что левый символ выходит за пределы ленты. Переход автомата из конфигурации k_1 в конфигурацию k_2 за один такт работы обозначается формулой $k_1 \vdash k_2$, причем под знаком \vdash при необходимости указывается обозначение автомата. Когда автомат из конфигурации k_1 переходит в конфигурацию k_n за несколько тактов работы, т. е. существует последовательность конфигураций k_1, k_2, \dots, k_n такая, что $k_i \vdash k_{i+1}$ при $i = 1, 2, \dots, n-1$, будем использовать обозначение $k_1 \vdash^* k_n$, также указывая при необходимости обозначение автомата под знаком \vdash .

Будем говорить, что цепочка ω входных символов автомата *допускается* автоматом, если $\omega = \lambda$ и $q_0 \in F$, или если $(q_0, \omega, Z_0) \vdash^* (f, \lambda, \gamma)$, где $f \in F$. Наконец,

языком, допускаемым автоматом, будем считать множество всех допускаемых им цепочек входных символов. Язык, допускаемый автоматом M , будем обозначать через $L(M)$.

Так как функция δ рассмотренных автоматов принимает значения из множества $K \times H^*$, эти автоматы являются детерминированными, т. е. каждый такт работы таких автоматов однозначно определяется конфигурациями, в которых они находятся.

Определение МП-автоматов можно несколько обобщить. Во-первых, изменив область значений функции δ , можно определить недетерминированные МП-автоматы. Недетерминированным МП-автоматом будем называть такой автомат, в котором функция δ определена как отображение множества $K \times \Sigma \times H$ в множество $2^{K \times H^*}$. Значениями δ в этом случае будут подмножества множества $K \times H^*$, а отношение $(q_1, a\omega, \alpha Z) \vdash (q_2, \omega, \alpha \gamma)$ имеет место, если $(q_2, \gamma) \in \delta(q_1, a, Z)$. Таким образом, недетерминированный автомат допускает цепочку ω , если, начиная работу в конфигурации (q_0, ω, Z_0) , он при удачном выборе тактов переходит в конфигурацию (f, λ, γ) .

Можно показать, что классы языков, допускаемых недетерминированными и детерминированными МП-автоматами, различны. В то время как первый класс совпадает с классом УКС-языков (этот факт мы рассмотрим подробнее ниже), детерминированные МП-автоматы допускают лишь языки, не являющиеся существенно неоднозначными. Для синтаксического анализа значительно удобнее использовать детерминированные автоматы, которые позволяют за один просмотр анализируемой цепочки определить ее принадлежность языку. При моделировании на ЭВМ недетерминированного автомата встает задача перебора вариантов работы автомата. Организация такого перебора связана как с дополнительными расходами памяти (так как требуется запоминать уже пройденные конфигурации автомата с тем, чтобы начинать с них другие варианты работы), так и с увеличением машинного времени.

Второй способ обобщения определения МП-автоматов заключается во введении такого режима работы автомата, при котором он меняет состояние и про-

изводит изменения в магазинной памяти, не читая символ на входной ленте. Формально в этом случае нужно изменить область задания функции δ , определив ее как отображение множества $K \times (\Sigma \cup \{\varepsilon\}) \times H$ в множество $K \times H^*$ (или в случае обобщения недетерминированного автомата, в множество $2^{K \times H^*}$)¹⁾. Определение языков, допускаемых такими *обобщенными МП-автоматами*, не изменяется. Можно показать, и это будет сделано ниже, что классы языков, распознаваемых обобщенными недетерминированными автоматами и недетерминированными автоматами, совпадают.

Обобщенные МП-автоматы используются для доказательства эквивалентности классов УКС-языков и языков, допускаемых МП-автоматами. Они удобны также для доказательства принадлежности к этим классам некоторых конкретных языков. С другой стороны, обобщенные МП-автоматы трудно применять для анализа. Способность этих автоматов «читать» пустой символ на входной ленте, не уменьшая длину записанной на ней цепочки, может привести к заикливанию, при котором автомат будет работать без чтения символов входной ленты неопределенно долгое время. В связи с этим затруднительно оценить число тактов обобщенного автомата, необходимых для распознавания конкретных цепочек.

Приведем примеры МП-автоматов.

Пример 1. Рассмотрим язык чисел АЛГОЛа-60, порождающая грамматика которого имеется в § 2 гл. I. Для этого языка построим детерминированный МП-автомат, имеющий 2-состояния: q_0 и q_1 , из которых q_0 — начальное, а q_1 — заключительное, и 5 символов магазинной памяти: Z_0, A, B, C, D . Содержательно символ Z_0 будем играть роль граничного маркера, A — означать, что анализируется целая часть десятичного числа (возможно, пустая); B — использоваться при анализе дробной части десятичного числа, D — порядка. C — символ, используемый при анализе возможного знака порядка. В магазинной памяти всегда будет записан лишь один из этих символов. Так как число заканчивается цифрой, автомат после чтения цифры всегда будет находиться в состоянии q_1 , а после чтения любого другого символа, который может использоваться в числе, — в состоянии q_0 . Приведем таблицу функции δ , которая после сделанных замечаний не требует особых пояснений.

¹⁾ ε — пустой символ.

1. $\delta(q_0, +, Z_0) = (q_0, A)$; 11. $\delta(q_0, -, C) = (q_0, D)$;
2. $\delta(q_0, -, Z_0) = (q_0, A)$; 12. $\delta(q_0, d, C) = (q_1, D)$;
3. $\delta(q_0, ., Z_0) = (q_0, B)$; 13. $\delta(q_0, d, D) = (q_1, D)$;
4. $\delta(q_0, 1_0, Z_0) = (q_0, C)$; 14. $\delta(q_1, ., A) = (q_0, B)$;
5. $\delta(q_0, d, Z_0) = (q_1, A)$; 15. $\delta(q_1, 1_0, A) = (q_0, C)$;
6. $\delta(q_0, ., A) = (q_0, B)$; 16. $\delta(q_1, d, A) = (q_1, A)$;
7. $\delta(q_0, 1_0, A) = (q_0, C)$; 17. $\delta(q_1, 1_0, B) = (q_0, C)$;
8. $\delta(q_0, d, A) = (q_1, A)$; 18. $\delta(q_1, d, B) = (q_1, B)$;
9. $\delta(q_0, d, B) = (q_1, B)$; 19. $\delta(q_1, d, D) = (q_1, D)$.
10. $\delta(q_0, +, C) = (q_0, D)$;

Здесь d означает любую цифру, и соответствующие строки таблицы определяют, таким образом, по 10 значений функции δ . Как и в большинстве других случаев, в данном примере функция δ является частично определенной. Будем считать, что если в процессе работы автомата встретилась комбинация состояния и символов, для которой значение δ не определено, автомат прекращает работу, а анализируемая цепочка не входит в язык, распознаваемый автоматом. Такое допущение не является принципиальным, так как частично определенную функцию δ можно всегда тривиальным образом доопределить, не меняя распознаваемый автоматом язык.

Рассмотрим анализ с помощью описанного автомата цепочек основных символов языка чисел, выписывая соответствующие последовательности конфигураций автомата. Число $+13.2_{10} - 4$ анализируется следующим образом:

$(q_0, +13.2_{10} - 4, Z_0)$, $(q_0, 13.2_{10} - 4, A)$, $(q_1, 3.2_{10} - 4, A)$, $(q_1, 2_{10} - 4, A)$, $(q_0, 2_{10} - 4, B)$, $(q_1, 1_0 - 4, B)$, $(q_0, -4, C)$, $(q_0, 4, D)$, (q_1, λ, D) .

Так как автомат находится в заключительном состоянии, анализ прошел успешно, и входная цепочка принадлежит языку.

Приведем пример анализа неправильной цепочки 45.1_02 .

$(q_0, 45.1_02, Z_0)$, $(q_1, 5.1_02, A)$, $(q_1, 1_02, A)$, $(q_0, 1_02, B)$.

Значение δ от $(q_0, 1_0, B)$ не определено, поэтому продолжение анализа невозможно, и цепочка не входит в язык чисел.

Отметим, что, как было показано ранее, множество чисел языка АЛГОЛ-60 является регулярным. Поэтому существует распознающий это множество конечный автомат. Этот автомат может быть получен из приведенного МП-автомата, для чего нужно поставить в соответствие каждой возможной комбинации состояния и символа магазинной памяти МП-автомата состояние конечного автомата и преобразовать соответствующим образом таблицу функции δ . Так как в магазинной памяти МП-автомата всегда содержится лишь один символ, полученный таким образом конечный автомат будет распознавать тот же язык.

Пример 2. Пусть имеется некоторый детерминированный конечный автомат $A = \langle K, \Sigma, \delta, q_0, F \rangle$, $\Sigma = \{a_1, a_2, \dots, a_n\}$, распознающий язык L . Введем в словарь основных символов языка два новых символа: $\Sigma' = \{a_1, \dots, a_n\} \cup \{[,]\}$ и образуем новый

язык L' следующим образом: если $\omega \in L$, то ω' , образованная вставкой в ω любого количества правильно расположенных пар квадратных скобок, входит в L' . В L' входят только цепочки указанного вида. Скобки будем считать правильно расположенными, если выполняются два обычных условия использования скобок в арифметических выражениях:

1. Если α — любая левая подцепочка цепочки ω' : $\omega' = \alpha\gamma$, то число левых скобок в α не меньше числа правых скобок.

2. Число левых и число правых скобок во всей цепочке ω' равны.

Построим детерминированный МП-автомат M , распознающий язык L' . Очевидно, этот автомат может работать аналогично автомату A , но должен в то же время проверять правильность расположения скобок. Для этой проверки будем использовать магазинную память. Включив в H автомата M три символа: $H = \{Z_0, Z_1, Z_2\}$, определим работу автомата так, чтобы он, встречая левую скобку, приписывал в магазин Z_1 , если крайний правый символ в магазине Z_0 , и Z_2 в противном случае, а встречая правую скобку, стирал один из этих двух символов. Автомат должен переходить в заключительное состояние в одном из следующих случаев:

1) если при анализе правой скобки автомат находится в состоянии, соответствующем заключительному состоянию автомата A , и правый символ в магазине — Z_1 ;

2) если при анализе символа из Σ M переходит в состояние, соответствующее заключительному состоянию автомата A , и в магазинной памяти находится символ Z_0 .

Из сделанных замечаний видно, что автомат M должен иметь все состояния автомата A , а также группу «двойников» заключительных состояний автомата A . Состояния этой группы будем обозначать соответствующими им состояниями из F с чертой сверху, например, \bar{q} — двойник состояния q . Множество надчеркнутых состояний обозначим через \bar{F} . Надчеркнутые состояния будем использовать в том случае, когда состояние M , соответствующее заключительному состоянию автомата A , не должно быть заключительным.

Итак, определим искомый МП-автомат:

$$M = \langle K \cup \bar{F}, \Sigma \cup \{[,]\}, \{Z_0, Z_1, Z_2\}, \delta', Z_0, q_0, F \rangle.$$

δ' определяется по функции δ следующим образом:

1. $\delta'(q, a_i, Z_0) = (p, Z_0)$, если $\delta(q, a_i) = p, q \in K$;
2. $\delta'(q, a_i, Z) = (p, Z)$, если $\delta(q, a_i) = p \notin F, q \in K \setminus F$;
3. $\delta'(q, a_i, Z) = (\bar{p}, Z)$, если $\delta(q, a_i) = p \in F, q \in K \setminus F$;
4. $\delta'(\bar{q}, a_i, Z) = (p, Z)$, если $\delta(q, a_i) = p \notin F, q \in F$;
5. $\delta'(\bar{q}, a_i, Z) = (\bar{p}, Z)$, если $\delta(q, a_i) = p \in F, q \in F$;
6. $\delta'(q, [, Z_0) = (q, Z_0 Z_1)$, $q \in K \setminus F$;
7. $\delta'(q, [, Z_0) = (\bar{q}, Z_0 Z_1)$, $q \in F$;
8. $\delta'(q, [, Z) = (q, ZZ_2)$, $q \in (K \setminus F) \cup \bar{F}$;

$$9. \delta'(q,], Z_1) = (q, \lambda), \quad q \in K \setminus F;$$

$$10. \delta'(\bar{q},], Z_1) = (q, \lambda), \quad q \in F;$$

$$11. \delta'(q,], Z_2) = (q, \lambda), \quad q \in (K \setminus F) \cup \bar{F}.$$

В формулах 2—5 и 8 Z может принимать значения Z_1 или Z_2 . Рассмотрим различные случаи анализа автоматом M входных цепочек. Пусть $\omega' \in L'$. Тогда ω' может заканчиваться либо символом из Σ , либо правой скобкой. Пусть имеет место первый вариант. Воспринимая последний символ входной цепочки, автомат перейдет в состояние, определяемое формулой 1. А так как $\omega' \in L$, это состояние окажется заключительным. Если цепочка заканчивается одной или несколькими левыми и правыми скобками, M при анализе последнего символа из Σ перейдет в некоторое состояние из F . Затем его работа будет определяться формулами 7, 8, 11 и 10, причем на последнем такте работы будет использоваться формула 10. Следовательно, и в этом случае автомат перейдет в заключительное состояние. Теперь посмотрим, как будет проходить анализ цепочек, не входящих в язык L' . Если $\omega' \notin L'$ и $\omega \notin L$, то автомат A , анализируя ω , может либо оказаться в тупиковой ситуации, при которой его дальнейшая работа не определена, а тогда, очевидно, и M , анализируя ω' , окажется в тупиковой ситуации, либо по окончании работы A окажется в незаключительном состоянии. В последнем случае M перейдет в соответствующее незаключительное состояние после восприятия последнего символа цепочки ω' из Σ , и это состояние не изменится при просмотре возможных левых и правых скобок (см. правила 6, 8, 9 и 11). Итак, цепочка ω' не допускается автоматом M . Наконец, рассмотрим цепочки, не входящие в L' из-за неправильного использования скобок, т. е. случай, когда $\omega' \notin L'$, но $\omega \in L$. Если в некоторый момент просмотра цепочки ω' правых скобок окажется больше, чем левых, то автомат M окажется в тупиковой ситуации, так как функция δ' от $(q,], Z_0)$ не определена. Если же в ω' левых скобок больше, чем правых, то по окончании просмотра цепочки M будет иметь в магазинной памяти символ Z_1 и, возможно, символы Z_2 . А в этом случае, как видно из формул, автомат M не может находиться в заключительном состоянии.

Приведенные соображения показывают, что автомат M действительно допускает язык L' .

Пример 3. Построим недетерминированный МП-автомат, допускающий язык L_7 , порождающая грамматика которого приведена в примере 7, § 2 гл. I. В L_7 входят все такие цепочки $x_1 b x_2$, что $x_1, x_2 \in \{a_1, a_2, \dots, a_n\}^*$ и $x_1 \neq x_2$. Обратим внимание на следующую особенность языка L_7 : для того, чтобы цепочка указанного вида входила в язык, достаточно, чтобы $l(x_1) \neq l(x_2)$, или в случае равенства длин при посимвольном сравнении цепочек x_1 и x_2 должна найтись хотя бы одна пара несовпадающих символов. Неоднозначную функцию δ определим таким образом, чтобы она задавала два возможных варианта работы автомата: при реализации первого варианта автомат будет допускать все цепочки с неравными по длине подцепочками x_1 и x_2 , а работая в режиме, определенном вторым вариантом, он будет допускать цепочки, в x_1 и x_2 которых имеются несовпадающие символы.

Определим сначала те значения δ , которые соответствуют первому варианту работы. Состояния автомата будем в этом случае обозначать буквой p с индексами, а символы магазинной памяти буквами B и C . Начальным состоянием и граничным маркером будем считать q_0 и Z_0 соответственно. Принцип работы автомата заключается в следующем. При просмотре первого символа подцепочки x_1 в магазинную память записывается символ B , а для каждого из последующих символов из x_1 — символ C . Как только входным символом окажется b , автомат меняет режим работы, переходя в состояние, являющееся заключительным, и стирая по одному символу C для каждого входного символа из подцепочки x_2 . В момент, когда правым символом в магазине оказывается B , состояние автомата меняется на незаключительное. Таким образом, при равной длине подцепочек автомат по окончании работы окажется в незаключительном состоянии. Если же $l(x_2) > l(x_1)$, автомат, просматривая оставшиеся входные символы, вновь переходит в заключительное состояние. Специальная строка таблицы δ отводится для случая, когда входная цепочка имеет вид bx_2 . Определим значения δ для указанного варианта работы:

- | | |
|--|---|
| 1. $(p_3, Z_0) \in \delta(q_0, b, Z_0)$; | 6. $(p_2, B) \in \delta(p_1, b, B)$; |
| 2. $(p_1, Z_0B) \in \delta(q_0, a_i, Z_0)$; | 7. $(p_2, \lambda) \in \delta(p_2, a_i, C)$; |
| 3. $(p_1, BC) \in \delta(p_1, a_i, B)$; | 8. $(p_3, \lambda) \in \delta(p_2, a_i, B)$; |
| 4. $(p_1, CC) \in \delta(p_1, a_i, C)$; | 9. $(p_2, Z_0) \in \delta(p_3, a_i, Z_0)$; |
| 5. $(p_2, C) \in \delta(p_1, b, C)$; | 10. $(p_2, Z_0) \in \delta(p_2, a_i, Z_0)$. |

Индекс i везде может принимать значения от 1 до n , заключительным является состояние p_2 .

Перейдем к описанию второго варианта работы автомата. Помимо граничного маркера нам понадобятся две группы символов магазинной памяти, по n символов в каждой. Обозначим их через Z_1, \dots, Z_n и A_1, \dots, A_n . Индекс здесь означает, какому входному символу соответствует тот или иной символ магазинной памяти. Работа автомата будет заключаться в следующем. Читая первый символ подцепочки x_1 , автомат записывает в магазин произвольно выбранный символ Z_i . Далее при просмотре каждого из последующих символов из x_1 в магазин записывается A с индексом, равным индексу выбранного Z_i . Этот процесс может оборваться, если очередной входной символ имеет тот же индекс. Переходя в этом случае в другое состояние, автомат просматривает оставшуюся часть подцепочки x_1 , не меняя содержимое магазина. После чтения символа b начинается следующий этап работы. Он заключается в стирании по одному A_i из магазина для каждого входного символа подцепочки x_2 . Как только правым символом в магазине окажется Z_i , происходит сравнение индексов этого и очередного входного символов. Заметим, что сравниваемый символ имеет тот же порядковый номер в подцепочке x_2 , что и символ из x_1 , на котором была прервана запись в магазин. Таким образом, несовпадение индексов означает, что соответствующие символы из x_1 и x_2 не совпадают. Дальнейшая работа автомата определяется только при несовпадении индексов и

заключается в переходе в заключительное состояние с просмотром оставшихся символов x_2 . Отдельный вариант значения δ предусмотрен для сравнения первых символов (при этом символы A_i в магазин не записываются).

Из сказанного видно, что при работе автомата в данном режиме возможно большое количество вариантов анализа входных цепочек. Недетерминированность возникает потому, что, во-первых, на первом такте в магазин может быть записан символ Z с любым индексом, а, во-вторых, процесс записи может быть прерван в любой момент, когда индексы входного символа и символов, записанных в магазине, окажутся равными. С другой стороны, очевидно, что если подцепочки x_1 и x_2 равной длины не совпадают, имеется успешный вариант анализа, который осуществляется, когда в магазин записываются символы с индексом, равным индексу одного из входных символов подцепочки x_1 , не совпадающего с соответствующим символом из x_2 , а процесс записи в магазин прерывается именно на данном входном символе из x_1 . Если же подцепочки равны, ни один из вариантов анализа не может быть успешным.

Приведем теперь таблицу функции δ , реализующую указанный процесс анализа, обозначая состояния автомата буквой q с индексами:

1. $(q_1, Z_i) \in \delta(q_0, a_i, Z_0), \quad i=1, \dots, n;$
2. $(q_1, Z_i) \in \delta(q_0, a_j, Z_0), \quad i, j=1, \dots, n;$
3. $(q_1, Z_i A_i) \in \delta(q_1, a_j, Z_i), \quad i, j=1, \dots, n;$
4. $(q_1, A_i A_i) \in \delta(q_1, a_j, A_i), \quad i, j=1, \dots, n;$
5. $(q_2, A_i A_i) \in \delta(q_1, a_i, A_i), \quad i=1, \dots, n;$
6. $(q_2, Z_i A_i) \in \delta(q_1, a_i, Z_i), \quad i=1, \dots, n;$
7. $(q_2, A_i) \in \delta(q_2, a_j, A_i), \quad i, j=1, \dots, n;$
8. $(q_2, Z_i) \in \delta(q_2, a_j, Z_i), \quad i, j=1, \dots, n;$
9. $(q_3, A_i) \in \delta(q_2, b, A_i), \quad i=1, \dots, n;$
10. $(q_3, Z_i) \in \delta(q_2, b, Z_i), \quad i=1, \dots, n;$
11. $(q_3, \lambda) \in \delta(q_3, a_j, A_i), \quad i, j=1, \dots, n;$
12. $(q_4, Z_0) \in \delta(q_3, a_j, Z_i), \quad i, j=1, \dots, n, i \neq j;$
13. $(q_4, Z_0) \in \delta(q_4, a_i, Z_0), \quad i=1, \dots, n;$

Заключительным здесь является состояние q_4 .

Рассмотрим вопрос о совпадении классов УКС-языков и языков, распознаваемых МП-автоматами. Доказательство совпадения должно состоять из двух частей. Во-первых, требуется показать, что по произвольной УКС-грамматике можно построить МП-автомат, допускающий соответствующий УКС-язык. Мы выполним это построение ниже при доказательстве леммы 8. Во-вторых, нужно показать и обратное: возможность построения по произвольному МП-автомату УКС-грамматики, порождающей язык, до-

пускаемый автоматом. Доказательство этого утверждения мы не будем приводить полностью, так как оно довольно громоздко. Кроме того, несомненно, что для разработки методов анализа языков программирования наиболее интересной является именно первая часть доказательства. Ниже мы воспользуемся обобщенными МП-автоматами, показав затем их эквивалентность обычным МП-автоматам.

Лемма 8. По любой УКС-грамматике можно построить обобщенный недетерминированный МП автомат, допускающий язык, порождаемый этой грамматикой.

Доказательство. Пусть имеется УКС-грамматика $G = \langle V_T, V_A, I, S \rangle$. Построим по этой грамматике следующий обобщенный МП-автомат M . Для каждого терминального или нетерминального символа грамматики выберем два новых символа, приняв для них следующие обозначения: если $\alpha \in V_T \cup V_A$, то α' и α'' — выбранные символы. Включим в множество символов магазинной памяти автомата все выбранные таким образом символы, а в качестве граничного маркера будем использовать I' . Для того, чтобы пояснить работу автомата M и указать способ его формального построения, введем следующие обозначения. Пусть $\omega \in (V_T \cup V_A)^*$. Цепочку, образованную из символов с двумя штрихами магазинной памяти автомата M , соответствующих символам из ω , обозначим через ω'' . Пусть ω — любая цепочка символов. Обозначим через ω_{in} так называемую инвертированную цепочку, т. е. такую, в которой символы из ω расположены в обратном порядке.

Пусть имеется цепочка $\omega \in L(G)$ с левосторонним выводом $\omega_0 = I, \omega_1, \dots, \omega_n = \omega$ в грамматике G . Анализ этой цепочки будет производиться автоматом M следующим образом. Если $\omega_1 = A\xi\alpha$, где $A \in V_A, \xi \in (V_T \cup V_A)^*$ и $\alpha \in V_T \cup V_A$, автомат, читая на входной ленте пустой символ, запишет в магазин цепочку $\alpha'\xi''inA''$. Если $\omega_1 = a\xi\alpha$, где $a \in V_T$, а ξ и α имеют тот же смысл, то автомат, читая на входной ленте символ a , запишет в магазин $\alpha'\xi''in$. Во всех этих случаях состояние автомата не меняется. Наконец, если $\omega_1 = a \in V_T$ или $\omega_1 = \lambda$, то, читая символ a или пустой символ соответственно, автомат запишет в магазинную память

пустую цепочку и перейдет в заключительное состояние. Заметим, что в грамматике G должно содержаться правило вывода $I \rightarrow \omega_1$. Дальнейшая работа автомата зависит от того, какому символу — терминальному или нетерминальному — соответствует правый символ в магазине и в какой модификации он находится. Если символ в магазине a'' , $a \in V_T$, то автомат стирает его, читая на входной ленте a . Если в магазинной памяти находится a' , автомат переходит, кроме того, в заключительное состояние. При наличии в магазине символа со штрихом, соответствующего нетерминальному символу грамматики, автомат работает так же, как на первом такте работы. Если же в магазине находится символ с двумя штрихами, соответствующий нетерминальному, то автомат производит аналогичные действия, записывая в магазин вместо a' символ a'' и не переходя в заключительное состояние.

Из сказанного видно, что осуществляя удачный вариант анализа, автомат как бы копирует левосторонний вывод анализируемой цепочки в магазине, представляя цепочки вывода в инвертированном виде и стирая символы, которые соответствуют терминальным символам этих цепочек, расположенным левее первого нетерминала. При этом самый левый символ в магазине всегда будет находиться в модификации со штрихом. Переход в заключительное состояние возможен только в случае, когда именно этот символ является тем, от которого зависит такт работы автомата, т. е. когда в магазине записан лишь один символ.

Конечно, помимо успешного варианта для большинства цепочек из $L(G)$ будут существовать и другие, при которых завершение анализа невозможно.

Так произойдет, например, если в магазине будут записаны символы, отвечающие не тому правилу, которое используется при выводе анализируемой цепочки, а другому правилу для того же нетерминального символа. Автомат в этом случае рано или поздно окажется в тупиковой ситуации. Если анализируемая цепочка не входит в язык, ее успешный анализ невозможен.

Перейдем к формальному заданию интересующего нас автомата:

$$M = \langle \{q_0, q_1\}, V_T, H' \cup H'', \delta, q_0, I', \{q_1\} \rangle,$$

где H' — множество символов магазинной памяти в модификации со штрихом, соответствующих терминальным и нетерминальным символам грамматики; H'' — то же для модификации с двумя штрихами.

Приведем правила задания функции δ , используя следующие обозначения: A и B — нетерминальные символы, a — терминальный символ, α — любой символ грамматики G , ξ — (возможно, пустая) цепочка над объединении словарей V_T и V_A .

1. $(q_0, \alpha' \xi_{in} B'') \in \delta(q_0, \varepsilon, A')$, если $A \rightarrow B \xi \alpha \in S$;
2. $(q_0, B') \in \delta(q_0, \varepsilon, A')$, если $A \rightarrow B \in S$;
3. $(q_1, \lambda) \in \delta(q_0, \varepsilon, A')$, если $A \rightarrow \lambda \in S$;
4. $(q_0, \alpha' \xi_{in}) \in \delta(q_0, a, A')$, если $A \rightarrow a \xi \alpha \in S$;
5. $(q_1, \lambda) \in \delta(q_0, a, A')$, если $A \rightarrow a \in S$;
6. $(q_0, \xi_{in} B'') \in \delta(q_0, \varepsilon, A'')$, если $A \rightarrow B \xi \in S$;
7. $(q_0, \lambda) \in \delta(q_0, \varepsilon, A'')$, если $A \rightarrow \lambda \in S$;
8. $(q_0, \xi_{in}) \in \delta(q_0, a, A'')$, если $A \rightarrow \alpha \in S$;
9. $(q_1, \lambda) \in \delta(q_0, a, a')$, если $a \in V_T$;
10. $(q_0, \lambda) \in \delta(q_0, a, a'')$, если $a \in V_T$.

Так как построенный автомат с магазинной памятью M , очевидно, распознает язык, порождаемый произвольной УКС-грамматикой G , доказательство леммы окончено.

Лемма 9. По любому обобщенному недетерминированному МП-автомату может быть построена УКС-грамматика, порождающая язык, допускаемый этим автоматом.

Эту лемму можно доказать следующим образом. Определим по любому МП-автомату M язык $Null(M)$. Будем считать, что $\omega \in Null(M)$ в том и только том

случае, если $(q_0, \omega, Z_0) \vdash_M^* (q, \lambda, \lambda)$, т. е. если существует

такой вариант работы автомата M , при котором он по окончании просмотра цепочки ω опустошает свою магазинную память. Заметим, что, если нас интересует лишь язык $Null(M)$, автомат M может не иметь заключительных состояний. Первый этап доказательства

леммы заключается в построении по исходному автомату M такого обобщенного недетерминированного МП-автомата N , что $Null(N) = L(M)$. Нетрудно показать, что такое построение всегда возможно.

Далее строится семейство грамматик, имеющих одинаковые множества терминальных символов, нетерминальных символов и правил вывода и отличающихся друг от друга лишь начальными символами. Нетерминальные символы этих грамматик могут быть условно обозначены тройками (p, Z, q) , где p и q — некоторые состояния автомата N , Z — некоторый символ его магазинной памяти. Для каждого такого нетерминального символа можно определить правила вывода таким образом, чтобы из него были выводимы лишь такие терминальные цепочки ω , для которых

$(p, \omega, Z) \vdash_N^* (q, \lambda, \lambda)$. Если $(q, \lambda) \in \delta(p, a, Z)$, то в S нужно включить правило вывода $(p, Z, q) \rightarrow a$; если же $(q, Z_1 Z_2 \dots Z_n) \in \delta(p, a, Z)$, то в S включаются правила $(p, Z, r_1) \rightarrow a(q, Z_n, r_n)(r_n, Z_{n-1}, r_{n-1}) \dots (r_2, Z_1, r_1)$ для каждой последовательности состояний r_1, r_2, \dots, r_n из K автомата N (здесь a может быть и пустым символом). Можно показать, что грамматика с указанными правилами вывода и начальным символом (q_0, Z_0, q) , где q_0 — начальное состояние и Z_0 — граничный маркер автомата N , будет порождать все цепочки ω , для которых $(q_0, \omega, Z_0) \vdash_N^* (q, \lambda, \lambda)$. Включим

в интересующее нас семейство грамматик все грамматики с начальными символами приведенного вида (их будет столько, сколько состояний имеется в автомате N). Очевидно, объединение языков, порождаемых этими грамматиками, равно языку $Null(N)$. Для завершения доказательства леммы остается доказать почти очевидное утверждение: объединение любого числа УКС-языков есть УКС-язык.

Полное доказательство леммы 9 имеется в [13].

Рассмотренные нами леммы 8 и 9 позволяют сформулировать следующую теорему, играющую большую роль в теории языков программирования:

Теорема 6. *Класс языков, распознаваемых обобщенными МП-автоматами, совпадает с классом УКС-языков.*

Выше было замечено, что обобщение определения недетерминированных МП-автоматов, связанное с тем, что на некоторых тактах работы допускается чтение пустого входного символа, не расширяет класс языков, распознаваемых автоматами. Рассмотрим доказательство этого утверждения.

Определение. *Правило вывода КС-грамматики вида $A \rightarrow a\xi$, где $a \in V_T$, $\xi \in (V_T \cup V_A)^*$, назовем левобуквенным. КС-грамматику, все правила вывода которой левобуквенные, назовем левобуквенной.*

Лемма 10. *По любой КС-грамматике может быть построена эквивалентная ей левобуквенная КС-грамматика.*

Доказательство. Пусть имеется произвольная КС-грамматика $G = \langle V_T, V_A, I, S \rangle$. Для каждого $A \in V_A$ грамматики G обозначим через L_A множество цепочек вида $\omega = a\xi$, $a \in V_T$, $\xi \in (V_T \cup V_A)^*$ таких, для каждой из которых существует последовательность цепочек $\omega_0 = A$, $\omega_1, \dots, \omega_n = \omega$ со следующим свойством: для любого $i < n$, $\omega_i = B\eta$, $B \in V_A$, $\omega_{i+1} = \xi\eta$ и $B \rightarrow \xi \in S$. Итак, в L_A входят все такие цепочки над объединением словарей грамматики G , которые начинаются с терминального символа и могут быть выведены в этой грамматике из A применением правил вывода к левым символам цепочек. Покажем, что существует левوليнейная грамматика, порождающая язык L_A . Терминальные символы этой грамматики должны включать в себя как терминальные, так и нетерминальные символы исходной грамматики G . Во множество нетерминальных символов включим все нетерминальные символы из V_A со штрихом: если $B \in V_A$, то B' — нетерминальный символ интересующей нас левوليнейной грамматики. В качестве начального символа выберем A' . Наконец, во множество правил вывода включим все правила вывода исходной грамматики G , преобразованные следующим образом:

1) если $B \rightarrow C\xi \in S$, C — нетерминальный символ грамматики G , то включим в левوليнейную грамматику правило $B' \rightarrow C'\xi$;

2) если $B \rightarrow a\xi \in S$, a — терминальный символ грамматики G , то в левوليнейную грамматику включим правило $B' \rightarrow a\xi$.

Построенная нами левوليнейная грамматика порождает язык L_A . Пусть $\omega \in L_A$ и, следовательно, в G существует вывод ω из A : $A, \omega_1, \dots, \omega_{n-1}, \omega$, в котором правила применяются к левым символам цепочек. Тогда в левوليнейной грамматике должен существовать вывод $A', \omega'_1, \dots, \omega'_{n-1}, \omega$, где ω'_i получена из ω_i приписыванием штриха к левому символу. Так же очевидно и обратное: если ω входит в язык, порождаемый левوليнейной грамматикой, то $\omega \in L_A$. Как было показано в § 4 гл. I, левوليнейные грамматики порождают автоматные языки. Поэтому L_A — автоматный язык и существует порождающая его автоматная грамматика.

Рассмотрим семейство Q автоматных грамматик, порождающих языки L_α для каждого $\alpha \in V_A$. Эти грамматики построим таким образом, чтобы их начальные символы не входили в правые части правил (любую автоматную грамматику можно преобразовать к этому виду, если выбрать новый нетерминальный символ, заменить в правых частях правил начальный символ на новый символ и включить во множество правил правила для нового символа с правыми частями, равными правым частям преобразованных правил для начального символа грамматики). Будем также считать, что нетерминальные символы в различных грамматиках из Q различны. Итак, в Q входят автоматные грамматики $G_\alpha = \langle V_{T\alpha}, V_{A\alpha}, I_\alpha, S_\alpha \rangle$, порождающие языки L_α . По семейству Q построим КС-грамматику $\hat{G} = \langle V_T, \hat{V}_A, I, \hat{S} \rangle$, в которой V_T и I совпадают с соответствующими объектами исходной грамматики G , $\hat{V}_A = \bigcup_{\alpha} (V_{A\alpha} \setminus \{I_\alpha\}) \cup V_A$, и в \hat{S} включены все правила грамматик семейства Q , в которых символы I_α заменены на α . Правила вывода грамматики \hat{G} можно представить в виде следующих шести схем:

- 1) $\alpha \rightarrow a,$ 4) $\delta \rightarrow \alpha,$
- 2) $\alpha \rightarrow a\beta,$ 5) $\delta \rightarrow a\gamma,$
- 3) $\delta \rightarrow a,$ 6) $\delta \rightarrow \alpha\gamma,$

где $a \in V_T$, $\alpha, \beta \in V_A$, $\beta, \gamma, \delta \in \hat{V}_A \setminus V_A$.

Покажем, что языки, порождаемые грамматиками G и \hat{G} , совпадают. Пусть имеется левосторонний вы-

вод цепочки $\omega \in L(G)$ в G . Представим этот вывод следующим образом: $I, \dots, \omega_{i1}, \dots, \omega_{i2}, \dots, \omega_{in} = \omega$, где цепочки $\omega_{i1}, \omega_{i2}, \dots, \omega_{in}$ получены в результате применения правил вида $A \rightarrow a\xi$, а остальные цепочки — в результате применения правил вида $A \rightarrow B\xi$, $\xi \in (V_T \cup \hat{V}_A)^*$. Так как $\omega_{i1} \in L_i$, имеет место выводимость $I \xRightarrow[G_i]{*} \omega_{i1}$, а, следовательно, и $I \xRightarrow[\hat{G}]{*} \omega_{i1}$. Представим ω_{i1} в виде $x A \xi$, где $x \in V_T^*$ и непуста, $\xi \in (V_T \cup V_A)^*$. Тогда цепочку ω_{i2} можно представить в виде $x \eta \xi$, где $\eta = a\xi_1$. Ясно, что $\eta \in L_A$ и поэтому $I_A \xRightarrow[G_A]{*} \eta$ и $A \xRightarrow[\hat{G}]{*} \eta$. Отсюда следует, что имеет место выводимость $I \xRightarrow[\hat{G}]{*} \omega_{i2}$. Продолжая подобным образом анализ следующих выделенных в выводе цепочек через n шагов докажем, что $I \xRightarrow[\hat{G}]{*} \omega$.

Пусть, наоборот, $\omega \in L(\hat{G})$. Рассмотрим правосторонний вывод этой цепочки в грамматике \hat{G} . На первом шаге вывода должно применяться правило первого типа (в этом случае вывод закончен и, очевидно, $\omega \in L(G)$) или правило второго типа. Далее, возможно, несколько раз применяются правила пятого или шестого типов, а затем правило третьего или четвертого типов. В этот момент окажется выведенной цепочка, не содержащая символов из $\hat{V}_A \setminus V_A$. Если в ней имеются нетерминальные символы (из V_A), к самому правому из них применяется правило первого или второго типов. Если применено правило первого типа, рассматриваем правый из оставшихся нетерминальных символов. В противном случае правым нетерминальным символом цепочки окажется символ из $V_A \setminus V_A$. Далее процесс вывода продолжается описанным образом. Выделим в выводе те цепочки, которые не содержат символов из $\hat{V}_A \setminus V_A$: $I, \dots, \omega_{i1}, \dots, \omega_{i2}, \dots, \omega_{in} = \omega$. Ясно, что $I \xRightarrow[G_i]{*} \omega_{i1}$ (здесь мы должны

вспомнить, что нетерминальные символы различных грамматик семейства Q различны и поэтому на дан-

ном этапе вывода могут применяться правила лишь из грамматики G_1), и поэтому $I \xRightarrow{*} \omega_{i1}$. Представим ω_{i1} в виде $\xi A x$, где как обычно, $\xi \in (V_T \cup V_A)^*$, $x \in V_T^*$ (возможно, пустая). Тогда ω_{i2} представима в виде $\xi \eta x$, $\eta \in (V_T \cup V_A)^*$ и, исходя из указанных выше соображений, должно иметь место $I_A \xRightarrow{*}_{G_A} \eta$, а, следовательно,

но, $A \xRightarrow{*}_G \eta$ и $I \xRightarrow{*}_G \omega_{i2}$. Продолжая анализ выделенных

цепочек вывода, через n шагов докажем, что $I \xRightarrow{*}_G \omega$.

Итак, языки, порождаемые грамматиками G и \bar{G} , совпадают. Преобразуем грамматику \bar{G} в грамматику $\tilde{G} = \langle V_T, \tilde{V}_A, I, \tilde{S} \rangle$, оставив без изменений множества терминальных и нетерминальных символов и начальный символ. Множество правил вывода образуем следующим образом. Включим в него все правила из \tilde{S} первого, второго, третьего и пятого типов. Вместо каждого правила четвертого или шестого типов включим в \tilde{S} группу правил, образованных из данного правила подстановкой вместо нетерминального символа α цепочек из правых частей всех правил для α в \tilde{S} . Иначе говоря, если имеется, например, правило $\delta \rightarrow \alpha \gamma$, то включим в \tilde{S} вместо него правила вида $\delta \rightarrow \alpha \beta \gamma$ и $\delta \rightarrow \alpha \gamma$ такие, что $\alpha \rightarrow \alpha \beta \in \tilde{S}$ и $\alpha \rightarrow \alpha \in \tilde{S}$. Очевидно, $L(\tilde{G}) = L(\bar{G})$ и грамматика \tilde{G} является левобуквенной.

Теорема 7. Классы языков, распознаваемых обобщенными недетерминированными и недетерминированными МП-автоматами, совпадают.

Доказательство. Для доказательства теоремы достаточно показать, что по любому обобщенному МП-автомату можно построить эквивалентный ему недетерминированный автомат. Пусть имеется некоторый обобщенный МП-автомат M . Построим УКС-грамматику, порождающую язык, распознаваемый автоматом M . Преобразуем эту грамматику в почти эквивалентную ей КС-грамматику, а по ней построим эквивалентную левобуквенную КС-грамматику. По последней грамматике построим МП-автомат N . Для его построения используем метод, описанный при доказательстве леммы 8. Так как N строится по левобуквенной КС-грамматике, то язык, распознаваемый N , совпадает с языком, распознаваемым M .

буквенной грамматике, из правил задания функции δ , приведенных в доказательстве леммы, мы должны будем использовать лишь правила 4, 5, 8, 9, 10. А все эти правила задают такой режим работы автомата, при котором на входной ленте читаются непустые символы. Поэтому N — необобщенный МП-автомат. Поскольку $L(N) = L(M) \setminus \{\lambda\}$, нам осталось рассмотреть вопрос о распознавании пустой цепочки. Если эта цепочка распознается исходным автоматом M (при этом начальный символ УКС-грамматики должен быть укорачивающим), преобразуем автомат N следующим образом. Включим в K новое состояние q_2 , которое будем считать начальным и одним из заключительных состояний. Функцию δ определим для состояния q_2 так, чтобы $\delta(q_2, a, I') = \delta(q_0, a, I')$. Преобразованный автомат N распознает язык $L(M)$. Доказательство теоремы закончено.

§ 4. Преобразователи с магазинной памятью

Как уже было отмечено в § 1 настоящей главы, важнейшим результатом синтаксического анализа программ следует считать не только ответ на вопрос, является ли данная программа правильной, но и информацию о синтаксической структуре программы, а также, возможно, другую информацию, требуемую на дальнейших этапах трансляции. Эту информацию можно, вообще говоря, извлечь, рассматривая работу МП-автомата, анализирующего интересующую нас программу. Это, однако, довольно трудоемкая операция, так как потребовалось бы, рассматривая последовательные конфигурации автомата, постепенно выписывать данные нужного вида. Поэтому было бы удобнее, если бы сам МП-автомат мог записывать эту информацию в процессе своей работы, действуя на основании заданных ему формальных правил. Как раз для этой цели и может служить добавочный вид памяти, включаемый в состав МП-автомата, — выходная лента. МП-автомат с выходной лентой, который называется МП-преобразователем, уже упоминался выше. На каждом такте работы МП-преобразователь, кроме обычных действий автомата, может записать на выходную ленту некоторую цепочку символов

из специального множества выходных символов. В настоящем параграфе мы рассмотрим строгое определение МП-преобразователя и приведем пример использования МП-преобразователя для анализа конструкций языков программирования.

Определение. Преобразователем с магазинной памятью (МП-преобразователем) называется следующая восьмерка объектов:

$$S = \langle K, \Sigma, H, \Delta, \mu, Z_0, q_0, F \rangle.$$

Здесь $K, \Sigma, H, Z_0, q_0, F$ имеют тот же смысл, что и при задании МП-автомата; Δ — конечное множество символов выходной ленты; μ — функция, отображающая множество $K \times \Sigma \times H$ в $K \times H \times \Delta$. Точно так же, как в случае МП-автоматов, можно определить недетерминированные или обобщенные МП-преобразователи.

Свойства МП-преобразователей аналогичны свойствам МП-автоматов. Поэтому мы ограничимся примером, показывающим, как может быть использована выходная лента.

Пример 1. Построим МП-преобразователь для анализа простых арифметических выражений языка АЛГОЛ-60, который в случае правильного выражения будет записывать на выходную ленту синтаксическое дерево вывода этого выражения в линейно-скобочной записи. В качестве порождающей грамматики мы будем использовать слегка измененную грамматику, которая задается формами Бекуса, описывающими конструкцию «простое арифметическое выражение». Для простоты исключим из рассмотрения указатели функций и переменные с индексами, считая, что первичные выражения могут быть лишь простыми переменными и числами без знака, а также простыми арифметическими выражениями в скобках. Будем, кроме того, считать, что простые переменные и числа уже выделены на первом просмотре программы и заменены кодами, которые можно рассматривать как терминальные символы грамматики. Упрощенная таким образом грамматика получается из грамматики, приведенной в примере 3, § 2, гл. I, заменой правил 8, 9 и 10 схемой правил $P \rightarrow \langle nr \rangle$, где $\langle nr \rangle$ может быть любым кодом, используемым для обозначения чисел и простых переменных.

Для лучшего соответствия между грамматикой и МП-преобразователем, распознающим простые арифметические выражения, исключим из грамматики так называемые леворекурсивные правила, т. е. такие, правые части которых начинаются с символов из своих левых частей. В нашей грамматике леворекурсивными являются правила 3, 5 и 7: $E \rightarrow EAT$, $T \rightarrow TOM$, $M \rightarrow M \uparrow P$. Для исключения этих правил используем обозначения, аналогичные уже введенным нами при рассмотрении регулярных множеств в

§ 2 настоящей главы: если A — нетерминальный символ, то $[A]$ — любая терминальная цепочка, выводимая из этого символа, или пустая цепочка; если ξ — цепочка над объединением терминального и нетерминального словарей, то $\{\xi\}^*$ — любая цепочка, входящая в итерацию множества терминальных цепочек, выводимых из ξ . С помощью этих обозначений правила для простых арифметических выражений, термов и множителей можно преобразовать, и мы получим следующую грамматику:

- | | |
|---|-----------------------------|
| 1) $E \rightarrow [A]T\{AT\}^*$, | 6) $O \rightarrow \times$, |
| 2) $T \rightarrow M\{OM\}^*$, | 7) $O \rightarrow /$, |
| 3) $M \rightarrow P\{\uparrow P\}^*$, | 8) $O \rightarrow \div$, |
| 4) $P \rightarrow \langle nr \rangle$, | 9) $A \rightarrow +$, |
| 5) $P \rightarrow (E)$, | 10) $A \rightarrow -$. |

Грамматика простых арифметических выражений в данном виде позволяет несколько упростить синтаксические деревья выводов, которые МП-преобразователь должен записывать на выходной ленте: в противоположность исходной грамматике часть арифметического выражения, находящаяся левее крайнего справа термина, не считается теперь арифметическим выражением. То же относится к термам и множителям.

Перейдем к описанию МП-преобразователя. Из грамматики хорошо видно, что основные символы простых арифметических выражений можно разбить на две группы: символы, начинающие синтаксические конструкции, и символы, ограничивающие их справа. К числу первых относятся левая скобка и коды переменных и чисел, которые, напомним, считаются основными символами. Левая скобка всегда начинает первичное выражение, представляющее собой простое арифметическое выражение в скобках. Другие первичные выражения начинаются кодами переменных или чисел. Кроме того, в зависимости от своего расположения эти символы могут начинать и другие, внешние по отношению к упомянутой конструкции. Так, если один из этих символов находится сразу после знака операции типа умножения, он начинает также конструкцию «множитель», а если перед ним находится знак $+$, — или левая скобка, то начинается и конструкция «терм». Знаки арифметических операций и правая скобка относятся ко второй группе символов. Каждый знак ограничивает первичное выражение, а также те конструкции, которые являются по отношению к нему внутренними, например, знак операции типа умножения — множитель, знак операции типа сложения — множитель и терм. Правая скобка ограничивает все конструкции от первичного до простого арифметического выражений, внутренние по отношению к этой скобке и парной к ней.

Построим МП-преобразователь таким образом, чтобы он работал в двух режимах: в режиме начала конструкций, если очередной символ входной цепочки относится к первой группе, и в режиме завершения конструкций в противном случае. При работе в первом режиме в магазин будут записываться символы, соответствующие начинающимся конструкциям в порядке от внешних конструкций к внутренним. На выходную ленту будут помещаться открывающие скобки и символы соответствующих конструкций,

а также прочитанный символ входной цепочки. При работе во втором режиме преобразователь будет стирать из магазина символы завершаемых конструкций, а на выходную ленту помещать символы этих конструкций, правые скобки и прочитанный входной символ. Учитывая, что в некоторых случаях завершается несколько конструкций, для чего требуется несколько тактов, преобразователь будет иногда работать без чтения входных символов.

Отметим также следующие особенности МП-преобразователя. Для учета левых и правых скобок каждая левая скобка из входной цепочки будет заноситься в магазин, а при чтении соответствующей правой скобки стираться. Знаки операций типа сложения могут не заканчивать никаких конструкций. Это происходит тогда, когда они являются левыми символами выражения или расположены непосредственно за левыми скобками. В соответствии с этим будем при помощи состояний преобразователя различать два случая: когда мы находимся в начале выражения и когда завершён анализ очередного термина. Учитывая, что правая скобка или код переменной или числа может оказаться последним символом выражения, необходимо предусмотреть особый вариант работы преобразователя, заключающийся в завершении всех конструкций вплоть до внешнего арифметического выражения и в переходе в заключительное состояние. В связи с этим вариантом нам придется конструкции «простое арифметическое выражение» поставить в соответствие два символа магазинной памяти — один для внешнего выражения (он может считаться граничным маркером) и второй для выражения в скобках. Если преобразователь, завершая конструкции, обнаружит в магазине второй из выбранных символов, то это будет означать, что во входном выражении еще не прочитана по крайней мере одна правая скобка и поэтому переход в заключительное состояние невозможен.

Рассмотрим теперь табл. 2, представляющую функцию μ МП-преобразователя с описанными свойствами. Так как круглые скобки являются основными символами арифметических выражений и должны помещаться на выходную ленту, границы конструкций на ней будем обозначать квадратными скобками. Индекс i , который будет встречаться в некоторых строках таблицы, везде может принимать значения 0 или 1.

Уточним смысл состояний и некоторых символов магазинной памяти определенного нами преобразователя. Состояние p_0 , которое является начальным, указывает, что преобразователь начинает разбор внешнего или внутреннего (находящегося в скобках) простого арифметического выражения, p_1 — это основное состояние, при котором преобразователь работает в режиме начала конструкций. Однако в случае, если требуется завершение лишь одной конструкции (так происходит при чтении символа \uparrow), состояние преобразователя не меняется. Если же требуется завершение нескольких конструкций и, следовательно, стирание из магазина нескольких символов, преобразователь переходит в другие состояния. Состояния p_{2i} ($i=0, 1, 2$) определяют режим завершения первичного выражения и множителя, а p_{3i} ($i=0, 1$) — также и термина. При этом индекс соответствует конкретному знаку операции, прочитанному на входной ленте, который после завершения конструкций должен быть записан на выходную ленту. В состоянии p_4 преобразователь переходит в случае, если на входной

ленте прочитана правая скобка. Это означает, что помимо завершения конструкций нужно убрать из магазина левую скобку и записать правую скобку на выходную ленту. Наконец, p_5 используется для проверки, может ли прочитанный символ быть последним в выражении. Эта проверка также связана с завершением конструкций и стиранием из магазина символов от P до E_0 . Если такое завершение оказалось возможным, преобразователь переходит в заключительное состояние p_6 . Преобразователь использует два варианта символа E в магазине: E_0 означает внешнее выражение, а E_1 — внутреннее.

Рассматриваемый преобразователь является обобщенным недетерминированным. Недетерминированность возникает из-за необходимости проверки конца выражения. В данном случае ее можно легко исключить, если рассматривать также символ, непосредственно следующий за арифметическим выражением в программе. Таким символом может быть точка с запятой, запятая, символы `end` и `else` и ряд других. Наличие символа, завершающего выражение, позволяет исключить переход в состояние p_5 при чтении пустого символа, заменив его переходом в это состояние при чтении завершающего символа. Нетрудно видеть, что такой измененный преобразователь окажется детерминированным. Другие значения функции μ от пустого входного символа не имеют существенного значения, так как они связаны лишь с необходимостью исключения из магазина нескольких символов.

Приведен пример анализа преобразователем правильного простого арифметического выражения. Для каждого такта работы будем указывать состояние, читаемый на данном такте входной символ, цепочку, записанную в магазине, и цепочку, помещенную на выходную ленту на предыдущем такте. Пусть анализируется выражение $a + b / (c - d) \uparrow 2$. Анализ этого выражения показан в табл. 3.

Посмотрим, как реагирует преобразователь на различные ошибки, которые могут быть допущены в простых арифметических выражениях. Пусть в выражении неправильно расставлены скобки. Если левых скобок больше, чем правых, то лишние левые скобки останутся в магазине и будут находиться правее символа E_0 . Следовательно, по окончании чтения выражения преобразователь не сможет перейти в заключительное состояние, так как значение μ от p_5 , e , E_1 не определено. Если при чтении входного выражения на некотором такте правых скобок окажется больше, чем левых, то преобразователь также придет в тупиковую ситуацию, так как значение μ от p_4 , e , E_0 не определено. Пусть в выражении имеются два знака арифметических операций подряд. После чтения первого из них правый символ в магазине будет одним из символов E_0 , E_1 , T или M . Поскольку значение μ от одного из этих символов и знака не определено ни для одного состояния, кроме p_0 , преобразователь вновь окажется в тупике. Наконец, если в выражении окажутся два следующих один за другим кода переменных или чисел, преобразователь окажется в тупике потому, что после чтения первого кода правым символом в магазине окажется P , а для этого символа и кода переменной или числа значение μ не определено ни для одного состояния. Заметим, что аргумент функции μ , для которого ее значение не определено, может использоваться для указания характера допущенной ошибки.

Таблица 2

Номер строки	Аргументы			Значения		
	Состояние	Входной символ	Символ в магазине	Состояние	Цепочка, помещаемая в магазин	Цепочка, помещаемая на выходную ленту
1	p_0	+или—	E_0	p_1	E_0	$[E+или [E—$
2	p_0	+или—	E_1	p_1	E_1	+или —
3	p_0	(E_0	p_0	$E_0TMP(E_1$	$[E[T[M[P([E$
4	p_0	(E_1	p_0	$E_1TMP(E_1$	$[T[M[P([E$
5	p_0	$\langle nr \rangle$	E_0	p_1	E_0TMP	$[E[T[M[P\langle nr \rangle$
6	p_0	$\langle nr \rangle$	E_1	p_1	E_1TMP	$[T[M[P\langle nr \rangle$
7	p_1	$\langle nr \rangle$	E_i	p_1	E_iTMP	$[T[M[P\langle nr \rangle$
8	p_1	(E_i	p_0	$E_iTMP(E_i$	$[T[M[P([E$
9	p_1	$\langle nr \rangle$	T	p_1	TMP	$[M[P\langle nr \rangle$
10	p_1	(T	p_0	$TMP(E_i$	$[M[P([E$
11	p_1	$\langle nr \rangle$	M	p_1	MP	$[P\langle nr \rangle$
12	p_1	(M	p_0	$MP(E_i$	$[P([E$
13	p_1	\uparrow	P	p_1	λ	$P]\uparrow$
14	p_1	\times	P	p_{20}	λ	$P]$
15	p_1	/	P	p_{21}	λ	$P]$
16	p_1	\div	P	p_{22}	λ	$P]$
17	p_1	+	P	p_{30}	λ	$P]$
18	p_1	—	P	p_{31}	λ	$P]$
19	p_1)	P	p_1	λ	$P]$
20	p_1	ε	P	p_5	λ	$P]$
21	p_{20}	ε	M	p_1	λ	$M]\times$
22	p_{21}	ε	M	p_1	λ	$M]/$
23	p_{22}	ε	M	p_1	λ	$M]\div$
24	p_{31}	ε	M	p_3	λ	$M]$
25	p_{30}	ε	T	p_1	λ	$T]+$
26	p_{31}	ε	T	p_1	λ	$T]-$
27	p_4	ε	M	p_6	λ	$M]$
28	p_4	ε	T	p_3	λ	$T]$
29	p_4	ε	E_1	p_3	λ	$E]$
30	p_4	ε	(p_1	λ)
31	p_5	ε	M	p_5	λ	$M]$
32	p_5	ε	T	p_5	λ	$T]$
33	p_5	ε	E_0	p_6	λ	$E]$

Таблица 3

№ такта	1	2	3	4	5	6	7	8	9	10
Состояние	p_0	p_1	p_{30}	p_{30}	p_1	p_1	p_{21}	p_1	p_0	p_1
Входной символ	a	+	ε	ε	b	/	ε	(c	—
Магазин	E_0	E_0TMR	E_0TM	E_0T	E_0	E_0TMR	E_0TM	E_0T	$E_0TMR(E_1)$	$E_0TMR(E_1TMR)$
Выходная почка	λ	$[E[T[M]P_0P]]$	M	M	$T] +$	$T[M]Pb$	$P]$	$M]/$	$[M[P[E$	$[T[M[Pc$

№ такта	11	12	13	14	15	16
Состояние	p_{31}	p_{31}	p_1	p_1	p_4	p_4
Входной символ	ε	ε	d)	ε	ε
Магазин	$E_0TMR(E_1TM)$	$E_0TMR(E_1T)$	$E_0TMR(E_1)$	$E_0TMR(E_1TMR)$	$E_0TMR(E_1TM)$	$E_0TMR(E_1T)$
Выходная почка	$P]$	$M]$	$T] -$	$[T[M[Pd$	$P]$	$M]$

№ такта	17	18	19	20	21	22	23	24	25
Состояние	p_4	p_4	p_1	p_1	p_1	p_5	p_5	p_5	p_6
Входной символ	ε	ε	\uparrow	2	ε	ε	ε	ε	λ
Магазин	$E_0TMR(E_1)$	$E_0TMR($	E_0TMR	E_0TM	E_0TMR	E_0TM	E_0T	E_0	λ
Выходная почка	$T]$	$E]$)	$P]\uparrow$	$[P_2$	$P]$	$M]$	$T]$	$E]$

Недетерминированные МП-преобразователи схожей структуры могут быть разработаны и для полной грамматики языка АЛГОЛ-60 и других языков программирования. Однако, в противоположность преобразователю, рассмотренному в примере, их нельзя легко привести к детерминированному виду. Мы уже рассматривали примеры синтаксически неоднозначных конструкций языка АЛГОЛ-60. Преобразователь, анализируя эти конструкции, должен выдавать несколько возможных вариантов деревьев вывода, что возможно только, если он является недетерминированным. Другой причиной служит то, что во многих языках различные конструкции могут иметь одинаковые начала. Простейшим примером из АЛГОЛа-60 являются любой помеченный оператор и непомеченный оператор присваивания: оба они могут начинаться с идентификатора, который в первом случае представляет метку, а во втором — переменную, которой присваивается значение.

ГЛАВА III

ПОСТРОЕНИЕ АНАЛИЗАТОРОВ КОНТЕКСТНО-СВОБОДНЫХ ЯЗЫКОВ ПО ПОРОЖДАЮЩИМ ГРАММАТИКАМ

В настоящей главе мы кратко рассмотрим один из наиболее важных и интересных вопросов теории языков программирования — методы автоматического построения синтаксических анализаторов по порождающим грамматикам. Иначе говоря, нас будут интересовать такие алгоритмы, которые, получая в качестве исходных данных порождающие грамматики, выдают в результате своей работы анализаторы порождаемых этими грамматиками языков.

Вообще говоря, один метод формального построения анализаторов УКС-языков в виде обобщенных недетерминированных МП-автоматов нами уже был рассмотрен в § 3 гл. II при доказательстве леммы 8. Однако этот метод предназначен лишь для теоретического доказательства данной леммы. Использование для анализа МП-автоматов, построенных по этому методу, сводится к перебору большого числа левосторонних выводов в соответствующих грамматиках. Поэтому он не имеет практической ценности. Нетрудно также описать более точно способ построения МП-преобразователей, который использовался нами в § 4 гл. II. Но этот способ также страдает серьезным недостатком: как уже было отмечено, МП-преобразователь может получиться недетерминированным, даже если он строится по однозначной грамматике.

Так как синтаксис языков программирования обычно задается в виде КС- или УКС-грамматик, мы

будем в первую очередь интересоваться пригодными к практическому использованию анализаторами для КС- и УКС-языков. В гл. IV мы рассмотрим также некоторые способы включения в формально задаваемый синтаксис контекстных условий и обсудим возможности построения анализаторов по соответствующим грамматикам с контекстными условиями.

В настоящее время предложено сравнительно много методов построения синтаксических анализаторов для КС- и УКС-языков. Синтаксические анализаторы, которые вырабатываются с помощью этих методов, можно разбить на два больших класса. К первому из них мы отнесем такие анализаторы, которые, читая входные цепочки, строят их деревья выводов в порождающих грамматиках, как говорят, *снизу вверх*, т. е. начиная с вершин, соответствующих наиболее мелким конструкциям, и кончая корнем (стандартное изображение дерева вывода имеет корень наверху). Те анализаторы, которые просматривают входные цепочки один раз слева направо или справа налево, являются анализаторами данного класса. К ним относятся, очевидно, и МП-автоматы. Исходя из требований практической применимости таких анализаторов, наиболее важным их семейством следует признать детерминированные анализаторы, читающие на каждом такте своей работы один непустой символ или несколько символов входной цепочки. Такие анализаторы, естественно, могут быть построены лишь для однозначных грамматик, а в большинстве случаев на грамматике налагаются еще и другие ограничения.

Ко второму классу анализаторов относятся те, которые строят деревья выводов, начиная с корня и вершин, соответствующих наиболее крупным синтаксическим конструкциям. Говорят, что такие анализаторы работают в режиме *сверху вниз*. Многие анализаторы второго класса просматривают входные цепочки многократно. При их реализации на электронных вычислительных машинах иногда оказывается удобным одновременно организовать просмотр цепочек слева направо и справа налево. Определенным недостатком таких анализаторов является то, что они в большинстве случаев оказываются недетерминированными.

Обычно анализаторы, работающие в режиме сверху вниз, требуют большего времени для работы, чем анализаторы первого класса. С другой стороны, анализаторы второго класса более универсальны — многие из них не требуют никаких ограничений на КС-грамматики.

Ниже мы рассмотрим три метода автоматического построения анализаторов по КС- и УКС-грамматикам. Анализаторы, получаемые с помощью первых двух методов, работают в режиме снизу вверх и могут быть представлены в виде детерминированных устройств, похожих на обобщенные детерминированные МП-автоматы. Анализаторы, которые строятся с помощью третьего метода, представляют собой алгоритмы упорядоченного перебора правил исходных грамматик с целью нахождения деревьев выводов входных цепочек сверху вниз.

§ 1. Анализаторы предшествования

Первый метод автоматического построения анализаторов, который будет нас интересовать, опирается на так называемые *отношения предшествования* [52, 69]. Это бинарные отношения, которые задаются на множествах символов КС-грамматик.

Пусть имеется КС-грамматика $G = \langle V_T, V_A, I, S \rangle$. Рассмотрим некоторую пару (α_1, α_2) символов из $V_T \cup V_A$.

Определения. 1) Будем считать, что для пары (α_1, α_2) в грамматике G выполняется отношение \equiv , если в S содержится правило вывода вида $A \rightarrow \xi_1 \alpha_1 \alpha_2 \xi_2$. В противном случае отношение \equiv для данной пары не выполняется.

2) Будем считать, что для пары (α_1, α_2) в грамматике выполняется отношение $< \cdot$, если в S содержится правило вывода вида $A \rightarrow \xi_1 \alpha_1 B \xi_2$ и из B выводима цепочка, начинающаяся с символа α_2 : $B \Rightarrow^* \alpha_2 \eta$. В противном случае отношение $< \cdot$ для данной пары не выполняется.

3) Будем считать, что для пары (α_1, α_2) в грамматике G выполняется отношение $\cdot >$, если справедливо хотя бы одно из следующих утверждений:

а) в S содержится правило вывода вида $A \rightarrow \xi_1 B \alpha_2 \xi_2$ и из B выводима цепочка, заканчивающаяся символом $\alpha_1: B \Rightarrow^* \eta \alpha_1$;

б) в S содержится правило вывода вида $A \rightarrow \xi_1 B C \xi_2$, причем из B выводима цепочка, заканчивающаяся символом α_1 , а из C — цепочка, начинающаяся с символа $\alpha_2: B \Rightarrow^* \eta \alpha_1$ и $C \Rightarrow^* \alpha_2 \xi$.

В противном случае отношение $\cdot >$ для данной пары не выполняется.

В этих определениях A, B, C — некоторые нетерминальные символы, ξ_1, ξ_2, η, ξ — цепочки из $(V_T \cup V_A)^*$.

Отношения предшествования обладают рядом свойств, которые могут использоваться при распознавании языков, порожаемых КС-грамматиками. Рассмотрим одно из них.

Лемма 11. Пусть даны приведенная КС-грамматика G и пара (α_1, α_2) символов из объединения терминального и нетерминального словарей этой грамматики. Для того, чтобы нашлась цепочка, выводимая в G из начального символа и содержащая вхождение подцепочки $\alpha_1 \alpha_2$, необходимо и достаточно, чтобы для пары (α_1, α_2) в G было выполнено хотя бы одно отношение предшествования.

Доказательство. Пусть $G = \langle V_T, V_A, I, S \rangle$ — приведенная КС-грамматика. Рассмотрим некоторую пару символов (α_1, α_2) , $\alpha_1, \alpha_2 \in V_T \cup V_A$, и предположим, что для нее выполняется хотя бы одно отношение предшествования. Покажем, что тогда в грамматике G из I выводима цепочка, содержащая подцепочку $\alpha_1 \alpha_2$. Пусть, например, $\alpha_1 \doteq \alpha_2$. По определению отношения \doteq в S содержится правило вывода вида $A \rightarrow \xi_1 \alpha_1 \alpha_2 \xi_2$. Так как G — приведенная грамматика, в ней существует вывод из I некоторой цепочки, содержащей $A: I \Rightarrow^* \eta_1 A \eta_2$. Следовательно, $I \Rightarrow^* \eta_1 \xi_1 \alpha_1 \alpha_2 \xi_2 \eta_2$. Так же просто можно доказать выводимость цепочек, содержащих $\alpha_1 \alpha_2$, в случаях, когда для данной пары выполняются остальные отношения предшествования.

Предположим теперь, что $\omega = \eta_1 \alpha_1 \alpha_2 \eta_2$ — цепочка, выводимая в G из начального символа, $\eta_1, \eta_2 \in (V_T \cup V_A)^*$. Покажем, что для пары (α_1, α_2) выполняется по крайней мере одно отношение предшествования. Рассмотрим дерево вывода цепочки ω . Его заключи-

тельные вершины помечены символами, входящими в ω , в том числе, возможно, нетерминальными. Рассмотрим пути, ведущие из корня дерева в заключительные вершины x_1 и x_2 , соответствующие выделенным вхождением символов α_1 и α_2 в ω . Поскольку эти пути начинаются в одной вершине, а заканчиваются в разных, найдется вершина y , в которой они разветвляются. Пусть эта вершина помечена символом A . Определим длины путей из данной вершины в интересующие нас заключительные вершины. Возможны следующие случаи:

а) Длины обоих путей равны 1. Следовательно, при выводе цепочки ω применялось правило вида $A \rightarrow \xi_1 \alpha_1 \alpha_2 \xi_2$, и поэтому $\alpha_1 = \alpha_2$.

б) Длина пути, ведущего в вершину x_1 , равна 1, а длина пути в вершину x_2 больше 1. Рассмотрим вершину z того же уровня, что и вершина x_1 , находящуюся на пути из y в x_2 . Пусть эта вершина помечена символом B . z должна быть соседней справа по отношению к вершине x_1 . Если бы это было не так, т. е. существовала бы вершина z_1 , того же уровня, такая, что $x_1 < z_1 < z$, то существовала бы и заключительная вершина x_3 , либо совпадающая с z_1 , либо следующая за ней, такая, что $x_1 < x_3 < x_2$. А это невозможно, так как выделенные нами вхождения символов α_1 и α_2 в цепочку ω образуют в ней подцепочку. Вершина x_2 является наименьшей заключительной, следующей за z , поскольку в противном случае вновь нашлась бы такая заключительная вершина x_3 , что $x_1 < x_3 < x_2$. Итак, при выводе цепочки ω применялось правило вида $A \rightarrow \xi_1 \alpha_1 B \xi_2$, а из B в грамматике G выводима цепочка, начинающаяся с символа α_2 . Поэтому $\alpha_1 < \cdot \alpha_2$.

в) Длина пути, ведущего в вершину x_1 , больше 1. С помощью аналогичных рассуждений можно показать, что в этом случае $\alpha_1 \cdot > \alpha_2$.

Таким образом, для пары (α_1, α_2) выполнено по крайней мере одно отношение предшествования. Доказательство леммы закончено.

Нетрудно видеть, что существуют такие КС-грамматики, для некоторых пар символов которых выполняются два или даже все три отношения предшествования.

Пример 1. Рассмотрим грамматику с правилами вывода:

- 1) $I \rightarrow ab$, 5) $B \rightarrow bD$,
- 2) $I \rightarrow aB$, 6) $C \rightarrow c$,
- 3) $I \rightarrow AB$, 7) $D \rightarrow d$.
- 4) $A \rightarrow Ca$,

Из правила 1) имеем $a \equiv b$, правила 2) и 5) показывают, что $a < b$ и, наконец, анализируя правила 3), 4) и 5), получаем $a > b$. Таким образом, для пары (a, b) выполнены все три отношения предшествования.

Определение. КС-грамматика называется *грамматикой предшествования*, если для каждой пары ее символов выполнено не более чем одно отношение предшествования.

Для языков, порождаемых грамматиками предшествования без правил вывода с одинаковыми правыми частями, можно предложить очень простой и эффективный детерминированный анализатор. Читая входную цепочку слева направо и учитывая для каждой пары ее соседних символов отношение предшествования, которое для нее выполняется, анализатор последовательно находит подцепочки, являющиеся правыми частями правил грамматики, и производит замены этих подцепочек символами из левых частей соответствующих правил. Такие замены будем в дальнейшем называть *свертками* подцепочек по правилам грамматики. Если цепочка входит в язык, порождаемый грамматикой, то после некоторого количества сверток анализатор преобразует ее в начальный символ этой грамматики.

Прежде чем описать анализатор указанного типа более точно, рассмотрим еще одно важное свойство отношений предшествования. Пусть имеется грамматика предшествования G . Введем в рассмотрение новый символ $\#$, не входящий в объединение терминального и нетерминального словарей грамматики G . Назовем его *маркером*. Расширим множество символов, на котором заданы отношения предшествования, включив в него маркер, и будем считать, что $\# < \cdot \alpha$ и $\alpha \cdot > \#$ для любого $\alpha \in V_T \cup V_A$. Цепочку вида $\# \omega \#$, где $\omega \in (V_T \cup V_A)^*$, назовем *маркированной цепочкой* ω . Пусть имеется некоторый вывод в грамматике G : $\Omega = \omega_0, \dots, \omega_n$. Последовательность цепочек $\Omega' = \omega_0, \dots, \omega_n$, где $\omega_i = \# \omega_i \#$, назовем *маркированным выводом* цепочки ω_n в грамматике G .

Лемма 12. Пусть дана грамматика предшествования G и правосторонний вывод цепочки из $L(G)$: $\Omega = \omega_0, \omega_1, \dots, \omega_n; \omega_n \in L(G)$. Образует какую-либо цепочку $\omega_j, 1 \leq j \leq n$. Выделим в ней первую слева подцепочку $\eta = \alpha_0 \alpha_1 \dots \alpha_k \alpha_{k+1}$, где α_0 и $\alpha_{k+1} \in V_T \cup V_A \cup \{\#\}$, $\alpha_l \in V_T \cup V_A$ при $1 \leq l \leq k$, такую, что $\alpha_0 < \cdot \alpha_1, \alpha_k \cdot > \alpha_{k+1}$, и в случае $k \geq 2, \alpha_l = \alpha_{l+1}$ при $1 \leq l \leq k-1$. Таким образом, $\omega_j = \xi_1 \eta \xi_2$. Тогда в грамматике G имеется правило вывода $A \rightarrow \alpha_1 \dots \alpha_k$ такое, что $\omega_{j-1} = \xi_1 \alpha_0 A \alpha_{k+1} \xi_2$.

Доказательство. Прежде всего заметим, что поскольку G — грамматика предшествования, из леммы 11 следует, что для каждой пары соседних символов цепочки ω_j выполнено одно и только одно отношение предшествования. При этом $\# < \cdot \beta$ и $\gamma \cdot > \#$, где β и γ — первый и последний символы цепочки ω_j . Следовательно, в ω_j должна найтись единственная подцепочка η с указанными в условии леммы свойствами. Обозначим $\tilde{\eta} = \alpha_1 \dots \alpha_k$ и покажем, что $\tilde{\eta}$ была порождена в выводе Ω в результате применения правила вида $A \rightarrow \tilde{\eta}$. Предположим, что это не так. Тогда могут иметь место лишь следующие случаи.

1) При $k \geq 2$ некоторая подцепочка $\xi = \alpha_p \dots \alpha_q$, входящая в $\tilde{\eta}$ и такая, что $l(\xi) < l(\tilde{\eta})$, порождена в выводе в результате применения правила вида $A \rightarrow \xi$, а остальные символы из $\tilde{\eta}$ порождены в результате применения других правил вывода. Пусть для определенности $p > 1$. Рассмотрим дерево вывода Ω и пути, ведущие из корня в вершины, соответствующие вхождению символов α_{p-1} и α_p в $\tilde{\eta}$. Найдем вершину, в которой эти пути разветвляются, и рассмотрим пути, ведущие из этой вершины в вершины, соответствующие α_{p-1} и α_p . Очевидно, длина по крайней мере одного из них больше 1. А тогда, как показано в лемме 11, для пары (α_{p-1}, α_p) должно выполняться одно из отношений $< \cdot$ или $\cdot >$. Так же можно показать, что и в случае $q < k$ для данной пары должно выполняться одно из этих отношений.

2) Пусть $\alpha_0 \neq \#$ и в выводе Ω найдется общий предок A вхождений символов α_0 и α_1 в η , к которому

применяется правило вида $A \rightarrow \xi_1 B \alpha_1 \xi_2$, где B — предок α_0 . Тогда по определению $\alpha_0 \cdot > \alpha_1$.

3) Пусть $\alpha_{k+1} \neq \#$ и в выводе Ω найдется общий предок вхождений символов α_k и α_{k+1} в η , к которому применяется правило вида $A \rightarrow \xi_1 \alpha_k B \xi_2$, где B — предок α_{k+1} . Тогда по определению $\alpha_k < \cdot \alpha_{k+1}$.

Таким образом, если имеет место любой из трех рассмотренных случаев, найдется пара символов, для которой в грамматике будут определены два различных отношения предшествования. Но в грамматике предшествования это невозможно. Следовательно, подцепочка $\tilde{\eta}$ была порождена в выводе Ω в результате применения правила вида $A \rightarrow \tilde{\eta}$.

Покажем теперь, что при свертке подцепочки с указанными свойствами в цепочке $\hat{\omega}_j, 1 \leq j \leq n$, по правилу, которое применялось в Ω для порождения этой подцепочки, мы получим цепочку $\hat{\omega}_{j-1}$. Пусть $j = n$, $\hat{\omega}_n = \xi_1 \alpha_0 \eta \alpha_{k+1} \xi_2$. После выполнения свертки получим цепочку $\hat{\omega}_{n-1} = \xi_1 \alpha_0 A \alpha_{k+1} \xi_2$. Нетрудно убедиться, что для пары (α_0, A) в грамматике G не может быть выполнено отношение $\cdot >$ и, следовательно, выделяемая для свертки подцепочка цепочки $\hat{\omega}_{n-1}$ либо включает выделенное вхождение символа A , либо находится правее него. Выполняя свертку выделенной подцепочки цепочки $\hat{\omega}_{n-1}$, образуем цепочку $\hat{\omega}_{n-2}$, в которой вхождение символа, полученное в результате свертки, является крайним правым вхождением нетерминального символа. Продолжая этот процесс, образуем последовательность цепочек $\hat{\omega}_n, \hat{\omega}_{n-1}, \dots, \hat{\omega}_1, \hat{\omega}_0$, причем в каждой из них вхождение символа, полученное в результате свертки в предыдущей цепочке последовательности, является крайним правым вхождением нетерминального символа. Так как, очевидно, $\hat{\omega}_0 = \omega_0$, то $\hat{\omega}_j = \omega_j$ при $j = 1, 2, \dots, n-1$. Лемма доказана.

Легко видеть, что, если в грамматике предшествования нет правил вывода с одинаковыми правыми частями, то указанный в доказательстве леммы 12 способ восстановления правостороннего вывода является детерминированным.

Приступим к более подробному описанию уже упоминавшегося анализатора для языков, порождаемых

грамматиками предшествования. Такой анализатор часто называют *анализатором предшествования*. Его можно представить в виде устройства, имеющего входную ленту и магазин и использующего в работе отношения предшествования и правила вывода грамматики. Как и в случае МП-автоматов, входную ленту и магазин удобно считать потенциально неограниченными в одну сторону (например, вправо). В начале работы на входной ленте размещается анализируемая цепочка с приписанным к ней справа маркером, а в магазине находится маркер. Анализатор пытается свернуть цепочку к начальному символу грамматики, действуя так, как указано в лемме 12. В процессе работы в магазине размещаются частично свернутые левые подцепочки входной цепочки, причем между теми их соседними символами, для которых выполнено отношение $< \cdot$, помещается этот знак; на входной ленте остаются еще неучтенные правые подцепочки входной цепочки (рис. 7). Заканчивая работу, анализатор допускает или отвергает входную цепочку.

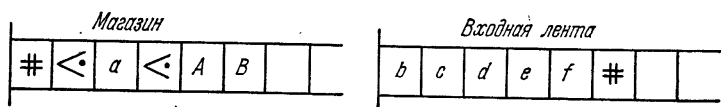


Рис. 7.

Такт работы анализатора начинается с определения отношения предшествования, которое выполняется для пары, образованной из правого символа магазина и левого символа входной цепочки. Возможны следующие варианты:

а) Ни одно из отношений не определено. Тогда входная цепочка отвергается.

б) Для пары определено отношение \neq или $< \cdot$. Тогда выполняется действие, которое мы назовем *сдвигом* и опишем ниже.

в) Для пары определено отношение $\cdot >$. Тогда анализатор пытается произвести свертку.

Осуществляя сдвиг, анализатор записывает в магазин символ $< \cdot$, если соответствующее отношение выполнено для данной пары, после чего переносит в магазин левый символ входной ленты, сдвигая остальную часть информации на ленте на одну ячейку влево.

Если требуется произвести свертку, анализатор работает следующим образом:

Шаг 1. Просматривает справа символы цепочки, записанной в магазине, пока не обнаружит символ $< \cdot$. Выделяет подцепочку, состоящую из просмотренных символов (без символа $< \cdot$). Переходит к шагу 2.

Шаг 2. Ищет правило вывода грамматики, правая часть которого совпадает с подцепочкой, выделенной на первом шаге. Если такое правило найдется, переходит к шагу 3, в противном случае — к шагу 5.

Шаг 3. Исключает из магазина найденные на первом шаге подцепочку и символ $< \cdot$. Образует пару, состоящую из символа, оказавшегося в магазине правым, и символа из левой части найденного на втором шаге правила. Переходит к шагу 4.

Шаг 4. Ищет отношение предшествования, выполненное для образованной на третьем шаге пары (как указывалось выше, это могут быть лишь отношения \equiv или $< \cdot$). Если ни одно из отношений для нее не выполнено, входная цепочка отвергается. В противном случае записывает в магазин символ $< \cdot$, если для пары выполнено данное отношение, и символ из левой части правила. После этого свертка заканчивается.

Шаг 5. Анализирует цепочки, записанные в магазине и на входной ленте. Если в магазине находится цепочка $\# < \cdot I$, где I — начальный символ грамматики, а на ленте остался лишь маркер, то входная цепочка допускается. В противном случае она отвергается.

Как следует из рассмотренных выше свойств отношений и грамматик предшествования, анализатор предшествования, работающий по грамматике без правил с одинаковыми правыми частями, допускает те и только те цепочки, которые принадлежат языку, порождаемому этой грамматикой, и является детерминированным. Для того, чтобы анализатор предшествования распознавал язык, порождаемый конкретной грамматикой, он должен быть снабжен информацией о правилах вывода этой грамматики и отношениях предшествования, заданных на множестве ее символов. Учитывая, что для каждой пары символов в грамматике предшествования может выполняться не более

чем одно отношение, их можно задать в виде квадратной матрицы, порядок которой равен числу терминальных и нетерминальных символов грамматики. Строки и столбцы матрицы помечаются различными символами грамматики. Элемент, расположенный в строке, помеченной символом α_1 , и столбце, помеченном символом α_2 , либо пуст, если для пары (α_1, α_2) не выполнено ни одно из отношений, либо представляет собой символ выполняемого для нее отношения. Для задания отношений $<\cdot$ и $\cdot>$ необходимо по каждому нетерминальному символу найти подмножества таких символов, с которых могут начинаться и которыми могут заканчиваться цепочки, порождаемые из этого символа. Процедура выделения таких подмножеств сравнительно проста. Например, первое из них можно получить, выполнив транзитивное замыкание бинарного отношения левой выводимости D_L , которое задается на множестве символов грамматики следующим образом: $\alpha_1 D_L \alpha_2$ в том и только том случае, если для α_1 в грамматике существует правило вывода, правая часть которого начинается с символа α_2 . Очевидно, в интересующее нас подмножество для символа A входят те и только те символы α_2 , для которых $AD_L \alpha_2$.

Пример 2. Рассмотрим грамматику со следующими правилами вывода:

- 1) $I \rightarrow a,$ 4) $T \rightarrow b,$
- 2) $I \rightarrow aT,$ 5) $T \rightarrow bT.$
- 3) $I \rightarrow (I),$

Матрица отношений предшествования для этой грамматики указана в табл. 4.

Приведем пример анализа входной цепочки $((abb))$ анализатором предшествования, работающим по данной грамматике:

$[\#, ((abb))\#] \text{сд} [\# < \cdot, ((abb))\#] \text{сд}$
 $[\# < \cdot (< \cdot, (abb))\#] \text{сд} [\# < \cdot (< \cdot (< \cdot a, bb))\#] \text{сд}$
 $[\# < \cdot (< \cdot (< \cdot a < \cdot b, b))\#] \text{сд} [\# < \cdot (< \cdot (< \cdot a < \cdot b < \cdot b,))\#] \text{сб4}$
 $[\# < \cdot (< \cdot (< \cdot a < \cdot bT,))\#] \text{сб5} [\# < \cdot (< \cdot (< \cdot aT,))\#] \text{сб2}$
 $[\# < \cdot (< \cdot (I,))\#] \text{сд} [\# < \cdot (< \cdot \{I\},)\#] \text{сб3}$
 $[\# < \cdot (I,)\#] \text{сд} [\# < \cdot (I,)\#] \text{сб3} [\# < \cdot I, \#] \text{— цепочка допущена.}$

Здесь в квадратных скобках через запятую указаны цепочки, находящиеся в магазине и на входной ленте, перед очередным тактом работы. sd означает, что на очередном такте был произведен сдвиг, а sbi означает свертку по i -му правилу.

Таблица 4

	I	T	a	b	$($	$)$
I						$\dot{=}$
T						$\dot{>}$
a		$\dot{=}$		$<\dot{\cdot}$		$\dot{>}$
b		$\dot{=}$		$<\dot{\cdot}$		$\dot{>}$
$($	$\dot{=}$		$<\dot{\cdot}$		$<\dot{\cdot}$	
$)$						$\dot{>}$

Применение анализаторов предшествования для распознавания языков программирования несколько осложняется вследствие того, что используемые для их описания грамматики не являются грамматиками предшествования.

Рассмотрим грамматику из примера 3, § 2, гл. I. Для нетерминальных символов E , T и M в ней имеются леворекурсивные правила вывода. В то же время эти символы входят в правые части других правил, занимая в них позиции, не являющиеся крайними левыми. Поэтому для ряда пар символов, например, для пары (A, T) выполняются отношения $\dot{=}$ и $<\dot{\cdot}$.

Известно, что любую КС-грамматику можно преобразовать в эквивалентную грамматику предшествования [65]. Для этого требуется включить в грамматику некоторое число новых нетерминальных символов, видоизменить имеющиеся в ней правила вывода, а также включить в нее ряд новых правил вида $A \rightarrow B$. В большинстве случаев среди новых правил будут встречаться правила с одинаковыми правыми

частями (отметим, что такие правила имеются и в исходных грамматиках, описывающих языки программирования, например, $\langle \text{идентификатор переменной} \rangle ::= \langle \text{идентификатор} \rangle$, $\langle \text{идентификатор процедуры} \rangle ::= \langle \text{идентификатор} \rangle$ и т. п.).

Итак, при использовании анализаторов предшествования для распознавания языков программирования грамматики, описывающие эти языки, нужно предварительно преобразовать в грамматики предшествования, а сами анализаторы следует модифицировать так, чтобы они могли работать с грамматиками, имеющими правила вывода с одинаковыми правыми частями.

§ 2. $LR(k)$ -анализаторы

Перейдем к следующему методу построения анализаторов, который обычно называется методом Кнута [60]. Анализаторы, которые строятся с помощью этого метода, носят название $LR(k)$ -анализаторов. Рассмотрим их определение и особенности.

$LR(k)$ -анализатор представляет собой устройство, состоящее из потенциально неограниченных вправо входной ленты и двух магазинов — верхнего и нижнего. На входной ленте помещается еще неучтенная правая подцепочка анализируемой цепочки, к которой справа приписываются k маркеров. В верхнем магазине помещаются цепочки, состоящие из символов состояний анализатора, в нижнем — цепочки над объединением множеств терминальных и нетерминальных символов грамматики, по которой построен анализатор. Читая входную цепочку слева направо, $LR(k)$ -анализатор пытается свернуть ее в начальный символ грамматики. В нижнем магазине находятся частично свернутые левые подцепочки входной цепочки, уже просмотренные анализатором (рис. 8). Состояния анализатора подбираются таким образом, чтобы они соответствовали возможным вариантам дерева вывода на различных тактах анализа входной цепочки.

На каждом такте работы $LR(k)$ -анализатор может выполнить одно из двух возможных действий: сдвиг или свертку. После выполнения определенного количества тактов анализатор допускает или отвергает анализируемую цепочку. Рассмотрим подробнее его

действия. Выполнение каждого такта можно разбить на два этапа. На первом происходит преобразование информации, записанной в нижнем и, возможно, верхнем магазинах. Информация, определяющая первый этап такта, — это правое состояние в цепочке состояний, записанной в верхнем магазине, и k левых символов еще не обработанной части входной цепочки.

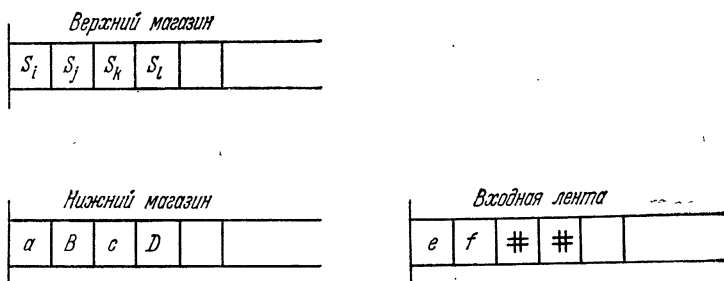


Рис. 8.

Таким образом, число k $LR(k)$ -анализатора определяет длину цепочки, учитываемой при выборе первого этапа. Работа, выполняемая на первом этапе, зависит от вида действия. Если это действие — сдвиг, то в нижний магазин записывается терминальный символ, являющийся левым символом наблюдаемой цепочки. Верхний магазин остается без изменений. Символ, записанный в нижний магазин, исключается из входной цепочки так, как это происходит при работе анализатора предшествования. Если выполняемое действие — свертка, то из нижнего и верхнего магазинов исключается по равному числу символов, после чего в нижний магазин записывается некоторый нетерминальный символ. При этом входная цепочка остается без изменений. Информацией, определяющей второй этап такта, служат правые состояние и символ из цепочек, находящихся, соответственно, в верхнем и нижнем магазинах после выполнения первого этапа. Второй этап не зависит от вида действия и всегда заключается в записи в верхний магазин некоторого состояния, называемого *переходным*. Посмотрим, как преобразуется информация за один такт работы анализатора. Пусть до выполнения такта

$S_0 S_1 \dots S_n$ — цепочка состояний, записанная в верхнем магазине;
 $Z_0 Z_1 \dots Z_n$ — цепочка символов, записанная в нижнем магазине;
 $a_1 \dots a_k a_{k+1} x$ — цепочка, записанная на входной ленте.
 Тогда информация преобразуется, как показано в табл. 5.

Таблица 5

Вид действия	Сдвиг	Свертка
После первого этапа	$S_0 S_1 \dots S_n$ $Z_0 Z_1 \dots Z_n a_1$ $a_2 a_3 \dots a_{k+1} x$	$S_0 S_1 \dots S_{n-m}$ $Z_0 Z_1 \dots Z_{n-m} A$ $a_1 a_2 \dots a_{k+1} x$
После второго этапа	$S_0 S_1 \dots S_n S$ $Z_0 Z_1 \dots Z_n a_1$ $a_2 a_3 \dots a_{k+1} x$	$S_0 S_1 \dots S_{n-m} S$ $Z_0 Z_1 \dots Z_{n-m} A$ $a_1 a_2 \dots a_{k+1} x$

Здесь $m \geq 0$ — число, определяющее количество символов, исключаемых из нижнего и верхнего магазинов при свертке, S — переходное состояние, A — терминальный символ грамматики.

Перед началом работы анализатора в верхний магазин помещается начальное состояние S_0 , а в нижний — граничный маркер Z_0 . Цепочка допускается, если правым в верхнем магазине окажется заключительное состояние S_F , а наблюдаемая цепочка будет состоять из k маркеров: $\# \# \dots \#$. Цепочка отвергается, если в процессе анализа возникнет такая ситуация, при которой очередной такт работы анализатора не определен.

Обращаясь к смыслу действий, производимых анализатором, заметим, что свертка соответствует случаю, когда возможно использование некоторого правила вывода порождающей грамматики, по которой построен $LR(k)$ -анализатор. При этом символы, исключаемые из нижнего магазина, совпадают с правой частью, а символ, который записывается в магазин, — с левой частью данного правила. Если такую свертку осуществить нельзя, анализатор производит сдвиг. Как уже упоминалось выше, $LR(k)$ -анализатор

является детерминированным устройством, и, следовательно, каждый такт его работы однозначно определяется по информации, от которой зависят его этапы. С другой стороны, возможны такие сочетания состояний и наблюдаемых цепочек, при которых дальнейшая работа анализатора не определена.

Рассмотрим теперь формальное определение $LR(k)$ -анализатора.

- Определение. $LR(k)$ -анализатор, соответствующий УКС-грамматике $G = \langle V_T, V_A, I, S \rangle$, представляется в виде девятки следующих объектов: $LR(k) = \langle U, \Sigma, H, T, \delta_1, \delta_2, S_0, Z_0, S_F \rangle$, где:

U — конечное множество состояний анализатора;

Σ — конечное множество входных символов, которое образовано включением маркера во множество терминальных символов грамматики $G: \Sigma = V_T \cup U\{\#\}$, $\# \notin V_T$;

H — конечное множество символов нижнего магазина, образованное из терминальных и нетерминальных символов грамматики и граничного маркера: $H = V_T \cup V_A \cup \{Z_0\}$, $Z_0 \notin V_T \cup V_A$;

T — множество, в котором в качестве элементов входят символ Q и конечное число пар вида (p, A) , где p — неотрицательное целое число, $A \in V_A$; содержательно множество T состоит из элементов, определяющих первые этапы тактов анализатора, причем Q означает сдвиг, а (p, A) — свертку с исключением из магазинов p символов и записью в нижний магазин символа A ;

δ_1 — частично определенная функция, задающая первые этапы тактов анализатора; она отображает множество $U \times \Sigma^k$ в T , где Σ^k — множество цепочек длины k над алфавитом Σ ;

δ_2 — частично определенная функция, задающая вторые этапы тактов анализатора; она отображает множество $U \times H$ в U ;

S_0 — начальное состояние, $S_0 \in U$;

Z_0 — граничный маркер;

S_F — заключительное состояние, $S_F \in U$.

При практическом построении $LR(k)$ -анализаторов k в большинстве случаев принимается равным 1, так как при $k > 1$ объем таблицы, задающей функцию δ_1 , оказывается чрезмерно большим. Любой $LR(k)$ -ана-

лизатор может быть преобразован в обобщенный детерминированный МП-автомат. При доказательстве этого утверждения следует воспользоваться тем свойством анализаторов, что на каждом такте они записывают по одному символу в верхний и нижний магазины, и исключаться из магазинов эти символы могут только одновременно (при свертке). Следовательно, верхний магазин может быть исключен, если каждый символ в нижнем магазине снабдить индексом, соответствующим тому состоянию, которое записывается в верхний магазин одновременно с данным символом. При этом каждый символ нижнего магазина, кроме граничного маркера, должен иметь n модификаций, где n — число состояний анализатора, соответствующих этому символу. С другой стороны, известно, что для любого языка, распознаваемого $LR(k)$ -анализатором при некотором $k > 1$, существует распознающий его $LR(1)$ -анализатор. Таким образом, класс языков, распознаваемых $LR(1)$ -анализаторами, совпадает с классом языков, распознаваемых $LR(k)$ -анализаторами при любых k , и входит в класс не существенно неоднозначных УКС-языков. Функции δ_1 и δ_2 $LR(k)$ -анализатора задаются обычно в виде общей таблицы, состоящей из конечного числа так называемых *рядов*. Каждый ряд соответствует некоторому состоянию и имеет структуру, указанную в табл. 6.

Для заключительного состояния S_F в таблице функций δ_1 и δ_2 имеется также следующая строка:

Состояние	Наблюдаемая цепочка	Функция
S_F	$\# \dots \#$	допуск

(наблюдаемая цепочка состоит здесь из k маркеров). Таблица указанной структуры называется анализирующей таблицей $LR(k)$ -анализатора.

Рассмотрим теперь алгоритм построения анализирующей таблицы по УКС-грамматике. Пусть задана УКС-грамматика G . Прежде всего эта грамматика должна быть несколько преобразована для того, чтобы все цепочки, входящие в язык, порождаемый грамматикой, заканчивались справа k маркерами. Преобразование заключается во включении в V_A нового символа I_0 , который будет считаться начальным, и в S — нового правила вывода $I_0 \rightarrow I \# \dots \#$. По пре-

Таблица 6

Состояние	Наблюдаемая цепочка	Функция δ_1 (действие)	Символ нижнего магазина	Функция δ_2 (переходное состояние)
S_ξ	x_{i_1}	(p_{i_1}, A_{i_1})	z_{i_1}	S_{i_1}
	x_{i_2}	Q	z_{i_2}	S_{i_2}

			z_{i_n}	S_{i_n}
	x_{i_l}	(p_{i_m}, A_{i_m})		

образованной грамматике и строится анализирующая таблица.

Обозначим преобразованную грамматику через $G' = \langle V_T', V_A', I_0, S' \rangle$. Пусть в ней имеется $q+1$ правило вывода. Перенумеруем эти правила числами от 0 до q , причем правилу $I_0 \rightarrow I \# \dots \#$ присвоим номер 0. Правило грамматики с номером r можно представить в виде $A_r \rightarrow \alpha_{r1} \alpha_{r2} \dots \alpha_{rn_r}$, $\alpha_{rj} \in V_T' \cup V_A'$, или $A_r \rightarrow \lambda$, если оно укорачивающее. Пусть ξ — некоторая непустая цепочка из $(V_T' \cup V_A')^*$. Введем в рассмотрение два множества $J(\xi)$ и $J'(\xi)$ терминальных цепочек грамматики G' длины k . Определим эти множества следующим образом. Если $\xi = x\eta$, $x \in V_T'^*$, $l(x) = k$, $\eta \in (V_T' \cup V_A')^*$, то $J(\xi) = J'(\xi) = \{x\}$. В противном случае включим в $J(\xi)$ все такие терминальные цепочки длины k , которые являются левыми подцепочками цепочек, выводимых из ξ : $x \in J(\xi)$ в том и только том случае, если при некотором $\eta \in (V_T' \cup V_A')^*$ имеет место выводимость $\xi \xRightarrow{G'}^* x\eta$. В $J'(\xi)$ включим все такие цепочки из $J(\xi)$, которые являются левыми подцепочками цепочек, выводимых из ξ без применения шагов вида

$A\omega \Rightarrow \omega$. Эти множества всегда могут быть найдены за конечное число шагов, однако трудоемкость соответствующей процедуры быстро увеличивается при возрастании k . При $k=1$ множества $J(\xi)$ и $J'(\xi)$, которые в этом случае можно рассматривать как подмножества терминальных символов грамматики G' , находятся следующим образом:

- 1) Пусть $\xi = a\xi'$, $a \in V_T$. Тогда $J(\xi) = J'(\xi) = \{a\}$.
- 2) Пусть $\xi = A_1 A_2 \dots A_j \xi'$, $j \geq 1$, A_1, A_2, \dots, A_{j-1} — укорачивающие символы, а A_j — неукорачивающий. Тогда в $J'(\xi)$ входят все такие символы a , что $A_1 \hat{D}_L a$, а в $J(\xi)$ — все такие a , что хотя бы при одном i , $A_i \hat{D}_s a$, $1 \leq i \leq j$. Здесь D_s — следующее бинарное отношение: $\alpha_1 D_s \alpha_2$, если $\alpha_1 D_L \alpha_2$, или имеется правило вида $\alpha_1 \rightarrow B_1 B_2 \dots B_n \alpha_2$, и все B_i — укорачивающие.

Процесс построения анализирующей таблицы по грамматике G' заключается в последовательном нахождении рядов для всех состояний анализатора. Каждое состояние характеризуется конечным набором так называемых частичных состояний, представимых в виде троек $[r, j, \omega]$, где r — номер правила вывода грамматики G' , j — число, $0 \leq j \leq n_r$, ω — терминальная цепочка длины k . Мы будем считать, что анализатор находится в частичном состоянии $[r, j, \omega]$, если в нижнем магазине находится цепочка $\eta \alpha_{r1} \dots \alpha_{rj}$, $\eta \in (V_T' \cup V_A)^*$. Таким образом, если анализатор находится в данном частичном состоянии, то он, просматривая анализируемую цепочку слева направо, уже произвел необходимые свертки и достиг положения, когда первые j символов правой части r -го правила находятся в нижнем магазине. При этом ω — k -символьная терминальная цепочка, которая может находиться в анализируемой цепочке правее подцепочки, выведенной из A_r , если применялось r -е правило.

Рассмотрим метод построения очередного ряда анализирующей таблицы, обозначив через S состояние анализатора, соответствующее этому ряду. При построении ряда будем последовательно находить следующие объекты:

- а) набор частичных состояний, характеризующих S ; ниже нам будет удобно рассматривать S как множество всех таких частичных состояний;

б) множество наблюдаемых цепочек, для которых при состоянии S должно быть определено значение функции δ_1 ; это множество будем обозначать через $L(S)$;

в) значения функции δ_1 для каждой из наблюдаемых цепочек, входящих в множество $L(S)$;

г) подмножество символов нижнего магазина, для которых при данном состоянии должно быть определено значение функции δ_2 ; это подмножество будем обозначать через $H(S)$;

д) значения функции δ_2 для каждого символа, входящего в $H(S)$; эти значения будут находиться как наборы некоторых частичных состояний, входящих в соответствующие переходные состояния.

Рассмотрим подробнее алгоритмы нахождения объектов указанного вида. Каждое состояние, кроме начального, вводится в рассмотрение тогда, когда оно появилось как значение функции δ_2 при вычислении некоторого ряда анализирующей таблицы. Следовательно, некоторые частичные состояния, входящие в S , уже известны в момент, когда начинается построение ряда для этого состояния. Все такие частичные состояния назовем основными состояниями. В подмножество основных частичных состояний начального состояния анализатора S_0 включим единственное состояние $[0, 0, \# \dots \#]$. Частичные состояния, включаемые в S и не являющиеся основными, назовем присоединенными. При построении ряда для состояния S как раз и должны быть определены все присоединенные частичные состояния, входящие в S . Соответствующий алгоритм можно представить в виде итеративного процесса, состоящего из следующих шагов.

Шаг 1. Рассматриваются все основные частичные состояния, включенные в S .

Шаг 2. По каждому рассматриваемому частичному состоянию $[r, j, \omega]$ такому, что $j < n_r$, находим все частичные состояния $[q, 0, x]$ такие, что $\alpha_{rj+1} = A_q$, $x \in I(\alpha_{rj+2} \dots \alpha_{rhr} \omega)$. Включим в S все найденные частичные состояния.

Шаг 3. Если на предыдущем шаге не найдено ни одного нового частичного состояния, процесс закончен. В противном случае рассматриваем частичные

состояния, найденные на предыдущем шаге, и переходим к шагу 2.

Так как для любой грамматики имеется конечное число частичных состояний, этот итеративный процесс закончится после выполнения конечного числа шагов.

Определение множества $L(S)$ и значений функции δ_1 для каждой цепочки, входящей в это множество, выполняется следующим образом:

1) Если $[r, j, \omega] \in S$ и $j < n_r$, то включим в $L(S)$ все такие цепочки y , что $y \in J'(\alpha_{rj+1} \dots \alpha_{rn_r} \omega)$. Значение δ_1 от всех таких цепочек определим как Q (сдвиг). Содержательно случай, когда $j < n_r$, означает, что анализатор еще не закончил анализ r -го правила вывода.

2) Если $[r, n_r, \omega] \in S$, то $\omega \in L(S)$, и значение δ_1 определим как (n_r, A_r) , т. е. как свертку по r -му правилу. Содержательно этот пункт соответствует случаю, когда анализ r -го правила закончен и, следовательно, может быть произведена свертка в соответствии с этим правилом.

Если для какой-либо цепочки из $L(S)$ в соответствии с пунктами 1 и 2 определяются два или более значений функции δ_1 , то процесс построения анализирующей таблицы заканчивается, так как его продолжение невозможно. Это означает, что исходная грамматика G не принадлежит к классу грамматик, по которым возможно построение $LR(k)$ -анализаторов.

В дальнейшем классы грамматик, по которым построение таких анализаторов возможно, будем называть $LR(k)$ -грамматиками.

Подмножество $H(S)$ образуем из всех таких символов Z , для которых найдется частичное состояние $[r, j, \omega] \in S$ такое, что $Z = \alpha_{rj+1}$. Наконец, по каждому $Z \in H(S)$ определим значение функции δ_2 как переходное состояние, в которое входят все такие основные частичные состояния $[r, j+1, \omega]$, что $[r, j, \omega] \in S$ и $Z = \alpha_{rj+1}$.

При последовательном построении рядов анализирующей таблицы необходимо иметь список состояний, для которых ряды еще не построены. Первоначально в этот список включается одно начальное состояние

S_0 . По окончании построения каждого ряда список изменяется следующим образом:

а) из него исключается состояние, для которого был построен данный ряд;

б) в него включаются все те состояния, которые определены при построении данного ряда как переходные, и подмножества основных частичных состояний которых не совпадают ни с одним из таких подмножеств состояний, ряды для которых уже построены.

Процесс построения анализирующей таблицы по $LR(k)$ -грамматике заканчивается тогда, когда данный список состояний окажется пустым. Так как множество частичных состояний для любой грамматики конечно, конечным является и множество состояний любого анализатора. Следовательно, процесс построения анализирующей таблицы закончится после построения конечного числа рядов.

Пример 1. Рассмотрим процесс построения анализирующей таблицы $LR(1)$ -анализатора для синтаксической конструкции языка АЛГОЛ-60 «заголовок процедуры». Соответствующая грамматика приведена в примере 7, § 3, гл. I. Для того, чтобы не перегружать анализирующую таблицу, будем, как и в ряде уже приведенных примеров считать, что идентификаторы, встречающиеся в конструкции, анализу не подлежат и могут считаться терминальными символами грамматики. Кроме того, будем считать терминальными символами ограничитель параметра и спецификацию. Для этих трех объектов используем обозначения $\langle i \rangle$, $\langle or \rangle$ и $\langle sp \rangle$. Наконец, исключим из грамматики лишнее правило 12 и включим в нее правило для ограничителя, которым должна заканчиваться каждая цепочка, анализируемая $LR(1)$ -анализатором. Изменив соответствующим образом грамматику из примера 7, получим грамматику, которую будем считать исходной для алгоритма построения анализирующей таблицы:

- | | |
|--|--|
| 0) $S_0 \rightarrow Z\#$ | 7) $\Phi \rightarrow \langle i \rangle$ |
| 1) $Z \rightarrow PC_1; C_2C_3$ | 8) $C_2 \rightarrow \text{value}C_5;$ |
| 2) $P \rightarrow \langle i \rangle$ | 9) $C_2 \rightarrow \lambda$ |
| 3) $C_1 \rightarrow (C_4)$ | 10) $C_3 \rightarrow \langle i \rangle$ |
| 4) $C_1 \rightarrow \lambda$ | 11) $C_3 \rightarrow C_5, \langle i \rangle$ |
| 5) $C_4 \rightarrow \Phi$ | 12) $C_3 \rightarrow C_3\langle sp \rangle C_5;$ |
| 6) $C_4 \rightarrow C_4\langle or \rangle\Phi$ | 13) $C_3 \rightarrow \lambda$ |

Анализирующая таблица $LR(1)$ -анализатора, распознающего язык, порождаемый данной грамматикой, будет содержать 8 колонок. В первой будет указываться состояние анализатора, во второй будут перечислены все основные частичные состояния,

включенные в данное состояние, а в третьей — все присоединенные частичные состояния. В случае построения $LR(1)$ -анализатора третьим объектом частичного состояния является терминальный символ. Если частичные состояния различаются лишь своими третьими компонентами, они будут для сокращения таблицы объединяться. Будем, кроме того, опускать запятые между компонентами, а номера правил 10, 11, 12 и 13 обозначать соответственно через $\bar{0}$, $\bar{1}$, $\bar{2}$ и $\bar{3}$. Так, например, запись $[22\langle\text{сп}\rangle\#]$ изображает два частичных состояния: $[12, 2, \langle\text{сп}\rangle]$ и $[12, 2, \#]$. Заметим, что вторая, третья, а также седьмая колонки нужны лишь при построении анализирующей таблицы. При использовании готовой таблицы для анализа они могут быть исключены, после чего таблица примет вид, указанный выше при определении $LR(1)$ -анализатора. В четвертой колонке нашей таблицы будут указываться наблюдаемые цепочки, которые в случае построения $LR(1)$ -анализатора могут рассматриваться как символы. Пятая колонка отводится для значений функции δ_1 , шестая — для символов нижнего магазина, от которых зависят значения функции δ_2 . Значения этой функции будут записываться в седьмой колонке в виде подмножеств основных частичных состояний. Наконец, в восьмой колонке будут указываться соответствующие этим подмножествам переходные состояния.

Анализирующая таблица для заголовков процедур, соответствующая приведенной выше грамматике, показана в табл. 7.

Посмотрим, как работает построенный нами $LR(1)$ -анализатор при анализе цепочки $f(a, b)$; $\text{value } a$; $\text{real } a$; $\#$. Для каждого такта анализатора будем указывать его номер, информацию, находящуюся в верхнем и нижнем магазинах до выполнения данного такта, и наблюдаемый символ. Например,

$$\begin{array}{cc} 2 & \\ S_0 & S_3 \\ (& \\ Z_0 & f \end{array}$$

означает, что рассматривается такт номер 2. При этом в верхнем магазине находятся цепочка состояний S_0S_3 , в нижнем — цепочка символов Z_0f (здесь Z_0 — граничный маркер), а наблюдаемый символ $($. Для анализа приведенного выше заголовка процедуры потребуется 26 тактов работы процессора, что иллюстрирует табл. 8. Поскольку по окончании анализа крайнее правое состояние в верхнем магазине S_1 , а наблюдаемый символ $\#$, входная цепочка анализатором допущена.

По данной таблице легко получить правосторонний вывод входной цепочки в исходной грамматике. Начало этого вывода стандартно и имеет вид: $Z_0, 3\#$. Для того, чтобы найти продолжение вывода, нужно отметить такты, на которых производилась свертка, и, просматривая эти такты справа налево, заменять в цепочках вывода результаты сверток (правые нетерминальные символы) цепочками в конце нижнего магазина, из которых они были получены. В нашем случае свертка производилась на тактах с номерами 2, 5, 6, 9, 10, 12, 16, 18, 19, 22, 24 и 25. Поэтому

Таблица 7

Состояния	Основные частичные состояния	Присоединенные частичные состояния	Наблюдаемые символы	δ_1	Символы в магазине	δ_2	Переходные состояния
S_0	[00#]	[10#] [20(;	<и>	Q	$\frac{3}{\pi}$ $\frac{1}{\langle \text{и} \rangle}$	$\frac{[01\#]}{[11\#]} \frac{[12\#]}{[31;]}$	$\frac{S_1}{S_2} \frac{S_3}{S_5}$
S_1	[01#]		#	Допуск			
S_2	[11#]	[30;] [40;]	(;	$\frac{Q}{(C_1, 0)}$	$\frac{C_1}{(}$	$\frac{[12\#]}{[31;]}$	$\frac{S_4}{S_5}$
S_3	[21(;		(;	$\frac{(\pi, 1)}{(\pi, 1)}$			
S_4	[12#]		;	Q	;	[13#]	S_6
S_5	[31;]	[50]<or> [60]<or> [70]<or>	<и>	Q	$\frac{C_4}{\phi}$ $\frac{1}{\langle \text{и} \rangle}$	$\frac{[32;]}{[61]\langle \text{or} \rangle} \frac{[14\#]}{[81]\langle \text{сп} \rangle \#}$	$\frac{S_7}{S_8} \frac{S_9}{S_{11}}$
S_6	[13#]	[80]<сп>#] [90]<сп>#]	value <сп> #	$\frac{Q}{(C_2, 0)}$ $\frac{1}{(C_2, 0)}$	$\frac{C_3}{\text{value}}$		

S_7	$[32;]$ $[61]\langle \text{or} \rangle$		$\frac{)}{\langle \text{or} \rangle}$	$\frac{Q}{Q}$	$\frac{)}{\langle \text{or} \rangle}$	$\frac{[33;]}{[62]\langle \text{or} \rangle}$	$\frac{S_{12}}{S_{13}}$
S_8	$[51]\langle \text{or} \rangle$		$\frac{)}{\langle \text{or} \rangle}$	$\frac{(C_4, 1)}{(C_4, 1)}$			
S_9	$[71]\langle \text{or} \rangle$		$\frac{)}{\langle \text{or} \rangle}$	$\frac{(\phi, 1)}{(\phi, 1)}$			
S_{10}	$[14\#]$	$\frac{[20]\langle \text{cn} \rangle \#}{[30]\langle \text{cn} \rangle \#}$	$\frac{\#}{\langle \text{cn} \rangle}$	$\frac{(C_3, 0)}{(C_3, 0)}$	C_3	$\frac{[15\#]}{[21]\langle \text{cn} \rangle \#}$	S_{14}
S_{11}	$[81]\langle \text{cn} \rangle \#$	$\frac{[00; \cdot]}{[10; \cdot]}$	$\langle u \rangle$	Q	C_5	$\frac{[82]\langle \text{cn} \rangle \#}{\frac{[11; \cdot]}{[01; \cdot]}}$	$\frac{S_{15}}{S_{16}}$
S_{12}	$[33;]$		$;$	$(C_1, 3)$			
S_{13}	$[62]\langle \text{or} \rangle$	$[70]\langle \text{or} \rangle$	$\langle u \rangle$	Q	$\frac{\phi}{\langle u \rangle}$	$\frac{[61]\langle \text{or} \rangle}{[71]\langle \text{or} \rangle}$	$\frac{S_{17}}{S_{18}}$
S_{14}	$\frac{[15\#]}{[21]\langle \text{cn} \rangle \#}$		$\frac{\#}{\langle \text{cn} \rangle}$	$\frac{(3, 5)}{Q}$	$\langle \text{cn} \rangle$	$[22]\langle \text{cn} \rangle \#$	S_{18}
S_{15}	$\frac{[82]\langle \text{cn} \rangle \#}{[11; \cdot]}$		$\frac{;}{;}$	$\frac{Q}{Q}$	$;$	$\frac{[83]\langle \text{cn} \rangle \#}{[12; \cdot]}$	$\frac{S_{19}}{S_{20}}$

Состояния	Основные частичные состояния	Присоединенные частичные состояния	Наблюдаемые символы	δ_1	Символы в магазине	δ_2	Переходные состояния
S_{16}	$[\bar{0}1; .]$		$-\frac{.}{-} - \frac{.}{-}$;	$-\frac{(C_5, 1)}{(C_5, 1)} -$			
S_{17}	$[63]<or>]$		$-\frac{)}{-} -$ <or>	$-\frac{(C_4, 3)}{(C_4, 3)} -$			
S_{18}	$[\bar{2}2<cp>\#]$	$[\bar{0}0; .]$ $[\bar{1}0; .]$	$<и>$	Q	C_5 $<и>$	$[\bar{2}3<cp>\#]$ $[\bar{1}1; .]$ $[\bar{0}1; .]$	S_{21} S_{16}
S_{19}	$[83<cp>\#]$		$<cp>$ #	$-\frac{(C_2, 3)}{(C_2, 3)} -$			
S_{20}	$[\bar{1}2; .]$		$<и>$	Q	$<и>$	$[\bar{1}3; .]$	S_{22}
S_{21}	$[\bar{2}3<cp>\#]$ $[\bar{1}1; .]$		$-\frac{.}{-} - \frac{.}{-}$;	$-\frac{Q}{-} -$ Q	$-\frac{.}{-} -$;	$[\bar{2}4<cp>\#]$ $[\bar{1}2; .]$	S_{23} S_{20}
S_{22}	$[\bar{1}3; .]$		$-\frac{.}{-} - \frac{.}{-}$;	$-\frac{(C_5, 3)}{(C_5, 3)} -$			
S_{23}	$[\bar{2}4<cp>\#]$		$<cp>$ #	$-\frac{(C_3, 4)}{(C_3, 4)} -$			

Таблица 8

$\begin{matrix} 1 \\ S_0 \\ Z_0 \end{matrix}$	$\begin{matrix} 2 \\ S_0 S_3 \\ Z_0 f \end{matrix}$	$\begin{matrix} 3 \\ S_0 S_3 \\ Z_0 \Pi \end{matrix}$	$\begin{matrix} 4 \\ S_0 S_2 S_5 \\ Z_0 \Pi (\end{matrix}$	$\begin{matrix} 5 \\ S_0 S_2 S_5 S_9 \\ Z_0 \Pi (a , \end{matrix}$	$\begin{matrix} 6 \\ S_0 S_2 S_5 S_8 \\ Z_0 \Pi (\Phi , \end{matrix}$	$\begin{matrix} 7 \\ S_0 S_2 S_5 S_7 \\ Z_0 \Pi (C_4 , \end{matrix}$
$\begin{matrix} 8 \\ S_0 S_2 S_5 S_7 S_{13} \\ Z_0 \Pi (C_4) \end{matrix}$	$\begin{matrix} 9 \\ S_0 S_2 S_5 S_7 S_{13} S_9 \\ Z_0 \Pi (C_4, b \end{matrix}$	$\begin{matrix} 10 \\ S_0 S_2 S_5 S_7 S_{13} S_{17} \\ Z_0 \Pi (C_4, \Phi \end{matrix}$	$\begin{matrix} 11 \\ S_0 S_2 S_5 S_7 \\ Z_0 \Pi (C_4 \end{matrix}$	$\begin{matrix} 12 \\ S_0 S_2 S_5 S_{12} \\ Z_0 \Pi (C_4) \end{matrix}$		
$\begin{matrix} 13 \\ S_0 S_2 S_4 \\ Z_0 \Pi C_1 \end{matrix}$	$\begin{matrix} 14 \\ S_0 S_2 S_4 S_6 \\ Z_0 \Pi C_1; \end{matrix}$	$\begin{matrix} 15 \\ S_0 S_2 S_4 S_6 S_{11} \\ Z_0 \Pi C_1; \text{ value } a \end{matrix}$	$\begin{matrix} 16 \\ S_0 S_2 S_4 S_6 S_{11} S_{16} \\ Z_0 \Pi C_1; \text{ value } a \end{matrix}$	$\begin{matrix} 17 \\ S_0 S_2 S_4 S_6 S_{11} S_{15} \\ Z_0 \Pi C_1; \text{ value } C_5 \end{matrix}$	$\begin{matrix} 18 \\ S_0 S_2 S_4 S_6 S_{11} S_{15} S_{19} \\ Z_0 \Pi C_1; \text{ value } C_5; \end{matrix}$	real
$\begin{matrix} 19 \\ S_0 S_2 S_4 S_6 S_{10} \\ Z_0 \Pi C_1; C_2 \end{matrix}$	$\begin{matrix} 20 \\ S_0 S_2 S_4 S_6 S_{10} S_{14} \\ Z_0 \Pi C_1; C_2 C_3 \end{matrix}$	$\begin{matrix} 21 \\ S_0 S_2 S_4 S_6 S_{10} S_{14} S_{18} \\ Z_0 \Pi C_1; C_2 C_3 \text{ real } a \end{matrix}$	$\begin{matrix} 22 \\ S_0 S_2 S_4 S_6 S_{10} S_{14} S_{18} S_{16} \\ Z_0 \Pi C_1; C_2 C_3 \text{ real } a \end{matrix}$	$\begin{matrix} 23 \\ S_0 S_2 S_4 S_6 S_{10} S_{14} S_{18} S_{21} \\ Z_0 \Pi C_1; C_2 C_3 \text{ real } C_5 \end{matrix}$		
$\begin{matrix} 24 \\ S_0 S_2 S_4 S_6 S_{10} S_{14} S_{18} S_{21} S_{23} \\ Z_0 \Pi C_1; C_2 C_3 \text{ real } C_5; \end{matrix}$	$\begin{matrix} 25 \\ S_0 S_2 S_4 S_6 S_{10} S_{14} \\ Z_0 \Pi C_1; C_2 C_3 \end{matrix}$	$\begin{matrix} 26 \\ S_0 S_1 \\ Z_0 3 \end{matrix}$	$\begin{matrix} 27 \\ S_0 S_1 \\ Z_0 3 \end{matrix}$			

правосторонний вывод нашей входной цепочки имеет вид:
 $Z_0 \# PC_1; C_2C_3 \# PC_1; C_2C_3 \text{ real } C_5; \# PC_1; C_2C_3 \text{ real } a; \# PC_1; C_2 \text{ real } a; \# PC_1; \text{value } C_5; \text{real } a; \# PC_1; \text{value } a; \text{real } a; \# P(C_4); \text{value } a; \text{real } a; \# P(C_4, \Phi); \text{value } a; \text{real } a; \# P(C_4, b); \text{value } a; \text{real } a; \# P(\Phi, b); \text{value } a; \text{real } a; \# P(a, b); \text{value } a; \text{real } a; \# f(a, b) \text{value } a; \text{real } a; \#$

По таблице работы анализатора нетрудно получить также дерево вывода входной цепочки без предварительного нахождения правостороннего вывода.

Следует отметить, что грамматики, задающие синтаксис языков программирования, близки к $LR(1)$ -грамматикам. Однако практическому использованию $LR(k)$ -анализаторов для анализа программ препятствует один их существенный недостаток. Дело в том, что число состояний анализатора, а, следовательно, и длина анализирующей таблицы, очень быстро возрастают с усложнением грамматики. Можно показать, что верхней границей числа состояний $LR(k)$ -анализатора служит 2^{PK} , где P — сумма длин правых частей правил исходной грамматики, а $K = T^k$, где T — число терминальных символов грамматики. Существует ряд способов сократить число состояний анализатора. Например, если некоторое состояние является подмножеством другого состояния (в смысле входящих в них частичных состояний), то его можно отождествить с последним состоянием, уменьшив число состояний на единицу. Более интересным является метод, основанный на расчленении исходной грамматики [61]. Его основная идея заключается в следующем. Пусть имеется грамматика $G = \langle V_T, V_A, I, S \rangle$, по которой требуется построить анализирующую таблицу анализатора. Выберем некоторый нетерминальный символ этой грамматики A и определим две новые грамматики следующим образом: $G_1 = \langle V_T \cup \{A\}, V_A \setminus \{A\}, I, S_1 \rangle$, где в S_1 включаются все правила из S , левые части которых отличны от A , и $G_2 = \langle V_T, V_A, A, S \rangle$. Таким образом, символ A считается в грамматике G_1 терминальным, а грамматика G_2 порождает все терминальные цепочки, выводимые в исходной грамматике G из A . Так как в G_1 и G_2 могут встречаться несущественные символы, построим по ним эквивалентные приведенные грамматики G'_1 и G'_2 . А теперь определим по грамматикам G'_1 и G'_2 два $LR(k)$ -анализатора. При

выполнении некоторых условий их анализирующие таблицы могут быть объединены в одну путем несложных преобразований. Полученная таким способом анализирующая таблица может оказаться значительно короче таблицы, построенной по грамматике без ее расчленения.

§ 3. Глобальный анализатор

Рассмотрим теперь метод анализа цепочек по порождающей грамматике, известный под названием *глобального анализатора* [67]. Глобальный анализатор является анализатором, работающим в режиме сверху вниз. Он может быть построен для любой УКС-грамматики, не имеющей выводимости вида $A \Rightarrow^* A$. Основная идея этого метода заключается в следующем. Пусть в исходной грамматике имеется некоторое количество правил для начального символа: $I = x_{i1}A_{i1}x_{i2} \dots A_{in-1}x_{in}$; $i = 1, \dots, k$. Здесь x_{ij} — возможно пустые, терминальные цепочки, A_{ij} — нетерминальные символы. Рассмотрим анализируемую цепочку ω . Если на первом шаге вывода этой цепочки в грамматике использовалось конкретное правило указанного вида, то она должна быть представима в виде: $\omega = x_{i1}y_{i1} \dots y_{in-1}x_{in}$, где x_{ij} — подцепочки, совпадающие с соответствующими подцепочками правила, а y_{ij} — некоторые, возможно пустые, подцепочки. Если цепочка ω в указанном виде не представима, возьмем другое правило для начального символа и выполним такую же проверку. Если ни одно из правил для I , включенных во множество правил вывода грамматики, не подходит, то анализируемая цепочка, очевидно, не входит в язык, порождаемый интересующей нас грамматикой. Пусть, наоборот, подходящее правило нашлось. Рассмотрим подцепочки $y_{i1}, y_{i2}, \dots, y_{in-1}$ и проверим, могут ли они быть выведены в грамматике из соответствующих нетерминальных символов правой части правила: $A_{i1}, A_{i2}, \dots, A_{in-1}$. Для проверки возможности вывода y_{ij} из A_{ij} рассмотрим все правила грамматики для символа A_{ij} и выполним для каждого из них точно такую же проверку, которая была указана выше. Таким образом, анализ в данном случае можно представить в виде рекурсивной проце-

дуры, параметрами которой являются анализируемая цепочка или ее часть и некоторое правило грамматики. Нетрудно показать, что при отсутствии в грамматике выводов с повторяющимися цепочками анализ должен закончиться за конечное число шагов. С другой стороны, процесс анализа не является детерминированным. Пусть цепочка ω представлена в интересующем нас виде, но оказалось, что подцепочка y_{ij} не может быть выведена из символа A_{ij} . В этом случае необходимо вернуться к рассмотрению цепочки ω и проверить, не существует ли другое ее разбиение, соответствующее данному правилу. Если такое разбиение существует, нужно произвести проверки для новых подцепочек y_{ij} , а в противном случае рассмотреть другое правило для символа, из которого по предположению выводится цепочка ω .

Рекурсивная процедура проверки содержит рекурсивные вызовы себя самой всегда, когда ее аргументом является правило, в правой части которого имеются нетерминальные символы, и найдено разбиение цепочки, соответствующее этому правилу. Если же правая часть правила не содержит нетерминальных символов, то процедура дает ответ немедленно — достаточно проверить, совпадает ли анализируемая цепочка с правой частью такого правила.

Процедура проверки, которую мы в дальнейшем будем называть МАТЧН, является своего рода ядром глобального анализатора. Она не зависит от грамматики, в соответствии с которой производится анализ. Однако для того, чтобы анализатор был практически применим для работы с конкретными грамматиками, он должен учитывать их индивидуальные особенности. Эти особенности учитываются глобальным анализатором с помощью так называемых *ускоряющих тестов*. Рассмотрим работу ускоряющих тестов на конкретном примере. Пусть цепочка, анализируемая процедурой МАТЧН, имеет вид $abdedfgh$, а правило, в соответствии с которым проверяется выводимость, $A \rightarrow BdC$. Возможны два разбиения нашей цепочки: $y_1 = ab, x_2 = d, y_2 = edfgh$ и $y_1 = abde, x_2 = d, y_2 = fgh$. Рассмотрим первое из них. Прежде чем обратиться к процедуре МАТЧН для проверки выводимости y_1 из B и y_2 из C , целесообразно посмотреть, не являются ли

y_1 и y_2 такими цепочками, которые не могут быть выведены из нетерминальных символов B и C в данной грамматике. Ответ на этот вопрос и дают ускоряющие тесты. Если, например, известно, что минимальная длина терминальных цепочек, выводимых в данной грамматике из B , больше двух, или цепочки, выводимые из C , не могут начинаться с символа e , не имеет смысла вызывать *MATCH* для первого разбиения — оно должно быть отвергнуто как неподходящее, и можно сразу перейти ко второму разбиению. Таким образом, для работы ускоряющих тестов нужно получить по исходной грамматике некоторую информацию о свойствах цепочек, выводимых из ее нетерминальных символов. Приведем примеры ускоряющих тестов и укажем, на основании какой информации они должны работать.

1) Проверка на длину. Для этого теста необходимо вычислить минимальную длину терминальных цепочек, выводимых из каждого нетерминального символа грамматики. Может использоваться также информация о максимальных длинах, если они существуют.

2) Проверка на начало и конец. Эта проверка работает на основании следующей информации. Обозначим через *ALPH* множество таких терминальных цепочек, которые входят как подцепочки в правые части правил вывода грамматики. Тогда для каждого нетерминального символа A могут быть найдены подмножества *PRE*(A) и *POST*(A). Они состоят, соответственно, из тех элементов *ALPH*, которыми могут начинаться и на которые могут кончаться терминальные цепочки, выводимые из символа A . Если в результате разбиения символу A поставлена в соответствие некоторая цепочка, то данная проверка заключается в выяснении, начинается ли анализируемая цепочка с некоторой подцепочки из *PRE*(A) и заканчивается ли она подцепочкой из *POST*(A).

3) Проверка на вхождение неправильных подцепочек. Для каждого нетерминального символа A из множества *ALPH* можно выделить подмножество *EXC*(A) таких элементов, которые не могут входить как подцепочки ни в одну из терминальных цепочек, выводимых в грамматике из символа A . Данная проверка работает на основе этого подмножества.

Информация, необходимая для работы указанных тестов, т. е. минимальные и максимальные длины, подмножества PRE , $POST$ и EXC , сравнительно легко находится по исходной грамматике. Так, например, подмножество $PRE(A)$ состоит из таких цепочек, которые начинают цепочки, входящие в множество L_A (см. доказательство леммы 10, § 3, гл. II). При этом для нахождения $PRE(A)$ достаточно просмотреть конечное подмножество цепочек из L_A , выводы которых в грамматике — нециклические. Схожие алгоритмы могут быть предложены для нахождения $POST$ и EXC . Рассмотрим подробнее алгоритм вычисления минимальных длин. Обозначим минимальную длину терминальных цепочек, выводимых из символа A , через $MIN(A)$. Эту функцию можно определить и для терминальных символов, полагая $MIN(a) = 1$, где $a \in V_T$. Распространим эту функцию на произвольные цепочки над объединением терминального и нетерминального словарей, положив $MIN(\lambda) = 0$ и $MIN(\alpha_1 \dots \alpha_n) =$

$= \sum_{i=1}^n MIN(\alpha_i)$. Таким образом, для любой грамматики функция MIN для терминальных символов и цепочек, а также для пустой цепочки известна а priori, а для нетерминальных символов и цепочек, в которых имеются вхождения нетерминальных символов, должна вычисляться.

Вычисление функции MIN от нетерминальных символов можно представить в виде итеративного процесса, заканчивающегося после конечного числа шагов. При вычислении MIN используется то очевидное обстоятельство, что каждая цепочка минимальной длины может быть выведена нециклически. Поэтому данная функция не зависит от рекурсивных правил анализируемой грамматики. Пусть имеется УКС-грамматика $G = \langle V_T, V_A, A_1, S \rangle$, причем $V_A = \{A_1, A_2, \dots, A_n\}$. Обозначим через S_{nr} подмножество нерекурсивных правил вывода из S . Разобьем S_{nr} на n подмножеств, каждое из которых состоит из правил для которого нетерминального символа: $S_{nr} = \bigcup_{i=1}^n S_{nr}(A_i)$.

Прежде чем начать итеративный процесс вычисления функции MIN , выделим в каждом из подмножеств те правила, которые можно не учитывать при определе-

нии минимальной длины цепочек. С этой целью разобьем правила вывода каждого из выделенных подмножеств на две группы: $S_{nr}(A_i) = S_k(A_i) \cup S_u(A_i)$, включив в $S_k(A_i)$ те правила, в простых частях которых находятся терминальные или пустые цепочки, и в $S_u(A_i)$ — остальные правила. Можно считать, что $\bigcup_{i=1}^n S_k(A_i)$ не пусто,

так как в противном случае $L(G) = \emptyset$. Обозначим через M_i минимальные длины правых частей правил из $S_k(A_i)$, а если подмножество $S_k(A_i)$ пусто, будем условно считать, что $M_i = \infty$. Наконец, примем

$M = \min M_i$. Нетрудно видеть, что если $A_i \Rightarrow^* \omega \in V_T$,

то $l(\omega) \geq M$ и, следовательно, для любого i $MIN(A_i) \geq M$. Теперь введем в рассмотрение функцию $PARTMIN$, определив ее на любой цепочке из $(V_T \cup V_A)^*$ следующим образом:

1) если ω — терминальная или пустая цепочка, то $PARTMIN(\omega) = MIN(\omega)$;

2) $PARTMIN(A_i) = M_i$;

3) $PARTMIN(\alpha_1 \alpha_2 \dots \alpha_p) = \sum_{i=1}^p PARTMIN(\alpha_i)$.

Легко показать, что функция $PARTMIN$ оценивает функцию MIN снизу: $MIN(\omega) \geq PARTMIN(\omega)$.

Теперь из каждого подмножества $S_u(A_i)$ легко выделить группу правил, которые при вычислении MIN можно не учитывать: если $A_i \rightarrow \xi \in S_u(A_i)$ и $PARTMIN(\xi) \geq M_i$, то, используя в любом выводе вместо этого правила правило из $S_k(A_i)$, длина правой части которого равна M_i , мы не увеличиваем длину выводимой терминальной цепочки. Поэтому при определении MIN достаточно учитывать лишь те правила из $S_u(A_i)$, функция $PARTMIN$ от правых частей которых меньше, чем M_i . Обозначим группы таких правил через $S'_u(A_i)$, $i=1, \dots, n$.

Теперь мы можем приступить к рассмотрению итеративной процедуры вычисления функции MIN от нетерминальных символов грамматики G . На каждом шаге итерации нетерминальным символам будет поставлено в соответствие значение, являющееся целым числом или символом ∞ ; мы обозначим его через $ITMIN_j$, где j — номер итерации. Примем

$ITMIN_0(A_i) = M_i$. Пусть уже вычислены значения $ITMIN_j(A_i)$, $i=1, \dots, n$. Для вычисления $ITMIN_{j+1}(A_i)$ рассмотрим группу правил вывода $S'_u(A_i)$. Если $S'_u(A_i) = \emptyset$, то $ITMIN_{j+1}(A_i) = ITMIN_j(A_i)$. Пусть $A_i \rightarrow \xi \in S'_u(A_i)$ и $\xi = x_0 A_{i1} x_1 A_{i2} \dots A_{ip} x_p$, где $x_k \in V_T^*$. Поставим в соответствие данному правилу значение, вычисленное по формуле $\sum_{k=0}^p l(x_k) + \sum_{k=1}^p ITMIN_j(A_{ik})$ (если для некоторого k $ITMIN_j(A_{ik}) = \infty$, то и интересующее нас значение равно ∞). Обозначим минимум значений, сопоставленных указанным способом с правилами из $S'_u(A_i)$, через $MD_{j+1}(A_i)$ и примем $ITMIN_{j+1}(A_i) = \min(ITMIN_j(A_i), MD_{j+1}(A_i))$.

Данная процедура может быть закончена, когда на каком-либо ее шаге все вычисленные значения $ITMIN$ окажутся равными соответствующим значениям, вычисленным на предыдущем шаге. Покажем, что если J — номер итерации, на которой ни одно из значений $ITMIN$ не было изменено, то, во-первых, $J \leq n$ и, во-вторых, $MIN(A_i) = ITMIN_J(A_i)$, $i=1, 2, \dots, n$.

Для доказательства этих утверждений введем понятие обобщенного шага в грамматике G . Обобщенным шагом вывода будем считать применение к каждому вхождению нетерминальных символов цепочки вывода по одному правилу. Таким образом, если исходная цепочка имела вид $x_0 A_i x_1 \dots A_p x_p$, где, как обычно, $x_j \in V_T^*$, то после выполнения одного обобщенного шага получим цепочку $x_0 \eta_1 x_1 \dots \eta_p x_p$ и $A_i \rightarrow \eta_i \in S$ при $i=1, 2, \dots, p$. Таким образом, обобщенный шаг вывода соответствует построению всех вершин некоторого уровня дерева этого вывода, а число обобщенных шагов вывода равно высоте его дерева.

А теперь покажем, что $ITMIN_j(A_i)$ указывает минимальную длину терминальных цепочек, которые могут быть выведены из A_i в данной грамматике не более чем за $j+1$ обобщенный шаг. Если при этом $ITMIN_j(A_i) = \infty$, то за указанное число шагов из A_i вообще нельзя вывести ни одной терминальной цепочки. Это утверждение очевидно при $j=0$: оно непосредственно следует из способа задания значения $ITMIN_0(A_i)$. Доказывая его в общем случае по индукции, предположим, что оно верно для некоторого j ,

Рассмотрим вывод из A_i терминальной цепочки, состоящей не более чем из $j+2$ обобщенных шагов, причем будем считать, что на первом шаге применяется правило из $S'_u(A_i): A_i, \omega_1, \dots, \omega_m, 2 \leq m \leq j+2$.

Тогда справедливо неравенство: $MD_{j+1}(A_i) \leq \sum_{k=0}^p l(x_k) + \sum_{k=1}^p ITMIN_j(A_{1k})$, где $\omega_1 = x_0 A_{11} x_1 \dots A_{1p} x_p$. Но, согласно индукционному предположению, из каждого символа A_{1k} за $j+1$ или меньшее число шагов могут быть выведены терминальные цепочки длиной не менее $ITMIN_j(A_{1k})$. Итак, $MD_{j+1}(A_i)$ оценивает снизу длину терминальных цепочек, выводимых из A_i не более чем за $j+2$ обобщенных шага при условии, что на первом шаге применяются правила из $S'_u(A_i)$. При этом длина некоторых таких цепочек равна значению $MD_{j+1}(A_i)$. Если $S'_u(A_i) = \emptyset$ или $MD_{j+1}(A_i) = \infty$, но таких цепочек не существует.

Так как, согласно индукционному предположению, $ITMIN_j(A_i)$ оценивает снизу длину цепочек, выводимых из A_i не более чем за $j+1$ обобщенный шаг, и $ITMIN_{j+1}(A_i) = \infty$, только если $MD_{j+1}(A_i) = ITMIN_j(A_i) = \infty$, наше утверждение можно считать доказанным.

Из доказанного свойства и того, что высота дерева нециклического вывода не превосходит числа нетерминальных символов грамматики (см. доказательство леммы 3, § 3, гл. I), следует, что самое большое после $(n-1)$ -й итерации мы вычислим функцию MIN от каждого нетерминального символа. При этом, если J — номер итерации, на которой ни одно из значений $ITMIN$ не изменилось, то минимальные длины цепочек, выводимых не более чем за J обобщенных шагов и не более, чем за $J+1$ обобщенный шаг, совпадают для любого A_i и, следовательно, $ITMIN_J = MIN$ для всех нетерминальных символов.

Пример 1 вычисления минимальных длин. Рассмотрим УКС-грамматику со следующими правилами вывода:

- | | | |
|---------------------|----------------------------|--------------------------------|
| 1) $A_1 = abcc$, | 4) $A_3 = a$, | 7) $A_4 \rightarrow abA_3$, |
| 2) $A_1 = A_2A_3$, | 5) $A_3 = A_2A_5$, | 8) $A_5 \rightarrow \lambda$, |
| 3) $A_2 = A_3A_4$, | 6) $A_4 \rightarrow A_5$, | 9) $A_5 \rightarrow abA_4$. |

Определим подмножества S_k и S_u . В $S_k(A_1)$ входит правило 1, в $S_u(A_1)$ — правило 2, $S_k(A_2)$ — пусто, в $S_u(A_2)$ входит правило 3, в $S_k(A_3)$ — правило 4, в $S_u(A_3)$ — правило 5, $S_k(A_4)$ — пусто, в $S_u(A_4)$ входят правила 6 и 7, в $S_k(A_5)$ — правило 8 и, наконец, в $S_u(A_5)$ входит правило 9. Вычислим M_i и M : $M_1=4$, $M_2=\infty$, $M_3=1$, $M_4=\infty$, $M_5=0$, $M=0$.

Вычисляя функцию $PARTMIN$ для правых частей правил из $S_u(A_i)$, выясняем, что $PARTMIN(abA_4)=2$, что больше значения M_5 . Поэтому подмножество $S_u(A_5)$ пусто. Остальные $S_u(A_i)$ совпадают с $S_u(A_i)$.

Таблица 9

Номер итерации	$ITMIN(A_1)$	$ITMIN(A_2)$	$ITMIN(A_3)$	$ITMIN(A_4)$	$ITMIN(A_5)$
0	4	8	1	8	0
1	4	8	1	0	0
2	4	1	1	0	0
3	2	1	1	0	0
4	2	1	1	0	0

В табл. 9 представлены значения функции $ITMIN$, вычисленные с помощью описанной итеративной процедуры. Поскольку после четвертой итерации ни одно из значений не изменилось, для всех i $MIN(A_i)=ITMIN_4(A_i)$.

Пример 2 глобального анализатора с ускоряющими тестами. Рассмотрим грамматику для конструкции языка АЛГОЛ-60 «заголовок процедуры», приведенную в примере 1, § 2, гл. III (без правила 0). При использовании глобального анализатора будем применять ускоряющие тесты проверки на длину, начало и конец. Информация для них указана в таблице 10.

Процедура $MATCH$ будет выполнять перебор возможных вариантов разбиений следующим образом. Сначала для данных нетерминального символа и терминальной подцепочки разбиение будет производиться по первому правилу для данного символа. В случае, если существует несколько вариантов разбиения по правилу, выбирается так называемое левое разбиение, при котором левым вхождением нетерминальных символов ставятся в соответствие самые короткие терминальные подцепочки, длина которых не меньше соответствующих значений функции MIN (таким образом, выполняется тест проверки на длину). Далее работает тест проверки на начало и конец для всех подцепочек и нетерминальных символов в данном разбиении. Если тест дает утвердительный ответ, для подцепочек и символов данного разбиения снова вызывается процедура $MATCH$. В противном случае рассматривается следующее разбиение, причем в первую очередь увеличивается, если это возможно, длина подцепочки, которая ставится в соответствие предпоследнему вхождению нетерминаль-

Таблица 10

Нетерминальные символы	MIN	PRE	POST
З	2	<и>	;
C ₁	0	()
C ₂	0	value	;
C ₃	0	<сп>	;
C ₄	1	<и>	<и>
C ₅	1	<и>	<и>
П	1	<и>	<и>
Ф	1	<и>	<и>

ного символа, и далее работа продолжается в указанном порядке. Если первое правило для данного нетерминального символа не подходит, рассматривается второе правило и т. д.

Рассмотрим анализ заголовка процедуры $f(a, b); \text{value } a; \text{integer } a; \text{real } b;$. Ускоряющие тесты для символа З и данной цепочки дают утвердительный ответ. Первое и единственное правило для символа З — правило 1. В правой части этого правила имеется терминальный символ ;. Выбираем левое разбиение, которое укажем в линейно-скобочной записи: $[ПfП][C_1(a, b)C_1]; [C_2C_2][C_3 \text{ value } a; \text{integer } a; \text{real } b; C_3]$. Это разбиение получено с учетом того, что $MIN(C_2) = 0$. Выполним тест проверки на начало и конец. Этот тест дает утвердительный ответ для П и C₁ и отрицательный для C₃, так как начало C₃ в данном разбиении — символ value, который не входит в $PRE(C_3)$. Поэтому данное разбиение неверно, и мы должны рассмотреть следующее: $[ПfП][C_1(a, b)C_1]; [C_2 \text{ value } C_2][C_3 a; \text{integer } a; \text{real } b; C_3]$. Это разбиение также неверно: тест дает отрицательный ответ при анализе цепочки, которая поставлена в соответствие символу C₂ в связи с тем, что она заканчивается символом value, который не входит в $POST(C_2)$. Такой же ответ и при анализе цепочки, сопоставленной C₃, — цепочка не должна начинаться с символа <и>.

Продолжая подобным образом, получим, наконец, разбиение $[ПfП][C_1(a, b)C_1]; [C_2 \text{ value } a; C_2][C_3 \text{ integer } a; \text{real } b; C_3]$. Тест проверки на начало и конец дает для этого разбиения утвердительный ответ. Поэтому для каждой из выделенных подцепочек и соответствующего ей нетерминального символа вызывается процедура MATCH. Первой будет анализироваться цепочка f и поставленный ей в соответствие символ П. В грамматике для П имеется единственная схема 2, и, поскольку в правой части этой схемы находится представляющая любой идентификатор цепочка <и>, процедура MATCH выдает утвердительный ответ.

Затем MATCH вызывается для цепочки (a, b) и символа C₁. Первым для C₁ является правило 3. Производим разбиение цепочки: $[(C_4 a, b C_4)]$. Ускоряющий тест для данного разбиения дает утвердительный ответ. Рассматриваем первое правило для

C_4 : $[\Phi a, b \Phi]$. В соответствии с тестом это разбиение возможно. Вызываем процедуру *MATCH* для a, b и Φ . Для Φ имеется единственная схема 7, но она не подходит, поэтому процедура дает отрицательный ответ. Поскольку других правил для Φ и других разбиений для правила 5 нет, пытаемся разбить цепочку a, b по правилу 6: $[C_4aC_4]$, $[\Phi b \Phi]$. Продолжая анализ, получим $[\Phi a \Phi]$, и вызывая *MATCH* для цепочек a и b и символа Φ , выясним, что данное разбиение верно. Итак, анализ подцепочки $f(a, b)$ закончен, результат имеет вид: $[P/P][C_1([C[\Phi a \Phi]C_4], [\Phi b \Phi]C_4)C_1]$. Таким же образом происходит дальнейший анализ, в результате которого мы получим дерево вывода данного заголовка процедуры: $[3[P/P][C_1([C_4[C_4[\Phi a \Phi]C_4], [\Phi b \Phi]C_4)C_1]; [C_2 \text{ value } [C_5aC_5]; C_2][C_3[C_3[C_3C_3] \text{ integer } [C_5aC_5]; C_3] \text{ real } [C_5bC_5]; C_3]3]$.

Следует отметить, что при реализации данного алгоритма анализа на ЭВМ совсем не обязательно сначала выбирать разбиения, а затем проверять их с помощью теста проверки на начало и конец. Возможные начала и концы терминальных цепочек удобнее учитывать непосредственно при построении разбиений, точно так же, как учитывается минимальная длина. Перебор вариантов может быть сокращен и за счет вычисления минимальных и максимальных длин цепочек, выводимых из нетерминальных символов при помощи определенных правил для этого символа.

Основным недостатком глобального анализатора является его недетерминированный характер в том смысле, что возможны возвраты к уже пройденным этапам анализа в случае, если выбранные на этих этапах варианты разбиений оказались неудачными. Вероятность таких возвратов существенно снижается при использовании ускоряющих тестов. В случае рассмотренного выше набора ускоряющих тестов основной причиной возникновения неудачных вариантов анализа являются, по-видимому, парные символы, например, скобки языка АЛГОЛ-60. Для того, чтобы найти пары соответствующих друг другу скобок, не обойтись без их подсчета в анализируемой цепочке. Однако и в этом случае можно использовать тест, проверяющий взаимное расположение таких пар. Необходимую для этого теста информацию часто нетрудно получить по грамматике. Так, из грамматики, задающей простые арифметические выражения АЛГОЛ-60, легко определить, что в любом арифметическом выражении число левых и правых скобок должно совпадать. Это может учитываться при выборе разбиений. Сравнение эффективности анализаторов глобального и Кнута показывает, что первый часто работает быстрее.

ГЛАВА IV

КОНТЕКСТНЫЕ УСЛОВИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

В главе I при рассмотрении примера синтаксически неоднозначной конструкции языка АЛГОЛ-60 (см. пример 5, § 3) мы уже упоминали тот факт, что КС-грамматики, описывающие синтаксис языков программирования, не отражают некоторые правила построения программ на этих языках. Это обстоятельство вызвано тем, что языки программирования не являются контекстно-свободными и имеют более сложную по сравнению с последними синтаксическую структуру. Синтаксические правила языков программирования, которые не описываются средствами КС-грамматик и часто задаются неформально — с помощью обычных естественных языков, называются контекстными условиями. В настоящей главе мы рассмотрим наиболее характерные типы контекстных условий, а также некоторые возможные способы их формального задания и проверки.

§ 1. Классификация контекстных условий

Первый вопрос, который требует обсуждения в настоящем параграфе, заключается в определении того, какие именно свойства языков программирования, не задаваемые КС-грамматиками, следует считать контекстными условиями. Поскольку контекстные условия должны характеризовать синтаксис языков, наиболее естественным представляется требование, чтобы эти условия могли быть проверены при анализе программ до их выполнения. В противоположность этому семантическими удобно считать такие свойства программ, которые можно обнаружить лишь в процессе выполнения алгоритмов, представленных этими программами. Однако такое простое деление свойств,

характеризующих программы алгоритмических языков, на контекстные и семантические оказывается весьма неточным. Рассмотрим, например, следующее арифметическое выражение языка АЛГОЛ-60: a/b . Если в момент вычисления этого выражения значение b оказалось равным 0, произойдет переполнение, а оно в большинстве случаев сигнализирует о семантической ошибке, которую невозможно обнаружить при анализе программы. С другой стороны, обнаружение ошибки в выражении $a/0$ не требует его выполнения. Следовательно, согласно предложенному выше критерию последнюю ошибку нужно считать синтаксической. Но это в корне противоречит характеру данной ошибки, которая возникла не потому, что были нарушены какие-либо правила написания арифметического выражения как цепочки символов, а потому, что данное выражение оказалось бессмысленным — его значение не может быть вычислено. Другой пример такого рода — переменная с постоянными индексами. Если соответствующий этой переменной массив имеет границы, представленные в программе в виде чисел, можно без выполнения программы проверить, является ли значение каждого индекса переменной допустимым. Однако и в этом случае возможную ошибку в значении индекса следует считать семантической. С другой стороны, некоторые по характеру синтаксические ошибки требуют для обнаружения без выполнения программы большой работы, в то время как при выполнении программы они находятся без особых трудностей. Такие ошибки могут возникнуть, например, при вызовах процедур, формальные параметры которых специфицированы как процедуры — в случае большой глубины вложенности таких вызовов затруднительно до выполнения программы проверить, соответствуют ли фактические параметры всех вызываемых процедур их формальным параметрам.

Общепринятого деления требований, которым должны удовлетворять правильные программы алгоритмических языков, на синтаксические и семантические, не существует, и различные трансляторы выявляют различные классы ошибок. Тем не менее, круг свойств языков, которые обычно рассматриваются как

контекстные условия, достаточно хорошо очерчен. Большинство контекстных условий связано с двумя важнейшими особенностями языков программирования. Первой такой особенностью является возможность представления на этих языках алгоритмов обработки значений различных *видов* (типов). Можно считать, что значения каждого вида образуют взаимно непересекающиеся множества, и по каждому конкретному значению можно определить, к какому виду оно относится. В то же время идентификаторы могут именовать в программах значения многих или даже всех видов, используемых в том или ином языке программирования. В большинстве языков по виду идентификатора нельзя определить вид значения, которое он именуется в данной программе. Поэтому информация об использовании идентификаторов в программе должна указываться в этой программе явно — иными словами, идентификаторы, используемые в программе, должны быть в этой программе описаны. Именно это обстоятельство является одной из основных причин наличия в языках программирования контекстных условий. Так как значения различных видов в программах обычно невзаимозаменяемы (например, в программе на языке АЛГОЛ-60 вместо целого или вещественного числа в арифметическом выражении нельзя подставить логическое значение), при проверке синтаксической правильности конструкций необходимо знать виды значений, именуемых в этих конструкциях идентификаторами. А эта информация может быть получена лишь по описаниям данных идентификаторов в программах.

Второй особенностью, также приводящей к необходимости включать в синтаксис языков программирования контекстные условия, является наличие так называемых *областей действия*. Области действия величин, именуемых идентификаторами, введены в алгоритмические языки для предоставления программисту возможности управления распределением памяти ЭВМ при трансляции программ на машинные языки. Они могут указываться различными средствами. Во многих языках таким средством являются блоки. Большинство идентификаторов, описанных в некотором блоке программы, могут использоваться в соответст-

вин с описаниями только внутри этого блока. Исключение составляют идентификаторы так называемых *глобальных величин*, которые могут употребляться во всей программе независимо от места, в котором они описаны. Таким образом, областью действия всех описанных в некотором блоке идентификаторов величин, не являющихся глобальными, служит данный блок, а идентификаторов глобальных величин — вся программа. Поэтому конструкции языков программирования могут считаться синтаксически правильными только в том случае, если области действия используемых в этих конструкциях идентификаторов включают данные конструкции.

В соответствии с ролью, которую играют контекстные условия в синтаксическом описании языков, они могут быть разбиты на несколько групп. Рассмотрим более подробно каждую из них. В первую группу естественно включить условия, связанные с правилами описания идентификаторов, именующих различные объекты программ. Типичным для этой группы можно считать контекстное условие единственности описания идентификаторов, которое формулируется следующим образом: в любом блоке идентификатор нельзя описывать более одного раза. Это условие языка АЛГОЛ-60 относится к идентификаторам, именующим такие объекты, как простые переменные, массивы, процедуры и процедуры-функции, метки, переключатели. Для формальных параметров процедур условие единственности формулируется несколько иначе: при описании любой процедуры или процедуры-функции ни один идентификатор не должен входить более одного раза в список формальных параметров, список значений и в совокупность спецификаций. Аналогичные контекстные условия существуют и в других языках программирования. Так, в языке АЛГОЛ-68 формулируются контекстные условия единственности описания идентификаторов и индикаторов, т. е. обозначений, используемых для вновь вводимых в программах операций и видов значений.

Ко второй группе контекстных условий мы отнесем правила соответствия между так называемыми *определяющими* и *использующими* вхождениями иденти-

фикаторов. Определяющим называется вхождение идентификатора в конструкцию, описывающую этот идентификатор. В языке АЛГОЛ-60 такими конструкциями являются описания, а также помеченные операторы (они содержат определяющие вхождения идентификаторов меток, которыми помечаются). Рассмотренная выше первая группа контекстных условий в основном определяет правила использования идентификаторов в определяющих вхождениях. Используемым мы будем называть вхождение идентификатора в такую конструкцию, которая не является его описанием. Используемыми являются вхождения идентификаторов в выражения, фактические параметры, непомеченные операторы. Вхождения идентификаторов в некоторые конструкции имеют промежуточный характер, относясь по одним своим признакам к определяющим, а по другим — к использующим. В языке АЛГОЛ-60 такими конструкциями являются уже упомянутые выше списки значений и совокупности спецификаций формальных параметров процедур. Хотя эти вхождения по своему смыслу относятся к определяющим, с точки зрения некоторых контекстных условий второй группы их удобно рассматривать как использующие. Интересным представляется и использование индикаторов при описании нестандартных операций в программах на языке АЛГОЛ-68. Описывая нестандартную операцию, требуется, во-первых, указать ее приоритет, для чего служит конструкция «описание приоритета», и, во-вторых, — алгоритм выполнения операции, который задается в конструкции «описание операции». В обеих конструкциях требуется указать тот конкретный индикатор, который будет обозначать описываемую операцию. Ниже будет видно, что вхождение индикатора в описание приоритета можно безоговорочно отнести к определяющему, а вот его вхождение в описание операции в одном контекстном условии выступает как определяющее, а в другом — как использующее.

Контекстные условия второй группы задают для различных случаев то известное каждому программисту, знакомому с языком АЛГОЛ-60, правило, согласно которому каждая используемая в программе величина, требующая описания, должна быть в этой

программе описана. Формулируя данное правило в терминах определяющих и использующих вхождений, мы можем сказать, что по каждому использующему вхождению нестандартного (т. е. не представляющего стандартную функцию) идентификатора, целого числа, представляющего метку, а в случае программы на языке АЛГОЛ-68 — также и нестандартного индикатора, должно найтись соответствующее ему определяющее вхождение того же идентификатора, числа или индикатора. Правила поиска определяющих вхождений часто называются *алгоритмами идентификации* и могут быть различными для разных конструкций и языков программирования. Эти алгоритмы являются основной частью формулировки контекстных условий второй группы. Приведем несколько примеров. Контекстное условие для идентификаторов языка АЛГОЛ-60, которые не являются формальными параметрами процедур, формулируется следующим образом. Каждый используемый в программе нестандартный идентификатор должен быть описан. Для проверки наличия описания идентификатора требуется выполнить следующий алгоритм идентификации определяющего вхождения по использующему вхождению интересующего нас идентификатора:

- 1) Рассматриваем минимальный блок, содержащий данное использующее вхождение. Переходим к шагу 2.

- 2) Ищем определяющее вхождение идентификатора в рассматриваемом блоке. Если оно найдено, процедура идентификации закончена и данное использующее вхождение идентификатора удовлетворяет контекстному условию. В противном случае переходим к шагу 3.

- 3) Ищем минимальный блок, содержащий блок, рассматривавшийся перед этим. Если он найден, рассматриваем его и переходим к шагу 2. В противном случае (т. е. когда мы рассматривали блок, совпадающий с программой) процедура идентификации закончена, и так как определяющее вхождение не найдено, данное использующее вхождение идентификатора не удовлетворяет контекстному условию.

Несколько более сложные контекстные условия, связанные с формальными параметрами процедур.

Они могут быть сформулированы следующим образом. Во-первых, каждый идентификатор, входящий в совокупность спецификаций, должен входить также в список формальных параметров. Во-вторых, каждый идентификатор, входящий в список значений, должен входить в совокупность спецификаций (а, следовательно, и в список формальных параметров). Наконец, идентификаторы, встречающиеся в телах процедур, могут быть описаны в блоке, представляющем данное тело, описаны вне этого блока или же могут быть включены в список формальных параметров. Поэтому приведенная выше процедура идентификации для таких идентификаторов несколько усложняется. Интересно отметить, что в первом из приведенных контекстных условий вхождение идентификатора в совокупность спецификаций играет роль использующего вхождения, а во втором — определяющего вхождения.

Рассмотрим контекстные условия, связанные с нестандартными операциями языка АЛГОЛ-68. Как уже упоминалось, обозначающий вводимую операцию индикатор должен входить в две конструкции, описывающие операцию: в описание приоритета и в описание операции. В противоположность большинству других конструкций подобного типа в одном и том же блоке могут встречаться несколько описаний операции, обозначаемой одним и тем же индикатором, — это допускается для того, чтобы можно было описывать операции, выполнимые над операндами многих различных видов. Таким образом, индикаторы операций не связаны условием единственности в описаниях операций, но связаны им в описаниях приоритета. Контекстные условия второй группы для индикаторов операций таковы. Во-первых, по каждому использующему вхождению индикатора операции должно найтись вхождение данного индикатора в описание операции, которое, таким образом, является в этом случае определяющим. Во-вторых, по каждому вхождению индикатора в описание операции должно найтись вхождение данного индикатора в описание приоритета. Здесь вхождение индикатора в описание операции рассматривается как использующее. Алгоритмы идентификации для данных контекстных условий аналогичны приведенному выше для идентификаторов АЛГОЛ-60.

Алгоритмы идентификации играют важную роль при переводе программ, написанных на алгоритмических языках, на машинные языки. Они необходимы не только для проверки контекстных условий второго типа, но и для получения необходимой для трансляции информации об объектах, именуемых идентификаторами или индикаторами, находящимися в позиции использующего вхождения. Так, при программировании на машинном языке алгоритмов вычисления значений выражений нужно знать виды значений, являющихся операндами соответствующих операций, при вызове процедур обязательно нужно учитывать виды значений, именуемых идентификаторами, входящими в фактические параметры. При обработке переменных с индексами нужно знать не только вид элементов соответствующих массивов, но и их размерность и т. п. В некоторых языках, в которых многие контекстные условия второго типа отсутствуют, алгоритмы идентификации выполняются исключительно с целью получения информации об использовании встречающихся в программах идентификаторов. К таким языкам относятся, например, ФОРТРАН IV и PL/I, характерной особенностью которых является то, что идентификаторы простых переменных не обязательно должны описываться. Если описание какого-либо идентификатора в программе, написанной на одном из этих языков, отсутствует, вид переменной, которую он именуется, определяется согласно следующему простому правилу: идентификаторы, начинающиеся с одной из букв I, J, K, L, M, N, обозначают целые переменные, а все остальные идентификаторы — вещественные переменные. Таким образом, контекстные условия второго типа для идентификаторов простых переменных в этих языках отсутствуют, а идентификация необходима лишь для того, чтобы определить, имеется ли явное описание анализируемого идентификатора в программе, или в соответствии с так называемым принципом умолчания для обработки этого идентификатора нужно пользоваться приведенным выше правилом. В более ранних вариантах языка ФОРТРАН — в ФОРТРАН II и в Basic FORTRAN — явные описания простых переменных отсутствуют, так что выполнять алгоритмы идентифи-

кации для соответствующих идентификаторов не требуется. С идентификацией использующих вхождения идентификаторов и других требующих этой процедуры объектов тесно связана проверка контекстных условий третьего типа, к рассмотрению которых мы и переходим. Контекстные условия этого типа регламентируют соответствие видов величин, входящих в синтаксические конструкции программ. Так, например, в простые арифметические выражения языка АЛГОЛ-60 не могут входить идентификаторы булевских переменных или функций, а внутри булевских выражений идентификаторы целых или вещественных переменных и функций могут использоваться лишь в отношениях. К контекстным условиям третьего типа естественно отнести также задание допустимых соответствий между фактическими параметрами в операторах процедур и формальными параметрами в описаниях соответствующих процедур, включая сравнение их количества. К этому же типу можно отнести и сравнение числа индексов у переменных с индексами и размерности соответствующих массивов. Для проверки большинства контекстных условий третьего типа требуется определить виды всех величин, входящих в анализируемые конструкции, для чего, в свою очередь, необходима идентификация использующих вхождения.

В таких языках, как ФОРТРАН и АЛГОЛ-60, многие контекстные условия третьего типа сравнительно просты. Однако проверки, связанные с использованием процедур, в особенности при наличии формальных параметров, специфицированных как процедуры, могут вызвать немалые трудности. Рассмотрим следующий пример.

Пример 1. Пусть имеется следующая модельная программа, написанная на языке АЛГОЛ-60:

begin

```

real procedure H(X, Y, Z);
    real procedure X, Y, Z; H := X(Y, Z);
real procedure G(X, Y);
    real procedure X, Y; G := X(Y, D);
real procedure F(X, Y);
real procedure X, Y; F := X(1, 10) + Y(1);
real procedure E(X); real X; := X + 1;
real procedure D(X); real X; D := X + 2;
real A;
A := H(G, F, E);

```

end

Данная программа содержит описания пяти процедур-функций, причем в трех из них формальные параметры специфицированы так же, как процедуры-функции. Это обстоятельство значительно усложняет проверку соответствий фактических и формальных параметров. Данная проверка начинается с анализа оператора процедуры $H(G, F, E)$. Непосредственным сравнением убеждаемся, что фактические параметры этого оператора, описанные как процедуры-функции вещественного типа, соответствуют формальным параметрам процедуры H . Однако проверка соответствий на этом не кончается. Подставляя фактические параметры в тело процедуры H , получаем другой оператор процедуры: $G(F, E)$, параметры которого также требуется сравнить с формальными. После того как мы убедимся, что контекстное условие здесь также не нарушено, получим еще один оператор процедуры: $F(E, D)$, анализ которого не обнаружит ошибки. И наконец, подставляя параметры E и D в тело процедуры F , получим операторы процедур $E(1, 10)$ и $D(1)$. Проверяя соответствие параметров, обнаружим, что число фактических параметров в операторе процедуры E не равно числу формальных параметров в описании этой процедуры. Следовательно, в данном случае нарушено контекстное условие, и в программе содержится синтаксическая ошибка, обнаружение которой потребовало довольно сложного анализа.

В языках с более сложной системой видов, таких, как АЛГОЛ-68 и PL/1, контекстные условия третьего типа имеют ряд характерных особенностей. Прежде чем перейти к их рассмотрению, уточним понятие вида. В каком случае различные значения принадлежат одному виду? Наиболее естественное требование для таких значений заключается в их взаимной замещаемости, не приводящей к синтаксическим ошибкам. Так, вместо вещественного или целого числа, а также булевого значения в любой программе на языке АЛГОЛ-60 можно подставить другое значение того же вида, причем синтаксическая ошибка в результате такой замены не возникнет (хотя может возникнуть семантическая). То же можно сказать и о простых переменных в предположении, что для них выполнены контекстные условия второго типа. Замещение целых чисел и переменных на вещественные и обратное замещение также допускаются в большинстве конструкций, семантика которых предполагает автоматическое преобразование чисел из одного вида в другой. Однако операнды операции деления нацело должны быть целыми, поэтому в данной конструкции замещение целых на вещественные невозможно. Невоз-

можно замена одного вида на другой и для отдельных переменных в списке левой части оператора присваивания, так как все переменные списка должны быть одного вида.

Если принять предложенный признак отнесения значений к одинаковому или различным видам, окажется, что традиционное представление о видах языка АЛГОЛ-60 не совсем точно. В самом деле, переменные и числа невзаимозамещаемы — последние не могут находиться в левых частях операторов присваивания, и поэтому нужно считать, что они относятся к различным видам. Эта идея получила развитие при описании языка АЛГОЛ-68, где различаются обычные значения, такие, как целые или вещественные числа, и имена, т. е. значения, которые могут ссылаться на значения других видов. Переменные языка АЛГОЛ-60 как раз и являются именами, которые в зависимости от описания могут ссылаться на целые или вещественные числа или на булевские значения. Аналогичная ситуация имеет место и для массивов различной размерности. Если например, массив A описан как одномерный, а B — как двухмерный, в переменной с индексом $A[i]$ нельзя произвести замену идентификатора A на идентификатор B . Поэтому массивы различной размерности также следует относить к различным видам. Итак, мы можем сделать вывод, что в языке АЛГОЛ-60 имеются обозначаемые идентификаторами имена и значения, не являющиеся именами, которые естественно назвать константами, и учитывая возможность описания массивов любой размерности, множество видов значений этого языка естественнее считать бесконечным.

Обсудим теперь структуру и способы описания значений различных видов в языке АЛГОЛ-68. Все значения этого языка можно разделить на простые и составные. К первым относятся такие значения, которые всегда обрабатываются целиком и не делятся на части. Это, например, целые, вещественные, булевские, символьные значения, а также имена, ссылающиеся на значения простых видов. Ко второй группе значений можно отнести массивы и структуры, т. е. составные значения, элементы которых могут принадлежать различным видам. В эту же группу входят имена, ссы-

лающиеся на составные значения. Множество видов в языке АЛГОЛ-68 бесконечно, причем существуют строгие синтаксические правила описания значений любого вида. Существуют два главных способа порождения бесконечного числа видов. Во-первых, в языке определены не только имена, ссылающиеся на значения, не являющиеся именами, но и имена, так сказать, высших порядков, т. е. такие, которые ссылаются на имена, ссылающиеся в свою очередь на имена или другие значения. Имена различных порядков следует считать принадлежащими к различным видам, и так как могут использоваться имена любых порядков, видов имен бесконечно много. Во-вторых, в языке имеется бесконечно много видов составных значений: к различным видам принадлежат массивы с различными размерностями, а также структуры, элементы (или, как их называют в языке АЛГОЛ-68, поля) которых принадлежат различным видам. Существенной особенностью языка является также возможность обозначения с помощью идентификаторов не только имен, но и других значений. Отметим, наконец, что процедуры в этом языке также считаются значениями особых видов, над которыми могут производиться определенные действия.

Перечисленные особенности языка АЛГОЛ-68 приводят к тому, что его контекстные условия третьего типа значительно сложнее соответствующих условий более простых языков. Значительно усложняются и алгоритмы идентификации. Приведем два примера условий третьего типа. В языке АЛГОЛ-60 контекстным условием, связанным с оператором присваивания, является следующее хорошо известное правило: если в левой части оператора находится целая или вещественная переменная, то в правой его части должно стоять арифметическое выражение, иначе говоря, конструкция, вырабатывающая целое или вещественное число; если же вид переменной левой части — булевский, справа должно находиться булевское выражение — конструкция, вырабатывающая булевское значение. В языке АЛГОЛ-68 конструкция, соответствующая оператору присваивания, называется присваиванием. С ним связано похожее, но более сложное контекстное условие: если правая часть присваивания выраба-

тывает значение некоторого вида, то его левая часть должна вырабатывать имя этого вида. Проверка этого условия затрудняется, во-первых, из-за наличия бесконечного множества видов и, во-вторых, в связи с наличием в языке специальных операций, которые называются приведениями и позволяют изменять вид значений, вырабатываемых различными конструкциями.

Пусть, например, идентификаторы A и B обозначают имена, ссылающиеся на значения одного и того же вида. Тогда, на первый взгляд, присваивание $A := B$ содержит синтаксическую ошибку, так как не выполнено только что сформулированное контекстное условие (левая и правая части вырабатывают один и тот же вид). Однако на самом деле это не так. Специальная операция, называемая разыменованием, которая явно в конструкции не указывается, но должна быть выполнена при вычислении правой части, позволяет вместо имени B взять значение, на которое это имя ссылается, после чего виды значений левой и правой частей будут удовлетворять контекстному условию. В качестве второго примера рассмотрим контекстное условие, связанное с конструкцией языка, называемой совместным предложением, вырабатывающим значение. Совместное предложение состоит из нескольких конструкций, разделенных запятыми, каждая из которых вырабатывает некоторое значение. Из этих значений может быть образовано структурное значение. Если же конструкции, входящие в совместное предложение, вырабатывают значения одного и того же вида, то из них может быть образован также массив (или, используя терминологию языка, мультизначение). Таким образом, значение, вырабатываемое совместным предложением,— это либо структурное значение, либо мультизначение. Предположим, что совместное предложение находится в такой позиции, что его значение должно рассматриваться как мультизначение. Это может быть, например, в случае, если оно находится в правой части присваивания, левая часть которого вырабатывает имя мультизначения. Тогда нужно проверить, действительно ли конструкции, входящие в совместное предложение, могут вырабатывать значения одного вида. Это и есть контекстное условие, связанное с сов-

местными предложениями. Несмотря на его кажущуюся простоту, алгоритм проверки довольно сложен: нужно учитывать возможные приведения (разыменование — лишь одно из них), при этом принимая во внимание так называемую балансировку, согласно которой возможность применения тех или иных приведений к одной конструкции зависит от приведений, примененных к другим конструкциям совместного предложения.

Для того, чтобы в формально задаваемую часть синтаксиса языка АЛГОЛ-68 можно было включить контекстные условия третьего типа, при описании языка вместо обычной УКС-грамматики была принята более сложная грамматика ван Вейнгаардена, которая подробно рассматривается в § 3 настоящей главы.

Контекстные условия четвертого типа имеют вспомогательное значение. К ним относятся различные количественные ограничения, большая часть которых налагается не на эталонные языки, не связанные с конкретными реализациями, а на так называемые языки конкретных представлений, т. е. входные языки различных трансляторов. Например, в некоторых входных языках типа АЛГОЛ-60 допускается лишь ограниченная глубина вложенности блоков, или ограничиваются размерности используемых массивов — соответствующие правила являются контекстными условиями этой группы.

К какому классу формальных языков относятся языки программирования с контекстными условиями? Какие методы формального описания контекстных условий уже используются или могут быть предложены? Как происходит анализ программ с учетом контекстных условий и какие анализаторы могут быть для них предложены? Обсуждению этих вопросов посвящены следующие три параграфа настоящей главы.

§ 2. Описание контекстных условий с помощью программных грамматик

Как уже отмечалось выше, контекстные условия языков программирования не описываются средствами контекстно-свободных грамматик. Для их формального задания требуется аппарат, превышающий по

своей порождающей мощности эти грамматики. Из общих соображений более или менее очевидно, что языки программирования являются НС-языками. Это может быть показано на основании теоремы 4, рассмотренной нами в § 1, гл. II, так как проверка контекстных условий с помощью ЛО-автоматов не является чрезмерно сложной задачей. С другой стороны, построение НС- или неукорачивающей грамматики, описывающей синтаксис, скажем, языка АЛГОЛ-60 — очень сложная задача. В примере 7, § 2, гл. I, нами был упомянут язык $L = \{x_1 b x_2 b \dots x_{p-1} b x_p\}$, $x_i \in \{a_1, a_2, \dots, a_n\}^*$, $x_i \neq x_j$ при $i \neq j$ и $p > 0$. Даже такой сравнительно простой язык, приближенно моделирующий условие единственности описания идентификаторов, описывается сложной неукорачивающей грамматикой, имеющей большое количество схем правил.

В связи с трудностями описания синтаксиса языков программирования с контекстными условиями средствами классических грамматик, рассмотренных нами в гл. I, для этой цели обычно предлагаются формальные системы, представляющие собой различные обобщения этих грамматик. Классы языков, порождаемых многими из этих систем, шире класса НС-языков. Так, например, порождающая мощность уже упоминавшихся нами грамматик ван Вейнгаардена равна мощности классических грамматик произвольного вида. Это утверждение будет нами доказано в следующем параграфе. Одним из интересных и сравнительно удобных средств формального представления полного синтаксиса языков программирования являются так называемые *программные грамматики* [34]. Правила вывода этих грамматик имеют тот же вид, что и у классических грамматик, однако в отличие от последних на каждом шаге вывода правила, которые могут быть применены, зависят не только от цепочек вывода, но и от того, какие правила применялись на предыдущих шагах. Возможность такого управления выводом значительно облегчает задачу описания контекстных условий.

Приведем теперь точное определение программных грамматик, после чего рассмотрим ряд приемов, при помощи которых средствами этих грамматик могут

быть описаны контекстные условия перечисленных выше типов.

Определение. Программной грамматикой называется следующая пятерка объектов:

$$G_{pr} = \langle V_T, V_A, I, J, S \rangle,$$

где V_T , V_A и I , так же как в классических грамматиках, являются, соответственно, конечным множеством терминальных символов, конечным множеством нетерминальных символов и начальным символом; J — конечное множество целых положительных чисел, которое мы будем называть *множеством меток*; S — конечное множество правил вывода.

Между правилами вывода из S и метками из J установлено взаимно-однозначное соответствие — каждое правило помечено меткой, отличной от меток всех остальных правил. Правило вывода имеет вид:

$$r) \varphi \rightarrow \psi \mid W_1 \mid W_2,$$

где r — метка, соответствующая данному правилу; $\varphi \rightarrow \psi$ — ядро правила, φ и ψ — цепочки из $(V_T \cup V_A)^*$; W_1 и W_2 — подмножества множества меток.

Применение правила указанного вида к некоторой промежуточной цепочке ω вывода в G_{pr} состоит в следующем. Если ω содержит вхождение подцепочки φ , то левое вхождение φ в цепочке ω заменяется на ψ , после чего к полученной цепочке вывода применяется одно из правил, помеченных метками из W_1 . Если, наоборот, ω не содержит вхождения φ , то цепочка вывода не меняется и к ней применяется некоторое правило, помеченное меткой из W_2 . Подмножества W_1 и W_2 в некоторых случаях могут быть пустыми. Если необходимо произвести выбор метки следующего правила из пустого подмножества, вывод не может быть продолжен и возникает тупиковая ситуация. В язык $L(G_{pr})$, порождаемый грамматикой G_{pr} , входят, так же как и в случае классических грамматик, все терминальные цепочки, для которых в G_{pr} существует вывод из начального символа. При этом на первом шаге вывода к начальному символу может применяться любое из имеющихся для него правил.

Так же как и классические грамматики, программные грамматики разбиваются на ряд классов в зависи-

мости от вида ядер их правил вывода. Можно рассматривать программные грамматики произвольного вида, неукорачивающие, т. е. такие, все ядра которых имеют тот же вид, что и правила классических неукорачивающих грамматик, и т. д. Классы языков, порождаемых программными грамматиками, шире или совпадают с соответствующими классами, порождаемыми классическими грамматиками. Известно, например, что класс языков, порождаемых контекстно-свободными программными грамматиками, шире класса КС-языков, в то время как классы языков, порождаемых неукорачивающими программными и классическими грамматиками, совпадают.

Рассмотрим метод построения неукорачивающей программной грамматики, порождающей в точности все синтаксически правильные программы языка АЛГОЛ-60. Как будет видно, этот метод является довольно общим и его можно использовать для построения грамматик, порождающих другие языки программирования, схожие по своей структуре с языком АЛГОЛ-60. Достоинством этого метода можно считать также и то, что используемый им аппарат неукорачивающих программных грамматик по своей порождающей мощности не превосходит аппарат классических неукорачивающих грамматик. Вывод программы на языке АЛГОЛ-60 в программной грамматике удобно представить в виде циклического процесса, при котором порождение чередуется с проверкой контекстных условий. Сначала порождается промежуточная форма программы, в которой некоторые синтаксические конструкции еще отсутствуют (например, внутренние блоки), а те, которые в ней представлены, состоят из особых нетерминальных символов — двойников. Такие символы уже использовались нами ранее в ряде примеров. Их характерным свойством является то, что они однозначно соответствуют терминальным символам — каждый из них может порождать лишь соответствующий ему терминальный символ. Заметим также, что все нетерминальные символы, в том числе и двойники, должны в процессе вывода использоваться в различных модификациях, причем множества выводимых из них терминальных цепочек не зависят от их модификации. После окончания первого этапа вывода

производятся проверки контекстных условий. Каждая такая проверка заключается в следующем. Сначала выделяются конструкции, участвующие в проверке. Выделение производится путем изменения модификации символов, из которых эти конструкции состоят. Далее производится сравнение выделенных конструкций, причем способ сравнения зависит от конкретного контекстного условия: сравнение может производиться на совпадение, на несовпадение, на наличие в выделенных конструкциях равного количества некоторых символов и т. п. Если хоть одна проверка даст отрицательный результат, возникнет тупик, означающий, что данный вывод не удовлетворяет контекстным условиям. Если результаты всех проверок положительны, производится вывод некоторых из еще не построенных конструкций, после чего снова проверяются контекстные условия. Этот процесс продолжается до тех пор, пока не будет получена вся программа. На заключительном этапе вывода все двойники заменяются соответствующими им терминальными символами.

Все указанные действия сравнительно легко описываются средствами неукорачивающих программных грамматик. Поскольку в программных грамматиках определяется порядок применения правил вывода, оказывается возможным сравнивать различные конструкции, не перемещая символы этих конструкций. Это значительно облегчает построение программных грамматик для языков программирования и является их основным преимуществом по сравнению с классическими грамматиками. Приведем примеры, иллюстрирующие способы представления в программных грамматиках действий, производимых при проверке контекстных условий. Правила вывода в примерах будем указывать в виде схем, в которых используются переменные индексы. Например, схема

$$r) A_i \rightarrow B_i | W_1 | W_2 \text{ при } i=1, \dots, k$$

означает, что в грамматику включаются правила:

$$r) A_1 \rightarrow B_1 | W_1 | r+1$$

$$r+1) A_2 \rightarrow B_2 | W_1 | r+2$$

$$r+k-1) A_k \rightarrow B_k | W_1 | W_2.$$

Пусть имеются схемы

$$\begin{array}{l} r) A_i \rightarrow B_i \mid s(\text{при том же } i) \mid W_2 \\ s) C_{ij} \rightarrow D_{ij} \mid W_1 \qquad \qquad \qquad \mid W_2 \end{array}$$

и ищется правило из схемы r , как это указано выше. Если некоторое правило из r при $i=i_0$ применимо, то следует перейти к правилам из схемы s , зафиксировав у них значение соответствующего индекса, т.е. к схеме вида $C_{i_0j} \rightarrow D_{i_0j}$. Если указание на одинаковое значение индекса отсутствует, то индексы у схемы, на которую производится переход, не фиксируются.

Пример 1. Пусть цепочка вывода имеет вид

$$\omega = \xi_1 X \eta \xi_2 Y \zeta \xi_3,$$

где η и ζ — цепочки символов из множеств $\{B_i\}$ и $\{C_i\}$ соответственно, X и Y — нетерминальные символы, ξ_1 , ξ_2 и ξ_3 — цепочки, не содержащие символов X и Y и не начинающиеся с символов B_i и C_i .

Будем считать, что контекстное условие для ω выполнено, если $\eta = B_{i_1} B_{i_2} \dots B_{i_m}$ и $\zeta = C_{i_1} C_{i_2} \dots C_{i_m}$. Это условие проверяется при помощи следующих правил:

- | | |
|---|----------------------|
| $r_1) XB_i \rightarrow X\bar{B}_i \mid r_2$ (при том же i) | \mid не определено |
| $r_2) YC_i \rightarrow Y\bar{C}_i \mid r_3$ | \mid тупик |
| $r_3) \bar{B}_i B_j \rightarrow B_i \bar{B}_j \mid r_4$ (при тех же i и j) | $\mid r_6$ |
| $r_4) \bar{C}_i C_j \rightarrow C_i \bar{C}_j \mid r_3$ | \mid тупик |
| $r_5) \bar{C}_i C_j \rightarrow C_i C_j \mid$ тупик | $\mid r_6$ |
| $r_6) \bar{B}_i \rightarrow B_i \mid r_7$ | \mid не определено |
| $r_7) \bar{C}_i \rightarrow C_i \mid$ выход | \mid не определено |

Посмотрим, как происходит проверка контекстного условия. При помощи правил из r_1 выделяется первый символ цепочки η (для этого используется другая модификация этого символа). Далее проверяется, имеет ли первый символ цепочки ζ тот же индекс, — для этого служит схема r_2 . Если правило из r_2 неприменимо, т.е. индекс первого символа цепочки ζ отличен от индекса выделенного символа из η , контекстное условие нарушено и возникает тупиковая ситуация. В противном случае с помощью схемы r_3 выделяется следующий символ из η , а возможность применения правила из r_4 говорит о том, что его индекс совпадает с индексом соответствующего символа из ζ . Если правило из r_4 неприменимо, то условие нарушено и снова возникает тупик. Далее правила из r_3 и r_4 применяются поочередно, пока не наступит тупик или

не будет исчерпана цепочка η . Если имеет место второй случай, применяются правила из r_5 , которые нужны для того, чтобы проверить, не является ли η левой подцепочкой цепочки ζ . В случае, если ни одно правило из схемы r_5 неприменимо, применяются по одному разу правила из схем r_6 и r_7 , которые восстанавливают первоначальные модификации символов. После этого производится выход, означающий, что контекстное условие для данной цепочки ω выполнено. Заметим, что правила из схем r_1 , r_6 и r_7 всегда применимы, поэтому подмножества меток для случая, когда их применение оказалось невозможным, могут не определяться.

Пример 2. Пусть цепочка вывода ω имеет вид, указанный в примере 1, но требуется проверить обратное контекстное условие, т.е. такое, которое считается выполненным, если $\xi \neq C_{i_1}C_{i_2} \dots C_{i_m}$. Это условие проверяется при помощи следующих правил:

- | | |
|---|---------------|
| $r_1) XB_i \rightarrow X\bar{B}_i \mid r_2$ (при том же i) | не определено |
| $r_2) YC_i \rightarrow Y\bar{C}_i \mid r_3$ | r_7 |
| $r_3) \bar{B}_iB_j \rightarrow B_i\bar{B}_j \mid r_4$ (при тех же i и j) | r_5 |
| $r_4) \bar{C}_iC_j \rightarrow C_i\bar{C}_j \mid r_5$ | r_6 |
| $r_5) \bar{C}_iC_j \rightarrow C_iC_j \mid r_7$ | тупик |
| $r_6) \bar{C}_i \rightarrow C_i \mid r_7$ | не определено |
| $r_7) \bar{B}_i \rightarrow B_i \mid$ выход | не определено |

Ядра всех этих правил полностью совпадают с ядрами правил из примера 1. Изменены лишь подмножества меток. Если контекстное условие формулируется для случая, когда $B_i = C_i$, в приведенных выше правилах C_i нужно заменить на B_i , на \bar{C}_i на \bar{B}_i .

Пример 3. Пусть цепочка вывода имеет вид

$$\omega = \xi_1 Z \eta_1 \xi_2 Z \eta_2 \dots \xi_m Z \eta_m \xi_{m+1},$$

где Z — нетерминальный символ, η_j — цепочки символов из подмножества $\{B_i\}$, ξ_j — цепочки символов, не содержащие символа Z и не начинающиеся с символов B_i .

Будем считать, что контекстное условие для ω выполнено, если $\eta_j \neq \eta_k$ при $j \neq k$. Проверка такого контекстного условия осуществляется путем попарного сравнения цепочек η_j , как это показано в примере 2. Для выделения очередной пары вхождения символа Z , предшествующие выделяемым цепочкам η_j , заменяются его модификациями X и Y . Кроме того, для организации перебора пар необходимы еще две модификации этого символа: Z_1 и Z_2 . Выделение пар осуществляется при помощи следующих правил:

- | | | |
|------------------------|--------------|---------------|
| 1) $Y \rightarrow Z_1$ | 3 | 2 |
| 2) $Z \rightarrow X$ | 3 | 6 |
| 3) $Z \rightarrow Y$ | на сравнение | 4 |
| 4) $X \rightarrow Z_2$ | 5 | не определено |
| 5) $Z_1 \rightarrow Z$ | 5 | 2 |
| 6) $Z_2 \rightarrow Z$ | 6 | выход |

После каждого успешного сравнения выделенных пар нужно применять правило 1. Выход означает, что перебор пар закончен.

Пример 4. Пусть цепочка вывода имеет вид

$$\omega = \zeta_1 A_1 \xi_1 U \eta_1 \dots \xi_m U \eta_m \xi_{m+1} A_2 \xi_2,$$

где A_1, A_2 и U — нетерминальные символы; ζ_1 и ξ_2 — цепочки символов, не содержащие A_1 и A_2 ; η_j — цепочки символов из $\{B_i\}$; ξ_j — цепочки символов, не содержащие A_1, A_2 и не начинающиеся с символов B_i .

Будем считать, что требуется проверить некоторые контекстные условия для цепочек $\eta_1, \eta_2, \dots, \eta_m$, т. е. для всех таких цепочек символов из $\{B_i\}$, перед которыми имеются вхождения символа U и которые находятся между A_1 и A_2 (заметим, что подобные цепочки могут содержаться также в ζ_1 и ξ_2). Для подготовки указанных цепочек к проверке заменим все вхождения символа U между A_1 и A_2 его модификацией Z , которая ранее в ω не использовалась. Это можно выполнить при помощи следующих правил:

- 1) $A_1 \rightarrow \bar{A}_1 \mid 2 \mid$ не определено
- 2) $\bar{U} \rightarrow \bar{Z} \mid 3 \mid 3$
- 3) $\bar{L}A_2 \rightarrow LA_2 \mid$ выход $\mid 4$
- 4) $\bar{L}K \rightarrow L\bar{K} \mid 2 \mid$ не определено

Эти правила должны рассматриваться как схемы, в которые вместо L и K могут быть подставлены любые нетерминальные символы. Метод замены U на Z заключается в просмотре всей подцепочки цепочки ω , ограниченной символами A_1 и A_2 . Для просмотра используются надчеркнутые символы. Если оказался надчеркнутым символ U , он заменяется на \bar{Z} . При окончании просмотра надчеркнутая модификация символа, предшествующего A_2 , заменяется его первоначальной модификацией.

Пример 5. В некоторых случаях вместо проверки контекстных условий для уже порожденных конструкций программ удобно порождать эти конструкции таким образом, чтобы в них заведомо были выполнены эти контекстные условия. В данном примере мы рассмотрим один из возможных приемов осуществления такого управляемого вывода. Пусть цепочка вывода имеет вид

$$\omega = \xi_1 Z \eta \xi_2 T \xi_3,$$

где Z и T — нетерминальные символы; η — цепочка символов из $\{B_i\}$; ξ_1, ξ_2, ξ_3 — цепочки, не содержащие Z и T , причем ξ_2 не начинается с символов B_i .

Будем считать, что на данном этапе вывода требуется заменить символ T цепочкой ζ , подобной цепочке η , т. е. если

$$\eta = B_{i_1} B_{i_2} \dots B_{i_m}, \text{ то } \zeta = C_{i_1} C_{i_2} \dots C_{i_m},$$

где C_i — символы некоторого множества. Такой вывод осуществляется при помощи следующих правил:

- $r_1) ZB_i \rightarrow \bar{Z}B_i \mid r_2, r_3 \text{ (при том же } i) \mid \text{ не определено}$
 $r_2) T \rightarrow C_i T \mid r_4 \mid \text{ не определено}$
 $r_3) T \rightarrow C_i \mid r_5 \mid \text{ не определено}$
 $r_4) \bar{B}_j B_i \rightarrow B_j \bar{B}_i \mid r_2, r_3 \text{ (при том же } i) \mid \text{ не определено}$
 $r_5) \bar{B}_i B_j \rightarrow B_i B_j \mid \text{ тупик} \mid r_6$
 $r_6) \bar{B}_i \rightarrow B_i \mid \text{ выход} \mid \text{ не определено}$

Для порождения цепочки ζ символы цепочки η последовательно выделяются слева направо надчеркиванием и на каждом шаге к формируемой цепочке ζ приписывается справа соответствующий выделенному символ. Для последнего действия можно пользоваться схемами правил r_2 или r_3 (для последнего символа). Если правило из схемы r_3 было использовано раньше времени, возникнет тупик. •

Любое контекстное условие языка АЛГОЛ-60 может быть проверено или учтено при выводе программы в программной грамматике с помощью приемов, описанных в примерах. При этом, конечно, в некоторых случаях указанные в них правила вывода должны быть несколько преобразованы. Так, например, в примерах предполагается, что символы, играющие роль ограничителей (X и Y в примерах 1 и 2, Z в примере 3 и т. д.), единственны. В действительности в соответствующих позициях могут находиться символы из некоторого конечного множества (множества ограничителей). В примерах 1 и 2 показаны приемы, которые могут использоваться для проверки условия единственности описания идентификаторов и для идентификации. Иногда вместо тупиков здесь должен осуществляться выход для выделения и сравнения следующих пар конструкций. Прием, который показан в примере 4, удобно применять для выделения определяющих вхождений идентификаторов того блока, в котором производится идентификация. Пусть нужный блок уже определен с помощью особой модификации операторных скобок (в примере — символы A_1 и A_2). Тогда данный прием позволяет изменить модификацию символов, расположенных перед каждым определяющим вхождением внутри этого блока

(в примере их представляют символы U до изменения, и Z — после изменения модификации). Если идентификатор описан вне того блока, в котором находится его использующее вхождение, то выделяемые для идентификации области могут оказаться разорванными (внешние блоки без входящих в них внутренних блоков). Нетрудно, однако, видеть, что эти и некоторые другие изменения, которые нужно внести в описанные приемы при составлении грамматик конкретных языков программирования, не представляют особых трудностей.

Отметим некоторые особенности правил, при помощи которых может быть осуществлен вывод *промежуточных форм* программ, т. е. цепочек, анализируемых для проверки контекстных условий. Ядрами этих правил могут служить несколько измененные нормальные формы Бекуса, описывающие стандартный синтаксис языков. Они изменяются таким образом, чтобы при выводе конструкций, на которых проверяются контекстные условия (такими конструкциями являются, например, простые переменные, идентификаторы массивов, метки, формальные параметры процедур), использовались различные модификации символов-двойников. При этом каждая конкретная модификация должна однозначно определять тип данной конструкции, а также ее позицию в программе (определяющее или использующее вхождение). Такой способ вывода промежуточной формы позволяет облегчить проверки контекстных условий. Пусть, например, требуется произвести идентификацию использующего вхождения идентификатора. Для этого достаточно выделить в блоке все определяющие, т. е. представленные символами особой модификации вхождения идентификаторов, и проверить, совпадает ли данный идентификатор с одним из выделенных. Если совпадающий идентификатор найден, условие второго типа выполнено. Тогда можно проверить контекстное условие третьего типа, т. е. сравнить модификации, используемые для представления типов данного и найденного идентификаторов. Определяя соответствующий порядок применения правил вывода, легко точно определить моменты перехода от порождения к проверкам,

Промежуточные формы должны представлять собой программы, в которых ряд блоков представлен в символах-двойниках, а другие еще не включены в программу, но их место зафиксировано специальным символом. При переходе от одной промежуточной формы к следующей производится вывод одного из таких еще не построенных блоков. Целесообразно принять следующий порядок вывода блоков:

1) Порождается внешний блок программы.

2) Рассматривается порожденный блок. Проверяется, включает ли он вхождения специального символа, указывающего на еще не построенные внутренние блоки. Если такие вхождения имеются, порождается самый левый из внутренних блоков, после чего управление передается в начало данного пункта. В противном случае выполняется пункт 3.

3) Символы, составляющие порожденный блок, переводятся в другую модификацию. После этого рассматривается объемлющий блок и для него выполняется пункт 2. Если объемлющего блока нет, т. е. порожденный блок является самым внешним, происходит замена всех двойников на соответствующие им терминальные символы, после чего вывод заканчивается.

Для выполнения данной процедуры необходимо иметь три группы модификаций:

а) группу пассивных модификаций — для блоков, вывод которых полностью закончен;

б) группу активных модификаций — для рассматриваемого в данный момент блока;

в) группу нейтральных модификаций — для блоков, объемлющих рассматриваемый.

Такая процедура порождения позволяет избежать подсчета операторных скобок для определения областей действия. Учитывая неограниченную глубину вложенности блоков и наличие параллельных блоков (т. е. непересекающихся, но имеющих один и тот же объемлющий блок), определение областей действия без использования описанного поэтапного вывода программ средствами программных грамматик осуществить затруднительно. Аналогичный метод может использоваться для вывода переменных с индексами, так как индексные выражения могут, в свою очередь,

содержать переменные с индексами. Показанный в примере 5 управляемый вывод удобно использовать при построении списка фактических параметров, соответствующего списку формальных параметров в описании процедуры, а также для того, чтобы список индексов содержал число индексов, равное размерности массива.

Программная грамматика для языка АЛГОЛ-60, при разработке которой использовались перечисленные приемы, состоит примерно из 200 схем правил вывода. Ядра правил для порождения получены из нормальных форм Бекуса с учетом тех изменений, которые рассматривались выше. Правила для проверки контекстных условий имеют тот же вид, что и в приведенных примерах. В этой грамматике учитываются основные контекстные условия языка:

1) В каждом блоке без внутренних блоков любой идентификатор должен быть описан не более одного раза. Это условие проверяется также для целых, представляющих метки.

2) По любому использующему вхождению идентификатора (или целого, представляющего метку) в данном или объемлющих блоках должно найтись описание этого идентификатора, причем тип, определенный в описании, должен совпадать с типом данного использующего вхождения.

3) Все переменные в левых частях операторов присваивания должны быть переменными одного типа.

4) Количество индексов у переменных с индексами должно совпадать с числом граничных пар в описаниях соответствующих массивов.

5) Количество и типы фактических параметров, указанных в операторах процедур, должны совпадать с количеством и типами формальных параметров, приведенных в описаниях соответствующих процедур.

6) Фактический параметр, который соответствует формальному параметру, вызываемому по наименованию и встречающемуся в теле процедуры в левых частях операторов присваивания, обязан быть переменной (не выражением).

7) Идентификаторы, входящие в выражения для границ в описаниях массивов, должны быть описаны в одном из объемлющих блоков.

Указанный метод построения программной грамматики может использоваться при описании синтаксиса языка АЛГОЛ-68. Учитывая, однако, что множество описателей видов (типов) значений этого языка бесконечно, приходится отказаться от представления видов модификациями символов. В этом случае несколько усложняется проверка контекстных условий третьего типа, так как определить виды значений, именуемых идентификаторами, можно, лишь анализируя описатели видов в их определяющих вхождениях.

Основным недостатком описанного метода формального задания контекстных условий является то, что для программных грамматик нелегко разработать эффективные анализаторы. Одним из приемлемых анализаторов может служить рассмотренный в § 3 гл. III глобальный анализатор, который нетрудно обобщить для случая программных грамматик. В случае неукорачивающих ядер процедура должна ставить в соответствие подцепочкам анализируемой цепочки не нетерминальные символы, а цепочки, находящиеся в левых частях правил вывода. Наличие строгого порядка применения правил в этих грамматиках не усложняет, а, наоборот, упрощает анализ, так как сокращается число возможных вариантов. Наибольшие трудности при использовании такого обобщенного глобального анализатора для анализа программ возникают при проверке контекстных условий. Эти проверки выполняются довольно нераационально в отношении времени. Однако эффективность анализатора можно существенно повысить, если произвести до его применения то предварительное преобразование программы, о котором уже упоминалось в § 2, гл. I: составить таблицу идентификаторов и все их вхождения в программу заменить ссылками на соответствующие строки этой таблицы. Так как глобальный анализатор может рассматривать эти ссылки как терминальные символы, идентификация значительно упрощается.

Важно также определить работу анализатора так, чтобы идентификация использующих вхождений одного идентификатора внутри одного блока производилась один раз.

§ 3. Грамматики ван Вейнгаардена

В § 1 настоящей главы мы уже упоминали *грамматики ван Вейнгаардена*. Грамматика этого типа использована для описания синтаксиса языка АЛГОЛ-68, причем с ее помощью формально представлены контекстные условия третьего типа этого языка. Грамматика ван Вейнгаардена представляет собой, по существу, две связанные друг с другом грамматики. Одна порождает правила вывода из их схем, другая — сам язык. Вывод в грамматике ван Вейнгаардена аналогичен выводу в классических грамматиках — он начинается с начального нетерминального символа и заканчивается терминальной цепочкой. Ниже мы приведем строгое определение этих грамматик, сравним их с классическими грамматиками, а также рассмотрим, как они используются для описания контекстных условий третьего типа языка АЛГОЛ-68 [2].

Определение. Грамматика ван Вейнгаардена — это две связанные друг с другом грамматики, которые называются *метаграмматикой*, или *грамматикой метаязыка*, и *строгой грамматикой*, или *грамматикой строгого языка*:

$$G_v = \langle G_m, G_{st} \rangle.$$

G_m представляет собой УКС-грамматику без выделенного в ней начального символа:

$$G_m = \langle V_{tm}, V_{am}, S_m \rangle.$$

Она может рассматриваться также как множество УКС-грамматик $\{G_m(A_i)\}$, для задания каждой из которых достаточно объявить начальным некоторый нетерминальный символ A_i из множества V_{am} .

Грамматика G_{st} является УКС-грамматикой с бесконечными множествами нетерминальных символов и правил вывода. Она задается в следующем виде:

$$G_{st} = \langle V_{tsi}, F, I_f, S_{st} \rangle.$$

Здесь F — конечное множество непустых цепочек над объединением терминального и нетерминального словарей метаграмматики, которые в дальнейшем будем называть *формами*: $F \subset (V_{tm} \cup V_{am})^* \setminus \{\lambda\}$; V_{tsi} — конечное множество терминальных символов

строгой грамматики; I_f — некоторая форма, не содержащая символов из V_{AM} ; S_{st} — конечное множество схем правил вывода; каждая схема имеет вид

$$\beta = \beta_1, \beta_2, \dots, \beta_n,$$

где β — некоторая форма из F ; β_i — форма из F или терминальный символ строгой грамматики.

Остановимся на смысле введенных обозначений. Каждая форма из F является обобщенным представлением, возможно, бесконечного подмножества нетерминальных символов строгой грамматики. Сами нетерминальные символы записываются в виде непустых цепочек терминальных символов метаграмматики G_M : если η — нетерминальный символ строгой грамматики, то $\eta \in V_{TM}^* \setminus \{\lambda\}$. Правила вывода строгой грамматики получаются из схем заменой всех входящих в них форм выводимыми из этих форм нетерминальными символами строгой грамматики. Для того, чтобы вывести из формы β некоторый нетерминальный символ, следует все вхождения символов из V_{AM} в β заменить их терминальными порождениями в грамматике G_M . При этом все вхождения одного и того же символа нужно заменить одним и тем же порождением этого символа. Пусть, например, β — форма, в которую входят два нетерминальных символа метаграмматики: $\beta = x_1 \alpha_1 x_2 \alpha_2 x_3 \alpha_1 \dots x_k \alpha_2 x_{k+1}$, где $x_i \in V_{TM}^*, i = 1, 2, \dots, k+1$; $\alpha_j \in V_{AM}, j = 1, 2$. Тогда льюбая цепочка вида $x_1 \eta_1 x_2 \eta_2 x_3 \eta_1 \dots x_k \eta_2 x_{k+1}$, где $\alpha_j \xRightarrow{G_M} \eta_j$

и $\eta_j \in V_{TM}^*, j = 1, 2$, представляет нетерминальный символ из подмножества, определяемого формой β . Если форма из F не содержит вхождений нетерминальных символов метаграмматики, она определяет единственный нетерминальный символ строгой грамматики, представление которого совпадает с представлением этой формы.

Нетерминальные символы, входящие в правила строгой грамматики, должны быть согласованными. Символы называются согласованными, если они получены подстановкой в каждую из соответствующих форм одних и тех же терминальных порождений вместо одинаковых нетерминальных символов мета-

грамматики. Так, например, если $u\alpha v$ и $w\alpha x$ — формы, $\alpha \xRightarrow{G_M} y$ и $\alpha \xRightarrow{G_M} z$, где u, v, w, x, y, z — терминальные цепочки, а α — нетерминальный символ метаграмматики, то uuv и wux — согласованные, а uuv и wzx — несогласованные нетерминальные символы строгой грамматики. Запятые в правых частях схем используются лишь как ограничители терминальных символов и форм.

В язык, порождаемый грамматикой ван Вейнгаардена, входят все цепочки терминальных символов строгой грамматики, выводимые в ней из начального символа применением полученных из схем правил вывода. Приведем пример грамматики ван Вейнгаардена.

Пример 1. Язык $L = \{a^n b^n c^n\}$, $n=0, 1, \dots$ порождается следующей грамматикой ван Вейнгаардена.

Объекты метаграмматики: $V_{TM} = \{\alpha, \beta, \gamma, \delta, \varepsilon\}$; $V_{AM} = \{A\}$; $S_M = \{A \rightarrow \delta A, A \rightarrow \lambda\}$.

Объекты строгой грамматики: $V_{TSt} = \{a, b, c\}$; $F = \{\varepsilon, \alpha, \beta, \gamma, \alpha A, \beta A, \gamma A, \alpha A \delta, \beta A \delta, \gamma A \delta\}$; $I_f = \varepsilon$; в S_{St} входят следующие схемы правил вывода:

- | | |
|--|---------------------------------|
| 1) $\varepsilon \rightarrow \alpha A, \beta A, \gamma A$ | 5) $\alpha \rightarrow \lambda$ |
| 2) $\alpha A \delta \rightarrow \alpha A, a$ | 6) $\beta \rightarrow \lambda$ |
| 3) $\beta A \delta \rightarrow \beta A, b$ | 7) $\gamma \rightarrow \lambda$ |
| 4) $\gamma A \delta \rightarrow \gamma A, c$ | |

Рассматривая возможные выводы терминальных цепочек в строгой грамматике, прежде всего заметим, что в метаграмматике из символа A выводятся цепочки, которые в общем виде можно представить как δ^n при $n=0, 1, \dots$. Поэтому правила вывода, определяемые первыми четырьмя схемами, имеют, соответственно, вид:

- 1) $\varepsilon \rightarrow \alpha \delta^n, \beta \delta^n, \gamma \delta^n$ (вспомним правило
- 2) $\alpha \delta^n \delta \rightarrow \alpha \delta^n, a$ согласованности!)
- 3) $\beta \delta^n \delta \rightarrow \beta \delta^n, b$
- 4) $\gamma \delta^n \delta \rightarrow \gamma \delta^n, c$

На первом шаге вывода применяется некоторое правило вида 1. Затем, используя правила видов 2, 3 и 4, мы можем получить цепочку вида $\alpha, a^n, \beta, b^n, \gamma, c^n$ (здесь запятые между терминальными символами опущены). Наконец, правила 5, 6 и 7 позволяют исключить из этой цепочки нетерминальные символы α, β и γ , после чего получается терминальная цепочка вида $a^n b^n c^n$. Рассмотрим, например, вывод цепочки $aabbcc$. Промежуточные цепочки вывода будем разделять точкой с запятой, а нетерминальные символы и подцепочки терминальных символов внутри

цепочек вывода — запятыми: ϵ ; $\alpha\delta\delta$, $\beta\delta\delta$, $\gamma\delta\delta$; $\alpha\delta$, a , $\beta\delta\delta$, $\gamma\delta\delta$; a , aa , $\beta\delta\delta$, $\gamma\delta\delta$; aa , $\beta\delta\delta$, $\gamma\delta\delta$; aa , $\beta\delta$, b , $\gamma\delta\delta$; aa , β , bb , $\gamma\delta\delta$; $aabb$, $\gamma\delta\delta$; $aabb$, $\gamma\delta$, c ; $aabb$, γ , cc ; $aabbcc$.

Каков класс языков, порождаемых грамматиками ван Вейнгаардена? Ответы на этот вопрос дает следующая теорема [66].

Теорема 8. По любой классической грамматике произвольного вида может быть построена эквивалентная ей грамматика ван Вейнгаардена.

Доказательство. Пусть имеется произвольная грамматика $G = \langle V_T, V_A, I, S \rangle$. Построим по этой грамматике следующую грамматику ван Вейнгаардена. В множество нетерминальных символов метаграмматики включим три символа: $V_{\Delta M} = \{\beta, \gamma, \delta\}$. Образует множества двойников терминальных и нетерминальных символов исходной грамматики G , отмечая чертой сверху соответствующие символы. Будем также надчеркивать цепочки и множества, состоящие из надчеркнутых символов. Образует множество терминальных символов метаграмматики как объединение множеств двойников: $V_{TM} = \bar{V}_I \cup \bar{V}_A$. Правила вывода метаграмматики выберем таким образом, чтобы из β и γ можно было бы вывести любые цепочки множества V_{TM}^* , а из δ — любые цепочки множества \bar{V}_T :

- 1) $\beta \rightarrow \alpha\beta$ для любого $\alpha \in V_{TM}$,
- 2) $\gamma \rightarrow \alpha\gamma$ для любого $\alpha \in V_{TM}$,
- 3) $\delta \rightarrow \alpha\delta$ для любого $\alpha \in \bar{V}_T$,
- 4) $\beta \rightarrow \lambda$; 5) $\gamma \rightarrow \lambda$, 6) $\delta \rightarrow \lambda$.

Рассмотрим теперь объекты строгой грамматики. Терминальными символами этой грамматики должны быть, естественно, символы из V_T . Множество форм образуем из всех тех форм, которые будут встречаться в схемах правил вывода. В качестве начального символа выберем символ, изображаемый надчеркнутым начальным символом исходной грамматики G . Образует множество схем правил вывода. Схемы, которые мы включим в это множество, удобно разбить на две группы: $S_{St} = S_{St}^1 \cup S_{St}^2$. В S_{St}^1 включим схемы, образованные по правилам вывода исходной

грамматики следующим образом:

если $\xi \rightarrow \eta \in S$, то $\beta\bar{\xi}\gamma \rightarrow \beta\bar{\eta}\gamma \in S_{St}^1$.

В S_{St}^2 включим по одной схеме для каждого терминального символа:

$\delta\bar{a} \rightarrow \delta$, а при $a \in V_T$.

Покажем, что построенная нами грамматика ван Вейнгаардена эквивалентна исходной грамматике. Пусть имеется вывод терминальной цепочки в исходной грамматике G : $\omega_0 = I$, $\omega_1, \dots, \omega_n = \omega$. Любые две соседние цепочки вывода можно представить в виде $\omega_i = \chi_i \xi_i \zeta_i$, $\omega_{i+1} = \chi_i \eta_i \zeta_i$, $i = 0, 1, \dots, n-1$, где χ_i, ξ_i, η_i и $\zeta_i \in (V_T \cup V_A)^*$ и $\xi_i \rightarrow \eta_i \in S$. Поскольку в S_{St}^1 имеются схемы для всех примененных при выводе ω правил, т. е. $\beta\bar{\xi}_i\gamma \rightarrow \beta\bar{\eta}_i\gamma \in S_{St}^1$, $i = 0, 1, \dots, n-1$, а из β и γ в метаграмматике возможен вывод любых цепочек, составленных из двойников, в строгой грамматике имеет место выводимость $\bar{I} \Rightarrow \bar{\omega}$, где $\bar{\omega}$ рассматривается как изображение нетерминального символа строгой грамматики при $l(\omega) > 0$ или является пустой цепочкой. В последнем случае вывод в строгой грамматике закончен. Если же $\bar{\omega}$ — непустая, то используя подходящие схемы правил из S_{St}^2 , «отщепляем» от $\bar{\omega}$ справа по одному символу, заменяя его на соответствующий терминальный символ, пока не получим терминальную цепочку ω . Итак, в грамматике ван Вейнгаардена, которую мы построили по исходной грамматике G , возможен вывод любой терминальной цепочки, выводимой в G .

Докажем обратное утверждение. Пусть имеется вывод терминальной цепочки в строгой грамматике: $\omega_0 = \bar{I}$, $\omega_1, \dots, \omega_n = \omega$. Покажем, что ω выводима и в исходной грамматике G . Пусть ω_{i1} — первая цепочка, к которой применяется правило, полученное из схемы подмножества S_{St}^2 . Заметим, что $i_1 > 0$, так как к $\omega_0 = \bar{I}$ на первом шаге вывода может применяться лишь правило, полученное из схемы первой группы. ω_{i1} является изображением нетерминального символа строгой грамматики и, следовательно, состоит из надчеркнутых символов: $\omega_{i1} = \bar{\xi}_{i1}$. Учитывая спо-

соб образования схем подмножества $S_{S_1}^1$ по правилам вывода грамматики G , легко показать, что $I \xRightarrow[G]{*} \xi_{i_1}$.

Пусть цепочка ω_{j_1} ($j_1 \geq i_1 + 1$) выведена из ω_{i_1} применением правил, полученных из схем второй группы, и ω_{j_1} — либо последняя цепочка вывода, либо к ней применяется правило, полученное из схемы первой группы. Если $j_1 = n$, то $\xi_{i_1} = \omega$ и $I \xRightarrow[G]{*} \omega$. В противном

случае $\omega_{j_1} = \bar{\zeta}_{i_1} x_{i_1}$, где $\bar{\zeta}_{i_1}$ — изображение нетерминального символа строгой грамматики, x_{i_1} — терминальная цепочка. Считая, что ω_{i_2} — первая цепочка вывода после ω_{j_1} , к которой применяется правило, полученное из схемы второй группы ($i_2 \geq j_1 + 1$), мы можем представить эту цепочку в виде $\omega_{i_2} = \bar{\xi}_{i_2} x_{i_2}$ и, очевидно, $\bar{\zeta}_{i_1} \xRightarrow[G]{*} \bar{\xi}_{i_2}$. Но $\xi_{i_1} = \bar{\zeta}_{i_1} x_{i_1}$ и, следовательно,

$I \xRightarrow[G]{*} \bar{\xi}_{i_2} x_{i_2}$. Далее, выделяя подобным образом цепочки вывода ω_{j_k} и $\omega_{j_{k+1}}$, мы при некотором k получим $j_k = n$,

после чего окажется доказанной выводимость $I \xRightarrow[G]{*} \omega$.

Для завершения доказательства утверждения заметим, что если в выводе использовались лишь правила, полученные из схем первой группы, то $\omega = \lambda$ и выводимость ω в исходной грамматике очевидна.

Итак, мы показали, что построенная нами грамматика ван Вейнгаардена эквивалентна исходной грамматике. Следовательно, теорема доказана.

Очевидным следствием этой теоремы является утверждение, что класс языков, порождаемых грамматиками ван Вейнгаардена, совпадает с классом рекурсивно-перечислимых множеств цепочек символов, а порождающая мощность этих грамматик та же, что и классических грамматик произвольного вида.

Вернемся теперь к вопросу о формальном описании контекстных условий языков программирования. Большая порождающая мощность грамматик ван Вейнгаардена гарантирует возможность описания с помощью этих грамматик любых контекстных условий. Однако при описании синтаксиса языка АЛГОЛ-68 аппарат этих грамматик был использован

лишь для формализации условий третьего типа. Остальные контекстные условия по-прежнему описываются неформально¹⁾). Грамматика ван Вейнгаардена, задающая синтаксис языка АЛГОЛ-68, представляется в виде правил вывода метаграмматики и схем строгой грамматики, причем для тех и для других используются несколько измененные металингвистические формулы. Каждый нетерминальный символ метаграмматики является цепочкой больших букв латинского (или, в русском варианте языка, — русского) алфавита. Малые буквы латинского (или русского) алфавита являются терминальными символами метаграмматики. Таким образом, слова, составленные из больших букв, играют роль металингвистических переменных. В формулах используются металингвистические связки : и ;, имеющие тот же смысл, что и :: = и |. Поскольку составленные из больших букв металингвистические переменные метаграмматики хорошо выделяются в тексте, угловые скобки в данном варианте металингвистических формул отсутствуют. Схемы правил строгой грамматики записываются в следующем виде: форма, металингвистическая связка :, последовательность форм и терминальных символов строгой грамматики, разделенных запятыми, и, возможно, еще несколько подобных последовательностей, которые отделяются друг от друга металингвистической связкой ;. Слова, составленные из больших букв, в формах по-прежнему являются нетерминальными символами метаграмматики. Внутри схем допускаются пробелы, которые облегчают понимание смысла тех или иных форм и полученных из них нетерминальных символов строгой грамматики. Эти пробелы никак не влияют на вывод. Однако пробелы, отделяющие друг от друга слова, составленные из больших букв, конечно, должны учитываться, так как определяют, какие нетерминальные символы метаграмматики входят в формы. Заметим также, что

¹⁾ В опубликованном в 1974 году «Пересмотренном сообщении об алгоритмическом языке АЛГОЛ-68» с помощью грамматик ван Вейнгаардена формализуются все контекстные условия этого языка. См. «Revised Report on the Algorithmic Language ALGOL 68», Supplement to ALGOL BULLETIN № 36, March, 1974, Edmonton, Alberta.

терминальные символы строгой грамматики также представляются в виде цепочек малых букв, в которые, однако, должна входить подцепочка *symbol* (в русском варианте — символ). Приведем пример, иллюстрирующий способы представления правил метаграмматики и схем строгой грамматики.

Пример 2. В русский вариант грамматики ван Вейнгаардена, описывающей синтаксис языка Алгол-68, входят следующие правила метаграмматики:

СЛОВО: БУКВА; СЛОВО БУКВА; СЛОВО ЦИФРА

БУКВА: буква АЛФАВИТА; буква алеф

АЛФАВИТ: а; б; с; ...; б; в; г; ...; я¹⁾

ЦИФРА: цифра АРАБСКАЯ

АРАБСКАЯ: нуль; один; два; ...; девять

Эти правила позволяют получить конкретные правила вывода строгой грамматики из следующих схем, взятых нами также из описания языка АЛГОЛ-68:

идентификатор: СЛОВО²⁾

СЛОВО БУКВА: СЛОВО, БУКВА

СЛОВО ЦИФРА: СЛОВО, ЦИФРА

БУКВА: символ БУКВА

ЦИФРА: символ ЦИФРА

Используя приведенные схемы и правила, выведем из нетерминального символа строгой грамматики «идентификатор» цепочку, которая должна быть представлена в виде «символ буква а, символ буква б, символ цифра один». Начиная вывод, мы, очевидно, должны использовать некоторое правило, полученное из первой схемы строгой грамматики. В свою очередь, для получения конкретного правила требуется вывести в метаграмматике из нетерминального символа СЛОВО какую-нибудь терминальную цепочку. Одной из таких цепочек является «букваабуквабцифраодин», или, используя соглашение о не влияющих на вывод пробелах, «буква а буква б цифра один». Итак, мы имеем правило строгой грамматики, которое применим на первом шаге вывода:

идентификатор: буква а буква б цифра один.

Рассмотрим теперь третью схему и получим из нее правило вывода, подставляя вместо СЛОВО цепочку «буква а буква б», а вместо ЦИФРА — цепочку «цифра один»:

буква а буква б цифра один: буква а буква б, цифра один. Это правило применим на втором шаге вывода. Аналогично можно получить из схем и остальные применяемые при выводе правила:

буква а: символ буква а

буква б: символ буква б

цифра один: символ цифра один.

¹⁾ Здесь АЛФАВИТА и АЛФАВИТ представляют, как и в других случаях использования разных падежей, один и тот же нетерминальный символ метаграмматики.

²⁾ Это правило для наглядности упрощено; в полном варианте правила, который включен в описание языка, левая форма имеет вид «идентификатор ВИТКИ».

Итак, имеем следующий вывод интересующего нас идентификатора в строгой грамматике:

идентификатор; буква а буква б цифра один; буква а буква б, цифра один; буква а, буква б, цифра один; символ буква а, буква б, цифра один; символ буква а, символ буква б, цифра один; символ буква а, символ буква б, символ цифра один.

В заключение напомним еще раз используемые обозначения. Цепочки вывода отделяются друг от друга точками с запятой, внутри цепочек запятыми выделяются нетерминальные или терминальные символы строгой грамматики; если цепочка между запятыми содержит подцепочку «символ», то она изображает терминальный символ, в противном случае — нетерминальный символ; наконец, пробелы в цепочках вывода используются лишь для удобства чтения — цепочка с пробелами и без них обозначает одну и ту же последовательность символов.

Приведенный пример иллюстрирует лишь технику вывода в грамматике ван Вейнгаардена. Конечно, идентификатор — простая конструкция и правила ее вывода могут быть указаны даже с помощью обычной автоматной грамматики. Посмотрим теперь, как включаются в грамматику, описывающую синтаксис языка АЛГОЛ-68, контекстные условия третьего типа. Основная идея формализации этих условий заключается в следующем. Как уже упоминалось в § 1 настоящей главы, условия третьего типа в основном зависят от того, каковы виды значений, вырабатываемых конструкциями языка. Поэтому при выводе удовлетворяющих им программ виды вырабатываемых значений должны учитываться, для чего в изображении нетерминальных символов, соответствующих конструкциям, можно включить названия видов. Тогда окажется, что одной конструкции соответствует не один нетерминальный символ строгой грамматики, а много, может быть, даже бесконечно много таких символов, каждый из которых используется при необходимости вывести конструкцию, вырабатывающую значение того вида, который указан в изображении этого символа. Так, например, в грамматике не существует единого нетерминального символа, соответствующего конструкции «идентификатор» (см. сноску 2 на стр. 206). Вместо него имеется бесконечно много символов, из которых выводятся идентификаторы, именующие значения определенных видов: «идентификатор вида вещественный», «идентификатор вида мультимльтицелый» (для идентификатора,

именующего двумерный массив целых чисел), «идентификатор вида процедура с параметром вида целый и параметром вида целый вырабатывающая логический» (для идентификатора, именующего процедуру с двумя целыми параметрами, значение которой — логическое) и т. п. Изображения различных нетерминальных символов, соответствующих одной конструкции, можно получить из одной формы, если включить в нее такой нетерминальный символ метаграмматики, из которого возможен вывод названий нужных видов. Такие символы действительно имеются в метаграмматике грамматики ван Вейнгаардена, описывающей синтаксис языка АЛГОЛ-68. Например, ВИД — нетерминальный символ, из которого в метаграмматике возможен вывод названия любого вида: СОСТАВНОЙ — нетерминальный символ для видов структурных значений и мультизначений и т. д. Таким образом, форма «идентификатор вида ВИД» позволяет получить нетерминальный символ строгой грамматики, соответствующий идентификатору, именующему значение любого данного вида.

Этот способ представления конструкций различными нетерминальными символами, но едиными формами, и положен в основу метода формализации контекстных условий третьего типа. Существенное значение имеет при этом принцип согласованности при замене форм, входящих в одну схему правил, нетерминальными символами строгой грамматики. Он позволяет согласовывать виды значений, вырабатываемых конструкциями, входящими в одну общую конструкцию, а такое согласование и является, по существу, формальным представлением соответствующих контекстных условий. Приведем пример формального представления контекстного условия третьего типа.

Пример 3. В § 1 настоящей главы нами уже рассматривалась конструкция «присваивание» языка АЛГОЛ-68, которая является обобщением оператора присваивания языка АЛГОЛ-60. В результате выполнения этой конструкции имени некоторого вида присваивается значение этого вида. В противоположность оператору присваивания данная конструкция сама вырабатывает значение, которым является имя, участвующее в присваивании. Контекстное условие для нее также упоминалось выше. Оно указывает допустимые соотношения видов значений, вырабатыва-

емых левой частью присваивания, которая называется получателем, и его правой частью — источником. Назовем эти виды соответственно левым и правым. Тогда левый вид должен быть именем правого вида. В данное контекстное условие естественно также включить требование на вид значения, вырабатываемого самим присваиванием: поскольку значение присваивания — это значение получателя, данный вид должен совпадать с левым видом.

Синтаксическую структуру присваивания без учета контекстного условия можно описать с помощью следующей простой формулы:

присваивание: получатель, символ присвоить, источник.

Формула примерно такого вида используется при описании синтаксической структуры оператора присваивания языка АЛГОЛ-60. Нас она, однако, не устраивает — мы хотим указать в формуле сформулированное выше допустимое соотношение видов значений, вырабатываемых самим присваиванием, получателем и источником. Воспользуемся для этого нетерминальным символом метаграмматики ВИД, из которого, как уже указывалось, выводятся названия всех употребляемых в языке видов. Будем считать, что название, выведенное из этого символа при выводе конкретного присваивания, определяет правый вид. Тогда левый вид изображается этим названием, к которому слева приписано слово «имя». Так, например, если правый вид — вещественный, то левый вид — имя вещественного (напомним, что мы не обращаем внимание на надежные окончания слов, используемых для описания видов). Теперь ясно, какой должна быть формула, учитывающая соотношение видов:

присваивание вида имя ВИДА: получатель вида имя ВИДА,

символ присвоить, источник вида ВИД.

Так как из нетерминального символа ВИД в метаграмматике выводится бесконечное число терминальных цепочек, из схемы для присваивания можно получить бесконечно много правил вывода строгой грамматики. Принцип согласованности, которому нужно следовать при получении этих правил вывода, гарантирует выполнение контекстного условия. Приведем некоторые правила вывода метаграмматики, которые могут использоваться при выводе терминальных цепочек из символа ВИД.

ВИД: КОРЕНЬ; ОБЪЕДИНЕНИЕ

КОРЕНЬ: ТИП; СОСТАВНОЙ

ТИП: ПРОСТОЙ; форматный; ПРОЦЕДУРА; имя ВИДА

СОСТАВНОЙ: составленный из ПОЛЕЙ; мульти ВИД

ПРОСТОЙ: АРИФМЕТИЧЕСКИЙ; логический; литерный

АРИФМЕТИЧЕСКИЙ: ЦЕЛЫЙ; ВЕЩЕСТВЕННЫЙ

ЦЕЛЫЙ: КВАЗИДЛИННЫЙ целый

ВЕЩЕСТВЕННЫЙ: КВАЗИДЛИННЫЙ вещественный

КВАЗИДЛИННЫЙ: длинный КВАЗИДЛИННЫЙ; ПУСТО

ПУСТО:

ПОЛЯ: ПОЛЕ; ПОЛЯ и ПОЛЕ

ПОЛЕ: поле СЛОВО вида ВИД

ПРОЦЕДУРА: процедура КВАЗИПАРАМЕТРЫ ДЕЙСТ-

ВУЮЩАЯ

КВАЗИПАРАМЕТРЫ: с ПАРАМЕТАМИ; ПУСТО

ПАРАМЕТРЫ: ПАРАМЕТР; ПАРАМЕТРЫ и ПАРАМЕТР

ПАРАМЕТР: параметр вида **ВИД**
ДЕЙСТВУЮЩИЙ: вырабатывающий **ВИД**; не вырабатывающий значения

Эти правила вывода вместе с приведенными в примере 2 правилами для символа **СЛОВО** полностью описывают способы вывода названий так называемых необъединенных видов, наиболее часто используемых при программировании. Приведем примеры конкретных правил вывода строгой грамматики для присваивания, которые получены из схемы подстановкой вместо символа **ВИД** различных цепочек, выводимых из него с использованием перечисленных правил вывода метаграмматики.

Для присваивания вещественного числа:

присваивание вида имя вещественного: получатель вида имя вещественного, символ присвоить, источник вида вещественный.

Для присваивания матрицы целых чисел:

присваивание вида имя мультимультителичного: получатель вида имя мультимультителичного, символ присвоить, источник вида мультимультителичный.

Для присваивания процедуры без параметров, вырабатывающей логическое значение:

присваивание вида имя процедуры вырабатывающей логический: получатель вида имя процедуры вырабатывающей логический, символ присвоить, источник вида процедура вырабатывающая логический.

Поскольку класс языков, порождаемых грамматиками ван Вейнгаардена, совпадает с классом рекурсивно-перечислимых множеств цепочек символов, эти грамматики могут порождать языки с неразрешимой проблемой распознавания. Это, конечно, не относится к языку АЛГОЛ-68, который является распознаваемым языком. Тем не менее, такая излишне большая порождающая мощность грамматик затрудняет разработку формальных методов построения анализаторов даже для распознаваемых языков. Это обстоятельство является существенным недостатком данного метода формального описания синтаксиса.

Может возникнуть вопрос: нельзя ли выделить подмножество грамматик ван Вейнгаардена, порождающих лишь распознаваемые языки? Такое подмножество действительно выделяется. Можно показать, что в это подмножество входят грамматики, у которых схемы правил вывода строгих грамматик удовлетворяют ряду ограничений [48]. Первое из них связано с характеристикой, которую уместно назвать *обобщенной длиной* и которая определяется следующим образом. Обобщенной длиной каждого терминального символа строгой грамматики будем считать

некоторое зафиксированное целое положительное число k ; обобщенной длиной формы — число входящих в нее терминальных символов метаграмматики; обобщенной длиной левой части схемы правил вывода строгой грамматики — обобщенную длину представляющей ее формы, и, наконец, обобщенной длиной правой части — сумму обобщенных длин входящих в нее терминальных символов строгой грамматики и форм и числа используемых в ней запятых. Данные определения можно использовать и для вычисления обобщенных длин цепочек, состоящих из терминальных и нетерминальных символов строгой грамматики. Сформулируем интересующие нас ограничения на схемы правил вывода:

1) При некотором k обобщенная длина левой части каждой схемы не превосходит обобщенной длины ее правой части.

2) Все нетерминальные символы метаграмматики, входящие в левую часть некоторой схемы, входят также в ее правую часть.

3) Число вхождений каждого нетерминального символа метаграмматики в левую часть некоторой схемы не превосходит числа вхождений этого символа в ее правую часть.

Нетрудно видеть, что из схем, удовлетворяющих этим ограничениям, можно получить лишь такие правила вывода строгих грамматик, у которых обобщенные длины левых частей не больше обобщенных длин правых частей. Выводы в соответствующих строгих грамматиках являются неукорачивающими в том смысле, что обобщенные длины промежуточных цепочек каждого из них образуют неубывающую последовательность, и они ограничены сверху обобщенной длиной выведенной терминальной цепочки.

Класс языков, порождаемых грамматиками ван Вейнгаардена с перечисленными ограничениями на схемы, совпадает с классом НС-языков. Полное доказательство этого утверждения довольно громоздко, и его изложение заняло бы слишком много места. Идея доказательства, однако, очень проста. Для того чтобы показать, что язык, порождаемый неукорачивающей грамматикой ван Вейнгаардена, является НС-языком, достаточно построить по грамматике ЛО-

автомат, распознающий этот язык. Это следует из теоремы 4, § 1, гл. II. Символами рабочей ленты такого ЛО-автомата должны быть терминальные символы метаграмматики и строгой грамматики, а также запятая и не влияющий на процесс анализа нейтральный символ. Запятая используется для выделения терминальных и нетерминальных символов строгой грамматики (напомним, что последние представляются в виде цепочек терминальных символов метаграмматики). Количество ячеек рабочей ленты определяется обобщенной длиной распознаваемой цепочки при выбранном для данной грамматики числе k . В начальный момент на ленте располагается распознаваемая цепочка с запятыми между символами, а оставшиеся свободными ячейки заполняются нейтральными символами. Анализ исходной цепочки представляет собой недетерминированный процесс, на каждом шаге которого выделяется некоторая подцепочка, записанная на рабочей ленте. Если эта подцепочка является правой частью некоторого правила вывода строгой грамматики, то она заменяется левой частью правила. В противном случае возникает тупик. Если после некоторого шага на рабочей ленте окажется начальный символ строгой грамматики, автомат переходит в заключительное состояние.

Таким образом, цепочка допускается только в том случае, если найдется такой вариант анализа, при котором она преобразуется в начальный символ. Для того, чтобы определить, является ли подцепочка правой частью какого-либо правила, нужно перебрать правые части всех схем и для каждой проверить, не преобразуется ли она в данную подцепочку подстановкой с учетом согласованности вместо нетерминальных символов метаграмматики выводимых из них цепочек. Все эти действия легко выполняются ЛО-автоматом.

Элементарно доказывается обратное утверждение: любой НС-язык порождается некоторой неукорачивающей грамматикой ван Вейнгаардена. Оно непосредственно следует из доказательства теоремы 8: если исходная грамматика является неукорачивающей, то и эквивалентная ей грамматика ван Вейнгаардена также окажется неукорачивающей.

Грамматика, приведенная в примере 1, является неукорачивающей. Грамматика из примера 2 не является неукорачивающей из-за первой схемы строгой грамматики. Однако ее можно преобразовать в неукорачивающую. Схема из примера 3 представляет неукорачивающие правила. К сожалению, грамматика ван Вейнгаардена, описывающая синтаксис языка АЛГОЛ-68, не является неукорачивающей. Главная причина этого в том, что в нетерминальные символы строгой грамматики включаются описания видов значений, что, как уже упоминалось, положено в основу формализации контекстных условий третьего типа. Так, например, схема

идентификатор ВИТКИ: СЛОВО,
которая указывает, что идентификатор метки или любого вида может быть любым идентификатором, не удовлетворяет ограничениям, налагаемым на схемы неукорачивающих грамматик ван Вейнгаардена.

§ 4. Другие способы описания контекстных условий

Рассмотренные в предыдущих параграфах способы формализации контекстных условий страдают определенными недостатками. Так, хотя программные грамматики являются универсальным средством формализации, проверка контекстных условий, задаваемых этими грамматиками, довольно сложна. Грамматики ван Вейнгаардена обладают излишне большой порождающей мощностью, что затрудняет разработку анализаторов для языков, порождаемых этими грамматиками. Между тем действия, предусмотренные для проверки контекстных условий в трансляторах, довольно просты. Для первого и второго типов условий они обычно сводятся к формированию в процессе синтаксического анализа таблиц идентификаторов и других описываемых объектов по их определяющим вхождениям. Для включенных в таблицу идентификаторов указываются виды именуемых ими величин, которые определяются по описаниям. Таблицы имеют стековую организацию, т. е. делятся на ряд подтаблиц, каждая из которых соответствует некоторому блоку и включает в себя описанные в этом блоке объекты. По окончании анализа блока его под-

таблица более не используется. Проверка контекстных условий первого типа по таким таблицам не представляет трудностей. Она сводится к проверке, не входит ли уже в подтаблицу анализируемого блока идентификатор (или индикатор, метка), определяющее вхождение которого рассматривается в данный момент. Легко выполняется и идентификация. При этом нужно лишь учитывать, что определяющие вхождения некоторых объектов (например, меток) могут находиться в программах правее использующих вхождений.

Представляет интерес разработка таких способов формального описания контекстных условий, которые, во-первых, базировались бы на хорошо изученных КС-грамматиках и, во-вторых, позволяли бы при проверке контекстных условий использовать указанные выше действия с таблицами. Некоторые методы такого рода уже известны в настоящее время. Рассмотрим так называемые грамматики с глобальными и локальными контекстными условиями, которые представляют достаточно удобный аппарат для описания контекстных условий первого и второго типов [9].

Граматики с глобальными и локальными контекстными условиями (ГКУ-грамматики и ЛКУ-грамматики) представляются в виде КС-грамматик с дополнительными условиями, которым должен удовлетворять каждый правильный вывод. Положительным свойством таких грамматик является то, что для порождаемых ими языков можно использовать лишь слегка модифицированные синтаксические анализаторы КС-языков. Прежде чем перейти к определению ГКУ- и ЛКУ-грамматик, отметим некоторые свойства контекстных условий первых двух типов. В терминах КС-грамматик объекты, требующие описания, являются терминальными порождениями некоторых нетерминальных символов, а их позиции (определяющие или использующие вхождения) можно определить, рассматривая предки этих объектов в конкретных выводах. Поэтому информация об определяющих и использующих вхождениях может задаваться следующим образом. Из множества нетерминальных символов КС-грамматики выделим подмножество символов, чьи терминальные порождения могут нахо-

даться в двух позициях. Эти символы в дальнейшем будем называть *контекстными*. Для каждого вхождения контекстных символов в правые части правил вывода укажем, в которой из двух позиций будет находиться терминальное порождение данного символа при использовании правила в выводе. Формальное задание областей действия требует выделения из множества нетерминальных символов еще одной группы — символов, имеющих смысл блока языка АЛГОЛ-60. Такие символы мы будем называть *локализаторами*. Рассмотрим сначала грамматики, позволяющие задать контекстные условия первых двух типов при отсутствии локальных областей действия.

Определение. Грамматика с глобальными контекстными условиями — это шестерка следующих объектов:

$$G_{g1} = \langle V_T, V_N, V_C, I, S, M_0 \rangle,$$

где V_T — конечное множество терминальных символов; V_N — конечное множество *бесконтекстных* нетерминальных символов; V_C — конечное множество контекстных нетерминальных символов; I — начальный символ грамматики, S — конечное множество правил вывода вида $A \rightarrow \xi$, где A — бесконтекстный или контекстный нетерминальный символ, ξ — цепочка символов из $V_T \cup V_N \cup V_C$; контекстные символы в ξ могут находиться в двух позициях — в позиции определяющего и в позиции использующего вхождения; определяющие вхождения контекстных символов будем выделять надстрочным символом \wedge ; будем считать, что из контекстных символов не выводятся цепочки, содержащие контекстные символы, а также пустые цепочки; M_0 — *соответствие*, установленное между элементами множеств V_C и $V_T^* \setminus \{\lambda\}$, т. е. некоторое множество пар (A, x) , $A \in V_C$, $x \in V_T^* \setminus \{\lambda\}$, которое мы будем считать конечным, а, возможно, и пустым.

Пусть $W \subseteq V_C$. Через $M_0(W)$ будем обозначать подмножество множества $V_T^* \setminus \{\lambda\}$, состоящее из всех образов элементов из W . Соответствие M_0 будем считать *инъективным*, т. е. таким, что $M_0(\{A_i\}) \cap M_0(\{A_j\}) = \emptyset$, где $A_i, A_j \in V_C$ и $A_i \neq A_j$.

По первым пяти объектам ГКУ-грамматики G_{g1} может быть однозначно получена КС-грамматика G' ,

которую мы будем называть *сопряженной* к исходной грамматике G_{g1} . Для получения сопряженной грамматики достаточно объединить множества V_N и V_C в множество нетерминальных символов и удалить из правых частей правил вывода знаки, выделяющие определяющие вхождения контекстных символов. Будем считать, что терминальная цепочка выводима из некоторого нетерминального символа в ГКУ-грамматике в том и только том случае, если она выводима из этого символа в сопряженной грамматике. Аналогично определим для нашей грамматики понятия вывода, дерева вывода и другие.

Перейдем к определению языка $L(G_{g1})$, порождаемого ГКУ-грамматикой. Этот язык определяется как подмножество языка, порождаемого сопряженной грамматики: $L(G_{g1}) \subseteq L(G')$. Мы включим в $L(G_{g1})$ такие выводимые из начального символа терминальные цепочки, выводы которых удовлетворяют сформулированным ниже дополнительным условиям. Для того, чтобы упорядочить алгоритм проверки этих условий, обобщим для грамматики с глобальными контекстными условиями понятие левостороннего вывода. Будем называть вывод в ГКУ-грамматике *обобщенным левосторонним*, если он строится на основании следующих правил:

1) если в цепочке вывода ω имеются вхождения символов из V_N , правило вывода применяется к самому левому из этих вхождений;

2) если в ω нет вхождений символов из V_N , но имеются определяющие вхождения символов из V_C , правило вывода применяется к самому левому из них;

3) если в ω имеются лишь вхождения терминальных символов и использующие вхождения символов из V_C , правило вывода применяется к самому левому использующему вхождению.

Любой обобщенный левосторонний вывод в ГКУ-грамматике можно разбить на три этапа. Сначала применяются правила к бесконтекстным символам до тех пор, пока очередная цепочка не окажется состоящей лишь из терминальных и контекстных символов. Затем правила вывода применяются к определяющим вхождениям контекстных символов и к их

потомкам. В результате данные символы последовательно слева направо заменяются терминальными цепочками. Наконец, аналогично обрабатываются использующие вхождения контекстных символов, в результате чего они также заменяются терминальными цепочками.

Определим понятие *вывода, удовлетворяющего контекстным условиям*. При его выполнении, кроме подстановок вместо нетерминальных символов правых частей соответствующих правил, на некоторых шагах требуется проверять, входят ли порожденные из контекстных символов терминальные цепочки в подмножества $M(\{A_i\})$, где M — некоторое *переменное соответствие*, а A_i — контекстные символы, $i=1, 2, \dots, n$. Первоначально считаем, что $M=M_0$, а затем, в процессе вывода, будем включать в M пары, состоящие из контекстных символов и выведенных из их определяющих вхождений терминальных цепочек.

Итак, пусть имеется обобщенный левосторонний вывод $\omega_0, \omega_1, \dots, \omega_m$ терминальной цепочки из начального символа в некоторой ГКУ-грамматике G_{g1} . Положим $M=M_0$. Рассмотрим последовательно выполняемые на втором этапе вывода замены определяющих вхождений контекстных символов терминальными цепочками. Пусть $\omega_i = \xi \hat{A}_k \eta$, где \hat{A}_k — определяющее вхождение контекстного символа, к которому применяется правило вывода. Тогда в выводе найдется цепочка $\omega_j = \xi T(\hat{A}_k) \eta$, $i < j$, где $T(\hat{A}_k)$ — терминальная цепочка, выводимая в G_{g1} из A_k . Будем считать, что реализованная в выводе замена указанного вхождения \hat{A}_k на $T(\hat{A}_k)$ в цепочке ω_i удовлетворяет контекстным условиям, если $T(\hat{A}_k) \notin M(V_c)$. Включим в этом случае пару $(A_k, T(\hat{A}_k))$ в M . Теперь рассмотрим замены терминальными цепочками использующих вхождений контекстных символов, которые выполняются на третьем этапе вывода. Будем считать, что реализованная в выводе замена использующего вхождения контекстного символа A_k терминальной цепочкой $T(A_k)$ удовлетворяет контекстным условиям, если $T(A_k) \in M(\{A_k\})$. Наконец, будем считать, что обобщенный левосторонний вывод в ГКУ-грамматике, удовлетворяет контекстным условиям,

если все реализованные в нем замены определяющих и использующих вхождений контекстных символов терминальными цепочками удовлетворяют контекстным условиям. В язык $L(G_{g1})$, порождаемый ГКУ-грамматикой G_{g1} , включим те и только те терминальные цепочки, для которых существуют обобщенные левосторонние выводы из начального символа, удовлетворяющие контекстным условиям.

Все понятия, введенные нами при определении ГКУ-грамматик и порождаемых ими языков, хорошо интерпретируются на языках программирования. Так, например, контекстные символы соответствуют требующим описания объектам языка АЛГОЛ-60. Заметим только, что этими объектами нужно считать в данном случае не сами идентификаторы, а представляемые ими величины различных видов: простые переменные вещественного, целого и логического типов, массивы соответствующих типов, процедуры, метки и пр. Подмножества $M_0(\{A_i\})$ должны состоять из идентификаторов, закрепленных в конкретных представлениях языка за стандартными функциями, библиотечными процедурами, системными переменными. Подмножества $M(\{A_i\})$ соответствуют таблицам идентификаторов величин различных видов, образуемым при трансляции конкретных программ.

Легко показать, что класс языков, порождаемых ГКУ-грамматиками, шире класса КС-языков, но уже класса НС-языков. Рассмотрим примеры.

Пример 1. Рассмотрим уже упоминавшийся нами язык $L = \{x_1 b x_2 b \dots b x_p\}$, $x_i \in \{a_1, a_2, \dots, a_n\}^* \setminus \{\lambda\}$, $x_i \neq x_j$ при $i \neq j$, p — любое целое положительное число (см. пример 7, § 2, гл. I). Этот язык порождается чрезвычайно простой ГКУ-грамматикой:

$$G_{g1} = \langle V_T, V_N, V_C, I, S, M_0 \rangle;$$

$V_T = \{a_1, a_2, \dots, a_n, b\}$; $V_N = \{I, A\}$; $V_C = \{K\}$, I — начальный символ; $S = \{I \rightarrow \hat{K} b I, I \rightarrow \hat{K}, K \rightarrow A, A \rightarrow a_i A; A \rightarrow a_i\}$ (последние две схемы для $i = 1, 2, \dots, n$), M_0 — пустое.

Интересно заметить, что в правых частях правил вывода используются лишь определяющие вхождения контекстного символа K . Этого в данном случае оказывается вполне достаточно: в выводах, удовлетворяющих контекстным условиям, все порождаемые цепочки должны быть различными. Правило вывода $K \rightarrow A$ включено потому, что символ K не может входить в цепочку, выводимую из K .

Пример 2. Рассмотрим язык, состоящий из цепочек вида $x_1 b x_2 b \dots b x_p$, $p = 1, 2, \dots$, где x_1 — произвольная цепочка из

$\{a_1, a_2, \dots, a_n\}^* \setminus \{\lambda\}$, не равная $a_1 a_2$, а остальные цепочки x_i ($i=2, 3, \dots, p$) совпадают с x_1 или равны $a_1 a_2$. Такой язык порождается ГКУ-грамматикой со следующими множествами: $V_T = \{a_1, a_2, \dots, a_n, b\}$; $V_N = \{I, A, P\}$; $V_C = \{K\}$; $M_0 = \{(K, a_1 a_2)\}$; $S = \{I \rightarrow \hat{K} b P, I \rightarrow K, P \rightarrow K b P, P \rightarrow K, K \rightarrow A, A \rightarrow a_i A, A \rightarrow a_i\}; i=1, 2, \dots, n.$

Грамматики с глобальными контекстными условиями недостаточны для описания контекстных условий языков, имеющих локальные области действия. Как уже указывалось, в грамматиках, предназначенных для описания таких языков, нужно выделять еще одно подмножество нетерминальных символов — локализаторов. Рассмотрим класс грамматик, являющихся обобщением ГКУ-грамматик, выделив в них подмножество локализаторов.

Определение. Грамматикой с локальными контекстными условиями (ЛКУ-грамматикой) будем считать семерку следующих объектов:

$$G_{lc} = \langle V_T, V_N, V_C, V_L, I, S, M_0 \rangle,$$

где V_T, V_N, V_C, I, S и M_0 имеют тот же смысл, что и в ГКУ-грамматиках, V_L — конечное множество нетерминальных символов, которые называются локализаторами.

Определения сопряженной грамматики и выводимости аналогичны рассмотренным для ГКУ-грамматик. Обобщенный левосторонний вывод для ЛКУ-грамматик определяется с помощью трех пунктов, указанных при рассмотрении ГКУ-грамматик (цепочки, упоминаемые в третьем пункте, могут содержать локализаторы), и следующего пункта: если в цепочке имеются лишь терминальные символы и локализаторы, правило вывода применяется к самому левому вхождению локализатора.

Рассмотрим понятие вывода в ЛКУ-грамматике, удовлетворяющего контекстным условиям. Поскольку эта грамматика предназначена для описания языков, имеющих локальные области действия, для каждой такой области нужно определить свои переменные соответствия, аналогичные тем, которые рассматривались при описании вывода в ГКУ-грамматике. Каждое переменное соответствие должно задаваться в момент, когда начинается вывод подцепочки, определяющей локальную область действия, т. е. при применении

правила вывода к локализатору. При проверке контекстных условий это соответствие должно учитываться до окончания вывода подцепочки, т. е. до того момента, пока не будет применено правило вывода к другому локализатору, не являющемуся потомком данного. При выводе в ЛКУ-грамматике устанавливается также глобальная область действия, включающая всю выводимую из начального символа цепочку. Будем считать, что в выводе, удовлетворяющем контекстным условиям, вне локальных областей действия могут встречаться лишь использующие вхождения контекстных символов, причем их терминальные порождения должны входить в $M_0(V_c)$. Поэтому для глобальной области действия зададим постоянное соответствие M_0 .

Для более точного описания метода проверки контекстных условий введем следующие обозначения. Пусть L — такое вхождение локализатора в некоторую цепочку вывода, к которому применяется правило. Переменное соответствие, которое задается при применении этого правила, обозначим через M_L . Постоянное соответствие, которое задается для глобальной области действия, обозначим через M_I ; $M_I = M_0$. Наконец, через $\text{Апс}(\alpha)$, где α — некоторое вхождение символа в цепочку вывода, обозначим ближайший предок α , являющийся локализатором, а если такого предка нет, то вхождение начального символа в начальную цепочку вывода. Таким образом, $\text{Апс}(\alpha)$ определяет минимальную область действия, в которой находится вхождение α . При применении правила вывода к вхождению локализатора L установим $M_L = \emptyset$. Будем считать, что реализованная в выводе замена определяющего вхождения контекстного символа \hat{A} на $T(\hat{A})$ удовлетворяет контекстным условиям, если $\text{Апс}(\hat{A})$ не является вхождением I в начальную цепочку вывода и $T(\hat{A}) \notin M_L(\{A\})$, где $L = \text{Апс}(\hat{A})$. Включим в этом случае $(A, T(A))$ в M_L . Проверка контекстных условий для использующих вхождений несколько сложнее, так как нужно учитывать не только соответствие, заданное для минимальной области, в которой находится анализируемое вхождение, но и все соответствия, заданные для областей, включающих данную. Пусть в выводе реали-

зована замена вхождения контекстного символа A на $T(A)$. Тогда проверка контекстных условий может быть представлена в виде следующей процедуры:

1) Рассматриваем $\text{Apc}(A)$. Переходим к шагу 2.

2) Обозначим через L рассматриваемое вхождение символа. Если $T(A) \in M_L(\{A\})$, то реализованная в выводе замена удовлетворяет контекстному условию и процедура закончена. В противном случае переходим к шагу 3.

3) Если L есть вхождение I в начальную цепочку вывода, то замена не удовлетворяет контекстному условию, и процедура закончена. В противном случае рассматриваем $\text{Apc}(L)$ и переходим к шагу 2.

Такой анализ использующего вхождения реализуется правилом идентификации, согласно которому определяющее вхождение ищется сначала в наименьшем блоке, содержащем данное использующее вхождение, затем в объемлющем блоке и т. д. Если определяющее вхождение в программе не найдено, данное использующее вхождение сравнивается со стандартными величинами.

Аналогично случаю ГКУ-грамматик будем считать, что обобщенный левосторонний вывод в ЛКУ-грамматике удовлетворяет контекстным условиям, если все реализованные в нем замены определяющих и использующих вхождений контекстных символов удовлетворяют контекстным условиям. Язык, порождаемый ЛКУ-грамматикой, определим как множество терминальных цепочек, для каждой из которых существует обобщенный левосторонний вывод из начального символа грамматики, удовлетворяющий контекстным условиям.

ЛКУ-грамматика, описывающая синтаксис языка АЛГОЛ-60, легко получается по КС-грамматике, задаваемой нормальными формами Бекуса. Для этого нужно уточнить некоторые из используемых в формах понятий. Например, вместо понятия «простая переменная» следует ввести понятия «целая простая переменная», «вещественная простая переменная», «булевская простая переменная». Аналогично уточняются некоторые другие понятия, такие как, «идентификатор массива» и пр. Введение новых металингвистических переменных требует некоторой модификации

форм Бекуса. Затем нужно выделить понятия, соответствующие контекстным символам, отметить их определяющие вхождения в правые части форм и, наконец, выделить понятия, соответствующие локализаторам. Полученная таким образом ЛКУ-грамматика включает контекстные условия первых двух типов языка АЛГОЛ-60.

Приведем пример грамматики, описывающий простую модель языка программирования.

Пример 3. Рассмотрим ЛКУ-грамматику, состоящую из следующих объектов:

$$\begin{aligned} V_T &= \{a, b, =, :, \text{begin}, \text{end}, \text{real}, \text{integer}\}, \\ V_N &= \{\langle \text{программа} \rangle, \langle \text{операторы} \rangle, \langle \text{оператор} \rangle, \langle \text{идент} \rangle\}; \\ V_C &= \{\langle \text{вещпер} \rangle, \langle \text{целпер} \rangle\}; \\ V_L &= \{\langle \text{блок} \rangle\}; \\ I &= \langle \text{программа} \rangle; M_0 = \emptyset. \end{aligned}$$

S содержит следующие правила, представленные в виде нормальных форм Бекуса:

$$\begin{aligned} \langle \text{программа} \rangle &::= \langle \text{блок} \rangle \\ \langle \text{блок} \rangle &::= \text{begin} \langle \text{операторы} \rangle \text{end} \\ \langle \text{операторы} \rangle &::= \langle \text{операторы} \rangle; \langle \text{оператор} \rangle | \langle \text{оператор} \rangle \\ \langle \text{оператор} \rangle &::= \text{real} \langle \text{вещпер} \rangle | \text{integer} \langle \text{целпер} \rangle | \langle \text{вещпер} \rangle = \\ &= \langle \text{вещпер} \rangle | \langle \text{целпер} \rangle = \langle \text{целпер} \rangle | \langle \text{блок} \rangle \\ \langle \text{вещпер} \rangle &::= \langle \text{идент} \rangle \\ \langle \text{целпер} \rangle &::= \langle \text{идент} \rangle \\ \text{идент} &::= a \langle \text{идент} \rangle | b \langle \text{идент} \rangle | a | b \end{aligned}$$

«Программы» языка, порождаемого этой грамматикой, могут включать в себя блоки с неограниченной глубиной вложенности, в каждом из которых могут встречаться «описания» переменных и «операторы», использующие описанные переменные. Переменные обозначаются идентификаторами, образованными из букв a и b .

Одним из недостатков ЛКУ-грамматик является необходимость иметь отдельный контекстный символ для каждого из используемых в языке видов значений. От количества видов зависит и число образуемых при проверке контекстных условий подмножеств $M(\{A_i\})$. Такой способ представления видов контекстными символами не годится для языков с бесконечным множеством видов. Однако ЛКУ-грамматики нетрудно обобщить так, чтобы число контекстных символов и подмножеств не зависело от количества видов. Для этого нужно выделить еще одну группу нетерминальных символов — тех, из которых выводятся описатели ви-

дов. В языке АЛГОЛ-60 одним из таких символов является «тип», а остальные, такие, как «тип массива» — для описателей массивов, «тип процедуры» — для описателей процедур и пр., должны быть включены в число нетерминальных символов грамматики, что, естественно, приведет и к некоторой модификации правил вывода. Конкретные описатели, выводимые из выделенных символов, могут использоваться как своего рода метки подмножеств $M(\{A_i\})$. Терминальные цепочки, выведенные из определяющих вхождений контекстных символов, включаются в подмножества, помеченные теми описателями, которые определяют виды величин, именуемых этими цепочками. При таком способе число образуемых подмножеств i равно не количеству всех возможных видов, а числу видов, используемых в конкретной программе. При этом уменьшается и число контекстных символов грамматики. Например, для описания языка АЛГОЛ-60 достаточно в этом случае иметь лишь один контекстный символ — «идентификатор» (в предположении, что метки не могут быть целыми числами).

Контекстные условия третьего и четвертого типов не описываются ЛКУ-грамматиками. Для их формального задания нужно включить в эти грамматики какую-либо формальную систему указания допустимых соответствий видов в различных конструкциях языков. Возможно, наиболее целесообразно использовать для разработки такой системы идеи, положенные в основу грамматик ван Вейнгаардена.

Большинство методов построения синтаксических анализаторов легко преобразуется для случая ГКУ и ЛКУ-грамматик. При этом в соответствующем анализаторе должны быть лишь предусмотрены действия для проверки контекстных условий. Посмотрим, например, как преобразуется для ЛКУ-грамматик анализатор Кнута. В состав $LR(k)$ -анализатора нужно, помимо верхнего и нижнего магазинов, включить еще следующие виды памяти:

а) Состоящий из блоков магазинов для хранения определяющих и использующих вхождений. Блок должен иметь $2n$ таблиц, где n — число контекстных символов, т.е. по две таблицы для каждого контекстного символа. Одна из них предназначена для хранения

определяющих, а другая — использующих вхождения терминальных цепочек, выводимых из данного контекстного символа. Заметим, что каждый блок магазина соответствует некоторой области действия, а в его таблицы помещаются находящиеся в этой области определяющие и использующие вхождения. Блоки могут быть открытыми или закрытыми, причем открываются они слева направо, а закрываются справа налево.

б) Линейно упорядоченная последовательность регистров, каждый из которых предназначен для хранения терминальной цепочки неограниченной длины. В эти регистры будут заноситься выводимые из контекстных символов цепочки по мере их анализа $LR(k)$ -анализатором.

При работе анализатора нужно предусмотреть следующие действия, связанные с проверкой контекстных условий.

1) Пусть очередной такт анализатора определяется состоянием, в которое входит частичное состояние $[r, 0, \omega]$, причем в левой части r -го правила вывода находится локализатор. Тогда открывается новый блок магазина с пустыми таблицами.

2) Пусть на очередном такте производится свертка по правилу, в левой части которого находится локализатор. Тогда использующие вхождения, включенные в таблицы крайнего правого блока магазина, сравниваются с включенными в соответствующие таблицы всех блоков определяющими вхождениями. Если для некоторого использующего вхождения не нашлось совпадающее с ним определяющее, а рассматриваемый блок не является в магазине крайним левым, то данное вхождение включается в соответствующую таблицу использующих вхождений блока, находящегося в магазине левее данного. После окончания сравнений данный блок закрывается и содержавшаяся в нем информация стирается. В противном случае (т. е. если данный блок — единственный открытый блок магазина) контекстные условия считаются нарушенными.

Такой способ проверки контекстных условий связан с тем обстоятельством, что в программах определяющие вхождения могут быть расположены правее

использующих (например, идентификаторы процедур в программах на языке АЛГОЛ-60). Это, очевидно, допускается и определением ЛКУ-грамматик. Для сравнения использующих вхождений с заранее закрепленными обозначениями — цепочками, входящими в подмножество $M_0(V_c)$, — достаточно приписать к магазину слева еще один «постоянно открытый» блок, состоящий лишь из таблиц определяющих вхождений.

3) Пусть в состояние, определяемое очередным тактом, входит частичное состояние $[r, 0, \omega]$, причем в левой части r -го правила находится контекстный символ. Тогда в очередной незанятый регистр будут записываться все читаемые символы входной цепочки, пока не произойдет свертка по данному правилу. Иначе говоря, в регистры записываются терминальные цепочки, выведенные из контекстных символов. Они сохраняются в регистрах до тех пор, пока не будет ясно, в какие таблицы блока магазина их следует записать.

4) Пусть производится свертка по правилу, в правой части которого имеются вхождения контекстных символов. Очевидно, выведенные из таких символов цепочки будут в этот момент находиться в регистрах. Если цепочка, расположенная в регистре, соответствует использующему вхождению контекстного символа в правую часть правила, она записывается в таблицу использующих вхождений, выведенных из данного символа, в последнем открытом блоке. В случае определяющего вхождения цепочка сравнивается со всеми цепочками, записанными в таблицы определяющих вхождений последнего открытого блока магазина. Если совпадающая с ней цепочка не найдена, контекстное условие выполнено и цепочка записывается в таблицу определяющих вхождений. При обнаружении в таблицах совпадающей цепочки контекстное условие единственности описания нарушено.

Примерно таким же образом может быть модифицирован для случая ЛКУ-грамматик и глобальный анализатор.

Перечисленные в настоящей главе способы описания контекстных условий составляют лишь незначительную часть предлагавшихся. Представляет инте-

рес, например, метод, при котором порождающая КС-грамматика изменяется в процессе вывода [27]. Основная идея метода состоит в том, что при выводе определяющего вхождения к правилам грамматики приписывается еще одно правило, позволяющее заменить нетерминальный символ, находящийся в позиции использующего вхождения, выведенной цепочкой. Таким образом, для вывода использующих вхождений можно брать только те идентификаторы, которые уже были описаны. Для описания контекстных условий могут применяться и так называемые окрестностные грамматики [7], но их изучение выходит за рамки материала данной книги.

Все предлагавшиеся методы полного формального описания синтаксиса языков программирования не являются вполне удовлетворительными: они либо не универсальны, либо слишком сложны и не позволяют разработать удобные методы анализа, либо требуют дальнейшего развития и уточнения. Создание удобных и эффективных средств описания и анализа языков остается одной из важных задач теории языков программирования.

ЗАКЛЮЧЕНИЕ

Мы рассмотрели основные методы синтаксического описания и анализа формальных языков, к числу которых относятся и языки программирования. Эти методы еще не являются совершенными и продолжают развиваться и улучшаться. Каковы основные цели дальнейшего развития теории формальных языков и чем вызван тот большой интерес, который проявляют к ней специалисты в области математического обеспечения электронных вычислительных машин? Как наилучшим образом использовать предлагаемые в этой теории методы для разработки и использования языков программирования? В главах данной книги имеется много соображений и примеров, которые отвечают на эти вопросы. Мы, однако, мало касались одного из важнейших разделов математического обеспечения — методов разработки трансляторов с машинных языков программирования на машинные. Между тем развитие и совершенствование именно

этих методов тесным образом связано с общим состоянием теории формальных языков. Какова структура и способы составления современных трансляторов и каковы перспективы их совершенствования — вот те вопросы, которые мы кратко обсудим ниже.

В настоящее время имеется большое число работающих трансляторов с самых различных языков программирования и для самых различных вычислительных машин. Накоплен большой практический опыт их разработки, имеется значительное количество приемов и методов программирования отдельных блоков трансляторов. Тем не менее, сложившееся в этой области положение не может удовлетворить специалистов. Разработка новых трансляторов все еще остается исключительно трудоемким и сложным делом, на которое уходят месяцы, а то и годы работы больших коллективов. Подавляющее большинство практически используемых трансляторов разработано целиком вручную, причем блоки или отдельные программы трансляторов, составленных ранее, трудно использовать без изменений для новых трансляторов.

Наиболее естественным средством сокращения времени и труда программистов, занятых разработкой трансляторов, является частичная или даже полная автоматизация программирования алгоритмов трансляции. В идеальном случае электронная вычислительная машина, получая в качестве исходных данных информацию о синтаксисе и семантике языка программирования, могла бы в результате ее обработки выдавать готовые к использованию программы транслятора с этого языка на некоторый другой (напримёр, машинный) язык. Программы такого автоматического создания трансляторов можно назвать метатрансляторами. Другой подход к проблеме автоматизации заключается в том, что разрабатываются такие трансляторы, которые были бы пригодны не для фиксированной пары входного и выходного языков, а для многих таких пар. Трансляторы этого типа часто называются синтаксически ориентированными. Работы в направлении создания метатрансляторов и синтаксически ориентированных трансляторов интенсивно ведутся во многих странах. До сих пор, однако, не имеется ни одного транслятора,

разработанного с помощью этих средств, который был бы сравним по своей эффективности с трансляторами, составленными вручную. Мы уже знакомы с трудностями, которые возникают при формальном описании синтаксиса языков программирования и автоматической разработке их синтаксических анализаторов. Далеко не решен еще вопрос и их семантического описания. Для того, чтобы лучше понять возникающие при этом проблемы, рассмотрим структуру типичного транслятора компилирующего типа, т. е. такого, который переводит программу, написанную на входном языке, на машинный язык, не выполняя описанный этой программой алгоритм до окончания перевода.

Работа такого транслятора представляет собой ряд последовательных просмотров программы, на каждом из которых текст программы читается слева направо (прямой просмотр) или справа налево (обратный просмотр). В то время каждого просмотра текст программы изменяется определенным образом, а также вырабатывается дополнительная информация, необходимая на дальнейших этапах работы: таблицы используемых в программе чисел и других констант, таблица идентификаторов с указанием видов именованных ими значений и блоков, в которых они описаны, и пр. Таким образом, каждый просмотр может интерпретироваться как некое преобразование программы из одной системы записи в другую. Функции отдельных просмотров в значительной степени зависят от особенностей входных языков, общей структуры трансляторов и других индивидуальных характеристик. Обычно транслятор делится на два больших блока: блок синтаксического анализа входной программы и блок синтеза выходной программы. Просмотры, относящиеся к первому блоку, выполняют следующие функции:

- 1) *Предварительный анализ программы* с целью представления ее в более удобном для дальнейшей работы виде. При анализе выделяются идентификаторы, числа и, возможно, некоторые другие простые объекты программы. Образуются таблицы выделенных объектов. Вхождения составных основных символов, таких, как **begin**, **end**, знак присваивания и др., заменяются более компактными кодами. В результа-

те анализа программа представляется в виде последовательности кодов стандартной длины, каждый из которых обозначает основной символ или выделенный объект.

2) *Синтаксический анализ программы без учета контекстных условий* (такой анализ иногда называют *видонезависимым*). Цель работы данного блока состоит в получении информации о синтаксической структуре программы, которая обычно является представленным в том или ином виде деревом вывода этой программы в порождающей грамматике.

3) *Проверка контекстных условий и идентификация*. На этом этапе вырабатываются таблицы идентификаторов и других описываемых объектов с указанием видов соответствующих им значений. В таблицах указываются также блоки, в которых описаны идентификаторы. Совпадающим идентификаторам, описанным в разных блоках, присваиваются различные коды.

4) *Представление программы на промежуточном языке*. Промежуточным языком часто называют систему обозначений, в которой представляется входная программа после обработки ее блоком синтаксического анализа. Программа на промежуточном языке обычно состоит из последовательности инструкций, которая вместе с таблицей идентификаторов и другой служебной информацией определяет работу синтезирующего блока транслятора. Инструкции промежуточного языка подбираются таким образом, чтобы в большинстве случаев был возможен локальный, т. е. не зависящий от контекста, их перевод на выходной язык. Иногда программа на промежуточном языке подвергается дополнительной обработке с целью улучшить качество выходной программы. Такая обработка называется *оптимизацией*.

Четвертый этап является последней частью блока синтаксического анализа. Какие трудности возникают при разработке алгоритмов автоматического построения программ синтаксического анализа указанной структуры? Это прежде всего уже известные нам проблемы полного формального описания синтаксиса языков с учетом контекстных условий и построения на основе такого описания их синтаксических анализа-

торов. Необходимо также иметь хорошие методы задания структуры таблиц и другой дополнительной информации, получаемой при анализе параллельно с нахождением деревьев выводов. Наконец, требуют разработки алгоритмы представления программ на промежуточных языках трансляторов по их деревьям выводов в порождающих грамматиках. Все эти вопросы в настоящее время изучены недостаточно, и от дальнейшего развития и совершенствования соответствующих методов во многом зависит практическая реализация автоматического составления трансляторов.

Еще в большей степени нуждаются в изучении и методы автоматической разработки синтезирующих блоков трансляторов. Одна из трудностей заключается в формальном описании семантики входных языков средствами языков, на которые производится перевод. Программы на выходном языке, представляющие те или иные конструкции входного языка, зависят не только от реализуемых операций, но и от видов значений, используемых в качестве операндов. Так, например, сложение целых и вещественных чисел, а также чисел, представленных в памяти с различной степенью точности, реализуется с помощью различных последовательностей команд. Одним из наиболее перспективных подходов к этой проблеме представляется разработка специализированных для целей трансляции макрогенераторов, которые позволяют описывать макрокоманды, генерирующие различные выходные программы в зависимости от своих параметров. Параметры на промежуточном языке в этом случае должны представляться в виде последовательностей макрокоманд. При наличии достаточно обширной библиотеки модификация системы макрокоманд при разработке нового транслятора может свестись к минимуму.

В целом развитие и совершенствование методов описания и анализа языков программирования представляет одну из наиболее интересных, сложных и актуальных задач современной прикладной математики.

ЛИТЕРАТУРА

В приводимой ниже библиографии указаны основные источники, цитируемые в тексте, а также некоторые работы по смежным вопросам.

Система обозначений и терминология, используемые в главах I и II, заимствованы, в основном, из [13, 15, 17]. Там же имеются доказательства некоторых утверждений, приводимые в этих главах. Из других публикаций, посвященных формальным грамматикам, нужно, в первую очередь, отметить широко известные работы Н. Хомского: [37, 38, 39, 40, 41, 42, 49], а также монографии [3, 16, 21] и носящую скорее научно-популярный характер [18]. Приведены также работы, посвященные как отдельным свойствам классических формальных грамматик, так и некоторым другим средствам описания языков [7, 19, 22, 25, 26, 30, 43, 45]. Среди публикаций, связанных с проблемой синтаксического анализа формальных языков, следует упомянуть монографии [13, 16, 21], а также сборник переводов [35] и работы [49, 50, 52, 53, 54, 56, 57, 60, 61, 64, 65, 66, 67, 69].

В списке приводится ряд публикаций, посвященных изучению формальных свойств языков программирования, в некоторых из них рассматривается проблема контекстных условий. Это работы [8, 9, 10, 14, 27, 34, 47, 48, 51, 55, 68]. Указаны также некоторые публикации, посвященные автоматизации разработки трансляторов: [4, 6, 12, 20, 23, 31, 32, 33, 36, 44, 58, 59, 62, 63].

[1] и [2] содержат формальные описания языков АЛГОЛ-60 и АЛГОЛ-68, в [11] и [28] приводится неформальное описание последнего и, наконец, монографии [5, 24, 29, 46] могут использоваться как руководства по вопросам теории графов, математической логики, теории рекурсивных функций и теории отношений.

1. Алгоритмический язык АЛГОЛ-60, пересмотренное сообщение, пер. с англ., «Мир», 1965.
2. Алгоритмический язык АЛГОЛ-68, сб. «Кибернетика», I, № 6, 1969, 17—144, II, № 1, 1970, 13—160.
3. Алфорова З. В., Теория алгоритмов, «Статистика», 1973.
4. Балугев А. Н., Братчиков И. Л., Некоторые особенности промежуточного языка транслятора с АЛГОЛа-68, «Теория языков и методы построения систем программирования», Труды симпозиума, Киев — Алушта, 1972, 391—396.
5. Бер ж К., Теория графов и ее применения, пер. с франц., ИЛ, 1962.
6. Болье Л., Методы построения компиляторов, пер. с англ., «Языки программирования», «Мир», 1972, 87—277.

7. Борщев В. Б., Хомяков М. В., Окрестностные грамматики и модели перевода. Часть I. Окрестностные грамматики, НТИ, серия 2, 1970, № 3, 39—44.
8. Братчиков И. Л., Об одном способе формального описания и анализа контекстных условий языков типа АЛГОЛ-60, «Теория языков и методы построения систем программирования», Труды симпозиума, Киев — Алушта, 1972, 381—390.
9. Братчиков И. Л., Тойрих Х. И., О формализации некоторых контекстных условий языков программирования», «Журнал вычислит. матем. и матем. физики», т. 14, 1974, № 4, 1004—1015.
10. Васильев В. А., О строении множества видов величин в алгоритмических языках, «Журнал вычислит. матем. и матем. физики», т. 10, 1970, № 5, 1247—1268.
11. Васильев В. А., Язык АЛГОЛ-68, основные понятия, «Наука», 1972.
12. Вельбицкий И. В., Метаязык К-грамматик, сб. «Кибернетика», 1973, № 3, 47—64.
13. Гинзбург С., Математическая теория контекстно-свободных языков, пер. с англ., «Мир», 1970.
14. Гинзбург С., Райс Х., Два класса языков типа АЛГОЛ, пер. с англ., «Кибернетический сборник», новая серия, вып. 6, «Мир», 1969.
15. Гладкий А. В., Лекции по математической лингвистике для студентов НГУ, Изд-во Новосибирского университета, 1966.
16. Гладкий А. В., Формальные грамматики и языки, «Наука», 1973.
17. Гладкий А. В., Диковский А. Я., Теория формальных грамматик, Труды 2-й Всесоюзной конференции по программированию, приглашенные доклады (1-й вып.), Новосибирск, 1970, 43—70.
18. Гладкий А. В., Мельчук И. А., Элементы математической лингвистики, «Наука», 1969.
19. Гладкий А. В., Мельчук И. А., Грамматики деревьев, Информационные вопросы семиотики, лингвистики и автоматического перевода, вып. 1, 1971, 16—41.
20. Глушков В. М., Вельбицкий И. В., Стогний А. А., Об одном подходе к построению системы математического обеспечения современных вычислительных машин, сб. «Кибернетика», 1972, № 3, 25—35.
21. Гросс М., Лантен А., Теория формальных грамматик, пер. с франц., «Мир», 1971.
22. Диковский А. Я., О соотношении между классом всех контекстно-свободных языков и классом детерминированных контекстно-свободных языков, «Алгебра и логика», т. 8, 1969, вып. 1, 44—64.
23. Ингерман П., Синтаксически ориентированный транслятор, пер. с англ., «Мир», 1969.
24. Клини С. К., Введение в математику, пер. с англ., ИЛ, 1957.
25. Летищевский А. А., Синтаксис и семантика формальных языков, сб. «Кибернетика», 1968, № 4, 1—9.
26. Летищевский А. А., Об отношениях, представимых в push-down автоматах, сб. «Кибернетика», 1969, № 1, 1—7.

27. Лети́чевский А. А., Об одном обобщении понятия контекстно-свободной грамматики, сб. «Кибернетика», 1972, № 3, 1—3.
28. Линдси Ч., ван дер Мюйлен С., Неформальное введение в АЛГОЛ-68, пер. с англ., «Мир», 1973.
29. Мальцев А. И., Алгоритмы и рекурсивные функции, «Наука», 1965.
30. Мейтус В. Ю., Обобщенные преобразующие грамматики, сб. «Кибернетика», 1972, № 6, 17—27.
31. Параметрические транслирующие системы, «Штиинца», Кишинев, 1974.
32. Разработка трансляторов, Изд-во Ростовского университета, 1972.
33. Редько В. Н., Ющенко Е. Л., Алгоритмические языки и транслирующие системы, сб. «Кибернетика», 1967, № 5, 87—91.
34. Розенкранц Д., Программные грамматики и классы формальных языков, пер. с англ., Сборник переводов по вопросам информационной теории и практики, ВИНТИ, 1970, № 16, 117—146.
35. Синтаксический анализ формальных языков, сб. переводов, Изд-во Ростовского университета, 1971.
36. Фельдман Дж., Грайс Д., Системы построения трансляторов, пер. с англ., «Алгоритмы и алгоритмические языки», вып. 5, 1971, 105—214.
37. Хомский Н., Три модели для описания языка, пер. с англ., «Кибернетический сборник», вып. 2, ИЛ, 1962, 237—266.
38. Хомский Н., Синтаксические структуры, пер. с англ., «Новое в лингвистике», вып. 2, ИЛ, 1962, 412—527.
39. Хомский Н., О некоторых формальных свойствах грамматик, пер. с англ., «Кибернетический сборник», вып. 5, ИЛ, 1962, 279—311.
40. Хомский Н., О понятии «правило грамматики», пер. с англ., «Новое в лингвистике», вып. 4, «Прогресс», 1965, 34—65.
41. Хомский Н., Формальные свойства грамматик, пер. с англ., «Кибернетический сборник», новая серия, вып. 2, «Мир», 1966, 121—230.
42. Хомский Н., Миллер Дж., Языки с конечным числом состояний, пер. с англ., «Кибернетический сборник», вып. 4, ИЛ, 1962, 233—255.
43. Хомский Н., Шютценберже М. П., Алгебраическая теория контекстно свободных языков, пер. с англ., «Кибернетический сборник», новая серия, вып. 3, «Мир», 1966, 195—242.
44. Хопгуд Ф., Методы компиляции, пер. с англ., «Мир», 1972.
45. Чулик К., Хорошо переводимые грамматики и языки типа АЛГОЛ, пер. с англ., НТИ, сер. 2, № 3, 1967, 21—23.
46. Шрейдер Ю. А., Равенство, сходство, порядок, «Наука», 1971.
47. Backus J. W., The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference, IFIP Proceedings, Paris, 1959.
48. Baker J. L., Grammars with structured vocabulary: a model for the ALGOL 68 definition, Information and Control 20, 1972, № 4, 351—395.
49. Chomsky N., Context-free grammars and pushdown storage, M. I. T., Res. Lab. Electron., Quart. Progr. Rept., 1962, № 65.

50. Earley J. C., An efficient context-free parsing algorithm, *Comm. ACM* 13, 1970, № 2, 94—102.
51. Floyd R. W., On the nonexistence of a phrase-structure grammar for ALGOL-60. *Comm. ACM* 5, 1962, 483—484.
52. Floyd R. W., Syntactic analysis and operator precedence, *Journ. ACM* 10, 1963, 316—333.
53. Floyd R. W., Bounded context syntactic analysis, *Comm. ACM* 7, 1964, 62—67.
54. Floyd R. W., The syntax of programming languages, a survey, *IEEE Trans EC* 13, 1964, 4, 346—353.
55. Gilbert P., On the syntax of algorithmic languages, *Journ. ACM* 13, 1966, 90—107.
56. Ginzburg S., Greibach S. A., Deterministic context-free languages, *Information and Control*, 9, 1966, № 6, 620—648.
57. Griffiths T. V., Petrick S. R., On the relative efficiencies of context-free grammar recognizers, *Comm. ACM* 8, 1965, № 3, 289—299.
58. Halpern M., Toward a general processor for programming languages, *Comm. ACM* 11, 1968, 15—26.
59. Irons E. T., The structure and use of the syntax-directed compiler. Annual review in Automatic Programming, vol. 3, 1963, 207—227.
60. Knuth D. E., On the translation of languages from left to right, *Information and Control* 8, 1965, 607—639.
61. Korenjak A. J., A practical method for constructing $LR(k)$ -processor, *Comm. ACM* 12, 1969, № 11, 613—623.
62. Köster C. H. A., Affix grammars in ALGOL-68 implementation, North-Holland Pub. Co., Amsterdam, 1971.
63. Köster C. H. A., A compiler compiler, Report MR 127, Mathematisch Centrum, Amsterdam, 1971.
64. Kral J., Demner J., Semi-top-down syntactic analysis, Technical University of Prague, 1973.
65. Lerner A., Lim A. L., A note on transforming context-free grammars to Wirth-Weber precedence form, *The Computer Journal* 13, 1970, № 2, 142—144.
66. Sintzoff M., Existence of a van Wijngaarden Syntax for every recursively enumerable set, *Ann. Soc. Sci.*, 1967, 81, Bruxelles, 115—118.
67. Unger S. H., A global parser for context free phrase structure grammars, *Comm. ACM* 11, 1968, № 4, 240—247.
68. van Wijngaarden A., Recursive definition of syntax and semantics, Formal Language description languages for computer programming, North-Holland Pub. Co., Amsterdam, 1966, 13—24.
69. Wirth N., Weber H., EULER — a generalization of ALGOL, and its formal definition, Part I, Part II, *Comm. ACM* 9, 1966, № 1, 13—25, № 2, 89—99.

Цена 74 коп.

307
Б874