Серия «Мастер»

Классика программирования:

АЛГОРИТМЫ ЯЗЫКИ АВТОМАТЫ КОМПИЛЯТОРЫ

Практический подход

М.В. Мозговой

НиТ

Мозговой М.В.
Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход. — СПб.: Наука и Техника, 2006. — 320 с.: ил.

Под редакцией М.В. Финкова

ISBN 5-94387-224-8

Серия «Секреты мастерства»

Книга, которую вы держите в руках, посвящена описанию фундаментальных основ компьютерной науки и их применению на практике. Рассмотрено большое количество алгоритмов и моделей, которые можно использовать в повседневном программировании. При этом показано, как их использовать.

Практически все книги подобной направленности имеют ярко выраженную теоретическую ориентацию. В них много формул, теорем и доказательств, но крайне мало листингов программ. Особенность же этой книги заключается в том, что автор изложил материал максимально доступным языком (насколько это возможно в рамках темы), по возможности делая акцент на реализуемые алгоритмы и модели, а не на формулы и теоремы. Приведены конкретные примеры.

Эта книга, с одной стороны, позволяет расширить кругозор и углубить понимание основных принципов и проблем компьютерной науки, а с другой стороны — пополнить собственный инструментарий, предназначенный для ежедневного применения. Книга предназначена всем, кто интересуется и занимается программированием.

ISBN 5-94387-224-8

Контактные телефоны издательства
(812) 567-70-25, 567-70-26
(044) 516-38-66

Официальный сайт www.nit.com.ru

ООО «Наука и Техника».
Лицензия №000350 от 23 декабря 1999 года.
198097, г. Санкт-Петербург, ул. Маршала Говорова, д. 29.
Подписано в печать 18.08.2005. Формат 70x100/16.
Бумага газетная. Печать офсетная. Объем 20 п. л.
Тираж 3000 экз. Заказ № 304

Отпечатано с готовых диапозитивов в ОАО «Техническая книга»
190005, Санкт-Петербург, Измайловский пр., 29

Содержание

Введение

О чем эта книга?

Компьютерная наука — область весьма прагматичная. В далеко не чуждой ей математике до сих пор существует (хотя и весьма условное) разделение на «чистую» и «прикладную» ветви. Насколько мне известно, многие математики и поныне занимаются лишь абстрактными моделями, не имеющими видимого применения в современном мире. Некоторые из таких, казалось бы, абсолютно бесполезных конструкций, изученных в прошлом, пришлись весьма кстати сейчас; другие все еще ждут своего момента, третьи, вероятно, не окажутся востребованными никогда, оставаясь не более чем затейливыми интеллектуальными игрушками.

Компьютерная наука не витает в облаках, а прочно стоит на земле. Здесь всегда действует марксистский принцип «практика — критерий истины». Любое, даже самое, на первый взгляд, абстрактное понятие призвано содействовать решению практических задач. Иногда, впрочем, компьютерные модели могут увести далеко от реальности; бывают и ситуации, когда изначально искусственные конструкции вызывают живые аналогии с процессами, происходящими в природе. Тогда компьютерная наука, оторвавшись от привычного жонглирования числами, начинает претендовать на роль мощного инструмента, с помощью которого можно попытаться познать мир и даже самих себя. Так или иначе, любая модель, любая умозрительная конструкция в конечном итоге подлежит реализации на компьютере и проверке практикой. Тогда становится ясно, какие идеи содержат в себе здравое зерно, а какие являются всего лишь неадекватной решаемой задаче игрой воображения.

Книга, которую вы держите в руках, посвящена описанию теоретических основ компьютерной науки с упором на практические примеры. В двадцатых-тридцатых годах XX века, когда компьютеров еще не было, ученые уже размышляли над такими нетривиальными вопросами:

- Что такое алгоритм?
- Почему одну задачу решить просто, другую сложно, а третью вообще никак не удается?
- Как создать машину, решающую задачи?

Достижения в анализе, в частности, этих тем привели к возникновению современной информатики и ее раздела, называемого *теорией вычислений*. Попутно было разработано большое число полезнейших моделей и алгоритмов, которые можно с успехом использовать в повседневном программировании. К сожалению, книги по теории вычислений на русском языке практически не издаются. Единственной полностью релевантной книгой на сегодняшний день я могу назвать, пожалуй, лишь классическую работу Дж. Хопкрофта, Р. Мотвани и Дж. Ульмана «Введение в теорию автоматов, языков и вычислений» (второе издание). Есть еще, конечно, различные методические пособия, выпускаемые университетами для своих студентов и старые книги советских еще времен, но будем реалистами. Если книга не продается в Доме Книги на Новом Арбате или на Невском проспекте, ее можно считать лишь условно доступной читателю (в других городах, к сожалению, ситуация навряд ли будет лучше).

Кроме того, практически все книги по теории вычислений имеют ярко выраженную теоретическую ориентацию. В них много формул, теорем и доказательств, но крайне мало листингов программ. Безусловно, я не считаю математизированность недостатком книги, но существующий явный перекос в сторону теории мне не совсем понятен. Изучение алгоритмов и моделей теории вычислений, на мой взгляд, будет весьма полезно людям, занимающимся в большей степени практическим программированием. С одной стороны, это отличная возможность расширить кругозор и углубить понимание основных принципов и проблем компьютерной науки; с другой стороны, это незаменимый способ пополнить собственный инструментарий, предназначенный для ежедневного применения.

Резюмируя, я могу сформулировать основную задачу этой книги так: познакомить практиков с конструкциями, обычно относимыми к теоретическим аспектам компьютерной науки, делая по возможности акцент на реализуемые алгоритмы, а не на формулы и теоремы.

Искренне надеюсь, что книга, на которую я потратил столько времени и сил, вам понравится. Перед тем, как перейти к следующему разделу, хочу поблагодарить за ценные замечания Сергея Караковского, который нашел время, чтобы прочесть рукопись книги, а также издательство «Наука и техника» в лице Марка Финкова за одобрение моих идей и постоянную поддержку.

О структуре книги

Вероятно, лишь перевернув последнюю страницу, вы сможете связать в единую цепь на первый взгляд почти независимые сведения из разных глав. Однако уже сейчас имеет смысл дать взгляд на книгу «с высоты птичьего полета», чтобы последовательность изложения не казалась совсем уж загадочной. Тем читателям, которые захотят сделать чтение более интригующим, я могу посоветовать пропустить этот раздел, простодушно

раскрывающий все тайны и лишающий удовольствия самостоятельно обнаружить неожиданное родство между самыми непохожими друг на друга концепциями.

Итак, в первой главе происходит знакомство с понятием *формального языка* и *задачами описания и распознавания формальных языков* на примере простейших *регулярных языков*. Вы увидите, что задача распознавания очень часто встречается на практике; глава содержит несколько полезных примеров использования регулярных языков (задаваемых с помощью *регулярных выражений*) в повседневном программировании. При этом будет показано, что в жизни возникают ситуации, когда возможностей регулярных языков уже недостаточно для удовлетворения наших потребностей.

Во второй главе описывается понятие *конечного автомата*, с помощью которого задачу распознавания регулярного языка удастся решить практически (в первой главе вся работа перекладывается на некую «волшебную функцию»). Конечный автомат можно воспринимать как самостоятельное устройство (простейшую разновидность компьютера) или как алгоритм, реализуемый на любом языке программирования.

В третьей главе между регулярными выражениями и конечными автоматами устанавливается прочная взаимосвязь. Показывается, как любому данному регулярному выражению можно сопоставить описывающий тот же самый язык конечный автомат и наоборот.

Четвертая глава полностью посвящена так называемому *автоматному программированию*. Суть этой концепции в том, чтобы писать алгоритмы, по своему поведению напоминающие устройства, подобные конечным автоматам. Надеюсь, мне удастся доказать, что автоматный подход в некоторых случаях является удобным и естественным для решения возникающих на практике задач.

В пятой главе, наконец, мы выйдем за рамки регулярных языков и рассмотрим более широкий класс *контекстно-свободных языков*. В первую очередь пятая глава посвящена *контекстно-свободным грамматикам* — средству, позволяющему описать любой контекстно-свободный язык, но не дающему практического алгоритма решения задачи распознавания. Будет показано, что регулярные языки представляют собой подмножество языков контекстно-свободных. Вы познакомитесь с интересным частным случаем контекстно-свободной грамматики — регулярной грамматикой, которая по выразительной мощи в точности соответствует регулярным выражениям. В этой же главе устанавливается важное соответствие между распознаванием некоторого языка и решением вычислительной задачи гораздо более общего вида. Таким образом, получив тот или иной результат для задачи распознавания, можно сделать глубокие выводы о возможностях решения вычислительных задач вообще.

Шестая глава описывает *автоматы с магазинной памятью*, относящиеся к контекстно-свободным языкам так же, как и обычные конечные автоматы к языкам регулярным. Автоматы с магазинной памятью позволяют решить задачу распознавания контекстно-свободного языка на практике.

Седьмая глава развивает идею использования автоматов с магазинной памятью для решения задачи распознавания контекстно-свободных языков. Контекстно-свободные языки являются достаточно мощными для того, чтобы, например, использоваться для описания синтаксиса языков программирования. Поэтому вопросу их распознавания на практике посвящено много публикаций. В седьмой главе на уровне исходного кода будет построен простой анализатор описываемого контекстно-свободной грамматикой языка, основанный на имитации работы автомата с магазинной памятью.

Восьмая глава завершает разговор об использовании формальных языков для описания и компиляции компьютерных алгоритмов. Упрощенные модели и методики, приведенные в книге, не могут сравниться с инструментарием разработчика компилятора промышленного уровня. Для того чтобы дать хоть какое-то представление о том, как происходит создание компилятора на практике, я решил кратко описать современное средство Coco/R, специально предназначенное для облегчения труда разработчиков языков программирования.

В девятой главе вы сможете немного отдохнуть и познакомиться с *системами Линденмайера* — очень интересным и никак не связанным с компиляторами применением формальных языков. Вы увидите, как формальные языки могут быть использованы при описании сложных объектов окружающего нас мира. Не исключено, что материал этой главы даст хороший импульс вашему творческому воображению.

Десятая глава снова возвращает нас к проблеме ограниченности средств, которыми мы располагаем. В главе будет показано, что существуют формальные языки, не являющиеся контекстно-свободными. Для описания и распознавания таких языков потребуются более мощные средства, чем контекстно-свободные грамматики и автоматы с магазинной памятью. С несколькими разновидностями одного такого средства — *машиной Тьюринга* — и знакомит глава. К особенно интересным практическим и философским выводам приводит рассмотрение *универсальной машины Тьюринга*, способной имитировать работу любой «обычной» машины Тьюринга.

Напоследок я оставил главу с наиболее серьезными философскими выводами. Здесь рассматриваются ограниченность машин Тьюринга и ограниченность человеческого мозга; феномен сложности, относящийся, по-видимому, к фундаментальным явлениям природы; суть понятия алгоритма и многие другие не менее интересные вещи.

Глава 1 Регулярные языки и регулярные выражения

Прежде чем приступить к материалу главы, я хотел бы привести несколько простых, неформальных определений из математики, которые нам понадобятся.

Хотя эта книга, как уже говорилось, не является учебником по теории вычислений, и все излишние математические подробности будут обходиться везде, где это только возможно, совсем без математики не получится. Все-таки теория вычислений — предмет, связанный с математикой по своей природе. Да и есть ли смысл каждый раз повторять, к примеру, «язык, соответствующий регулярному выражению `a`», если можно просто написать L(a)?[1]

1.1. Начальные понятия

Алфавитом **называется любое конечное множество некоторых символов.** Например, множество {1, 2} — это алфавит, состоящий из единицы и двойки, множество {A, B, ..., Z} представляет собой алфавит заглавных латинских букв, а множество $\{a_1, a_2, a_3, a_4\}$ описывает алфавит из четырех элементов, о природе которых мы можем только догадываться. Как правило, алфавит обозначается какой-нибудь заглавной греческой буквой (например, $\Sigma = \{0, 1\}$).

Из алфавитных символов можно естественным образом составлять *строки*. **Если строка составлена из символов алфавита Σ, ее называют *строкой над алфавитом*** Σ. Например, 1001010111 — это строка над алфавитом {0, 1}, а `computer` — строка над алфавитом латинских букв. Пустая строка (то есть строка, вообще не содержащая символов) обычно обозначается буквой ε («эпсилон»). Длина такой строки равна нулю. Пожалуй, единственная операция над строками, которая нам понадобится — это *конкатенация* (приписывание одной строки в хвост другой). Так, если a = «abcd», а b = «efg», то ab = «abcdefg».

[1] Вообще говоря, смысл есть, если издательство платит «за кубометры», но этот вопрос остается за рамками книги.

Множество всех строк алфавита Σ имеет специальное обозначение: Σ^*. Таким образом, если Σ = {0, 1}, то Σ^* = {ε, 10, 0, 000, 1010, 01110, ...}.

Понятно, что над любым, даже простым алфавитом, можно составить бесконечно много строк. А что если выбрать не все, а лишь некоторые строки из множества Σ^*? Тогда мы получим *язык*[2]. Иначе говоря, **язык — это любое подмножество множества строк над используемым алфавитом**. Например, из всех возможных строк над алфавитом, состоящим из цифр, точки и знака «минус», можно выбрать лишь те, которые являются корректной записью некоторого вещественного числа: L = {0, -1.5, 1002.12123, -3.100, 78, ...}.

В большинстве случаев языки обозначают заглавными латинскими буквами. В отличие от множества строк, язык может представлять собой конечное множество. К примеру, если L — это язык над алфавитом {a, b}, в который входят лишь строки короче трех символов, то его элементами окажутся всего 7 строк: ε, a, b, aa, ab, ba и bb.

1.2. Регулярные выражения в теории

Только что мы затронули одно из основных понятий этой книги — понятие языка. При работе с любым конкретным языком первым делом возникает такая задача: поскольку язык — это некоторое множество, требуется каким-то образом уметь его описывать. В разговоре с приятелем, возможно, сгодятся и неформальные определения вроде «все корректные действительные числа», «любые веб-документы, удовлетворяющие стандарту HTML 4.0», «тексты программ на Паскале, реализующие сортировку пузырьком и не превосходящие при этом двух килобайт по размеру».

На практике состав того или иного языка приходится «объяснять» не только приятелю, но и компьютеру. Здесь ситуация, естественно, меняется коренным образом: любое описание обязательно должно быть строго формальным вне зависимости от наших желаний. Сейчас мы рассмотрим один популярный инструмент, часто использующийся для формального описания языка: *регулярные выражения*. Каждое регулярное выражение по сути является шаблоном, описывающим целое семейство (нередко бесконечное) строк.

Итак, допустим, требуется описать некоторый язык над алфавитом Σ. Регулярными выражениями над Σ являются любые элементы алфавита Σ, а также пустая строка ε:

- a, где a — это любой элемент алфавита Σ;
- ε.

[2] Для большей ясности нередко говорят «формальный язык».

Примечание.

Нередко к регулярным выражениям добавляют еще пустое множество ∅.

Также в свою очередь регулярными выражениями будут являться и результаты операций с регулярными выражениями над Σ:

(a ∪ b), (ab) и a*, где a и b — любые регулярные выражения над Σ

Пусть, к примеру, Σ = {a, b, c}. Любой непосредственный элемент алфавита является регулярным выражением над ним. Поскольку a и b — регулярные выражения над Σ, из них можно построить более сложное выражение (ab), которое также является регулярным выражением; из него, в свою очередь, получается конструкция (ab)* и так далее.

Другие примеры регулярных выражений над алфавитом {a, b, c}: ((ac)a)*, ((((ab) ∪ c)b)*c)*, ((a ∪ b)(b ∪ c)a).

Любое «базовое» регулярное выражение, состоящее из единственного элемента алфавита, описывает всего одну строку, включающую в себя лишь этот элемент. К примеру, выражению a соответствует строка «a». Кстати, язык, описываемый регулярным выражением r, обозначается L(r).

Регулярному выражению вида (a ∪ b) соответствуют все строки, задаваемые выражением a и все строки, задаваемые выражением b. Таким образом, если выражению[3] a соответствуют строки «a», «ab», «aaab», а выражению b — строки «baab» и «bbbb», итоговая конструкция (a ∪ b) опишет язык, состоящий из всех перечисленных пяти строк. Иначе говоря, операция ∪ («объединение») создает язык, являющийся объединением двух языков-операндов. Поскольку языки по определению являются обычными множествами, термин «объединение» не нуждается в дополнительных комментариях.

Регулярному выражению (ab) соответствует язык, состоящий из строк вида xy, где x — некоторая строка из языка L(a), а y — строка из языка L(b). То есть в язык L((ab)) входят все строки, которые можно получить путем дописывания в хвост некоторой строке из языка L(a) строки из языка L(b). Эта операция называется конкатенацией. Если воспользоваться примером из предыдущего пункта, L((ab)) = {«abaab», «abbbb», «abbaab», «abbbbb», «aaabbaab», «aaabbbbb»}.

Последняя операция, a* (так называемое *замыкание Клини*) задает язык, состоящий из строк ε, s_1, s_1s_2, $s_1s_2s_3$, ..., где s_i — любая строка из языка L(a). Поскольку мы уже обсудили смысл операции (ab) для регулярных выражений, замыкание Клини можно определить иначе: L(a*) = {ε, a, (aa), ((aa)a), (((aa)a)a), ...}, то есть, проще говоря, нуль

[3] Искренне сожалею, что приходится использовать слово «выражение» по три раза в каждом предложении... но понятность, увы, здесь важнее литературной красоты.

или более повторений a. Пусть снова L(a) = {«a», «ab», «bbbb»}. Тогда L(a*) = {ε, «a», «aaaaab», «ababbbbb», ... (любые комбинации, составленные из произвольного числа исходных строк)}.

Нетрудно заметить[4], что при составлении сколько-нибудь сложных регулярных выражений, можно очень легко утонуть в этих бесконечных открывающих и закрывающих скобках. Поэтому, хотя в теории подобные вещи не приветствуются, были введены приоритеты операций. В первую очередь выполняется замыкание Клини, затем конкатенация, и лишь затем — объединение. Таким образом, нет нужды писать перегруженное скобками выражение ((ab) ∪ (ba)) вместо более простого ab ∪ ba.

Поразмыслив пару минут, можно прийти также к выводу, что операции конкатенации и объединения обладают свойством ассоциативности, то есть L((a ∪ b) ∪ c) = L(a ∪ (b ∪ c)) и L((ab)c) = L(a(bc)). Этот факт позволяет также избавиться от лишних скобок в выражениях вроде a(b(cd)), заменяя их удобочитаемым abcd.

Вернемся теперь к первоначальной задаче, ради решения которой мы и погрузились в пучину регулярных выражений: требуется описать язык, то есть некоторое (возможно, бесконечное) множество строк строго формальным способом для дальнейшего использования на практике.

Регулярные выражения, несомненно, дают *некоторый* способ для подобного описания, но где гарантия, что с их помощью удастся описать *любой* язык, который мы в состоянии выдумать? Как и следовало ожидать, гарантий нет; более того, существуют достаточно простые языки, которые заведомо нельзя описать регулярным выражением. К примеру, такой: «любые строки, в которых буква “a” встречается столько же раз, сколько буква “b”».

Мы еще вернемся к этому грустному факту в последующих главах; пока же достаточно запомнить, что любой язык, который можно при желании описать регулярным выражением, сам называется *регулярным*.

Практически полезных регулярных языков, с одной стороны, быть может, не так много, но с другой — не так уж и мало. И, что более важно, встречаются они часто. Например, любой язык, состоящий из конечного числа строк, заведомо является регулярным. Так, язык L = {«abc», «aaa», «bab», «c»} описывается регулярным выражением abc ∪ aaa ∪ bab ∪ c.

[4] В данном случае использую этот оборот с чистой совестью!

1.3. Регулярные выражения на практике

Небольшая прелюдия

нигах теоретического характера обычно сначала рассматривают не-орую задачу и способы ее решения, а уже потом, если повезет — во-ос о том, зачем эту задачу решать вообще и кому это нужно. Сугубо кладные книги, как правило, проповедуют «проблемно-ориентиро-ный» подход: сначала рассмотрим возникшую на практике ситуацию, отом обсудим пути выхода из нее.

той книге, посвященной теории и практике одновременно, два опи-ных подхода приходится комбинировать. В теории для близкого зна-ства с понятием регулярных выражений достаточно самого факта су-ствования формальных языков, которые приходится как-то описывать. практике полезность тех же самых понятий еще требуется доказать.

смотрим вполне практический пример. Предположим, в интерфейсе шей новой программы требуется запросить у пользователя ввод неких ных. Любая уважающая себя программа должна иметь «защиту от рака», то есть проверять корректность введенной информации. Таким разом, принимается решение о добавлении в программу функции

```
ool isCorrect(string s),
```

вращающей `true` или `false` в зависимости от корректности строки `s`.

ли от пользователя запрашивается что-либо, имеющее очень простую уктуру, с написанием функции `IsCorrect()` не будет никаких про-м. К примеру, проверка корректности вводимого двоичного числа ограммируется в четыре строки:

```
or(int i = 0; i < s.Length; i++)
   if(s[i] != '0' && s[i] != '1')
       return false;
eturn true;
```

ли входные данные, напротив, имеют сложную структуру (к примеру, буется проверить, является ли строка `s` синтаксически корректной про-ммой на Паскале), функция `IsCorrect()` может оказаться запредельно жной. Интуитивно ясно, что вероятность третьего случая, когда входные ные вроде бы и не слишком сложны, но все же заметно хитрее двоичных сел (например, десятичные дроби), не так уж и мала.

ожество всех строк, являющихся корректными входными данными, определению есть некоторый язык. А что если этот язык удастся опи-ь с помощью регулярного выражения? Тогда, используя «волшебную нкцию», принимающую в качестве параметров тестируемую строку

s и регулярное выражение, задающее язык, можно узнать, является s элементом языка или нет. Осталось добавить только два замечания:

- «Волшебная функция», о которой только что шла речь, уже давным-давно написана.
- Будучи людьми любопытными, мы рассмотрим принцип ее устройства в следующей главе.

Итак, первое полезное свойство регулярных выражений очевидно. Если некий язык является регулярным, то проверить любую строку на принадлежность к этому языку очень просто (с помощью «волшебной функции»).

Перед тем, как перейти к деталям практического использования регулярных выражений, обсудим еще один пример.

Я скачал много-много страниц с различных интернет-форумов. Теперь мне требуется проанализировать скачанные страницы и скопировать все найденные на них корректные адреса электронной почты в отдельный файл[5].

Если «все корректные адреса электронной почты» — это язык (а по определению любое множество строк есть язык), то задачу можно сформулировать так: найти в некотором документе все строки интересующего языка.

К сожалению, решение этой задачи выходит за «пределы компетенции» стандартных регулярных выражений, рассматриваемых в теории вычислений. Дело в том, что теоретически регулярные выражения используются именно для описания языка — и, опять же, в теории их функция этим, в принципе, и ограничивается. Задача же вычленения из входной строки некоторой подстроки, принадлежащей языку, обходится молчанием.

Расширенные регулярные выражения

Для того чтобы все-таки воспользоваться инструментом регулярных выражений в данном случае (и в других задачах, которые мы рассмотрим позже), его придется расширить. Кроме того, есть смысл ввести кое-какие дополнительные элементы в «базовую конфигурацию» исключительно ради удобства.

[5] И не спрашивайте! Я все равно не признаюсь, зачем мне это понадобилось!

Некоторые элементы регулярных выражений в Microsoft .NET[6] *Таблица 1.1*

Элемент	Описание
одиночные символы, кроме . $ ^ {[(|) * + ? \	В программировании под регулярными выражениями подразумеваются регулярные выражения над алфавитом ASCII символов. Поэтому любой одиночный символ регулярного выражения (кроме перечисленных исключений) описывает соответствующий символ ASCII таблицы
\r	Обозначает символ с кодом 0Dh (возврат каретки)
\n	Обозначает символ с кодом 0Ah (новая строка)
\символ	Обозначает данный символ, если только сочетание не относится к отдельно обрабатываемому случаю (такому как \r или \n)
\\	Символ обратного слэша
.	Точка обозначает любой символ, за исключением новой строки. Фактически соответствует регулярному выражению 0 ∪ ... ∪ 9 ∪ A ∪ ... ∪ Z ∪ a ∪ ... ∪ z ∪ (другие ASCII-символы). Таким образом, точка — типичный символ, введенный для удобства. В теории же можно обойтись без нее. Если требуется указать саму точку (а не «любой символ, кроме новой строки»), придется воспользоваться записью «\.»
(выражение)	Круглые скобки используются обычным образом — так же, как и в теории.
|	Вертикальная черта («или») соответствует знаку объединения ∪
[символы] (например, [abcz])	Соответствует любому из перечисленных символов. Так, запись [abcz] эквивалентна выражению a ∪ b ∪ c ∪ z
[^символы] (например, [^abcz])	Соответствует любому символу, не входящему в набор. Таким образом, запись [^abcz] эквивалентна выражению 0 ∪ ... ∪ 9 ∪ A ∪ ... ∪ Z ∪ d ∪ ... ∪ y ∪ (другие ASCII-символы). В наборах символов разрешается также использовать диапазоны вида [a-z]. К примеру, запись [a-zA] обозначает любую строчную латинскую букву или заглавную A.
\w	Обозначает любой символ из набора [a-zA-Z_0-9]. Смысл может несколько варьироваться в зависимости от используемого стандарта регулярных выражений.
\W	Обозначает любой символ, не входящий в набор \w.
\s	Любой пробельный символ (пробел, табуляция, возврат каретки и т. п.)
\S	Любой непробельный символ.
\d	Любая цифра.
\D	Любой символ, не являющийся цифрой.
*	Замыкание Клини (жадное).
*?	Замыкание Клини (ленивое).
+	Единица или более повторов (жадное поведение). Запись a+ эквивалентна aa*.
+?	«Ленивый плюс». Запись a+? эквивалентна aa*?.
e{n}	Ровно n повторений выражения e. Так, e{3}, фактически, означает eee.
e{n,}	Как минимум n повторений выражения e (жадное поведение). Например, e{3,} — это eee+.
e{n,}?	Как минимум n повторений выражения e (ленивое поведение). Например, e{3,}? — это eee+?.

[6] К сожалению, человечество не пришло еще к единому стандарту расширенного синтаксиса регулярных выражений. К примеру, регулярные выражения в языке Perl отличаются от так называемого POSIX-стандарта. В таблице перечислены лишь наиболее широко используемые в различных стандартах элементы.

Продолжение табл. 1.1

Элемент	Описание
e{n,m}	Не менее n и не более m повторений выражения e (жадное поведение).
e(n,m)?	Не менее n и не более m повторений выражения e (ленивое поведение).
e?	Одно или ни одного вхождения выражения e (жадное поведение). Эквивалент записи e{0,1}. В терминах «теоретических» регулярных выражений эту конструкцию можно записать как (e ∪ ε).
e??	Одно или ни одного вхождения выражения e (ленивое поведение). Эквивалент записи e{0,1}?.
^e	Указывает, что выражение e должно быть найдено либо в начале файла, либо в начале очередной его строки.
e$	Указывает, что выражение e должно быть найдено либо в конце файла, либо в конце очередной его строки.
\Ae	Указывает, что выражение e должно быть найдено в начале файла.
e\z	Указывает, что выражение e должно быть найдено в конце файла.
\b	Указывает, что в данной позиции должна находиться граница слова (то есть символ из набора \w соседствует с символом из набора \W).

Из перечисленных в таблице элементов по-настоящему расширяют стандартные регулярные выражения лишь спецификации поведения (жадное / ленивое). Другие важные расширения мы обсудим позже, а пока вернемся к примерам, и, прежде всего, к поиску адресов электронной почты.

Не вдаваясь в тонкости спецификаций, адрес электронной почты можно рассматривать как строку, состоящую из произвольного количества символов (но не менее одного) из набора [a-zA-Z0-9._-], за которыми следует символ @ («собака»), затем опять идут элементы набора [a-zA-Z0-9._-]. Замыкают адрес точка и доменное имя — две, три или четыре буквы латинского алфавита. По счастливому стечению обстоятельств такую конструкцию можно описать с помощью регулярного выражения (то есть, выражаясь по-научному, язык допустимых адресов электронной почты регулярен):

```
[a-zA-Z0-9._-]+@[a-zA-Z0-9._-]+\.[a-zA-Z]{2,4}
```

Обратите внимание, что точку приходится экранировать символом обратного слэша, ибо простой знак «.» означает не точку, а «любой символ» (см. таблицу 1.1).

Формально описание языка теперь имеется, но на практике оно все еще несовершенно. Предположим, функция поиска корректных адресов обрабатывает вот такой файл:

```
За SIMM прощаюсь!
      С наилучшими пожеланиями, Маша Петрова
(МАШApetrova@pproducts.infoooo)
```

Адрес Маши Петровой определенно некорректен. Во-первых, он содержит буквы кириллицы, а во-вторых, имя домена здесь состоит из семи знаков. Тем

не менее, функция поиска вычленит подстроку `petrova@pproducts.info` в качестве корректного адреса и будет права.

К счастью, эта проблема решается просто. Надо лишь указать, что первый символ адреса электронной почты приходится на начало некоторого слова, а последний, соответственно — на конец:

```
\b[a-zA-Z0-9._-]+@[a-zA-Z0-9._-]+\.[a-zA-Z]{2,4}\b
```

Теперь разберемся, как регулярные выражения работают на уровне кода программы.

Листинг 1.1. Решение задачи о поиске корректных адресов электронной почты

```
static void Main(string[] args)
{
    // первый аргумент - имя обрабатываемого файла
    StreamReader sr = new StreamReader(args[0],
                          System.Text.Encoding.Default);
    string text = sr.ReadToEnd();
                    // считать входной файл в строку
    sr.Close();     // закрыть файл

    // создать объект "регулярное выражение"
    Regex r = new
      Regex("\\b[a-zA-Z0-9._-]+@[a-zA-Z0-9._-]+\\.[a-zA-Z]{2,4}\\b");
    MatchCollection mc = r.Matches(text);
                    // получить список найденных адресов

    for(int i = 0; i < mc.Count; i++)
                    // и распечатать его
        Console.WriteLine(mc[i].Value);
}
```

Регулярные выражения являются частью спецификации некоторых популярных языков программирования, таких как Perl и PHP, а также платформ Java и .NET. Для других языков, напрямую не поддерживающих это понятие, существуют нестандартные библиотеки. Если есть спрос, то появляются и предложения, такие как PCRE для C/C++ или TRegExp для Delphi. Эти библиотеки можно свободно скачать в интернете и насладиться всеми преимуществами регулярных выражений. В большинстве случаев авторы предоставляют сходные функции, поэтому работа с регулярными выражениями в C# мало чем принципиально отличается от применения библиотеки PHP.

Для поиска корректных адресов электронной почты в листинге 1.1 используются два класса: `Regex` и `MatchCollection`.

Экземпляр класса `Regex` в Microsoft .NET служит для хранения одиночного регулярного выражения. Идея хранения регулярного выражения не просто в какой-то строковой переменной, а в особой структуре, достаточно популярна[7]. Дело в том, что перед использованием регулярное выражение приходится компилировать[8]. Хранить в строковой переменной можно только некомпилированное выражение, поэтому при каждой операции поиска требуется перекомпиляция. Объект `Regex`, напротив, хранит уже откомпилированную версию, что при многократном использовании дает выигрыш в скорости работы.

С помощью метода `Matches()` можно получить ссылку на коллекцию найденных строк, соответствующих регулярному выражению. В качестве параметра методу передается строка, в которой, собственно, и осуществляется поиск.

Обратите внимание, что внутри строк в C# (как и во всех родственных языках) символ обратного слэша должен дублироваться.

Я не случайно поместил реализацию решения задачи о поиске адресов электронной почты до программы, проверяющей корректность входных данных (помните этот пример с «защитой от дурака»?) Дело в том, что проверку строки на принадлежность языку можно рассматривать всего лишь как частный случай задачи поиска: надо лишь указать в регулярном выражении, что первый символ найденной строки должен совпадать с началом проверяемого файла, а последний — с концом:

```
\A(выражение)\z
```

Если метод `Matches()` при этом найдет соответствие, можно сделать вывод о корректности введенной информации.

Чтобы довести эту идею до практического воплощения, предположим, что нас интересует корректный ввод действительных чисел. Действительное число представляет собой необязательный знак «минус», за которым следует некоторое количество цифр, а за ними — необязательная часть из десятичной точки и еще одной последовательности цифр. Тогда на языке регулярных выражений (с учетом указания начала и конца файла) действительное число можно записать так:

```
\A-?[0-9]+(.[0-9]+)?\z
```

[7] В принципе, в данном случае можно было бы воспользоваться статическим методом `Regex.Matches()`, но при этом среда .NET все равно создает неявный объект `Regex`.

[8] Конечно, процесс компиляции регулярного выражения отличается от компиляции программы на C#, тем не менее, использование этого термина прижилось. Сейчас главное то, что перед процессом поиска приходится сначала выполнять некоторый достаточно трудоемкий алгоритм.

Программа, приведенная в листинге 1.2, печатает True или False в зависимости от корректности ввода. Выход — по вводу пустой строки.

Листинг 1.2. Проверка корректности действительных чисел

```
string s;
Regex r = new Regex("\\A-?[0-9]+(.[0-9]+)?\\z");

while((s = Console.ReadLine()) != "")
    Console.WriteLine(r.Matches(s).Count == 1);
```

Ленивое и жадное поведение

Перед тем как перейти к другим применениям регулярных выражений, стоит уделить немного внимания понятиям *ленивого* (lazy) и *жадного* (greedy, eager) поведения, ссылки на которые уже приводились в табл. 1.1. Сделать это проще всего с помощью двух несложных примеров.

Пример 1. В некотором текстовом документе есть (одна и только одна) секция со списком людей и их телефонных номеров:

```
1. Иванов 123-45-67
2. Петров 234-56-78
3. Сидоров 345-67-89
```

Требуется напечатать на экране эту секцию.

Пример 2. В том же самом документе автор выделил блоки особо важного текста при помощи своих собственных тегов [imp]...[/imp]:

```
Этот текст может быть интересным, и даже очень.
Но он совсем не важен.
[imp]А вот этот текст на редкость скучен; но что
поделаешь — он важен![/imp]
Здесь идет секция юмора...
[imp]А здесь — протокол заседания комитета JX1321 от
11.05.2004[/imp]
```

Требуется напечатать на экране содержимое всех важных блоков.

На первый взгляд, обе задачи просты. В первом случае секцию с информацией о людях можно описать с помощью регулярного выражения

```
1\..+\d\d\d-\d\d-\d\d,
```

а блоку важного текста из второго случая соответствует выражение

```
\[imp].*\[/imp]
```

В действительности все не так просто. Описанию «единица, точка, затем сколько угодно любых символов, затем телефонный номер в формате nnn-nn-nn» соответствует и блок `1. Иванов 123-45-67`. Никаких объективных причин для того, чтобы метод `Matches()` продолжил поиск до строки о данных Сидорова, нет.

Отсюда вывод: в некоторых случаях требуется, чтобы поведение функции поиска было жадным, то есть захватывался бы как можно больший участок текста, соответствующий регулярному выражению.

Так, может быть, жадное поведение — это то, что нужно всегда? Нет. Во втором примере жадная функция поиска вернет весь текст от первого [imp] до последнего [/imp]. Таким образом, в некоторых случаях мы, наоборот, нуждаемся в ленивом поведении — метод `Matches()` должен вернуть подстроки наименьшей возможной длины.

Изменить текст метода `Matches()` нельзя, зато можно использовать спецификации жадного и ленивого поведения. Точнее, жадное поведение используется по умолчанию (поэтому регулярное выражение из первого примера переделывать не нужно), а для указания поведения ленивого придется добавить вопросительный знак (см. таблицу 1.1) после звездочки во втором примере:

```
\[imp].*?\[/imp]
```

Итоговый код программы приводится в листинге 1.3.

Листинг 1.3. Ленивое и жадное поведение

```
StreamReader sr = new StreamReader(args[0],
                      System.Text.Encoding.Default);
string text = sr.ReadToEnd(); // считать входной файл
                              // в строку
sr.Close();                   // закрыть файл

Regex r1 = new Regex("1\\..+\\d\\d\\d-\\d\\d-\\d\\d",
                     RegexOptions.Singleline);
Regex r2 = new Regex("\\[imp].*?\\[/imp]",
                     RegexOptions.Singleline);

Match m = r1.Match(text);
                // метод Match(), аналогичный Matches()
// возвращает лишь первую найденную подстроку
Console.WriteLine(m.Value);
Console.WriteLine();
```

```
MatchCollection mc = r2.Matches(text);
for(int i = 0; i < mc.Count; i++)
   Console.WriteLine(mc[i].Value.Substring(5, mc[i].Value.Length - 11));
```

Остается лишь добавить несколько замечаний к теме жадности и лени:

- По умолчанию в .NET точка считается не «любым символом», а «любым символом, кроме символа новой строки». Поскольку для нас подобное поведение нежелательно, приходится устанавливать для регулярного выражения особый параметр `RegexOptions.Singleline`. Теперь точка — это действительно любой символ.
- Метод `Matches()` возвращает список непересекающихся подстрок, удовлетворяющих критерию поиска. К примеру, вызов `Matches()` для регулярного выражения `aaa` и тестируемой строки `aaaa` вернет лишь одну подстроку, найденную в самом начале тестируемой строки.
- При нахождении участков «важного» текста на печать надо было выводить не весь найденный фрагмент, а лишь подстроку внутри тегов [imp]...[/imp]. В программе это сделано не самым красивым образом: от строки просто отрезалось пять символов слева и шесть символов справа. Очень скоро мы разберемся, как решить такую задачу более правильно.

Группы, ссылки, подстановки

Ситуация, когда требуется использовать не всю найденную с помощью регулярного выражения подстроку, а лишь некоторую ее часть, довольно часто встречается на практике:

- В примере с конструкцией [imp]...[/imp] полезным текстом являлась лишь подстрока между тегами;
- при анализе найденных адресов электронной почты нас вполне может интересовать частота встречаемости тех или иных доменов;
- при работе с иногородними телефонными номерами код города может быть полезен как отдельное поле.

Любая реализация регулярных выражений предоставляет возможность заключить часть выражения в *группу*, чтобы затем обращаться не ко всей найденной подстроке сразу, а к любой ее группе по отдельности.

Задать группу несложно: для этого требуется лишь заключить часть регулярного выражения в круглые скобки. К примеру, если в адресе электронной почты имя пользователя (часть до «собаки»), название сервера (часть после «собаки») и доменное имя (2-4 символа за последней точкой) представляют самостоятельный интерес, это можно выразить так:

```
\b([a-zA-Z0-9._-]+)@([a-zA-Z0-9._-]+\.([a-zA-Z]{2,4}))\b
```

Обратите внимание, что группы могут быть вложенными. Так, группа доменного имени является частью группы названия сервера. В Microsoft .NET группы могут быть именованными. Для того чтобы задать имя группы, сразу после открывающей круглой скобки придется поставить вопросительный знак, а за ним — имя в угловых скобках: (?<имя_группы> ...) Регулярное выражение, соответствующее адресу электронной почты, с использованием именованных групп выглядит так:

```
\b(?<user>[a-zA-Z0-9._-]+)@
(?<serv>[a-zA-Z0-9._-]+\.(?<dom>[a-zA-Z]{2,4}))\b
```

Если регулярное выражение достаточно сложно и содержит множество групп, использование имен может сделать работу с ним намного удобнее (работая с именами, гораздо легче ориентироваться, где что. Кроме того, далее будет рассмотрен механизм ссылок, использующий имена групп для доступа к ним).

Получить доступ к группам найденной подстроки (объекта `Match` или элемента коллекции `MatchCollection`) можно при помощи свойства `Groups`. Если группа является именованной, то доступ к ней производится, что логично, с помощью имени:

```
MatchCollection mc = r.Matches(text);
                // получить список найденных адресов
for(int i = 0; i < mc.Count; i++)
                // и распечатать его
   Console.Write("имя={0} сервер={1} домен={2}\n",
                    mc[i].Groups["user"].Value,
                    mc[i].Groups["serv"].Value,
                    mc[i].Groups["dom"].Value);
```

В противном случае придется использовать порядковый номер:

```
MatchCollection mc = r.Matches(text);
                // получить список найденных адресов
for(int i = 0; i < mc.Count; i++)
                // и распечатать его
    Console.Write("имя={0} сервер={1} домен={2}\n",
                    mc[i].Groups[1].Value,
                    mc[i].Groups[2].Value,
                    mc[i].Groups[3].Value);
```

Группы нумеруются по порядку слева направо. Самая левая группа[9] имеет порядковый номер 1, затем идет группа 2 и т. д. Нулевой номер зарезервирован для всей подстроки, соответствующей регулярному выражению.

[9] Точнее, группа, чей первый символ находится левее первого символа любой другой группы.

С группами связана еще одна тема — *ссылки* (backreferences). Ее просто нельзя обойти стороной, поскольку при использовании ссылок «расширенные» (то есть выходящие за рамки строгого математического определения, данного в начале главы) регулярные выражения выходят далеко за рамки того, что называется регулярными выражениями в теории, и эти различия надо понимать.

Ссылки позволяют использовать в регулярном выражении содержимое уже найденной группы. Допустим, нас интересуют строки `aaa-aaa`, `server-server`, `name-name` и тому подобные, в которых первая часть в точности совпадает со второй частью и отделена от нее знаком «минус». Как найти такие строки с помощью регулярного выражения? Ответ: с помощью ссылок. Запись `\номер` внутри регулярного выражения означает найденную строку, соответствующую группе с номером `номер`:

```
(.*)-(\1)
```

Левая часть регулярного выражения находит некоторую строку, заканчивающуюся минусом, а правая проверяет, совпадает ли найденная строка с участком той же длины справа от минуса.

В Microsoft .NET ссылка на именованную группу выглядит как `\k<имя_группы>`. Таким образом, выражение `(.*)-(\1)` с использованием ссылок на именованные группы выглядит так:

```
(?<left>.*)-(\k<left>)
```

Понятно, что ссылки — полезное средство, увеличивающее выразительную силу регулярных выражений. Проблема лишь в том, что язык, описываемый приведенным выражением, *не является регулярным*. Вернее сказать, никаких проблем в этом факте нет; просто необходимо отметить четкое различие. Определение регулярного языка как языка, который может быть описан при помощи регулярного выражения, не относится к регулярным выражениям Perl, Java или .NET. Оно относится лишь к тому понятию, которое описано под названием регулярных выражений в посвященной теоретическим аспектам части этой главы. С помощью же «настоящих» регулярных выражений, упоминаемых в любом учебнике по теории вычислений, описать язык, состоящий из строк вида `aaa-aaa`, `server-server`, `name-name`, действительно нельзя.

При помощи большинства популярных библиотек, реализующих работу с регулярными выражениями, можно не только *искать* некоторые подстроки, но и *заменять* их на новые. Для этого служит, в частности, метод `Replace()` класса `Regex` и механизм *подстановок*. Формат вызова метода прост:

```
Regex r = new Regex(рег_выражение);
результат = r.Replace(исходная_строка, подстановка);
```

В тривиальном случае подстановка — это просто строка, на которую заменяется любая найденная последовательность, соответствующая регулярному выражению. Так, фрагмент

```
Regex r = new Regex("[0-8]");
Console.WriteLine(r.Replace("Я хочу зарабатывать $10 000" +
                                        "в месяц", "9"));
```

заменяет любую найденную в тексте цифру на девятку, тем самым печатая на экране гораздо более похожие на правду пожелания.

В строке подстановка можно использовать также специальные шаблоны замены (ограничимся тремя):

Шаблоны замены в Microsoft .NET *Таблица 1.2*

Шаблон	Описание
$number	Заменяется содержимым группы number регулярного выражения.
${name}	Заменяется содержимым именованной группы name регулярного выражения.
$$	Заменяется одиночным знаком доллара.

Рассмотрим практический пример применения шаблонов замены.

Некий вебмастер почему-то недолюбливает людей, извлекающих адреса электронной почты с его страниц при помощи программы из листинга 1.1. Чтобы осложнить им жизнь, он решает заменить в адресах символ @ на строку « at », а точку перед именем домена — на строку « dot ». Человек, решивший написать письмо, все равно догадается, что к чему, а вот автоматическое извлечение адресов из интернет-страничек работать перестанет.

Программа, автоматизирующая работу этого вебмастера, приводится в листинге 1.4.

Листинг 1.4. Использование шаблонов замены

```
// первый аргумент - имя обрабатываемого файла
StreamReader sr = new StreamReader(args[0],
                        System.Text.Encoding.Default);
string text = sr.ReadToEnd();
                        // считать входной файл в строку
sr.Close();             // закрыть файл

// найти и заменить
Regex r = new Regex("\\b(?<user>[a-zA-Z0-9_.-]+)@" +
```

```
                    "(?<server>[a-zA-Z0-9._-]+)" +
                    "\\.(?<domain>[a-zA-Z]{2,4})\\b");
string result = r.Replace(text, "${user} at ${server} dot ${domain}");

// записать результирующий файл
StreamWriter sw = new StreamWriter("new " + args[0], false,
                                   System.Text.Encoding.Default);
sw.Write(result);
sw.Close();
```

Комментировать здесь особо нечего: адрес электронной почты, записанный в стандартном формате, фактически переписывается в виде `{user} at {server} dot {domain}`, где `{user}`, `{server}` и `{domain}` — содержимое соответствующих групп.

1.4. Регулярные выражения в программных продуктах

Как мы уже выяснили, на практике регулярные выражения обычно применяются для поиска в тексте участков, подходящих под заданное (как раз с помощью регулярного выражения) описание. Сразу же хочу заметить, что авторы программ, так или иначе связанных с поиском в документах, уже оценили мощь регулярных выражений. Так, мой любимый файловый менеджер Total Commander позволяет находить на диске файлы с именами, соответствующими задаваемому регулярным выражением образцу. Например, можно выделить лишь звукозаписи, названные в стиле «исполнитель — композиция.mp3». Подобным названиям будет соответствовать регулярное выражение .* – .*\.mp3 (см. рис. 1.1).

Total Commander позволяет искать также подстроки, заданные регулярным выражением, в содержимом файлов. Кроме того, предусмотрено средство пакетного переименования файлов, поддерживающее регулярные выражения.

Поскольку единого стандарта для синтаксиса регулярных выражений и набора средств, ими предоставляемых, нет, вам, к сожалению, вряд ли удастся обойтись без чтения документации в большинстве случаев.

Поиском и заменой подстрок в содержимом файлов занимается и утилита PowerGREP, которую можно скачать с сайта www.powergrep.com. По правде говоря, она мне кажется не очень полезной в повседневном использовании (кроме того, программа стоит денег), но для изучения регулярных выражений и наглядного демонстрирования их возможностей весьма ценна.

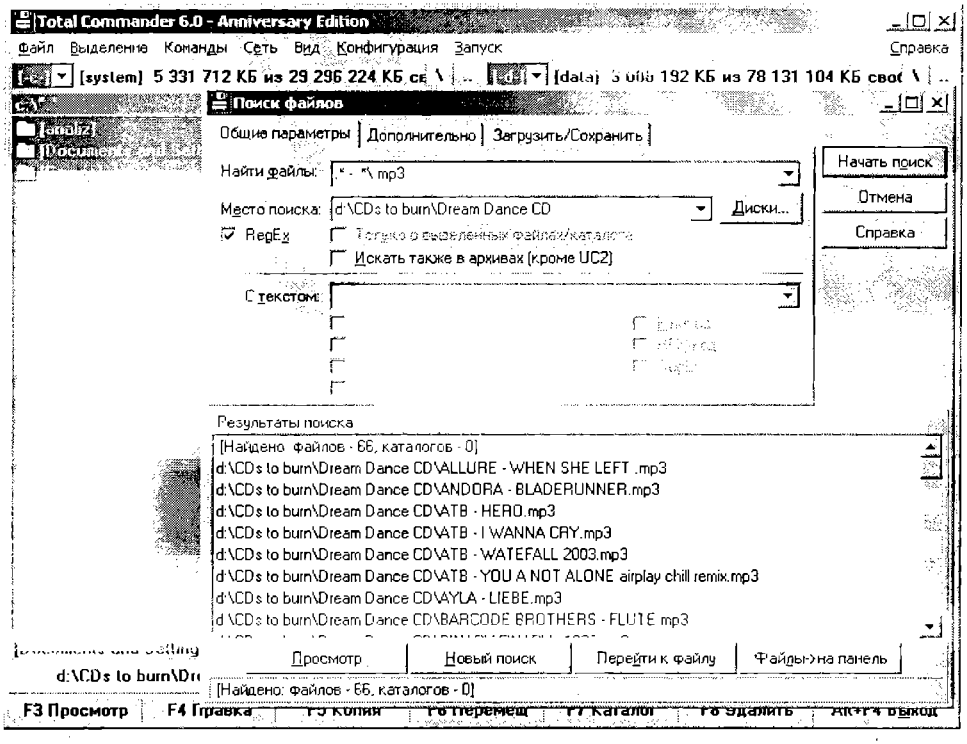


Рис. 1.1. *Поиск файлов с помощью регулярного выражения*

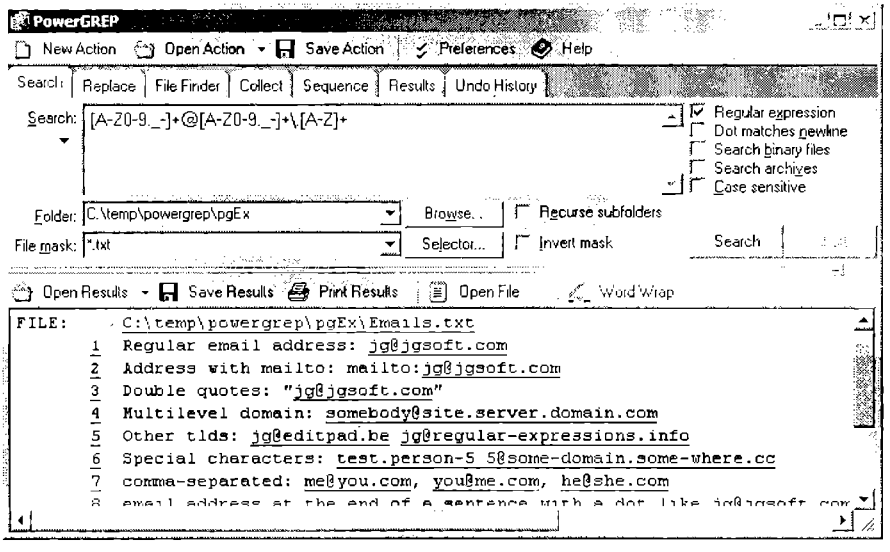


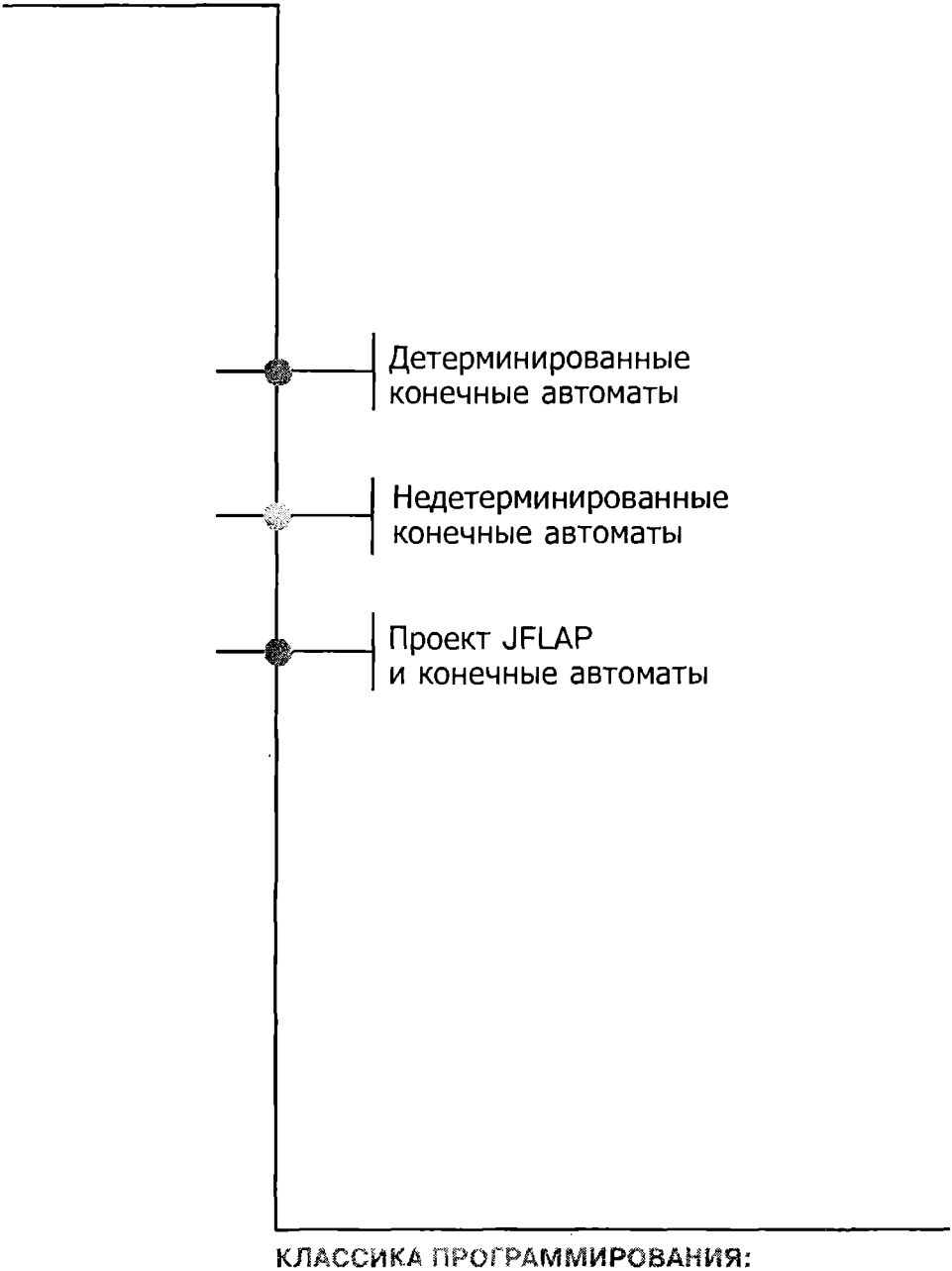
Рис. 1.2. *Утилита PowerGREP*

Интерфейс программы вполне стандартен. Вы задаете путь к документам и регулярное выражение, служащее шаблоном для поиска, а система выдает список найденных совпадений с подсветкой, что действительно наглядно (см. рис. 1.2).

Итоги

- Регулярные выражения (в теории) — это инструмент, позволяющий сравнительно просто описать некое, возможно, бесконечное множество строк (язык).
- Регулярные выражения дают нам возможность выделить языки, ими описываемые, в отдельный класс — класс регулярных языков.
- На практике регулярные выражения используются обычно для поиска (и, возможно, последующей замены) строк регулярного языка в некотором текстовом документе.
- Поведение функции поиска может быть ленивым или жадным. Программист может явно указать требуемое поведение в любом конкретном случае.
- Отдельные средства регулярных выражений платформы .NET, а также некоторых других инструментов разработки (Perl, PHP, Java, ...), позволяют описывать строки языков, строго говоря, не являющихся регулярными.
- Обратите также внимание, что мы пока не рассматривали вопрос о том, насколько регулярные выражения эффективны в алгоритмическом плане (иначе говоря, как быстро они работают). В следующей главе этот недостаток будет исправлен.

Глава 2 Конечные автоматы



Конечные автоматы — едва ли не самая интересная тема этой книги. Конечные автоматы являются исключительно полезными как в теории компьютерной науки, так и в реальной программистской практике.

В теоретических работах автоматы используются в качестве средства описания формальных языков, а также «модели вычисления» (смысл этого выражения станет ясен позже), а на практике они представляют собой мощный психологический инструмент, помогающий в некоторых случаях писать простые и надежные программы.

В этой главе мы познакомимся с разнообразными видами конечных автоматов, рассмотрим различные алгоригмы, с ними связанные, и обсудим применение автоматов как в теоретических построениях, так и в реальных программах.

2.1. Детерминированные конечные автоматы

Неформальное введение

Начнем, как и в предыдущий раз, с задачи формального описания языка. Как мы уже выяснили, регулярные выражения позволяют решить эту задачу, но лишь для некоторых, так называемых регулярных языков. Что же делать, если интересующий нас язык не является регулярным? Придется изобретать другие, более мощные инструменты. Кроме того, регулярные выражения, как мы уже упоминали, анализируются на практике «волшебной функцией»[10], чье внутреннее устройство пока что остается скрытым.

Запрограммировать подобную «волшебную функцию», а затем и вовсе вырваться за пределы регулярных языков нам помогут конечные автоматы. Первым в ряду конечных автоматов является так называемый *детерминированный конечный автомат* (deterministic finite automaton).

[10] Определяющей, является ли переданная строка элементом языка, задаваемого регулярным выражением.

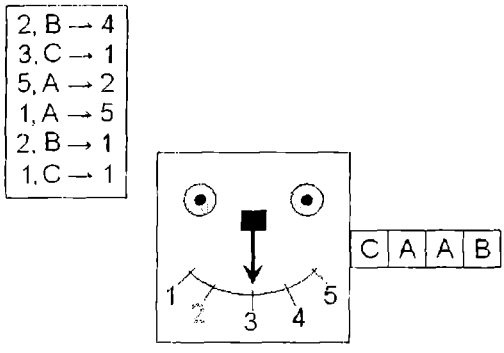


Рис. 2.1. *Детерминированный конечный автомат (наглядное пособие)*

Посмотрите на рис. 2.1. Квадратный серый ящик со шкалой и стрелкой посередине и есть модель детерминированного конечного автомата. Заметьте, я говорю о «модели», а не об «определении». Речь сейчас идет об абстрактном понятии, и любые картинки могут лишь иллюстрировать те или иные его особенности, но ни одна из них не в состоянии претендовать на истинное определение.

Итак, на какие элементы автомата следует обратить внимание?

- Сам по себе автомат является неким «виртуальным устройством», умеющим выполнять определенного рода работу.
- У автомата есть подвижная «стрелка», указывающая на то или иное число, называемое его *состоянием*. Таким образом, изображенный на рисунке автомат в настоящий момент находится в состоянии номер три.
- Самих состояний может быть сколь угодно много, но их число *конечно* (поэтому и автомат называется конечным).
- Среди всех состояний можно выделить «особые»: это, в первую очередь, *начальное* состояние и одно или более *допускающих* (favorable)[11] состояний. Начальное состояние — это состояние автомата на момент запуска (то есть пока он еще не выполнил никакой работы). Допускающие состояния ничем не отличаются от обычных, просто их набор оговаривается заранее, и мы в любой момент знаем, какое состояние считается допускающим, а какое — нет. На рисунке допускающие состояния выделены контурным шрифтом (таким образом, у нашего автомата всего одно допускающее состояние — номер два). **Допускающие состояния нужны для анализа результатов работы автомата. В зависимости от того, относится ли на момент завершения всех действий текущее состояние к допускающим или не относится, мы будем делать различные выводы о данных, поступающих автомату на вход.**

[11] Допускающие состояния иногда называют еще *заключительными* (final), но, на мой взгляд, это название не слишком удачное.

- На правой стенке автомата установлено считывающее устройство, умеющее обрабатывать бумажные ленты конечной длины, разграфленные на клетки и содержащие в каждой клетке некий символ (разумеется, пустая клетка тоже в известном смысле может считаться символом). Лента в нашем случае содержит строку CAAB.
- В памяти автомата (нарисована слева вверху) находится конечная «таблица правил» (или «таблица *переходов*»), сверяясь с которой, автомат определяет, что делать в той или иной ситуации. По существу, набор его действий состоит только из переходов в то или иное состояние (как видно из рисунка, кроме подвижной стрелки у автомата нет никаких активно действующих узлов вроде встроенного лазерного принтера или бортовой крупнокалиберной пушки). Элементы таблицы правил следует читать так: «если очередной входной символ — B, а текущее состояние — второе, то переходим в состояние номер четыре», «если очередной входной символ — C, а текущее состояние — третье, то переходим в состояние номер один» и так далее.
- Пара (текущее состояние, очередной входной символ) однозначным образом определяет используемое в данной ситуации правило перехода. Поэтому автомат называется детерминированным.

Автомат работает по следующей простой схеме. Сначала считывается очередной входной символ с ленты, затем производится переход в новое состояние согласно таблице правил, затем цикл повторяется. Работа заканчивается после того, как входная лента будет обработана целиком. Таким образом, процесс работы автомата, изображенного на рис. 2.1, выглядит так:

```
начальное состояние = 3
считать символ (результат = C)
перейти к состоянию 1 (правило 3, C -> 1)
считать символ (результат = A)
перейти к состоянию 5 (правило 1, A -> 5)
считать символ (результат = A)
перейти к состоянию 2 (правило 5, A -> 2)
считать символ (результат = B)
перейти к состоянию 4 (правило 2, B -> 4)
закончить работу
```

На момент завершения работы текущим состоянием автомата является состояние номер четыре.

Представление детерминированного конечного автомата в виде графа

Представлять конечные автоматы в виде рисунков удобно лишь для объяснения принципа их действия. Для того же, чтобы уяснить структуру

того или иного конкретного автомата, гораздо удобнее изобразить его в виде чертежа, состоящего из кружочков и стрелок (направленного графа). Этот способ очень часто используется в книгах по теории автоматов.

Рассмотрим (см. рис. 2.2) граф, соответствующий уже знакомому нам по рис. 2.1 автомату.

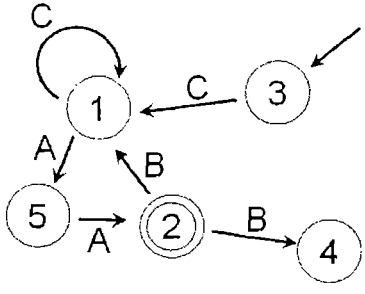


Рис. 2.2. *Представление конечного автомата с помощью графа*

Как «читать» такой чертеж? Кружочки (*узлы* графа) представляют собой состояния конечного автомата. Сразу же видно, что их пять, и обозначены они с помощью чисел от 1 до 5.

Допускающее состояние (второе) отмечено двойной границей круга. Стрелки (*ребра* графа) наглядно изображают таблицу правил (например, правило 3, $C \rightarrow 1$ обозначается ребром, помеченным буквой C, которое соединяет узел 3 с узлом 1). Стрелка, идущая «из ниоткуда», указывает на начальное состояние автомата (третье).

FAQ по конечным автоматам (первая часть)

Вопрос. Почему состояния автомата помечены числами, а символы входной ленты — буквами латинского алфавита?

Ответ. Просто так. Вы можете выбирать любой способ указания состояний (например, q_0, q_1,..., q_N или `StateOne`, `StateTwo`, `LastState`, `InitState`) и какие угодно элементы в качестве допустимых символов входной ленты.

Вопрос. Может ли автомат не иметь начального состояния? И как насчет допускающих?

Ответ. Начальное состояние должно быть указано всегда. Теоретически, в автомате вообще без допускающих состояний нет ничего некорректного, но практическая польза от него весьма сомнительна.

Вопрос. Может ли автомат, находящийся в допускающем состоянии, выйти из него? Или переход в такое состояние означает конец работы?

Ответ. Цитирование самого себя — дурной тон, но все-таки этот вопрос уже был освещен в главе: «допускающие состояния ничем не отличаются от обычных, просто их набор оговаривается заранее, и мы в любой момент знаем, какое состояние считается допускающим, а какое — нет».

Вопрос. А что будет, если указать в таблице несколько правил, противоречащих друг другу? Например, 5, A → 1 и 5, A → 2? В какое состояние перейдет автомат из состояния номер пять, если на входе встретился символ A?

Ответ. Такие ситуации запрещены. Построенное устройство попросту не будет детерминированным конечным автоматом.

Вопрос. А если произойдет нештатная ситуация? Предположим, наш автомат (рис. 2.2) находится в состоянии номер два, а на вход поступил символ C? Такой вариант не предусмотрен правилами!

Ответ. Я рад, что вы спросили — это действительно интересный случай. В теории такого быть не должно. Иными словами, следует всегда предусматривать все возможные случаи. Некоторые авторы действительно с фанатичным упорством рисуют в графах автоматов все мыслимые и немыслимые ребра. На практике[12] случай ненайденного правила перехода следует рассматривать как особую ситуацию (аналогичную, к примеру, ошибке типа «файл не найден») со всеми вытекающими отсюда последствиями.

Что вы будете делать, если открываемый вашей программой файл не найден? Вам дадут знать о возникновении этой нештатной ситуации, но лишь вам решать, как поступить дальше. Можно просто проигнорировать ошибку (скорее всего, ничего хорошего из этого не выйдет). Можно вывести сообщение пользователю и аварийно завершить выполнение процедуры. А можно и отформатировать винчестер для разнообразия.

Вопрос. Хм... если у нас есть правило 1, A → 2, рисуем стрелку между узлами 1 и 2; если добавляется правило 1, B → 2, то стрелок будет уже две? А если требуется перейти из первого состояние во второе по любой букве латинского алфавита, кроме Z? Потребуется 25 стрелок?!

[12] Здесь выражение «на практике» означает «на уровне реально работающей программы». Программной реализацией автоматов мы займемся чуть позже.

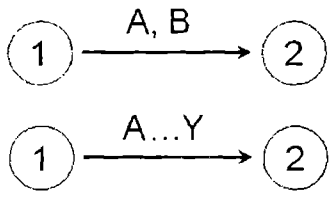


Рис. 2.3. *Сокращенная запись праил перехода*

Ответ. Поскольку граф конечного автомата предназначен для чтения человеком, а не компьютером, различного вида сокращенные записи приветствуются. Как следует поступить в данном случае, показано на рис. 2.3.

Описание языка с помощью конечного автомата

Вернемся теперь к задаче, красной нитью проходящей через весь материал глав — задаче формального описания некоторого языка. В качестве затравки предлагаю рассмотреть уже изученный нами язык, состоящий из всех допустимых адресов электронной почты, и обсудить, как же его можно описать при помощи конечного автомата. В прошлой главе мы ограничились несколько упрощенным представлением о допустимом почтовом адресе, которому соответствует регулярное выражение

```
[a-zA-Z0-9._-]+@[a-zA-Z0-9._-]+\.[a-zA-Z]{2,4}
```

Сейчас это выражение, к сожалению, придется упростить еще раз, иначе автомат получится слишком громоздким:

```
[a-zA-Z0-9._-]+@[a-zA-Z0-9_-]+(\.[a-zA-Z0-9_-]+)+
```

Хотя на первый взгляд кажется, что выражение стало даже сложнее, в действительности его структура упростилась[13].

Рассмотрим теперь работу конечного автомата, изображенного на рис. 2.4 на конкретных примерах.

Пусть входная лента (считается, что на ней могут быть записаны символы a...z, A...Z, 0...9, точка, знак нижнего подчеркивания, а также знаки «минус» и «собака») содержит строку `zeus@olympus`.

Изначально автомат находится в состоянии 1. Считав первый символ ленты (букву `z`), он перейдет во второе состояние. Оставшиеся три символа до «собаки» не изменят текущего состояния автомата, а сама «собака»

[13] Я понимаю, что на данном этапе это совсем неочевидно, но в будущем, надеюсь, станет ясно, чем первое выражение сложнее второго.

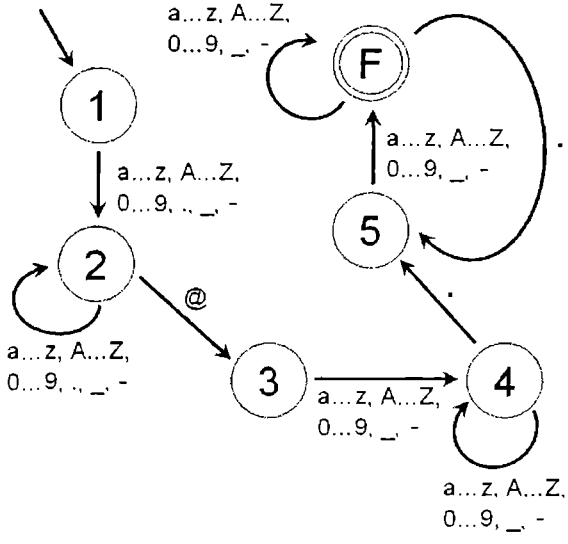


Рис. 2.4. *Конечный автомат, распознающий корректные адреса электронной почты*

ереведет его в состояние номер 3. Буква o заставит автомат перейти в остояние 4. Оставшиеся символы ничего не изменят, и в итоге после бработки всей ленты автомат так и будет находиться в состоянии 4.

Теперь посмотрим, что будет, если входная лента содержит строку `zeus@olympus.gov`.

Символы до точки, как мы уже знаем, заставят автомат перейти в состояние 4. Сама точка переведет его в состояние 5, а оставшиеся символы — в состояние F, на чем работа автомата и завершится.

Я не буду заниматься строгим доказательством, но, надеюсь, на неформальном уровне видно: автомат, изображенный на рис. 2.4, заканчивает работу в допускающем состоянии тогда и только тогда, когда входная лента содержит корректный адрес электронной почты в соответствии с приведенным выше регулярным выражением.

Немного терминологии. Говорят, что автомат *допускает* (accepts) строку, если эта строка переводит его в одно из допускающих состояний. В противном случае говорят, что автомат *отвергает* (rejects) строку. Множество всех строк, допускаемых данных автоматом, образует язык, им *распознаваемый*.

Напомню, что под строкой понимается цепочка символов над произвольным алфавитом, поэтому прибор, анализирующий последователь-

ности объектов любой природы, также может относиться к конечным автоматам.

Итак, любая входная строка либо допускается, либо отвергается автоматом. Нештатные ситуации («правило не найдено»), как уже было сказано, могут обрабатываться как угодно, но допускать строку, вызвавшую сбой, автомат уж точно не должен.

Детерминированный конечный автомат на языке C#

Об автоматах можно сказать еще очень много. Мы до сих пор так и не определили само понятие автомата более строго, не разобрались с разновидностями автоматов, взаимосвязями между ними, не выяснили, что роднит детерминированные конечные автоматы с регулярными выражениями. Тем не менее, мне кажется, что пора остановиться и привести хоть один реально работающий на практике пример.

Замечательная особенность детерминированного конечного автомата состоит в том, что, глядя на его граф, можно легко, механически написать программу, этот автомат реализующую. Более того, такую работу вполне реально поручить и компьютеру. С точки зрения эффективности полученная программа будет оптимальной (при наличии хорошего компилятора, но об этом позже). Нетрудно также написать и «обобщенный автомат», оперирующий в соответствии с правилами, считываемыми из любого заданного файла (правда, сделать такую программу столь же скоростной сложнее).

Предположим, построен некоторый автомат, состояния которого называются `S1`, `S2`, ..., `Sn` (при этом `S1` является начальным состоянием), а символы входной ленты обозначены как `C1`, `C2`, ..., `Cm`. Кроме того, существует специальный символ `Cend`, обозначающий конец ленты.

На псевдокоде программа, реализующая этот автомат, выглядит так:

```
enum StateType {S1, S2, ...,  Sn};
enum SymbolType {C1, C2, ..., Cm, Cend};

StateType state = StateType.S1;
SymbolType symbol;
try
{
    while((symbol = СчитатьСледующийСимволЛенты()) !=
                    SymbolType.Cend)
    {
        switch(state)
```

```
        {
         case StateType.S1:
              обработка правил вида (S1, symbol -> Sk)
              если правило не найдено, throw new Exception();
         case StateType.S2:
              обработка правил вида (S2, symbol -> Sk)
              если правило не найдено, throw new Exception();
            ...
         case StateType.Sn:
              обработка правил вида (Sn, symbol -> Sk)
              если правило не найдено, throw new Exception();
         }
    }

    предпринимаем какие-то действия в зависимости от того,
    является ли state допускающим состоянием
}
catch(Exception)
{
    вывод сообщения об ошибке «недопустимый входной символ»
}
```

Осталось разобраться подробнее с тем, что подразумевается под «обработкой правил». Поскольку каждое правило всего лишь указывает, в какое состояние должен перейти автомат в той или иной ситуации, наша задача сводится к простому изменению переменной `state`. Допустим, в рассматриваемом автомате для состояния `S1` заданы правила: `S1,C2 → S1`; `S1,C5 → S8` и `S1,C3 → S2`. Тогда фрагмент, их обрабатывающий, будет иметь вид:

```
switch(symbol)
{
    case SymbolType.C2: state = StateType.S1;
                        break;
    case SymbolType.C5: state = StateType.S8;
                        break;
    case SymbolType.C3: state = StateType.S2;
                        break;
    default: throw new Exception();
}
```

Разумеется, не стоит воспринимать эти наброски как неоспоримое, единственно возможное руководство к действию в каждом возможном случае. Например, в листинге 2.1, реализующем все тот же автомат с рис. 2.4, я немного изменил управляющие конструкции, чтобы сократить код.

Листинг 2.1. Реализация автомата, распознающего корректные адреса электронной почты

```
static bool InMainRange(char c) // в диапазоне [a-zA-Z0-9_-]
{
    return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') ||
           (c >= '0' && c <= '9') || (c == '_' || c == '-');
}

static void Main(string[] args)
{
    string s = Console.ReadLine() + '\0';
                            // символ '\0' - признак конца
    char state = '1';       // начальное состояние
    int i = 0;              // индекс текущего символа
    char symbol;            // текущий символ

    try
    {
        while((symbol = s[i++]) != '\0')
                            // пока «лента» не кончилась
        {
            switch(state)   // обычная реализация
                            // конечного автомата
            {
               case '1': if(InMainRange(symbol) || symbol == '.')
                               state = '2';
                         else
                               throw new Exception();
                         break;

                case '2': if(InMainRange(symbol) || symbol == '.')
                               state = '2';
                         else if(symbol == '@')
                               state = '3';
                         else
                               throw new Exception();
                         break;

                case '3': if(InMainRange(symbol))
                               state = '4';
                         else
                               throw new Exception();
                         break;
```

```
                case '4': if(InMainRange(symbol))
                              state = '4';
                          else if(symbol == '.')
                              state = '5';
                          else
                              throw new Exception();
                          break;

                case '5': if(InMainRange(symbol))
                              state = 'F';
                          else
                              throw new Exception();
                          break;

                case 'F': if(InMainRange(symbol))
                              state = 'F';
                          else if(symbol == '.')
                              state = '5';
                          else
                              throw new Exception();
                          break;
            }
        }
        // печатаем True или False в зависимости типа
        // состояния state
        Console.WriteLine(state == 'F');
    }
    catch(Exception)    // «правило не найдено»
    {
        Console.WriteLine("обнаружен недопустимый символ");
    }
}
```

На что в листинге стоит обратить внимание? Во-первых, поскольку символы входной ленты, равно как и метки состояний автомата, могут быть записаны в переменные типа `char`, создавать типы данных `StateType` `SymbolType` незачем. Во-вторых, использование оператора `if` в паре функцией `InMainRange()` позволило сэкономить немало места. Если все делать «по науке», то обработка правил, относящихся, к примеру, к первому состоянию, выглядела бы так:

```
switch(symbol)
{
    case 'a': state = '2'; break;
```

```
        case 'b': state = '2'; break;

        case 'z': state = '2'; break;

        case 'A': state = '2'; break;
        case 'B': state = '2'; break;

        case 'Z': state = '2'; break;

        case '0': state = '2'; break;
        case '1': state = '2'; break;

        case '9': state = '2'; break;

        case '-': state = '2'; break;
        case '_': state = '2'; break;
        case '.': state = '2'; break;

        default:  throw new Exception();
    }
```

Теперь можно сказать пару слов об эффективности автомата, реализованного таким образом. Поскольку подавляющую часть кода составляют элементы конструкции `switch...case`, скорость работы программы зависит, в первую очередь, от качественной компиляции этой конструкции.

К счастью, большинство современных компиляторов, встретив инструкцию `switch`, генерируют так называемую таблицу переходов[14] (jump table). С ее помощью определение выполняемого `case`-блока происходит практически мгновенно и, главное, за фиксированный, не зависящий от общего количества `case`-блоков интервал времени.

[14] Ничего общего с таблицей правил конечного автомата не имеющую.

Таким образом, время работы программы реально оказывается пропорциональным длине входной строки, то есть приведенный алгоритм имеет *линейную сложность*.

Еще немного теории

Чтобы несколько размытое понятие детерминированного конечного автомата приобрело надлежащую форму, рассмотрим его на более строгом уровне[15].

Итак, в объекте, который мы называем детерминированным конечным автоматом, можно выделить элементы:

- Конечное множество состояний $Q = \{q_1, q_2, ..., q_n\}$.
- Конечное множество символов входной ленты $\Sigma = \{a_1, a_2, ..., a_m\}$.
- Начальное состояние q_s (просто некоторый элемент множества Q, выделяемый особо).
- Множество допускающих состояний $F = \{f_1, f_2, ..., f_k\}$. Элементы множества F, как уже говорилось ранее, — обычные состояния конечного автомата, выделенные в самостоятельный набор (таким образом, F есть подмножество множества Q).
- Функция переходов $\delta(q_1, a) = q_2$, сопоставляющая (в теории — любой) паре вида (состояние, символ) некоторое новое состояние. Таким образом, функция δ формирует хорошо известную нам таблицу переходов.

В книгах часто можно встретить определение автомата в виде списка пяти рассмотренных элементов. Например: $A = (Q, \Sigma, \delta, q_s, F)$. Каким именно образом такой автомат функционирует, мы уже обсуждали.

Минимизация детерминированного конечного автомата

Длинная и неэффективная программа вполне может выполнять ту же самую работу, что и другая — короткая и скоростная. Точно так же не исключено, что два различных автомата распознают один и тот же язык. А «если нет разницы, зачем платить больше?» Зачем использовать громоздкий, сложный автомат вместо маленького и простого?

Прежде чем обсуждать эту тему дальше, справедливости ради стоит отметить одно различие между «некоторой данной программой» и программной реализацией детерминированного конечного автомата. Дело в том, что две программы, реализующие одну и ту же функциональность, могут иметь совершенно различные характеристики как по скорости

[5] Зачем нужен «более строгий уровень» в этой книге? Хотя бы для того, чтобы у вас не было проблем со стандартными обозначениями, используемыми в литературе по теории автоматов.

работы, так и по требуемым ресурсам. Автоматы же всегда затрачивают на работу время, пропорциональное длине входной ленты, независимо от сложности своего устройства. Поэтому единственным параметром, отличающим более предпочтительный автомат от менее предпочтительного, является количество состояний. Понятно, что автомат с меньшим числом состояний требует меньшего объема памяти, да и устроен он обычно проще.

По какой причине некоторый рассматриваемый нами автомат может оказаться неоптимальным (то есть содержащим больше состояний, чем необходимо для распознавания интересующего языка)? Во-первых, разрабатывая автомат с помощью карандаша и бумаги, вы естественным образом думаете о распознаваемом языке, а не о минимизации состояний. Во-вторых, нередки случаи (и мы с ними познакомимся), когда конечный автомат создается, простите за тавтологию, автоматически некоторой компьютерной программой. При этом полученное устройство почти всегда будет очень далеким от оптимального[16].

К счастью, существует алгоритм, позволяющий получить оптимальный автомат, эквивалентный данному (то есть выполняющий ту же самую работу, но имеющий при этом минимально возможное количество состояний). Рассмотрим этот полезный алгоритм подробно.

Прежде всего нам потребуется понятие *эквивалентности состояний*. Два состояния называются эквивалентными (не вдаваясь в математические подробности), если дальнейшее поведение автомата в первом состоянии совпадает с дальнейшим поведением автомата во втором состоянии. Сказанное иллюстрирует рис. 2.5.

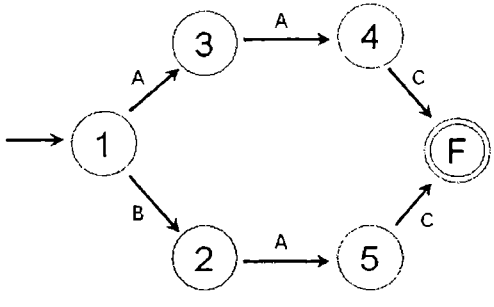


Рис. 2.5. *Эквивалентность состояний*

[16] Заметьте, здесь речь идет не о машинной генерации текста программы, реализующего уже разработанный автомат, а о проектировании компьютером самой конфигурации автомата.

Состояния 2 и 3 эквивалентны. Не имеет значения, в каком из них находится автомат в данный момент времени — любые входные данные ленты будут обработаны одинаковым образом.

Суть минимизации состоит в объединении эквивалентных состояний в единственное новое (конечно, при этом придется подправить и таблицу переходов). На псевдокоде алгоритм минимизации выглядит так:

```
удалить недостижимые состояния и связанные с ними правила;

создать таблицу всевозможных пар состояний вида (p, q);

ОТМЕТИТЬ те пары, где одно из состояний является допускающим,
а другое  —  нет;

DO
    found = false;
    ЕСЛИ существует неотмеченная пара (p, q), такая,
    что для некоторого элемента входного алфавита a
    пара (δ(p, a), δ(q, a)) отмечена
        ОТМЕТИТЬ пару (p, q);
        found = true;
WHILE found  // то есть пока изменения происходят

заменить каждое множество эквивалентных друг другу состояний
на единственное новое; соответствующим образом изменить
таблицу переходов;
```

качестве комментариев к этому алгоритму нужно сказать следующее:

- В автомате могут существовать так называемые *недостижимые* состояния, то есть состояния, которые не могут быть достигнуты никакой последовательностью символов входной ленты (см. рис. 2.6). Разумеется, никто не будет вносить в автомат подобную бессмыслицу, рисуя его вручную. Но при машинной генерации автомата недостижимые состояния вполне могут появиться. Алгоритм минимизации, очевидно, должен уничтожать как все недостижимые состояния, так и правила, с ними связанные.
- Какое именно действие скрывается за словом «отметить»? Где-то в программе каждой паре состояний должно сопоставляться некоторое булево значение (true = отмечена / false = не отмечена). «Отметить» означает установить это значение в true.
- Алгоритм отмечает те или иные пары в соответствии со вполне определенным критерием: элементы отмечаемой пары не должны быть эквивалентными состояниями. Итоговая цель цикла DO... WHILE — отметить все такие пары. После этого должно быть понят-

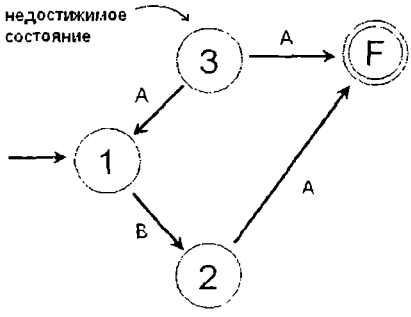


Рис. 2.6. *Пример недостижимого состояния в конечном автомате*

но, как именно работает алгоритм. Поскольку никакое допускающее состояние не может быть эквивалентно какому-либо обычному состоянию, мы можем сразу же отметить пары, состоящие из обычного и допускающего состояний. Далее, предположим, что в соответствии с таблицей правил автомат переходит из некоторого состояния p в состояние p′ по входному символу a (то есть существует правило (p, a) → p′). Допустим, существует также аналогичное правило для состояния q: (q, a) → q′. Тогда если состояния p′ и q′ не являются эквивалентными, состояния p и q также заведомо неэквивалентны. Так как записи δ(p, a) и δ(q, a) обозначают правые части правил (p, a) → p′ и (q, a) → q′, условный оператор, приведенный в псевдокоде, претворяет эту идею в жизнь.

- Предположим, алгоритм нашел некоторую неотмеченную пару состояний (p, q). Теперь его задача — перебирать все возможные символы входной ленты в поисках отмеченной пары (δ(p, a), δ(q, a)). А что будет, если для текущего рассматриваемого символа соответствующего правила не предусмотрено? Выше мы уже обсуждали подобную ситуацию. Напомню, реализация автомата на уровне программы — нечастый случай, когда отсутствие того или иного правила довольно легко сходит с рук (можно просто сообщить об ошибке). Алгоритмы, работающие с автоматами, нередко предполагают, что правило вида (p, a) → q существует для любых допустимых p и a. На практике это несколько (хотя и отнюдь не фатально) усложняет жизнь. Следует лишь ввести новое недопускающее состояние в автомат и направить в него все до этого момента не предусмотренные переходы. Предположим, что допустимыми символами ленты для автомата с рис. 2.5 являются латинские буквы A, B и C. Тогда перед минимизацией его придется модифицировать, как показано на рис. 2.7.
- После выполнения цикла DO...WHILE в нашем распоряжении оказывается список пар эквивалентных друг другу состояний. Но как заменить каждый *класс эквивалентности* (то есть множество эквивалентных друг другу состояний) единственным состоянием? Если

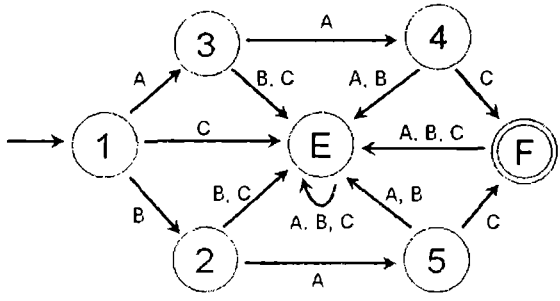


Рис. 2.7. *Автомат с полным набором правил*

состояния обозначаются целыми числами (предположим это), выявить классы эквивалентности поможет следующий код:

```
ЦИКЛ по всем состояниям q
    e_class[q] = q;
ЦИКЛ по всем неотмеченным парам (p, q)
   ЦИКЛ по всем состояниям s
      ЕСЛИ e_class[s] == p, ТО e_class[s] = q
```

Изначально каждое состояние представляет свой собственный класс эквивалентности (`e_class[q] = q`). Затем, если найдена какая-либо пара эквивалентных состояний (`p`, `q`), все представители класса состояния `p` «переписываются» в класс состояния `q`. В итоге для любого состояния соответствующее значение массива `e_class` будет равно номеру класса эквивалентности, к которому это состояние относится.

К сожалению, алгоритмы теории автоматов — не самое увлекательное чтение. Однако пройти мимо них просто нельзя; думаю, никого не надо убеждать в том, что, к примеру, минимизация автомата — вещь на редкость полезная.

Поскольку эта книга ориентирована скорее на практиков, мне хочется довести большинство важных алгоритмов до реально работающего кода. С другой стороны, загромождать книгу листингами нежелательно, поэтому реализация будет несколько ограниченной.

Во-первых, мы будем предполагать, что все состояния автомата обозначаются целыми, подряд идущими числами, начиная с единицы. Во-вторых, допустимыми элементами входной ленты считаются только стандартные символы из таблицы ASCII. В-третьих, в целях экономии места мы не будем заниматься удалением недостижимых состояний. В общих чертах удаление недостижимых состояний производится достаточно просто. Для этого надо завести очередь и сразу же добавить в нее стартовое состояние (которое, естественно, является достижимым). Далее выполняется алгоритм:

```
ПОКА очередь непуста
    извлечь очередной элемент S;

    пометить его как достижимый;

    внести в очередь все состояния, в которые существует
    прямой переход из S;
```

После выполнения цикла все недостижимые состояния останутся непомеченными; их и все связанные с ними правила следует удалить.

Программа минимизации читает входные данные (конфигурацию автомата) с консоли в таком формате:

```
количество состояний

список символов ленты

список допускающих состояний

стартовое состояние

правило1
правило2
...
правилоN
```

Таким образом, автомату, изображенному на рис. 2.7, соответствует ввод

```
7
A B C
6
1
1 A 3
1 B 2
1 C 7
2 A 5
2 B 7
2 C 7
3 A 4
3 B 7
3 C 7
4 A 7
4 B 7
4 C 6
5 A 7
```

```
5 B 7
5 C 6
6 A 7
6 B 7
6 C 7
7 A 7
7 B 7
7 C 7
```

Поскольку все состояния должны быть обозначены числами, я заменил состояние F номером 6, а состояние E — номером 7.

Полный текст программы минимизации автомата приведен в листинге 2.2.

Листинг 2.2. Минимизация детерминированного конечного автомата

```
struct Pair      // пара состояний (p, q)
{
    public int p, q;

    public Pair(int thep, int theq) {p = thep; q = theq;}
}

struct Leftside // левая часть правила - (состояние, символ)
{
    public int state;
    public char symbol;

    public Leftside(int st, char sym) {state = st; symbol = sym;}
}

static void Main(string[] args)
{
    // количество состояний
    int Nstates = Convert.ToInt32(Console.In.ReadLine());

    string[] symstr = (Console.In.ReadLine()).Split(' ');
    char[] symbols = new char[symstr.Length];
    for(int i = 0; i < symstr.Length; i++)
    // символы входной ленты
        symbols[i] = Convert.ToChar(symstr[i]);

    // список допускающих состояний
    string[] Flist = (Console.In.ReadLine()).Split(' ');
```

```
bool[] favorable = new bool[Nstates + 1];
for(int i = 1; i <= Nstates; i++)
    favorable[i] = false;
foreach(string id in Flist)
    favorable[Convert.ToInt32(id)] = true;

// стартовое состояние
int StartingState = Convert.ToInt32(Console.In.ReadLine());

// правила хранятся как (левая часть, состояние)
Hashtable rules = new Hashtable();
Hashtable pairs = new Hashtable();
                          // список пар состояний
string s;
while((s = Console.In.ReadLine()) != "")
                          // считывание правил
{
    string[] tr = s.Split(' ');
    rules.Add(new Leftside(Convert.ToInt32(tr[0]),
              Convert.ToChar(tr[1])),
              Convert.ToInt32(tr[2]));
}

// ОТМЕТИТЬ те пары, где одно из состояний является
// допускающим,а другое - нет
for(int i = 1; i <= Nstates; i++)
    for(int j = 1; j <= Nstates; j++)
    {
        bool Marked = ((favorable[i] == true &&
                        favorable[j] == false) ||
                       (favorable[i] == false &&
                        favorable[j] == true));
        pairs.Add(new Pair(i, j), Marked);
     }

bool found;
do
{
    found = false;
    for(IDictionaryEnumerator e = pairs.GetEnumerator();
                                  e.MoveNext(); )
    {
     if((bool)e.Value == false)
                       // найдена неотмеченная пара (p, q)
```

```
      foreach(char a in symbols)
      {
         int d1 = (int)rules[new Leftside(((Pair)e.Key).p, a)];
         int d2 = (int)rules[new Leftside(((Pair)e.Key).q, a)];

            // найдена отмеченная пара (d(p, a), d(q, a))
            if((bool)pairs[new Pair(d1, d2)] == true)
            {
             pairs[new Pair(((Pair)e.Key).p,
                          ((Pair)e.Key).q)] = true;
             found = true;
             goto exit;
             }
      }
      }
      exit: ;
}
while(found);

// выявление классов эквивалентности
int[] e_class = new int[Nstates + 1];
for(int i = 1; i <= Nstates; i++)
    e_class[i] = i;

for(IDictionaryEnumerator e = pairs.GetEnumerator();
                              e.MoveNext();)
    if((bool)e.Value == false)
        for(int i = 1; i <= Nstates; i++)
            if(e_class[i] == ((Pair)e.Key).p)
                e_class[i] = ((Pair)e.Key).q;

bool[] StatePrinted = new bool[Nstates + 1];
    for(int i = 1; i <= Nstates; i++)
        StatePrinted[i] = false;

Console.Write("Состояния: ");     // вывод состояний
for(int state = 1; state <= Nstates; state++)
     if(!StatePrinted[e_class[state]])
                                  // если состояние еще не
     {                             // выведено на печать
        Console.Write("{0} ", e_class[state]);
        StatePrinted[e_class[state]] = true;
     }
Console.WriteLine();              // вывод особых состояний
```

```
    Console.WriteLine("Стартовое состояние: {0}\n" +
                      "Допускающее состояние: {1}",
                      e_class[StartingState],
                      e_class[Convert.ToInt32(Flist[0])]);
    Console.WriteLine("Правила:");   // вывод правил
    Hashtable RulePrinted = new Hashtable();
    for(IDictionaryEnumerator e = rules.GetEnumerator();
                                  e.MoveNext();)
    {
      string rule = "(" + e_class[((Leftside)e.Key).state].ToString() +
                    ", " + ((Leftside)e.Key).symbol.ToString() +
                    ") -> " + e_class[(int)e.Value].ToString();
       if(RulePrinted.ContainsKey(rule) ==  false)
                                    // если правило еще не
        {                            // выведено на печать
           Console.WriteLine(rule);
           RulePrinted[rule] = true;
        }
    }
}
```

Результирующий автомат для приведенного примера, построенный программой минимизации, содержит всего пять состояний:

```
Состояния: 1 3 5 6 7
Стартовое состояние: 1
Допускающее состояние: 6
Правила:
(7, C) -> 7
(7, B) -> 7
(7, A) -> 7
(6, C) -> 7
(5, C) -> 6
(6, B) -> 7
(5, B) -> 7
(6, A) -> 7
(5, A) -> 7
(3, C) -> 7
(3, B) -> 7
(3, A) -> 5
(1, C) -> 7
(1, B) -> 3
(1, A) -> 3
```

Граф этого автомата приведен на рис. 2.8.

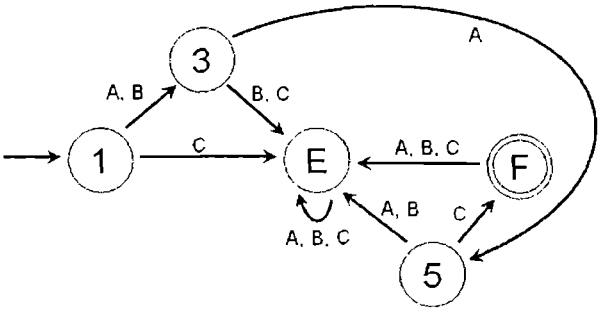


Рис. 2.8. *Минимизированный автомат*

2.2. Недетерминированные конечные автоматы

Добавим немного магии

Детерминированный конечный автомат — подходящий инструмент для решения задач вроде распознавания корректных адресов электронной почты. Казалось бы, если в процессе работы встретилась «автоматная» на вид задача, дело за малым: берем карандаш и бумагу и за пять минут изобретаем автомат, ее решающий. А дальше уж дело техники.

На самом же деле у детерминированного автомата (а если быть до конца честным, не у него, а у нас) есть одна проблема: порою разработать подходящую конструкцию для решения данной (вроде бы несложной) задачи оказывается весьма трудно. После пяти минут раздумий на бумаге обнаруживается штук десять-пятнадцать состояний, весьма причудливым образом между собой соединенных. Нить рассуждений безнадежно потеряна, и все начинается сначала. Шансы на успех при этом отнюдь не возрастают.

Переход к недетерминированным конечным автоматам не позволит нам решать более сложные задачи, но сам процесс поиска решения может заметно упростить. Итак, что же такое *недетерминированный конечный автомат* (nondeterministic finite automaton)? Если в двух словах, это устройство, очень похожее на детерминированный конечный автомат, но теперь противоречащие друг другу правила разрешены. К примеру, у вас могут одновременно существовать правила 5, A → 3, 5, A → 7 и 5, A → 1. Вопрос в том, как такой автомат может функционировать.

В 1957 году физик Хью Эверетт (Hugh Everett) выдвинул идею о том, что любое событие разбивает наш мир на несколько новых миров, в каждом из которых это событие закончилось по-своему. Например, в одном из миров,

параллельных нашему, сборная России все-таки стала чемпионом мира по футболу (хотя в это и очень трудно поверить). В другом — Гитлер выиграл вторую мировую войну. В третьем — Ньютон вместо физики занялся бизнесом, и открытие законов классической механики пришлось отложить лет на пятьдесят. Миры появляются даже в результате менее значительных событий: например, когда вы, гуляя по улице, на очередном перекрестке сворачиваете налево, ваш параллельный двойник в только что родившемся мире решает повернуть направо.

Я не берусь обсуждать здесь теорию Эверетта (хотя футбольный пример свидетельствует явно не в ее пользу), но в мире недетерминированных конечных автоматов она оказывается очень удобной моделью.

Допустим, наш автомат столкнулся с ситуацией, когда три правила (5, A → 3; 5, A → 7; 5, A → 1) одновременно требуют перейти в три разных состояния. В «обычном» мире это было бы проблемой, но для «мультимира» подобное положение вполне в порядке вещей: мир, в котором существовал конечный автомат, расщепляется на три новых мира, в каждом из которых было принято свое решение. В первом мире автомат теперь находится в состоянии 3, во втором мире — в состоянии 7, а в третьем — в состоянии 1.

Чтобы сделать какие-то выводы из работы автомата, следует рассмотреть все «миры», то есть изучить любые варианты выбора конфликтующих правил на каждом шаге. После того, как вся входная лента будет считана, в игру вступает новое соглашение: мы полагаем, что недетерминированный конечный автомат допускает данную строку ленты, если он завершил работу в допускающем состоянии *хотя бы в одном* из множества «миров». Соответственно, автомат отвергает строку, если он завершил работу в недопускающем состоянии *в каждом* «мире».

Пожалуй, пора доказать полезность недетерминированного поведения на практике. Попробуйте, к примеру, решить такую задачу:

Сконструировать детерминированный конечный автомат, допускающий все строки, содержащие подстроку ABC. Входным алфавитом считать символы A, B и C.

То есть ваш автомат должен допускать строки типа ABAABCAA или AABCCC, отвергая при этом строки AABBCC и ABBBACC. Фраза «попробуйте решить такую задачу» означает именно то, что она означает — отложите ненадолго книгу и попытайтесь нарисовать требуемый автомат на бумаге. Это относительно несложно; вам вполне хватит четырех состояний.

К детерминированному автомату мы еще вернемся, а пока рассмотрим решение, полученное при использовании автомата недетерминированного (см. рис. 2.9).

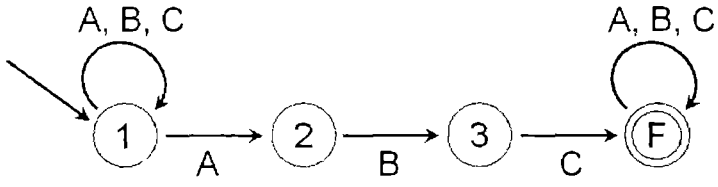


Рис. 2.9. *Решение задачи при помощи недетерминированного конечного автомата*

Возьмем в качестве примеров строки AABBCC и AABCCC. Процесс работы автомата в обоих случаях показан на рис. 2.10.

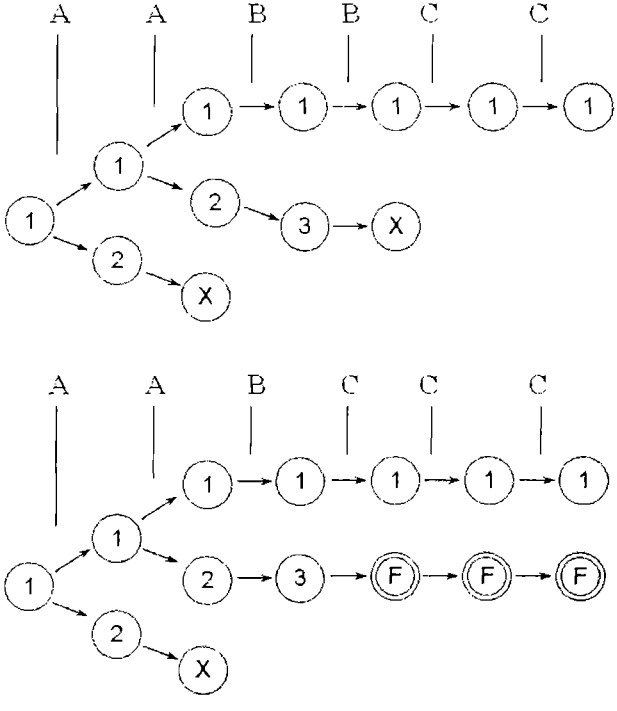


Рис. 2.10. *Обработка строк AABBCC и AABCCC автоматом*

Разбирая строку AABBCC, автомат уже на первом шаге сталкивается с необходимостью выбора: перейти во второе состояние (правило 1, A → 2) или остаться в первом (1, A → 1). Поскольку мы условились действовать по правилам «мультимира», придется рассмотреть оба варианта (соответствующих двум новым «мирам»).

Автомат из первого остается в начальном состоянии (что влечет за собой новое «расщепление» на втором шаге), в то время как автомат из второго

мира переходит в состояние номер 2, и очередной символ входной ленты (им оказывается A) вызывает нештатную ситуацию. Впрочем, другие варианты поведения на поверку оказываются ничуть не лучше: автомат либо так и остается в своем первоначальном состоянии, либо происходит нештатная ситуация при считывании символа B в состоянии номер 3.

Аналогичным образом анализируя поведение автомата при входной строке AABCCC, можно прийти к выводу, что выбор правила 1, A → 1 на первом шаге и правила 1, A → 2 на шаге втором приводит нас прямо к цели (распознаванию строки).

Возможно, вы уже заметили, что строить полный граф возможных действий обычно нет нужды. Если наша цель — показать, что данная строка может быть допущена автоматом, достаточно (во многом интуитивно) отыскать любой маршрут, ведущий к допускающему состоянию. Остальные ветви графа при этом совершенно несущественны.

Разумеется, если стоит обратная задача — доказать, что строка не допускается автоматом, ситуация заметно усложняется (придется либо рассмотреть все варианты, либо построить обратный, «инвертированный» автомат — но об этом позже).

От магии — к реальному миру

Недетерминированный конечный автомат — инструмент, конечно, замечательный. Смущает лишь одна мелочь: сымитировать его на реальном компьютере не удастся, поскольку компьютер «живет» в обычном, нерасщепляющемся мире.

С другой стороны, дела обстоят не так плохо, как могли бы. Мы с вами, находясь в одном и том же мире (даже если миры расщепляются, мы-то в каждый момент времени осознаем лишь один из них), все-таки сумели проанализировать работу недетерминированного автомата, построив граф его возможных действий. Почему бы не доверить эту работу компьютеру?

В действительности существует еще более красивое решение. Любой недетерминированный конечный автомат можно преобразовать в автомат детерминированный (*детерминизировать*), выполняющий ту же самую работу (или, говоря более формально, распознающий тот же самый язык). Перед тем, как заняться алгоритмом детерминизации, рассмотрим полезное расширение недетерминированного конечного автомата под названием ε-автомат.

ε-автомат

Представьте себе такую ситуацию. У вас есть автомат, распознающий некоторый язык (например, корректные адреса электронной почты). У

вас также есть автомат, распознающий какой-то другой язык (например, все четные целые числа). Первый автомат (A) в силу каких-то конструктивных особенностей имеет несколько допускающих состояний (в этом же нет ничего плохого), а второй (B) — одно. Схематически ситуация изображена на рис. 2.11.

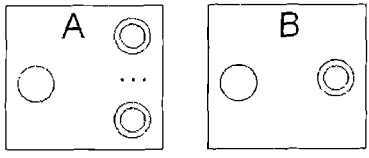


Рис. 2.11. *Схематическое изображение автоматов задачи*

Задача состоит в том, чтобы разработать автомат, допускающий строки, состоящие из двух подряд идущих адресов электронной почты, за которыми следует четное целое число[17]. Создавать такой автомат «с нуля», конечно, совсем нет желания, но возможно ли грамотно соединить существующие автоматы между собой, чтобы получить решение задачи?

Сложность в том, что любое «соединение» (по сути дела переход) должно быть помечено каким-либо символом входного алфавита, но наши автоматы уже отлажены, и после того, как предыдущий автомат цепочки отработал, следующий должен запускаться безо всяких промежуточных действий.

Имея автоматы с единственным допускающим состоянием, еще можно исхитриться, объединив допускающее состояние предыдущего автомата с начальным состоянием следующего, но множественные допускающие состояния делают такое простое решение невозможным.

Выход заключается в введении понятия ε-*переходов* (см. рис. 2.12, итоговое допускающее состояние отмечено двумя внутренними кольцами).

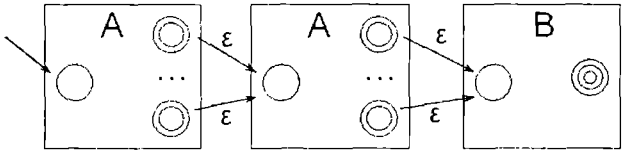


Рис. 2.12. *Соединение автоматов при помощи ε-переходов*

[17] Понимаю, пример несколько надуманный, но сейчас не хочется отвлекаться на серьезные задачи.

ющее состояние — пятое. Поэтому допускающими состояниями нового автомата будут состояния {5}, {1, 5}, {2, 5}, {3, 5}, {4, 5}, {1, 2, 5}, {1, 3, 5}, {1, 4, 5}, {2, 3, 5}, {2, 4, 5}, {3, 4, 5}, {1, 2, 3, 5}, {1, 2, 4, 5}, {1, 3, 4, 5}, {2, 3, 4, 5} и {1, 2, 3, 4, 5}.

Теперь можно описать на псевдокоде дальнейшие шаги алгоритма детерминизации.

```
для каждого состояния {s1, ..., sn} создаваемого автомата
    для каждого символа входного алфавита c
        R = {};
        для каждого элемента s_i
            R = R ∪ P'(s_i, c)[19];
        добавить в новый автомат правило {s_1, ..., s_n}, c →R;
оптимизировать (минимизировать) полученный автомат;
```

Таким образом, ядро алгоритма по сути дела отвечает на следующий вопрос. Дано состояние $\{s_1, ..., s_n\}$. В какие состояния можно попасть по символу c входной ленты из состояний исходного автомата $s_1, ..., s_n$? Множество полученных состояний и будет определять единственное целевое состояние R создаваемого автомата, участвующее в правиле $\{s_1, ..., s_n\}$, c → R.

Рассмотрим пару примеров для автомата, изображенного на рис. 2.13 (полное исследование каждого из $2^5 - 1 = 31$ состояний способно навести тоску на кого угодно).

Возьмем состояние {1, 3} и входной символ A. Так как P'(1, A) = {1, 2, 3, 4, 5}, а P'(3, A) = {5}, результирующее состояние R = P'(1, A) ∪ P'(3, A) = {1, 2, 3, 4, 5}. Таким образом получаем правило: {1, 3}, A → {1, 2, 3, 4, 5}.

Теперь разберем ситуацию с состоянием {2, 3, 5} и входным символом C. Поскольку P'(2, C) = {}, P'(3, C) = {1, 2}, а P'(5, C) = {}, получаем R = P'(2, C) ∪ P'(3, C) ∪ P'(5, C) = {1, 2}. В новый автомат добавляется правило {2, 3, 5}, C → {1, 2}.

[19] Как уже упоминалось выше, если автомат не содержит ε-переходов, вместо множества P' можно использовать множество P (его проще получить).

Детерминизация недетерминированного автомата (практика)

Как известно, бумажный тигр — не пара настоящему, поэтому давайте доведем алгоритм детерминизации до воплощения в программном коде. Как и в случае с алгоритмом минимизации, придется пойти на некоторые уступки. Мы снова будем считать символами входной ленты стандартные ASCII-символы, а метками состояний — подряд идущие целые числа. Кроме того, обработка ε-переходов останется за пределами реализации (считайте детерминизацию ε-автомата самостоятельным упражнением).

Алгоритм детерминизации проще пояснить по частям.

Листинг 2.3. Детерминизация конечного автомата (вспомогательные функции)

```
static ArrayList NewStates;    // новые состояния

static string MakeStateName(string state)
                               // создать имя состояния
{                              // по двоичной записи
    string result = "{";

    for(int i = 0; i < state.Length; i++)
                               // единица превращается
        if(state[i] == '1')    // в номер состояния
            result += Convert.ToString(i + 1) + " ";
        result += "}";

    return result;
}

static void GenerateNewStates(int depth, string s)
                               // сгенерировать
{                              // набор состояний
    if(depth == 0)             // нового автомата
        NewStates.Add(MakeStateName(s));
    else
    {
        GenerateNewStates(depth - 1, "0" + s);
        GenerateNewStates(depth - 1, "1" + s);
    }
}
```

Как уже отмечалось, детерминизация начинается с процесса создания списка состояний создаваемого автомата. Если исходный автомат имел

состояния, помеченные как 1, 2, ..., N, каковы будут состояния нового автомата? Иными словами, как получить все подмножества множества {1, 2, ..., N}?[20] Здесь используется такой алгоритм. Представьте себе строку из N нулей и единиц. Единица означает «включить данный элемент в подмножество», нуль — «не включать». Примеры для N = 8 показаны ниже.

```
исходное множество: {1, 2, 3, 4, 5, 6, 7, 8}
строка 1:  0  0  1  1  0  0  1  0   (подмножество {3, 4, 7})
строка 2:  1  1  0  0  1  0  0  1   (подмножество {1, 2, 5, 8})
строка 3:  0  0  0  0  0  1  1  0   (подмножество {6, 7})
```

Сгенерировать все возможные строки из нулей и единиц фактически и означает сгенерировать интересующие нас подмножества. Рекурсивная функция `GenerateNewStates()` создает имена новых состояний в два этапа. Сначала генерируется уникальная строчка из нулей и единиц, а затем она преобразуется в запись в виде подмножества при помощи вызова `MakeStateName()`. Первоначально требуется вызвать `GenerateNewStates()` с параметром `depth`, равным `N` и параметром `s`, равным пустой строке.

Листинг 2.4. Детерминизация конечного автомата (анализ входных данных)

```
// структуры, взятые из алгоритма минимизации
struct Pair {...}          // пара состояний
struct Leftside {...}      // левая часть правила -
                           // (состояние, символ)

static void Main(string[] args)
{
    int Nstates = Convert.ToInt32(Console.In.ReadLine());
                           // кол-во состояний
    NewStates = new ArrayList();
    GenerateNewStates(Nstates, "");
                           // создать набор новых состояний
    NewStates.RemoveAt(0); // стираем состояние, помеченное
                           // пустым множеством
                           // (оно  -  первое в списке)
    // взято из программы минимизации автомата
    string[] symstr = (Console.In.ReadLine()).Split(' ');
    char[] symbols = new char[symstr.Length];
    for(int i = 0; i < symstr.Length; i++)
```

[20] Классическая задачка для старшеклассников или первокурсников.

```
        symbols[i] = Convert.ToChar(symstr[i]);
                            // символы входной ленты

    // список допускающих состояний
    string[] Flist = (Console.In.ReadLine()).Split(' ');
    bool[] favorable = new bool[Nstates + 1];
    for(int i = 1; i <= Nstates; i++)
        favorable[i] = false;
    foreach(string id in Flist)
        favorable[Convert.ToInt32(id)] = true;

    // стартовое состояние
    int StartingState =
      Convert.ToInt32(Console.In.ReadLine());
    // процедура считывания правил модифицирована
    // (чтобы недетерминированные переходы
    // обрабатывались корректно)
    Hashtable rules = new Hashtable();
    string s;
    while((s = Console.In.ReadLine()) != "")
                            // считывание правил
    {
        string[] tr = s.Split(' ');
        Leftside ls = new Leftside(Convert.ToInt32(tr[0]),
                                   Convert.ToChar(tr[1]));
        if(!rules.ContainsKey(ls))
            rules.Add(ls, new SortedList());

         // правая часть правила - множество состояний,
         // куда ведет переход
         ((SortedList)rules[ls]).Add(Convert.ToInt32(tr[2]), null);
    }

    // далее идет алгоритм детерминизации
    // ...
}
```

Листинг 2.4 практически полностью повторяет процедуру считывания исходного автомата в алгоритме минимизации. Единственное существенное различие состоит в считывании правил. Множество недетерминированных правил вида $(s, c) \to d_1$, $(s, c) \to d_2$, ..., $(s, c) \to d_n$ в программе можно записать в виде одного-единственного правила, правая часть которого представляет собой список состояний $d_1, d_2, \ldots, d_n$: $(s, c) \to (d_1, d_2, \ldots, d_n)$.

Список реализуется при помощи класса `SortedList` (обратите внимание, что в таком списке хранятся пары (ключ, значение), но нам требуется лишь ключ — значение везде устанавливается равным `null`).

Листинг 2.5. Детерминизация конечного автомата (основная часть)

```
static void Main(string[] args)
{
    // ...
    // анализ исходного автомата завершен

    // вывод стартового состояния
    Console.Write("Стартовое состояние: {{{0}}}\n" +
                  "Допускающие состояния: ", StartingState);

    // вывод допускающих состояний
    for(IEnumerator state = NewStates.GetEnumerator();
                            state.MoveNext();)
        for(int i = 1; i <= Nstates; i++)
            if(favorable[i])
               // если состояние i - допускающее
            {  // любое состояние нового автомата,
               // содержащее i - тоже допускающее
               if(((string)state.Current).IndexOf(" " +
                        Convert.ToString(i) + " ") != -1)
               {
                  Console.Write(state.Current + " ");
                  break;
               }
            }
    Console.WriteLine("\nПравила:");

    // алгоритм детерминизации
    for(IEnumerator state = NewStates.GetEnumerator();
                            state.MoveNext();)
        foreach(char c in symbols)
                  // для каждого состояния state и символа c
        {
            Hashtable R = new Hashtable();

            // вычленяем "элементы" составного
            // состояния {S1, ..., Sn}
            string[] state_elems = ((string)state.Current).Split(' ');
            for(int i = 1; i < state_elems.Length - 1; i++)
```

```
            {
               // для каждого элемента Si
                 int Si = Convert.ToInt32(state_elems[i]);
               // если существует правило с левой частью (Si, c)
               // записываем в R состояния, куда можно
               // попасть из Si по c
                 if(rules.ContainsKey(new Leftside(Si, c)))
                     foreach(int r_state in
                            ((SortedList)rules
                                [new Leftside(Si, c)]).Keys)
                         R[r_state] = 1;
            }

            if(R.Keys.Count > 0)   // Если R непусто
            {
                 // формируем правую часть в виде строки
                 string right_side = "";
                 foreach(int r_state in R.Keys)
                   right_side = Convert.ToString(r_state) +  " " +
                                right_side;
                    right_side = "{" + right_side + "}";

                    // выводим готовое правило на экран
                    Console.WriteLine((string)state.Current +
                             ", " + c + " -> " + right_side);
            }
        }
    }
```

Несколько первых строк листинга (до вывода правил) посвящены печати допускающих состояний. Хотя внешне этот фрагмент выглядит несколько громоздко (два вложенных цикла и два ветвления), суть его проста: если некоторое состояние нового автомата в своей записи содержит хотя бы одно допускающее состояние исходного автомата, оно само становится допускающим.

Дальнейшие действия производятся в соответствии с псевдокодом (я постарался сохранить те же обозначения). Поскольку в .NET не предусмотрен класс для представления множества (очень досадное упущение), множество R хранится в хэш-таблице.

Ключи хэш-таблицы уникальны, поэтому набор ключей (соответствующие им значения при этом не важны) может использоваться для хранения множеств (хотя, конечно, это не самое элегантное решение). Фрагмент программы до строки `if(R.Keys.Count > 0)` формирует множество R в соответствии с псевдокодом (так как ε-переходы не рассматривают-

ся, достаточно проанализировать лишь непосредственно правые части правил). Полученное правило `state, c → R` добавляется в автомат.

Теперь самое время проверить этот алгоритм в деле, например, в задаче детерминизации автомата, изображенного на рис. 2.9. Пометив допускающее состояние числом 4 (в оригинале была буква F), получаем входные данные для программы:

```
4
A B C
4
1
1 A 1
1 B 1
1 C 1
1 A 2
2 B 3
3 C 4
4 A 4
4 B 4
4 C 4
```

В процессе детерминизации создается достаточно много правил (40 штук):

```
Стартовое состояние: {1}
Допускающие состояния: {4} {1 4} {2 4} {1 2 4} {3 4} {1 3 4}
                       {2 3 4} {1 2 3 4}
Правила:
{1}, A -> {1 2}
{1}, B -> {1}
{1}, C -> {1}
{2}, B -> {3}
{1 2}, A -> {1 2}
{1 2}, B -> {1 3}
{1 2}, C -> {1}
{3}, C -> {4}
{1 3}, A -> {1 2}
{1 3}, B -> {1}
{1 3}, C -> {1 4}
{2 3}, B -> {3}
{2 3}, C -> {4}
{1 2 3}, A -> {1 2}
{1 2 3}, B -> {1 3}
{1 2 3}, C -> {1 4}
{4}, A -> {4}
{4}, B -> {4}
```

```
{4}, C -> {4}
(1 4}, A -> {1 2 4}
{1 4}, B -> {1 4}
{1 4}, C -> {1 4}
{2 4}, A -> {4}
{2 4}, B -> {3 4}
{2 4}, C -> {4}
{1 2 4}, A -> {1 2 4}
{1 2 4}, B -> {1 3 4}
{1 2 4}, C -> {1 4}
{3 4}, A -> {4}
{3 4}, B -> {4}
{3 4}, C -> {4}
{1 3 4}, A -> {1 2 4}
{1 3 4}, B -> {1 4}
{1 3 4}, C -> {1 4}
{2 3 4}, A -> {4}
{2 3 4}, B -> {3 4}
{2 3 4}, C -> {4}
{1 2 3 4}, A -> {1 2 4}
{1 2 3 4}, B -> {1 3 4}
{1 2 3 4}, C -> {1 4}
```

У меня нет никакого желания рисовать этот монстрообразный агрегат в книге. Однако если вы проявите немного терпения и набросаете его на бумаге, довольно быстро станет ясно, что лишь шесть его состояний — {1}, {1, 2}, {1, 3}, {1, 4}, {1, 2, 4} и {1, 3, 4} являются достижимыми. Если вычеркнуть недостижимые состояния и правила, связанные с ними, из полученного автомата, а заодно и переименовать состояния в более удобных обозначениях — 1, 2, 3, 4, 5 и 6, мы получим гораздо более простое устройство (см. рис. 2.14).

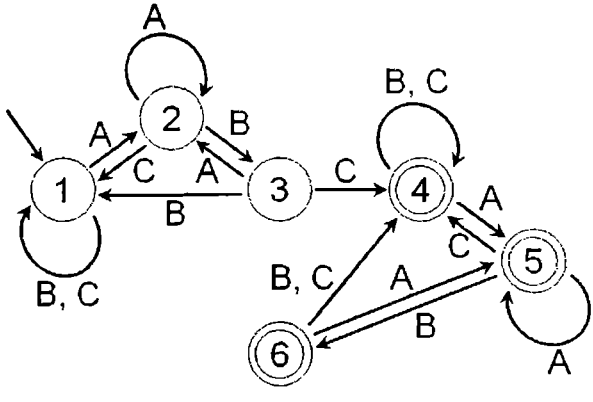


Рис. 2.14. *Детерминизированный автомат без недостижимых состояний*

Этот автомат уже близок к оптимальному, но все еще не оптимален. К счастью, уже рассмотренный нами алгоритм минимизации может его улучшить. Результат минимизации показан на рис. 2.15.

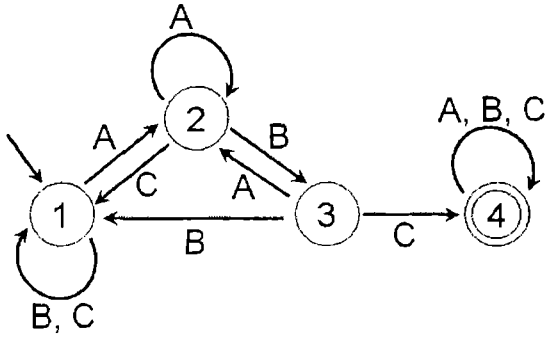


Рис. 2.15. *Детерминизированный и минимизированный автомат*

Итак, полученный автомат решает ту же самую задачу, что и автомат на рис. 2.9. При этом он обладает одним немаловажным преимуществом: его можно непосредственно запрограммировать на компьютере. Хотя этот автомат содержит всего четыре состояния, я думаю, вы согласитесь с тем, что изобрести его «с ходу» совсем не так просто, как эквивалентный ему недетерминированный. Поэтому обычно гораздо проще воспользоваться возможностями недетерминированного поведения при проектировании, а затем автоматически преобразовать созданный автомат в детерминированный.

Полученное в последнем примере устройство содержит столько же состояний, сколько и исходное, но на практике далеко не всегда так везет. Как уже упоминалось, в худшем случае при детерминизации автомата количество состояний может вырасти с N до $2^N - 1$ (то есть весьма и весьма прилично).

Непосредственная имитация недетерминированного автомата

Существует возможность сымитировать недетерминированный автомат на обычном компьютере, не тратя при этом слишком больших объемов памяти. Идея алгоритма достаточно проста. Автомат имитируется примерно так же, как и детерминированный, но в любой ситуации, когда происходит «расщепление миров», компьютер должен выбрать какой-то один мир и продолжить работу в нем. Если после считывания входной строки автомат оказался в недопускающем состоянии, придется вернуться назад к моменту «расщепления» и попробовать пойти по другой ветви.

Мне не хочется слишком сильно углубляться в технические детали этого алгоритма, поскольку в нем нет ничего интересного с точки зрения теории автоматов. Перебор с возвратом — обычная процедура, применяемая во многих ситуациях. К примеру, аналогичным образом можно запрограммировать поиск выхода из лабиринта: находясь в «коридоре» (где можно двигаться лишь вперед и назад), движемся вперед; на встретившейся развилке выбираем некоторый путь и продолжаем движение. Если путь оказался тупиковым, возвращаемся к развилке и выбираем другое направление.

Разумеется, в имитации недетерминированного автомата есть свои тонкости. К примеру, если программа на текущем шаге выбирает ε-переход, считывать очередной символ входной ленты не требуется. Считывание всех символов входной ленты еще не означает завершения работы автомата. Ведущие из текущего состояния ε-переходы в данном случае являются тоже своего рода альтернативами: автомат может либо закончить работу, либо продолжить ее, воспользовавшись ε-переходом.

Более серьезная проблема состоит в «зацикливании» алгоритма. Если из состояния A в состояние B ведет ε-переход, и такой же переход ведет из B в A, программа может запросто переключаться между этими состояниями до бесконечности. Справиться с таким затруднением можно, например, сохраняя на каждом шаге пару (состояние, номер_текущего_символа_на_ленте). Если в какой-то момент окажется, что мы собираемся перейти в состояние 3, считав пятый по счету символ с ленты, но при этом такая ситуация уже возникала раньше, подобный переход следует тут же отсечь.

Согласитесь, не имеет никакого значения, каким образом мы оказались в третьем состоянии после считывания пятого символа ленты. Быть может, это результат пяти обычных переходов, а быть может, пяти обычных и десяти ε-переходов. Та же ситуация возникает и в лабиринтах: алгоритм обхода должен помечать посещенные локации, чтобы не проходить их по второму (третьему и так далее) кругу.

Экономия памяти при непосредственной имитации недетерминированного автомата оборачивается значительным временем выполнения, поскольку программе в худшем случае приходится не раз возвращаться к уже рассмотренным состояниям и принимать иное решение при выборе альтернативы. Переборный алгоритм может вызвать экспоненциальный рост времени работы программы, точно так же как детерминизация экспоненциально увеличивает расход памяти.

Интересно отметить, что детерминизация сама по себе тоже требует экспоненциальных затрат времени (для N состояний недетерминированного автомата придется построить $2^N - 1$ состояний автомата детерминированного). Однако детерминизация выполняется всего лишь один раз, после чего мы получаем быстрый (хотя и громоздкий) автомат.

Формальное определение недетерминированного автомата

Мы практически на одном дыхании пробежались по недетерминированным и ε-автоматам, не затрагивая их формального определения. К счастью, недетерминированное поведение очень мало меняет определение автомата. Единственный пункт, в который придется внести коррективы, касается функции переходов. Напомню, что в детерминированном случае функция переходов сопоставляет паре (состояние, символ) единственное новое состояние: $\delta(q_1, a) = q_2$.

Введение ε-переходов формально означает лишь допустимость символа ε в качестве второго аргумента функции δ(). Недетерминированное поведение меняет результат функции. Вместо одного целевого состояния мы теперь получаем целое множество состояний (в программе детерминизации правила представлялись именно таким образом — левой части правила соответствовал отсортированный список целевых состояний): $\delta(q_1, a) = \{q_{A1}, q_{A2}, \ldots, q_{An}\}$.

FAQ по конечным автоматам (вторая часть)

Вопрос. Будет ли в этой части FAQ хоть один вопрос, ответа на который я не знаю?

Ответ. Вряд ли, хотя все может быть. В тексте разбросано много интересных сведений, но искать их по отдельности утомительно; я думаю, что одна дополнительная страничка в книге — небольшая цена за удовольствие собрать их воедино.

Вопрос. Итак, недетерминированный автомат служит лишь психологическим инструментом, который упрощает жизнь человека, с карандашом в руках разрабатывающего распознаватель некоторого формального языка?

Ответ. Не совсем. Не только человеку проще не ограничивать себя детерминированными переходами — в равной степени это относится и к компьютеру. Некоторые алгоритмы занимаются конструированием конечных автоматов, и нередки случаи, когда такой алгоритм реализуется куда проще, если не ограничивать тип автомата детерминированным. Разумеется, полученный автомат все равно придется детерминизировать и минимизировать, но здесь работает принцип сведения к уже решенной задаче, благо процедуры минимизации и детерминизации давно известны и реализованы.

Вопрос. Но ведь детерминизация не дается даром? Количество состояний теоретически может расти непомерными темпами, да и на скорости работы автомата это, наверно, отражается?

Ответ. Да, детерминизация не дается просто так. Действительно, вводя единственное дополнительное состояние в недетерминированный автомат, надо быть готовым к тому, что количество состояний соответствующего ему детерминированного автомата вырастет вдвое. Сказанное иллюстрирует рис. 2.16.

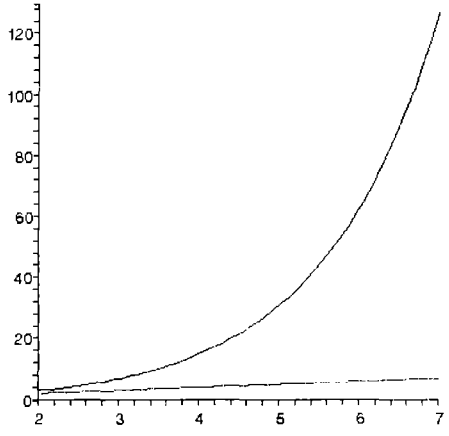


Рис. 2.16. *Экспоненциальный рост числа состояний при детерминизации*

Пологий нижний график указывает количество состояний недетерминированного автомата, в то время как безудержно растущий верхний дает хорошее представление о ситуации. С другой стороны, на рисунке изображен наихудший случай, далеко не всегда наблюдаемый на практике (к примеру, детерминизация автомата с рис. 2.9 с последующей минимизацией привела к созданию всего лишь четырех состояний, хотя в теории их могло быть $2^4 - 1 = 15$). С точки зрения скорости работы дело обстоит гораздо лучше. Любой детерминированный автомат (независимо от числа состояний) выполняет столько шагов, сколько символов записано на входную ленту («линейное время»).

Как уже отмечалось выше, можно сэкономить память, имитируя работу недетерминированного автомата непосредственно, но при этом скорость анализа входной строки может оказаться пропорциональной экспоненте от ее длины. Фактически мы получаем те же самые графики линейной и показательной функций, только относятся они уже не к расходу памяти, а к времени работы алгоритма.

Вопрос. Есть какие-нибудь особые замечания по поводу ε-автоматов?

Ответ. Пожалуй, нет. Переходы по пустой строке (ε-переходы) — всего лишь дополнительный удобный инструмент, который может здорово пригодиться в самых разных ситуациях.

Вопрос. Хм... Мы все время говорили о языках, состоящих из «допустимых адресов электронной почты» или из «строк, содержащих подстроку ABC». А что если требуется распознать язык, который состоит из любых строк, *кроме* допустимых адресов электронной почты? Или, аналогично, из любых строк, *не* содержащих подстроку ABC. Можно ли построить такой автомат?

Ответ. Да, это отличная задача. К сожалению (или к счастью) на поверку очень простая. Если некоторый язык L можно распознать при помощи автомата, «обратный» ему язык $\Sigma^* \setminus L$ тоже обладает этим свойством. Мне кажется, что человеку гораздо проще думать в позитивных терминах: корректные адреса почты; строки, содержащие данные подстроки... Если автомат для языка L уже построен, получить автомат для $\Sigma^* \setminus L$ нетрудно: для этого необходимо лишь сделать все его допускающие состояния недопускающими и наоборот.

Единственная тонкость состоит в том, чтобы сделать автомат полным перед модификацией, то есть определить все все синтаксически допустимые переходы. Примером может служить рис. 2.7. Если вы помните, там мы ввели состояние E лишь для того, чтобы любая пара вида (состояние, символ) была левой частью какого-либо правила. Здесь же ситуация более серьезная: после смены ролей допускающих и недопускающих состояний состояние E станет допускающим — и поэтому «просто забыть» о нем никак нельзя.

2.3. Проект JFLAP и конечные автоматы

В деле изучения конечных автоматов вам поможет прекрасный инструмент JFLAP. Хотя этот проект разрабатывался специально для обучения различным моделям из теории вычислений и считать его серьезным продуктом промышленного уровня вряд ли можно, многообразие реализованных в JFLAP возможностей восхищает. Я еще не раз буду возвращаться к JFLAP, чтобы проиллюстрировать различные модели и алгоритмы, описываемые в книге.

JFLAP является свободно распространяемым продуктом (еще и в исходных текстах). Скачать программу можно на сайте

```
www.cs.duke.edu/~rodger/tools/jflap/
```

Поскольку JFLAP написан на Java, вам также потребуется установить Java Runtime Environment (JRE) с вебсайта www.java.sun.com.

Так как сейчас речь идет о конечных автоматах, давайте разберемся, каким образом JFLAP может нам пригодиться на данном этапе.

Если среда Java установлена корректно, то по двойному щелчку по файлу JFLAP.jar программа запустится, и вы увидите внешне весьма неизысканное, но функциональное меню, в котором предлагается выбрать тип интересующей вас модели (см. рис. 2.17).

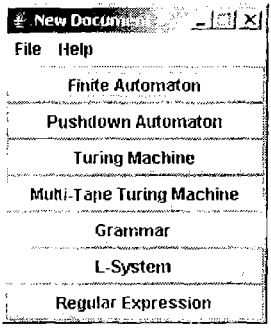


Рис. 2.17. *Главное меню JFLAP*

Шесть последних пунктов мы еще успеем обсудить, а пока что перейдем в режим Finite Automaton.

На экране должна появиться рабочая область, в которой можно довольно легко сконструировать конечный автомат любого типа (недетерминированные и ε-переходы допускаются). Выбор значка **State Creator** на панели инструментов (второй слева) означает создание новых состояний автомата: теперь любой щелчок мышью в рабочей области будет приводить к появлению очередного состояния (см. рис. 2.18).

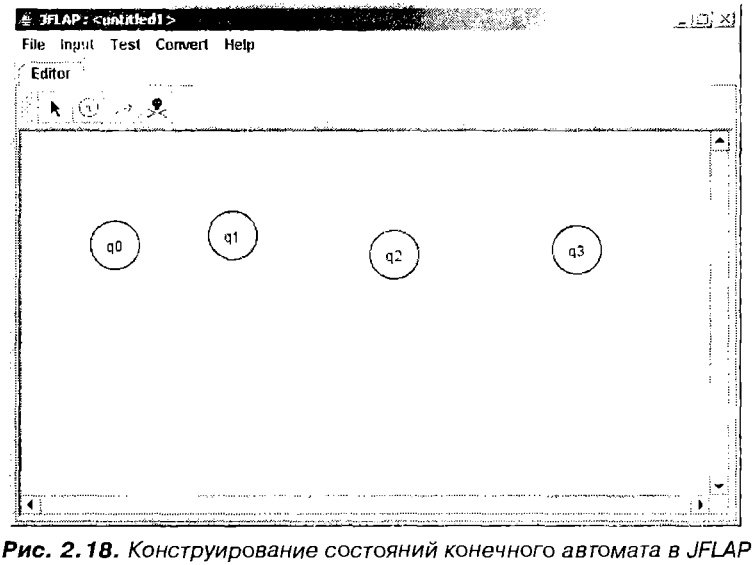


Рис. 2.18. *Конструирование состояний конечного автомата в JFLAP*

Перейдя обратно к режиму **Attribute Editor** (с помощью выбора значка со стрелкой курсора — самого левого на панели) и щелкнув правой кнопкой по любому созданному состоянию, можно сделать его начальным (Initial) или допускающим (Final)[21].

Режим **Transition Creator** (значок со стрелкой) предназначен для создания переходов. Наведите мышь на исходное состояние, затем нажмите левую кнопку мыши и, удерживая ее нажатой, укажите целевое состояние (то есть методом drag and drop). После этого вам будет предложено выбрать символ, соответствующий переходу.

Если просто нажать Enter, получится ε-переход; правда, в JFLAP они называются λ-переходами: в качестве метки пустой строки используется символ λ. Чтобы создать переход из некоторого состояния в него же, вам придется совершить нехитрый трюк: нажать левую кнопку мыши в одном месте кружочка-состояния, а отпустить чуть-чуть поодаль (но тоже в пределах состояния, разумеется). Переходы нельзя метить несколькими символами. Если некоторый переход совершается по символам a и b, вам придется создать две стрелочки. Правда, JFLAP тут же правильно оценит ситуацию и отобразит одну стрелку, помеченную двумя символами.

Таким образом, например, можно сконструировать уже известный нам недетерминированный автомат для распознавания строк над алфавитом `{a, b, c}`, содержащих подстроку `abc` (см. рис. 2.9). Его JFLAP-версия изображена на рис. 2.19.

Готовый автомат (даже недетерминированный) можно протестировать, задав какую-либо строку в качестве содержимого входной ленты. Быстрее всего это можно сделать с помощью пункта меню **Input → Fast Run**. Так, задав в качестве входа строку `aaabcca`, мы получим сообщение о ее допускании (см. рис. 2.20).

Как видно из рисунка, JFLAP не только сообщает о самом факте допускания строки, но и указывает, какая именно последовательность переходов переводит автомат в допускающее состояние. Если существует несколько таких последовательностей, их можно вывести по очереди, нажимая кнопку **Keep looking**. Щелчок по кнопке **I'm done** вернет управление в редактор JFLAP.

Что можно дальше делать с недетерминированным автоматом? Разумеется, детерминизировать! Для выполнения этого действия предназначен пункт меню **Convert → Convert to DFA**.

Поскольку JFLAP предназначается в основном для учебных целей, в него встроено немало средств пошагового и даже ручного исполнения

[21] Я стараюсь описывать интерфейс JFLAP по возможности кратко. Вместе с проектом поставляется приличная документация, которую просто нет смысла дублировать.

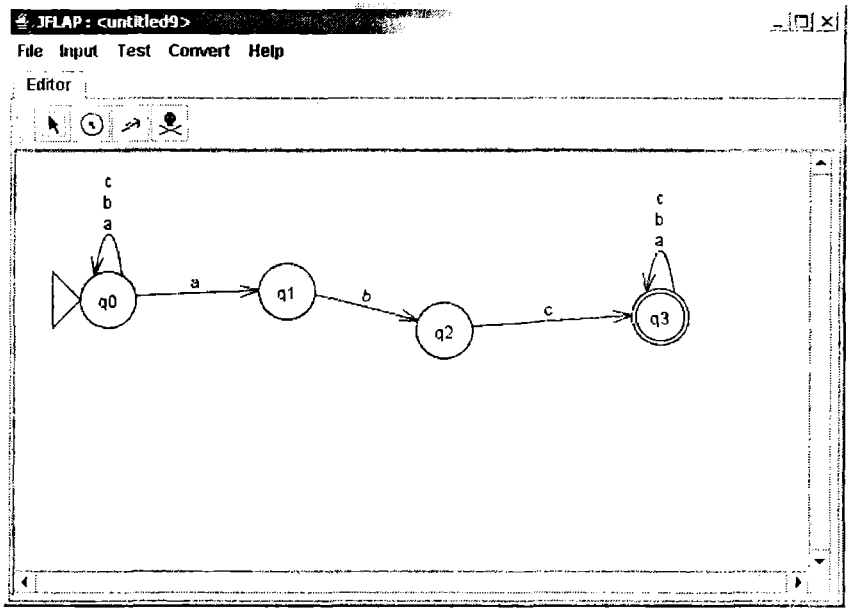


Рис. 2.19. *Автомат для распознавания строк, содержащих подстроку abc*

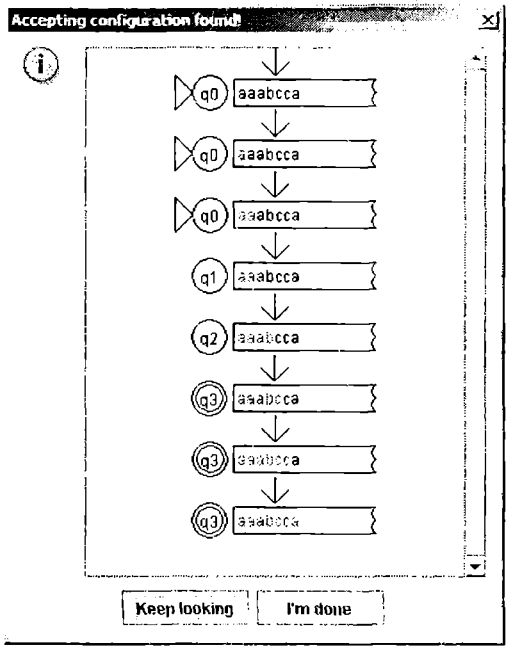


Рис. 2.20. *Допускание автоматом строки aaabcca*

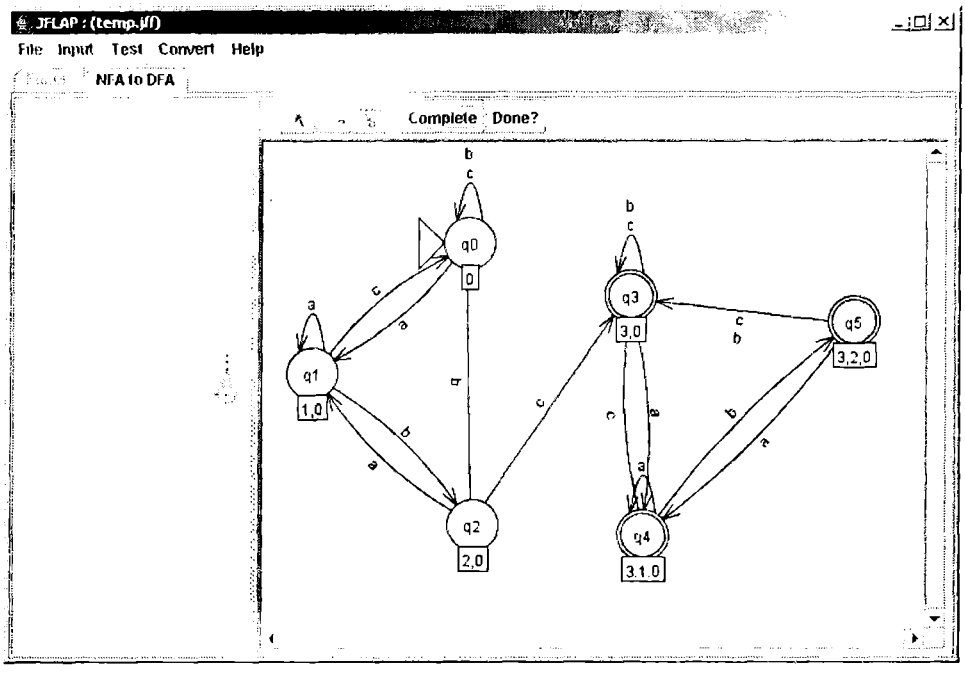


Рис. 2.21. *Выполнение алгоритма детерминизации конечного автомата*

алгоритмов. Впрочем, вы можете просто нажать кнопку **Complete**, чтобы сразу получить детерминированную версию автомата (см. рис. 2.21).

К сожалению, построенный автомат почти всегда «вылезает» за пределы видимой области, и нет никакой другой возможности «сжать» конструкцию, кроме ручного перетаскивания. Мелочь, но...

После щелчка по кнопке **Done**? готовый автомат будет перенесен в новое окно.

Следующий шаг — минимизация автомата — тоже предусмотрен в JFLAP. Процедура минимизации запускается при выборе пункта меню **Convert → Minimize DFA**.

Если минимизируемый автомат неполон (то есть не все синтаксически допустимые переходы определены), JFLAP автоматически исправит этот недостаток, направляя недостающие переходы к специально созданному состоянию (этот алгоритм был уже проиллюстрирован на рис. 2.7).

В нашем случае никаких дополнительных усилий не требуется. Автомат полон, и можно сразу же приступить к минимизации. Опять-таки, в образовательных целях JFLAP позволяет выполнить минимизацию по шагам; если этого не требуется, просто щелкните по корню дерева в правом

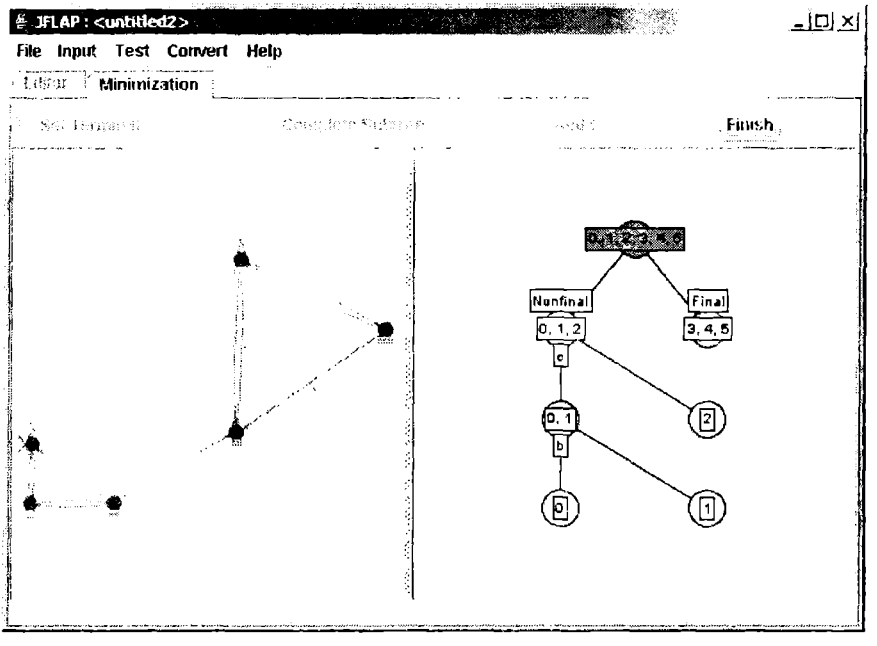


Рис. 2.22. *Минимизация конечного автомата с помощью JFLAP*

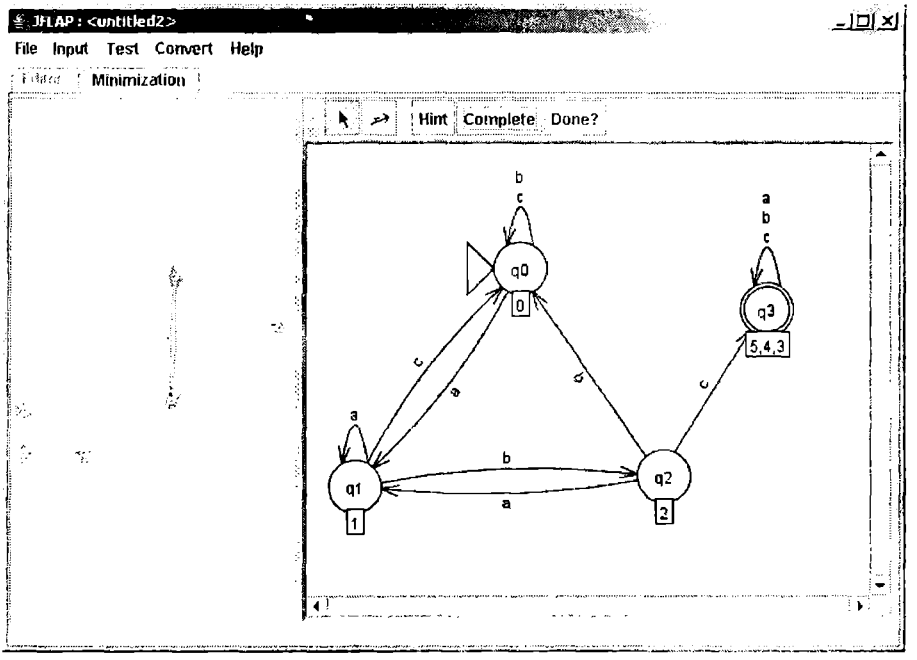


Рис. 2.23. *Минимизация конечного автомата с помощью JFLAP*

окне и нажмите кнопку **Complete Subtree**. Различные терминальные узлы сгенерированного дерева будут соответствовать различным классам эквивалентных состояний (рис. 2.22).

Кнопка **Finish** открывает в новом окне кружочки, соответствующие состояниям минимизированного автомата. Переходы пока не выводятся, поскольку JFLAP еще раз предлагает вам проверить собственные знания и выполнить алгоритм нахождения правильных переходов вручную. Кнопкой **Complete** можно быстро закончить построение автомата (см. рис. 2.23), затем щелчком по кнопке **Done**? перенести автомат в новое окно.

Обратите внимание, что полученный автомат является точной копией автомата, изображенного на рис. 2.15 (таким образом, запрограммированные нами процедуры работают так же, как и их аналоги из JFLAP).

Итоги

- Детерминированный конечный автомат — это второе (после регулярных выражений) рассматриваемое нами средство для формального описания языка.
- Детерминированный конечный автомат можно представить как некоторое «компьютерообразное» устройство. Его можно спроектировать на бумаге, а затем довольно просто преобразовать в программу, выполняемую реальным компьютером.
- Введение недетерминированного поведения не дает никакого усиления выразительной мощи, однако позволяет проектировать гораздо более простые автоматы.
- «Пустые» ε-переходы также не дают ничего нового в терминах мощности устройств, но являются таким же полезным средством проектировщика, как и недетерминированное поведение.
- Детерминированный конечный автомат можно минимизировать, уменьшив количество его состояний без потери функциональности.
- Недетерминированный конечный автомат можно выполнять на компьютере непосредственно, но работа такого алгоритма «эмуляции» может занять очень длительное («экспоненциальное») время.
- Недетерминированный конечный автомат можно детерминизировать. При этом количество состояний может резко возрасти, зато скорость работы гарантированно будет линейной.

Глава 3 Связь конечных автоматов и регулярных выражений

- Преобразование регулярного выражения в конечный автомат
- Преобразование конечного автомата в регулярное выражение
- Практические следствия. Поиск подстрок, удовлетворяющих заданному регулярному выражению
- Функции конвертирования в JFLAP

В предыдущей главе конечный автомат использовался для описания языка, состоящего из корректных адресов электронной почты. С другой стороны, еще раньше мы употребили для той же цели регулярное выражение. Там же указывалось, что существуют языки нерегулярные, то есть не представимые с помощью регулярного выражения. Какова же вычислительная мощь конечных автоматов? Можно ли с их помощью описать нерегулярный язык?

Так вот, оказывается, что описательная мощь конечных автоматов и регулярных выражений[22] *абсолютно одинакова*. Иными словами, конечные автоматы могут служить для описания любых регулярных языков, но при этом сила автоматов регулярными языками и ограничивается. Это утверждение можно доказать конструктивно, разработав процедуры конвертирования регулярного выражения в конечный автомат и наоборот, чем мы сейчас и займемся.

3.1. Преобразование регулярного выражения в конечный автомат

Напомню определение регулярных выражений первой главы.

Регулярными выражениями над Σ являются любые элементы алфавита Σ, а также пустая строка ε:

a, где a — это любой элемент алфавита Σ;
ε.

Примечание.

Нередко к регулярным выражениям добавляют еще пустое множество ∅.

[22] Разумеется, речь идет о стандартных («теоретических») регулярных выражениях, не включающих в себя популярные расширения вроде поддержки ссылок.

кже в свою очередь регулярными выражениями будут являться и ре-ьтаты операций с регулярными выражениями над Σ:

a ∪ b), (ab) и a*, где a и b — любые регулярные выражения над Σ

пользуя эти «атомы», можно построить любые, даже самые сложные улярные выражения. Таким образом, для превращения регулярного вы-кения в автомат необходимо научиться представлять в виде автоматов простейшие, атомарные конструкции. Рассмотрим их по порядку.

- Одиночный символ используемого алфавита a. Автомат для распознавания символа строится довольно очевидным образом (см. рис. 3.1). Заменяя переход по a на ε-переход, получаем автомат, распознающий пустую строку ε.

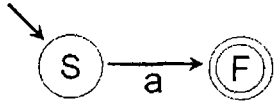


Рис. 3.1. *Автомат, распознающий одиночный символ алфавита*

- Объединение двух регулярных выражений a и b (то есть (a ∪ b)). Предположим, что для выражений a и b уже построены соответствующие автоматы A и B. Задача состоит в том, чтобы их правильно объединить. Решение показано на рис. 3.2. Здесь уже можно (и нужно) воспользоваться недетерминированным поведением. Допустим, на вход автомата поступает регулярное выражение b. Первый же шаг расщепляет мир на два новых. В первом мире автомат переходит к стартовому состоянию блока A, во втором — к стартовому состоянию блока B. Блок A не распознает входное выражение, и автомат в первом мире заканчивает работу в недопускающем состоянии. Блок B во втором мире входное выражение распознает, что в итоге означает допускание выражения недетерминированным автоматом.

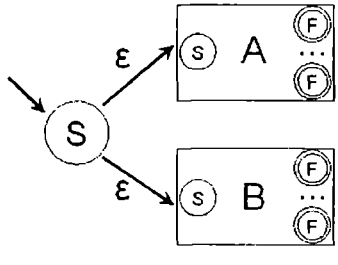


Рис. 3.2. *Автомат, распознающий объединение двух регулярных выражений*

- Конкатенация выражений (ab). Поскольку в этом случае автомату требуется распознать два подряд идущих выражения, следует расположить блоки A и B последовательно (рис. 3.3). Допускающие состояния автомата A соединяются со стартовым состоянием автомата B.

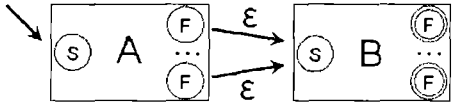


Рис. 3.3. *Автомат, распознающий конкатенацию двух регулярных выражений*

- Автомат для замыкания Клини («звездочка») выглядит, пожалуй, несколько экстравагантнее предыдущих. Поскольку запись a* означает «нуль или более повторений», стартовое состояние автомата должно быть допускающим. Если же на вход поступила какая-либо строка, придется ее целиком разобрать, а затем снова перейти к стартовому состоянию автомата (см. рис. 3.4).

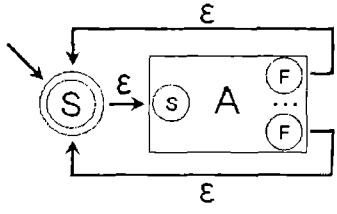


Рис. 3.4. *Автомат, соответствующий замыканию Клини*

Если вспомнить теорию регулярных выражений, любая конкатенация или объединение обязательно заключается в скобки. Поэтому всегда можно найти самое внутреннее атомарное подвыражение и преобразовать его в автомат. Затем мы продолжаем конвертирование для внешнего подвыражения и так далее. Это очень похоже на процесс разбора и вычисления значения арифметического выражения со скобками, только вместо выполнения арифметических операций мы переводим текстовые конструкции в состояния и правила перехода. На практике регулярное выражение обычно не перегружается скобками, но приоритеты операций все равно существуют, и при желании недостающие скобки можно расставить автоматически.

Чтобы окончательно разобраться с процессом конвертирования регулярного выражения в конечный автомат, имеет смысл рассмотреть простой пример. Построим автомат, соответствующий выражению $(a \cup b)^*c$[23] (см. рис. 3.5).

[23] С точки зрения теории здесь недостает лишь внешних скобок: $((a \cup b)^*c)$.

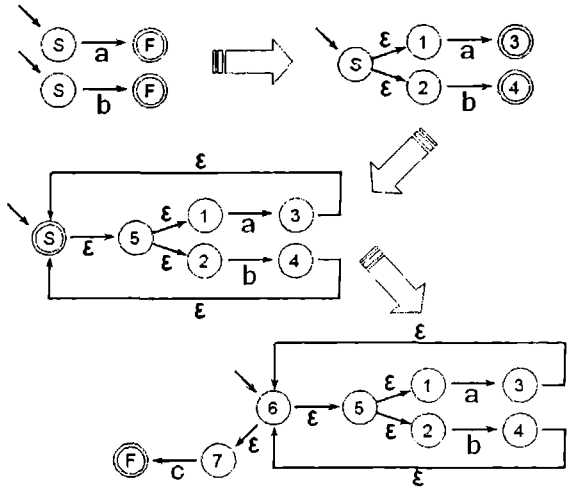


Рис. 3.5. *Конструирование автомата для выражения (a ∪ b)*c*

Итак, конструирование начинается с двух атомарных автоматов, распознающих одиночные символы a и b. На втором шаге они объединяются согласно рис. 3.2, чтобы получить распознаватель объединения a ∪ b. Следующий этап — получение замыкания Клини (рис. 3.4). На последнем шаге используется последовательное соединение автоматов, соответствующее конкатенации выражений (a ∪ b)* и c. При осуществлении этой операции мы соединяем допускающие состояния первого автомата со стартовым состоянием второго, и тот факт, что допускающее состояние первого автомата в данном случае является одновременно и стартовым, не играет никакой роли.

3.2. Преобразование конечного автомата в регулярное выражение

Процесс получения регулярного выражения, эквивалентного данному автомату, не так нагляден, но все же достаточно прост. Перед тем, как рассмотреть его подробнее, стоит сделать пару замечаний об ε-строках:

- Конкатенация любого количества пустых строк представляет собой пустую строку: εε...ε = ε.
- Конкатенация пустой строки ε с непустой строкой a есть строка a: εa = aε = a.

Эти свойства позволяют «сжимать» строки, содержащие ε-подстроки: aaεbεεεcεεddε = aabcdd.

Автомат, пригодный для перевода в регулярное выражение, должен обладать двумя свойствами:

- Ни один переход не должен вести в стартовое состояние S. Если автомат не удовлетворяет этому свойству, следует ввести новое стартовое состояние S' и добавить правило перехода S', ε → S.
- Автомат должен содержать лишь одно допускающее состояние. Запрещаются переходы, исходящие из допускающего состояния. Выполнение этого свойства достигается аналогичным образом. Предположим, F_1, F_2, ..., F_n — допускающие состояния автомата. Вычеркнем их из списка допускающих, введем новое состояние F, помеченное как допускающее, а затем добавим правила перехода F_1, ε → F, F_2, ε → F, ..., F_n, ε → F.

Обратите внимание, что переход, ведущий из состояния в него же, является одновременно входящим и исходящим. Поэтому автомат с переходом из стартового состояния в стартовое или из допускающего в допускающее также нуждается в доработке.

Суть алгоритма преобразования состоит в том, чтобы последовательно заменять состояния соответствующими им регулярными выражениями. Если в автомате есть переход из состояния A в состояние B, а из B — в C, можно добавить прямой переход из A в C, пометив его регулярным выражением, полученным из переходов A → B, B → C и B → B, если он существует (см. рис. 3.6).

Полученное устройство формально уже нельзя назвать «автоматом». Скорее это диаграмма переходов, где переход может осуществляться уже не только по одиночным символам, но и по целым регулярным выражениям. Как только в диаграмме останется лишь один переход, соединяющий стартовое состояние с допускающим, его метка и будет искомым регулярным выражением.

Поскольку в процессе работы алгоритма мы будем работать с диаграммой переходов (по сути дела графом), исходный автомат тоже удобнее представлять в терминах узлов и ребер, а не состояний и правил. Таким

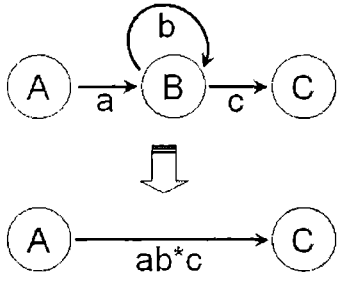


Рис. 3.6. *Замена состояния регулярным выражением*

іразом, вместо «существуют правила A, a → B, A, b → B и A, c → B» я буду ворить «ребро (A, B) помечено строкой «a, b, c»»[24].

:перь можно привести алгоритм преобразования полностью (на псев-ікоде):

```
ДЛЯ ВСЕХ ребер автомата

    ЕСЛИ метка ребра равна a1, a2, ..., an

        заменить метку на (a1 ∪ a2 ∪ ... ∪ an)

ДЛЯ КАЖДОГО узла диаграммы K (кроме стартового и допускающего)

    ДЛЯ КАЖДОГО узла A (A ≠ K)

        ДЛЯ КАЖДОГО узла B (B ≠ K)

            ЕСЛИ существуют ребра (A, K) и (K, B)
                Mak = метка ребра (A, K);
                Mkb = метка ребра (K, B);

                ЕСЛИ существует ребро (K, K) (Mkk = метка ребра (K, K))
                    M = Mak Mkk* Mkb;

                ИНАЧЕ
                    M = Mak Mkb;

                ЕСЛИ существует ребро (A, B) (Mab = метка ребра (A, B))
                    метка ребра (A, B) = Mab ∪ M;

                ИНАЧЕ
                    метка ребра (A, B) = M;

        КОНЕЦ ЦИКЛА

    КОНЕЦ ЦИКЛА

    удалить узел K и связанные с ним ребра;

КОНЕЦ ЦИКЛА
```

Здесь важно отметить, что на диаграмме переходов не может существовать трех ребер (A, B). Если из одного состояния можно попасть в другое по одному из нескольких символов ходной ленты, следует перечислить эти символы в метке единственного ребра.

В итоге будут удалены все узлы, кроме стартового и допускающего, а метка единственного оставшегося ребра и будет ответом. Из полученного регулярного выражения сразу же следует исключить лишние ε-символы, а затем по возможности упростить.

Пошаговое выполнение алгоритма для результирующего автомата на рис. 3.5 (то есть процесс обратного конвертирования автомата в регулярное выражение) показано на рис. 3.7 (исключаемые состояния помечены крестиками). Учитывая, что стартовое состояние автомата имеет входящие переходы, я добавил новое стартовое состояние в соответствии с правилами, описанными выше.

Результирующее выражение ε(ε(εaε ∪ εbε))*εc после удаления избыточных символов ε превращается в уже знакомое нам (a ∪ b)*c.

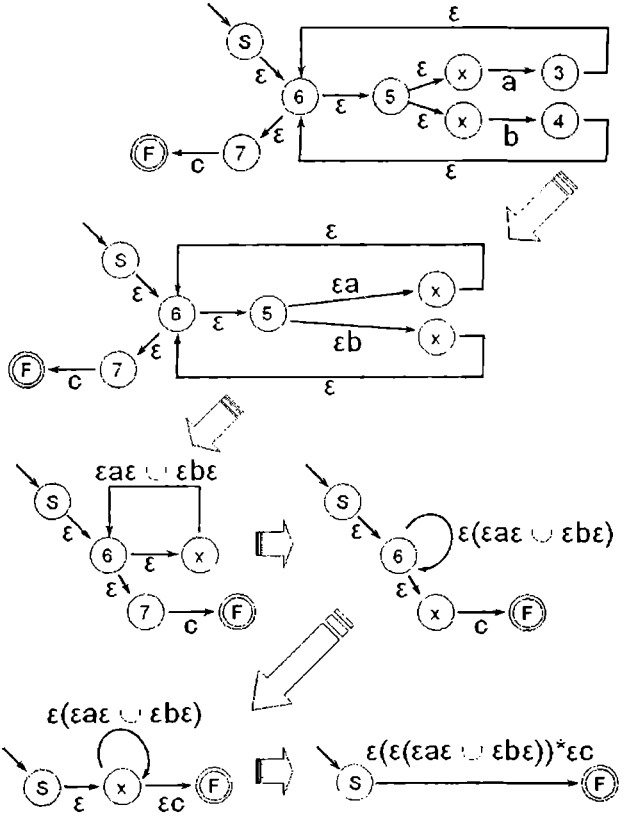


Рис. 3.7. *Преобразование автомата в регулярное выражение*

3.3. Практические следствия. Поиск подстрок, удовлетворяющих заданному регулярному выражению

Постановка задачи

принципе, на этом раздел можно было бы и закончить. Автомат можно реобразовать в регулярное выражение, а регулярное выражение — в втомат. Алгоритмы (хотя и не доведенные до уровня листингов) описаны, римеры рассмотрены, рисунки приведены. Однако кое-какие практиче-кие аспекты эти алгоритмы не проясняют. Не берусь рассуждать об этих спектах слишком подробно (тем более что тема хорошо рассмотрена в звестной книге Фридла[25]), но обойти их стороной тоже не могу.

теории регулярные выражения служат для описания языка, то есть ножества строк. Точка. Это значит, что регулярное выражение (или онечный автомат) может ответить «да» или «нет» на вопрос: «является и данная строка элементом данного языка?» Другие функции в эти нструменты попросту не заложены.

граничиваясь «теоретическим» пониманием регулярных выражений и онечных автоматов, можно написать «волшебную функцию», о которой ла речь в предыдущей главе: регулярное выражение «компилируется», о есть конвертируется в форму конечного автомата. Конечный автомат етерминизируется[26] и минимизируется, а затем либо эмулируется на омпьютере, либо переводится в непосредственно исполняемый код. риятное свойство скомпилированного таким образом регулярного вы-ажения — в скорости его работы. Поскольку конечный автомат оста-авливается после считывания всех символов ленты, а на каждом шаге итывается ровно один символ, время распознавания строки, заданной егулярным выражением, пропорционально ее длине.

а практике нам хочется большего. Например, мира во всем мире. Или обы компьютер не зависал. Или, по крайней мере, чтобы по заданному екоторым образом шаблону можно было бы найти интересующие нас рагменты во входной строке. Вернемся к автомату, изображенному на ис. 2.9. Он может определить, содержит ли данная строка подстроку BC, но не может сообщить, где именно найдено вхождение. С другой ороны, если выйти за пределы теоретических построений, автомат — о просто алгоритм, выполняемый обычным компьютером. Никто не ешает добавить нам в этот алгоритм блоки, отслеживающие подстроки

[25] Дж. Фридл. Регулярные выражения. СПб, Питер, 2003.

[26] Поскольку в данном случае детерминизация сводится лишь к удалению ε-переходов, количество состояний обычно ненамного увеличивается, а то и даже уменьшается.

входной строки, которые перевели автомат в допускающее состояние (что означает конец поиска).

Итак, постановка задачи звучит так: найти фрагменты входной строки, удовлетворяющие заданному регулярному выражению. Для начала рассмотрим процесс поиска обычной (то есть конкретной, не заданной регулярным выражением) подстроки в строке. Благо, автомат с рис. 2.9 и его детерминированная версия с рис. 2.15 именно этим и занимаются. Разберем работу детерминированного автомата на примере входной строки AABCCC. Где в ней расположена подстрока ABC? После обработки первых трех символов автомат окажется в состоянии 3. После этого символ C переведет его в четвертое (допускающее) состояние. Итак, символ C — последний символ найденной подстроки ABC. При этом он является четвертым по счету во входной ленте, следовательно, третий и второй символы — это, соответственно, B и A, составляющие начало искомой подстроки.

К счастью, при поиске конкретной подстроки нам всегда известна ее длина, поэтому, зная, где найден конец, всегда можно вычислить, где находится начало. Если же подстрока задана выражением наподобие (ab)*c, непонятно, сколько символов она содержит — один, три или тридцать три. Таким образом, возникает довольно забавное затруднение: мы сравнительно легко можем найти количество вхождений и указать позиции их последних символов, но в каждом случае найти «начало того конца» — отдельная нетривиальная задача. Решения предлагаются разные. Наиболее простой мне кажется идея использования *обращенного* регулярного выражения.

Метод обращенного регулярного выражения

Допустим, требуется найти подстроку, заданную регулярным выражением (ab)*c (не будем ходить далеко за примерами) в строке `aaababca`. Автомат, соответствующий регулярному выражению (ab)*c, находит последний символ вхождения искомой подстроки:

```
aaababca
последний символ: c, позиция в строке: 7
```

Теперь построим автомат, распознающий язык, заданный «перевернутым» (обращенным) регулярным выражением — c(ba)*[27]. Автомату на вход подается та же самая строка. Текущим символом считается *следующий за*

[27] Это, кстати, нетрудно сделать, имея автомат для исходного выражения. Требуется лишь поменять местами стартовое и допускающее состояния (считаем, что допускающее состояние единственно; если нет — это легко исправить), а также изменить направление каждого перехода на противоположное.

только что найденным (то есть в нашем примере восьмой). Сам автомат при этом будет анализировать строку *справа налево*. Как только автомат окажется в допускающем состоянии, текущий символ входной ленты и будет искомым началом:

```
aababcba
(движемся справо налево в поисках подстроки c(ba)*)
последний символ: a, позиция в строке: 3
```

Вот и все, искомая подстрока ababc найдена в позиции 3. Теперь можно продолжить анализ следующей части входной строки.

Пока что мы не отошли далеко от «теоретических» регулярных выражений. Поиск подстрок, заданных регулярным выражением, осуществляется с помощью обычного конечного автомата (хотя и с некоторыми сложностями, но все-таки). Настоящие «интересности» начинаются с введением спецификаций ленивого и жадного поведения.

Описанный выше алгоритм поиска, основанный на детерминированном автомате, Фридл называет «text-directed», то есть управляемым входной строкой. Смысл этого термина в том, что процедура разбора действительно полностью управляется поступающей на вход информацией. Очередной символ переводит автомат в то или иное состояние, и нам остается лишь ждать, пока строка не кончится или пока текущее состояние не окажется допускающим. Большой плюс такого алгоритма — в его непревзойденной скорости. Порою даже кажется удивительным, что поиск подстроки по сложному шаблону может выполняться за время, пропорциональное длине входной строки. Да что говорить о шаблонах! Девяносто процентов программистов напишут «в лоб» поиск обычной подстроки:

```
// S  -  входная строка, L(S)  -  ее длина
// M  -  искомая подстрока, L(M)  -  ее длина
for(i = 0; i < L(S)  -  L(M); i++)
                // просматриваем входную строку
    for(j = 0; j < L(M); j++)
                // сравниваем L(M) символов во входной строке
    {           // (начиная с позиции i) со строкой M
        bool found = true;
        if(S[i + j] != M[j])
        {
            found = false;      // несовпадение...
            break;
        }

        if(found)
            // сообщить о найденной подстроке в позиции i
    }
```

Скорость работы такой процедуры в худшем случае будет пропорциональна квадрату длины строки S, явно проигрывая линейному («автоматному») алгоритму.

Недостаток «управляемого входной строкой» поиска — в нехватке гибкости. В частности, с помощью механизма конечного автомата не удастся запрограммировать спецификации ленивого/жадного поведения. Стандартной (в соответствии с определением POSIX[28]) считается реализация, в которой запрограммирована жадная процедура поиска. Алгоритм, обрисованный выше, стандарту POSIX не удовлетворяет, но переделать его довольно просто (пусть это будет упражнением для интересующихся).

«Regex-directed»- механизм

Чтобы расширить возможности регулярных выражений, придется использовать «regex-directed» (управляемый регулярным выражением) механизм. Его основа — непосредственная имитация недетерминированного автомата, подобного изображенному на рис. 2.9. Стартовое состояние автомата обязательно содержит переходы по любому символу входной ленты в себя. Из стартового состояния можно также попасть в «основную часть», распознающую регулярное выражение (именно так устроен автомат, распознающий подстроки ABC).

При имитации подобного автомата (мы уже обсуждали использование перебора с возвратом) обе проблемы предыдущего решения снимаются сами собой. Во-первых, найденная строка полностью «ловится» в процессе перевода автомата из стартового состояния в допускающее (полагаем, что стартовое состояние не имеет входящих переходов). Поэтому позиция начала найденной подстроки известна — требуется лишь посмотреть, какую часть входной строки анализировал автомат, находясь в стартовом состоянии. Во-вторых, полный перебор, как известно, всегда находит решение задачи. В процессе работы будут найдены как «ленивые», так и «жадные» вхождения — и нам останется лишь выбрать интересующие в каждом конкретном случае.

Проблема такого подхода уже была описана. В худшем случае поиск подстроки может занять весьма долгое («экспоненциальное») время.

На практике хорошие библиотеки регулярных выражений нередко комбинируют оба рассмотренных механизма. Если искомая подстрока описана регулярным выражением, не содержащим спецификаций по-

[28] Я ссылаюсь на стандарты, разработанные для ОС UNIX, поскольку Windows на уровне API без дополнительных библиотек регулярные выражения не поддерживает.

дения и других расширений, можно использовать первый подход, в отивном случае — второй. Составление хорошего регулярного выжения (не приводящего к неопределенно долгому поиску) — тоже оего рода искусство, основами которого можно овладеть, прочитав игу Фридла.

3.4. Функции конвертирования в JFLAP

астало время вернуться к старому доброму JFLAP'у и посмотреть, как нем реализованы функции преобразования регулярного выражения в нечный автомат и наоборот.

зьмем опять в качестве примера автомат с рис. 2.9. Чтобы получить вивалентное ему регулярное выражение, требуется выбрать в главном ню JFLAP пункт Finite Automaton, сконструировать автомат (это мы е делали), а затем выбрать команду **Convert → Convert FA to RE**.

сле этого вам в очередной раз придется столкнуться с настойчивыми пытками JFLAP научить вас алгоритму путем пошагового его выполния. В нынешнем случае потребуется несколько раз нажать на кнопку **it**, пока на экране не появится сообщение "Generalized Transition aph Finished!". Если вы нажмете **Do it** еще раз, то получите не слишком жливый ответ: "You're done. Go away."[29]

нопка **Export** открывает готовое регулярное выражение в новом окне м. рис. 3.8).

братите внимание, что синтаксис регулярных выражений в JFLAP отчается от стандартного. Символ «плюс» обозначает не «одно или более вторений», а операцию объединения: a+b в нотации JFLAP — это ∪ b в более привычном стиле.

ким образом, построенное выражение можно переписать в стандартной рме как

```
(((a ∪ b) ∪ c)*)(abc(((a ∪ b) ∪ c)*))
```

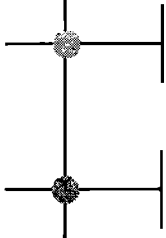
аляя лишние скобки, получаем еще более очевидное

```
(a ∪ b ∪ c)*abc(a ∪ b ∪ c)*
```

мвол пустой строки (ε в нашей нотации или λ в нотации JFLAP) в регулярных выражениях обозначается с помощью восклицательного знака. пример, a+! — это a ∪ ε.

Явное (и не единственное подобное) свидетельство бесплатности программы. Попробовали бы авторы коммерческого софта вставить что-нибудь в таком духе!

Глава 4 Конечные автоматы на практике



Конечные автоматы, по сути, вымышленные устройства, оказались на редкость полезными в практическом программировании. Ранее мы подробно говорили об использовании автоматов для работы со строками регулярных языков. Уже эту возможность трудно переоценить. Представьте, что вам требуется написать программу, вычленяющую из входного текста телефонные номера, адреса электронной почты, вещественные числа или содержимое HTML-тегов. Без использования регулярных выражений (и стоящих за ними автоматов) подобная задача превратилась бы в настоящий кошмар, а готовая программа (будем считать, что вы все-таки сумели ее написать) оказалась бы совершенно запутанной и не поддающейся даже простым модификациям.

Однако работа со строками и подстроками (в повседневном понимании) — не единственная область, в которой конечные автоматы могут принести пользу. Существуют процессы, удобно представимые в виде состояний и переходов (заметьте, не «представимые при большом рвении», а именно *удобно* представимые).

4.1. Простейшие автоматные модели

Наглядная модель: лифт

Одна из моих любимых иллюстраций — работа лифта[31].

В двухэтажном здании есть лифт, «умеющий» реагировать на команды: «открыть двери» (`o`), «закрыть двери» (`c`), «спуститься на первый этаж» (`D`), «подняться на второй этаж» (`U`). Изначально лифт находится на первом этаже, его двери открыты. Лифт может выполнить и «программу», например, такую: `cUo`, что означает «закрыть двери, подняться на второй этаж и открыть двери».

[31] Пример из моей книги «Занимательное программирование» (СПб, Питер, 2004). Не отказываться же от хорошего примера только потому, что я уже использовал его однажды?

Некоторые программы для лифта будут некорректными. К примеру, нельзя двигаться с открытыми дверями или подниматься на второй этаж, когда лифт и так находится на втором этаже. Задача заключается в том, чтобы смоделировать лифт на компьютере, то есть написать программу, которая будет уметь выполнять команды лифта (либо говорить, что команда некорректна), а также возвращать его текущее состояние (этаж и положение дверей).

В принципе, можно сразу попытаться сесть и написать соответствующие процедуры, но давайте попробуем сначала взглянуть на задачу по-другому. Лифт — это устройство, которое может иметь четыре различных состояния: «первый этаж, двери закрыты (`cD`)»; «первый этаж, двери открыты (`oD`)»; «второй этаж, двери закрыты (`cU`)» и «второй этаж, двери открыты (`oU`)». Любая команда переводит лифт из одного состояния в другое: к примеру, если лифт был в состоянии `cU`, а на вход пришла команда `D`, то лифт перейдет в состояние `cD`. Все эти факты можно изобразить в виде конечного автомата (рис. 4.1).

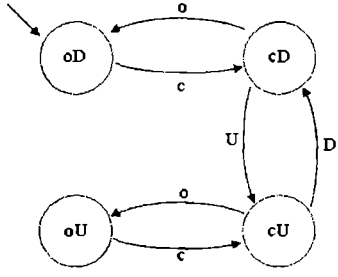


Рис. 4.1. *Схема функционирования лифта*

После десятков страниц, посвященных конечным автоматам, я думаю, вам не составит труда перевести этот автомат в текст на языке C#. Впрочем, листинг 4.1 все равно не будет лишним.

Листинг 4.1. Моделирование лифта при помощи конечного автомата

```
enum StateType {oD, cD, cU, oU};

static void Main(string[] args)
{
    string program = Console.ReadLine() + '\0';
    StateType State = StateType.oD;
    int i = 0;
    char symbol;
```

```
    try
    {
        while((symbol = program[i++]) != '\0')
        {
           Console.WriteLine("Текущее состояние: {0}", State);
           switch(State)
           {
                case StateType.oD: switch(symbol)
                      {
                         case 'c': State = StateType.cD; break;
                         default:  throw new Exception();
                      }
                      break;
                case StateType.cD: switch(symbol)
                      {
                         case 'o': State = StateType.oD; break;
                         case 'U': State = StateType.cU; break;
                         default:  throw new Exception();
                      }
                      break;
                case StateType.cU: switch(symbol)
                      {
                         case 'D': State = StateType.cD; break;
                         case 'o': State = StateType.oU; break;
                         default:  throw new Exception();
                      }
                      break;
                case StateType.oU: switch(symbol)
                      {
                         case 'c': State = StateType.cU; break;
                         default:  throw new Exception();
                      }
                      break;
           }
        }
        Console.WriteLine("Текущее состояние: {0}", State);
    }
    catch(Exception)
    {
        Console.WriteLine("Недопустимый переход");
    }
}
```

Боюсь, что осмысленно прокомментировать код, машинально созданный на основе графа, я просто не сумею. Кроме того, мы уже занимались созданием автоматов на языке программирования в начале главы. А вот результат работы программы для типичных входных данных приведу с удовольствием.

```
c:\LiftAutomaton\bin\Debug>liftautomaton
cUocDo
Текущее состояние: oD
Текущее состояние: cD
Текущее состояние: cU
Текущее состояние: oU
Текущее состояние: cU
Текущее состояние: cD
Текущее состояние: oD

c:\LiftAutomaton\bin\Debug>liftautomaton
cocUoo
Текущее состояние: oD
Текущее состояние: cD
Текущее состояние: oD
Текущее состояние: cD
Текущее состояние: cU
Текущее состояние: oU
Недопустимый переход
```

Первый ввод содержит корректную последовательность команд. Лифт послушно выполняет все шесть действий, после чего программа заканчивает работу. Во втором случае лифт не может выполнить последнюю команду («открыть двери»), поскольку двери уже и так открыты; в результате генерируется исключение.

Кофейный автомат

Другой хороший пример — функционирование автомата, продающего кофе. В прорезь опускается произвольное количество монет любого достоинства. Автомат содержит три группы кнопок:

- «обычный кофе», «капучино», «эспрессо», «какао»;
- «с сахаром», «без сахара»;
- «с молоком», «без молока»;
- «со стаканом», «без стакана».

Нажимая кнопки в каждой группе, пользователь выбирает «конфигурацию» напитка и наличие стакана (то есть должен ли автомат сам подставить картонный стаканчик или предоставить человеку возможность использовать свою тару). Последняя кнопка — «налить» — запускает процесс. Автомат не только наливает напиток, но и выдает сдачу (для простоты можно считать, что цена любой «конфигурации» одинакова, тем более что обычно так оно и есть).

Похожим образом может работать таксофон (если он принимает монеты). Очевидно, что представление подобных устройств с помощью конечных

автоматов может заметно сэкономить наши усилия по их моделированию. Мне трудно судить об инженерных аспектах разработки реальных («железных») устройств вроде таксофона или кофейного автомата, но, вероятно, воплотить конечный автомат в электронных схемах гораздо проще, чем вставлять в каждый таксофон полноценный компьютер под управлением Windows, да еще и с требуемым программным обеспечением.

4.2. Немного об автоматном программировании

Несколько общих слов

Говоря о конечных автоматах, нельзя не затронуть еще одну тему, довольно популярную в последние годы: автоматное программирование. Я бы не рискнул назвать автоматное программирование отдельной парадигмой, столь же развитой, как ООП или структурное программирование. На мой взгляд, это скорее психологический инструмент, в некоторых случаях помогающий найти более простое решение задачи. Впрочем, находятся энтузиасты (порою достаточно известные люди), доходящие в своей любви к автоматному программированию до крайностей. В одной статье мне даже попался пассаж, который, опуская стилистические изыски, можно переформулировать так: «эта программа плоха, потому что она не написана в автоматном стиле».

Позволив себе небольшое лирическое отступление, замечу, что, по моему глубокому убеждению, ни одна парадигма не может быть полностью универсальной. Конечно, теоретически каждый язык программирования одинаково пригоден для описания любого алгоритма (в том смысле, что вам, так или иначе, удастся это сделать); вопрос не в самой возможности как таковой, а в удобстве. При желании можно забить гвоздь пассатижами (сам неоднократно это делал) или приготовить яичницу на утюге (не пробовал), но лучше все-таки воспользоваться более подходящими средствами.

Хороший язык программирования (и в этом можно полностью согласиться с мнением создателя C++ Бьярна Страуструпа) должен быть мультипарадигмальным. Не случайно библиотека STL, во многом изменившая лицо C++, поддерживает стиль, куда более близкий к функциональному программированию (о котором еще пойдет речь в конце книги), чем к ООП. Поэтому, говоря об автоматном программировании, я хочу представить его именно как новый блестящий инструмент для вашей коллекции, но не как Универсальный Молоток, который надо применять везде и всюду. Впрочем, на любую хорошую идею найдется неприятность в виде слишком рьяных последователей.

Видимо, расплывчатое понятие «удобства» использования той или иной парадигмы во многом определяется личными склонностями и образованием («бэкграундом») программиста. Вполне возможно, что математику всегда проще думать в терминах предикатов и функций, а инженеру — в терминах автоматов.

Что же такое «автоматное программирование»?

Идея автоматного программирования очень проста: попробуйте представить выполнение алгоритма в виде работы аппарата, похожего на конечный автомат.

Под «автоматом» в автоматном программировании понимается куда более мощное устройство, чем конечный автомат из предыдущей главы. Общих черт у них две:

- автомат в любой момент времени находится в некотором состоянии;
- в зависимости от выполнения того или иного условия автомат переходит в какое-то другое состояние.

Как вы, надеюсь, помните, конечный автомат представляет собой очень простое устройство, чья единственная способность состоит в реагировании (путем изменения текущего состояния) на последовательность символов, записанных на входную ленту. Кроме того, я с самого начала пытался описать его как некоторый вполне реалистичный механизм, при желании воплощаемый в виде настоящего, «железного» аппарата.

Сейчас разговор идет об инструменте программиста-практика. Поэтому, во-первых, с самого начала понятно, что полученный автомат будет в том или ином виде реализован на обычном языке программирования, таком как Pascal или C#. А во-вторых,

- при переходе автомата из одного состояния в другое может выполняться сколь угодно сложный фрагмент кода на языке программирования;
- очередное состояние автомата определяется с помощью вычисления любой, сколь угодно сложной функции.

Таким образом, в полученном гибридном устройстве чисто автоматная модель сочетается с произвольными фрагментами текста на обычном языке программирования.

Практическое применение автоматного программирования: автоматное описание компьютерной игры

Пожалуй, о сути автоматного программирования мне сказать больше нечего. Надеюсь, его идеология станет яснее из примеров.

В принципе, первыми примерами применения автоматного подхода были модели лифта и кофейного автомата. По логике эти программы можно назвать полностью автоматными, хотя они и слишком просты — нам нигде не приходилось выходить за рамки обычных конечных автоматов.

Наиболее явными кандидатами на использование автоматного программирования (если не придерживаться мнения о повсеместном его применении) являются логически сложные модели, в которых можно выделить несколько устойчивых «состояний». С каждым из таких состояний связан свой собственный алгоритм моделирования.

Рассмотрим общую схему работы типичной компьютерной игры. Сразу после загрузки пользователь попадает в главное меню:

```
новая игра

загрузка игры

информация об авторах

таблица почета

выход
```

Самые простые действия связаны с пунктами `информация об авторах` (выводится список авторов, после чего производится возврат в главное меню), `таблица почета` (печатается таблица почета, затем управление вновь передается в главное меню) и `выход` (работа программы завершается).

Пункт `новая игра` переводит программу в режим игры. То же самое происходит и в результате выбора пункта `загрузка игры`, но при этом пользователю предварительно предлагается указать файл, содержащий сохраненную игру.

Даже таким простым действиям, как вывод таблицы почета или информации об авторах, могут соответствовать довольно сложные алгоритмы, поскольку мы живем не в начале восьмидесятых, и не украсить обычную текстовую распечатку всякими графическими эффектами и музыкой — моветон. Да и при выходе из программы тоже обычно приходится сначала освобождать память и закрывать открытые файлы.

Сама игра тоже обычно представляет собой весьма нетривиальную процедуру, в которой можно выделить те или иные «состояния». Например, обычный ход игры может прерываться «подыграми», в которых вам предлагается, допустим, подобрать пароль к компьютеру или выиграть в дартс у Большого Джо. Кроме того, в игре должно быть предусмотрено свое собственное меню, вызываемое обычно по клавише ESC. Как правило, оно содержит, по крайней мере, следующие пункты:

```
возврат к игре

сохранение игры

выход в главное меню
```

В любой игре должно быть, как минимум, две концовки: либо игрок проиграл (сообщаем, что Game over и возвращаемся в главное меню), либо дошел до конца (в этом случае сначала надо как-то поприветствовать победителя, но потом все равно произвести переход в главное меню).

Обратите внимание, что все описанные процедуры очень слабо связаны между собою. Например, алгоритму вывода таблицы почета ничего не надо знать об игре, подыграх и концовках. Такой ситуации отлично соответствует модель состояний (возможно, в данном случае больше подходит термин «режимов») и переходов между ними (см. рис. 4.2).

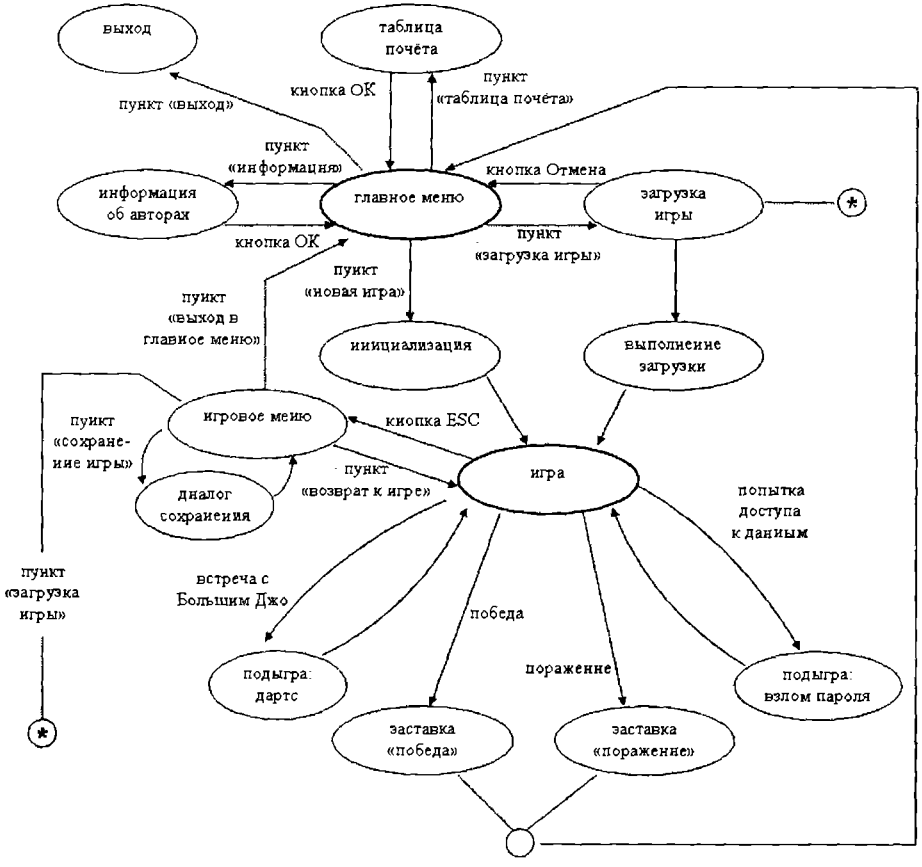


Рис. 4.2. *Автоматное описание игры*

Полученный автомат можно механически преобразовать в switch-case онструкцию, как было показано ранее.

Для начала следует создать перечисление, описывающее все состояния гры:

```
enum TGameState
{MAIN_MENU, EXIT, LOAD_GAME, HISCORE_TABLE, ABOUT_AUTHORS,
INITIALIZATION, PERFORM_LOAD, GAME, GAME_MENU, SAVE_DIALOGUE,
DARTS_SUBGAME, PASS_SUBGAME, VICTORY, DEFEAT};
```

Также нам потребуется глобальная переменная `CurrentGameState`:

```
TGameState CurrentGameState;
```

Если теперь написать для каждого состояния соответствующую функцию ее именем может служить имя состояния с префиксом fun_), то главный правляющий блок программы приобретает простой и логичный вид:

```
CurrentGameState = TGameState.MAIN_MENU;
                        // изначально выводится главное меню

while(CurrentGameState != TGameState.EXIT)
                        // пока текущее состояние не EXIT
 switch(CurrentGameState)
 {
    case TGameState.MAIN_MENU:      fun_MAIN_MENU();      break;
    case TGameState.EXIT:           fun_EXIT();           break;
    case TGameState.LOAD_GAME:      fun_LOAD_GAME();      break;
    case TGameState.HISCORE_TABLE:  fun_HISCORE_TABLE();  break;
    case TGameState.ABOUT_AUTHORS:  fun_ABOUT_AUTHORS();  break;
    case TGameState.INITIALIZATION: fun_INITIALIZATION(); break;
    case TGameState.PERFORM_LOAD:   fun_PERFORM_LOAD();   break;
    case TGameState.GAME:           fun_GAME();           break;
    case TGameState.GAME_MENU:      fun_GAME_MENU();      break;
    case TGameState.SAVE_DIALOGUE: fun_SAVE_DIALOGUE(); break;
    case TGameState.DARTS_SUBGAME: fun_DARTS_SUBGAME(); break;
    case TGameState.PASS_SUBGAME:  fun_PASS_SUBGAME();  break;
    case TGameState.VICTORY:       fun_PASS_VICTORY();  break;
    case TGameState.DEFEAT:        fun_DEFEAT();        break;
 }
```

Переходы между состояниями программируются непосредственно в ункциях `fun_ИМЯ()`. Чтобы перейти в состояние `ИМЯ_СОСТОЯНИЯ`, достаточно лишь изменить значение переменной `CurrentGameState`

```
CurrentGameState = TGameState.ИМЯ_СОСТОЯНИЯ;
```

вернуться в главный управляющий блок.

Чтобы окончательно прояснить ситуацию, имеет смысл привести некоторые функции на псевдокоде (можно и все, но места жалко):

```
void fun_MAIN_MENU()
{
    // вывести главное меню и ожидать выбора пользователя
    // ...
    ЕСЛИ выбран пункт «новая игра»
        CurrentGameState = TGameState.INITIALIZATION; return;
    ЕСЛИ выбран пункт «загрузка игры»
        CurrentGameState = TGameState.LOAD_GAME; return;
    ЕСЛИ выбран пункт «информация об авторах»
        CurrentGameState = TGameState.ABOUT_AUTHORS; return;
    ЕСЛИ выбран пункт «таблица почета»
        CurrentGameState = TGameState.HISCORE_TABLE; return;
    ЕСЛИ выбран пункт «выход»
        CurrentGameState = TGameState.EXIT; return;
}

void fun_LOAD_GAME()
{
    // вывести диалог загрузки игры
    // ...
    ЕСЛИ пользователь отменяет загрузку
        CurrentGameState = TGameState.MAIN_MENU; return;

    // загрузка прошла успешно
    CurrentGameState = TGameState.PERFORM_LOAD;
}

void fun_HISCORE_TABLE()
{
    // вывести на экран таблицу почета
    // ...
    CurrentGameState = TGameState.MAIN_MENU;
}

void fun_ABOUT_AUTHORS()
{
    // вывести на экран информацию об авторах
    // ...
    CurrentGameState = TGameState.MAIN_MENU;
}

//
```

```
void fun_GAME()
{
    while(true)  // основной цикл игры
    {
        // здесь вывод спрайтов, обработка клавиатуры
        // и так далее ...
       ЕСЛИ нажата клавиша ESC
          CurrentGameState = TGameState.GAME_MENU; return;
       ЕСЛИ встретились с Большим Джо
          CurrentGameState = TGameState.DARTS_SUBGAME; return;
       ЕСЛИ попытались войти в компьютер
          CurrentGameState = TGameState.PASS_SUBGAME; return;
       ЕСЛИ победили в игре
          CurrentGameState = TGameState.VICTORY; return;
       ЕСЛИ закончились все жизни
          CurrentGameState = TGameState.DEFEAT; return;
    }
}
```

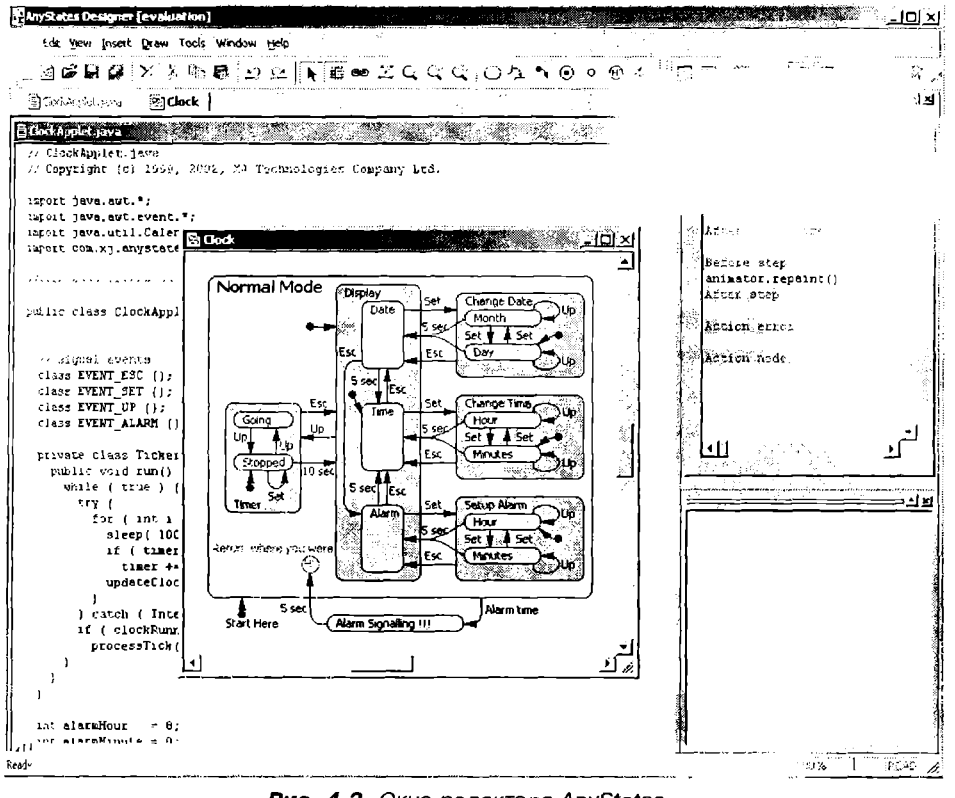


Рис. 4.3. *Окно редактора AnyStates*

Автомат на рис. 4.2 состоит в основном из очень крупных «строительных блоков», таких как основная процедура игры или подыгра «дартс». Вероятно, их тоже, в свою очередь, можно представить в виде автоматов. Например, типичная процедура, описывающая поведение охранника, разбивается на три режима: «патрулировать территорию» (по умолчанию), «бежать к месту происшествия» (если подан сигнал тревоги) и «сопровождать объект» (если отдан приказ о защите конкретного персонажа).

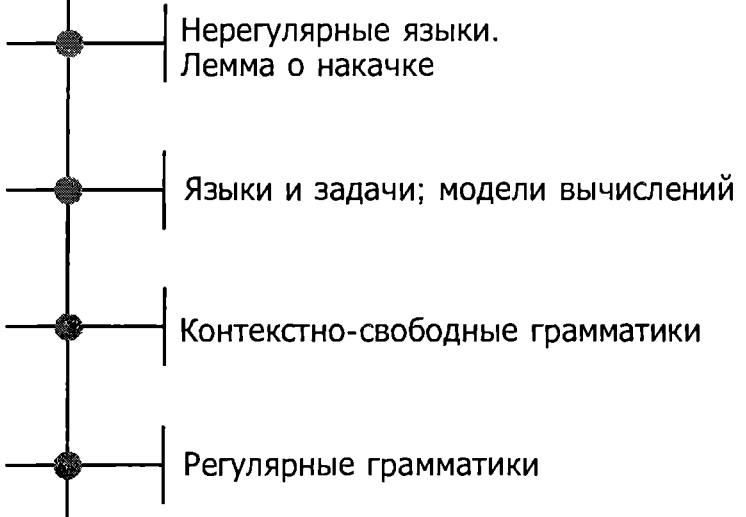
С другой стороны, очень простые действия вроде вывода таблицы почета (без графических эффектов) можно без особого вреда для модульности вставить прямо в главную управляющую процедуру, тем самым упрощая схему автомата.

На основе автоматного подхода можно даже создать полноценную среду разработки. Например, таковой является инструмент `AnyStates`, выпущенный (кстати, российской) компанией XJ Technologies. Представление о нем дает рис. 4.3, на котором видно, что создаваемая программа представляется в виде «виртуального устройства», включающего в себя состояния и переходы между ними.

Итоги

- Психологическое значение автоматов очень велико. Компьютеру все равно, какую программу выполнять. Для него не существует простых или сложных для понимания алгоритмов, не существует различия между элегантным и уродливым кодом. Человеку же необходимы абстракции.
- Программировать на уровне машинного кода — занятие на редкость тяжелое и утомительное. Для того чтобы думать эффективнее, нам нужны процедуры и функции, объекты и модули. Конечный автомат — еще одна абстракция, которая может помочь взглянуть на задачу по-новому.
- Автоматное программирование — не «серебряная пуля», но оно может быть очень полезным во многих случаях (особенно если вы сразу видите, как представить алгоритм в виде состояний и переходов).

Глава 5 Нерегулярные языки и контекстно-свободные грамматики



Тема *формальных грамматик* всегда казалась мне весьма теоретизированной и трудной для понимания. К сожалению, без некоторого объема теории все равно не обойтись до того, как перейти к реальным листингам, но я постараюсь сделать чтение не слишком скучным.

Формальные грамматики стоят того, чтобы с ними ознакомиться. Они используются, например, для описания синтаксиса языков программирования и предложений естественного языка[32], а также для задания правил развития систем (таких как системы Линденмайера, о которых речь пойдет в девятой главе).

Дойдя до практических алгоритмов, вы поймете всю их полезность (надеюсь, полезность автоматов мне доказать удалось).

5.1. Нерегулярные языки. Лемма о накачке

Итак, в очередной раз возвращаемся к задаче описания формального языка. Мы уже выяснили, что для некоторых языков, называемых *регулярными*, можно использовать конечные автоматы и регулярные выражения. В главе о регулярных выражениях уже указывалось, что не все языки регулярны. В частности, язык «любые строки, в которых буква "a" встречается столько же раз, сколько буква "b"» не является регулярным; следовательно, он «не по зубам» конечным автоматам и регулярным выражениям.

Перед тем, как перейти к более мощным средствам описания (вы, вероятно, уже догадались, что речь пойдет о *контекстно-свободных грамматиках*), разберемся, чем же «внутреннее устройство» регулярных языков отличается от устройства языков нерегулярных. В этом нам поможет *лемма о накачке* (pumping lemma).

Пусть L — некоторый регулярный язык с бесконечным количеством элементов. Тогда существует число N, такое, что любая строка языка L, длина которой равна или превосходит N, может быть представлена в виде такой конкатенации трех строк `xyz`, что выполняются условия:

- подстрока y непуста ($y \neq \varepsilon$);
- длина подстроки xy не превосходит N;
- строки xz, xyz, xyyz, xyyyz, ... принадлежат языку L.

Иными словами, в любой достаточно длинной строке любого регулярного языка существует подстрока, которую можно повторить произвольное число раз (в том числе исключить, то есть «повторить» нуль раз) — и полученная строка все равно будет принадлежать данному регулярному языку.

[32] Это очень спорный способ, имеющий, однако, немало сторонников.

Таким образом, лемма о накачке формулирует свойство, присущее всем регулярным языкам с бесконечным количеством элементов[33]. Обратите внимание, что нерегулярный язык тоже может обладать свойством «накачиваемости», поэтому лемма о накачке дает не так много для определения регулярности интересующего языка. Если строки языка не «накачиваются», то язык не является регулярным — и это все, что мы вправе утверждать (то есть «накачиваемость» — необходимое, но не достаточное условие регулярности языка).

С другой стороны, лемма позволяет легко доказать, что не любой язык в мире является регулярным. Чтобы далеко не ходить за примерами, рассмотрим уже знакомый язык, строки которого состоят из равного количества букв "a" и "b". Поскольку лемма должна выполняться для любой строки не короче N символов (значение N нам неизвестно), возьмем строку aa...abb...b, состоящую из N букв "a", за которыми следуют N букв "b". Таким образом, всего в этой строке присутствует 2N символов. По лемме наша строка может быть разбита на подстроки xyz, причем длина подстроки xy не превосходит N, а y — «накачиваемая» подстрока. С другой стороны, мы точно знаем, что первые N символов строки — это буквы "a". Получается, что подстрока y может состоять только из букв "a". Но любая попытка «накачки» приведет к тому, что общее число букв "a" в строке изменится, а число букв "b" останется постоянным. В итоге получается, что утверждение леммы для нашего языка не удовлетворяется, и, следовательно, он не является регулярным.

Утверждение леммы о накачке с первого взгляда выглядит довольно специфическим. Сразу трудно понять, почему строки регулярного языка должны обладать подобным свойством. Поэтому мне кажется разумным уделить несколько минут обсуждению того, откуда эта лемма берется и почему выполняется.

В действительности все обстоит достаточно просто. Если L — регулярный язык, то он распознается некоторым конечным автоматом. Пусть N — число состояний этого автомата. Изначально автомат находится в стартовом состоянии. Очередной символ входной ленты переводит автомат в какое-то другое (или в текущее) состояние. В любом случае после считывания N-1 символов (а, вероятно, и раньше) автомат обязательно окажется в каком-нибудь состоянии S, в котором он уже бывал раньше. Получается, что некоторая подстрока входной строки, ранее обозначенная нами как y, переводит автомат из состояния S снова в состояние S (возможно, по довольно хитрому маршруту). А коли так, мы можем повторить эту подстроку любое число раз, либо вообще исключить ее — и это никак не отразится на конечном результате работы автомата, что и утверждается в лемме о накачке.

[33] Как уже упоминалось, любой конечный язык является регулярным, поэтому проводить какие-то дополнительные исследования для таких языков нет надобности.

5.2. Языки и задачи; модели вычислений

Уже сейчас можно преступить к рассмотрению формальных грамматик, но я все-таки сделаю еще одно небольшое отступление, крайне важное для понимания теоретической ценности исследования тем этой книги. Поговорим немного о связи между *языками* и *задачами*.

Под «задачей» (неформально, разумеется) понимается обычная задача, решение которой можно попытаться поручить компьютеру. Есть некоторый набор исходных данных, на основании которого формируется решение, представляющее собой набор «выходных» значений. Например, входными данными задачи сортировки («дано») является массив значений, а выходными («найти») — отсортированный массив тех же самых значений. Решить задачу «определить корень уравнения $x^2 + a = 10$» значит получить результирующее значение x по входному значению параметра a.

Главный интерес для нас представляют сейчас *задачи принятия решений* (decision problems). Результатом решения такой задачи является ответ «да» или «нет». Таким образом, задача сортировки не принадлежит к задачам принятия решений, а задача проверки данного массива на упорядоченность — принадлежит.

На первый взгляд класс задач принятия решений кажется очень узким, но в действительности это не совсем так. Чуть позже мы обсудим классификацию всех задач по их сложности (условно говоря, «простые», «средней сложности», «сложные», «вообще не решаемые»). Так вот, задачи принятия решения встречаются в каждом из таких классов. Так же, как зоологу нет нужды исследовать всех коров в мире, чтобы понять анатомию коровы, специалистам по теории вычислений не требуется анализировать все задачи данного класса, чтобы изучить их особо важные свойства.

Задачи принятия решений наиболее просты по формулировке, и поэтому любимы учеными в качестве «подопытной коровы». Кроме того, при желании можно свести любую задачу, разброс результатов работы которой не бесконечен[34] (а во многих случаях такое предположение сделать можно) к набору задач принятия решений.

Например, задача сортировки массива элементов (1, 3, 2) сводится к поочередной проверке упорядоченности всех массивов, состоящих из тех же самых элементов:

```
упорядочен_ли (1, 3, 2)
упорядочен_ли (1, 2, 3)
упорядочен_ли (2, 1, 3)
упорядочен_ли (2, 3, 1)
```

[34] Иными словами, результат работы которой (число, массив, строка...) является элементом некоторого конечного множества.

```
упорядочен_ли (3, 1, 2)
упорядочен_ли (3, 2, 1)
```

Как только решение оказывается положительным («да, упорядочен»), можно сказать, что исходная задача тоже решена, поскольку мы знаем тот массив, для которого вызов `упорядочен_ли()` дал утвердительный ответ. Сведение сортировки к принятию решения оказалось возможным, поскольку результат работы алгоритма сортировки (массив из чисел 1, 2 и 3) является представителем конечного множества {(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)}. Нам оставалось лишь проанализировать все элементы множества возможных ответов в поисках правильного.

Разумеется, подобный метод решения задачи нельзя назвать оптимальным, но в качестве теоретического инструмента исследований свойств задачи он бывает полезен.

Вернемся к задачам принятия решений. Обратите внимание, что входные данные любой задачи можно закодировать в виде строки[35]. Например, массив чисел (10, 45, 1, 53) можно закодировать как последовательность цифр и пробелов:

```
10 45 1 53
```

Таким образом, решить задачу означает проанализировать некоторую строку и вынести вердикт: «да» или «нет». Вам это ничего не напоминает? Это же задача распознавания языка!

Хотя на первый взгляд задачи — это задачи, а языки — это языки, при более близком рассмотрении оказывается, что языки и задачи — суть одно и то же. Давайте проведем параллель между задачей проверки упорядоченности массива целых чисел и соответствующим ей языком.

Язык	Задача
Множество всех строк Σ^* над алфавитом Σ	Множество всех строк, состоящих из цифр и пробелов
Язык L как подмножество строк Σ^*	Подмножество тех строк, в которых записаны разделенные пробелами целые числа в порядке возрастания
Определение принадлежности данной строки языку L	Определение упорядоченности данного массива
Алгоритм распознавания языка L	Умение определить упорядоченность любого массива, переданного в качестве параметра

Тождественность языков и задач поможет нам ближе подобраться к основаниям компьютерной науки. Сейчас у нас есть компьютеры, кото-

[35] На практике любая компьютерная программа, считывающая входные данные из файла, получает их как раз в таком виде.

рые «как-то» умеют решать задачи. Но давайте задумаемся, во-первых, о том, как можно реально разработать «железный» механизм, решающий задачу, а во-вторых, каковы пределы сложности задач, решаемых обычным компьютером.

Так же, как и тема распознавания формальных языков, тема решающих задачи устройств и их мощности будет еще не раз подниматься в книге. Сейчас нам все равно придется ограничиться материалом предыдущих глав. К тому же самое интересное мне хочется оставить на десерт.

Итак, представьте, что задачи у нас есть, а компьютеров нет. Как можно создать «железное» устройство, решающее задачу? Далее, предположим, у нас уже есть механизм, решающий задачу A. Проще или сложнее будет создать устройство, решающее задачу B?

Мы уже обсуждали существование регулярных и нерегулярных языков. Регулярный язык может быть распознан конечным автоматом, в то время как для нерегулярных языков мощи конечного автомата уже недостаточно. Теперь можно провести ту же самую аналогию для задач. Существуют задачи, разрешимые конечным автоматом, например, проверка того, что данная строка содержит букву a. В то же время «более сложные» задачи, описываемые нерегулярными языками, конечному автомату не по силам.

С инженерной точки зрения этот результат очень важен. Конечный автомат — это ведь не какое-то абстрактное устройство, но и схема пригодного для практического воплощения «в железе» механизма, можно сказать, простейшего компьютера. Но вместе с радостной мыслью о том, что мы в силах сконструировать коробочку, распознающую регулярный язык, приходит сомнение: а удастся ли создать что-то более совершенное, подходящее для распознавания нерегулярного языка?

Да и вообще, что есть там, в мире нерегулярных языков? Быть может, среди них тоже есть своя иерархия, и, научившись распознавать одни нерегулярные языки, мы все еще будем полностью бессильными перед другими? Кроме того, не забывайте, что для любого интересующего языка придется строить отдельный физический автомат (эта необходимость тоже не добавляет энтузиазма).

Впрочем, мое небольшое отступление превращается в большое отступление. Пора вернуться к формальным грамматикам.

5.3. Контекстно-свободные грамматики

Каким образом еще можно описать язык? Рассмотрим один известный способ, называемый *контекстно-свободной грамматикой*.

```
S → A_N
A → умный
A → вежливый
A → глупый
A → гордый
A → богатый
N → кот
N → бурундук
N → пес
N → селезень
```

итать эту запись следует так: «S есть A_N[36], где A есть умный или вежливый или глупый или гордый или богатый, а N есть кот или бурундук ли пес или селезень». Чему в итоге может быть равно значение строки ? Например, «умный_кот» (Матроскин) или «богатый_селезень» (дядя крудж). Полный список всех возможных значений S формирует язык, даваемый грамматикой.

зучим теперь устройство контекстно-свободной грамматики поближе. акие элементы можно в ней выделить? Во-первых, это стартовый символ задающий вид строк описываемого языка на самом верхнем уровне. Во-орых, это *нетерминальные символы* (или, проще говоря, переменные), адиционно обозначаемые заглавными латинскими буквами. В-третьих, о *терминальные символы* — базовые, атомарные элементы языка, ормирующие его алфавит.

ерминальные символы обычно записывают в нижнем регистре, чтобы личить их от переменных. В-четвертых, это собственно *правила вывода*, ждое из которых сопоставляет некоторой переменной[37] строку, состоую из конкатенации произвольных терминальных и нетерминальных мволов.

олее формально (но по сути совершенно аналогично) контекстно-своодная грамматика определяется как объект (Σ, V, R, S), где Σ — множево терминальных символов (алфавит), V — множество переменных, — множество правил, а S — стартовый символ. При этом правила еют вид A → c, где A — переменная, а c — строка над алфавитом ∪ V). Такая грамматика называется контекстно-свободной потому, что авило A → c может применяться независимо от того, в каком контексте третилась переменная A (то есть независимо от того, какие символы окружают).

Я использовал знак подчеркивания вместо пробельного символа лишь для того, чтобы сделать его более заметным.

Или стартовому символу, который тоже является переменной.

В нашем случае терминальными символами являются слова умный, вежливый, глупый, гордый, богатый, кот, бурундук, пес и селезень, а также символ нижнего подчеркивания. Переменные — символы A и N, стартовый символ — S. Обычно стартовый символ грамматики не выделяют явным образом: по умолчанию считается, что переменная в левой части самого первого правила и есть стартовый символ.

Грамматику можно записать короче, пользуясь вертикальной чертой для объединения правил с одинаковой левой частью:

```
S → A_N
A → умный | вежливый | глупый | гордый | богатый
N → кот | бурундук | пес | селезень
```

Такая запись выглядит читабельнее, но не забывайте, что здесь все равно десять правил, а не два.

Только что заданный при помощи грамматики язык является конечным, а как же можно описать бесконечный язык? Ценой некоторых усилий. Возьмем, к примеру, регулярный язык из главы о конечных автоматах, описывающий корректные адреса электронной почты. Напомню, он описывается регулярным выражением

```
[a-zA-Z0-9._-]+@[a-zA-Z0-9_-]+(\.[a-zA-Z0-9_-]+)+
```

Этот же самый язык можно задать и контекстно-свободной грамматикой:

```
S → H@BT
H → C''H | C''
B → C'B | C'
T → .BT | .B
C' → a | ... | z | A |     | Z | 0 |     | 9 | _ | - |
C'' → C' |
```

Запись с помощью грамматики, на мой взгляд, не отличается особой понятностью. Регулярные выражения читаются куда легче. Впрочем, разобраться в том, почему приведенная грамматика действительно описывает адреса электронной почты, вполне реально.

Итак, корректный адрес (S) состоит из:

- «головы» (H), которой соответствует регулярное выражение `[a-zA-Z0-9._-]+`;
- знака @;
- «тела» (B), описываемого выражением `[a-zA-Z0-9_-]+`;
- наиболее сложной части — «хвоста» (T), которому соответствует регулярное выражение `(\.[a-zA-Z0-9_-]+)+`.

На «нижнем уровне» — уровне терминальных символов мы работаем с множеством C', представляющим собой любой допустимый символ

«тела» и «хвоста», а также с множеством допустимых символов «головы» C''. Это множество помимо всех символов из C' включает в себя также символ точки.

Что такое «голова»? Это либо одиночный символ набора C'', либо символ C'', за которым следует корректная «голова». Подобное рекурсивное определение может здорово запутать, поэтому давайте разберемся как следует в том, что именно происходит[38] (см. рис. 5.1).

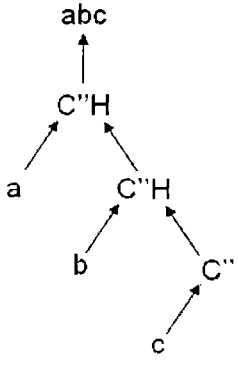


Рис. 5.1. *Анализ работы правила H → C''H | C''*

Итак, H есть C''H или C''. Выбрав вторую альтернативу, мы сразу же получим одиночный символ из набора C'' в качестве значения H. Поскольку этот вариант неинтересен, выберем первую альтернативу: H есть C''H. Значение C'' — это простой символ, допустим, `a`. Таким образом, H есть aH. Теперь необходимо определить, что такое H из правой части правила. Здесь у нас опять две возможности, и опять мы выберем C''H. Пусть на сей раз C'' = b, откуда H есть abH. Теперь, наконец, выберем вторую альтернативу — H есть C''. Указав символ `c` в качестве значения C'', получаем итоговый результат: H есть `abc`.

Здесь уместен вопрос: а что мы вообще делаем? Пытаемся понять, является ли данная строка корректным адресом электронной почты или сами формируем строку на основе грамматики? Ответ таков: вообще говоря, грамматики могут использоваться как для анализа, так и для генерации строк. В данном случае я использую направление от грамматики к строке, поскольку так проще понять принцип работы грамматики.

Вернемся к разбору адресов электронной почты. «Тело» устроено так же, как и «голова», с той лишь разницей, что вместо символа C'' на сей раз

[38] Вот где пригодится общая культура программирования! Читатели, знакомые с функциональными и логическими языками, безусловно, сочтут определение «головы» простым и естественным.

используется C'. «Хвост», в свою очередь, состоит из точки, за которой следует строка того же вида, что и строка B, составляющая «тело». Затем идет либо очередной сегмент «хвоста», либо ничего, что означает конец строки. Вот и все.

Впрочем, язык корректных адресов электронной почты регулярен. А как насчет нерегулярных языков? По крайней мере, язык, строки которого состоят из произвольного количества букв a, за которыми следуют столько же букв b (его нерегулярность уже была доказана) описывается с помощью контекстно-свободной грамматики очень просто:

```
S → aSb | ε
```

5.4. Регулярные грамматики

Что такое регулярные грамматики: польза ограниченности

Итак, мы убедились, что контекстно-свободные грамматики способны описать, по крайней мере, один регулярный язык и один нерегулярный. Перед тем, как заняться исключительно нерегулярными языками, мне кажется уместным привести один эффектный результат. Дело в том, что, наложив некоторые ограничения на допустимые в грамматиках правила, можно получить так называемые *регулярные грамматики*, пригодные для описания любых регулярных языков, но не более того.

Регулярные грамматики делятся на *праволинейные* (right linear) и *леволинейные* (left linear). В праволинейных грамматиках разрешены лишь правила вида

```
A → Σ*,
A → Σ*V.
```

В леволинейных грамматиках действуют аналогичные ограничения. Здесь допустимы правила

```
A → Σ*,
A → VΣ*.
```

На более понятном языке эти записи читаются так. В праволинейных грамматиках допустимы лишь те правила, в которых переменной сопоставляется:

- либо строка из произвольного количества символов алфавита[39];
- либо строка, состоящая из произвольного количества символов алфавита, за которыми следует одна (и только одна) переменная.

[39] Обратите внимание, что пустая строка ε также допустима (ε ∈ Σ*).

Аналогично, в леволинейных грамматиках допустимы лишь те правила, в которых переменной сопоставляется:

- либо строка из произвольного количества символов алфавита;
- либо строка, состоящая из переменной, за которой следует произвольное количество символов алфавита.

Таким образом, например, правило A → abcN разрешено в праволинейной грамматике, но запрещено в леволинейной; правило A → abc разрешено, а правило A → AN запрещено в обеих грамматиках.

Следуя традиции, займемся сначала праволинейными грамматиками.

Создание конечного автомата из регулярной грамматики.

Как же доказать, что праволинейная грамматика может описывать регулярные языки? Разумеется, построить регулярное выражение или конечный автомат, соответствующий заданной грамматике. Займемся построением конечного автомата. В принципе, можно написать два «обобщенных» правила, объясняющих, как сконструировать требуемый автомат, но мне кажется, что такое описание будет менее понятным, поэтому я предлагаю потратить немного больше времени и разобраться в процессе основательнее.

Итак, нам известна некоторая грамматика. Как на ее основе построить конечный автомат? Во-первых, необходимо создать:

- Стартовое состояние и пометить его стартовым символом грамматики.
- Допускающее состояние (метка не имеет значения, пусть будет Fin).
- По одному состоянию на каждый нетерминальный символ грамматики и пометить их соответствующими буквами. Например, если в грамматике встречаются нетерминалы N, K и T, следует создать три состояния. Первое будет помечено буквой N, второе — буквой K, а третье — буквой T.

Эти состояния будут не единственными в нашем автомате, но метки остальных не имеют значения.

Затем следует проанализировать грамматику правило за правилом и добавить в автомат состояния и переходы в соответствии с алгоритмом[40]:

- Если правило грамматики имеет вид A → aB, добавить в автомат переход A, a → B.
- Если правило имеет вид $A \to a_1a_2...a_nB$, для начала добавить в автомат n − 1 новых состояний (их метки не имеют значения; я буду обращать-

[40] Как и раньше, прописные буквы обозначают переменные (нетерминалы), а строчные — символы алфавита.

ся к ним по номерам 1, 2, ..., n–1). Затем добавить переходы A, a_1 → 1; 1, a_2 → 2; ...; (n–2), a_{n-1} → (n–1) и (n–1), a_n → B.
- Если правило имеет вид A → B, добавить в автомат переход A, ε → B.
- Если правило имеет вид A → $a_1a_2...a_n$, добавить состояния и переходы как в пункте 2, за исключением последнего перехода — он будет иметь вид a_n → Fin.
- Если правило имеет вид A → a, добавить в автомат переход A, a → Fin.
- Если правило имеет вид A → ε, добавить в автомат переход A, ε → Fin.

В более наглядном виде эти правила показаны на рис. 5.2.

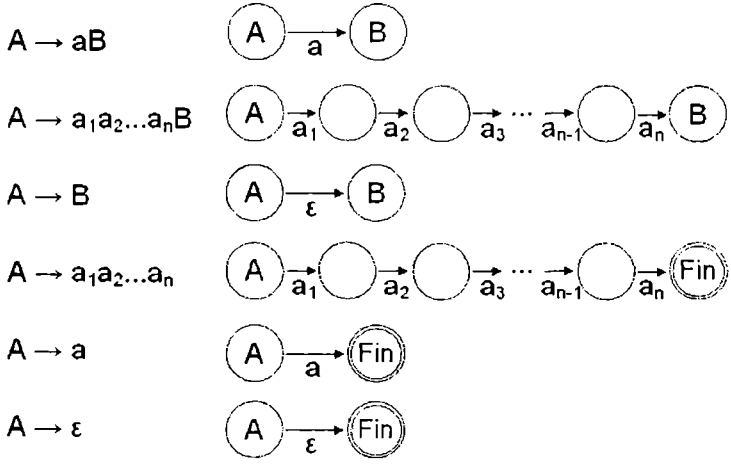


Рис. 5.2. *Правила для преобразования праволинейной грамматики в автомат*

Рассмотрим какой-нибудь практический пример. Праволинейная грамматика

```
S → aS | bA | ε
A → bA | ε
```

определяет регулярный язык, состоящий из любого количества символов a, за которыми следует любое количество символов b (в терминах регулярных выражений — язык a*b*). Запишем грамматику в развернутом виде:

```
S → aS
S → bA
S → ε
A → bA
A → ε
```

В соответствии с алгоритмом создаем три состояния конструируемого автомата: S, A и Fin, где S — стартовое состояние, а Fin — допускающее. Теперь добавляем переходы, соответствующие каждому из пяти правил грамматики:

```
S → aS: переход S, a → S;
S → bA: переход S, b → A;
S → ε: переход S, ε → Fin;
A → bA: переход A, b → A;
A → ε: переход A, ε → Fin.
```

Готовый автомат показан на рис. 5.3.

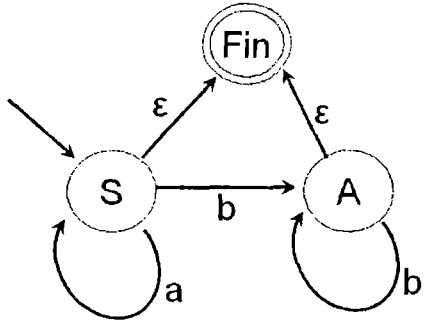


Рис. 5.3. *Автомат, построенный для данной праволинейной грамматики*

Создание регулярной грамматики из конечного автомата

Сгенерировать регулярную грамматику на основе существующего автомата еще проще:

- Обозначить все состояния автомата заглавными латинскими буквами. Выбрать метку стартового состояния автомата в качестве стартового символа конструируемой грамматики. Убедиться, что автомат содержит лишь одно допускающее состояние (множественные допускающие состояния легко объединяются в одно при помощи ε-переходов; этот метод использовался в главах 2 и 3).
- Добавить в грамматику правило F → ε, где F — метка допускающего состояния автомата.
- Для любого перехода A, c → B автомата добавить в грамматику правило A → cB.

Рассмотрим автомат, который допускает строки, состоящие из символов a, b и c и содержащие подстроку abc. Этот автомат уже приводился в пример в главе о конечных автоматах, но я повторю его на рис. 5.4, чтобы вам не приходилось листать книгу туда и обратно.

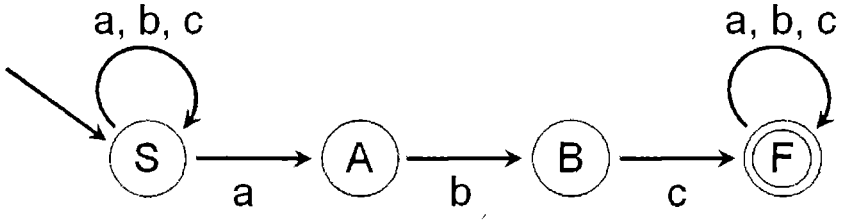


Рис. 5.4. *Автомат для распознавания языка* $(a \cup b \cup c)^*abc(a \cup b \cup c)^*$

Девять правил переходов автомата преобразуются в девять правил вывода праволинейной грамматики:

Автомат	Грамматика
S, a → S	S → aS
S, b → S	S → bS
S, c → S	S → cS
S, a → A	S → aA
A, b → B	A → bB
B, c → F	B → cF
F, a → F	F → aF
F, b → F	F → bF
F, c → F	F → cF

Последнее правило вывода создается для допускающего состояния: F → ε. В итоге получаем грамматику:

```
S → aS | bS | cS | aA
A → aA
B → cF
F → aF | bF | cF | ε
```

При создании грамматики «в лоб» количество правил вывода в ней не будет минимальным, но в данном случае это для нас не имеет особого значения.

Регулярность леволинейных грамматик

Вероятно, мое решение не всем придется по вкусу, но я не буду доказывать регулярность леволинейных грамматик. В конце концов, что хорошо для теоретического учебника, не обязательно хорошо для книги, стоящей скорее на стыке теории и практики.

Да, по-хорошему неплохо бы доказать, что леволинейная грамматика может быть преобразована в конечный автомат и наоборот. Можно попробовать построить для любой леволинейной грамматики эквивалентную праволинейную (и, опять же, наоборот). Но в целом леволинейная грамматика очень похожа на праволинейную (естественно, это ничего не доказывает), и, поверьте мне на слово, провести для нее аналогичные рассуждения не так сложно. В некоторых учебниках эта задача вообще дается как самостоятельное упражнение.

Мы же лучше сконцентрируемся на дальнейших темах, на мой взгляд, куда более интересных, чем конвертирование линейных грамматик друг в друга.

Поддержка регулярных грамматик в системе JFLAP

JFLAP умеет как создавать конечные автоматы на основе праволинейных грамматик, так и генерировать праволинейную грамматику по данному конечному автомату. Рассмотрим эти действия на примере все того же языка $(a \cup b \cup c)^*abc(a \cup b \cup c)^*$, который состоит из строк над алфавитом {a, b, c}, содержащих подстроку `abc`.

Чтобы преобразовать автомат в праволинейную грамматику, следует выбрать пункт **Convert → Convert to Grammar** в окне с автоматом, а затем нажать кнопку **Show All**, чтобы пропустить пошаговое выполнение алгоритма.

Результат для автомата с рис. 5.4 показан на рис. 5.5.

Построение автомата на основе грамматики ничем не сложнее. После выбора пункта **Grammar** в главном меню JFLAP откроется окно редактора формальной грамматики.

Левый символ самого первого правила будет считаться стартовым. Если правую часть какой-либо строки оставить пустой, вы получите правило вида $A \to \varepsilon$ (в обозначениях JFLAP $A \to \lambda$). В каждой строке можно записать лишь одно правило (таким образом, вместо записи $A \to Ab \mid Ac$ вам придется ввести две строки — $A \to Ab$ и $A \to Ac$).

Давайте на сей раз выберем грамматику попроще, например, описывающую язык a^*b^*:

```
S → aS | bA | ε
A → bA | ε
```

Грамматика, набранная в редакторе JFLAP, показана на рис. 5.6.

Построение автомата начинается с выбора пункта меню **Convert → Convert Right-Linear Grammar to FA**. Далее опять придется нажать кнопку **Show All** для завершения операции.

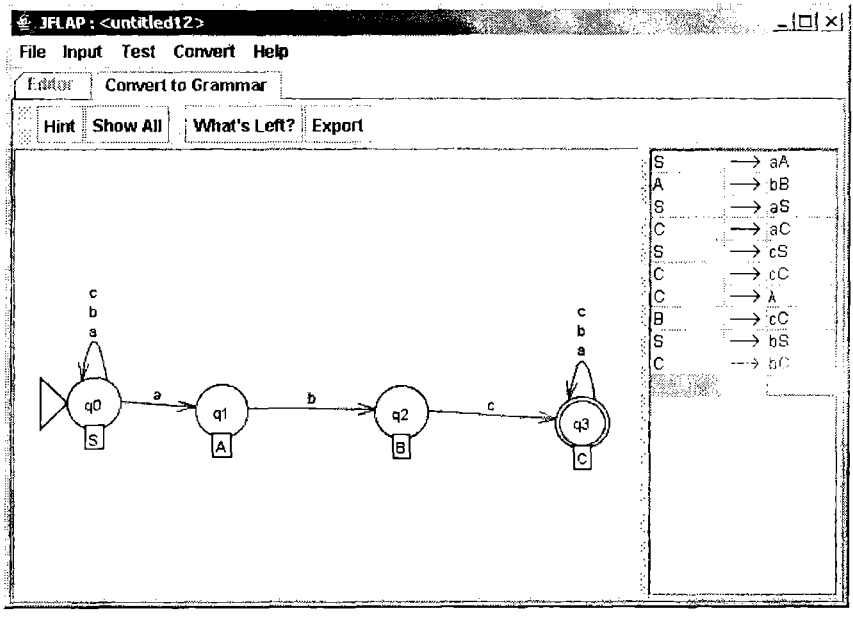


Рис. 5.5. *Преобразование конечного автомата в праволинейную грамматику (JFLAP)*

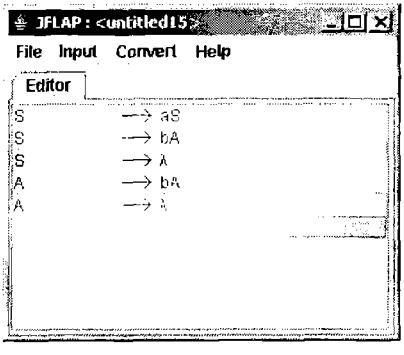


Рис. 5.6. *Редактор формальных грамматик системы JFLAP*

Готовый автомат (см. рис. 5.7), как и ожидалось, полностью идентичен автомату, изображенному на рис. 5.3.

Если введенная грамматика не будет праволинейной, JFLAP выведет диагностическое сообщение и отменит конструирование автомата.

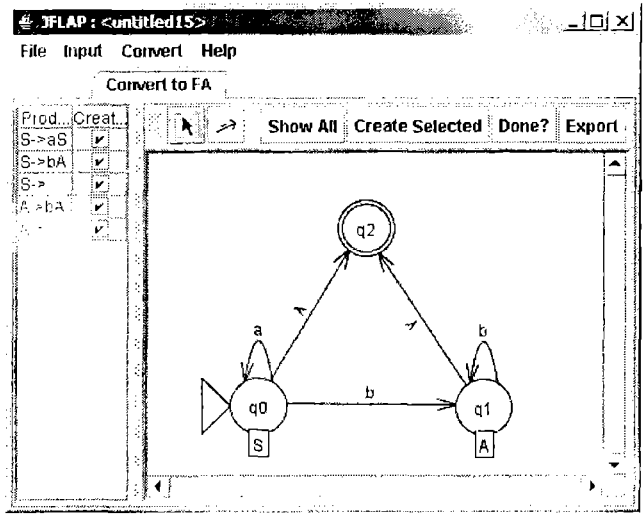


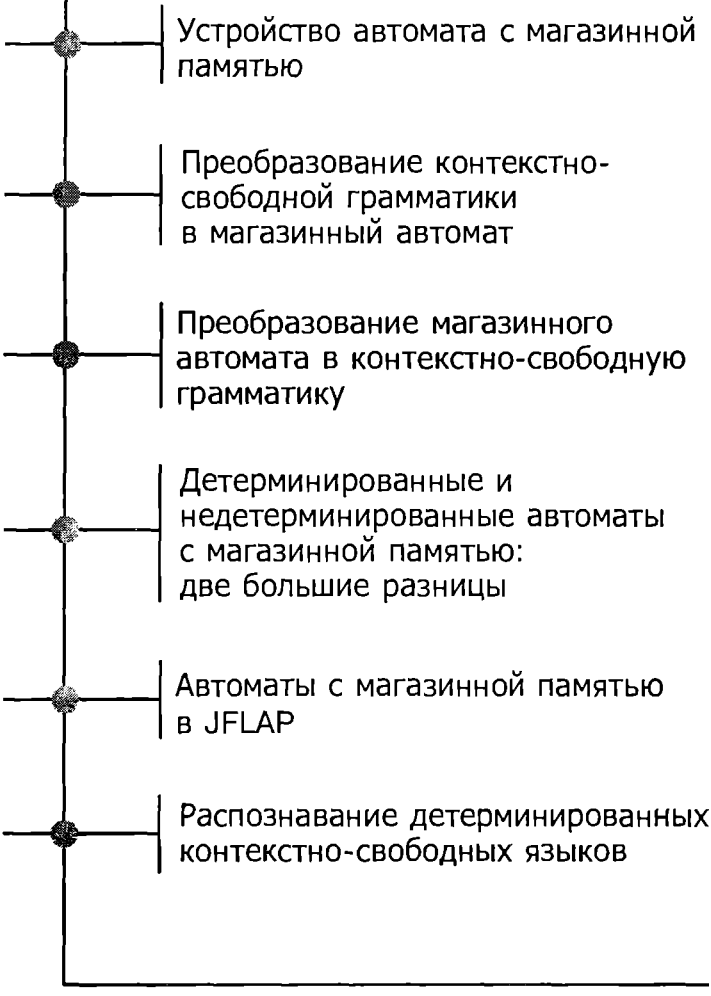
Рис. 5.7. *Построенный автомат (JFLAP)*

Итоги

- Наиболее общий класс языков, известный нам на данный момент — класс контекстно-свободных языков. Существование нерегулярных языков несложно доказать при помощи леммы о накачке.
- Пока что мы познакомились лишь с одним средством, пригодным для описания столь непростых множеств — с контекстно-свободными грамматиками. Обратите внимание, что контекстно-свободные грамматики сами по себе не дают алгоритма распознавания языка, подобного конечному автомату. С их помощью язык можно лишь описать.
- Принудительно ограничив набор допустимых правил контекстно-свободной грамматики, можно превратить ее в регулярную (лево- или праволинейную). Регулярные грамматики пригодны лишь для описания регулярных языков.
- Ценой некоторых усилий регулярную грамматику можно сконвертировать в конечный автомат и наоборот. Эти два замечательных алгоритма (в совокупности с алгоритмами преобразования конечного автомата в регулярное выражение и наоборот) перебрасывают мостик между регулярными грамматиками, конечными автоматами и регулярными выражениями.

- Один из самых глубоких результатов: тождественность языков и задач принятия решений. Мы еще не раз будем возвращаться к обсуждению к этой темы. Пока что можно сказать, что между задачами существует примерно такая же иерархия, как и между языками. Как для распознавания разных типов языков требуется инструмент разной мощности, так и для решения одной задачи может потребоваться более сложный механизм, чем для решения другой. «Сложность» задачи характеризуется, прежде всего, затратами времени. Так, мы уже убедились, что определить принадлежность любой строки данному регулярному языку вполне удается за время, пропорциональное длине этой строки («линейное время»). Конечно, можно возразить: если некоторое конкретное устройство (конечный автомат) решает задачу за линейное время, это еще не значит, что ее нельзя решить быстрее с помощью другого, более совершенного устройства. Обсуждение этого вопроса тоже пока откладывается на будущее.

Глава 6 Автоматы с магазинной памятью



Обсуждая регулярные выражения, мне то и дело приходилось упоминать некую «волшебную функцию», при помощи которой любую строку можно было бы проверить на соответствие данному регулярному выражению. Уже в следующей главе стало ясно, что конечные автоматы являются механизмом, при помощи которого эта «волшебная функция» реализуется довольно просто. Регулярные грамматики, как уже упоминалось, сами по себе не содержат алгоритма построения «волшебной функции», но, будучи регулярными, могут быть преобразованы в старые добрые конечные автоматы.

С контекстно-свободными грамматиками ситуация сложнее. Мощи конечных автоматов, понятно, недостаточно для распознавания нерегулярных языков (коими являются, к примеру, контекстно-свободные языки). Какое же устройство окажется достаточно совершенным?

В этой главе речь пойдет о так называемых *автоматах с магазинной памятью* (pushdown automata). «Чистые» автоматы с магазинной памятью имеют меньшее значение, чем обычные конечные автоматы, однако их устройство само по себе интересно; кроме того, автоматы с магазинной памятью заполняют пробел в парах «описание-механизм»:

- регулярное выражение / регулярная грамматика — конечный автомат
- контекстно-свободная грамматика — автомат с магазинной памятью

В некоторых случаях идеология автоматов с магазинной памятью успешно применяется и на практике, но об этом речь пойдет в следующей главе.

6.1. Устройство автомата с магазинной памятью

Общие положения

По сути, автомат с магазинной памятью — это несколько «доработанный» обычный конечный автомат. «Доработка» заключается в добавлении к автомату *магазинной памяти*. Слово «магазинный» в данном контексте, разумеется, должно ассоциироваться не с гастрономом, а, скорее,

с патронным магазином «калашникова». В современной терминологии такого рода память называется всем известным (надеюсь) словом *стек*. Таким образом, наша нынешняя задача — рассмотрение автоматов со стековой памятью.

Напомню, что стеком называется хранилище (обычно однородных) данных, для обращения к которому существуют две операции[41] (см. рис. 6.1):

- положить объект данных на вершину стека (операция push);
- взять объект данных с вершины стека (операция pop).

Стек автомата считается бесконечным, поэтому проблемы переполнения не возникает. Но что случится, если в процессе работы автомата произойдет попытка извлечь из пустого стека очередное значение? Что ж, это новый пример исключительной ситуации, аналогичный попытке перейти по несуществующему правилу. Как правило, такое событие должно расцениваться просто как недопускание входной строки.

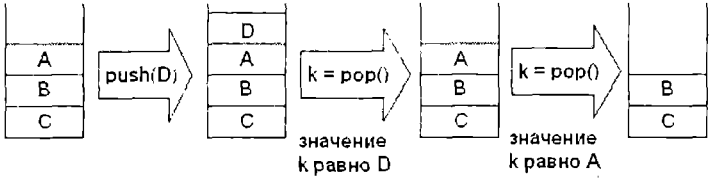


Рис. 6.1. *Принцип работы стека*

Отличия автоматов с магазинной памятью от обычных конечных автоматов

Автоматы с магазинной памятью во многом похожи на обычные конечные автоматы из второй главы. Отличия приведены в списке:

1. Автомат с магазинной памятью включает в себя стек, изначально пустой. В стеке можно хранить элементы, являющиеся элементами «множества стековых символов». Это множество мы выбираем сами исходя из каких-то личных соображений.
2. Правила перехода обычного конечного автомата сопоставляют паре (состояние, символ) какое-то новое состояние. Правила перехода магазинного автомата немного сложнее. Каждой тройке (состояние, символ, стек) теперь сопоставляется пара (состояние', стек'). Смысл компонентов «состояние», «символ» и «состояние'» — тот же, что

[41] Вносить в этот список некоторые дополнения не возбраняется. Магазинный автомат сам использует, строго говоря, чуть-чуть иные средства управления стеком.

и раньше, при этом переходы без считывания очередного символа (ε-переходы) считаются допустимыми. Компонент `стек` определяет, какие символы[42] должны находиться на вершине стека в данный момент (иначе текущее правило попросту не подходит в этой ситуации). При выполнении перехода эти символы снимаются с вершины стека. Компонент стек' правой части правила определяет символы, которые кладутся на вершину стека при выполнении перехода. Так, правило 1, c, a → 2, b означает «если текущее состояние — первое, очередной символ входной ленты равен `c`, а на вершине находится элемент `a`, то снять `a` с вершины стека, перейти в состояние `2` и положить символ `b` на вершину стека». Символ ε в качестве компонента `стек` означает «ничего не снимать со стека» (при этом текущее состояние стека не играет роли; считается, что «пустой символ» присутствует на вершине всегда). Аналогично, ε в роли компонента `стек'` означает «ничего не класть на стек». Обратите внимание, что, вообще говоря, значения компонентов `стек` и `стек'` — строки, а не одиночные символы. При этом первый символ такой строки считается лежащим на вершине стека, а последний, соответственно, — лежащим наиболее «глубоко» (см. рис. 6.2).

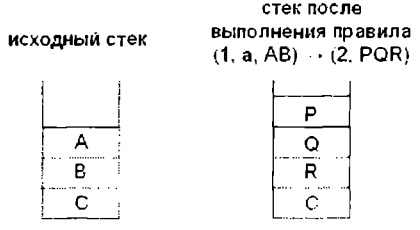


Рис. 6.2. *Модификация стека при выполнении правил перехода*

3. В отличие от обычного конечного автомата, завершение работы автомата с магазинной памятью в допускающем состоянии еще не означает допускания строки. Здесь существует еще второе условие: при завершении работы стек должен быть пустым[43].
4. Магазинный автомат по умолчанию считается недетерминированным. Правила, предлагающие различные действия в одних и тех же ситуациях, допустимы.

[42] Из множества стековых символов.

[43] Но это не есть догма. Можно сконструировать автомат, допускающий строку только «по допускающему состоянию» (после считывания строки автомат находится в допускающем состоянии) или только «по пустому магазину» (после считывания строки стек пуст).

При изображении магазинного автомата в виде графа переходы обычно метят тройками (c, s, e), где c — считываемый символ входной ленты, s — извлекаемая из стека последовательность элементов, а e — последовательность элементов, помещаемая на вершину стека.

Пример автомата, распознающего нерегулярный язык

Самое время рассмотреть автомат, распознающий какой-нибудь нерегулярный язык. Например, язык, строки которого содержат произвольное количество букв a, за которыми следует столько же букв b (см. рис. 6.3).

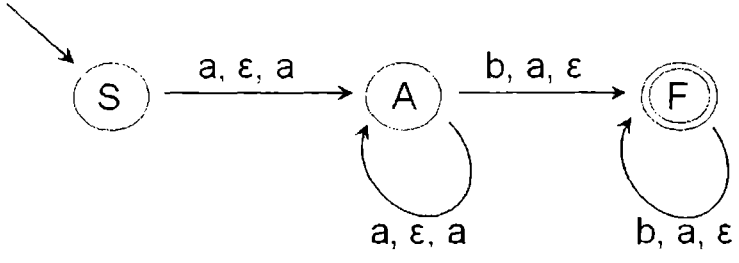


Рис. 6.3. *Магазинный автомат, распознающий нерегулярный язык*

Поскольку это первый законченный магазинный автомат, который мы обсуждаем, я опишу его структуру подробнее.

- Алфавит состоит из двух символов: a и b.
- Стартовое состояние — S.
- Допускающее состояние — F.
- Множество стековых символов содержит те же элементы, что и алфавит.
- Автомат включает четыре правила перехода:

```
S, a, ε → A, a
A, a, ε → A, a
A, b, a → F, ε
F, b, a → F, ε
```

Работает этот автомат очень просто. Допустим, входная строка содержит N символов a, за которыми следуют N символов b. Считывание символов a приведет к тому, что в стеке окажется N элементов, равных a (и больше ничего). Первый же символ b переводит автомат в допускающее состояние, но потребуется еще N — 1 считываний b, чтобы очистить стек. Таким образом, любая строка, содержащая иную конфигурацию символов, не будет допущена.

Довольно легко доказывается (попробуйте!) следующий полезный факт: если разрешить в каждом правиле чтение/запись лишь одного элемента стека, это не отразится на выразительной мощи магазинных автоматов.

Еще одно замечание. Иногда возникает необходимость проверить стек на пустоту. Для такой операции никакой особой возможности не предусмотрено, тем не менее, всегда можно на первом же шаге положить в стек «особый» символ, служащий признаком дна. Затем уже можно проверять элемент вершины стека на равенство этому особому символу.

6.2. Преобразование контекстно-свободной грамматики в магазинный автомат

Построить магазинный автомат на основе существующей контекстно-свободной грамматики несложно:

- Создайте два состояния автомата (других вообще не будет) — стартовое P и допускающее F.
- Создайте правило перехода P, ε, ε → F, S, где S — стартовый символ грамматики.
- Для каждого терминального символа грамматики c создайте правило F, c, c → F, ε.
- Для каждого правила грамматики A → x (где x — обычная правая часть, то есть строка из терминалов и нетерминалов) создайте переход F, ε, A → F, x. Стоит добавить, что стековыми символами такого автомата являются все терминальные и нетерминальные символы грамматики.

На первый взгляд кажется удивительным, что любой контекстно-свободный язык может быть распознан автоматом, содержащим всего лишь два состояния. Однако прежде, чем делать выводы, давайте обсудим алгоритм его работы.

Начать лучше (как всегда) с конкретного примера. Возьмем тот же самый язык

```
S → aSb
S → ε
```

Даже не буду говорить, какого вида строки он описывает! Уже рассмотренный автомат с рис. 6.3 распознает этот язык, но давайте построим теперь новый автомат, пользуясь четырьмя указанными выше правилами.

Итак, создаваемый автомат содержит два состояния — P (стартовое) и F (допускающее). Добавим переход P, ε, ε → F, S (правило 2).

Наша грамматика содержит два терминальных символа — a и b. Следовательно, требуется добавить два перехода (правило 3): F, a, a → F, ε и F, b, b → F, ε.

Далее, правило грамматики S → aSb порождает переход F, ε, S → F, aSb, а правило S → ε — переход F, ε, S → F, ε. В итоге получаем автомат, изображенный на рис. 6.4.

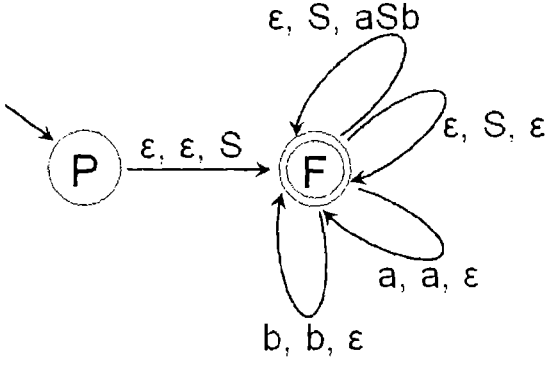


Рис. 6.4. *Готовый автомат*

Конечно, этот автомат сильно отличается от автомата с рис. 6.3, но почему два различных автомата не могут распознавать один и тот же язык? Совсем другое дело, что автомат с рис. 6.3 является детерминированным, в то время как новый автомат использует недетерминированные правила (и это, как мы увидим впоследствии, очень важно).

На всякий случай упомяну явно, что смысл «недетерминированности» магазинного автомата точно такой же, как и в случае с обычным конечным автоматом. Любая возможность поступить двумя или более способами приводит к «развилке» в мирах. Если хотя бы в одном из миров входная строка была допущена, мы считаем ее принадлежащей данному языку.

Кстати, наличие операций на стеке приводит к тому, что использование одинаковых символов входного алфавита (либо знака ε) в левых частях правил еще не означает недетерминированности автомата. Например, переходы P, c, a → F, b и P, c, b → F, k не конфликтуют, поскольку в окончательном выборе перехода участвует не только символ входной строки (c), но и текущая вершина стека, а она разная в каждом из правил. Таким же образом не конфликтуют правила P, ε, a → F, b и P, ε, b → F, k.

Любой построенный по приведенному выше алгоритму автомат работает примерно по тому же сценарию, что был описан при разборе корректности «головы» адреса электронной почты (см. предыдущую главу). Напомню, мы анализировали в основном работу правила H → C"H | C",

выбирая на каждом шаге некоторую альтернативу. Автомат поступает примерно так же. На первом шаге в стек помещается стартовый символ. Затем выполняется алгоритм:

- Если на вершине стека находится терминальный символ, извлечь его из стека и считать очередной символ входной ленты. Если этот «очередной символ» не совпадает с только что извлеченным, строка не допускается.
- Если на вершине стека находится нетерминал, извлечь его из стека, заменить «определением» (то есть заменить терминал правой частью любого подходящего правила) и поместить обратно в стек.

Недетерминированное поведение по своей сути здесь означает выбор той или иной альтернативы на каждом шаге, аналогичный выбору «H → C"H" или H → C"» в предыдущей главе.

Рассмотрим, к примеру, допускание автоматом строки `aabb`.

Сначала в стек помещается символ S. Затем у автомата есть выбор: воспользоваться либо переходом F, ε, S → F, aSb, либо переходом F, ε, S → F, ε (что аналогично выбору одного из правил S → aSb или S → ε грамматики). Предположим, автомат останавливается на первой альтернативе[44]: F, ε, S → F, aSb. Теперь стек содержит последовательность a, S, b (a на вершине), и производится переход F, a, a → F, ε; при этом также считывается первый символ a входной ленты. После перехода стек содержит символы S и b.

Теперь перед автоматом стоит уже знакомый выбор: F, ε, S → F, aSb или F, ε, S → F, ε. Предположим, опять предпочтен вариант первого из этих правил. Таким образом, стек теперь содержит символы a, S, b и b. Автомат второй раз переходит по правилу F, a, a → F, ε. Вторая a входной строки считана, стек содержит символы S, b и b — и мы в третий раз попадаем на «развилку».

Пусть теперь выбран переход F, ε, S → F, ε. В стеке после перехода находятся лишь два символа b. Автомату остается лишь считать оставшуюся часть входной строки (bb) и параллельно опустошить стек. Строка допущена.

Разумеется, любая другая последовательность выбора правил не приводит к положительному результату, но, как уже упоминалось, нас удовлетворит успех даже в каком-либо единственном мире.

[44] Или, в терминах множественных миров, «рассмотрим мир, в котором автомат пошел по пути первой альтернативы».

6.3. Преобразование магазинного автомата в контекстно-свободную грамматику

Пусть заголовок не вводит вас в заблуждение: эта тема рассматриваться в книге не будет. Я лишь хотел сказать, что такой алгоритм существует, и он окончательно устанавливает тождественность между магазинными автоматами и контекстно-свободными грамматиками. Просто его рассмотрение уводит нас в сторону без особой надобности. Алгоритм конвертирования магазинного автомата в грамматику достаточно сложен (с переводом грамматики в автомат не сравнить), а практическая польза от него сомнительна. Интересующимся же советую обратиться к любому серьезному учебнику по теории вычислений. Мы же лучше перейдем к более близким к реальному миру темам.

6.4. Детерминированные и недетерминированные автоматы с магазинной памятью: две большие разницы

Теперь настало время сообщить один весьма неприятный факт. Дело в том, что в отличие от обычных конечных автоматов, детерминированные и недетерминированные автоматы с магазинной памятью имеют *разную выразительную мощь*. С помощью недетерминированного магазинного автомата можно описать любой контекстно-свободный язык. С помощью детерминированного магазинного автомата удастся описать лишь *некоторые* контекстно-свободные языки (хотя и любые регулярные).

Но, быть может, работу недетерминированного магазинного автомата можно просто сымитировать на компьютере? Этот подход для обычного конечного автомата уже рассматривался. К сожалению, в общем случае здесь мы сталкиваемся с новой проблемой.

Представьте себе грамматику, содержащую правило A → Ac. Соответствующий ему переход автомата имеет вид F, ε, A → F, Ac. Что происходит при выполнении такого перехода? С вершины стека снимается символ A (при этом никаких считываний входной ленты не производится), а затем на стек кладутся символы A и c, причем A снова оказывается на вершине. Теперь ничто не мешает автомату снова выбрать этот же самый переход, заменяя A на Ac на вершине стека. «Мгновенный снимок» обычного конечного автомата однозначно определяется указателем его текущего состояния и текущего символа на входной ленте.

Для магазинного автомата не меньшее значение имеет содержимое стека. А так как стек бесконечен, то и различных «снимков» может быть бес-

конечное множество. В главе о конечных автоматах мы строили деревья, отражающие все «развилки», что возникают в процессе анализа строки. Эти деревья могли быть сколь угодно большими, но *конечными*. Теперь же длина каждой ветки теоретически может оказаться бесконечной.

Так как же быть с контекстно-свободными языками? Ну, например, стоит заметить, что использование автомата с магазинной памятью — это не единственный способ распознать контекстно-свободный язык. Такое замечание вполне естественно приводит к вопросу: а зачем тогда вообще изучать магазинные автоматы? На это у меня есть ответ: дело в том, что даже *детерминированные* контекстно-свободные языки (то есть распознаваемые детерминированным магазинным автоматом) интересны сами по себе и заслуживают некоторого внимания. В частности, детерминированными языками ограничиваются при описании синтаксиса языков программирования.

Разумно было бы спросить, а какие именно языки являются детерминированными? Не сочтите за издевательство, но детерминированные языки (по определению) — это языки, распознаваемые детерминированным автоматом с магазинной памятью. При этом автомат должен использовать допускание по допускающему состоянию, то есть содержимое стека после считывания входной строки игнорируется; имеет значение лишь текущее состояние автомата.

Для недетерминированных автоматов критерий допускания не имеет особого значения, но существенно влияет на выразительную мощь автоматов детерминированных. Так, детерминированным автоматам, допускающим по пустому магазину, соответствует более узкий класс распознаваемых языков; автоматам, допускающим по допускающему состоянию — более широкий.

Перед тем, как перейти к детерминированным языкам, я приведу реальный пример языка недетерминированного. Таковым, в частности, является язык, состоящий из строк вида aa^r, где a — любая строка, а a^r — та же строка, только записанная «задом наперед».

6.5. Автоматы с магазинной памятью в JFLAP

Теперь рассмотрим поддержку автоматов с магазинной памятью в системе JFLAP. Нас интересуют два алгоритма: построение контекстно-свободной грамматики по заданному автомату и преобразование автомата с магазинной памятью в контекстно-свободную грамматику.

В JFLAP магазинные автоматы допускают строку исключительно по допускающему состоянию. Иначе говоря, если после считывания строки автомат оказался в допускающем состоянии, строка будет допущена не-

зависимо от содержимого стека. Поэтому автоматы из книги придется немного доработать для использования в JFLAP: мы-то считали, что строка допускается лишь при условии пустого стека. К счастью, доработка совсем проста. В JFLAP изначально стек всегда содержит специальный «признак дна» — символ Z. Поэтому проверка на наличие Z на вершине стека фактически означает проверку на пустоту.

Такая проверка на пустоту всегда используется в автоматах, сгенерированных системой JFLAP для заданной контекстно-свободной грамматики. Рассмотрим, например, построение магазинного автомата для грамматики, задающей все тот же язык {ε, ab, aabb, aaabbb, ...}:

```
S → aSb
S → ε
```

В главном меню JFLAP выберите пункт **Grammar** и введите оба правила грамматики в открывшемся окне редактора. Далее, в меню **Convert** можно выбрать один из двух алгоритмов преобразования: **Convert CFG to PDA (LL)** или **Convert CFG to PDA (LR)**. В любом случае вы получите корректное устройство, но от себя замечу, что в главе был описан LL-алгоритм (то есть, выбрав его, вы сгенерируете автомат, более похожий на тот, что изображен на рис. 6.4).

Для завершения процесса конструирования автомата в открывшемся окне следует нажать кнопку **Show All** (см. рис. 6.5).

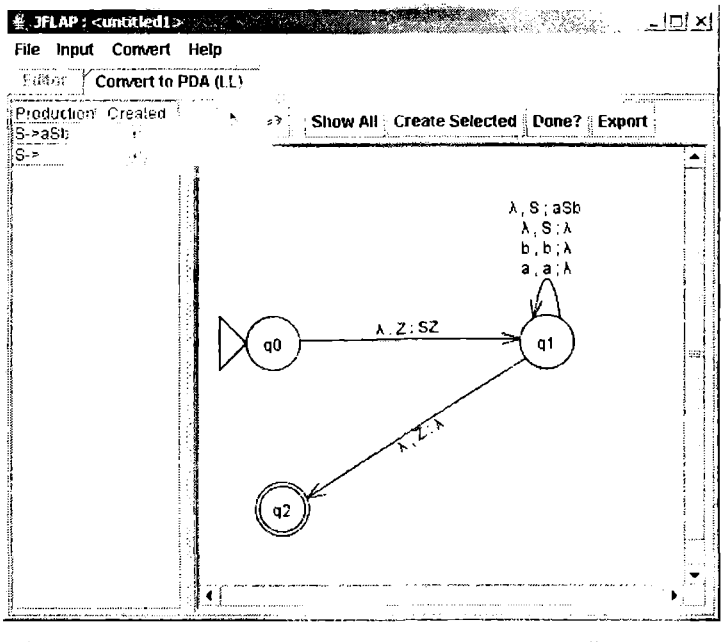


Рис. 6.5. *Конструирование магазинного автомата по заданной грамматике (JFLAP)*

К сожалению, получение грамматики на основе данного автомата уже не столь просто.

Чтобы преобразовать автомат в контекстно-свободную грамматику, первым делом требуется перейти в режим конструирования автоматов с магазинной памятью (пункт **Pushdown Automaton** в главном меню JFLAP) и нарисовать необходимые состояния и переходы.

Конструирование магазинных автоматов не слишком отличается от создания обычных конечных автоматов: просто при добавлении нового перехода вам придется задать не один символ в качестве метки, а три: считываемый с ленты, извлекаемый из стека и помещаемый в стек.

Для генерирования грамматики выберите пункт **Convert → Convert to Grammar.**

Если вы думаете, что теперь с помощью пары щелчков мышью сумеете получить грамматику, то, к сожалению, глубоко заблуждаетесь. Во-первых, ваш автомат должен содержать лишь одно допускающее состояние. Во-вторых, если у вас есть переходы к допускающему состоянию, в которых со стека извлекается хоть что-нибудь отличное от одиночного символа Z, JFLAP не оставит сей факт без внимания, выводя сообщение: «Transitions to final must pop only 'Z'». Таким образом, любой автомат, не удовлетворяющий этому условию, нуждается в доработке. В-третьих, встроенный в JFLAP алгоритм преобразования магазинного автомата в контекстно-свободную грамматику работает лишь для автоматов, в которых при любом переходе:

- со стека снимается ровно один символ;
- на стек кладется либо два символа, либо вообще ничего не кладется.

Вообще говоря, любой автомат можно привести к требуемому виду (см. рис. 6.6, на котором изображен преобразованный автомат с рис. 6.5), но удовольствие от процесса весьма сомнительно.

Хуже другое. Генерируемая грамматика оказывается явно избыточной. В процессе определения правил грамматики (кнопка **Show All**) JFLAP использует временные имена вроде (q1Sq1) или (q3Sq0) для нетерминальных символов. Получить же грамматику в более привычном виде вам, скорее всего, не удастся.

По идее, кнопка **Export** вызывает алгоритм переименования, но, поскольку все нетерминалы представляются заглавными латинскими буквами, их количество не может превышать 26. Таким образом, в большинстве случаев JFLAP попросту откажется экспортировать сгенерированную грамматику, сославшись на нехватку символов.

Разумеется, вы также можете проверить правильность работы любого магазинного автомата с помощью пункта меню **Input → Fast Run**.

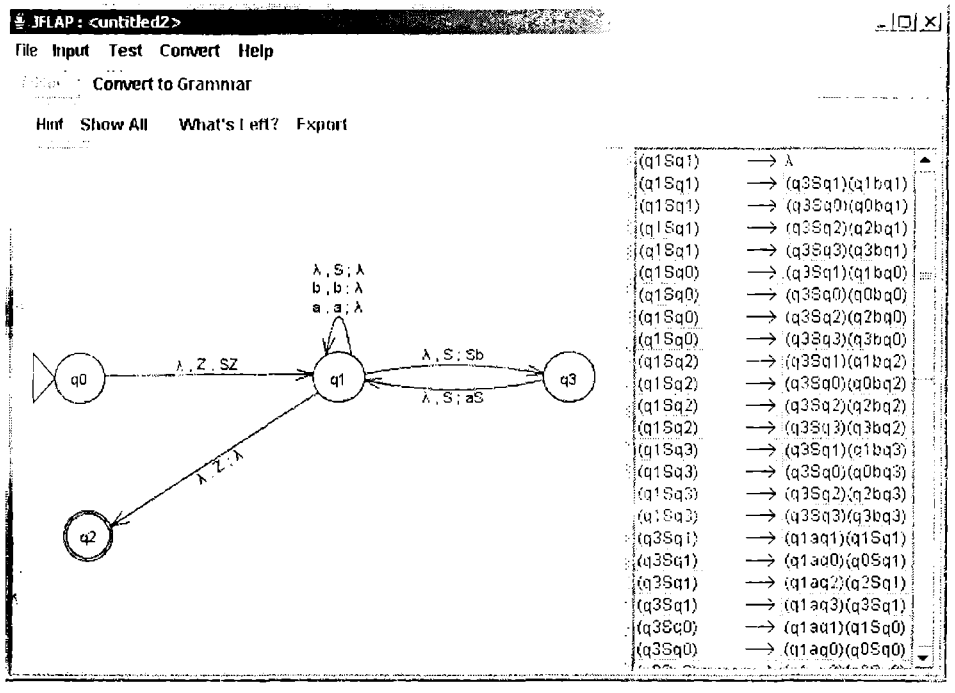


Рис. 6.6. Конструирование грамматики по заданному магазинному автомату (JFLAP)

6.6. Распознавание детерминированных контекстно-свободных языков

Задача распознавания детерминированного контекстно-свободного языка в классической постановке («принадлежит ли данная строка языку») не слишком интересна. Если в случае регулярных языков мы часто можем довольствоваться ответом «да/нет» на вопрос о корректности вещественного числа или адреса электронной почты, то от куда более мощных контекстно-свободных языков мы и ожидаем большего.

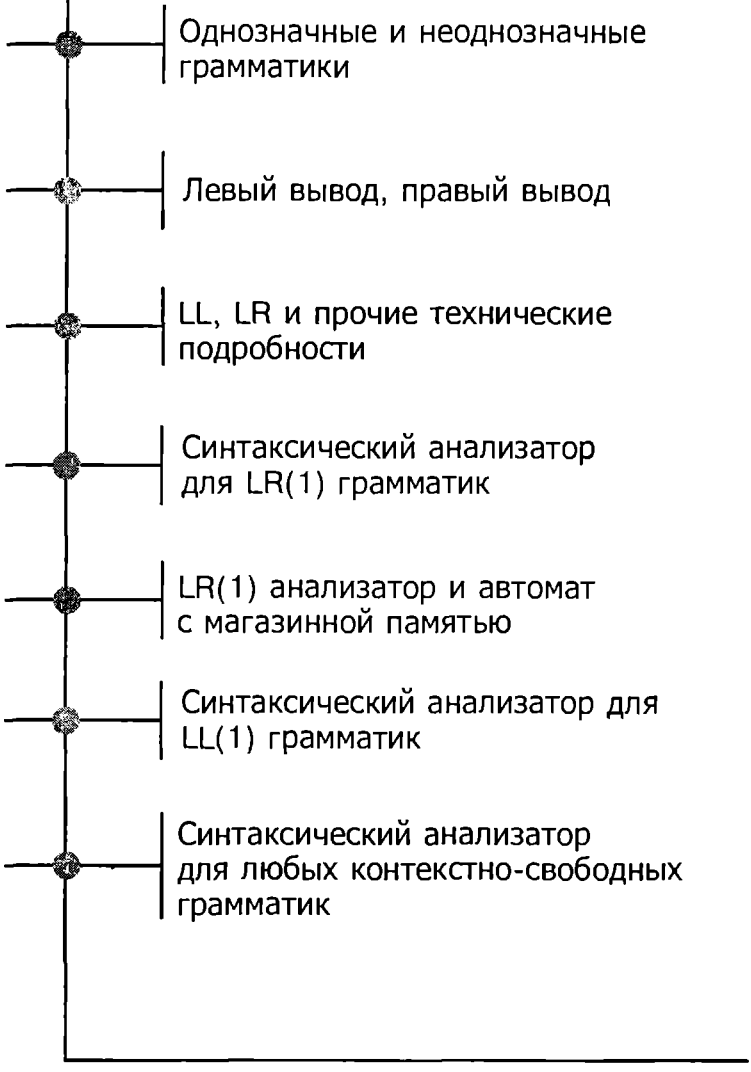
Задача определения принадлежности строки языку, в частности, решается в процессе *синтаксического анализа* (parsing) строки. К сожалению, описать построение магазинного автомата для детерминированного языка невозможно без обсуждения некоторых тем, относящихся, скорее, к задаче синтаксического анализа, а не к автоматам.

Синтаксический анализ же рассматривается в следующей главе. Поэтому давайте пока что отложим тему детерминированных магазинных автоматов (мы еще к ней вернемся) и перейдем к обсуждению задачи синтаксического анализа.

Итоги

- Автоматы с магазинной (стековой) памятью представляют собой устройство, достаточно мощное для распознавания любых контекстно-свободных языков. При этом распознавательная сила магазинных автоматов контекстно-свободными языками и ограничивается.
- Доказательства предыдущего утверждения конструктивны: существуют алгоритмы конвертирования контекстно-свободной грамматики в магазинный автомат и наоборот.
- Недетерминированные магазинные автоматы довольно трудно эмулировать на обычном компьютере.
- Детерминированные магазинные автоматы способны распознать лишь некоторое подмножество контекстно-свободных языков. Языки этого подмножества называются детерминированными (при условии, что магазинный автомат принимает строку по допускающему состоянию).
- Детерминированные магазинные автоматы можно (и нужно!) выполнять на компьютерах. Только сначала необходимо разобраться, как построить автомат на основе данной грамматики. Кроме того, неясно, какие грамматики описывают детерминированные языки, а какие — нет. Выяснением этих вопросов мы займемся в следующей главе.

Глава 7 Синтаксический анализ



Вероятно, эта глава будет самой трудной для чтения. Я постарался как можно меньше углубляться в теоретические и технические подробности, но, как ни крути, без минимальных (но порою громоздких) построений не обойтись. Впрочем, почитав любую сколько-нибудь серьезную книгу по теории компиляции, вы убедитесь, что я затрагиваю лишь самую вершину этого необъятного айсберга[45].

Итак, в прошлой главе мы остановились на том, что с контекстно-свободных языков совершенно иной спрос, чем с языков регулярных. Стандартное (хотя и не единственное) применение контекстно-свободных языков — описание синтаксиса языков программирования. Разумеется, используя грамматику для описания языка программирования, мы рассчитываем не только на привычную по предыдущим главам «да/нет» процедуру, определяющую синтаксическую корректность программы («принадлежит ли данная строка языку корректных программ на Паскале»), но и на нечто большее. А именно: мы должны иметь возможность определить тип той или иной синтаксической конструкции, обнаруженной в программе, и предпринять в каждом случае какие-либо действия.

Таким образом, для контекстно-свободных языков задача распознавания (в классической «да/нет» постановке) обычно не рассматривается. Вместо этого ставится *задача синтаксического анализа*, решением которой является построенное *дерево грамматического разбора* (parse tree).

7.1. Однозначные и неоднозначные грамматики

Вернемся к примеру, с которого мы начинали знакомство с грамматиками:

```
S → A_N
A → умный | вежливый | глупый | гордый | богатый
N → кот | бурундук | пес | селезень
```

Как узнать, что строка `умный_кот` принадлежит языку, описываемому этой грамматикой? Например, можно рассуждать так. Любая строка язы-

[45] Классические монографии как по теории, так и по практике компиляции нередко содержат 700-1000 страниц...

ка (S) есть A_N. Но что такое A? В частности, «умный», следовательно, S → умный_N. А что такое N? В частности, «кот», следовательно, S → умный_кот. Получается такая цепочка вывода:

```
S → A_N → умный_N → умный_кот
```

Этот вывод можно изобразить в виде дерева, называемого деревом грамматического (или синтаксического) разбора (см. рис. 7.1).

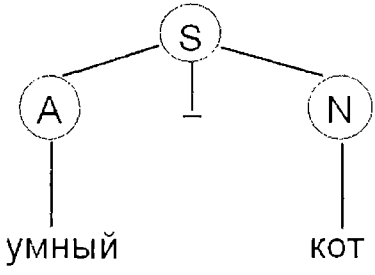


Рис. 7.1. *Дерево разбора строки умный_кот*

Любой узел дерева, помеченный нетерминальным символом, имеет некоторое количество потомков, раскрывающих суть примененного правила. Три потомка узла S указывают, что было использовано правило S → A_N. Аналогично, отношение «предок/потомок» между узлами A и `умный` сигнализирует о правиле A → умный.

В процессе вывода строки `умный_кот` мы сначала заменили левую переменную правила S → A_N (то есть A) на терминал `умный`, а потом перешли к переменной N. Разумеется, можно поступить наоборот: сначала подставить слово `кот` на место переменной N, а уже потом заняться переменной A. В этом случае цепочка вывода будет иметь несколько иной вид:

```
S → A_N → A_кот → умный_кот
```

Тем не менее, на конфигурацию дерева грамматического разбора порядок применения правил никак не влияет. Однако это свойство выполняется далеко не для всех грамматик. Рассмотрим немного неуклюжую грамматику, описывающую простой язык a*b*:

```
S → A | B | AB
A → aA | ε
B → bB | ε
```

Как убедиться, что строка `aa` является корректной строкой языка a*b*? Например, построив цепочку

```
S → A → aA → aaA → aa
```

Ничем не хуже и цепочка

```
S → AB → aAB → aaAB → aaB → aa
```

Соответствующие им деревья грамматического разбора показаны на рис. 7.2.

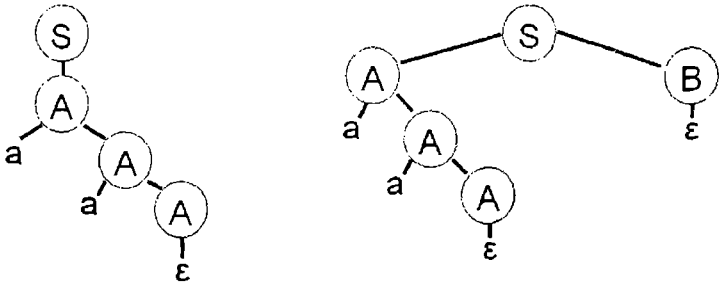


Рис. 7.2. *Различные деревья разбора строки aa*

Такое положение вещей (называемое *неоднозначностью* (ambiguity) *грамматики*), вообще говоря, ни к чему хорошему не приводит. На нынешнем этапе проблемы, вероятно, еще не видны, но давайте заглянем чуть-чуть глубже. Допустим, в некоторой части описания (с помощью формальной грамматики) синтаксиса языка программирования встречается запись

```
КОНСТРУКЦИЯ → ОПЕРАЦИЯ | КОММЕНТАРИЙ | КЛЮЧЕВОЕ_СЛОВО
```

Что будет, если в процессе синтаксического анализа текста программы в некоторой ситуации удастся с одинаковым успехом применить как правило КОНСТРУКЦИЯ → ОПЕРАЦИЯ, так и правило КОНСТРУКЦИЯ → КОММЕНТАРИЙ? Такой исход означает, что компилятор не в состоянии определить смысл текущего элемента: является ли он операцией или началом комментария. Разумеется, разработчики реальных языков программирования пытаются изобретать синтаксис, исключающий возникновение подобных ситуаций, но все же от возможной неоднозначности никто не застрахован. Вот классический пример для языка C:

```
// total  -  целая переменная,
// n_elements_ptr  -  указатель на целое
int result = total/*n_elements_ptr;  // разделить total
                                     // на *n_elements_ptr
```

Проблема этого фрагмента в том, что запись `total/*n_elements_ptr` может быть распознана не только как деление числа на значение, адресуемое указателем, но и как запись переменной `total`, за которой следует начало комментария `/*`. В языке C последовательность /* всегда считается началом комментария, поэтому компиляция фрагмента закончится

неудачей (было бы хуже, если бы все откомпилировалось, но вы бы получили совсем не то, чего хотели). К счастью, если вы используете не самый древний редактор, подсветка синтаксиса моментально подскажет ошибку. Исправить код лучше всего с помощью скобок:

```
int result = total/(*n_elements_ptr); // разделить total
                                      // на *n_elements_ptr
```

Аналогичное затруднение возникает и в следующем фрагменте кода на C++:

```
#include <vector>
using namespace std;
struct MyType {};
...
vector<vector<MyType>> c;
```

Что такое `MyType>> c`? Либо правая часть записи вложенного шаблона, либо попытка вызова операции побитового сдвига вправо (`a >> b`). В приведенной ситуации среди компиляторов уже нет прежнего единодушия. Так, компилятор gcc выдает ошибку, разумно предлагая заменить `>>` на `> >` (то есть вставить пробел между знаками):

```
error: '>>' should be '> >' within a nested template argument list
```

Мой любимый Borland C++Builder поступает более либерально, выдавая лишь предупреждение:

```
[C++ Warning]: Use '> >' for nested templates Instead of '>>'
```

Компилятор великодушно прощает мне эту неточность, лишь мягко, по-дружески указывая на мою неправоту. Забавно, что в третьей версии C++Builder именно так все и происходило, а вот в шестой версии фрагмент уже перестал компилироваться, хотя предупреждающее сообщение осталось без изменений[46].

Неоднозначность — это, как правило, проблема используемой грамматики, а не описываемого языка. Так, язык a*b* можно описать и однозначной грамматикой:

```
S → aS | bA | ε
A → bA | ε
```

Как бы вы ни старались, вам не удастся построить два или более разных дерева разбора для любой конкретной строки языка при использовании этой грамматики. С другой стороны, существуют языки, неоднозначные по своей природе. Классическим примером служит язык, состоящий из всех строк вида $a^n b^n c^m d^m$ и строк вида $a^n b^m c^m d^n$, где запись вида s^k озна-

[46] Вероятно, разработчики просто забыли изменить статус сообщения с Warning на Error.

чает строку из k символов s (ss...s — всего k раз). Этот язык является контекстно-свободным. Для него существуют различные грамматики, в частности, такая:

```
S → AB | C
A → aAb | ab
B → cBd | cd
C → aCd | aDd
D → bDc | bc
```

Неоднозначность проявляется, к примеру, при анализе строки aabbccdd. К какому виду она принадлежит? К $a^nb^nc^md^m$ или к $a^nb^mc^md^n$ ($n = m = 2$)? Языки, для которых нельзя построить однозначную грамматику, называются *существенно неоднозначными*[47] (inherently ambiguous). К сожалению, нет общего алгоритма, который бы мог выяснить, является ли данный язык существенно неоднозначным или нет. Более того, в общем случае нельзя определить, является ли данная грамматика неоднозначной; также нельзя автоматически получить однозначную грамматику по данной неоднозначной, даже если известно, что язык, описываемый ею, не является существенно неоднозначным.

Несмотря на эти разочаровывающие подробности, на практике дела обстоят не так плохо. Во многих случаях можно наложить на грамматики некоторые ограничения и тем самым упростить задачу. Конечно, ограниченные грамматики не сумеют описать любой контекстно-свободный язык, но это не трагедия. Мы же с успехом используем регулярные выражения, хотя они описывают лишь регулярные языки.

7.2. Левый вывод, правый вывод...

Пора перейти от скорее теоретических вопросов к весьма своеобразной смеси теории и практики. С точки зрения теории мы уже рассмотрели контекстно-свободные грамматики и их основные свойства. Теперь разберем первый по-настоящему практический вопрос. Рассмотрим грамматику, описывающую простые арифметические выражения:

```
E → EOE | 0 | 1 | ... | 9
O → + | - | * | / | =
```

Выражение (E) — это <выражение><операция><выражение> (EOE), либо отдельная цифра. Операция (O) — это знак любого из четырех арифметических действий, либо знак «равно». Как убедиться, что строка

[47] Слово «существенно» в данном контексте следует понимать как «по своему существу», «по своей сути». Термин, прямо скажем, не особо удачный... «Существенно неоднозначный» термин. Чем-то напоминает «вполне непрерывные функции» в математике.

+2=4 принадлежит языку, описываемому грамматикой? Например, с омощью цепочки

```
E → EOE → 2OE → 2+E → 2+EOE → 2+2OE → 2+2=E → 2+2=4
```

от же результат можно получить и с помощью другой последовательно- ги применения правил (что доказывает неоднозначность грамматики):

```
E → EOE → EO4 → E=4 → EOE=4 → EO2=4 → E+2=4 → 2+2=4
```

еревья грамматического разбора для обоих случаев показаны на ис. 7.3.

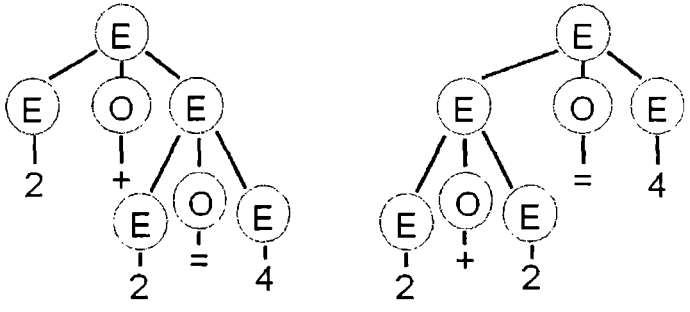


Рис. 7.3. *Деревья разбора строки 2+2=4*

братите внимание, что в каждой последовательности вывода использу- ся своя негласная договоренность. В первом случае я всегда «раскры- ю» самый левый нетерминал. Так, после применения правила E → EOE есто левого E подставляется его определение. Затем к полученному ражению 2OE применяется то же действие: на сей раз самым левым терминалом будет O.

) втором случае работает тот же принцип, но теперь в первую очередь рабатывается самый правый нетерминал.

хнология с заменой самого левого нетерминала на каждом шаге назы- ется *левым выводом*[48] (leftmost derivation); замена же самого правого не- рминала, соответственно, — *правым выводом* (rightmost derivation).

специально взял неоднозначную грамматику, чтобы сделать очевид- ми отличия левого и правого выводов. Разумеется, если грамматика нозначна, оба дерева окажутся одинаковыми, различным будет лишь оцесс их получения. На практике любой синтаксический анализатор лжен использовать какое-то конкретное правило, а не раскрывать

[48] Ну, положим, сочетание «левый вывод» у меня тоже ассоциируется с чем-то совершенно не связанным с грамматиками. Впрочем, хватит критиковать.

случайно выбранный нетерминал на каждом шаге. Быть может, пока что последствия того или иного выбора кажутся несущественными (действительно, какая разница, левый или правый нетерминал раскрывать в первую очередь?), однако на самом деле это не так, в чем мы скоро убедимся.

7.3. LL, LR и прочие технические подробности

Типы синтаксических анализаторов (парсеров)

Теперь давайте полностью переключимся на задачу синтаксического разбора со стороны программ (пока что абстрактных), на практике выполняющих разбор. Первое предположение, которое делается для таких программ, касается просмотра анализируемой строки. Считается, что строка просматривается слева направо (достаточно разумное предположение для западных цивилизаций[49]). Далее, предполагается, что анализатор всегда использует либо левый, либо правый вывод при работе с грамматиками. Наконец, считается, что анализатору достаточно знать не всю входную строку, а лишь подстроку из k символов, начиная с текущего.

Исходя из этих предположений, можно выделить LL(k) и LR(k) анализаторы[50]. Запись LL(k) означает «просмотр слева направо, левый вывод, k символов предпросмотра» (**L**eft to right, **L**eftmost derivation, **k** symbols lookahead). Аналогично, запись LR(k) читается как «просмотр слева направо, правый вывод, k символов предпросмотра» (**L**eft to right, **R**ightmost derivation, **k** symbols lookahead).

Практические аспекты

Определившись с типами синтаксических анализаторов, приходится решать новые, теперь уже вовсе не теоретические, а чисто практические, насущные вопросы:

- Какова должна быть контекстно-свободная грамматика, чтобы для нее удалось построить LR(k) (ну или LL(k)) анализатор?

[49] Где-то описывался случай, когда западная фармацевтическая фирма разместила в одном арабском государстве свои стандартные рекламные щиты. На каждом щите были изображены (слева направо): лежащий в постели больной человек, этот же человек, принимающий лекарство, и, наконец, он же, но уже здоровый. Арабы, воспринимая информацию естественным для них образом (справа налево), вероятно, приходили в ужас. Рекламная кампания фирме успеха не принесла.

[50] Синтаксический анализатор в наши дни все чаще называют англоязычным термином *парсер* (parser).

- Какова должна быть контекстно-свободная грамматика, чтобы анализатор оказался максимально простым и быстрым, не требующим глубокого предпросмотра (LR(1) или LL(1))?
- Как на практике построить синтаксический анализатор?
- Можно ли построить синтаксический анализатор, умеющий работать с любыми контекстно-свободными языками?
- Какое отношение все это имеет к автоматам с магазинной памятью?

Чтобы не отходить слишком далеко от основных тем книги (в конце концов, мы же не теорией компиляции занимаемся), я сразу же сделаю несколько замечаний.

Грамматики, для которых существует LR(k) анализатор, называются LR(k) грамматиками. Соответственно, если для некоторой грамматики существует LL(k) анализатор, она является LL(k) грамматикой.

Хотя на первый взгляд LL(k) и LR(k) грамматики кажутся практически одинаковыми, на самом деле это не так. Даже LR(1) грамматика может описать *любой* детерминированный язык. В то же время ни одна LL(k) грамматика (при любом конечном k) не в состоянии сделать этого. Однако увеличение значения параметра k для LR-грамматик не приводит к усилению выразительности. Любые LR(2), LR(3) и т. д. грамматики точно так же описывают детерминированные языки, как и LR(1). К сожалению, не существует алгоритма, по данной LL- или LR-грамматике строящего LR(1) грамматику, которая описывала бы тот же самый язык.

Эти результаты очень важны, поэтому я повторю их в таблице:

Грамматика	Язык
произвольная контекстно-свободная	любой контекстно-свободный
LR(k) для любого конечного k ≥ 1	любой детерминированный
LL(k)	некоторое подмножество детерминированных языков, зависящее от k

Стоит отметить, что мощность LL(k) грамматики растет с увеличением k. Так, синтаксис языка C слишком сложен, чтобы его можно было бы описать с помощью LL(1) грамматики. Однако возможностей LL(2) грамматики для описания языка C уже вполне достаточно.

Следующая часть главы будет посвящена только самым полезным (на мой предвзятый взгляд) задачам и алгоритмам. Во-первых, мы будем заниматься только LL(1) и LR(1) грамматиками, а также контекстно-свободными грамматиками «общего вида». LL(1) грамматики интересны тем, что для них существует простой и элегантный алгоритм построения синтаксического анализатора.

Поскольку LR(1) грамматики могут описать любой детерминированный язык, их связь с детерминированными магазинными автоматами очень важна. Изучив этот вопрос, мы сможем построить синтаксический анализатор и для LR(1) грамматик. Наконец, общий алгоритм разбора любой контекстно-свободной грамматики я просто не имею морального права пропустить. Сразу скажу, почему им не пользуются во всех ситуациях: он медлителен, и если есть возможность описать язык с помощью LL/LR грамматик, гораздо лучше применить специализированный (быстрый) алгоритм разбора.

7.4. Синтаксический анализатор для LR(1) грамматик

Чтобы окончательно не забыть о магазинных автоматах в разговорах о синтаксических анализаторах, давайте начнем с LR(1) грамматик. Помните регулярные грамматики? Наложив нехитрые ограничения на используемые правила, мы легко получили класс грамматик с интересующими нас возможностями. Так вот, к сожалению (в который раз я вынужден это говорить!), для LL(1) и LR(1) грамматик такого простого решения не существует. Можно запрограммировать алгоритм, который бы анализировал некоторую грамматику и сообщал, обладает ли она LL(1) или LR(1) свойствами. Но нет метода, который позволил бы после первого же взгляда на грамматику сказать что-то вроде: «ага! здесь применено такое-то правило, значит, грамматика не принадлежит типу LR(1)».

На практике я бы просто посоветовал при разработке языка (если вы сочиняете грамматику для собственного языка) не закладывать «часовых мин» вроде последовательностей /* или >> в языке C++. Если же грамматика известна заранее, ее можно просто скормить синтаксическому анализатору, и сообщения о любых трудностях будут напечатаны вам незамедлительно.

Должен предупредить, что программа LR(1) анализа, по-видимому, самая сложная во всей главе (а глава — самая сложная в книге). Поэтому наберитесь терпения (или читайте по диагонали).

7.4.1. Предварительные замечания

Перед тем, как писать программу, давайте договоримся о входных данных. Синтаксический анализатор будет считывать с консоли LR(1) грамматику, а также анализируемую строку. На выходе ожидается либо сообщение о том, что грамматика не принадлежит виду LR(1), либо, если грамматика пригодна для разбора, результат анализа строки. В случае отрицательного

ответа (строка не принадлежит языку, описываемому грамматикой) достаточно просто сообщить об этом; если же строка разобрана, придется вывести какую-то информацию, позволяющую получить полное дерево разбора.

Грамматика должна быть задана в обычном виде, только вместо символа → будет использоваться пробел:

```
S Ac
A aA
```

Чтобы упростить работу с грамматикой, синтаксис вариантов, разделенных вертикальной чертой, не поддерживается. Так, вместо A → a | b | Ac следует писать

```
A a
A b
A Ac
```

Нетерминалы (все-таки укажу это явно) будут обозначаться заглавными латинскими буквами, а терминалы — буквами строчными. Кроме того, в качестве терминалов допускается использовать символы операций («плюс», «минус» и т. п.) Стартовым символом грамматики будет считаться буква S (независимо от того, какое правило указано первым в списке).

Следующее замечание относится к ε-правилам, то есть правилам вида A → ε. Во-первых, вместо символа ε (которого нет на клавиатуре), я буду использовать строчную русскую букву «э» (первая в слове «эпсилон»). Во-вторых, этот символ может появляться только в ε-правилах (в других контекстах его использование лишено смысла).

Последняя ремарка связана со входной строкой. Синтаксический анализатор должен быть в состоянии определить ситуацию, когда все символы строки уже считаны; то есть, попросту говоря, строку требуется закончить особым символом — «признаком конца». Традиционно при описании LR-анализаторов для этой цели используют символ доллара ($).

Разработку программы можно разделить на следующие этапы:

- Считывание грамматики с консоли. Самый простой, но все-таки заслуживающий внимания этап.
- Удаление из грамматики ε-правил. Позволяет упростить вид используемой грамматики, а следовательно, и конструкцию самого анализатора.
- Программирование процедуры синтаксического анализа. Для работы анализатору потребуются дополнительные таблицы данных, задачу генерации которых мы также рассмотрим.

7.4.2. Считывание грамматики

Правила грамматики, терминальные и нетерминальные символы будут храниться в соответствующих переменных:

```
static ArrayList Grammar = new ArrayList();
                           // правила грамматики
static string Terminals;   // список терминалов
static string Nonterminals; // список нетерминалов
```

Процедура считывания грамматики достаточно проста:

```
static void ReadGrammar()
{
    string s;
    Hashtable term = new Hashtable();
                           // временная таблица терминалов
    Hashtable nonterm = new Hashtable();  // и нетерминалов

    while((s = Console.In.ReadLine()) != "")
                           // считывание правил
    {
        Grammar.Add(s);    // добавить правило в грамматику

        for(int i = 0; i < s.Length; i++)
                           // анализ элементов правила
            if(s[i] != ' ')
            {
                           // если текущий символ - терминал,
                           // еще не добавленный в term
                if(s[i] == s.ToLower()[i] && !term.ContainsKey(s[i]))
                    term.Add(s[i], null);

                // аналогично для нетерминалов
                if(s[i] != s.ToLower()[i] && !nonterm.ContainsKey(s[i]))
                    nonterm.Add(s[i], null);
            }
    }

    // переписываем терминалы и нетерминалы
    // в строки Terminals и Nonterminals
    for(IDictionaryEnumerator c = term.GetEnumerator(); c.MoveNext();)
        Terminals += (char)c.Key;
    for(IDictionaryEnumerator c = nonterm.GetEnumerator();
        c.MoveNext();)
        Nonterminals += (char)c.Key;

}
```

На каждой итерации в грамматику добавляется очередное считанное равило. Символы грамматики (как терминалы, так и нетерминалы) по-ещаются сначала в хэш-таблицы в качестве ключей (пользуемся тем, что лючи уникальны, поэтому два раза один и тот же символ в таблицу не удет записан). В последнюю очередь формируются строки терминалов erminals и нетерминалов Nonterminals.

7.4.3. Удаление ε-правил

алее, из входной грамматики следует исключить ε-правила, то есть равила вида A → ε. Не могу сказать, что наличие ε-правил фатально, днако необходимость их обработки серьезно усложняет конструкцию R-парсера. Удалить же ε-правила несложно:

```
ПОКА существуют ε-правила
    найти правило вида A → ε;
    для любого правила, содержащего A в правой части (X → αAβ)
        добавить аналогичное правило, но без A: X → αβ;
    удалить правило A → ε;
КОНЕЦ ЦИКЛА
```

десь α и β — любые строки, в том числе пустые. Например, грамма-ику

```
S → Ac
A → aA
A → ε
```

ожно преобразовать с помощью приведенного алгоритма к виду

```
S → Ac
S → c
A → aA
A → a
```

начала добавляются правила S → c и A → a, затем исключается строка → ε. Исключение одних ε-правил может привести к образованию других. апример, если грамматика содержит правило X → N, при исключении ывода N → ε к ней придется добавить правило X → ε. Если выполнение горитма исключения ε-правил приводит к таким результатам, придется овторить его еще раз (а, быть может, и несколько раз). Поэтому внешний икл и выполняется до тех пор, пока существуют ε-правила.

сли нетерминал, участвующий в правиле A → ε, встречается в правой асти некоторого вывода несколько раз, придется добавить по одному овому правилу для каждой возможной комбинации, где те или иные сим-олы A вычеркнуты. Так, при исключении правила A → ε, грамматика

```
S → aAbAc
A → ε
```

превращается в

```
S → aAbAc | abAc | aAbc | abc
```

Таким образом, если нетерминал встречается N раз в правой части вывода, вы получите $2^N - 1$ новых правил.

Единственная сложность связана с правилом S → ε, где S — стартовый символ грамматики (если такое правило существует изначально или возникает в процессе удаления ε-выводов). Удалив правило S → ε, мы упускаем из виду, что пустая строка принадлежит описываемому языку. С точки зрения практического программирования, конечно, проблемы в этом нет никакой. Если в определенный момент времени процедура исключения ε-правил натыкается на правило S → ε, она просто должна сделать пометку: «пустые строки допускаются». А уж для того, чтобы определить, что данная строка — пустая, особого синтаксического анализа и не требуется.

Перейдем к разработке. Быть может, в данном конкретном случае можно запрограммировать более специализированную процедуру, но у нас уже есть готовое решение для генерации различных комбинаций нуля и единицы. Небольшая модификация позволит применить эту процедуру и сейчас:

```
// список найденных комбинаций
static ArrayList combinations = new ArrayList();

static void GenerateCombinations(int depth, string s)
{
    if(depth == 0)
        combinations.Add(s);
    else
    {
        GenerateCombinations(depth - 1, "0" + s);
        GenerateCombinations(depth - 1, "1" + s);
    }
}
```

Громоздкая служебная функция GenerateRulesWithout(A) создает список правил, в которых вычеркнут один или более символов A в правой части:

```
static ArrayList GenerateRulesWithout(char A)
{
    ArrayList result = new ArrayList();     // итоговый список
```

```
// цикл по правилам
for(IEnumerator rule = Grammar.GetEnumerator(); rule.MoveNext();)
{
    string current = (string)rule.Current;
                        // текущее правило,
    string rhs = current.Substring(2);
                        // его правая часть,
    string[] rhs_split = rhs.Split(A);
                        // отдельные сегменты rhs
    int counter;

    if(rhs.IndexOf(A) != -1)
                        // если правая часть содержит A
    {
        counter = 0;        // подсчитываем количество
                            // вхождений A
        for(int i = 0; i < rhs.Length; i++)
            if(rhs[i] == A)
                counter++;

        combinations.Clear();
        GenerateCombinations(counter, "");
                            // генерация комбинаций

        for(IEnumerator element = combinations.GetEnumerator();
                                    element.MoveNext();)
        if(((string)element.Current).IndexOf('1') != -1)
        {
             // если текущая комбинация содержит хоть один
             // вычеркиваемый символ (т.е. единицу)

            string combination = (string)element.Current;
            string this_rhs = rhs_split[0];

             // если текущий символ комбинации - единица
             // (вычеркиваем A), просто соединяем сегменты
             // правой части правила, иначе вставляем
             // дополнительный символ A
            for(int i = 0; i < combination.Length; i++)
                this_rhs += (combination[i] == '0' ?
                            A.ToString() : "") +
                            rhs_split[i + 1];

            result.Add(current[0] + " " + this_rhs);
```

```
            }
        }
    }

    return result;
}
```

Главная же процедура удаления ε-правил теперь оказывается довольно простой и прямолинейной:

```
static bool AcceptEmptyString; // допускать ли пустую строку

static void RemoveEpsilonRules()
{
    AcceptEmptyString = false;
    bool EpsilonRulesExist;

    do
    {
        EpsilonRulesExist = false;
        for(IEnumerator rule = Grammar.GetEnumerator();
                                rule.MoveNext();)
            if(((string)rule.Current)[2] == 'э')
                                // нашли эпсилон-правило
            {
                // принимаем пустую строку, если левая часть
                // правила содержит стартовый символ
                char A = ((string)rule.Current)[0];
                if(A == 'S')
                    AcceptEmptyString = true;

                // добавляем все новые правила для нетерминала A
                Grammar.AddRange(GenerateRulesWithout(A));
                Grammar.Remove(rule.Current);
                                // удаляем эпсилон-правило
                EpsilonRulesExist = true;
                break;
            }
    }
    while(EpsilonRulesExist);
                // пока существуют эпсилон-правила
}
```

Как уже указывалось выше, случай пустой строки будет рассматриваться отдельно. Поэтому переменная `AcceptEmptyString` содержит флаг принадлежности пустой строки описываемому языку.

7.4.4. Синтаксический анализ

Таблицы ACTION и GOTO

Следующий шаг после удаления ε-правил — создание таблиц ACTION и GOTO (вместе составляющих так называемую LR(1)-таблицу). Вообще говоря, это не задача синтаксического анализатора. Синтаксический анализатор лишь распознает входную строку с помощью заранее сгенерированных таблиц.

Генерация же самих таблиц ACTION и GOTO — цель работы программы, обычно называемой «генератором парсеров» (parser generator) или «компилятором компиляторов» (compiler compiler). Идеологически LR(1) анализатор представляет собой автомат с магазинной памятью. Так вот, таблицы ACTION и GOTO описывают конструкцию автомата (иными словами, распознаваемый язык), а программа, имитирующая его работу с помощью готовых таблиц, и есть парсер как таковой.

Традиционно в литературе сначала описывают, как работает синтаксический анализатор «сам по себе» (то есть при наличии готовых таблиц ACTION и GOTO). Затем уже разбирают задачу создания этих таблиц. Вероятно, нарушать традицию не стоит; кроме того, довольно трудно говорить о создании таблиц ACTION и GOTO (а процедура, скажем прямо, довольно сложная, см. пункт 7.4.5), не объяснив даже их назначения.

Таблица ACTION сопоставляет паре (состояние, символ) одну из трех команд:

- shift (состояние')
- reduce (правило грамматики)
- accept

Элемент `состояние` в паре (состояние, символ) означает номер состояния магазинного автомата (будем считать, что состояния обозначены числами). Элемент `символ` — это некоторый терминальный символ грамматики.

Таблица GOTO сопоставляет такой же паре (состояние, символ) (только на этот раз `символ` будет нетерминалом) номер некоторого состояния автомата:

```
GOTO[состояние, символ] = состояние'
```

Ситуация, когда некоторой паре (состояние, символ) вообще ничего не сопоставлено (клетка таблицы пуста), тоже не исключена.

Процедура синтаксического анализа

Перед тем, как запускать процесс синтаксического анализа, в грамматику вносится еще два простых изменения:

- Добавляется новый нетерминал П, который теперь будет стартовым символом грамматики, и создается правило П → S.
- Добавляется новый терминал $ (признак конца).

Теперь самое время проверить, пуста ли входная строка. Если да, то решение (допустить или не допустить) принимается на основе переменной `AcceptEmptyString`. Если же строка непуста, придется выполнить полный анализ.

Процесс анализа представляет собой имитацию работы немного доработанного магазинного автомата:

```
поместить в стек номер стартового состояния автомата;
ЦИКЛ
    s = число на вершине стека;
    a = текущий символ входной строки;
    action = ACTION[s, a];

    ЕСЛИ action = shift s'
        поместить в стек a;
        поместить в стек s';
        перейти к следующему символу входной строки;

    ЕСЛИ action = reduce "A α"
        извлечь из стека 2 * длина(α) элементов;
        s' = число на вершине стека;
        поместить в стек A;
        поместить в стек значение GOTO[s', A];
        вывести на экран правило "A -> α";

    ЕСЛИ action = accept
        сообщить о допускании строки, выйти из цикла;
КОНЕЦ ЦИКЛА
```

Думаю, пара комментариев не помешает. Первое, что бросается в глаза — это отсутствие выхода из цикла, кроме как по команде `accept`. Второй случай выхода происходит при генерации исключения в строке

```
action = ACTION[s, a]
```

если элемент таблицы (s, a) не существует. Возникновение исключения расценивается как «неуспех» (то есть входная строка не допущена), о

чем и следует сообщить пользователю. Вероятно, считать непринятие строки автоматом «исключительной ситуацией» идеологически не совсем правильно, но в данном случае упрощает программирование.

Команды таблицы ACTION имеют и устоявшиеся русские названия. Так, команду `shift` в русскоязычной литературе именуют «сдвигом», а `reduce` — «сверткой». Команду `accept`, на мой взгляд, лучше всего перевести как «допуск» (хотя мне встречались и другие варианты, например, «ввод» или даже «прием»).

Параметром команды `reduce` является любое корректное правило грамматики. В псевдокоде с помощью символа α обозначена строка, являющаяся его правой частью. Таким образом, длина(α) — это просто длина строки в символах. Команда reduce особенно интересна тем, что при ее выполнении делается вывод о применении того или иного правила грамматики. Печатая очередное используемое правило, можно полностью восстановить дерево синтаксического разбора, а ведь именно в его определении и состоит работа парсера.

Программирование синтаксического анализа

Что ж, пора от псевдокода перейти к реальному коду. Для начала нам потребуется структура, упрощающая доступ к хэш-таблицам[51] ACTION и GOTO:

```
struct Tablekey   // ключ таблиц ACTION и GOTO
{
    public int I;
    public char J;

    public Tablekey(int i, char j) {I = i; J = j;}
}
```

Теперь в программе можно использовать конструкции вроде `ACTION[new Tablekey(s, a)]`. Готовый анализатор (без генерации таблиц) показан на листинге 7.1.

[51] Реализация таблиц анализатора с использованием класса `Hashtable` мне показалась наиболее простой.

Листинг 7.1. Основная процедура синтаксического анализа LR(1)

```
static void Main(string[] args)
{
    ReadGrammar();
    RemoveEpsilonRules();

    Grammar.Add("П S"); // дополнить грамматику правилом П -> S
    Nonterminals += "П";
    Terminals += "$";

    Console.WriteLine("Правила:");
    for(IEnumerator rule = Grammar.GetEnumerator();
                          rule.MoveNext();)
        Console.WriteLine(rule.Current);

    Console.WriteLine("Терминалы: " + Terminals);
    Console.WriteLine("Нетерминалы: " + Nonterminals);
    Console.WriteLine("———");

    // здесь будет генерация LR(1) таблицы
    // ...

    // синтаксический анализ
    string input = Console.In.ReadLine() + "$";
                                // считать входную строку
    if(input.Equals("$"))       // случай пустой строки
    {
        Console.WriteLine(AcceptEmptyString ?
                          "Строка допущена" :
                          "Строка отвергнута");
        return;
    }

    Stack stack = new Stack();          // стек автомата
    stack.Push("0");                    // поместить стартовое
                                        // (нулевое) состояние

    try
    {
        for(;;)
        {
            int s = Convert.ToInt32((string)stack.Peek());

                                        // вершина стека
```

```
            char a = input[0];          // входной символ
            string action = (string)ACTION[new Tablekey(s, a)];
                                        // элемент
                                        //ACTION-таблицы
            if(action[0] == 's')        // shift
            {
               stack.Push(a.ToString()); // поместить в стек a
                 stack.Push(action.Substring(2));
                                        // поместить в стек s'
                 input = input.Substring(1);
                                        // перейти к следующему
                                        // символу строки
            }
            else if(action[0] == 'r') // reduce
            {
                 // rule[1] = A, rule[2] = alpha
                 string[] rule = action.Split(' ');
                 // удалить 2 * Length(alpha) элементов стека
                 for(int i = 0; i < 2 * rule[2].Length; i++)
                     stack.Pop();

                 // вершина стека
                 int state = Convert.ToInt32((string)stack.Peek());
                 // поместить в стек A и GOTO[state, A]
                 stack.Push(rule[1]);
                 stack.Push((GOTO[new Tablekey(state,
                       rule[1][0])]).ToString());

                 // вывести правило
                 Console.WriteLine(rule[1] + " -> " + rule[2]);
            }
            else if(action[0] == 'a')       // accept
                 break;
        }
        Console.WriteLine("Строка допущена");
    }
    catch(Exception)
    {
        Console.WriteLine("Строка отвергнута");
    }
}
```

качестве элементов стека я для простоты использую обычные строки. ри надобности извлекаемое из стека значение конвертируется в целое

число. Элементами ACTION-таблицы также являются строки. Первый символ такой строки — `s`, `r` или `a` — задает тип операции (shift / reduce / accept). Далее через пробел записываются ее параметры (у операции accept параметров нет). Например, операция «shift 5» будет представлена как `s 5`, а «reduce A → abc» — как `r A abc`.

Обратите внимание, что метод Peek() возвращает элемент, находящийся на вершине стека, не извлекая его. Чтобы извлечь элемент, требуется вызов метода Pop().

Также обратите внимание, что в самом начале на стек кладется нулевое состояние в качестве стартового. Разумеется, чтобы нулевое состояние оказалось стартовым, требуется приложить некоторые дополнительные усилия; здесь же я просто пообещаю, что в итоге так оно и окажется.

7.4.5. Генерация таблиц ACTION и GOTO

Теперь займемся самой трудоемкой частью программы — «компиляцией компилятора», то есть созданием таблиц ACTION и GOTO. Но сначала — некоторые предварительные построения.

Множества FIRST

Первая подзадача — создание множеств FIRST. Множество FIRST для любой комбинации символов грамматики α (обозначаемое как FIRST(α)) представляет собой набор терминальных символов, с которых начинаются строки, выводимые из α с помощью правил грамматики (возможно, после поочередного выполнения целого ряда правил). При вычислении множеств FIRST сразу видно, как могут испортить жизнь ε-правила.

Предположим, грамматика содержит правила X → aNC и Y → KTb. Какие терминалы попадут во множество FIRST(X)? А во множество FIRST(Y)? Из X непосредственно выводится строка `aNC`. Что бы ни выводилось из переменных N и C, понятно, что терминальный символ `a` всегда будет началом этой строки. Таким образом, `a` заведомо попадает во множество FIRST(X). Ситуация с множеством FIRST(Y) обстоит сложнее.

Попадает ли в него терминал `b`? Это зависит лишь от наличия ε-правил. Если их нет, то сразу же можно сделать вывод: `b` не попадает в FIRST(Y), поскольку переменные K и T не могут быть превращены в пустые строки по правилам грамматики. Если же допустить наличие ε-правил, придется заниматься дополнительными проверками «зануляемости» K и T.

Упрощенная процедура определения множеств FIRST для каждого символа грамматики (предполагающая отсутствие ε-правил) приведена на псевдокоде:

```
для любого терминала c
    FIRST[c] = {c};
для любого нетерминала X
    FIRST[X] = {};
ПОКА вносятся изменения
    для каждого правила грамматики X →Y0Y1...Yn
        для каждого терминального символа a
            ЕСЛИ множество FIRST[Y0] содержит a
                добавить a в FIRST[X] (если его там еще нет);
КОНЕЦ ЦИКЛА
```

На C# все выглядит ненамного сложнее:

```
static Hashtable FirstSets = new Hashtable();
                          // набор множеств First

static void ComputeFirstSets()
{
    for(int i = 0; i < Terminals.Length; i++)
                          // FIRST[c] = {c}
        FirstSets[Terminals[i]] = Terminals[i].ToString();

    for(int i = 0; i < Nonterminals.Length; i++)
                          // FIRST[X] = ""
        FirstSets[Nonterminals[i]] = "";

    bool changes;
    do
    {
        changes  = false;

          for(IEnumerator rule = Grammar.GetEnumerator();
                                 rule.MoveNext();)
        {
            // для каждого правила X -> Y0Y1...Yn
            char X = ((string)rule.Current)[0];
            string Y = ((string)rule.Current).Substring(2);

            for(int k = 0; k < Terminals.Length; k++)
            {
                char a = Terminals[k]; // для всех терминалов a
```

```
                    // a принадлежит First[Y0]
                    if(((string)FirstSets[Y[0]]).IndexOf(a) != -1)
                        if(((string)FirstSets[X]).IndexOf(a) == -1)
                        {
                            // добавить a в FirstSets[X]
                            FirstSets[X] = (string)FirstSets[X] + a;
                            changes = true;
                        }
                }
            }
        }
        while(changes);  // пока вносятся изменения
    }
```

Для простоты в качестве типа данных «множество» используются строки, поскольку элементами множеств являются обычные ASCII-символы.

Этот алгоритм позволяет получить коллекцию множеств FIRST для всех одиночных символов грамматики, но нам потребуется еще определение FIRST для строк. К счастью, исключив ε-правила, мы получаем очень простое определение:

$$FIRST(X_1X_2...X_n) = FIRST(X_1)$$

Для удобства можно ввести две удобные функции доступа ко множествам FIRST:

```
static string First(char X)
{
    return (string)FirstSets[X];
}

static string First(string X)
{
    return First(X[0]);
}
```

Множество ситуаций и его замыкание

Теперь (нравится нам это или нет) придется ввести еще одно понятие — понятие *ситуации*. Ситуация характеризует текущее состояние процесса анализа строки автоматом. Запись ситуации выглядит точно так же, как запись обычного правила грамматики, но при этом в правой части отмечена «анализируемая позиция». Кроме того, в ситуации должен быть указан терминальный символ, следующий за анализируемым символом грамматики.

Например, если П → S — «стартовое» правило грамматики, автомат начинает работу с ситуации П → ·S, $. Точкой обозначается текущая позиция (автомат собирается проанализировать символ S), а после обработки S ожидается конец строки ($). В дальнейшем нам придется работать только со множествами ситуаций, а не с отдельными ситуациями.

Замыкание (closure) множества ситуаций представляет собой новое множество ситуаций, в котором помимо исходных элементов находятся достижимые из них по правилам входной грамматики ситуации. Определить замыкание множества ситуаций можно с помощью алгоритма:

```
// I  -  входное множество ситуаций
// R  -  замыкание I (изначально пустое)

добавить в R все элементы I;

ПОКА вносятся изменения

 для каждого элемента R (имеющего вид A →α·Bβ, a)[52]

   для каждого правила грамматики вида B →γ

     для каждого элемента b из множества FIRST(βα)

       добавить в R элемент B →·γ, b (если его еще нет в R);

КОНЕЦ ЦИКЛА
```

Алгоритм утверждает достаточно простой факт. Если текущая ситуация имеет вид A → α·Bβ, a, то, начав анализ нетерминала B, автомат попадет в ситуацию A → ·γ, b, где b — первый символ некоторой строки, выводимой из βa.

Как обычно, разобравшись с идеей, переходим к реализации. Каждая ситуация у нас будет представлена строкой, в которой в качестве маркера позиции используется точка, а в качестве разделителя — запятая (понятно, что точка и запятая теперь зарезервированы; использовать их как символы грамматики не стоит):

```
A α.Bβ,a
```

Множество ситуаций (как и его замыкание) в программе описывается с помощью списка (ArrayList) строк. В целом программа довольно точно отражает смысл псевдокода; вот только необходимость вычленять подстроки из ситуаций делает ее немного раздутой:

```
static ArrayList Closure(ArrayList I)
{
    ArrayList result = new ArrayList();
```

[52] Как и раньше, строки α и β могут быть пустыми; B — обязательно нетерминал.

```
// добавляем все элементы I в замыкание
for(IEnumerator item = I.GetEnumerator(); item.MoveNext();)
    result.Add(item.Current);

bool changes;
do
{
    changes = false;

    // для каждого элемента R
    for(IEnumerator item = result.GetEnumerator();
          item.MoveNext();)
    {
        // A -> alpha.Bbeta,a
        string itvalue = (string)item.Current;
        int Bidx = itvalue.IndexOf('.') + 1;
        char B = itvalue[Bidx];                    // B

        if(Nonterminals.IndexOf(B) == -1)
            continue;

        string beta = itvalue.Substring(Bidx + 1);
          beta = beta.Substring(0, beta.Length - 2);  // beta
        char a = itvalue[itvalue.Length - 1];     // a

        // для каждого правила B -> gamma
        for(IEnumerator rule = Grammar.GetEnumerator();
                              rule.MoveNext();)
        if(((string)rule.Current)[0] == B)
                                      // B -> gamma
        {
            string gamma = ((string)rule.Current).
                           Substring(2);          // gamma
            string first_betaa = First(beta + a);

             // для каждого b из FIRST(betaa)
             for(int i = 0; i < first_betaa.Length; i++)
             {
                 char b = first_betaa[i];              // b
                string newitem = B + " ." + gamma + "," + b;

                 // добавить элемент B -> .gamma,b
                 if(!result.Contains(newitem))
```

```
                    {
                        result.Add(newitem);
                        changes = true;
                        goto breakloop;
                    }
                }
            }
        }
        breakloop:;
    }
    while(changes);

    return result;
}
```

Функция GoTo и последовательность C

Для дальнейших построений нам потребуется функция **GoTo()** (не путать с таблицей GOTO!) В книгах она обычно называется **goto()**, но поскольку это слово зарезервировано, я немного изменил написание.

Функция GoTo(I, X), где I — множество ситуаций, а X — некоторый символ грамматики, возвращает множество, определяемое по алгоритму:

```
J = пустое множество
для всех ситуаций вида A →α·Xβ, a из I
    добавить в J ситуацию A →αX·β, a;
return Closure(J);
```

На C# текст функции выглядит немногим сложнее:

```
static ArrayList GoTo(ArrayList I, char X)
{
    ArrayList J = new ArrayList();

    // для всех ситуаций из I
    for(IEnumerator item = I.GetEnumerator(); item.MoveNext();)
    {
        string itvalue = (string)item.Current;
        string[] parts = itvalue.Split('.');

        if((parts[1])[0] != X)
            continue;
```

```
            // если ситуация имеет вид A alpha.Xbeta,a
            J.Add(parts[0] + X + "." + parts[1].Substring(1));
        }

        return Closure(J);
    }
```

Один из ключевых моментов генерации синтаксического анализатора — создание так называемой канонической последовательности множеств ситуаций C. Последовательность C состоит из некоторого количества элементов I_0, I_1, ..., I_n, где каждый элемент представляет собой уже знакомое нам множество ситуаций. Строится C с помощью сравнительно несложных действий:

```
I0 = Closure({"П .S,$"})

ПОКА в C вносятся изменения

 для каждого символа грамматики X

    для каждого элемента Ii из последовательности C

       ЕСЛИ множество GoTo(Ii, X) непусто и оно еще не включено в C
            добавить множество GoTo(Ii, X)
            в хвост последовательности C

КОНЕЦ ЦИКЛА
```

Смысл последовательности C и функции GoTo() состоит в непосредственном построении автомата. Итоговый автомат будет содержать столько состояний, сколько элементов в последовательности C (пока что можно пометить каждое состояние именем соответствующего множества — I_i). Функция же GoTo(I, X) определяет переход автомата из состояния I по символу X.

Процедура получения последовательности C содержит лишь одно нетривиальное действие (заслуживающее отдельной функции): проверка того, что множество GoTo(I, X) еще не включено в C. Для начала нам потребуется функция, определяющая равенство двух данных множеств:

```
static bool SetsEqual(ArrayList lhs, ArrayList rhs)
{
    string[] lhsArr = new string[lhs.Count];
                                    // преобразование списка
    lhs.CopyTo(lhsArr);             // в массив
    Array.Sort(lhsArr);             // и его сортировка

    string[] rhsArr = new string[rhs.Count];
                                    // то же для второго
```

```
    rhs.CopyTo(rhsArr);                  // множества
    Array.Sort(rhsArr);

    if(lhsArr.Length != rhsArr.Length)  // если размеры не равны
        return false;                    // множества точно
                                         // не равны

    for(int i = 0; i < rhsArr.Length; i++)
                                         // если же размеры равны
        if(!lhsArr[i].Equals(rhsArr[i]))
                                         // проверяем по элементам
            return false;

    return true;
}
```

Для начала оба множества преобразуются в сортированные массивы. Затем эти массивы сравниваются поэлементно, и если хотя бы в одном случае соответствующие элементы оказались неравны, делается вывод о неравенстве множеств.

Функция `SetsEqual()` используется функцией `Contains()`, определяющей, является ли множество `g` элементом списка `C`:

```
static bool Contains(ArrayList C, ArrayList g)
{
    for(IEnumerator item = C.GetEnumerator(); item.MoveNext();)
        if(SetsEqual((ArrayList)item.Current, g))
            return true;

    return false;
}
```

Теперь можно запрограммировать основную функцию генерации последовательности `C`. В будущем нам будет удобнее обращаться к отдельным элементам `C`, если она хранится в виде массива. Поэтому итоговое возвращаемое значение представляет собой индексированный массив множеств (`ArrayList[] CArray`).

```
static ArrayList[] CreateCArray()
{
    string Symbols = Terminals + Nonterminals;
                            // все символы грамматики

    ArrayList C = new ArrayList();

    // Добавить элемент I0 = Closure({"П .S,$"})
```

```
    C.Add(Closure(new ArrayList(new Object[]{"П .S,$"})));

    bool modified;
    do
    {
      modified = false;
      for(int i = 0; i < Symbols.Length; i++)
                            // для каждого символа
      {                     // грамматики X
           char X = Symbols[i];

           // для каждого элемента последовательности C
           for(IEnumerator item = C.GetEnumerator();
                                  item.MoveNext();)
           {
                ArrayList g = GoTo((ArrayList)item.Current, X);
                              // GoTo(Ii, X)

                // если множество g непусто и еще не включено в C
                if(g.Count != 0 && !Contains(C, g))
                    {C.Add(g); modified = true; break;}
           }
      }
    }
    while(modified);          // пока вносятся изменения

    ArrayList[] CArray = new ArrayList[C.Count];
                              // преобразование
    C.CopyTo(CArray);         // списка C в массив

    return CArray;
}
```

Обратите внимание, что начальный элемент последовательности (`CArray[0]`) был создан на основе стартовой ситуации П → ·S, $. Именно по этой причине нулевое состояние итогового автомата будет стартовым.

Алгоритм создания таблиц ACTION и GOTO

Что ж, мы выходим на финишную прямую. Имея последовательность C, можно без особых проблем создать таблицы ACTION и GOTO, чем мы незамедлительно и займемся.

Алгоритм генерации таблицы ACTION несколько громоздок, но по своей сути несложен:

```
пусть C = {I0, I1, ..., In};

FOR i = 0 TO n

    для каждой ситуации s из множества Ii

        ЕСЛИ s имеет вид A →α·aβ, b (где a — терминал) и
           существует элемент Ij, такой, что GoTo(Ii, a) = Ij
           ACTION[i, a] = shift j;

        ЕСЛИ s имеет вид A →α·, a и A не является
              стартовым символом (A ≠ П)
           ACTION[i, a] = reduce A →a;

        ЕСЛИ s равно П →S·, $
           ACTION[i, '$'] = accept;
END FOR
```

Если при попытке записи в ACTION-таблицу оказывается, что в данной позиции уже существует другая запись[53], алгоритм должен сообщить о непригодности грамматики (не LR(1) вид).

Чтобы немного сократить программу, я выделил запись элемента ACTION-таблицы в отдельную несложную функцию:

```
static bool WriteActionTableValue(Hashtable ACTION, int I,
                                  char J, string action)
{
    Tablekey Key = new Tablekey(I, J);

    if(ACTION.Contains(Key) && !ACTION[Key].Equals(action))
        return false; // не LR(1) вид

    ACTION[Key] = action;
    return true;
}
```

Основная процедура заполнения ACTION-таблицы, реализующая алгоритм, приведенный на псевдокоде, выглядит так:

```
static Hashtable CreateActionTable(ArrayList[] CArray)
{
    Hashtable ACTION = new Hashtable();
```

[53] «Другая» означает «отличная от текущей». В процессе работы процедура может попытаться записать в некоторую клетку ACTION-таблицы одну и ту же команду несколько раз, в чем нет ничего страшного.

```
for(int i = 0; i < CArray.Length; i++)
                        // цикл по элементам C
{
    // Для каждой ситуации из множества CArray[i]
    for(IEnumerator item = CArray[i].GetEnumerator();
                        item.MoveNext();)
    {
        string itvalue = (string)item.Current;
                        // ситуация
        char a = itvalue[itvalue.IndexOf('.') + 1];
                        // символ за точкой

        // Если ситуация имеет вид "A alpha.abeta,b"
        if(Terminals.IndexOf(a) != -1)
                        // если a - терминал
            for(int j = 0; j < CArray.Length; j++)
                if(SetsEqual(GoTo(CArray[i], a),
                             CArray[j]))
                {
                    // существует элемент CArray[j], такой,
                    // что GoTo(CArray[i], a) == CArray[j]

                    // запись ACTION[i, a] = shift j
                    if(WriteActionTableValue(ACTION, i, a,
                       "s " + j) == false)
                        return null;
                            // грамматика не LR(1)
                    break;
                }

        // Если ситуация имеет вид "A alpha.,a"
        if(itvalue[itvalue.IndexOf('.') + 1] == ',')
                        // за точкой запятая
        { // определить значение a
            a = itvalue[itvalue.Length -1];
            string alpha = itvalue.Split('.')[0].
                        Substring(2);  // и alpha

            if(itvalue[0] != 'П')
                        // если левая часть не равна П
            {
                // ACTION[i, a] = reduce A -> alpha
                if(WriteActionTableValue(ACTION, i, a,
                  "r " + itvalue[0] + " " + alpha) == false)
```

```
                        return null; // грамматика не LR(1)
                    }
                }

                // Если ситуация имеет вид "П S.,$"
                if(itvalue.Equals("П S.,$"))
                {
                // ACTION[i, '$'] = accept
                  if(WriteActionTableValue(ACTION, i, '$', "a")
                     == false)
                        return null;    // грамматика не LR(1)
                }
            }
        }

        return ACTION;
    }
```

Таблица GOTO генерируется по более простому алгоритму. На высшем уровне абстракции все вообще выражается одной строкой:

```
если GoTo(Ii, A) = Ij, то GOTO[i, A] = j
```

Если спуститься немного ниже, выполняемые действия примут следующий вид:

```
для каждого нетерминала A
    для каждой пары элементов (Ii, Ij) из последовательности C
        ЕСЛИ GoTo(Ii, A) = Ij
            GOTO[i, A] = j
```

Наконец, на уровне кода C# функция `CreateGotoTable()` выглядит так:

```
static Hashtable CreateGotoTable(ArrayList[] CArray)
{
    Hashtable GOTO = new Hashtable();

    for(int c = 0; c < Nonterminals.Length; c++)
                        // для каждого нетерминала A
        for(int i = 0; i < CArray.Length; i++)
                        // для каждого элемента Ii из C
        {
            ArrayList g = GoTo(CArray[i], Nonterminals[c]);
                        // g = GoTo[Ii, A]
            for(int j = 0; j < CArray.Length; j++)
```

```
                        // если в C есть Ij = g
                if(SetsEqual(g, CArray[j]))
                        // GOTO[i, A] = j
                    GOTO[new Tablekey(i, Nonterminals[c])] = j;
        }

    return GOTO;
}
```

Генерация LR(1)-таблицы

Осталось всего лишь грамотно вызвать только что созданные функции из тела `Main()`. Содержимое листинга 7.2 вставляется в листинг 7.1 вместо комментария

```
// здесь будет генерация LR(1) таблицы
```

(см. листинг 7.1).

Листинг 7.2. Генерация LR-таблицы

```
ComputeFirstSets();             // вычислить множества FIRST

ArrayList[] CArray = CreateCArray();
                                // создать последовательность C

Hashtable ACTION = CreateActionTable(CArray);
                                // создать ACTION таблицу
if(ACTION == null)
    {Console.WriteLine("Грамматика не является LR(1)"); return;}

Hashtable GOTO = CreateGotoTable(CArray);
                                // создать GOTO таблицу


// распечатать содержимое ACTION и GOTO таблиц
for(IDictionaryEnumerator c = ACTION.GetEnumerator(); c.MoveNext();)
    Console.WriteLine("ACTION[" + ((Tablekey)c.Key).I + ", " +
                    ((Tablekey)c.Key).J + "] = " + c.Value);

for(IDictionaryEnumerator c = GOTO.GetEnumerator(); c.MoveNext();)
    Console.WriteLine("GOTO[" + ((Tablekey)c.Key).I + ", " +
                        ((Tablekey)c.Key).J + "] = " + c.Value);
```

Теперь мне остается лишь поздравить себя и вас (в равной степени) с окончанием этой непростой программы. В следующих главах ничего равного ей по масштабу не планируется.

7.4.6. Тестирование готового анализатора

Давайте насладимся результатами работы нашего приложения, скормив ему какую-нибудь грамматику. Начнем с чего-нибудь очень простого, например, с уже использовавшейся для описания языка `a*b*` грамматики

```
S → aS | bA | ε
A → bA | ε
```

Итак, на вход программы поступает последовательность строк

```
S aS
S bA
S э
A bA
A э
```

Первым делом происходит исключение ε-правил и добавление вывода П → S; в результате мы получаем преобразованную грамматику:

```
Правила:
S aS
S bA
A bA
S a
S b
A b
П S
Терминалы: эab$
Нетерминалы: SAП
```

Содержимое таблиц `ACTION` и `GOTO` вы можете изучить самостоятельно; сейчас более интересна конфигурация дерева разбора. Несколько примеров допускаемых строк и процесс их вывода из правил грамматики показано в таблице:

Корректная строка	Процесс вывода
a	S -> a
ab	S -> b
	S -> aS
aabbbb	A -> b
	A -> bA
	A -> bA
	S -> bA
	S -> aS
	S -> aS

Чтобы восстановить, допустим, дерево разбора строки aabbbb с помощью печатаемой анализатором информации, придется рассмотреть выводимые правила снизу вверх. Последнее выведенное правило S → aS указывает на корень дерева. Следующее правило (опять S → aS) объясняет, как раскрыть переменную S, являющуюся правым потомком корневого узла дерева. Продолжая процесс применения правил снизу вверх, получаем готовое дерево (см. рис. 7.4).

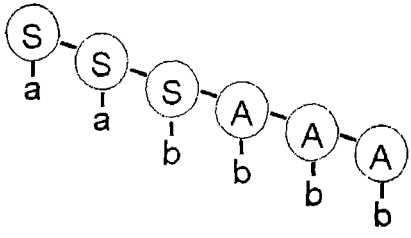


Рис. 7.4. *Дерево разбора строки aabbbb*

Разумеется, приведенная грамматика очень проста. Давайте рассмотрим какой-нибудь пример посложнее. Например, «классику» — грамматику, описывающую арифметическое выражение со скобками:

```
S → E
E → T+E | T*E | T
T → a | b | c | (E)
```

В процессе анализа входной строки `a+b+c*(a+b)` программа выдает правила[54]:

```
T -> a
T -> b
T -> c
T -> a
T -> b
E -> T
E -> T+E
T -> (E)
E -> T
E -> T*E
E -> T+E
E -> T+E
S -> E
```

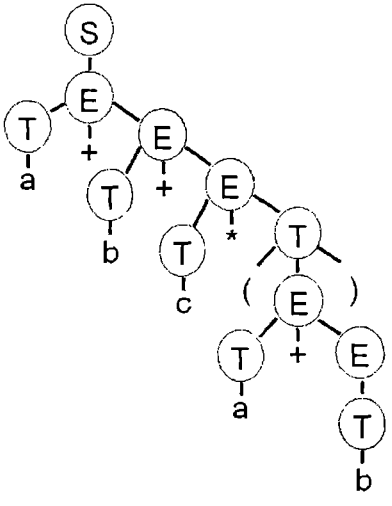


Рис. 7.5. *Дерево разбора строки a+b+c*(a+b)*

[54] Обратите внимание: некоторые правила были сгенерированы несколько раз. По-хорошему дубликаты следовало бы удалить, но их присутствие не должно нарушить правильного хода работы программы.

Нетерминалы в правилах рассматриваются справа налево (в общем, все как положено — справа налево, снизу вверх). В первом правиле S → E` присутствует только один нетерминал — E, поэтому возможности выбора нет. Смотрим следующее правило: E → T+E. Здесь уже присутствуют два нетерминала — T и E. Очередное правило раскрывает определение правого нетерминала (E): E → T+E и так далее.

Что делать с полученным деревом — уже, как говорится, вопрос второй. Вычислять полученные выражения или транслировать процесс их вычисления в последовательность машинных команд — уже задача интерпретатора или компилятора, но никак не синтаксического анализатора.

Разумеется, не все грамматики так удачно анализируются. Например, попытка анализа простой на вид, но неоднозначной грамматики

```
S S+S
S a
```

приводит к не самому радостному результату:

```
Грамматика не является LR(1)
```

7.5. LR(1) анализатор и автомат с магазинной памятью

Думаю, нет надобности доказывать, что построенный LR(1) анализатор (см. п. 7.4) очень похож на автомат с магазинной памятью. При этом он все-таки несколько отличается от классического магазинного автомата. Я не буду досконально описывать процедуру построения «строгого» автомата по данным таблицам ACTION и GOTO (это связано с некоторыми сложностями, да и польза чисто теоретическая), но некоторые идеи все-таки обозначить не помешает.

Итак, какие аспекты готового анализатора отличают его от магазинного автомата? Их не так много:

- **Синтаксический анализатор работает с переменными, записанными в таблицы ACTION и GOTO.** Где же здесь заранее определенные состояния и переходы?

На самом деле значения, записанные в таблицы `ACTION` и `GOTO`, являются константами. Не забывайте, что генерация таблиц — задача компилятора компиляторов, а не синтаксического анализатора. Синтаксический анализатор использует готовые, заранее определенные таблицы — и в этом смысле он не отличается от автомата с фиксированными состояниями и переходами.

- **Магазинный автомат может проанализировать очередной символ входной строки, лишь считывая его и переходя к следующему.** Кроме того, автомат может использовать в работе значения, лежащие на стеке, лишь снимая их с вершины стека. Наш LR(1) анализатор выбирает очередное действие на основе текущего символа входной строки и вершины стека, но не всегда переходит к рассмотрению оставшейся части строки и не всегда снимает верхний элемент со стека.

Магазинный автомат выбирает тот или иной переход на основе очередного считываемого символа входной ленты и содержимого вершины стека. Можно и ничего не считывать со стека / с ленты, если это не нарушает детерминизма. Но считав элемент, воспользоваться им удастся всего лишь один раз: нельзя считать символ строки, перейти в другое состояние, а затем узнать, каков был текущий символ в предыдущем состоянии.

Разумеется, создавать себе какие-то искусственные ограничения в процессе написания парсера просто глупо. Но где гарантия, что обход этих ограничений не выводит нас за пределы мощности магазинных автоматов? В действительности можно написать парсер и не расширяя классического определения автомата. Для этого придется немного схитрить со стеком.

Заметьте, что автомат всегда может считать элемент со стека, не снимая его с вершины: S, a, p → S', p. Чтобы этот переход сработал, на вершине стека должен находиться элемент p. Но при выполнении перехода p не только снимается, но и кладется обратно на стек!

Аналогично можно поступить и со входным символом. Кто мешает начать работу автомата с перехода, подобного P, a, ε, → S, a? Я беру очередной символ строки и записываю его в стек. Далее, если для очередного действия требуется считать следующий элемент входа, он заменит собой старый элемент на стеке. Если же считывать ничего не требуется, для автомата это означает ε-переход, а решение о выборе конкретного перехода принимается на основе содержимого стека (где, в частности, лежит и считанный ранее символ входной строки).

- **В магазинном автомате не предусмотрены действия вроде «считать 2 * длина(α) элементов стека».**

Да, это так, но проблема решается. Во-первых, заметьте, что длина строки α — заранее известная константа. Таким образом, считать со стека требуется не просто сколько-то, а известное на момент создания автомата количество символов. Во-вторых, не забывайте, что множество стековых символов конечно, поэтому несложно сгенерировать *все* возможные строки заданной длины, состоящие из его элементов. Что значит «снять три символа со стека»? Если на стеке могут лежать лишь символы A и B, это значит, что в качестве снимаемой последовательности будет указана любая из строк

```
AAA
AAB
ABA
BAA
ABB
BAB
BBA
BBB
```

Таким образом, потребуется восемь различных переходов для «корректной» имитации этого несложного действия. В более близких к реальности случаях количество переходов будет просто огромным, но никто же не заставляет нас писать программы именно таким образом! Главное сейчас, что сведение анализатора к автомату теоретически возможно, а работающая на практике программа уже написана.

Обратите внимание, что в случае команды reduce потребуется отдельный переход и для каждого значения элемента GOTO[s', A].

Все. Полагаю, мы и так уделили LR(1) анализу слишком много времени. Пора переходить к другим темам.

7.6. Синтаксический анализатор для LL(1) грамматик

Грамматики пригодные и непригодные для LL(1) анализаторов

Технологии LR-анализа называют еще *разбором снизу вверх* (bottom-up parsing). Название отражает тот факт, что анализатор сначала спускается на самые нижние уровни дерева разбора, а потом уже объединяет отдельные ветви в цельные поддеревья. Поэтому стартовое правило П → S, $ всегда оказывалось в самом конце распечатки.

Можно поступить и по-другому: начиная со стартового правила, последовательно раскрывать определение каждого нетерминала, пока вся строка не будет разобрана. При этом на каждом шаге принимается четкое решение об используемом правиле. Такой вид разбора, как нетрудно догадаться, называется *разбором сверху вниз* (top-down parsing) или LL-анализом.

Как я уже говорил, мощность LL-парсера зависит от количества просматриваемых вперед символов входной строки. Наименее мощным (но самым простым и при этом достаточно интересным) является LL(1) анализатор, обсуждением которого мы сейчас и займемся.

Разбор сверху вниз может работать с более узким классом грамматик, чем разбор снизу вверх, а для LL(1) анализа это тем более верно. Например, успешно использованная только что грамматика для арифметических выражений

```
S → E
E → T+E | T*E | T
T → a | b | c | (E)
```

не по силам LL(1)-анализатору. На первом шаге анализатор применяет правило S → E. Но какое правило использовать на следующем шаге? E → T+E, E → T*E или E → T? Конечно, можно спуститься ниже и посмотреть, к чему приведет выбор того или иного правила, но такая техника уже не будет LL(1)-анализом. Смысл как раз в том, что требуемое правило определяется сразу же; максимум, что можно сделать — это считать очередной символ входной строки.

К счастью, в данном конкретном случае грамматику несложно преобразовать так, чтобы она оказалась пригодной для LL(1) разбора. Введем новый нетерминал F и несколько правил для его определения:

```
F → +E | *E | ε
```

Теперь оставшиеся правила грамматики можно переписать с учетом F:

```
S → E
E → TF
T → a | b | c | (E)
```

Новая грамматика уже гораздо проще. На первом шаге разбора применяется правило S → E, затем — E → TF. Далее на основании следующего символа входной строки выбирается одно из правил T → a | b | c | (E). Точно так же раскрывается определение F: если следующий символ входной строки есть знак «+», выбирается правило F → +E, если «*» — правило F → *E; в остальных ситуациях считается, что F → ε.

Пишем парсер для LL(1)-грамматики

Парсер для LL(1) грамматики написать очень просто. Для каждого нетерминального символа создается одноименная функция (так, функция, разбирающая нетерминал A, будет называться A()). Тело каждой такой функции выполняет похожий алгоритм:

```
на основании следующего символа входной строки выбрать правило;

напечатать выбранное правило;

ЦИКЛ по символам правой части правила

    ЕСЛИ символ  —  нетерминал,
```

```
        вызвать соответствующую функцию;
    ЕСЛИ символ — терминал, считать его и перейти
        к следующему символу входной строки;
КОНЕЦ ЦИКЛА
```

Если в какой-то момент времени окажется, что текущий символ входной строки не соответствует ожиданиям, строка отвергается. Если же функция S() отрабатывает без проблем, а текущий символ равен признаку конца, строка допускается. Задача же функции Main() — простой вызов S()[55]. Признак конца нужен для того, чтобы убедиться, что строка прочитана полностью. Так, грамматике

```
S → aa
```

соответствует лишь строка aa; тем не менее, функция S() завершится успешно и при входной строке aaaaa. Первые два символа входной строки будут проанализированы, а дальше программа попросту не заглянет.

Как это выглядит на практике (на характерном примере)

Думаю, в качестве примера достаточно построить анализатор для LL(1) грамматики, задающей арифметические выражения (приведу ее еще раз):

```
S → E
E → TF
F → +E | *E | ε
T → a | b | c | (E)
```

Программа очень проста и понятна практически без комментариев (см. листинг 7.3).

Листинг 7.3. LL(1) анализатор арифметических выражений

```
static string input;              // входная строка

static char NextChar()            // очередной символ входной
                                  // строки
{
    return input[0];
}
```

[55] Описанная техника называется рекурсивным нисходящим синтаксическим анализом или синтаксическим анализом методом рекурсивного спуска (recursive descent parsing).

```
static void AdvancePointer()   // перейти к следующему символу
{
    input = input.Substring(1);
}

static void S()                // S -> E
{
    Console.WriteLine("S -> E");
    E();
}

static void E()                // E -> TF
{
    Console.WriteLine("E -> TF");
    T(); F();
}

static void F()                // F -> +E | *E | э
{
    string rule;
    if(NextChar() == '+')      rule = "+E";
    // выбор альтернативы
    else if(NextChar() == '*') rule = "*E";
    else                       rule = "э";

    Console.WriteLine("F -> " + rule);
    switch(rule)
    {
        case "+E": AdvancePointer(); E(); break;  // F -> +E
        case "*E": AdvancePointer(); E(); break;  // F -> *E
        case  "э": break;                         // F -> э
    }
}

static void T()                // T -> a | b | c | (E)
{
    string rule;

    if(NextChar() == 'a')      rule = "a";
      // выбор альтернативы
    else if(NextChar() == 'b') rule = "b";
    else if(NextChar() == 'c') rule = "c";
    else if(NextChar() == '(') rule = "(E)";
    else                       throw new Exception();
```

```
    Console.WriteLine("T -> " + rule);
    switch(rule)
    {
        case   "a": AdvancePointer(); break;       // T -> a
        case   "b": AdvancePointer(); break;       // T -> b
        case   "c": AdvancePointer(); break;       // T -> c
        case "(E)": AdvancePointer();              // T -> (E)
                    E();
                  if(NextChar() != ')')  throw new Exception();
                    AdvancePointer();
                    break;
    }
}

static void Main(string[] args)
{
    input = Console.ReadLine() + "$";
    try
    {
        S();
        Console.WriteLine((NextChar() == '$') ?
                          "Строка принята" :
                          "Строка отвергнута");
    }
    catch(Exception)
    {
        Console.WriteLine("Строка отвергнута");
    }
}
```

Грамматика, используемая на сей раз, немного сложнее своего LR-аналога. Поэтому результирующий вывод (и дерево разбора) для строки `a+b+c*(a+b)` будет сложнее вывода и дерева, производимых LR(1) анализатором:

```
S -> E
E -> TF
T -> a
F -> +E
E -> TF
T -> b
F -> +E
E -> TF
T -> c
F -> *E
```

```
E -> TF
T -> (E)
E -> TF
T -> a
F -> +E
E -> TF
T -> b
F -> э
F -> э
```

Обратите внимание, что теперь вывод производится не снизу вверх, а как полагается, сверху вниз. Дерево разбора показано на рис. 7.6.

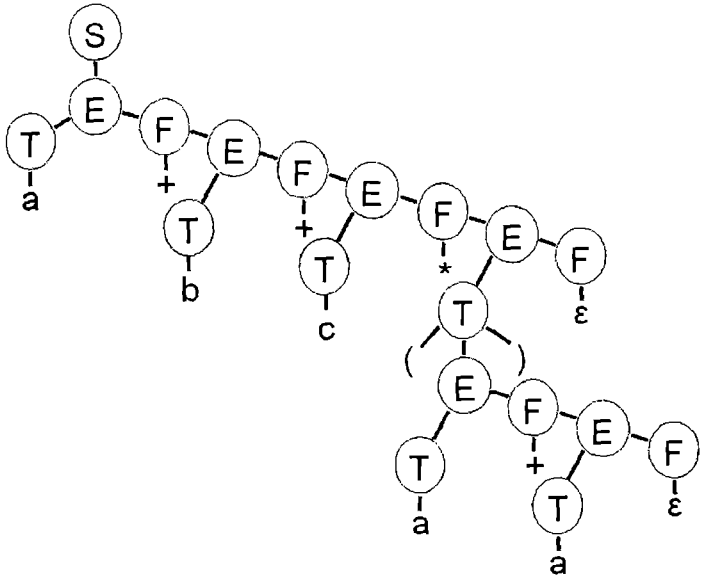


Рис. 7.6. *Дерево разбора строки a+b+c*(a+b), случай LL(1) грамматики*

Правила, помогающие привести грамматику к LL(1) виду

Закончить тему LL(1) разбора предлагаю обсуждением двух простых правил, помогающих во многих случаях привести грамматику к LL(1)-виду.

Правило первое. Выводы вида

$A \to \alpha\beta \mid \alpha\gamma$, где $\alpha \neq \varepsilon$, $\beta \neq \gamma$

нарушают LL(1) форму. Действительно, если две разные правые части начинаются с одинаковых символов, синтаксический анализатор не сумеет

правильно выбрать альтернативу. Чтобы избавиться от подобных правил, достаточно ввести новую переменную B и заменить $A \to \alpha\beta \mid \alpha\gamma$ на

$A \to \alpha B$
$B \to \beta \mid \gamma$

Правило второе. Выводы вида

$A \to A\beta \mid \alpha$, где $\beta \neq \varepsilon$

также нарушают LL(1) форму (эти правила еще называют непосредственной левой рекурсией). Здесь проблема другая: функция A() будет первым же делом вызывать сама себя, пока стек не закончится. Избавиться от непосредственной левой рекурсии можно опять-таки с помощью новой переменной:

$A \to \alpha B$
$B \to \beta B \mid \varepsilon$

Это преобразование не слишком очевидно; я советую проверить его на паре простых примеров, чтобы убедиться в том, что оно работает корректно. Более простой случай — правила вида $A \to A$ (порою возникающие при механическом выполнении того или иного алгоритма преобразования) — можно просто удалить без вреда для грамматики. Если в правиле существуют другие леворекурсивные правые части (то есть выводы имеют вид $A \to A\beta \mid \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$), придется создать по отдельному правилу для каждой строки α_i:

$A \to \alpha_1 B$
$A \to \alpha_2 B$
...
$A \to \alpha_n B$
$B \to \beta B \mid \varepsilon$

Настоящие проблемы начинаются с возникновением левой рекурсии «общего вида», то есть ситуации, когда строку $A\beta$ можно получить из A путем применения целой последовательности правил грамматики. К сожалению, в этом случае простых рецептов решения не существует. А результатом работы LL(1) парсера будет все то же переполнение стека в процессе выполнения бесконечной непрямой рекурсии.

Существует алгоритм, определяющий, принадлежит ли данная грамматика к LL(1) виду (он требует вычисления уже знакомого множества FIRST и других, не менее «занимательных» вещей), но я не буду на нем останавливаться. Интересующие без труда найдут его в любой хорошей книге по теории и практике компиляции.

7.7. Синтаксический анализатор для любых контекстно-свободных грамматик

Итак, позади остались частные случаи LR(1) и LL(1) парсеров. Пора привести алгоритм, способный определить принадлежность строки к языку, описанному контекстно-свободной грамматикой общего вида.

Напомню, в контекстно-свободных грамматиках допускаются правила, сопоставляющие нетерминальному символу любую строку, состоящую из терминалов и нетерминалов, либо пустую строку. Но, прежде чем познакомиться с этим алгоритмом, известным как *алгоритм Кока-Янгера-Касами* (Cocke-Younger-Kasami, CYK), придется разобраться с понятием *нормальной формы Хомского* (Chomsky normal form, CNF).

7.7.1. Нормальная форма Хомского

Правила, составляющие нормальную форму Хомского

Изобретение формальных грамматик — заслуга американского лингвиста (и известного анархиста) Наома Хомского (Noam Chomsky[56]). Ему же принадлежит **«нормальная форма», представляющая собой минимальный набор вариантов правил, позволяющих описать любой контекстно-свободный язык, не содержащий пустых строк.** Как я уже отмечал, пустая строка — тривиальный случай, и вынести его в отдельную проверку (одна строчка кода) нетрудно.

Вероятно, это кажется непостижимым, но для описания любого контекстно-свободного языка достаточно всего двух типов правил:

A → BC, где B и C — переменные

и

→ a, где a — терминальный символ

[56] Отец Наома Уильям Хомский эмигрировал из России еще до революции. В Америке его фамилия превратилась в «Chomsky», и на сегодняшний день большинство зарубежных ученых (включая, скорее всего, самого Наома) произносят ее как «Чомски». В русскоязычной литературе почти всегда используют оригинальное написание.

Алгоритм преобразования в нормальную форму Хомского

Преобразование в нормальную форму Хомского производится в три этапа:

1. **Удаление правил вида A → ε.**

Мы уже решали эту задачу, когда занимались построением LR(1) парсера. Перечитайте раздел «Удаление ε-правил».

2. **Удаление правил вида A → B.**

Эта процедура, в принципе, аналогична предыдущей:

```
ПОКА существуют правила вида A →B
    найти правило вида A → B;
    для любого правила вида B →α (где α — любая строка,
            кроме одиночного нетерминала)
        добавить правило A →α;
    удалить правило A → B;
КОНЕЦ ЦИКЛА
```

3. **Преобразование правил вида A → α, если строка α длиннее одного символа.** Этот алгоритм громоздок, но не очень сложен для программирования:

```
преобразование правила A → B0B1...Bk (где Bi — одиночный символ
грамматики)
добавить в грамматику правила
    A → X_B0X_B1...Bk
    X_B1...Bk →X_B1X_B2...Bk
    X_B2...Bk →X_B2X_B3...Bk
    ...
    X_Bk-1Bk →X_Bk-1X_Bk
соблюдая соглашения: если Bi — нетерминал,
                         вместо X_Bi использовать Bi
                     если Bi — терминал,
                         добавить правило X_Bi →Bi
удалить исходное правило
```

Немного трудные для чтения обозначения $X_{индекс}$ представляют собой новые нетерминальные символы. Каждый такой нетерминал определяется с помощью правила вида A → BC, где C — один из новых нетерминалов, а B — либо один из исходных нетерминалов, либо нетерминал, за одно действие превращающийся в терминальный символ.

Каждое правило в построенной цепочке «отъедает» один символ исходного правила A → B0B1...Bk, и, в конце концов, правило целиком будет разобрано на составляющие. Понять смысл происходящего должен помочь пример. Рассмотрим преобразование правила S → FabDFG:

$\rightarrow FX_{abDFG}$
$_{abDFG} \rightarrow X_aX_{bDFG}$
$_{bDFG} \rightarrow X_bX_{DFG}$
$_{DFG} \rightarrow DX_{FG}$
$_{FG} \rightarrow FG$
$_a \rightarrow a$
$_b \rightarrow b$

ɔограммирование преобразования в нормальную форму Хомского

ıрограммировать преобразование грамматики в нормальную форму мского не очень сложно, тем не менее, думаю, реально работающая ɔграмма лучше любых псевдокодов. К сожалению, одна проблема не-ɔго отравляет жизнь: индексирование новых нетерминалов. Поскольку обозначаем нетерминалы заглавными буквами (и только ими), записи ɔде X_{abDFG}, конечно, не удастся использовать.

ıредлагаю такое решение: как только нам понадобится новый нетер-ıал, будем использовать в его качестве очередную (начиная с «А») ıву кириллического алфавита. Разумеется, через 33 нетерминала все ıвы исчерпаются; разумеется, многие буквы кириллицы похожи на ıвы латиницы («А», «Е», «Т», ...), но... боюсь, иного выхода нет (если ьезно не усложнять программу). Кроме того, для учебных целей лне сгодится.

нкцию, считывающую грамматику, а также код, реализующий первый п (удаление ε-правил) я просто скопировал из предыдущей програм-(знаю, так делать нельзя, но создавать сейчас какие-то общие модули считаю целесообразным). Таким образом, функции `ReadGrammar()`, `erateCombinations()`, `GenerateRulesWithout()` и `RemoveEpsilonRules()` рограмме уже присутствуют.

ıерь добавим функцию `RemoveAtoBRules()`, удаляющую правила ıа $A \rightarrow B$:

```
tatic void RemoveAtoBRules() // удаление правил A -> B
                             // аналогично удалению
                             // эпсилон-правил
  bool AtoBRulesExist;

  do                          // пока правила A -> B
                              // существуют
  {
      AtoBRulesExist = false;
```

```
        for(IEnumerator rule = Grammar.GetEnumerator();
                                rule.MoveNext();)
        if(((string)rule.Current).Length == 3 &&
                                // нашли A -> B правило
                    Nonterminals.IndexOf(((string)rule.
                                        Current)[2]) != -1)
        {
            char lhs = ((string)rule.Current)[0];
                                // левая часть (A)
            char rhs = ((string)rule.Current)[2];
                                // правая часть (B)

            ArrayList newrules = new ArrayList();
                                // добавляемые правила

           // для любого правила вида (B -> alpha)
           for(IEnumerator ruleBA = Grammar.GetEnumerator();
                                    ruleBA.MoveNext();)
           {
                char new_lhs = ((string)ruleBA.Current)[0];
                                // левая часть (B)
                // правая часть (alpha)
                string new_rhs = ((string)ruleBA.Current).
                                    Substring(2);

                // если alpha  -  не нетерминал
                if(new_lhs == rhs && !(new_rhs.Length == 1 &&
                   Nonterminals.IndexOf(new_rhs[0]) != -1))
                   newrules.Add(lhs + " " + new_rhs);
                                // добавить правило
           }

            Grammar.Remove(rule.Current);
                                // удаляем A -> B правило
            Grammar.AddRange(newrules);
                                // добавляем новые правила
            AtoBRulesExist = true;
            break;
        }
    }
    while(AtoBRulesExist);    // пока существуют
                                // A -> B -правила
}
```

Как видите, функция по форме сложнее псевдокода, но по сути прямо соответствует ему. Учитывая, что теперь дублирующиеся правила могут серьезно раздуть грамматику, я написал функцию удаления дубликатов:

```
static void KillDuplicates()
{
    // вставляем все правила в хэш-таблицу в качестве ключей
    Hashtable rules = new Hashtable();
    for(IEnumerator rule = Grammar.GetEnumerator();
                          rule.MoveNext();)
        rules[(string)rule.Current] = true;

    // заполняем список Grammar ключами таблицы rules
    Grammar.Clear();
    for(IDictionaryEnumerator c = rules.GetEnumerator();
                                  c.MoveNext();)
        Grammar.Add((string)c.Key);
}
```

Идея этого алгоритма уже применялась. Мы кладем все элементы коллекции в хэш-таблицу в качестве ключей, пользуясь тем, что ключи уникальны. Затем просто проходим по всем ключам и заполняем ими результирующую коллекцию.

Третий этап (конвертирование правил) программируется, как нетрудно догадаться, не так прямолинейно. Этот алгоритм лучше привести сначала в псевдокоде:

```
NewV = русская 'A';

для любого правила вида A →α, где длина(α) > 1

    поместить A →α в список удаляемых;

    L = A;

    ПОКА длина(α) >= 2
        X = GetNewVar(α[0]);
        Y = (длина(α) == 2) ? GetNewVar(α[1]) : NewV;

        поместить правило L →XY в список добавляемых;
        α = α.Substring(1)
        L = NewV++;

    КОНЕЦ ЦИКЛА
```

```
удалить из грамматики правила, подлежащие удалению;
добавить новые правила в грамматику;

char GetNewVar(char OldVar)
    ЕСЛИ OldVar — нетерминал, вернуть OldVar;

    NewVar = NewV++;

    добавить правило NewVar → OldVar;

    вернуть NewVar;
```

Переменная `NewV` хранит значение очередного нетерминала, который будет использоваться, как только в нем появится потребность. А потребность появляется в двух случаях. Во-первых, при конвертировании любого длинного правила вида A → B0B1...Bn в A → $X_{B0}X_{хвост}$ переменная $X_{хвост}$ — заведомо новая. Во-вторых, переменная X_{B0} может быть как новой, так и «старой», то есть равной одному из нетерминалов исходной грамматики.

Функция `GetNewVar()` определяет, создавать ли новую переменную во втором случае. Если переданный ей параметр является нетерминалом (исходной грамматики), он возвращается без изменений. Если же значение `OldVar` представляет собой терминальный символ, возвращается новый, созданный функцией нетерминал `NewVar`. Кроме того, в грамматику добавляется правило NewVar → OldVar. Последнее действие соответствует строке

```
если Bi — терминал, добавить правило X_Bi → Bi
```

псевдокода.

Основной алгоритм создает новые правила следующим образом. Известно, что добавляемое правило имеет вид L → XY, где L изначально равно A, а значения X и Y еще надо определить. Переменная X заведомо равна `GetNewVar(α[0])` (преобразуем X_{Bi} в Bi или добавляем правило X_{Bi} → Bi). Со значением Y дело обстоит не так просто. Если α представляет собой одиночный символ, то работа алгоритма практически закончена: осталось лишь выполнить присваивание Y = `GetNewVar(α)` и добавить в грамматику последнее правило L → XY. В противном случае придется ввести новый нетерминал Y = NewV, добавить правило L → XY и повторить алгоритм для новой левой части L = Y и укороченной правой части α = `α.Substring(1)`.

Готовый алгоритм преобразования правил вместе с простой функцией `Main()` приведен в листинге 7.4.

Листинг 7.4. Получение нормальной формы Хомского

```
static char NewV = 'A';          // русская A
static ArrayList newrules = new ArrayList();

static char GetNewVar(char OldVar)
                                 // получить символ из нового
                                 // правила
{                                // по данному символу старого
    char NewVar;

    if(Nonterminals.IndexOf(OldVar) != -1)
                                 // если символ  —  нетерминал
        NewVar = OldVar;         // он переходит в новое правило
    else                         // без изменений
    {
        NewVar = NewV++;         // если же символ  —  терминал
        newrules.Add(NewVar + " " + OldVar);
                                 // добавить правило вида X -> a
    }                            // и вернуть новый нетерминал X

    return NewVar;
}

static void ConvertRules()
{
    ArrayList todel = new ArrayList();
                                 // список удаляемых правил

    // для любого правила вида A -> alpha, где длина alpha > 1
    for(IEnumerator rule = Grammar.GetEnumerator();
                            rule.MoveNext();)
        if(((string)rule.Current).Length > 3)
        {
            char L = ((string)rule.Current)[0];
                                 // левая и правая части
            string alpha = ((string)rule.Current).Substring(2);

            todel.Add(rule.Current);
                                 // поместить правило в список
                                 // удаляемых
            while(alpha.Length >= 2)
                                 // пока правило целиком не
                                 // разобрано
            {
```

```
                // генерация нового правила вида L -> XY
                // сначала получаем X по alpha[0]
                char X = GetNewVar(alpha[0]);

                // если в исходном правиле осталось всего
                // символа, получаем значение Y с помощью
                // вызова GetNewVar() иначе берем новый
                // нетерминал из списка
                 char Y = (alpha.Length == 2) ?
                          GetNewVar(alpha[1]) : NewV;

                                // добавить правило L -> XY
                newrules.Add(L + " " + X + Y);
                alpha = alpha.Substring(1);
                                // укорачиваем исходное правило
                L = NewV++;
                                // берем новый нетерминал
                                // из списка
            }
        }

    for(IEnumerator rule = todel.GetEnumerator(); rule.MoveNext();)
        Grammar.Remove((string)rule.Current);

    Grammar.AddRange(newrules);
}

static void Main(string[] args)
{
    ReadGrammar();
    RemoveEpsilonRules();
    RemoveAtoBRules();
    KillDuplicates();
    ConvertRules();

    Console.WriteLine("Правила:");
    for(IEnumerator rule = Grammar.GetEnumerator();
                           rule.MoveNext();)
        Console.WriteLine(rule.Current);
    Console.WriteLine("Принимать пустую строку:
                      " + AcceptEmptyString);
}
```

В качестве примера предлагаю пропустить сквозь алгоритм правило S → FabDFG (тем более что правильный ответ нам известен):

```
S FabDFG

Правила:

S FA

Б a

A БВ

Г b

В ГД

Д DE

E FG
```

Я специально выбрал в качестве нетерминалов исходной грамматики символы, не похожие ни на одну букву русского алфавита, поэтому путаницы в обозначениях не возникнет. Если сопоставить X_{abDFG} = А, X_a = Б, X_{bDFG} = В, X_b = Г, X_{DFG} = Д, а X_{FG} = Е, то распечатка, выданная программой, идентична построенному вручную преобразованию (см. выше).

Нормальная форма Хомского в JFLAP

Нормальную форму Хомского можно получить также с помощью JFLAP (режим **Grammar**, пункт **Convert → Transform Grammar**).

Алгоритм Кока-Янгера-Касами

Этот алгоритм, позволяющий определить принадлежность строки к языку, описываемому произвольной контекстно-свободной грамматикой, требует, чтобы входная грамматика была записана в нормальной форме Хомского.

Сейчас, когда алгоритм построения нормальной формы уже запрограммирован, можно перейти непосредственно к процедуре Кока-Янгера-Касами. Сразу же напомню, что случай пустой строки должен рассматриваться отдельно.

В псевдокоде используются обозначения:

- N — длина входной строки a;
- S — стартовый символ грамматики;
- V — двумерный массив, каждый элемент V[i, j] которого является множеством нетерминалов.

А вот и сам псевдокод (в наиболее простой вариации):

```
for i = 1 to N
    V[i, 1] = множество переменных A, таких,
              что существует правило A →a[i][57];

for j = 2 to N
    for i = 1 to N  -  j + 1
        V[i, j] = пустое множество;
        for k = 1 to j  -  1
            V[i, j] = V[i, j] ∪ множество нетерминалов A,
                                таких, что существует правило
                                A →BC, где B содержится в V[i, k],
                                а C содержится в V[i+k, j-k]

ЕСЛИ S содержится в V[1, N], допустить строку;
     иначе отвергнуть;
```

Понять работу этого алгоритма проще всего на примере. Рассмотрим грамматику в нормальной форме Хомского:

```
S b
N b
S a
A a
S AS
B b
N BN
Д b
S ДN
```

Она получена из простой грамматики

```
S → aS | bN | ε
N → bN | ε,
```

задающей язык a*b* (пустая строка новой грамматикой не принимается, но, как уже отмечалось, это отдельный случай). При разборе строки aabb алгоритм CYK строит таблицу V (см. рис. 7.7).

Клетки левого столбца таблицы (элементы V[1, 1] — V[4, 1]) содержат нетерминалы, из которых можно получить соответствующие строки единичной длины (то есть отдельные символы входной строки). Из S и A можно получить a, из S, N, B и Д — b.

[57] В литературе обычно начинают индексирование с единицы. Таким образом, a[1] — самый первый символ входной строки.

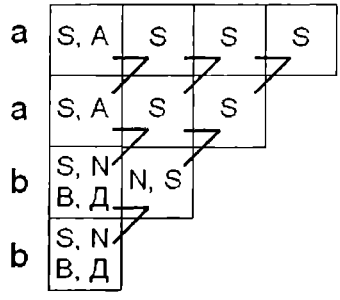


Рис. 7.7. *CYK-таблица для строки aabb*

едующий столбец содержит нетерминалы, из которых можно получить
оки длиной уже в два символа. Так, верхняя клетка второго столбца
ержит нетерминал, из которого выводится строка aa. Строка ab тоже
водится из нетерминала S, а вот строка bb может быть успешно вы-
ена как их нетерминала N, так и из нетерминала S.

одолжая построение столбцов, получаем в клетке V[1, N] список не-
миналов, из которых может быть выведена вся входная строка. Если
том списке присутствует стартовый символ грамматики, строка до-
кается.

таблице V можно построить и дерево разбора. Мы не будем этим
иматься, поскольку задача не так проста, как в случае LL(1) и LR(1)
серов. При выполнении алгоритма CYK, вообще говоря, получается
ая коллекция деревьев: не забывайте, что грамматика может быть
днозначной, и вывести одну и ту же строку можно совершенно раз-
ными путями. В процессе построения таблицы можно найти их все.

овая программа (почти однозначно соответствующая псевдокоду)
ведена в листинге 7.5.

стинг 7.5. Алгоритм Кока-Янгера-Касами

```
tatic ArrayList Grammar = new ArrayList();
                          // правила грамматики

tatic void ReadGrammar()

   string s;

   while((s = Console.In.ReadLine()) != "")
                          // считывание правил
```

```
        Grammar.Add(s);       // добавить правило в грамматику
}

static ArrayList FindA_aiRules(char a)
                              // найти правила вида A -> a
{
    ArrayList result = new ArrayList();

    for(IEnumerator rule = Grammar.GetEnumerator();
                           rule.MoveNext();)
    {
        string cur_rule = (string)rule.Current;

         // добавить левую часть найденного правила
         // в список result
        if(cur_rule.Length == 3 && cur_rule[2] == a)
            result.Add(cur_rule[0]);
    }

    return result;
}

// найти правила вида A -> BC, где B из BSet, а C из CSet
static ArrayList FindA_BCRules(ArrayList BSet, ArrayList CSet)
{
    ArrayList result = new ArrayList();

    for(IEnumerator rule = Grammar.GetEnumerator();
                           rule.MoveNext();)
    {
        string cur_rule = (string)rule.Current;

        // добавить найденное правило в список result
        if(cur_rule.Length == 4 && BSet.Contains(cur_rule[2])
                               && CSet.Contains(cur_rule[3]))
                              result.Add(cur_rule[0]);
    }

    return result;
}

static void Main(string[] args)
{
    ReadGrammar();
```

```
    string input = Console.ReadLine();
                                // входная строка
    int N = input.Length;

    ArrayList[,] V = new ArrayList[N + 1, N + 1];
                                // таблица V

    for(int i = 1; i <= N; i++)
                                //заполняем левый столбец
        V[i, 1] = FindA_aiRules(input[i - 1]);

    for(int j = 2; j <= N; j++)// основная часть
         for(int i = 1; i <= N - j + 1; i++)
         {
            V[i, j] = new ArrayList();
                                // V[i, j] = {}
            for(int k = 1; k <= j - 1; k++)
                V[i, j].AddRange(FindA_BCRules(V[i, k],
                                   V[i + k, j - k]));
         }

    Console.WriteLine("Строка принадлежит языку: "+
                        V[1, N].Contains('S'));
}
```

оскольку алгоритм Кока-Янгера-Касами содержит три вложенных цик-, каждый из которых выполняет прямо пропорциональное N количество ераций, расходуемое программой время в итоге пропорционально N^3. читается, что для задачи распознавания контекстно-свободного языка бическая сложность — признак большой эффективности алгоритма.

онечно, в частных случаях LL и LR грамматик ему далеко до специа-зированных процедур, имеющих линейную сложность. Вообще если ворить о практике, пожалуй, наиболее популярны на сегодняшний день к называемые LALR-анализаторы. Они работают подобно LR-парсе-м, но используют более ограниченные LALR-грамматики, ACTION- и OTO-таблицы которых требуют гораздо меньшего объема.

Итоги

- Задача синтаксического анализа состоит в построении дерева разбора для любой строки языка или вывода сообщения о том, что строка языку не принадлежит. Таким образом, задача распознавания языка является частным случаем задачи синтаксического анализа.
- Задача синтаксического анализа разрешима для любых контекстно-свободных языков. Для распознавания контекстно-свободного языка требуется мощь недетерминированного автомата с магазинной памятью.
- Поскольку на практике недетерминированные устройства недоступны, можно воспользоваться алгоритмом Кока-Янгера-Касами.
- Детерминированный магазинный автомат может распознать лишь детерминированный контекстно-свободный язык. Построенный нами LR(1) анализатор, имитируя работу магазинного автомата, распознает любой язык, записанный с помощью так называемой LR(1) грамматики. Доказано, что с помощью LR(1) грамматики можно описать любой детерминированный язык.
- LL(1) анализатор — это шаг немного в сторону. Программа LL(1) анализа использует более ограниченные LL(1) языки, но сама идея ее создания очень проста и любопытна, поэтому я не решился пройти мимо нее.
- Алгоритм Кока-Янгера-Касами, предназначенный для разбора любого контекстно-свободного языка, выполняет порядка N^3 операций при анализе строки длиной N символов. Анализатор, основанный на LR(1) грамматиках, работает гораздо быстрее (линейное время), но контекстно-свободные языки, выходящие за рамки детерминированных, ему не по силам.

Глава 8

Генерация компиляторов. Практика создания своего компилятора

- Основные понятия
- Практический пример: транслятор простейшего языка программирования
- Генерация лексического и синтаксического анализаторов
- Создание промежуточного представления программы
- Интерпретация промежуточного кода

К
Алго ры.
ПРАК

8.1. Основные понятия

Трансляторы, компиляторы, интерпретаторы

После разговора о синтаксическом анализе имеет смысл затронуть и тему генерации компиляторов. Хотя парсеры могут использоваться для разных целей, нет сомнений, что в первую очередь их все-таки связывают с задачей компиляции.

Не думаю, что кто-нибудь из читателей завтра же возьмется за написание нового языка программирования общего назначения, такого как Pascal, C# или Java. С другой стороны, если речь идет о выполнении некоторой последовательности команд для вычерчивания фигур на экране, управлении какими-либо объектами в компьютерной игре или вычислении несложных математических функций, то ситуация становится уже не столь далекой от реальности.

В конце концов, общая задача создания компилятора заключается в распознавании некоторых управляющих конструкций и адекватном на них реагировании, а не в попытках переплюнуть Microsoft или Sun Microsystems.

Здесь можно немного порассуждать также о термине «компилятор». Я использую его лишь потому, что инструмент, с которым мы будем иметь дело на протяжении главы, относится к классу программ, называемых «компиляторами компиляторов» (compiler compilers)[58]. Если инструмент является компилятором компиляторов, то результатом его работы должен быть, соответственно, компилятор.

На самом деле с помощью такого средства вы можете создать как компилятор, так и интерпретатор, транслятор или вообще просто синтаксический анализатор без каких-либо дополнительных функций. Чтобы не путаться в терминологии, имеет смысл напомнить, что обычно подразумевают под интерпретатором, транслятором и компилятором.

[58] Если помните, это сочетание уже упоминалось в главе о синтаксическом анализе.

- **Транслятор — это программа, переводящая текст, написанный на одном языке программирования, в текст на другом языке.** Например, программу на Паскале в программу на C. Или текст с языка C в текст на языке Ассемблера.
- **Компилятор — это программа, преобразующая текст, написанный на каком-либо языке программирования, в программу в машинных кодах.** Таким образом, компилятор является транслятором с языка высокого уровня на язык машинных кодов.
- **Интерпретатор — это программа, непосредственно (команда за командой) выполняющая текст на каком-либо языке программирования.**

Эта классификация достаточно условна, поскольку в различных ситуациях одни и те же действия могут расцениваться по-разному. К тому же реализация конкретного инструмента разработчика (не буду здесь употреблять ни один из трех рассматриваемых терминов!) может состоять из комбинации компилятора и транслятора (или даже нескольких трансляторов).

Например, является ли компилятором транслятор, преобразующий программу на Бейсике в программу на C#? Как правило, нет. А если «родным» языком процессора является C# (теоретически такое возможно)? Тогда конечно. Сам по себе процессор в известном смысле всегда является интерпретатором, поскольку он по шагам выполняет инструкции, записанные в машинном коде.

Интерпретатор тоже нередко сначала преобразует исходную программу в некоторое промежуточное представление, так как интерпретируемый язык программирования может быть слишком сложен для непосредственного пошагового выполнения (особенно если речь идет о языке высокого уровня).

Создатели «настоящих» (то есть реально преобразующих программу в машинный код) компиляторов тоже могут упростить себе задачу, если вспомнят, что хорошие компиляторы для Ассемблера или, скажем, C давным-давно созданы.

Так, первые компиляторы C++ преобразовывали входную программу на C++ в эквивалентную программу на C, а затем вызывали готовый компилятор C. Среда программирования Borland C++ 3.1, с которой я когда-то работал, умеет производить трансляцию кода на C++ в ассемблерный текст. Его, в свою очередь, полагается компилировать при помощи Tasm, после чего получается полноценный exe-файл. Популярный (в узких кругах любителей) компилятор языка Ada GNAT переводит программу на Ada в программу на C, а затем вызывает GCC для генерации исполняемого кода.

Проект Coco/R

Обычно в качестве инструментов разработчика компиляторов прежде всего упоминают классические программы YACC и Bison, уже не первое десятилетие служащие программистам.

Ознакомившись с современными аналогами, я решил остановиться на генераторе компиляторов Coco/R[59], недавно разработанном в университете г. Линца (Австрия). В отличие от основанных на языке C YACC и Bison, Coco/R использует в качестве базового языка C# (что для нас, несомненно, является плюсом). Кроме того, Coco/R достаточно прост в использовании, если слово «простота» вообще применимо к задаче создания компилятора.

Этот генератор можно скачать со страницы

```
http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/
```

Там находится как уже скомпилированная версия Coco/R, так и комплект исходных текстов. Для запуска программы в текущем каталоге должны находиться три файла:

```
Coco.exe
Parser.frame
Scanner.frame
```

С удовольствием замечу, что вместе с программой поставляется довольно приличная документация: краткая и вполне читабельная.

Идеология компилятора

Полноценный компилятор (и вообще, любой транслятор) состоит, по крайней мере, из трех модулей: лексического анализатора, синтаксического анализатора и генератора кода.

Первая и, пожалуй, самая несложная часть — **лексический анализатор — занимается преобразованием исходного текста программы в последовательность специальных символов — *токенов***. На низшем уровне любая программа состоит из обычных ASCII символов, но воспринимать, например, ключевое слово `for` языка C# как последовательность трех букв `f`, `o` и `r` на этапе синтаксического анализа бессмысленно.

Ключевые слова, такие как `for`, `while` или `using`, являются одиночными символами, «токенами» программы в том смысле, что их теоретически возможное разложение на символы более низкого уровня не дает нам никаких новых знаний о программе. То же самое можно сказать об опе-

[59] Все буквы в названии латинские, поэтому читать, видимо, следует «коко» (во всяком случае, не «сосо»).

рациях и прочих обозначениях языка, состоящих из двух или более ASCII символов, но фактически представляющих собой единую конструкцию. Например, последовательность /* означает символ начала комментария, по смыслу никак не выводящийся из знаков деления и умножения.

То же самое можно сказать и о таких элементах программы, как идентификаторы и константы. Лексический анализатор, с одной стороны, не обязан определять, является ли `MyId_123` именем класса, переменной или функции; с другой же стороны, он должен избавить синтаксический анализатор от необходимости распознавать набор символов M, y, I, d, _, 1, 2, 3 как некую единую конструкцию, являющуюся идентификатором.

Преобразование символьной последовательности -1.232 в количественное значение — задача лексического анализатора, использование полученного значения его уже не должно заботить.

Возможно, вы уже заметили, что распознавание идентификаторов, ключевых слов и констант — это задача, решаемая при помощи регулярных выражений. На практике конструирование лексического анализатора происходит следующим образом. Вы предоставляете специальной программе-генератору лексических анализаторов спецификацию токенов языка (с помощью некоторой разновидности регулярных выражений), а программа печатает листинг лексического анализатора, выполняющего распознавание требуемых токенов.

Как известно, если некоторая строка задается регулярным выражением, распознать ее проще всего с помощью соответствующего конечного автомата. Обычно генераторы лексических анализаторов так и поступают: окинув взглядом полученный листинг анализатора, вы без труда заметите в нем характерные для конечных автоматов конструкции.

Генератор лексических анализаторов может быть отдельной программой (классический пример — Lex), а может и встраиваться в компилятор компиляторов, как в случае Coco/R. Единственное замечание: в Coco/R лексический анализатор называется *сканером* (scanner).

Устройство синтаксического анализатора в книге уже обсуждалось. Синтаксис языка описывается с помощью некоторой разновидности контекстно-свободной грамматики, на основании которой парсер может быть сгенерирован автоматически. Чем больше ограничений в правилах грамматики, тем проще и быстрее синтаксический анализатор, но тем уже круг синтаксических средств, этой грамматикой выразимых.

Обратите внимание, что при обсуждении парсеров мы вообще не рассматривали задачу лексического анализа: каждый символ входной строки всегда считался отдельным токеном.

Напомню, что обычный подход к решению задачи синтаксического анализа состоит в имитации работы детерминированного магазинного автомата. Алгоритм CYK для реальных компиляторов оказывается слишком медленным, а рекурсивный LL(1)-анализатор — слишком ограниченным. Проект Coco/R, однако, использует именно LL(1)-подход. При этом в грамматику вносятся дополнительные возможности[60], на практике расширяющие класс распознаваемых языков до LL(k).

Предположим, что синтаксический анализатор создан. Что дальше? Ключевым аспектом синтаксического анализатора я бы назвал возможность не только принимать/отвергать входную строку, но и в каждый момент времени определять используемое правило вывода (благодаря чему можно построить дерево разбора). Ранее при определении текущего правила вывода мы просто печатали его на экране, и это называлось «синтаксическим анализом». Но при желании на каждое правило вывода можно «навесить» свое собственное действие.

Например, если при обнаружении строки `a : Integer;` выводить в выходной файл строку `int a;`, мы получим процедуру, входящую в транслятор Pascal → C. Если, распознав управляющую конструкцию, немедленно ее выполнить, парсер превратится в классический интерпретатор. Если же переводить каждую анализируемую инструкцию во фрагмент машинного кода, получится компилятор.

Для описания токенов используются регулярные выражения, для определения синтаксических конструкций — формальные грамматики. А какое средство, какой язык пригоден для составления действий, соответствующих тем или иным распознаваемым конструкциям?

Ответ, возможно, одновременно очевиден и неожиданен. Какой язык используется для описания действий? Например, сортировки? Конечно, любой обычный язык программирования! Любой компилятор компиляторов должен предоставлять разработчику возможность вставлять в определенные участки описания языка фрагменты на некотором языке программирования. В Coco/R таким языком является C#. Именно это я подразумевал, в частности, когда назвал C# «базовым» языком Coco/R[61]. Кроме того, готовый компилятор, сгенерированный проектом Coco/R, будет получен в виде листинга на C#.

[60] Которые нам, впрочем, не потребуются.

[61] Существуют, правда, версии и для других языков, но нас они сейчас не интересуют.

8.2. Практический пример: транслятор простейшего языка программирования

Давайте теперь попробуем написать хотя бы самый простой проект с помощью инструмента Coco/R.

Я предлагаю создать транслятор простейшего языка программирования в набор более строгих по синтаксису операций, которые затем интерпретируются.

Быть может, выбор конструкций языка покажется вам несколько странным; на самом деле он навеян устройством служебного языка одного проекта, над которым мне довелось работать пару лет назад.

Выбор конструкций языка

В языке предусмотрено два вида переменных: целые и булевы. Значения целых переменных всегда ограничены, и этот факт отражается в описании:

```
VarName: integer(Min..Max) := IntValue;
```

Например, описание переменной a, значение которой лежит в промежутке [-5...10] и равно изначально 3, выглядит так:

```
a: integer(-5..10) := 3;
```

Все элементы этой записи обязательны. Если в процессе работы программы значение переменной выходит за пределы указанного промежутка, сообщается об ошибке времени выполнения.

Описание булевой переменной выглядит проще:

```
VarName: boolean := BoolValue;
```

В качестве начального значения булевой переменной используется одна из предопределенных констант true или false.

Программа оформляется в стиле, похожем на паскалевский:

```
program НазваниеПрограммы.
var   // описания переменных
      // ...
begin
      // инструкции
      // ...
end.
```

Комментарии начинаются с последовательности // и заканчиваются символом конца строки. Любые пробельные символы игнорируются, располагать инструкции можно свободно (хоть по десять штук на одной строке).

В нашем языке будет всего три управляющие конструкции: присваивание, ветвление и безусловный переход.

Допускаются три различные формы присваивания.

1. простое присваивание: `let a := b;` (для любых переменных);
2. присваивание с операцией not: `let a := not b;` (только для булевых переменных);
3. присваивание с бинарной операцией: `let a := b op c;` (где `op` есть «+» или «–» для целых переменных, «and» или «or» для булевых).

Ветвление представляет собой обычную инструкцию `if` с необязательной частью `else`:

```
if a ifop b goto метка1; [else goto метка2;]
```

Здесь в качестве `ifop` могут использоваться операции «=», «<>», «<», «<=», «>» и «>=». Для булевых значений допустимы лишь операции «=» и «<>».

Безусловный переход имеет самый бесхитростный синтаксис:

```
goto метка;
```

Любая инструкция может быть предварена меткой, используемой в if- и goto-конструкциях. Строка `end.` не считается инструкцией и не может быть помечена.

Таким образом, например, сосчитать сумму чисел 1, 2, 3, 4, 5 можно при помощи следующей программы:

Листинг 8.1. Подсчет суммы чисел

```
program CountSum.

var s: integer(0..100) := 0; // сумма
    b: integer(1..6) := 1;   // счетчик

begin
loop: let s := s + b;
      let b := b + 1;
      if b < 6 goto loop;
end.
```

Операторов печати не предусмотрено. Результаты работы программы можно будет узнать по содержимому переменных.

Вероятно, у вас уже возник вопрос: почему операция присваивания требует указания ключевого слова `let`? Почему вместо `let s := s +b;` нельзя

записать просто `s := s + b;`? Дело в том, что исключение слова `let` приведет к проблемам с LL(1) формой. Напомню, при LL(1) разборе считывания одного-единственного токена на каждом этапе разбора должно быть достаточно для принятия правильного решения о его смысле.

Предположим, в начале очередной инструкции синтаксический анализатор встретил токен, представляющий собой идентификатор. Если разрешить присваивание без `let`, то немедленно возникает затруднение: чем является этот идентификатор — меткой инструкции или именем переменной в левой части присваивания? Как я уже говорил, Coco/R включает механизмы для разрешения подобных неоднозначностей, но зачем нам создавать себе лишние проблемы в таком простом языке?

Выбор промежуточного представления

Хотя инструкции нашего языка (назовем его TinyCode) можно выполнять и непосредственно, все-таки проще сначала преобразовать их в некоторое «промежуточное представление». В отличие от полностью текстового исходного описания программы, промежуточное представление будет состоять из трех различных элементов:

1. хэш-таблицы, содержащей переменные программы;
2. упрощенного листинга программы, в котором отсутствуют описания переменных и метки, а каждая инструкция записана на отдельной строке;
3. хэш-таблицы, сопоставляющей символическим именам меток номера соответствующих строк упрощенного листинга.

На содержимом упрощенного листинга программы имеет смысл остановиться немного подробнее. Можно, конечно, просто исключить метки и разбросать инструкции по отдельным строкам, но мне кажется, что вид инструкций поддается дальнейшему упрощению:

1. Запись вида `let a := b;` превращается в `asgn a b`.
2. Запись вида `let a := not b;` превращается в `asgn a not b`.
3. Присваивание с бинарной операцией `let a := b op c;` записывается как `asgn a b op c`.
4. В простом ветвлении `if x ifop y goto label;` удаляется точка с запятой: `if x ifop y goto label`.
5. Полное ветвление (с else) `if x ifop y goto label1; else goto label2;` превращается в `if x ifop y goto label1 elsegoto label2`.
6. В конец программы добавляется команда `end`.

Немаловажно отметить, что составляющие упрощенных инструкций всегда разделены ровно одним пробелом.

8.3. Генерация лексического и синтаксического анализаторов

Основные правила описания языка в Coco/R

Как уже отмечалось, Coco/R генерирует как лексический, так и синтаксический анализаторы по заданному описанию языка. Описание представляет собой обычный текстовый файл с расширением `atg` (в нашем случае разумно назвать его `tinycode.atg`), содержащий инструкции, распознаваемые программой Coco/R.

Описание языка начинается с указания его названия вслед за ключевым словом `COMPILER`:

```
COMPILER TinyCode
```

Далее после директивы `CHARACTERS` можно указать имена, обозначающие любой символ из данного набора:

```
CHARACTERS
    letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijkl
             mnopqrstuvwxyz".
    digit = "0123456789".
```

Эти определения позволяют теперь использовать слова `letter` и `digit` вместо явного перечисления соответствующих наборов символов.

Теперь можно указать список токенов, распознаваемых лексическим анализатором. Как уже говорилось выше, токены описываются при помощи некоторой разновидности регулярных выражений. В Coco/R приняты следующие соглашения:

- Выражение, заключенное в фигурные скобки, может повторяться нуль или более раз (замыкание Клини).
- Вертикальная черта обозначает альтернативу (объединение в терминах регулярных выражений).
- В квадратные скобки заключается необязательная часть: запись [a] эквивалентна $a \cup \varepsilon$ в стандартной нотации.
- Каждая запись завершается точкой.

Таким образом, идентификатор (Identifier) — это буква, за которой следует любое количество букв или цифр. Число (Number) состоит из необязательного знака «минус» и любого не меньшего единицы количества цифр. Токены `true` и `false` определяются соответствующими строками:

```
TOKENS
    Identifier = letter {letter | digit}.
    Number = ['-'] digit {digit}.
    true = "true".
    false = "false".
```

Строковые константы вроде `true` или `false` необязательно перечислять в списке токенов. Я сделал это лишь потому, что в дальнейшем на эти константы придется ссылаться, по крайней мере, два раза. В этом случае присвоение имен мне кажется разумным. А вообще Coco/R автоматически преобразует в неименованный токен любую последовательность символов, заключенную в кавычки.

После секции токенов можно задать синтаксис комментариев и множество игнорируемых символов:

```
COMMENTS FROM "//" TO '\n'
IGNORE '\r' + '\n' + '\t'
```

Комментарии считаются важным элементом любого языка, поэтому Coco/R выделяет их в отдельный блок (кроме того, вложенные комментарии невозможно задать с помощью регулярных выражений). Строка `COMMENTS FROM "//" TO '\n'` указывает, что комментарии языка начинаются с последовательности "//" и заканчиваются символом конца строки '\n'.

Множество символов, записанное после ключевого слова `IGNORE`, обозначает пропускаемые символы, не несущие смысловой нагрузки в описываемом языке. Пробел всегда считается несущественным. Кроме пробела я добавил к списку незначащих символов перевод строки, возврат каретки и символ табуляции.

Теперь можно приступить к описанию правил вывода языка с помощью формальной грамматики. В правой части любого правила грамматики можно использовать нетерминалы, токены, строковые константы, а также те же символы для выделения необязательных фрагментов, повторений и альтернатив, что использовались при определении токенов. Левая часть правила будет представлять собой некоторый нетерминал; имя стартового символа грамматики совпадает с названием конструируемого транслятора. Любое правило завершается точкой.

Не совсем понятно? Не беспокойтесь, сейчас все прояснится.

Синтаксические правила описываются после ключевого слова PRODUCTIONS. В первую очередь можно задать правило, содержащее в левой части стартовый символ `TinyCode` (совпадающий с названием нашего транслятора):

```
TinyCode = "program" Identifier "." ProgramBody.
```

Итак, программа на языке `TinyCode` состоит из строки "program", за которой следует некоторый идентификатор (токен `Identifier`), точка и нетерминал `ProgramBody`.

Тело программы (`ProgramBody`) определяется следующим образом:

```
ProgramBody = ["var" DeclareVar {DeclareVar}]
              "begin" {Statement} "end.".
```

По-русски эту запись можно сформулировать так. Тело состоит из необязательной секции `var`, состоящей из слова `var` и произвольного (но не меньшего единицы) количества описаний переменных (`DeclareVar`). За секцией переменных следует слово `begin`, любое количество инструкций и маркер конца программы «`end.`».

С секцией описания переменных связаны два нетерминала — `DeclareVar` и `DeclareVarBody`:

```
DeclareVar = Identifier ":" DeclareVarBody.
DeclareVarBody = "integer" "(" Number ".." Number ")""":="
                 Number ";"| "boolean" ":=" (true | false) ";".
```

За именем любой переменной следует «тело описания» `DeclareVarBody`, различное для целых и для булевых переменных.

Инструкция (`Statement`) состоит из необязательной метки и собственно инструкции, являющейся присваиванием, ветвлением или переходом:

```
Statement = [Identifier ":"] StatementBody.
StatementBody = Assignment | Branching | Goto.
```

Проще всего описать, конечно, переход. Это всего лишь слово `goto`, за которым следует идентификатор метки и знак «точка с запятой»:

```
Goto = "goto" Identifier ";".
```

С присваиванием и ветвлением дела обстоят несколько сложнее. В присваивании мы, во-первых, встречаемся с тремя различными синтаксисами:

```
Assignment = "let" LValue ":=" (RValue [OP RValue] |
             "not" RValue) ";".
OP = '+' | '-' | "and" | "or".
```

Во-вторых, здесь появляются понятия `LValue` (левая часть инструкции присваивания, представляющая собой переменную) и `RValue` (правая часть инструкции, являющаяся либо переменной, либо значением):

```
LValue = Identifier.
RValue = Identifier | Number | true | false.
```

Ветвление несколько проще присваивания:

```
Branching = "if" CompareExpr "goto" Identifier ";"
                             ["else" "goto" Identifier ";"].
CompareExpr = RValue CompareOp RValue.
CompareOp = '=' | "<>" | '<' | "<=" | '>' | ">=".
```

Описание языка заканчивается директивой END:

```
END TinyCode.
```

Полное описание языка TinyCode в стандарте Coco/R

Итак, синтаксический анализатор для языка TinyCode полностью готов! Вернее, готово исходное описание TinyCode для Coco/R (полный текст файла `tinycode.atg` показан в листинге 8.2).

Листинг 8.2. Описание синтаксиса языка TinyCode (в стандарте проекта Coco/R)

```
COMPILER TinyCode

CHARACTERS
 letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
 digit = "0123456789".

TOKENS
    Identifier = letter {letter | digit}.
    Number = ['-'] digit {digit}.
    true = "true".
    false = "false".

COMMENTS FROM "//" TO '\n'
IGNORE '\r' + '\n' + '\t'

PRODUCTIONS
    TinyCode = "program" Identifier "." ProgramBody.
    ProgramBody = ["var" DeclareVar {DeclareVar}] "begin"
                  {Statement} "end.".

    DeclareVar = Identifier ":" DeclareVarBody.
    DeclareVarBody = "integer" "(" Number ".." Number ")"
                     ":=" Number ";"
                     | "boolean" ":=" (true | false) ";".

    Statement = [Identifier ":"] StatementBody.
    StatementBody = Assignment | Branching | Goto.

    Goto = "goto" Identifier ";".

    Assignment = "let" LValue ":=" (RValue [OP RValue] |
                 "not" RValue) ";".
    OP = '+' | '-' | "and" | "or".
    LValue = Identifier.
    RValue = Identifier | Number | true | false.
```

```
    Branching = "if" CompareExpr "goto" Identifier ";"
                                    ["else" "goto" Identifier ";"].
    CompareExpr = RValue CompareOp RValue.
    CompareOp = '=' | "<>" | '<' | "<=" | '>' | ">=".

END TinyCode.
```

Генерация и компиляция синтаксического анализатора

Сгенерировать исходный код синтаксического анализатора теперь можно с помощью вызова Coco/R:

```
coco.exe tinycode.atg
```

Если не произойдет ничего неожиданного, вы увидите на экране сообщение об успешном создании лексического и синтаксического анализаторов:

```
Coco/R (Oct.27, 2004)
checking
parser + scanner generated
0 errors detected
```

В текущем каталоге появятся два файла: `Scanner.cs` (лексический анализатор) и `Parser.cs` (синтаксический анализатор). Перед тем, как компилировать полноценный exe-файл, придется еще вручную создать главный класс программы (его определение можно поместить, например, в файл `Compiler.cs`). В простейшем варианте он займет всего несколько строк:

```
using System;

public class Compiler
{
    public static void Main(string[] arg)
    {
        Scanner.Init(arg[0]);
                          // запуск лексического анализатора
        Parser.Parse();   // запуск синтаксического
                          // анализатора
        Console.WriteLine("Ошибок: " + Errors.count);
                          // количество ошибок
    }
}
```

Предполагается, что имя входного (анализируемого) файла передается в качестве первого аргумента командной строки. Программа выполняет сначала лексический анализ, затем синтаксический анализ и печатает

сообщение о количестве найденных ошибок. Если ошибки обнаружатся, их описания будут выведены синтаксическим анализатором автоматически.

Теперь можно запустить компилятор C# для создания исполняемого файла проекта[62]:

```
csc.exe compiler.cs scanner.cs parser.cs
```

Тестирование синтаксического анализатора

Если подать на вход полученному исполняемому файлу compiler.exe программу подсчета суммы чисел (см. листинг 8.1), мы получим ожидаемый ответ о ее синтаксической корректности:

```
Ошибок: 0
```

Можно попробовать намеренно внести в программу пару синтаксических ошибок, чтобы убедиться в адекватной реакции парсера:

```
var s: integer(0..) := 0;     // удалили верхнее число
                              // промежутка 0..100
...
loop: let s := s + b          // удалили завершающую точку
                              // с запятой
```

Теперь синтаксический анализатор, как и ожидалось, выводит два сообщения о найденных ошибках:

```
-- line 3 col 19: Number expected
-- line 8 col 7: ";" expected
Ошибок: 2
```

8.4. Создание промежуточного представления программы

Предварительные замечания

Полученный «транслятор» `compiler.exe` еще, конечно, не является транслятором. На данный момент у нас есть лишь связка из лексического и синтаксического анализаторов, умеющая определять синтаксическую корректность введенной программы.

[62] Если путь к компилятору csc.exe не прописан в переменной PATH, можно явно воспользоваться полным путем. Например, на моей машине проект компилируется при помощи команды `C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\csc.exe compiler.cs scanner.cs parser.cs`.

Напомню, что итоговое приложение должно состоять из только что сгенерированных анализаторов, генератора промежуточного представления и интерпретатора.

Исходная программа на TinyCode поступает на вход анализаторам (сначала лексическому, а затем и синтаксическому). Специальные команды, внедренные в код синтаксического анализатора (мы их еще не написали), создают промежуточное представление, эквивалентное исходной программе. Затем интерпретатор выполняет его шаг за шагом.

Действия, отвечающие за генерацию промежуточного представления программы, вставляются прямо в описание грамматики языка между скобками «(.» и «.)». В каждом таком блоке (записанном на языке C#) программисту доступен объект `t` типа `Token`, предоставляемый Coco/R. Нам потребуется только одно его поле `val`, хранящее значение последнего распознанного токена (видимо, авторам было до невозможности лень дописать две буквы, чтобы получить осмысленное имя `value`).

Кроме действий, записываемых внутри блока (. ... .), в области между ключевыми словами `COMPILER` и `CHARACTERS` можно описывать любые функции и глобальные переменные: Coco/R просто скопирует их в определение класса синтаксического анализатора. В самом начале файла `tinycode.atg` (до слова COMPILER) разрешается поместить список директив `using`.

Программирование генератора промежуточного кода

Приступим к созданию генератора промежуточного кода. Для начала нам потребуется описать несколько глобальных переменных:

```
using System.Collections;

COMPILER TinyCode

static Hashtable bools = new Hashtable();
                         // булевы переменные программы
static Hashtable ints = new Hashtable();
                         // целые переменные программы
static ArrayList program = new ArrayList();
                         // промежуточное представление
static Hashtable labels = new Hashtable();
                         // метки
static int StatementNo = 0;
                         // номер текущей инструкции
static string cmd;       // переменная для формирования
                         // текущей команды
```

```
:atic string VarName;     // имя текущей рассматриваемой
                          // переменной
:atic string ProgName;    // название программы

HARACTERS
'...
```

:ерь можно перейти в секцию PRODUCTIONS и добавить действия, ечающие за генерацию промежуточного кода:

```
RODUCTIONS
'...
.nyCode = "program" Identifier    (. ProgName = t.val; .)
          "." ProgramBody.

:ogramBody = ["var" DeclareVar {DeclareVar}] "begin"
             {Statement} "end."(. program.Add("end"); .)
             .
'...
```

инение правильных действий — задача достаточно трудоемкая и чре- ия ошибками. Поэтому будем продвигаться медленно, шаг за шагом.

ограмма на языке TinyCode состоит из ключевого слова `program`, за ко- ым следует название (токен `Identifier`), точка и тело (`ProgramBody`). только название программы распознано, мы можем извлечь его из пере- ной `t.val` и записать в `ProgName`. Далее, после анализа тела программы., нец листинга промежуточного представления добавляется команда `end` струкция `program.Add("end");`).

Описания переменных

исания переменных содержат больше интересующих нас токенов:

```
:clareVar = Identifier    (. VarName = t.val; .)
    ":" DeclareVarBody.
:clareVarBody = "integer"
    "(" Number (. int LBound = Convert.ToInt32(t.val); .)
    ".." Number (. int UBound = Convert.ToInt32(t.val); .)
    ")" ":=" Number (. int Value = Convert.ToInt32(t.val); .)
    ";" (. if(Value < LBound || Value > UBound)
              throw new Exception("выход за диапазон!");
```

```
                    ints.Add(VarName,
                         new IntVar(LBound, UBound, Value)); .)
    | "boolean"
     ":=" (true | false) (. bool Value = Convert.ToBoolean(t.val); .)
      ";" (. bools.Add(VarName, new BoolVar(Value));  .)
      .
```

Общая идея обработки описания любой переменной такова. Сначала считывается идентификатор переменной и записывается в `VarName`. Затем выполняется алгоритм, специфичный для типа переменной. Если описываемая переменная — целая, считываются верхняя и нижняя границы допустимого диапазона, а также начальное значение.

Обратите внимание, что значение `t.val` имеет строковый тип, поэтому приходится выполнять его преобразование в число. Если описание корректно, переменная добавляется в соответствующую хэш-таблицу. Аналогичные действия производятся и в случае булевого значения.

Переменные хранятся в виде объектов, принадлежащих типам `IntVar` и `BoolVar`, описанным следующим образом (не забывайте, что описания типов должны располагаться до секции CHARACTERS):

```
struct IntVar   // целая переменная
{
    public int LBound, UBound, Value;
    public IntVar(int lb, int ub, int v)
                  {LBound = lb; UBound = ub; Value = v;}
}

struct BoolVar   // булева переменная
{
    public bool Value;
    public BoolVar(bool v) {Value = v;}
}
```

При анализе инструкций языка нельзя забывать заносить метки в таблицу labels:

```
Statement = [Identifier (. labels.Add(t.val, StatementNo); .)
            ":"] StatementBody.
```

Переходы, присваивания и ветвления

Запрограммируем теперь обработку переходов, присваиваний и ветвлений. Переходу соответствует команда `goto метка` промежуточного представления:

```
Goto = "goto" Identifier  (. program.Add("goto " + t.val);
                             StatementNo++; .)
       ";".
```

рисваивание преобразуется в команду `asgn`:

```
Assignment = "let" LValue (. cmd = "asgn " + t.val + " "; .)
             ":=" (RValue (. cmd += t.val;  .)
             [OP          (. cmd += " " + t.val;  .)
             RValue       (. cmd += " " + t.val;  .)
             ]
             |
             not          (. cmd += "not ";  .)
             RValue       (. cmd += t.val;  .)
             ) ";"    (. program.Add(cmd); StatementNo++;  .)
             .
```

сталось внести изменения в описание инструкции ветвления:

```
Branching = "if"       (. cmd = "if "; .)
       CompareExpr
       "goto"          (. cmd += " goto "; .)
       Identifier      (. cmd += t.val; .)
       ";"
       ["else" "goto"  (. cmd += " elsegoto "; .)
       Identifier      (. cmd += t.val; .)
       ";"]            (. program.Add(cmd); StatementNo++;  .)
       .

CompareExpr = RValue     (. cmd += t.val; .)
              CompareOp  (. cmd += " " + t.val + " "; .)
              RValue     (. cmd += t.val; .)
```

Вывод промежуточного кода

осле компиляции измененного файла `tinycode.atg` должно полу-ться приложение, сохраняющее переменные и метки в хэш-таблицах, акже генерирующее промежуточный код входной программы на языке nyCode. Проблема лишь в том, что результаты этой работы пока что возможно увидеть. Впрочем, доработать наш транслятор несложно.)-первых, добавим в файл `tinycode.atg` описание несложного метода nProgram(), выводящего на экран сгенерированное промежуточное исание:

```
public static void RunProgram()
{
    Console.WriteLine("variables:");
    // булевы переменные
    for(IDictionaryEnumerator e = bools.GetEnumerator();
                                  e.MoveNext(); )
        Console.WriteLine("bool: " + e.Key + " = " +
                          ((BoolVar)e.Value).Value);
    // целые переменные
    for(IDictionaryEnumerator e = ints.GetEnumerator();
                                  e.MoveNext(); )
    {
        IntVar v = (IntVar)e.Value;
        Console.WriteLine("int(" + v.LBound + ".." +
                          v.UBound + "): " +  e.Key +
                          " = " + v.Value);
    }

    Console.WriteLine("\nlabels:");  // метки
    for(IDictionaryEnumerator e = labels.GetEnumerator();
                          e.MoveNext(); )
        Console.WriteLine(e.Key + ": " + e.Value);

    Console.WriteLine("\nintermediate code:");
                                  // промежуточный код
    for(int i = 0; i < program.Count; i++)
        Console.WriteLine(program[i]);
}
```

Во-вторых, добавим вызов `RunProgram()` в описание метода `Main()` класса `Compiler` (файл `Compiler.cs`):

```
public static void Main(string[] arg)
{
    Scanner.Init(arg[0]);
    Parser.Parse();
    if(Errors.count > 0)
        Console.WriteLine("Ошибок: " + Errors.count);
    else
        Parser.RunProgram();
}
```

Теперь при анализе программы подсчета суммы (листинг 8.1) будет выведена такая информация о промежуточном коде:

```
variables:
int(0..100): s = 0
int(1..6): b = 1
```

```
labels:
loop: 0

intermediate code:
asgn s s + b
asgn b b + 1
if b < 6 goto loop
end
```

Метке `loop` соответствует число 0, поскольку инструкции у нас нумеруются, начиная с нуля. Таким образом, инструкция `asgn s s + b` — нулевая по счету.

Итоговое описание генератора промежуточного кода

Перед тем, как перейти к следующему разделу, я хотел бы полностью привести содержимое секции PRODUCTIONS описания нашего транслятора (см. листинг 8.3). Больше оно изменяться не будет.

Листинг 8.3. Содержимое секции PRODUCTIONS описания транслятора TinyCode

```
TinyCode = "program" Identifier    (. ProgName = t.val; .)
           "." ProgramBody.

ProgramBody = ["var" DeclareVar {DeclareVar}] "begin" {Statement}
                    "end."   (. program.Add("end"); .)


DeclareVar = Identifier      (. VarName = t.val; .)
             ":" DeclareVarBody.
DeclareVarBody = "integer"
    "(" Number          (. int LBound = Convert.ToInt32(t.val); .)
    ".." Number         (. int UBound = Convert.ToInt32(t.val); .)
    ")" ":=" Number     (. int Value = Convert.ToInt32(t.val); .)
    ";"                 (. if(Value < LBound || Value > UBound)
                               throw new Exception("value is
                                                   out of bounds!");
                           ints.Add(VarName,
                           new IntVar(LBound, UBound, Value));
                       .)
  | "boolean"
```

```
    ":=" (true | false) (. bool Value = Convert.ToBoolean(t.val); .)
    ";"                 (. bools.Add(VarName, new BoolVar(Value)); .)
    .

Statement = [Identifier (. labels.Add(t.val, StatementNo); .)
            ":"] StatementBody.

 StatementBody = Assignment | Branching | Goto.

 Goto = "goto" Identifier (. program.Add("goto " + t.val);
                             StatementNo++; .) ";".

Assignment = "let" LValue (. cmd = "asgn " + t.val + " "; .)
             ":=" (RValue  (. cmd += t.val;  .)
             [OP           (. cmd += " " + t.val;  .)
             RValue        (. cmd += " " + t.val;  .)
             ]
             |
             not          (. cmd += "not ";  .)
             RValue       (. cmd += t.val;  .)
             ) ";"        (. program.Add(cmd); StatementNo++;
                          .)
             .
LValue = Identifier.
RValue = Identifier | Number | true | false.
OP = '+' | '-' | "and" | "or".

Branching = "if"          (. cmd = "if "; .)
       CompareExpr
       "goto"             (. cmd += " goto "; .)
       Identifier         (. cmd += t.val; .)
       ";"
       ["else" "goto"     (. cmd += " elsegoto "; .)
       Identifier         (. cmd += t.val; .)
       ";"]               (. program.Add(cmd); StatementNo++;  .)
       .

CompareExpr = RValue      (. cmd += t.val; .)
              CompareOp   (. cmd += " " + t.val + " "; .)
              RValue      (. cmd += t.val; .)
              .
CompareOp = '=' | "<>" | '<' | "<=" | '>' | ">=".
```

8.5. Интерпретация промежуточного кода

Замечания о качестве синтаксического анализатора

Вот мы и добрались до последнего модуля нашего проекта — интерпретатора промежуточного кода. Наверно, вы уже обратили внимание, что вместо генерации промежуточного кода можно было бы легко преобразовать программу на TinyCode в программу на Паскале или С, тем самым получив полноценный транслятор TinyCode. Мне, однако, хотелось реализовать полноценную среду, выполняющую любую программу на TinyCode. Учитывая, что самое сложное уже все равно позади, потратить еще несколько минут на интерпретатор — не слишком большая жертва.

Перед тем, как перейти к интерпретатору, я бы хотел сказать еще пару слов по поводу только что созданного генератора промежуточного кода.

Даже если оставить в стороне обсуждение многочисленных недостатков языка TinyCode, синтаксический анализатор тоже можно покритиковать. Пытаясь уменьшить размер анализатора, я намеренно упростил его функции. Например, программа

```
program MyProg.

var s: integer(0..100) := 0;
    b: boolean := true;

begin
    let s := b;
end.
```

будет успешно скомпилирована, хотя присваивание целой переменной булевого значения не имеет смысла. Эта ошибка будет выявлена лишь на этапе выполнения программы. Точно также ошибкой времени выполнения будет считаться переход по недопустимой метке, хотя, в принципе, синтаксический анализатор мог бы такую ситуацию отловить, но программирование этих действий, конечно, требует отдельных усилий.

Вообще говоря, чем больше ошибок переносятся с этапа выполнения на этап компиляции, тем лучше. В чисто интерпретируемых программах простейшие синтаксические ошибки могут неделями оставаться невыявленными, если содержащий их код не вызывается.

Программирование интерпретатора

Для интерпретации программы нам потребуется ввести новую глобальную переменную

```
static int InstructionPtr = 0;        // указатель выполняемой
                                      // инструкции
```

и добавить несколько строк в конец метода RunProgram():

```
public static void RunProgram()
{
    //...
    // пока текущая инструкция не "end",
    // продолжать выполнение
    Console.WriteLine("\nRunning " + ProgName);
    while((string)program[InstructionPtr] != "end")
        ExecuteStatement();

    // распечатать значения всех переменных
    // после работы программы
    Console.WriteLine("\nvariables:");
    for(IDictionaryEnumerator e = bools.GetEnumerator();
                              e.MoveNext(); )
        Console.WriteLine("bool: " + e.Key + " = " +
                          ((BoolVar)e.Value).Value);
    for(IDictionaryEnumerator e = ints.GetEnumerator();
                              e.MoveNext(); )
    {
        IntVar v = (IntVar)e.Value;
        Console.WriteLine("int(" + v.LBound + ".." + v.UBound +
                          "): " +  e.Key + " = " + v.Value);
    }
}
```

Оставшиеся методы интерпретатора показаны в листинге 8.4.

Листинг 8.4. Ядро интерпретатора языка TinyCode

```
// вернуть значение RValue по строковому имени
static Object GetRValue(string value)
{
    if(bools.ContainsKey(value))
                        // если value  -  булева переменная
        return bools[value];
    if(ints.ContainsKey(value))
                        // если value  -  целая переменная
        return ints[value];

    // если value  -  булева константа
    if(value == "true" || value == "false")
         return new BoolVar(Convert.ToBoolean(value));

    // иначе value  -  целая константа
```

```
  int v = Convert.ToInt32(value);
  return new IntVar(v - 1, v + 1, v);


выполнить бинарную операцию rvalue1 op rvalue2
atic Object PerformOperation(Object rvalue1,
                             Object rvalue2, string op)

операции and и or работают только для булевых переменных
  if(op == "and")
      return new BoolVar(((BoolVar)rvalue1).Value &&
                         ((BoolVar)rvalue2).Value);
  if(op == "or")
      return new BoolVar(((BoolVar)rvalue1).Value ||
                         ((BoolVar)rvalue2).Value);

  int value = 0;
  // операции + и - работают, соответственно,
  // только для целых переменных
  if(op == "+")
     value = ((IntVar)rvalue1).Value + ((IntVar)rvalue2).Value;
  if(op == "-")
      value = ((IntVar)rvalue1).Value - ((IntVar)rvalue2).Value;

  return new IntVar(value - 1, value + 1, value);


выполнить операцию присваивания: на вход поступает строка
команды, разбитая на элементы (см. метод ExecuteStatement())
atic void ExecuteAssignment(string[] v)

  if(v[2] == "not")   // NOT-присваивание (asgn x not y)
  {                   // v[0] = "asgn", v[1] = x,
                      // v[2] = "not", v[3] = y
      BoolVar rvalue = (BoolVar)GetRValue(v[3]);
                      // получить значение y
      rvalue.Value = !rvalue.Value;
                      // выполнить отрицание
      bools[v[3]] = rvalue;
                      // записать новое значение
  }
  else                // asgn x y [op z]
  {                   // v[0] = "asgn", v[1] = x, v[2] = y
      Object rvalue1 = GetRValue(v[2]);
                      // получить значение y

      if(v.Length > 3) // часть op z присутствует
```

```
        {
            Object rvalue2 = GetRValue(v[4]);
                        // получить значение z

            // выполнить операцию y op z
            rvalue1 = PerformOperation(rvalue1, rvalue2, v[3]);
        }

        if(bools.ContainsKey(v[1]))
                        // выполнить присваивание x := rvalue1
            bools[v[1]] = rvalue1;
                        // алгоритм зависит от типа переменной x
        if(ints.ContainsKey(v[1]))
        {
            IntVar val = (IntVar)ints[v[1]];
            int newvalue = ((IntVar)rvalue1).Value;
            if(newvalue < val.LBound || newvalue > val.UBound)
                throw new Exception("value is out of bounds!");
            ints[v[1]] = new IntVar(val.LBound, val.UBound,
                                    newvalue);
        }
    }

    InstructionPtr++;   // перейти к следующей инструкции
}
static void ExecuteBranching(string[] v)
                        // выполнить операцию ветвления
{
    Object lhs = GetRValue(v[1]);     // v[0] = "if", v[1] = x
    Object rhs = GetRValue(v[3]);     // v[2] = op, v[3] = y
    string op = v[2];   // v[4] = "goto", v[5] = метка1
    bool result = false; // v[6] = "elsegoto", v[7] = метка2

    // для булевых значений результат определен
    // только для операций = и <>
    if(lhs is BoolVar)
    {
        bool l = ((BoolVar)lhs).Value, r = ((BoolVar)rhs).Value;
        result = (op == "=" && l == r) || (op == "<>" && l != r);
    }
    else     // для целых допустимы все операции
    {
        int l = ((IntVar)lhs).Value, r = ((IntVar)rhs).Value;
        result = (op == "<" && l < r) || (op == ">" && l > r) ||
                 (op == "<=" && l <= r) || (op == ">=" && l >= r) ||
                 (op == "="  && l == r) || (op == "<>" && l != r);
    }
```

```
    if(result == true)  // выполняем переход по метке
        InstructionPtr = Convert.ToInt32(labels[v[5]]);
    else if(v.Length > 6) // если часть elsegoto присутствует
        InstructionPtr = Convert.ToInt32(labels[v[7]]);
    else   // просто выполняем переход к следующей строке
        InstructionPtr++;


tatic void ExecuteStatement()
                        // выполнить очередную инструкцию

    // разбиваем инструкцию на элементы
    // (пробел является разделителем)
    string[] v = ((string)program[InstructionPtr]).Split(' ');

    switch(v[0])        // в зависимости от типа инструкции
    {
        case "asgn": ExecuteAssignment(v); break;
                        // выполняем присваивание
        case "if":   ExecuteBranching(v);  break;
                        // выполняем ветвление
        case "goto": InstructionPtr =
                        // выполняем переход
                    Convert.ToInt32(labels[v[1]]); break;

    }
```

деюсь, дополнительные комментарии к листингу не требуются. Кроме о, интерпретация промежуточного кода все-таки не имеет прямого ношения к теме главы, и я не хотел бы уделять ей слишком много мания.

сле компилирования проекта можно попробовать запустить программу числения суммы чисел:

```
ompiler.exe countsum.txt
```

езультате сложения чисел от 1 до 5 значение переменной s должно заться равным 15. Распечатка подтверждает правильность работы ограммы:

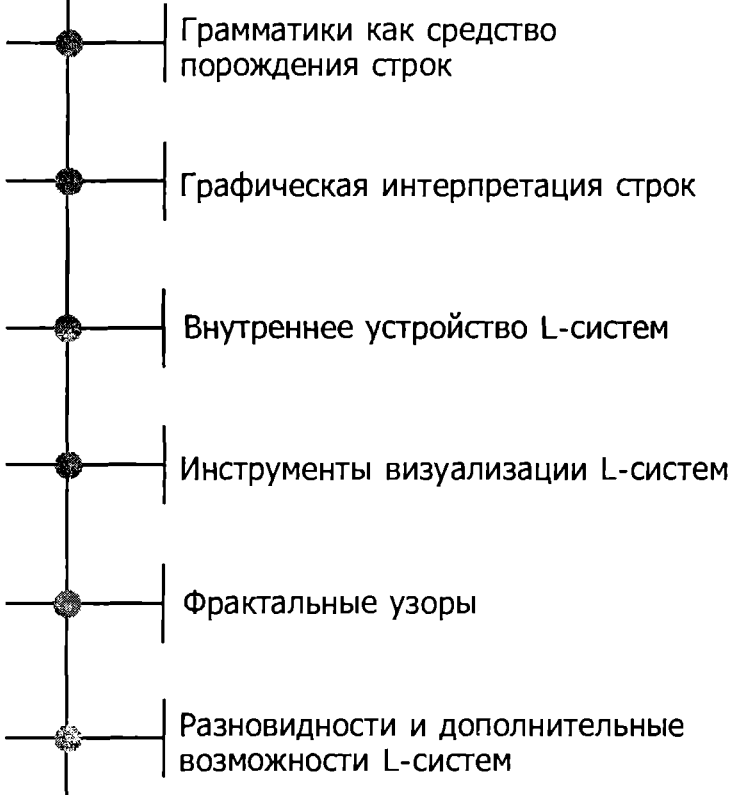
```
ariables:
nt(0..100): s = 0
nt(1..6): b = 1
```

```
labels:
loop: 0

intermediate code:
asgn s s + b
asgn b b + 1
if b < 6 goto loop
end

Running CountSum

variables:
int(0..100): s = 15
int(1..6): b = 6
```

Итоги

- Классические способы разработки компиляторов связаны с использованием уже известных нам моделей, таких как регулярные выражения, конечные автоматы, формальные грамматики, автоматы с магазинной памятью и рекурсивные синтаксические анализаторы.
- Такие важные части компилятора, как лексический и синтаксический анализаторы, могут быть сгенерированы автоматически на основе некоторого высокоуровневого описания.
- Для лексического анализатора высокоуровневое описание представляет собой ту или иную разновидность набора регулярных выражений, задающую набор токенов языка программирования. Созданный в автоматическом режиме лексический анализатор обычно основывается на идеологии детерминированного конечного автомата.
- Для синтаксического анализатора функцию высокоуровневого языка описаний выполняет какая-либо формальная грамматика. Сгенерированный синтаксический анализатор использует в качестве базовой модели автомат с магазинной памятью, или рекурсивный парсер.
- Алгоритмические действия, превращающие синтаксический анализатор в транслятор, интерпретатор или компилятор, обычно записываются на каком-либо языке общего назначения. В основе классических инструментов разработчика компиляторов обычно лежит язык C. Средство Coco/R использует в качестве базового языка C#.

Глава 9 Системы Линденмайера (L-системы)



Наверняка после рассуждений о синтаксическом анализе вам может показаться, что формальные грамматики используются только в компиляторах, парсерах, системах проверки синтаксиса и тому подобных вещах. Эта глава призвана убедить, что дело обстоит вовсе не так уж плохо. Системы Линденмайера представляют собой отличный пример применения формальных грамматик в задаче, не имеющей никакого отношения к синтаксическому анализу.

L-системы названы так в честь своего изобретателя, шведского ботаника Аристида Линденмайера, сумевшего показать, что структура растений может быть очень точно смоделирована при помощи формальных грамматик[63]

Разумеется, здесь я могу всего лишь затронуть тему L-систем. Надеюсь, знакомство с ними доставит вам удовольствие. При желании найти более основательные материалы, посвященные системам Линденмайера, не так уж и трудно.

9.1. Грамматики как средство порождения строк

Если вы помните, в предыдущих главах мы использовали грамматики исключительно для распознавания строк. Типичная задача звучала так. Дается грамматика G и строка `str`. Требуется определить, выводится ли строка `str` с помощью правил грамматики G, и если да, построить дерево разбора. Разумеется, это как раз то, что нам нужно, если речь идет об анализе какой-либо компьютерной программы или просто строки, возможно, принадлежащей тому или иному интересному языку.

Однако, в принципе, процесс можно и развернуть: начав со стартового символа грамматики, получить некоторую корректную строку языка. Рассмотрим, например, уже знакомую грамматику, описывающую арифметические выражения:

```
E → EOE | 0 | 1 | ... | 9
O → + | - | * | / | =
```

[63] P. Prusinkiewicz, A. Lindenmayer. The Algorithmic Beauty of Plants. Springer-Verlag, NY, USA, 1990.

лаве о синтаксическом анализе показывалось, как можно с ее помо-
ю убедиться, что выражение вроде 2+2=4 действительно является
ифметическим выражением. Давайте теперь развернем процесс в об-
тную сторону.

начально у нас имеется стартовый символ E. Из него можно вывести
бую цифру и закончить на этом, либо выбрать более интересный путь,
родив строку EOE:

```
→ EOE
```

рока EOE предоставляет куда больше простора для фантазии. Если
задаваться целью сгенерировать чего-нибудь подлиннее, можно про-
заменить левое E на 2, O на знак умножения, а правое E — снова
EOE:

```
OE → 2*EOE
```

ленim теперь в полученной строке левое E на 3, O на знак равенства,
равое E — на 7:

```
*EOE → 2*3=7
```

нечно, 2*3 не равно 7, но данное выражение является синтаксически
ректным, а его семантическая (смысловая) нагрузка уже не в компе-
ции синтаксического анализатора.

9.2. Графическая интерпретация строк

льшинству людей строки вроде «2*2=4» или «восхитительное клуб-
ное мороженое» кажутся осмысленными; этого не скажешь о строках
badfaa» или «файрвол — это брандмауэр». В действительности, конеч-
строки сами по себе являются лишь тем, чем они являются: наборами
следовательно записанных символов. В слове «мороженое» нет ничего,
напоминало бы о холодном лакомстве; символ «2» обозначает двойку
шь потому, что мы сами так между собой договорились.

ли строки можно интерпретировать как числа или как продукты пита-
, почему в виде строки нельзя закодировать, скажем, рисунок? Речь
т, конечно, не о какой-то размытой формулировке вроде «фикус в
шке на подоконнике», а о конкретных командах рисования на гра-
ческом дисплее.

идея была воплощена давным-давно в виде так называемой *черепа-
ей графики*. Черепашью графику напрямую поддерживает язык LOGO,
акже старые версии Бейсика (лично мне эта концепция известна со
мен Бейсика компьютера БК).

Обычно графические примитивы (точки, линии, прямоугольники) рисуют с помощью вызовов соответствующих функций. Таким образом, чтобы нарисовать «домик» из прямоугольника и двух отрезков, формирующих крышу, вам придется написать код наподобие этого (в предположении, что ось ординат направлена вниз):

```
прямоугольник(0, 200, 100, 100); // (X1, Y1, X2, Y2)
отрезок(0, 100, 50, 50);
отрезок(50, 50, 100, 100);
```

Альтернативный подход состоит в составлении программы для «черепашки», которая ползает по экрану и рисует линии.

Единого стандарта Языка Программирования Черепашек, по-видимому, не существует. Например, в Бейсике черепашка понимает следующие основные команды (на самом деле их больше, но нас это сейчас не интересует):

Команда	Описание
Mx,y	Передвинуть черепашку в позицию x, y на экране. Если перед числами, обозначающими координаты, стоят знаки + или — (например, «M-10,+5»), вместо абсолютных координат экрана будут использоваться координаты относительно предыдущего положения черепашки
Un	Передвинуть черепашку на n точек вверх
Dn	Передвинуть черепашку на n точек вниз
Ln	Передвинуть черепашку на n точек влево
Rn	Передвинуть черепашку на n точек вправо
TAn	Повернуть черепашку на n градусов против часовой стрелки. Так, если выполнить последовательность TA90U10, черепашка будет передвинута на десять точек влево, а не вверх
B	Не рисовать при выполнении следующей команды. Двигаясь, черепашка оставляет за собой след. Команда B временно отменяет рисование. Например, последовательность BU10R10 означает: сдвинуться (не рисуя) на десять точек вверх, затем сдвинуться (рисуя) на десять точек вправо

Используя команды черепашки, я могу нарисовать на Бейсике приведенный выше «домик» следующим образом:

```
DRAW "BM0,100R100D100L100U100M+50,-50M+50,+50"
```

Сейчас я, наверно, назвал бы такую последовательность команд «скриптом».

Вернемся, однако, к грамматикам.

9.3. Внутреннее устройство L-систем

9.3.1. Эволюция объектов

Как я уже упомянул выше, L-системы по внешнему виду напоминают обычные формальные грамматики, но работают они иначе.

Поскольку изначально L-системы были предназначены для моделирования растений, традиционно их изучение начинают с рисования каких-нибудь простых кустиков (пусть даже и весьма условных). Я тоже решил не нарушать этой традиции. Представьте себе, что растение состоит из элементов трех видов: участок ствола (I), левый лист (L) и правый лист (R). Тогда простейшую конфигурацию, состоящую из короткого ствола (см. рис. 9.1), на котором растут два листа, можно записать в виде строки ILR.

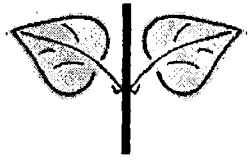


Рис. 9.1. *Куст ILR*

Обратите внимание: я не утверждаю, что существует компьютерная программа, рисующая приведенную картинку по заданной строке ILR. Я всего лишь говорю, что последовательность ILR можно трактовать как куст, у которого есть ствол и пара листьев.

Понятно, что задание сколько-нибудь сложной системы таким способом — дело весьма трудоемкое. Но природа тоже не создает полноценные организмы в одно мгновение: и дубы когда-то были желудями. Учитывая это, системы Линденмайера позволяют задавать правила развития конфигурации в виде

```
предок → потомок
```

Используя эту запись, я легко могу «вырастить» куст любой высоты. Добавим к описанию куста новый элемент — «корень» (A) и укажем способ его развития:

```
A → AILR
```

Изначально наш куст будет состоять всего лишь из одного корня. После трехкратного применения правила роста можно получить куст, состоящий из трех сегментов (см. рис. 9.2):

```
A → AILR → AILRILR → AILRILRILR
```

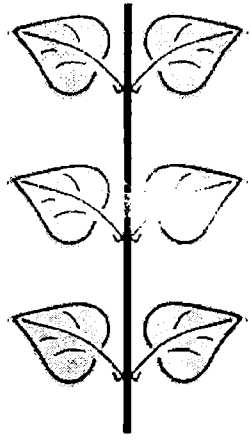


Рис. 9.2. *Куст AILRILRILR*

На рисунке никак не обозначен корень. Это не ошибка, поскольку мы не обязаны сопоставлять каждому символу вроде A какой-либо осмысленный графический элемент. Можно считать, что корень представляет собой всего одну точку белого цвета или даже вообще ничего не представляет с точки зрения графики.

9.3.2. Порождение и визуализация строк

С точки зрения логики я бы разделил функционирование L-систем на два независимых аспекта. С одной стороны, L-системы представляют собой механизм генерации строк на основе начальной строки (называемой *аксиомой*) и некоторой разновидности формальной грамматики. С другой стороны, полученная в итоге строка должна каким-то образом интерпретироваться подсистемой визуализации, чтобы на экране появился осмысленный рисунок.

Порождение строк

Механизм вывода строк ничего не знает о графическом выводе; на этом этапе мы всего лишь подставляем на место отдельных символов те или иные строки в соответствии с правилами грамматики. Здесь важно сделать одно замечание. В отличие от обычной грамматики, где на каждом шаге применялось лишь одно правило, в L-системах одновременно происходит замена всех символов, для которых существуют правила развития (хотя бы потому, что все части растения развиваются одновременно, а не по очереди).

Рассмотрим, например, эволюцию аксиомы g в процессе применения правила g → g[+g][-g]:

```
шаг 0: g
шаг 1: g[+g][-g]
шаг 2: g[+g][-g][+g[+g][-g]][-g[+g][-g]]
```

Таким образом, на каждом шаге мы заменяем любое вхождение символа g на строку g[+g][-g].

Этот процесс можно продолжать бесконечно. В реальности интересно бывает либо проследить эволюцию аксиомы на протяжении некоторого конечного числа шагов, либо попросту выполнить заранее известное количество подстановок и визуализировать полученную строку.

Визуализация строк

Мы еще вернемся к принципам работы механизма строкового вывода, а пока переключимся на модуль визуализации.

Здесь не существует единого стандарта. К сожалению, разные визуализаторы используют разные наборы команд. Есть, однако, представление об общих принципах работы подсистемы графического вывода, а также некоем «джентльменском наборе» команд.

Базовых принципов всего два:

- визуализатор функционирует по принципу черепашьей графики;
- все нераспознанные команды игнорируются.

Любая система визуализации должна предоставлять следующие команды (синтаксис может отличаться, но смысл остается неизменным):

Команда	Описание
g или F[64]	Нарисовать отрезок длины distance по направлению движения черепашки
+	Повернуть черепашку по часовой стрелке на угол angle
–	Повернуть черепашку против часовой стрелки на угол angle
[	Начать новую ветвь (в текущей точке создается новая черепашка, и управление передается ей)
]	Закончить ветвь (черепашка снимается с экрана, управление передается активной ранее черепашке)

Значения параметров `distance` и `angle` обычно задаются где-то в настройках визуализатора.

Посмотрим теперь, какой рисунок (см. рис. 9.3) соответствует уже знакомой нам системе, состоящей из аксиомы `g` и правила вывода g → g[+g][-g]. Изначально черепашка находится в середине экрана и смотрит вверх; distance = 15 (точек), angle = 20 (градусов).

На нулевом шаге все просто: команда `g` означает рисование единственного отрезка. Первый шаг уже интереснее. Самый левый символ `g`, как и раньше, рисует отрезок. Затем начинается ветвь: в текущей позиции создается новая черепашка, затем она поворачивается на `angle` градусов направо (символ «+») и рисует отрезок (команда `g`). После этого ветвь заканчивается, и управление передается исходной черепашке. Затем аналогичным образом рисуется левая ветвь кустика. На втором шаге сначала происходит рисование уже известного нам фрагмента g[+g][-g], после чего рисуется более сложная ветвистая структура [+g[+g][-g]][-g[+g][-g]].

[4] В одних реализациях — `g`, в других — `F`.

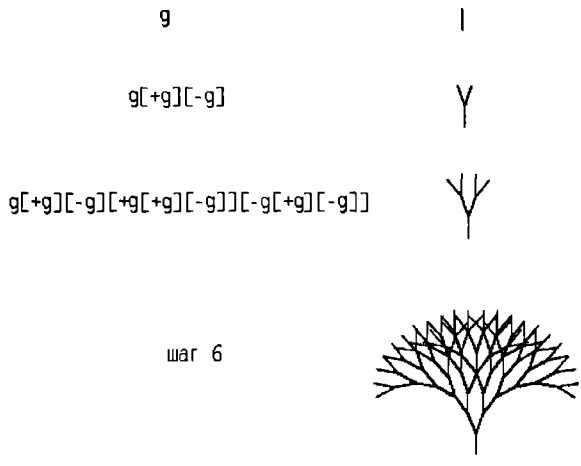


Рис. 9.3. *Визуализация простой L-системы*

Обратите внимание, что с алгоритмической точки зрения приведенная система неоптимальна, поскольку многие участки дерева перерисовываются многократно. Чтобы избавиться от этого недостатка, систему несложно доработать:

```
Аксиома: A
Правило вывода: A → g[+A][-A]
```

Смысл символа A можно сравнить с почкой дерева. Отныне развитие на последующих шагах происходит не в каждом фрагменте дерева (то есть не в символах g), а только в «почках»:

```
шаг 0: A
шаг 1: g[+A][-A]
шаг 2: g[+g[+A][-A]][-g[+A][-A]]
шаг 3: g[+g[+g[+A][-A]][-g[+A][-A]]][-g[+g[+A][-A]][-g[+A][-A]]]
```

9.4. Инструменты визуализации L-систем

Коль скоро мы добрались до практической визуализации L-систем, пора уделить внимание существующим программным инструментам. Чтобы не тратить время на обзор возможностей тех или иных утилит (возможно, это интересно читать, но на редкость скучно описывать), я воспользуюсь уже знакомым вам инструментом JFLAP. Должен сказать, что для

работы с L-системами существуют и куда более совершенные программы, поставляемые, к тому же, с множеством интереснейших примеров. На их фоне JFLAP выглядит достаточно бледно; тем не менее, как я уже сказал, полноценный обзор программ в наши планы сейчас не входит. Если у вас будет время и желание, исследуйте их самостоятельно:

- **LS Sketchbook**: http://coco.ccu.uniovi.es/malva/sketchbook/
- **LSystem**: http://severinghaus.org/projects/lsystem/
- **Fractint**: http://spanky.triumf.ca/www/fractint/fractint.html
- **L-studio**: http://algorithmicbotany.org/lstudio/index.html

Итак, для исследования L-системы прежде всего следует выбрать в главном меню программы JFLAP пункт `L-system` (что логично). На экране появится простой редактор, позволяющий ввести аксиому системы и любое количество правил вывода (см. рис. 9.4). Нижняя часть редактора предназначена для переопределения значений по умолчанию глобальных параметров вроде `distance` или `angle`.

Обратите внимание, что в правилах вывода все одиночные символы необходимо разделять пробелами, если вы хотите, чтобы система работала правильно. Несколько раздражает, но поделать с этим ничего нельзя.

Выбрав пункт **Input** → **Render System**, можно перейти в окно визуализации (см. рис. 9.5). Числовое поле справа предназначено для выбора отображаемого шага (так, на рисунке изображен шаг №3). Слева от него показана визуализируемая строка, если она только не слишком длинна. Координаты снизу позволяют вращать рисунок в трехмерном простран-

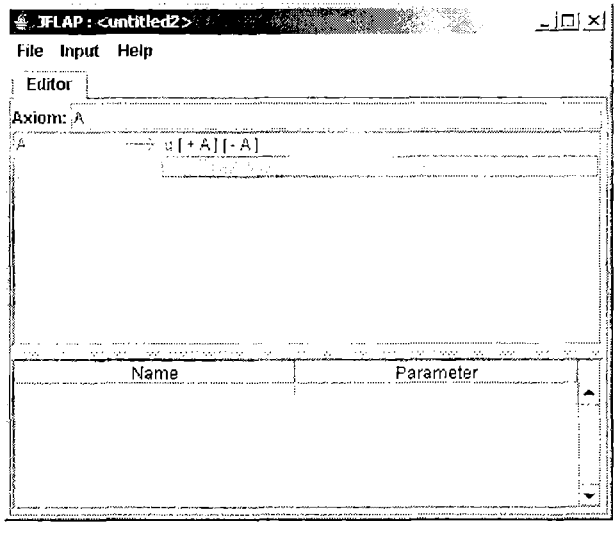


Рис. 9.4. *Редактор L-системы (JFLAP)*

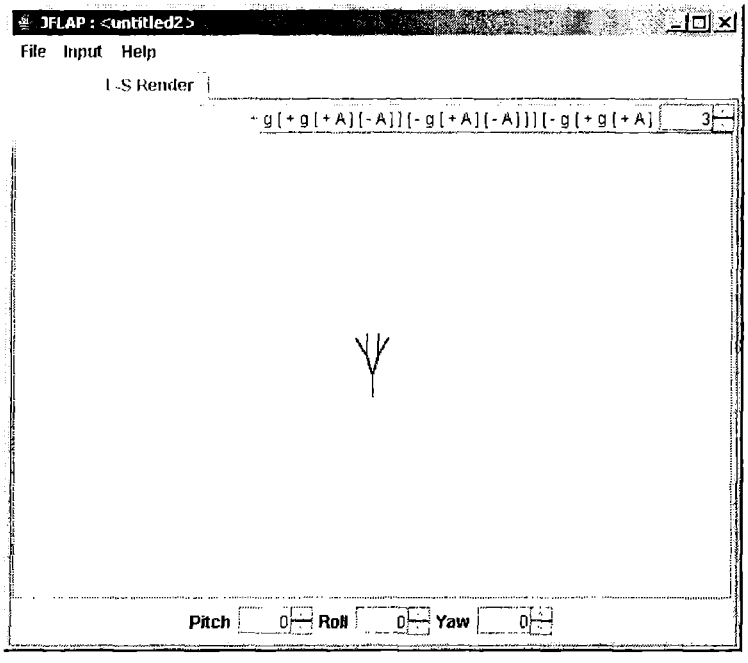


Рис. 9.5. *Окно визуализации L-системы (JFLAP)*

стве. Чтобы вернуться назад к редактору, необходимо выбрать пункт меню **File → Dismiss Tab**.

Вот и все.

На рис. 9.6 показаны два более сложных примера L-систем[65]. Надеюсь, они дадут представление о выразительной мощи этого инструмента.

9.5. Фрактальные узоры

L-системы очень хорошо подходят и для рисования так называемых фрактальных узоров. По определению Бенуа Мандельброта (человека, с чьим именем обычно связывают рождение фрактальной геометрии), фрактал — это структура, состоящая из частей, которые в каком-то смысле подобны целому.

Конечно, это определение достаточно размыто, но с его помощью можно уловить саму суть фракталов. Те кусты, что мы рисовали в предыдущем

[65] Автор — Адриан Мариано.

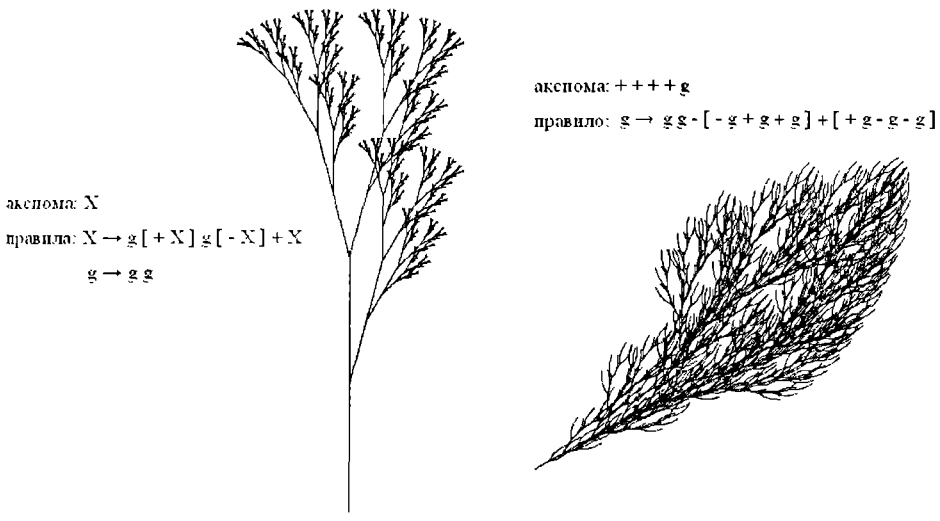


Рис. 9.6. *Примеры L-систем*

разделе, вполне подходят под определение фракталов. Действительно, каждая ветка «в каком-то смысле» подобна всему кусту.

Рисование природных объектов — одно из самых популярных применений фракталов на сегодняшний день. Фрактальные кусты, волны, текстуры заняли прочное место в компьютерной графике.

Здесь я хочу продемонстрировать визуализацию с помощью L-систем двух классических, широко известных фрактальных объектов — кривой Коха и треугольника Серпинского. Не менее известны, скажем, кривая Гильберта или драконова ломаная, но всего, конечно, не охватишь.

Кривая Коха состоит из некоторого числа отрезков одинаковой длины[66] (на нулевом шаге — из одного отрезка). На очередном шаге каждый такой отрезок заменяется изображенной на рис. 9.7 фигурой из четырех равных отрезков.

Записать определение кривой Коха с помощью грамматики несложно. Потребуется всего одно правило вывода:

```
g → g + g - - g + g
```

Отрезок прямой (g) заменяется последовательностью: отрезок, поворот направо (сегмент рисуется справа налево), второй отрезок, два поворота

[66] Истинной фрактальной кривой (а не ломаной) она становится лишь после бесконечного числа шагов.

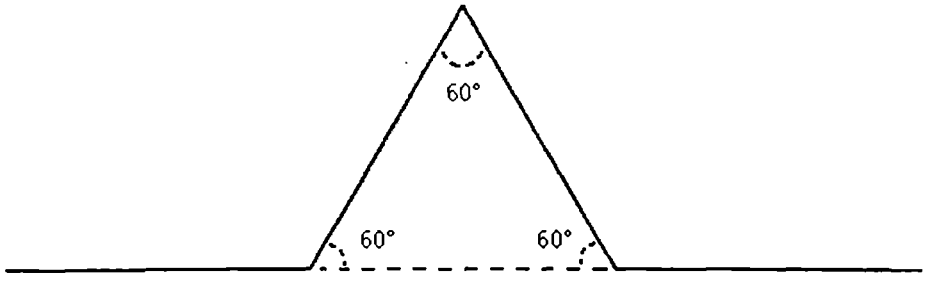


Рис. 9.7. *Сегмент кривой Коха*

налево, третий отрезок, поворот направо, четвертый отрезок. Чтобы при выводе получилась правильная ломаная, необходимо еще установить значение параметра angle равным 60[67].

Кривая Коха после третьего шага показана на рис. 9.8.

Треугольник Серпинского запрограммировать несколько сложнее. Эта интересная фигура получается следующим образом. Треугольник нуле-

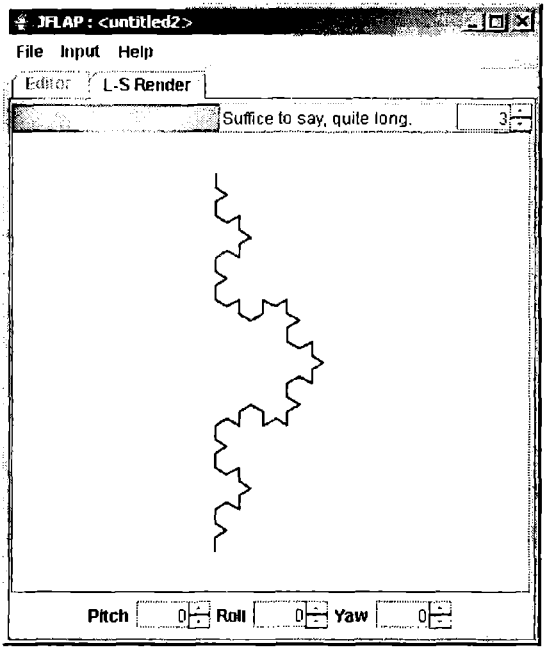


Рис. 9.8. *Кривая Коха третьего порядка*

[67] В поле Name редактора системы записывается строка angle, а в поле Parameter — число 60.

вого порядка представляет собой самый обыкновенный равносторонний треугольник. Для получения треугольника первого порядка треугольник нулевого порядка разделяется на четыре равные части (см. рис. 9.9).

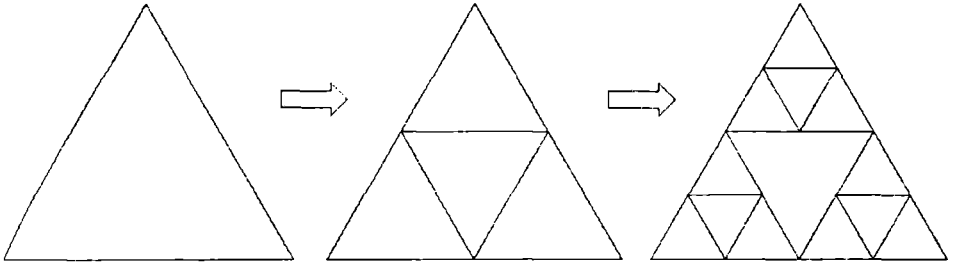


Рис. 9.9. *Построение треугольника Серпинского*

Центральная часть фигуры «извлекается». Иногда извлекаемую область раскрашивают в другой цвет, но это необязательно. Главное, что на следующих шагах разделение будет продолжено только для неизвлеченных участков. Разделяя оставшиеся после предыдущего шага треугольники, можно получить треугольник Серпинского очередного порядка.

Грамматика, соответствующая такому преобразованию, будет выглядеть довольно хитро.

Аксиома системы соответствует равностороннему треугольнику: g X g - - g g - - g g. Если считать, что угол каждого поворота равен шестидесяти градусам, мы действительно получим треугольник (см. рис. 9.10).

Символ X помечает позицию, из которой будут исходить разделяющие треугольник линии.

Чтобы получить фигуру следующего порядка, нам потребуется написать правило вывода, разделяющее этот треугольник на четыре части. Однако просто так разделить его нельзя. Дело в том, что внутри треугольника должен образоваться новый полноценный треугольник, в середине одной из сторон которого также записан символ X. Для этого нам попросту не хватает места: длина каждой линии разбивки должна составить 2*g, в то время как сейчас это значение равно всего лишь g. Исправить ситуацию просто: для этого придется на очередном шаге увеличить

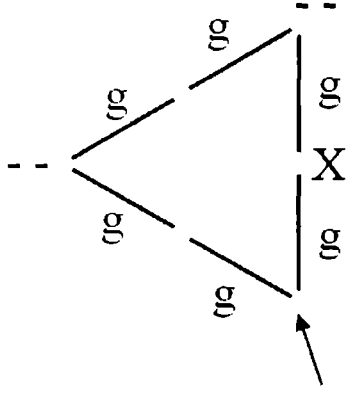


Рис. 9.10. *Аксиома для модели треугольника Серпинского*

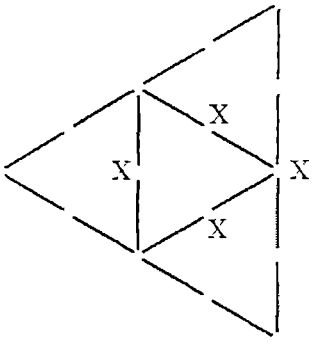


Рис. 9.11. *Вывод очередного шага*

каждую сторону треугольника вдвое. Такому действию соответствует правило

```
g → g g
```

Теперь нужно запрограммировать такое правило для X, чтобы из треугольника, изображенного на рис. 9.10, получалась фигура с рис. 9.11.

Здесь сложно сообразить, почему новая фигура получается из старой именно таким образом; имея же правило в графическом виде, записать вывод грамматики уже не так тяжело:

```
X → - - g X g + + g X g + + g X g - -
```

Вот и все. Готовый треугольник показан на рис. 9.12.

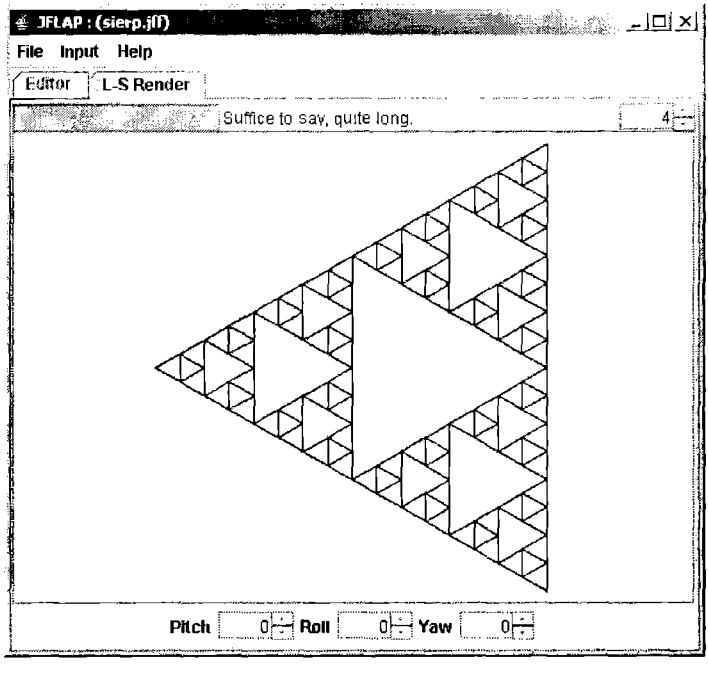


Рис. 9.12. *Треугольник Серпинского четвертого порядка*

9.6. Разновидности и дополнительные возможности L-систем

Конечно, мы пользовались только самыми простыми инструментами из тех, что обычно предоставляют L-системы. Чтобы дать более полное представление о мощи серьезного анализатора/визуализатора L-систем, я кратко опишу возможности, не потребовавшиеся нам на протяжении главы[68].

Стохастические L-системы

Рассматриваемые нами системы работали совершенно детерминированным образом. На основе заданной аксиомы генерировалась строка первого шага. На основе этой строки, в свою очередь, однозначным образом создавалась строка второго шага и так далее.

Стохастические L-системы позволяют внести некоторое разнообразие в этот процесс, разрешая использование конкурирующих правил вывода. Например, из почки (A) вырастает белый цветок (W). Это можно записать с помощью правила

```
A → W
```

А что делать, если из почки может вырасти как белый, так и красный (R) цветок? В таком случае на помощь приходит механизм конкурирующих правил:

```
(0.6) A → W
(0.4) A → R
```

Каждому из таких правил некоторым образом сопоставляется вероятность его использования. Так, встретив символ A, интерпретатор на следующем шаге заменит его либо на W (с вероятностью 0.6), либо на R с вероятностью 0.4).

Очень интересно бывает запустить несколько раз такой стохастический генератор, передавая ему одну и ту же аксиому. Рисунки, конечно, получатся разные, но их родство будет заметно невооруженным взглядом. Например, дважды моделируя развитие цветка розы, вы получите две разные розы, но никак не розу и тюльпан.

Контекстно-зависимые правила

Рассмотрим теперь другую ситуацию. Из почки (A) вырастает цветок (W), если рядом с почкой уже есть два цветка. Если же рядом с почкой

[68] Разумеется, далеко не все из них поддерживаются программой JFLAP.

располагаются два листа (L), то из почки также вырастает лист. Эти правила можно записать так:

```
2 WAW → W
2 LAL → L
```

Двойка указывает на порядковый номер символа, к которому относится правило (второй слева, то есть A). Символы справа и слева от A формируют контекст, в котором правило срабатывает.

Наличие подобных правил соответствует так называемой контекстно-зависимой грамматике, о которой я еще упомяну в дальнейшем.

Передача численных параметров

В приведенных выше примерах любой символ обозначал некоторый полностью определенный объект. Цветок либо есть, либо его нет; в то же время все цветки, закодированные каким-нибудь символом (скажем, W), выглядят и ведут себя совершенно одинаково. В то же время исследователь всегда найдет разумный набор параметров, который при желании можно приписать тому или иному объекту. Например, ветке может соответствовать толщина, цветку — возраст и размер, почке — некий запас «жизненной силы» и так далее. Значения параметров могли бы учитываться как в правилах развития, так и командах черепашки.

Помните, мы начинали изучение L-систем с выращивания кустика с помощью простого правила:

```
A → AILR
```

Последовательно применяя это правило, можно получить кустик сколь угодно большой высоты. Используя параметры, можно создать более реалистичную картину. Для начала перепишем правило несколько иначе:

```
A → ILRA
```

Теперь A — это не «корень», из которого растет куст, а «почка» на вершине уже развитого сегмента. Теперь добавим параметр, обозначающий «жизненную силу»:

```
аксиома: A(50)I(50)L(50)R(50)
правило вывода: A(x) → I(0.8*x)L(0.8*x)R(0.8*x)A(0.8*x)
```

Если на этапе рисования трактовать значение параметра как размер, то мы получим куст, в котором более высоко расположенные ветви и листья окажутся меньше по размеру, чем ветви и листья, расположенные ниже.

Отделение логики от графики

С описанием объектов при помощи L-систем связана одна сложность, на которую вы, возможно, уже обратили внимание: приходится одновременно думать как о смысловой нагрузке каждого фрагмента, так и о его графической интерпретации. Было бы приятно обозначить цветок одной буквой (например, W) и спокойно писать правила его развития; однако при попытке визуализации на экране ничего не появится, поскольку команда W не имеет смысла для черепашки.

Некоторые программы позволяют вводить определения вида

```
W: g+g+g+g
```

Перед визуализацией строки на место каждого определенного таким образом символа подставляется текст определения:

```
W-W → g+g+g+g-g+g+g+g
```

Лишь после этого строка будет передана подсистеме визуализации.

Трехмерная визуализация

Если черепашка распознает команды поворота влево и вправо, что нам мешает добавить инструкции для движения в трехмерном пространстве? Абсолютно ничего. Если представить, что черепашка ведет себя как самолет в воздухе, то командам + и — будут соответствовать повороты влево и вправо, а командам ^ и & (по соглашениям JFLAP) — кабрирование (штурвал на себя) и пикирование (штурвал от себя).

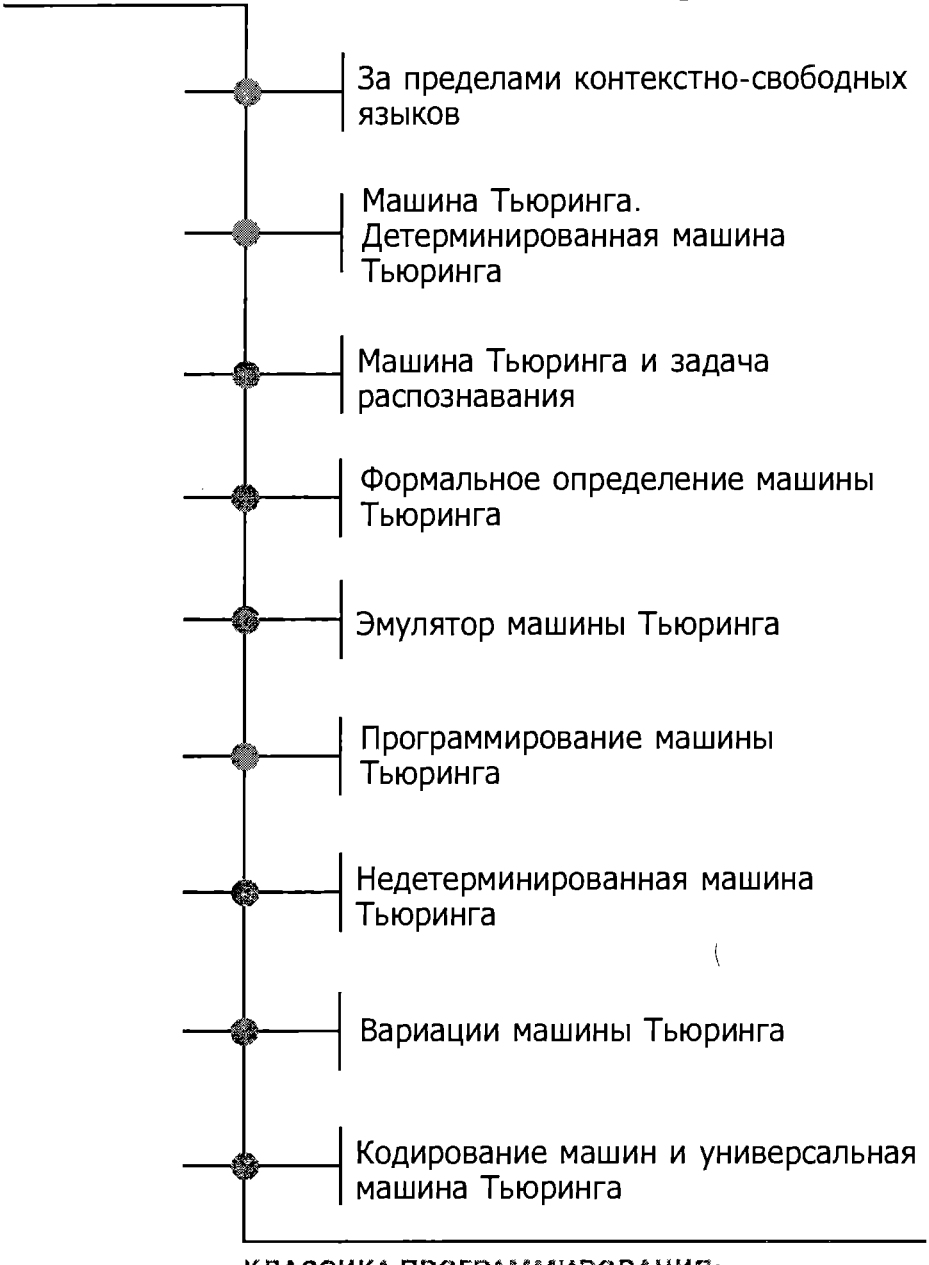
Дополнительные графические команды

Здесь фантазию авторов уже никто не может ограничить. Самое простое, что может прийти в голову — это повороты на произвольный угол и переходы на произвольные расстояния; смена рисующего цвета; переход без рисования; динамическая установка текущих параметров системы (цвет, ширина и длина рисующего отрезка, величины углов) на основе текущих (например, «увеличить текущий угол поворота на 10 градусов»); рисование многоугольников.

Итоги

- Системы Линденмайера представляют собой прекрасный пример использования формальных грамматик в задаче, не имеющей никакого отношения к синтаксическому анализу.
- В синтаксических анализаторах грамматики служат для определения принадлежности какой-либо строки интересующему нас языку. В L-системах процесс развернут на сто восемьдесят градусов: используя правила вывода грамматики, мы сами создаем корректные строки языка, взяв за основу некоторую «начальную строку», называемую аксиомой.
- Сгенерированные строки можно рассматривать как последовательности команд для подсистемы визуализации («черепашки»). С помощью черепашки невразумительный набор символов может превратиться в красивый кустик или во фрактальный рисунок.
- Сейчас существует достаточно программ, позволяющих создавать свои собственные L-системы. Каждая такая программа может предоставлять различные расширения как в логике систем, так и в командах визуализации.

Глава 10 Машины Тьюринга



10.1. Оглядываясь назад

Давайте в очередной раз окинем взглядом текущую ситуацию, чтобы не потерять нить рассуждений.

На протяжении всех глав мы рассматривали устройства, умеющие распознавать те или иные формальные языки. Первым таким устройством был конечный автомат. Мы подробно разбирали принцип работы конечного автомата, алгоритмы, с ним связанные, а также применение конечных автоматов на практике. Далее, мы убедились, что конечные автоматы умеют решать задачу распознавания некоторых формальных языков.

В частности, язык, строки которого состоят из произвольного количества букв `a` и `b` (то есть задаваемый регулярным выражением $(a \cup b)+$), может быть распознан конечным автоматом. В то же время другие языки (например, язык, состоящий из корректных арифметических выражений со скобками) конечному автомату уже не по силам. Таким образом, хотя любой формальный язык по определению представляет собой обычное (конечное или бесконечное) множество строк, различные языки в некотором смысле сильно отличаются друг от друга.

Конечные автоматы умеют распознавать лишь самые «простые» языки, называемые регулярными. Для нерегулярных языков приходится изобретать более мощные устройства. Так, детерминированные языки (включающие в себя все регулярные и некоторые нерегулярные языки) могут быть распознаны детерминированным автоматом с магазинной памятью.

Очень важно отметить, что устройства вроде конечного автомата или детерминированного автомата не являются какими-то далекими от жизни абстракциями. С некоторыми допущениями их вполне можно воплотить «в железе», то есть сконструировать машину, на практике решающую задачу распознавания какого-либо языка. Если случаи регулярных и детерминированных языков уже понятны, более общий класс контекстно-свободных языков пока не поддается непосредственному переводу в то или иное «железное» устройство.

Случай недетерминированного магазинного автомата в расчет не берем: сконструировать аппарат, имитирующий недетерминированное поведе-

ние, — задача не из легких. Конечно, для распознавания контекстно-свободных языков можно использовать обычный компьютер (алгоритм Кока-Янгера-Касами это наглядно демонстрирует), но устройство «обычный компьютер» пока что не вписывается в нашу иерархию аппаратов-распознавателей.

Задача распознавания языков переходит совершенно в иную плоскость, как только становится ясно, что любая задача принятия решения (то есть требующая вывода в формате «да/нет» в ответ на множество входных параметров) может быть сведена к распознаванию некоторого языка. В конце концов, так уж ли важно для нас умение распознавать те или иные языки? Вероятно, да, но не все с этим согласятся. А вот задачи принятия решения — совсем иной случай.

В нашей жизни обнаруживается огромное количество таких задач, начиная от чисто вычислительных — «отсортирован ли данный телефонный справочник по алфавиту?», «существует ли выигрышный алгоритм для игры в крестики-нолики?», и заканчивая нетривиальными вопросиками в стиле «стоит ли Джону жениться на Элизабет?» или «вредны ли гамбургеры для здоровья?»

Задачи принятия решения, конечно, представляют собой лишь узкий класс среди задач общего вида («отсортировать справочник по алфавиту», «найти подходящую невесту для Джона»); тем не менее, они встречаются во всех известных нам «классах сложности». Иначе говоря, представьте, что для решения некоторой задачи общего вида требуется устройство X. Так вот, всегда можно изобрести задачу принятия решения, неразрешимую устройствами слабее (в смысле вычислительной мощи), чем X.

Таким образом, разрабатывая устройства для распознавания языков, мы конструируем «машины-решатели» (пока не могу назвать их настоящими компьютерами). Кроме того, мы теперь не просто интуитивно понимаем, что задачи отличаются друг от друга по сложности, но и можем непосредственно оценить разницу между «машинами-решателями», по сути дела определяющую их вычислительную силу.

10.2. За пределами контекстно-свободных языков

Разумеется, контекстно-свободные языки не охватывают всех возможных случаев формальных языков, которые мы в состоянии вообразить. Существуют формальные языки, не описываемые контекстно-свободными грамматиками. Например, к таковым относятся языки *контекстно-зависимые*.

Напомню, правила контекстно-свободных грамматик имеют вид A → c, где A — переменная, а c — строка над алфавитом (Σ ∪ V). В соответствующей главе также объяснялось, что термин «контекстно-свободный» означает применимость правила независимо от контекста, в котором встретилась переменная A. В *случае контекстно-зависимых* (context-sensitive) *грамматик* правила приобретают несколько усложненный вид:[69]

$$\alpha A \beta \rightarrow \alpha c \beta$$

Как и раньше, здесь нетерминал A заменяется строкой c, состоящей из терминалов и нетерминалов, но теперь правило применяется лишь тогда, когда переменная A окружена строками (над алфавитом Σ ∪ V) α и β. Строки α и β (но не c) могут быть пустыми. Правило S → ε (где S — стартовый символ) допустимо, только если S не встречается в правых частях правил.

Мы уже рассматривали контекстно-свободный язык $\{a^n b^n\}$, состоящий из строк ε, ab, aabb, aaabbb и так далее. Добавив элемент c^n, мы получим контекстно-зависимый язык $\{a^n b^n c^n\}$ = {ε, abc, aabbcc, aaabbbccc, ...} Возможно, в это трудно поверить сразу, но язык $\{a^n b^n c^n\}$ нельзя задать контекстно-свободной грамматикой. В то же время мощности контекстно-зависимой грамматики уже достаточно для его описания:

```
S → P | ε
P → aPBc | abc
cB → Bc
bB → bb
```

В качестве примера можно привести дерево разбора строки `aabbcc` (см. рис. 10.1).

Но и контекстно-зависимые грамматики не решают задачу описания формальных языков полностью. Существуют еще более сложные языки, не поддающиеся описанию с помощью контекстно-зависимых грамматик.

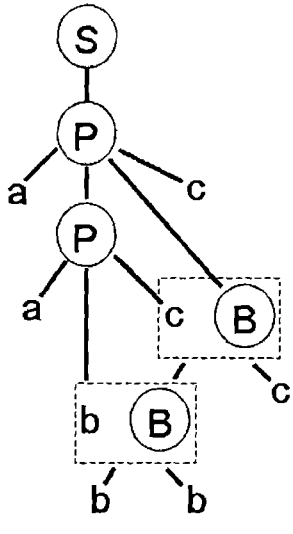


Рис. 10.1. *Дерево разбора строки aabbcc*

[69] Таким образом, контекстно-свободная грамматика является частным случаем контекстно-зависимой грамматики.

10.3. Машина Тьюринга. Детерминированная машина Тьюринга

Познакомимся теперь с новым устройством, которое сумеет умножить наши успехи в деле распознавания формальных языков и решения задач — с *машиной Тьюринга* (Turing machine). Начнем с машины как таковой, а уже затем рассмотрим подробнее ее историю, возможности и место в современной компьютерной науке.

Сразу оговорюсь: определения машины Тьюринга из разных книг могут иметь существенные отличия. Все получаемые устройства будут иметь одинаковую вычислительную мощь, но различаться тонкостями архитектуры и удобством конструирования.

По своему устройству машина Тьюринга похожа на конечный автомат. Сравните рис. 10.2 с моделью конечного автомата из второй главы.

Машина Тьюринга также в каждый момент времени находится в некотором состоянии, причем общее количество состояний конечно. Работу машина начинает в (как нетрудно догадаться) начальном состоянии, а заканчивает либо в допускающем (на рисунке обозначенном как Y), либо в отвергающем (N) состоянии. Таким образом, в отличие от конечного автомата, где любое состояние, не являющееся допускающим, считается недопускающим, в машине Тьюринга явным образом предусмотрено допускающее и отвергающее состояния.

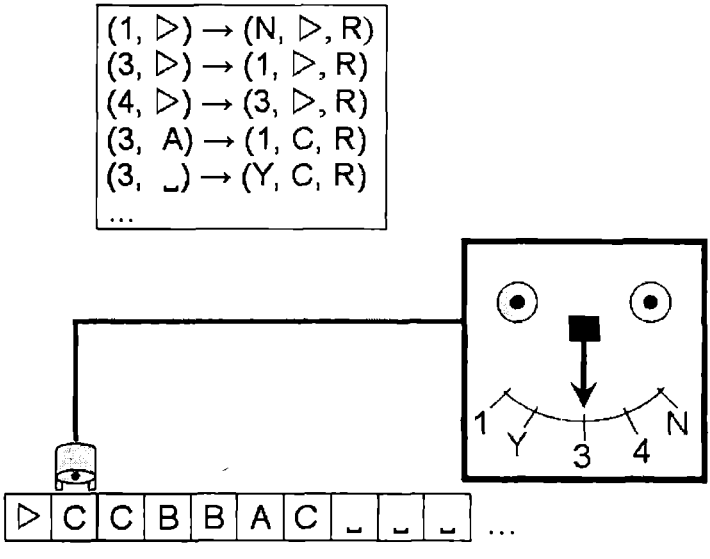


Рис. 10.2. *Машина Тьюринга (наглядное пособие)*

Другое отличие от конечного автомата состоит в том, что у машины Тьюринга есть считывающая/записывающая головка, способная перемещаться по входной ленте в любую сторону. Конечный автомат может анализировать символы входной ленты лишь последовательно, друг за другом. Машина Тьюринга не связана подобными ограничениями. Кроме того (и это очень важно), машина Тьюринга способна не только считывать информацию с входной ленты, но и производить запись на нее.

Входная лента ограничена слева (то есть у нее есть «начало»), имея при этом потенциально бесконечную длину. В каждый момент времени лента содержит конечное число символов, но головка машины Тьюринга может перемещаться вправо на любое требуемое количество ячеек. Первая (самая левая) ячейка ленты содержит специальный символ ▷, служащий признаком края ленты. Далее располагаются символы, формирующие входные данные (так же, как и в случае конечного автомата). Пустые ячейки заполняются пробельным символом ␣.

Правила переходов для машины Тьюринга (по смыслу аналогичные правилам для конечных автоматов) сопоставляют различным парам вида (`состояние`, `символ`) тройки вида (`состояние'`, `символ'`, `команда`). Как и раньше, пара (`состояние`, `символ`) описывает текущую ситуацию: состояние, в котором находится устройство плюс рассматриваемый в данный момент символ входной строки (в случае машины Тьюринга — символ, находящийся под считывающей головкой). В результате перехода машина оказывается в состоянии `состояние'`, а на ленту (в ячейку, находящуюся под головкой) записывается символ `символ'`. Кроме того, головка сдвигается в соответствии с компонентой `команда`. Различных команд всего три:

- **L**: сдвинуть головку на одну ячейку влево;
- **R**: сдвинуть головку на одну ячейку вправо;
- **H**: оставить текущее положение головки без изменений.

Поскольку конечные автоматы анализируют входную строку посимвольно — с первого и до последнего элемента по порядку — после ее считывания работа автомата завершается, и мы можем оценить результаты.

Машина Тьюринга способна «гулять» по строке сколько угодно, и по этой причине признаком завершения работы может служить лишь переход в допускающее либо в отвергающее состояние. Можно сконструировать машину Тьюринга, вообще никогда не заканчивающую работу. Обратите внимание на важную деталь: зацикливание машины всегда означает недопускание входной строки (как определить, что машина зациклилась — уже вопрос второй).

Пока что мы будем заниматься детерминированными машинами Тьюринга, не содержащими конфликтующих правил переходов.

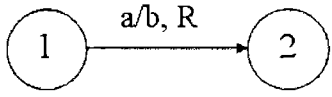


Рис. 10.3. *Графическое изображение переходов машины Тьюринга*

Так же, как и конечный автомат, машину Тьюринга нетрудно изобразить с помощью графа. Ребра графа помечаются исходным и заменяемым символами ленты, а также командой для головки (L, R или H). Сказанное иллюстрирует рис. 10.3, на котором изображен переход

```
(1, a) → (2, b, R)
```

10.4. Машина Тьюринга и задача распознавания

Рассмотрим теперь функционирование машины Тьюринга на примерах.

Распознавание регулярного языка

Начнем с самого простого случая — с распознавания регулярного языка a*b*. Поскольку такой язык может быть распознан даже обычным конечным автоматом, никаких особых изощренностей от машины Тьюринга не потребуется. Готовое устройство будет содержать всего три состояния (1, 2, 3) и пять переходов:

```
(1, a) → (1, a, R)
(1, ␣) → (3, ␣, H)
(1, b) → (2, b, R)
(2, b) → (2, b, R)
(2, ␣) → (3, ␣, H)
```

Стартовое состояние — первое, допускающее — третье. Так же, как и в случае конечных автоматов, здесь может встретиться «нештатная ситуация», для которой не предусмотрено перехода (например, считывание символа a с ленты, когда машина находится во втором состоянии). Здесь и далее в главе возникновение нештатной ситуации будет означать простое недопускание строки.

Сконструированная машина в виде графа показана на рис. 10.4.

Распознавание контекстно-свободного языка

Теперь попробуем построить машину Тьюринга, распознающую более сложный (контекстно-свободный) язык $\{a^n b^n\}$, состоящий из строк ε,

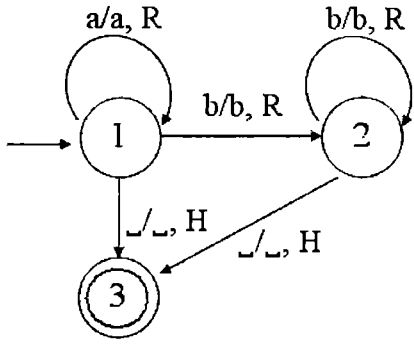


Рис. 10.4. *Машина Тьюринга, распознающая язык a*b**

ab, aabb, aaabbb и так далее. Как уже было показано, этот язык может быть распознан автоматом с магазинной памятью. У машины Тьюринга нет магазинной памяти, зато есть лента, на которую можно осуществлять запись.

Можно придумать несколько различных алгоритмов распознавания. Например, такой:

```
если лента машины Тьюринга пуста, допустить строку;

если первый непробельный символ ленты – a, а последний – b,
заменить оба этих символа на пробелы; в противном случае
отвергнуть строку;

перейти к первому шагу;
```

Приведенная процедура на каждом шаге удаляет крайние символы входной строки, если они равны a и b соответственно. Если в итоге получилась пустая строка, это значит, что исходная строка принадлежала языку $\{a^n b^n\}$. Если же в процессе работы на ленте оказалась строка, не имеющая по краям символов a и b, это означает, что ее следует отвергнуть. Перед тем, как перевести алгоритм в форму машины, я запишу его более подробно:

```
если текущий символ ленты – пробел, допустить строку (лента
пуста);

если текущий символ ленты – не a, отвергнуть строку;

заменить a на пробел;

двигаться вправо до первого пробельного символа ленты, затем
сдвинуться на ячейку влево;

если текущий символ ленты – не b, отвергнуть строку;
```

```
заменить b на пробел;

двигаться влево до первого пробельного символа ленты, затем
сдвинуться на ячейку вправо;

перейти к первому шагу;
```

Теперь можно непосредственно построить машину Тьюринга (см. рис. 10.5).

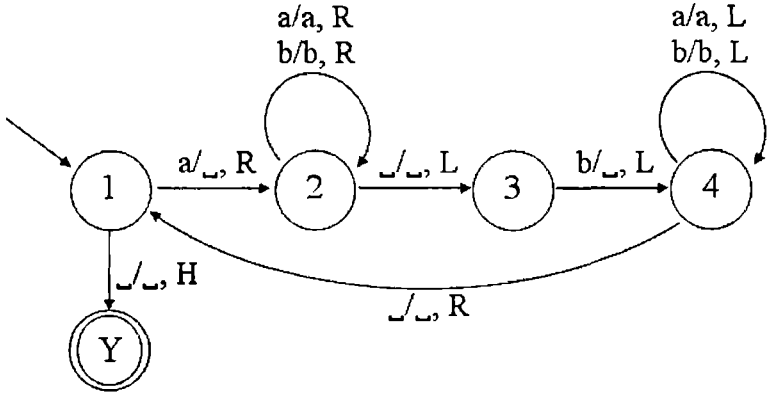


Рис. 10.5. *Машина Тьюринга для распознавания языка* $\{a^n b^n\}$

Распознавание контекстно-зависимых языков

Итак, машина Тьюринга обладает достаточной мощью для распознавания, по крайней мере, некоторых контекстно-свободных языков.

Следующий шаг — распознавание языка, не являющегося контекстно-свободным. Как уже упоминалось, примером такого языка может служить язык $\{a^n b^n c^n\}$, состоящий из строк ε, `abc`, `aabbcc`, `aaabbbccc` и так далее.

Я предлагаю использовать, быть может, не самый короткий, зато простой и понятный алгоритм. Сначала конец строки помечается специальным символом ◁ (иначе говоря, первый пробельный символ ленты заменяется на ◁). Затем выполняются операции:

```
если лента пуста (первый непробельный символ равен ◁),
допустить строку;

найти на ленте первый символ a; заменить его на пробел;

далее на ленте (начиная с текущей позиции) найти первый
символ b и заменить его на пробел;
```

```
на оставшейся части ленты найти первый символ c и также
заменить его на пробел;

перейти к первому шагу;
```

Если в процессе работы что-то происходит не так, как ожидается (например, очередной символ b не удается найти), входную строку следует отвергнуть.

Алгоритм последовательно заменяет найденные друг за другом символы a, b и c на пробелы. Если в определенный момент лента окажется пустой, это означает, что исходная строка принадлежит распознаваемому языку и должна быть допущена. Если одна из операций поиска завершилась неудачно либо на входе обнаружился непредвиденный символ[70], строка отвергается.

Рассмотрим теперь, как и в предыдущий раз, более подробную версию алгоритма:

```
пометить конец строки символом ◁ (то есть заменить первый
пробельный символ ленты на ◁);

ЦИКЛ

    если первый непробельный символ ленты равен ◁,
    допустить строку;

    если первый непробельный символ ленты не равен a,
    отвергнуть строку;

    заменить a на пробел;

    двигаться вправо, пока текущий символ равен пробелу или a;

    если текущий символ ленты не равен b, отвергнуть строку;

    заменить b на пробел;

    двигаться вправо, пока текущий символ равен пробелу или b;

    если текущий символ ленты не равен c, отвергнуть строку;

    заменить c на пробел;

    двигаться вправо, пока текущий символ равен пробелу или c;

    если текущий символ не равен ◁, отвергнуть строку;

    двигаться влево до начала ленты;

КОНЕЦ ЦИКЛА
```

[70] Пробел считается служебным символом. Будем полагать, что ни одна входная строка не содержит пробелов внутри.

Чтобы окончательно прояснить ход работы алгоритма, взгляните на состояние ленты машины во время последовательного выполнения его шагов в процессе анализа строки `aaabbbccc`:

```
▷aaabbbccc
▷aaabbbccc◁
▷␣aabbbccc◁
▷␣aa␣bbccc◁
▷␣aa␣bb␣cc◁
▷␣␣aVbb␣cc◁
▷␣␣a␣␣b␣cc◁
▷␣␣a␣␣b␣␣c◁
▷␣␣␣␣␣b␣␣c◁
▷␣␣␣␣␣␣␣␣c◁
▷␣␣␣␣␣␣␣␣␣◁
```

Готовая машина Тьюринга показана на рис. 10.6.

Важно отметить, что, как и любой конечный автомат, каждая машина Тьюринга предназначена для решения какой-либо одной задачи. Поэтому для распознавания различных языков придется изобретать различные машины Тьюринга (чем мы, собственно, уже занимались).

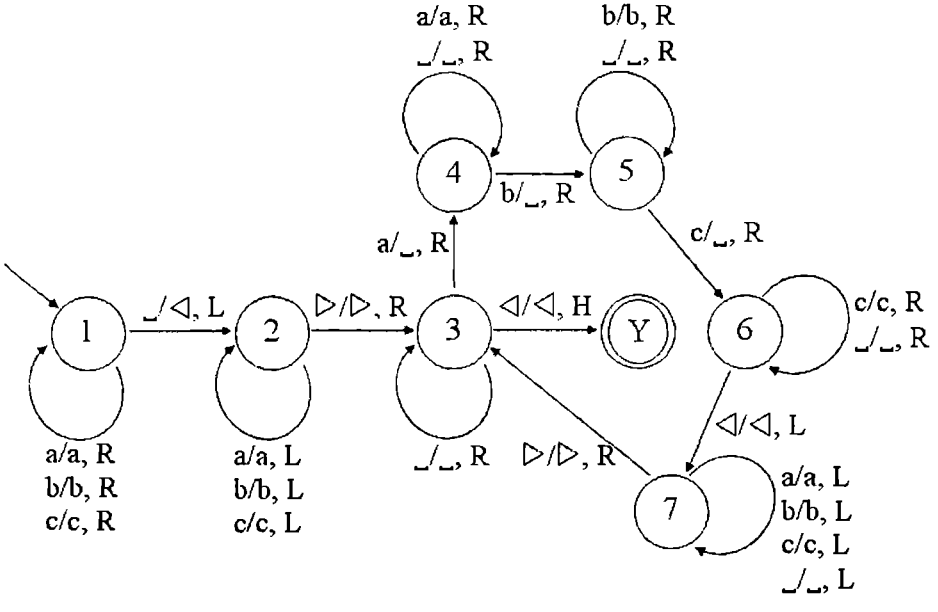


Рис. 10.6. *Машина Тьюринга для распознавания языка* $\{a^n b^n c^n\}$

10.5. Формальное определение машины Тьюринга

После подробного разбора примеров более строгое определение машины Тьюринга не должно вызывать особых затруднений (к тому же, оно похоже на определение конечного автомата).

Итак, в машине Тьюринга выделяют следующие элементы:

1. Конечное множество состояний $Q = \{q_1, q_2, \ldots, q_n\}$.
2. Конечное множество символов ленты $\Sigma = \{a_1, a_2, \ldots, a_m\}$.
3. Функция переходов $\delta(q_1, a) = (q_2, b, C)$ сопоставляющая паре вида (состояние, символ) тройку вида (состояние', символ', команда).
4. Начальное состояние q_0.
5. Допускающее состояние q_{yes} и отвергающее q_{no}.

В книгах нередко используют запись вида «$A = (Q, \Sigma, \delta, q_0, q_{yes}, q_{no})$». В таком обозначении нет ничего плохого, но вот когда пишут что-то вроде «машина Тьюринга — это кортеж из шести элементов», у меня это вызывает недоумение[71]. В конце концов, граф — это не просто набор черточек и точек; точно так же описание машины Тьюринга не ограничивается «кортежем элементов».

Безусловно, из формального описания мы сразу можем получить массу информации о состояниях и переходах, но кортеж не объясняет самого процесса работы машины Тьюринга. Даже имея на руках полную таблицу состояний и переходов, надо уметь ею воспользоваться правильно.

10.6. Эмулятор машины Тьюринга

Перед тем, как продолжить изучение возможностей машины Тьюринга, давайте создадим программу, имитирующую ее работу. Думаю, такая программа окажет большую помощь в наших дальнейших экспериментах.

Работать эмулятор будет следующим образом. Сначала с клавиатуры вводятся три числа — номера стартового, допускающего и отвергающего состояний[72]. Затем программа считывает начальное содержимое ленты и набор правил, каждое из которых состоит из пяти элементов, записанных через пробел:

```
состояние символ состояние' символ' команда
```

[71] На самом деле и куда более яркие чувства, ну да ладно.

[72] Предполагается, что состояния обозначаются целыми числами.

Так, запись

```
3 c 4 _ R
```

означает переход (3, c) → (4, ␣, R); пробельный символ ленты обозначается символом нижнего подчеркивания.

Для хранения правил в памяти полезно ввести две служебные структуры данных — левую и правую части:

```
struct LeftSide                          // левая часть
{
    public int state;                    // состояние
    public char symbol;                  // символ ленты

    public LeftSide(int st, char sym)
        {state = st; symbol = sym;}
}
struct RightSide                         // правая часть
{
    public int state;                    // состояние
    public char symbol, command;         // символ ленты и команда

    public RightSide(int st, char sym, char c)
        {state = st; symbol = sym; command = c;}
}
```

Работу машины Тьюринга можно проиллюстрировать с помощью простого псевдокода:

```
считать конфигурацию машины с клавиатуры;
State = q0;
TapeIdx = 1;
ПОКА State != qyes && State != qno
    если пользователь нажал пробел, прервать работу машины;
    найти правило r с левой частью (State, ЛЕНТА[TapeIdx]);
    пусть (состояние', символ', команда) – правая часть r;
    если правило не найдено, отвергнуть строку;
    State = состояние';
    ЛЕНТА[TapeIdx] = символ';
    изменить TapeIdx в соответствии с командой из правой части;
КОНЕЦ ЦИКЛА
если State == qyes, допустить строку, иначе отвергнуть;
```

Поскольку машина Тьюринга может работать бесконечно, следует предусмотреть возможность прерывания эмуляции по нажатию клавиши (например, пробела). Проще всего текущее состояние той или иной клавиши можно выяснить с помощью функции WinAPI `GetKeyState()`. Для ее использования, во-первых, требуется добавить еще одну using-директиву в начало программы:

```
using System.Runtime.InteropServices;
```

Во-вторых, в определение главного класса приложения придется добавить объявление метода `GetKeyState()`:

```
[DllImport("user32.dll", EntryPoint = "GetKeyState")]
public static extern short GetKeyState(int nVirtKey);
```

Еще одна тонкость связана с тем, что машина Тьюринга работает с бесконечной лентой. Разумеется, создать бесконечную строку в памяти нельзя, но можно отслеживать значение переменной `TapeIdx`, и, если головка окажется за пределами ленты, ленту нетрудно расширить (благо, головка машины на каждом шаге сдвигается не более чем на одну ячейку вправо).

Теперь можно привести полный текст основной процедуры эмулятора машины Тьюринга (см. листинг 10.1).

Листинг 10.1. Эмулятор машины Тьюринга

```
int Sstate = Convert.ToInt32(Console.In.ReadLine());
                    // стартовое состояние
int Ystate = Convert.ToInt32(Console.In.ReadLine());
                    // YES-состояние
int Nstate = Convert.ToInt32(Console.In.ReadLine());
                    // NO-состояние
string Tape = ">" + (string)Console.In.ReadLine() + "_";
                    // лента машины

Hashtable rules = new Hashtable();
string s;
while((s = Console.In.ReadLine()) != "")
                    // считывание правил
{
    string[] tr = s.Split(' ');
                    // state symbol state' symbol' command
    rules.Add(new LeftSide(Convert.ToInt32(tr[0]),
                           Convert.ToChar(tr[1])),
              new RightSide(Convert.ToInt32(tr[2]),
```

```
                            Convert.ToChar(tr[3]),
                            Convert.ToChar(tr[4])));
}

int State = Sstate;          // текущее состояние
int TapeIdx = 1;             // текущая позиция на ленте
const int VK_SPACE = 0x20;   // код клавиши пробел
try
{
    while(State != Ystate && State != Nstate)
                             // основной цикл
    {
        if(((int)GetKeyState(VK_SPACE) & 0x8000) != 0)
                             // если нажат пробел
            {Console.WriteLine("Работа прервана"); return;}

        // извлекаем правую часть правила по известной левой
        // если правило не найдено, генерируется исключение
        RightSide r = (RightSide)rules[new LeftSide(State,
                                        Tape[TapeIdx])];
        State = r.state;     // обновляем текущее состояние
                             // и ленту
        Tape = (Tape.Remove(TapeIdx, 1)).Insert(TapeIdx,
                r.symbol.ToString());

        if(r.command == 'L') // сдвиг головки машины
            TapeIdx--;
        else if(r.command == 'R')
            TapeIdx++;

        if(TapeIdx >= Tape.Length)
                             // если головка вышла за край ленты
            Tape += "_";     // расширить ленту
    }
    Console.WriteLine(State == Ystate ? "Строка допущена" :
                                        "Строка отвергнута");
}
catch(Exception)
{
    Console.WriteLine("Строка отвергнута");
}

Console.WriteLine("Лента: " + Tape);
```

Думаю, несколько комментариев к тексту эмулятора лишними не будут.

1. Как вы, вероятно, уже догадались, в качестве маркера края ленты используется символ >.
2. Функция `GetKeyState()` в качестве аргумента принимает так называемый виртуальный код клавиши, состояние которой требуется проверить. Код пробела равен 20 (шестнадцатеричному), поэтому для ясности я определяю константу:

```
const int VK_SPACE = 0x20;        // код клавиши пробел
```

3. Если функция `GetKeyState()` возвращает число с установленным старшим битом[73], это означает, что проверяемая клавиша в данный момент нажата. Поскольку возвращаемое значение занимает два байта (согласно спецификации функции `GetKeyState()`), мы должны проверить состояние пятнадцатого бита (младший бит — нулевой, старший — пятнадцатый). Если выполнить операцию побитового И для возвращаемого функцией значения и двоичного числа 1000 0000 0000 0000 (8000 шестнадцатеричное), мы получим нуль, если пятнадцатый бит результата `GetKeyState()` сброшен, и ненулевое значение в противном случае.

4. В C# нельзя так просто изменить произвольный символ строки. Быть может, на первый взгляд это кажется странным, но применение операции [] к строке позволяет лишь узнать, но не изменить значение, записанное в данной позиции:

```
string s = "abc";
char c = s[1];      // правильно: c = 'a'
s[1] = 'b';         // неправильно: ошибка компиляции
```

5. Так происходит потому, что в C# все строки являются неизменяемыми. Выполнение любой операции, приводящей к модификации строки, на самом деле приводит к генерированию новой строки. Проще всего добиться желаемого результата с помощью методов класса string можно в два действия. Предположим, требуется заменить символ строки `s`, расположенный в позиции `idx`, на символ `c`. Для этого сначала следует удалить символ `s[idx]` с помощью вызова `s.Remove()`, а затем вставить в позицию `idx` символ `c`, используя метод `s.Insert()`. На уровне кода эмулятора эта операция выглядит так:

```
Tape = (Tape.Remove(TapeIdx, 1)).Insert(TapeIdx,
                                    r.symbol.ToString());
```

[73] Да, работа с WinAPI всегда требовала определенного хакерского подхода.

В качестве хорошего примера входных данных можно передать эмулятору код машины Тьюринга с рис. 10.6:

```
1
8
0
aaabbbccc
1 a 1 a R
1 b 1 b R
1 c 1 c R
1 _ 2 < L
2 a 2 a L
2 b 2 b L
2 c 2 c L
2 > 3 > R
3 _ 3 _ R
3 < 8 < H
3 a 4 _ R
4 a 4 a R
4 _ 4 _ R
4 b 5 _ R
5 b 5 b R
5 _ 5 _ R
5 c 6 _ R
6 c 6 c R
6 _ 6 _ R
6 < 7 < L
7 a 7 a L
7 b 7 b L
7 c 7 c L
7 _ 7 _ L
7 > 3 > R
```

Состояние 1 является стартовым, состояние 8 — допускающим. Поскольку явного отвергающего состояния в этой машине не предусмотрено, я выбрал для него номер 0, нигде далее в определении машины не использующийся. Для входной строки `aaabbbccc` программа выдает вполне ожидаемый результат:

```
Строка допущена
Лента: >_________<
```

Если же на вход передать, например, строку `aabbbccc`, машина ее отвергнет:

```
Строка отвергнута
Лента: >___b__c<
```

10.7. Программирование машины Тьюринга

По сравнению с конечными автоматами машина Тьюринга имеет одну совершенно новую возможность. Помимо перехода в допускающее/отвергающее состояние машина Тьюринга изменяет содержимое своей ленты. В дальнейшем мы можем использовать это содержимое в качестве «выходных данных», что существенно расширяет сферу применения машины Тьюринга, выводя ее за пределы задач принятия решений[74]. Например, можно создать машину Тьюринга, выполняющую сложение двух чисел. Входная лента будет содержать два числа, разделенные пробелами. Машина в процессе работы вычислит их сумму, запишет результат на ленту и перейдет в допускающее состояние.

Давайте рассмотрим несколько машин Тьюринга, умеющих решать подобные задачи.

Определение разности целых чисел

Начнем, пожалуй, с вычитания. Пусть лента машины содержит два целых числа. После перехода машины в допускающее состояние на ленте должна быть записана разность исходных чисел.

Для простоты будем считать, что числа записаны в унарном коде[75], то есть любое значение представлено с помощью равного ему количества символов (например, точек). Далее, ограничимся случаем, когда получаемая разность неотрицательна. Наконец, необходимо выбрать конкретный формат записи входных данных. Например, такой:

```
уменьшаемое-вычитаемое=
```

(то есть разделителями чисел служат символы «минус» и «равно»). Таким образом, выражение «5 — 2» будет записано на ленте как

```
.....-..=
```

Алгоритм вычитания достаточно прост:

```
ЦИКЛ
    найти на ленте символ «равно»;
    если ему предшествует символ «минус»,
    стереть служебные символы и допустить строку;
```

[74] Справедливости ради отмечу, что память магазинного автомата тоже могла бы использоваться аналогичным образом, но я решил не развивать эту тему.

[75] Хорошее, наукообразное слово для обозначения простой штуки, известной любому со времен счетных палочек, не правда ли?

```
    сдвинуть символ «равно» на позицию влево;
    заменить последний символ уменьшаемого на пробел;
КОНЕЦ ЦИКЛА
```

В процессе вычисления выражения «5 — 2» лента меняется так:

```
.....-..=
.... -.=
...  -=
...
```

Готовая машина Тьюринга изображена на рис. 10.7.

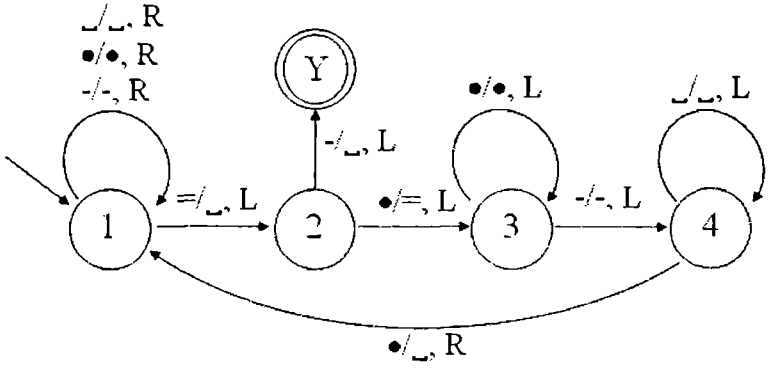


Рис. 10.7. *Машина Тьюринга, находящая разность двух чисел*

Осталось лишь привести текстовое представление машины, пригодное для выполнения эмулятором, и выводимый результат:

```
1
5
0
.....-..=
1 _ 1 _ R
1 . 1 . R
1 - 1 - R
1 = 2 _ L
2 - 5 _ L
2 . 3 = L
3 . 3 . L
3 - 4 - L
```

```
4 _ 4 _ L
4 . 1 _ R

Строка допущена
Лента: >...________
```

Дублирование входной строки

Следующий пример — дублирование строки, содержащейся на входной ленте (если входная строка равна `aabba`, то на выходе ожидается `aabbaaabba`). Для простоты будем считать, что входной алфавит состоит лишь из символов `a` и `b`.

Не претендуя на оптимальность, предлагаю такой алгоритм удваивания строки:

```
ПОКА входная строка содержит хотя бы один символ
в нижнем регистре

    найти первый символ строки, записанный в нижнем регистре;

    ЕСЛИ этот символ  —  a

        заменить его на A и дописать A в конец строки;

    ИНАЧЕ

        заменить его на B и дописать B в конец строки;

КОНЕЦ ЦИКЛА

заменить все заглавные буквы на строчные;
```

Соответствующая машина Тьюринга показана на рис. 10.8.

Код для эмулятора с примером данных ленты приведен ниже.

```
1
6
0
aabba
1 a 3 A R
1 b 2 B R
1 A 1 A R
1 B 1 B R
1 _ 5 _ L
2 a 2 a R
2 b 2 b R
2 A 2 A R
2 B 2 B R
2 _ 4 B L
```

```
3 a 3 a R
3 b 3 b R
3 A 3 A R
3 B 3 B R
3 _ 4 A L
4 A 4 A L
4 B 4 B L
4 a 4 a L
4 b 4 b L
4 > 1 > R
5 A 5 a L
5 B 5 b L
5 > 6 > H

Строка допущена
Лента: >aabbaaabba_
```

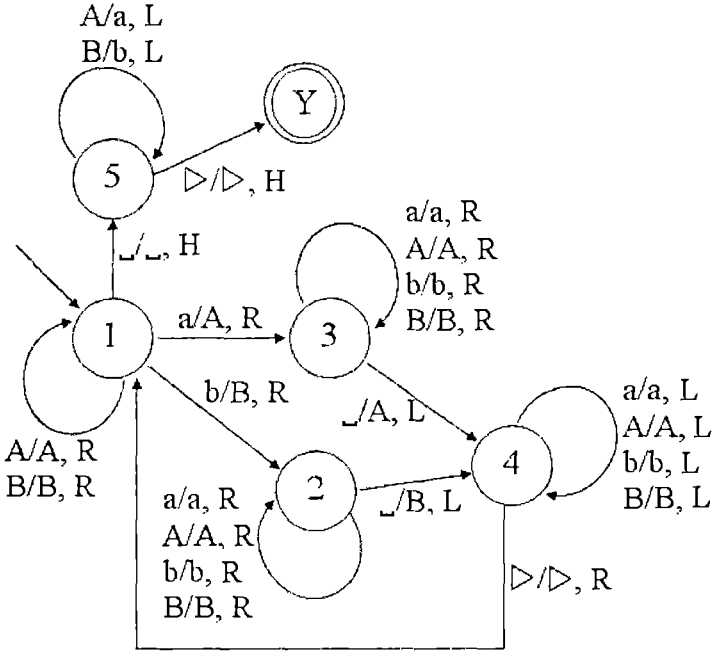


Рис. 10.8. *Машина Тьюринга для дублирования входной строки*

Сортировка данных

Последний пример демонстрирует реализацию простого алгоритма сортировки данных ленты. Пусть лента, как и в прошлый раз, содержит строку, состоящую из символов `a` и `b`. Требуется перенести все элементы `a` в начало строки, а элементы `b` — в ее конец (так, `babbbab` превращается в `aabbbbb`).

Алгоритм для такой специализированной сортировки выглядит так:

```
ПОКА строка не отсортирована (в строке есть подстрока ba)
    найти в строке первое вхождение символа b;
    заменить его на a;
    найти в строке последнее вхождение символа a;
    заменить его на b;
КОНЕЦ ЦИКЛА
```

Машина (см. рис. 10.9) работает в точном соответствии с алгоритмом. Первые два состояния ответственны за проверку упорядоченности элементов. Если строка оказывается еще не отсортированной, состояния 3-7 меняют местами первую `b` и последнюю `a` строки.

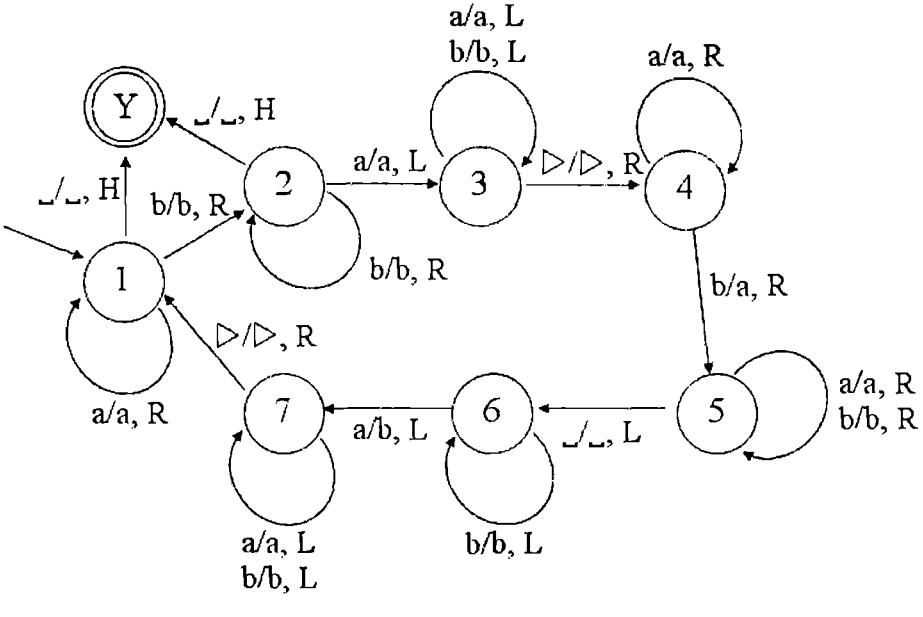


Рис. 10.9. *Машина Тьюринга, выполняющая сортировку*

Как обычно, описание машины заканчивает входной набор данных для эмулятора и результат его работы:

```
1
8
0
babbbab
1 b 2 b R
1 a 1 a R
1 _ 8 _ H
2 b 2 b R
2 _ 8 _ H
2 a 3 a L
3 a 3 a L
3 b 3 b L
3 > 4 > R
4 a 4 a R
4 b 5 a R
5 a 5 a R
5 b 5 b R
5 _ 6 _ L
6 b 6 b L
6 a 7 b L
7 a 7 a L
7 b 7 b L
7 > 1 > R

Строка допущена
Лента: >aabbbbb_
```

10.8. Недетерминированная машина Тьюринга

Как и предыдущие устройства, описанные в книге, машина Тьюринга может быть недетерминированной. Это означает, что в определении машины могут встречаться правила с одинаковыми левыми, но разными правыми частями:

```
(1, b) → (2, b, R)
(1, b) → (3, c, H)
```

Мы уже обсуждали принципы работы недетерминированных устройств, поэтому сейчас не буду останавливаться на них подробно. Любой недетерминизм, любая возможность выбора как бы расщепляет мир на

несколько параллельных, в каждом из которых принимается одно из допустимых решений.

Мы прослеживаем действия машины в каждом из этих миров (либо в некоторых, особо нас интересующих). Если хотя бы в одном из миров машина Тьюринга допускает входную строку (содержимое ленты в расчет не принимается), итоговым ответом будет «допустить», в противном случае — «отвергнуть».

Если запастись терпением, можно попробовать написать эмулятор недетерминированной машины Тьюринга. Возможная идеология подобных программ (хотя и без подробностей) уже предлагалась. Эмулятор может выполнять перебор с возвратом. Движемся до первой «развилки», далее выбираем любой возможный путь (запоминая его). Если машина завершила работу, возвращаемся на ближайшую развилку и выбираем другое направление.

Поскольку эта книга имеет скорее практическую направленность, я хотел бы сделать акцент не на тонкостях определения или конструирования недетерминированных машин Тьюринга, а на их применении в компьютерной науке.

Возможно, основная ценность недетерминированных машин Тьюринга вытекает из того факта, что детерминированная и недетерминированная машины имеют одинаковую вычислительную мощь[76]. Пока мы не будем разбираться, какие именно языки способна распознавать машина Тьюринга; сейчас важно другое: множество языков, распознаваемых детерминированной машиной Тьюринга, полностью совпадает с множеством языков, распознаваемых недетерминированной машиной Тьюринга[77].

В программистской практике построение любой (неважно, детерминированной или недетерминированной) машины Тьюринга редко бывает полезным. Зато если требуется определить, способна ли машина Тьюринга решить ту или иную задачу[78], обычно бывает гораздо проще построить недетерминированную машину, а затем просто воспользоваться в рассуждениях эквивалентностью вычислительной мощи обеих версий машины.

Пример: может ли машина Тьюринга определить, является ли введенное (записанное на ленте) число составным (то есть делящимся без остатка на какое-либо число, отличное от себя и от единицы)?

---

[76] Речь сейчас идет о самом факте распознавания языка (распознает — не распознает), но не о времени, затрачиваемом на решение одной и той же задачи.

[77] Этот факт доказывается в учебниках по теории вычислений.

[78] В дальнейшем станет понятно, почему этот вопрос может кого-либо интересовать.

Конечно, можно попытаться реализовать обычный алгоритм проверки последовательным делением на целые числа (думаю, многие из вас не раз делали это, изучая программирование) с помощью детерминированной машины Тьюринга, но подобное задание явно не из легких. Есть и более простое решение.

Для начала давайте договоримся, что все числа на ленте записываются в унарном коде (как в примере с вычитанием). Далее, поверьте мне на слово (просто не хочется терять на это время; если есть желание, проверьте сами), что не так сложно написать фрагмент машины Тьюринга, находящий произведение двух чисел.

Теперь создадим простой модуль недетерминированной машины, генерирующий некоторое целое число (см. рис. 10.10).

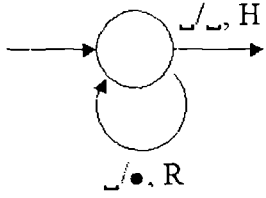


Рис. 10.10. *Генератор целых чисел*

В каждом из миров будет записано различное количество точек, прежде чем машина перейдет в следующее состояние.

Теперь требуется всего лишь создать машину, действующую по алгоритму:

```
сгенерировать на ленте два числа;
перемножить их и записать результат;
если результат совпадает с проверяемым числом, перейти в
допускающее состояние;
```

Вероятно, подобная идея проверки числа на делимость на первый взгляд может показаться совершенно безумной; тем не менее, теоретически все верно. Если проверяемое число является составным, в одном из миров недетерминированная машина сумеет сгенерировать два числа, произведение которых окажется равным проверяемому (после чего мы немедленно получим утвердительный ответ).

Если же проверяемое число является простым, то машина Тьюринга будет бесконечно продолжать перебирать варианты в поисках его несуществующего разложения на множители. Немного усложнив машину, можно ограничить величины генерируемых ею чисел. Тем самым удастся за конечное время найти разложение числа на множители или определить, что его не существует.

Поскольку недетерминированное поведение не увеличивает вычислительной мощи машины Тьюринга, можно сделать вывод о том, что рассматриваемая задача может быть решена с помощью обычной, детерминированной машины.

10.9. Вариации машины Тьюринга

В книгах очень часто рассматривают некоторые расширения машины Тьюринга, во многих случаях упрощающие процесс построения машины для решения тех или иных конкретных задач. Однако необходимо отметить, что ни одно из этих расширений не увеличивает вычислительной мощи машины Тьюринга, то есть не расширяет множество распознаваемых языков. Я решил вкратце описать наиболее популярные расширения, чтобы встреча с любым из них не была для вас сюрпризом.

Бесконечная лента

Наиболее простое расширение состоит в том, чтобы продолжить ленту не только вправо, но и влево (см. рис. 10.11). Таким образом, движение головки машины влево больше не может привести к нештатной ситуации. Теперь нельзя сказать, что входные данные расположены во второй или третьей ячейке ленты (поскольку лента бесконечна в обе стороны); можно, однако, указать, что строка начинается прямо под головкой машины или, скажем, на одну ячейку левее.

Многомерная лента

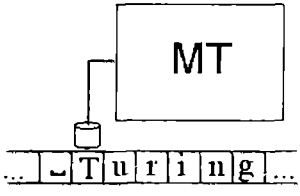


Рис. 10.11. *Машина Тьюринга с бесконечной (в обе стороны) лентой*

Каждая ячейка ленты стандартной машины Тьюринга может содержать лишь один символ. Почему бы не сделать ячейки многомерными (см. рис. 10.12)? На каждом переходе машина считывает множество символов, содержащихся в текущей ячейке и, соответственно, записывает на их место множество других символов:

```
(состояние, символы) → (состояние', символы', команда)
```

Для k-мерной ленты любой переход можно записать в виде

(состояние, <s_1, s_2, ..., s_k>) → (состояние', <s_1', s_2', ..., s_k'>, команда)

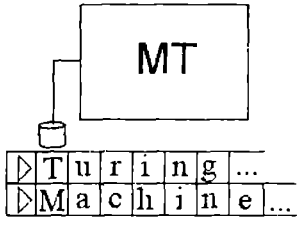


Рис. 10.12. *Машина Тьюринга с двумерной лентой*

Несколько головок

Следующее расширение — добавить в машину Тьюринга несколько дополнительных головок (см. рис. 10.13). Изначально все головки расположены над одной клеткой ленты. Каждый переход теперь сопоставляет паре (состояние, символ) четверку (состояние', номер_головки, символ', команда). Новый элемент номер_головки указывает, с какой головкой производится операция. Таким образом, каждый переход имеет дело лишь с одной головкой, но не с несколькими одновременно.

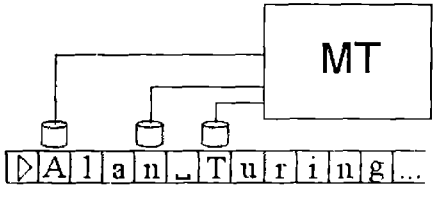


Рис. 10.13. *Машина Тьюринга с тремя головками*

Несколько лент

Последнее из расширений, которое мы обсудим, связано с добавлением дополнительных лент (см. рис. 10.14). Если машина может иметь несколько головок, почему бы не выделить каждой головке отдельную бесконечную ленту? На сей раз действия со всеми головками производятся одновременно, то есть любой переход затрагивает каждую головку и каждую ленту:

(состояние, символы) → (состояние', символы', команды)

Для k головок/лент во время любого перехода считывается/записывается k символов, после чего каждая головка сдвигается в соответствии с собственной командой:

(состояние, <s_1, s_2, …, s_k>) →
(состояние', <s_1', s_2', …, s_k'>, <c_1, c_2, …, c_k>)

Конечно, любые расширения можно комбинировать между собой, получая еще более изощренные вариации машины Тьюринга.

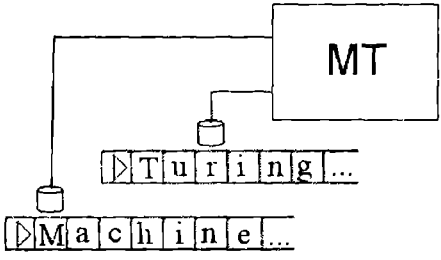


Рис. 10.14. *Машина Тьюринга с двумя лентами*

Имитация машины Тьюринга системой JFLAP

Несомненно, авторы JFLAP не могли обойти стороной столь важную модель, каковой является машина Тьюринга. Система JFLAP поддерживает как стандартную машину с одной лентой, так и многоленточные модели. Поскольку в книге приведены осмысленные примеры лишь для машин с одной лентой, я не буду останавливаться на работе в JFLAP с другими разновидностями.

Приступить к созданию новой машины Тьюринга можно, выбрав в главном меню JFLAP пункт **Turing Machine**. Процесс редактирования происходит абсолютно таким же образом, как и в случае конечных/магазинных автоматов. Переходы (вполне ожидаемо) помечаются тремя символами: считываемый символ, записываемый символ и команда перехода.

Отличий у той вариации машины Тьюринга, которая используется в JFLAP, от модели, выбранной в книге, не так много:

- лента любой машины в JFLAP бесконечна в обе стороны;
- в качестве команды головки «оставаться на месте» используется символ S;
- машина не имеет специальных yes- и no-состояний: машина завершает работу в допускающем состоянии, а результаты (строка принята/отвергнута) предполагается хранить на ленте;
- ничего не содержащая ячейка входной ленты обозначается пустым квадратом.

ачестве примера можно привести вычитающую машину, изображен-
) на рис. 10.7 (JFLAP-версия показана на рис. 10.15).

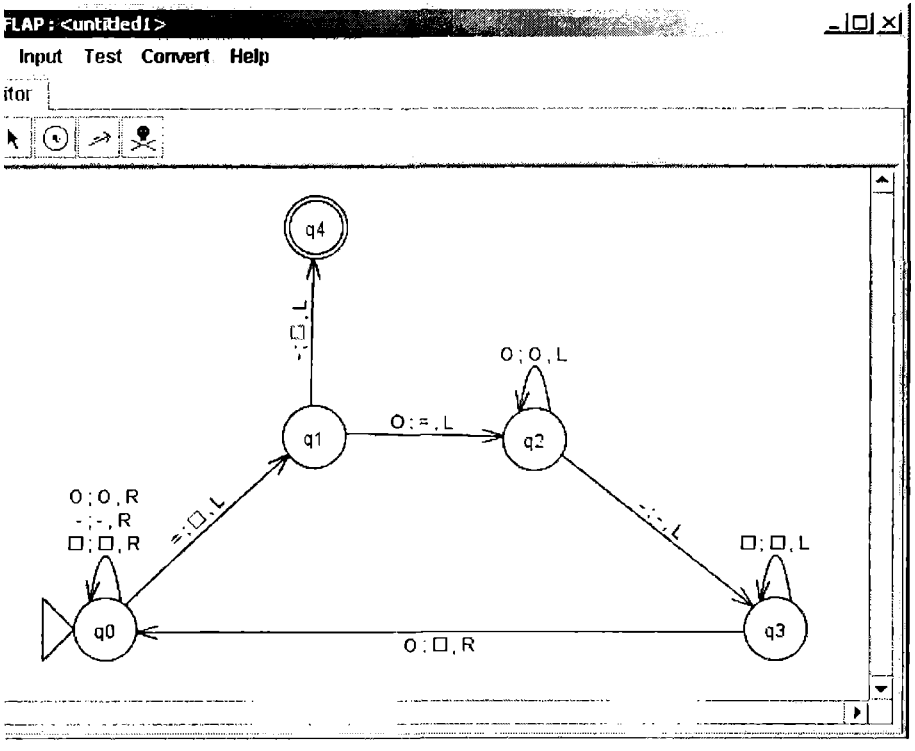


Рис. 10.15. *Машина Тьюринга, находящая разность двух чисел (JFLAP)*

кольку использовавшаяся ранее точка на экране довольно плохо за-
на, я заменил ее заглавной буквой O.

ультат работы машины (**Input** → **Fast Run**) для входной строки
OO-OO= показан на рис. 10.16.

10.10. Кодирование машин и универсальная машина Тьюринга

ротяжении всей книги (и эта глава не исключение) рассматриваемые
ойства не имели никакого отношения к анализируемым строкам дан-
. Действительно, устройство — это некоторый аппарат, характеризу-
йся состояниями, переходами, той или иной организацией памяти.
ходные данные — это просто строка символов, такая как aaabbca

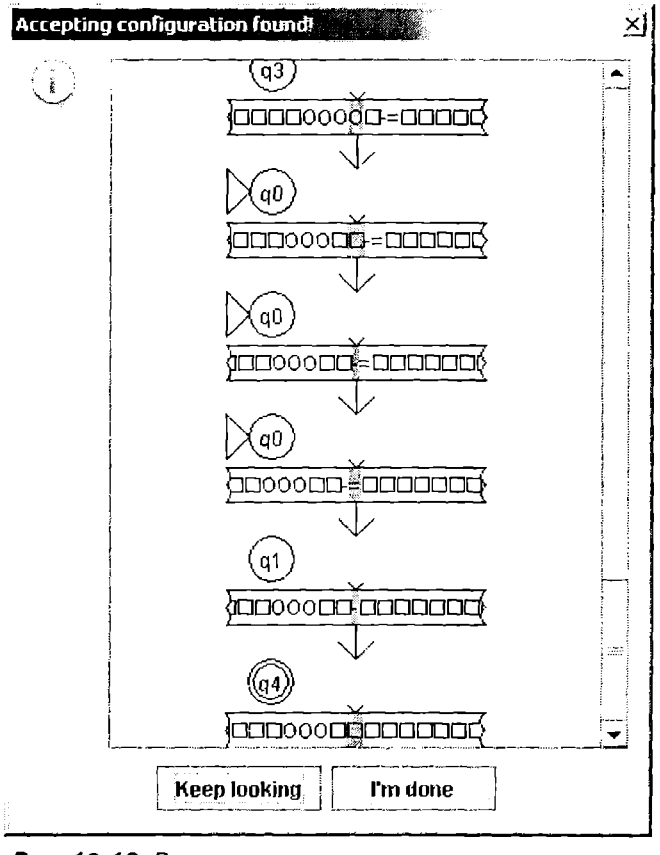


Рис. 10.16. *Результат выполнения процедуры вычитания*

или `zeus@olympus.gov`. Но что будет, если машину саму представить в виде строки символов?

На самом деле мы уже выполняли подобное кодирование, просто я не акцентировал внимание на том, что определение машины незаметно превращается в последовательность букв и цифр. В конце концов, чем является определение машины Тьюринга вроде

```
1
5
0
.....-..=
1 _ 1 _ R
1 . 1 . R
1 - 1 - R
1 = 2 _ L
```

```
2 - 5 _ L
2 . 3 = L
3 . 3 . L
  - 4 - L
4 _ 4 _ L
4 . 1 _ R
```

ія нашего эмулятора? Пусть вас не смущает, что описание машины за-имает несколько строк. Символ возврата каретки / перевода строки ни-ть не лучше и не хуже любого другого символа. Тема перевода машины ьюринга (вместе со входными данными) в строковое представление не жется мне особо сложной. Думаю, многие из вас сумеют без проблем идумать собственные схемы хранения, не уступающие идеям авторов ассических учебников.

книгах чаще всего описывают своего рода «окончательный» алгоритм, особный закодировать любую машину Тьюринга в строку, состоящую лько из нулей и единиц. С другой стороны, мне хочется предупредить бого интересующегося изобретением своей собственной схемы хране-ія: удовлетворительное теоретическое решение этой задачи несколько ожнее той процедуры, что была реализована в программе эмулятора.

реальном эмуляторе мы всегда делаем какие-либо «разумные» предпо-жения. Например, что номер состояния не превосходит максимального ачения, которое может быть помещено в переменную типа int, или что ементы входной ленты целиком состоят из символов ASCII-набора. В ории вы не должны делать никаких предположений о количестве состо-ий кодируемой машины, ее таблице переходов или символах ленты.

ем интересна машина Тьюринга, преобразованная в строку данных? Раз-ается, тем, что ее теперь можно анализировать с помощью некоторого угого устройства, например, с помощью другой машины Тьюринга или же с помощью самой себя.

акие именно свойства машины Тьюринга могут быть интересны для ализа? Об этом мы еще поговорим в дальнейшем, а пока обсудим, зможно, самую очевидную задачу: имея строковое описание машины ьюринга и содержимое ленты, смоделировать процесс ее работы.

бственно, речь идет, конечно же, о написании универсального эму-тора машин Тьюринга — о задаче, решением которой (с помощью мпьютера) мы уже вроде как занимались. Сейчас я возвращаюсь к даче эмуляции уже на новом уровне. Дело в том, что для ее решения олне достаточно вычислительной мощи машины Тьюринга! Иначе воря, можно создать машину Тьюринга, которая:

- принимает в качестве входной строки закодированное описание некоторой машины Тьюринга с ее входной строкой;

- имитирует работу закодированной машины Тьюринга до тех пор, пока та не перейдет в допускающее (отвергающее) состояние (если эмулируемая машина зацикливается, машина-эмулятор тоже имеет право зациклиться);
- печатает на ленте результат работы эмулируемой машины (допущена или отвергнута ее входная строка) и содержимое ее ленты на момент останова.

Такой эмулятор называется *универсальной машиной Тьюринга* (см. рис. 10.17).

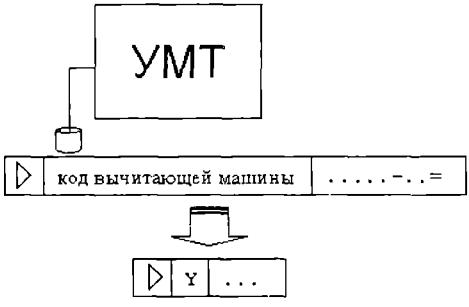


Рис. 10.17. *Универсальная машина Тьюринга*

Универсальной машине Тьюринга на вход можно подать и закодированное определение ее самой. При этом, конечно, входная лента второй машины должна содержать какое-нибудь разумное определение, например, определение еще одной машины Тьюринга — для вычитания чисел (см. рис. 10.18). Можно сколько угодно раз «нанизывать» универсальные машины Тьюринга друг на друга; это похоже на запуск эмулятора вроде Virtual PC или VMware Workstation в уже эмулируемой системе.

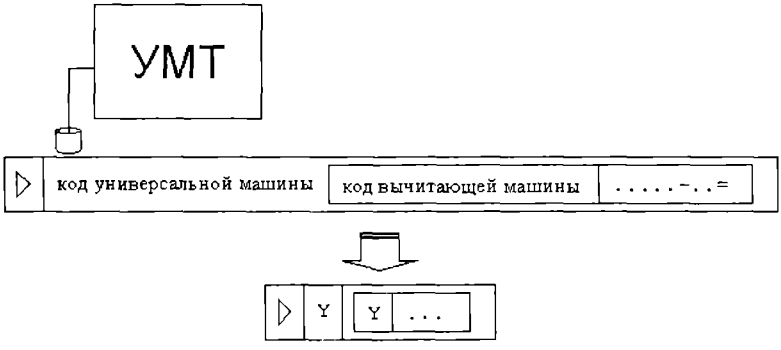


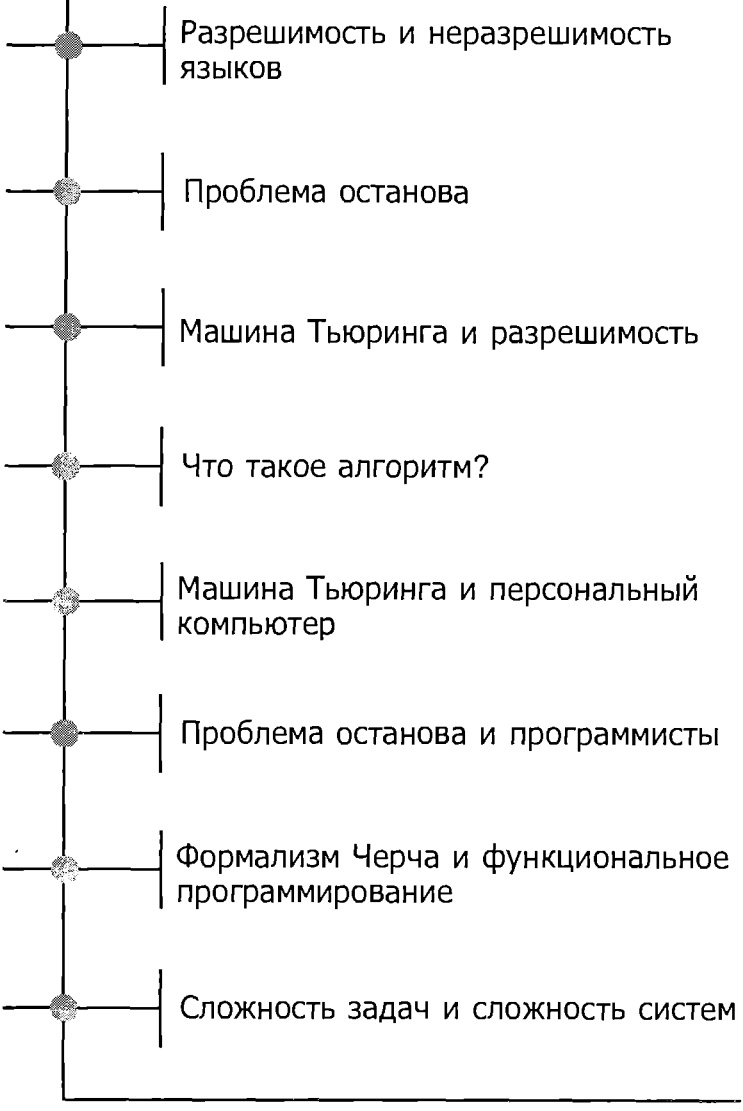
Рис. 10.18. *Эмуляция универсальной машины универсальной машиной Тьюринга*

Итоги

- Машина Тьюринга — новое устройство, по своей вычислительной мощности превосходящее как обычные конечные автоматы, так и автоматы с магазинной памятью. Это очевидно даже из самого устройства машины Тьюринга: намеренно ограничивая свободу действий при составлении таблицы переходов, можно превратить машину Тьюринга в любое из рассмотренных ранее устройств.
- Машина Тьюринга умеет распознавать, по крайней мере, некоторые языки, не являющиеся контекстно-свободными (например, язык $\{a^n b^n c^n\}$).
- Машина Тьюринга (как и другие автоматы) может быть недетерминированной (что, впрочем, не увеличивает ее вычислительной мощи). Недетерминированная машина Тьюринга имеет в основном теоретическое значение; с ее помощью гораздо проще показать, распознаваем ли некоторый язык средствами машины Тьюринга.
- В литературе часто описываются некоторые расширения машин Тьюринга. Например, можно сделать ленту бесконечной в обе стороны[79], превратить одномерные ячейки в многомерные, добавить несколько дополнительных считывающих головок или лент (с отдельной головкой для каждой ленты). Эти расширения служат лишь для удобства конструирования машин Тьюринга; на вычислительную мощь ни одно из расширений не влияет.
- Машина Тьюринга достаточно мощна, чтобы решить задачу эмуляции другой машины Тьюринга. Таким образом, можно создать универсальную машину Тьюринга, которая принимает на ленте в качестве входных данных закодированное описание некоторой машины Тьюринга (вместе с описанием содержимого ее ленты), имитирует переходы переданной машины и печатает на ленте результаты ее работы.

[79] Эта версия нередко используется в качестве базовой.

Глава 11 Разрешимость и сложность



11.1. Разрешимость и неразрешимость языков

В предыдущей главе я не стал акцентировать внимание на классе языков, распознаваемых машиной Тьюринга. Этот вопрос слишком серьезен для того, чтобы упомянуть его лишь вскользь. Сейчас, когда устройство машины Тьюринга и принципы ее функционирования понятны, можно сконцентрироваться на языках, с нею связанных.

Итак, язык, для которого машина Тьюринга может однозначно решить задачу принадлежности (то есть определить, принадлежит данная строка языку или нет, переходя в состояние q_{yes} или q_{no} соответственно), называется *рекурсивным* (recursive) или *разрешимым* (decidable).

Если же задачу принадлежности можно решить лишь для строк, входящих в язык (машина Тьюринга переходит в состояние q_{yes}, если строка принадлежит языку; в противном случае машина либо перейдет в состояние q_{no}, либо никогда не завершит работу), то язык называется *рекурсивно перечислимым* (recursively enumerable) или *частично разрешимым* (partially decidable).

Термины «разрешимый» и «частично разрешимый» напрямую связаны с машиной Тьюринга. Хотя в прошлой главе я использовал термины из теории автоматов, ситуация с машиной Тьюринга несколько сложнее, поскольку помимо перехода в допускающее или отвергающее состояние она может попросту «зависнуть», чего никогда не случалось с конечным автоматом.

Поэтому применительно к машине Тьюринга обычно говорят «разрешает» (decides), если подразумевается, что задача принадлежности решается полностью (то есть входной язык рекурсивен). Термин «допускает» (accepts) употребляется и для рекурсивно перечислимых языков.

Действительно, использование этого термина не противоречит логике, поскольку машина Тьюринга допускает каждую строку языка, в то время

как любая не принадлежащая ему строка будет заведомо отвергнута, либо машина перейдет в бесконечный цикл.

Теперь можно перечислить все типы языков, обсуждаемые в книге (новых языков уже не будет):

Язык	Устройство для распознавания
Регулярный	конечный автомат
Детерминированный	детерминированный автомат с магазинной памятью
Контекстно-свободный	недетерминированный автомат с магазинной памятью
Контекстно-зависимый	машина Тьюринга с ограниченной лентой[80]
Рекурсивный	машина Тьюринга с бесконечной лентой
рекурсивно перечислимый	машина Тьюринга с бесконечной лентой
Неразрешимый	неизвестно

В литературе также часто упоминается иерархия Хомского, в которой перечислены языки и соответствующие им формальные грамматики:

Язык	Грамматика для описания
Регулярный	тип 3: регулярная
Контекстно-свободный	тип 2: контекстно-свободная
Контекстно-зависимый	тип 1: контекстно-зависимая
рекурсивно перечислимый	тип 0: без ограничений[81]

Давайте не будем отвлекаться на детали, а перейдем сразу к главному. Надеюсь, вы уже обратили внимание на появление нового класса — класса *неразрешимых* (undecidable) языков.

Поскольку последним рассмотренным устройством, пригодным для распознавания языков, была машина Тьюринга, термин «неразрешимый» означает «нераспознаваемый с помощью машины Тьюринга».

[80] Для определения принадлежности строки контекстно-зависимому языку достаточно ленты, состоящей из пропорционального длине строки числа ячеек.

[81] В грамматике без ограничений допускаются любые правила вида p → q, где p и q — любые строки, состоящие из терминалов и нетерминалов.

11.2. Проблема останова

Какие же языки не могут быть распознаны машиной Тьюринга? Классический пример связан с анализом свойств машин Тьюринга, закодированных в текстовых строках. В предыдущей главе мы уже обсуждали, как можно представить машину Тьюринга в виде текстовой строки. Там же было сказано, что можно написать эмулятор, имитирующий работу закодированной машины Тьюринга, передаваемой эмулятору в качестве входных данных.

Эмуляция — сравнительно простая задача. Имея закодированную машину Тьюринга, можно придумать много нетривиальных вопросов о ее свойствах. Рассмотрим, к примеру, знаменитую *проблему останова* (halting problem):

```
По заданному строковому представлению машины Тьюринга (с
данными ее входной ленты) за конечное время определить,
заканчивает ли она свою работу или продолжает вычисления
бесконечно.
```

С этой задачей связан язык[82], очень просто формулируемый на русском языке, но недоступный для разрешения машиной Тьюринга:

```
закодированные в виде строк машины Тьюринга, переходящие
(рано или поздно) в состояние q_yes или q_no
```

Доказать, что некоторый язык разрешим, обычно бывает сравнительно просто: для этого требуется лишь построить соответствующую машину Тьюринга. С неразрешимостью дело обычно обстоит сложнее. Однако доказательство неразрешимости задачи останова лаконично и при этом остроумно, поэтому я решился его здесь вкратце описать.

Предположим, что машина Тьюринга, решающая задачу останова, существует. Эта машина (обозначим ее H(m, s)) анализирует закодированную машину m с содержимым ее ленты s. Если машина m заканчивает работу на ленте s[83], машина H переходит в состояние q_{yes}; в противном случае будет произведен переход в состояние q_{no} (обратите внимание, что машина H всегда заканчивает работу). Теперь сконструируем другую машину Тьюринга `T(str)` (содержимое ее ленты обозначено через `str`), выполняющую нехитрый алгоритм:

---

[82] Надеюсь, вы помните об эквивалентности задач и языков?

[83] Разумеется, задача останова решается для некоторой машины Тьюринга и тех или иных данных ее ленты. Одна и та же машина может заканчивать работу или переходить в бесконечный цикл в зависимости от входных данных.

```
имитировать работу машины H(str, str);
ЕСЛИ в результате машина H перешла в состояние q_no
    перейти в состояние q_yes;
ИНАЧЕ выполнять бесконечный цикл;
```

Таким образом, машина T заканчивает работу, если переданная ей в виде строки машина str «зависает».[84] В противном случае T переходит в бесконечный цикл сама.

«Зависает» ли машина T(t) (где t — закодированное представление машины T)?

Предположим, что да. Это означает, что выполнение алгоритма пошло по ветви ИНАЧЕ, что, в свою очередь, возможно, если H(t, t) заканчивает работу в состоянии q_{yes}. По определению машины H это переводится как «машина t с лентой t не зависает». Получается, что машина T не зависает! Мы пришли к противоречию.

Рассмотрим теперь вторую альтернативу: машина T(t) заканчивает работу. Это означает, что машина H(t, t) переходит в состояние q_{no}, то есть «машина t с лентой t не заканчивает работу (зависает)». Противоречие!

Противоречия в рассуждениях возникли потому, что было предположено существование машины H. Следовательно, машины, решающей задачу останова, не существует.

11.3. Машина Тьюринга и разрешимость

Машина Тьюринга — не первое устройство, которому не по зубам те или иные языки, но на сей раз проблема нехватки вычислительной мощи в корне отличается от похожего недостатка конечных или магазинных автоматов. Дело в том, что машина Тьюринга представляет собой самое мощное вычислительное устройство, когда-либо изобретенное человеком.

Алан Тьюринг отнюдь не пытался изобрести какую-то абстрактную машину, наделенную неизвестно откуда взявшейся функциональностью. Идея Тьюринга была в изобретении механизма, в некоторой степени имитирующего работу человека, занимающегося решением вычислительной задачи. При этом Тьюринг попытался выделить атомарные действия, к которым теоретически удастся свести все остальные.

[84] Конечно, это очевидно, но все-таки обратите внимание, что выражения «зависает» и «заканчивает работу» противоположны по смыслу (порою возникает неожиданная путаница).

В итоге устройство, называемое сейчас машиной Тьюринга, было описано в статье 1936 года «On Computable Numbers, with an Application to the Entscheidungsproblem». Через двенадцать лет (уже после изобретения компьютеров) Тьюринг пишет: «Можно воспроизвести результат работы вычислительной машины, если записать множество правил, составляющих процедуру, и попросить человека их выполнить. Такое сочетание человека с письменными инструкциями можно назвать «бумажной машиной». Человек с бумагой, карандашом и ластиком при условии строгой дисциплинированности представляет собой универсальную машину».[85]

На первый взгляд, современные компьютеры умеют делать гораздо больше, чем машина Тьюринга. На самом же деле это не так. Можно показать, что на базовом уровне любая инструкция компьютера может быть сымитирована на машине Тьюринга.

Разумеется, речь идет лишь о вычислительных задачах. Не совсем правильно указывать на отсутствие у машины Тьюринга хорошего монитора или звуковой карты, поскольку решение настоящей вычислительной задачи всегда можно свести к получению некоторого набора чисел (результатов) на основе другого набора чисел (входных данных). Вот преобразование очередной порции данных сжатого при помощи MPEG4 файла в набор из 1024 x 768 = 786432 чисел, представляющих собой выводимые на монитор точки, вполне вписывается в формулировку вычислительной задачи.

Тот факт, что мы за последние семьдесят лет не сумели придумать ничего более мощного, чем машина Тьюринга, определенно свидетельствует в пользу невероятно сильного, но по сей день так и не доказанного (и не опровергнутого, разумеется) утверждения, известного как тезис Черча-Тьюринга (Church-Turing thesis):

Любая теоретически разрешимая вычислительная задача может быть решена при помощи машины Тьюринга.

Таким образом, тезис Черча-Тьюринга не только утверждает тот факт, что машина Тьюринга на сегодняшний день является самым совершенным вычислительным устройством, но и выражает предположение, что ничего более мощного не может быть изобретено в принципе.

[85] «Intelligent Machinery» (1948); перевод фрагмента мой.

11.4. Что такое алгоритм?

Историческая перспектива

Задачи неотделимы от алгоритмов решения. Понятие алгоритма в науке далеко не новое, но лишь в XX веке была предпринята (на сегодняшний день кажущаяся удачной) попытка его формализовать. Вы будете смеяться, но речь опять пойдет о машине Тьюринга.

Справедливости ради я обязательно хочу отметить вклад и других ученых в основы теории вычисления. Как это нередко бывает, если задача по-настоящему назрела, сразу несколько специалистов примерно в одно и то же время находят интересные решения. Подход Тьюринга, вероятно, оказался наиболее продуктивным для развития компьютерной науки (мне кажется, не в последнюю очередь потому, что сам Тьюринг очень активно работал с вычислительными машинами и практическими задачами, в то время как другие ученые нередко рассматривали свои исследования скорее с точки зрения математики или логики). Тем не менее, примерно в те же годы различные модели для формализации вычислительных процедур предложили Алонзо Черч (чье имя уже упоминалось и еще встретится в дальнейшем), Эмиль Пост, Андрей Марков и другие ученые. Было показано, что все эти модели могут быть сведены друг к другу.

Неформальная инструкция как алгоритм

Вернемся, однако, к алгоритмам. Как обычно определяют понятие алгоритма, например, в школе? Обычно говорят что-то о способе решения задачи, представляющем собой точную инструкцию для получения результата на основе исходных данных. Конечно, это определение хорошо тем, что оно позволяет нам (более или менее) легко понять, о чем вообще идет речь — об инструкции для решения задачи или о бутылке кефира. С другой стороны, можно легко придраться к сочетанию «точная инструкция». Насколько такая инструкция должна быть точна?

Кому из нас не попадались книги наподобие «100 способов похудеть» (если хотя бы один работает хорошо, зачем оставшиеся 99?), «как стать успешным человеком» или «как добиться душевного спокойствия»? Ведь рецепты из этих книг — по сути своей алгоритмы, но уровень их описания безнадежно высок по сравнению с уровнем той же машины Тьюринга или языка Pascal. Я сам вполне могу изобрести гениальный «алгоритм» (вполне четкую пошаговую инструкцию), позволяющей любой команде стать чемпионом мира по футболу, выигрывая все матчи подряд:

```
ПОКА время матча не истекло
    ЕСЛИ мяч у соперника
        перехватить мяч, пока нам не забили гол;
    забить гол сопернику;
КОНЕЦ ЦИКЛА
```

Машина Тьюринга как алгоритм

Теперь можно перейти от нечеткого определения алгоритма как «инструкции» к совершенно формальному представлению об алгоритме как о некотором устройстве (пусть это даже на первый взгляд кажется довольно странным). По сути дела, машина Тьюринга — это не устройство, выполняющее некоторый алгоритм; *это и есть сам алгоритм*.

Помните, каким образом решались задачи из главы о машине Тьюринга? Имея формулировку задачи, мы пытались построить машину Тьюринга, требуемым образом преобразующую входные данные в выходные. А ведь построение такой машины и означает построение алгоритма решения! Сравните, например, такие утверждения:

```
Существует алгоритм решения задачи сортировки массива.
Существует машина Тьюринга, сортирующая массив.
```

Для каждой задачи требуется своя собственная машина Тьюринга, точно так же, как и для решения той или иной задачи требуется отдельный алгоритм. Единственное серьезное возражение связано с ограниченными возможностями машины Тьюринга. Все-таки под алгоритмом понимается некая инструкция, которую можно выполнить в принципе, а не вариация какого-то (возможно, недостаточно мощного) устройства, умеющего решать лишь заведомо неполный набор задач. Принятие на веру тезиса Черча-Тьюринга[86] снимает это возражение, устанавливая полную эквивалентность между понятиями алгоритма и машины Тьюринга.

Если алгоритм и машина Тьюринга — это одно и то же, можно провести еще несколько интересных параллелей.

Машина Тьюринга и языки программирования

Написание компьютерной программы на любом языке программированию эквивалентно созданию машины Тьюринга[87] для решения конкретной задачи. Поскольку на любом, даже самом простом языке можно написать

[86] Конечно, в науке ничего нельзя принимать на веру, но семьдесят лет безуспешных попыток изобрести хотя бы одно более мощное устройство, несводимое к машине Тьюринга, говорят сами за себя.

[87] Точнее, закодированной в строку машины Тьюринга. К сожалению, в процессе компиляции программы настоящая железная машина на столе сконструирована не будет.

эмулятор машины Тьюринга, с точки зрения теории вычислений все языки программирования одинаково мощны. Конечно, одни языки могут быть удобнее других с точки зрения программиста, но удобство не добавляет ничего к выразительной мощи. То же самое можно сказать о дополнительных возможностях вроде поддержки работы с COM объектами или умения генерировать DLL-модули: без них трудно себе представить современное программирование, но к решению вычислительных задач все это не имеет никакого отношения.

Чтобы окончательно убедить вас в том, что любой язык программирования не лучше машины Тьюринга (в смысле выразительной мощи, разумеется), я приведу версию доказательства неразрешимости проблемы останова для языка C#.

Итак, предположим, что существует функция

```
bool Halts(string func, string param),
```

принимающая два параметра:

1. Текстовое представление `func` некоторой корректной функции, записанной на C#. Например, `func = "void myprint(string s) {Console.WriteLine(s);}"`. Функция, записанная в переменной `func`, должна принимать единственный аргумент, принадлежащий типу `string`.
2. Текстовый аргумент `param`, передающийся на вход функции `func`.

Функция `Halts()` возвращает `true`, если функция `func` заканчивает работу на входных данных `param` (значение `false` возвращается в противном случае).

Рассмотрим функцию `Test()`:

```
void Test(string str)
{
    if(Halts(str, str))
        for(;;) ;
}
```

Теперь разберемся, что происходит при вызове `Test(s)`, если `s` содержит текст функции `Test()` (то есть `s = "void Test(string str) {if(Halts(str, str)) for(;;) ;}"`). Если `Test(s)` завершает работу, это означает, что значение `Halts(s, s)` равно `false`, то есть `Test(s)` «зависает»; получили противоречие. Если `Test(s)` не завершает работу, то значение `Halts(s, s)` равно `true`, что, в свою очередь, означает, что `Test(s)` не «зависает». Опять противоречие. Таким образом, функция `Halts()` не может существовать.

В теории вычислений известна теорема Райса, которая гласит, что любая задача, связанная с выявлением какого-либо «нетривиального» семантического

йства машины Тьюринга (ну или компьютерной программы), неразреши-
Неразрешимость проблемы останова — очень мощный результат, по-
ляющий во многих случаях судить о разрешимости или неразрешимости
или иной задачи, поскольку многие семантические свойства могут быть
ко сведены к задаче останова. Допустим, имеется функция

```
oid f()

   // здесь идет некоторый код
   ...
   Console.WriteLine("hello, world!");
```

жно ли определить, выполнится ли когда-нибудь строка, выводящая
экран "hello, world!"? Если переписать программу в виде

```
oid h()

   // некоторый код
   ...


oid f()

   h();
   Console.WriteLine("hello, world!");
```

новится ясно, что печать строки зависит от того, заканчивает ли функ-
h() свою работу или нет. В общем виде задача неразрешима (хотя
некоторых частных случаев решение существует), поэтому можно
лать вывод о неразрешимости исходной задачи.

11.5. Машина Тьюринга и персональный компьютер

ли компьютерная программа представляет собой (в некотором при-
жении) код машины Тьюринга, то персональный компьютер вполне
кно назвать универсальной машиной Тьюринга, умеющей имитировать
оту других закодированных машин.

самом деле обычный компьютер в нормальных условиях даже слабее
лины Тьюринга. Дело в том, что машина Тьюринга обладает бесконеч-
лентой, в то время как память компьютера (или память, используе-
той или иной конкретной программой) конечна. Имея возможность
ключать дополнительную неограниченную память (например, на

CD-RW носителях), можно получить «настоящую» машину Тьюринга; хотя, конечно, это скорее теоретическое замечание.

Здесь я хочу обратить ваше внимание на то, что проблема останова теоретически может быть решена для программы, оперирующей конечной памятью (хотя на практике это решение все равно не годится из-за неприемлемых требований к памяти и времени).

Давайте обсудим решение задачи останова для самого простого случая. Пусть анализируемая программа в процессе работы модифицирует конечное, заранее известное количество ограниченных ячеек памяти[88] (назову их «переменными»).

Пусть $V_1, V_2, \ldots, V_k$ — значения всех переменных программы, а N — номер текущей (выполняемой) строки. Допустим также, что их начальные значения переменных (перед тем, как программа запускается на выполнение) заранее определены. Тогда величины всех этих параметров в совокупности однозначно определяют текущее состояние программы, то есть, зная их значения в некоторый момент времени, можно однозначно предсказать, в каком состоянии программа окажется после выполнения очередной инструкции.

Непонятно? Рассмотрим простой пример:

```
    int i = 1, j = 1;
1: i = 5;
2: j = 7;
```

Начальное состояние этой малополезной программы, состоящей из двух строк (давайте не будем считать секцию описания начальных значений переменных отдельной строкой), может быть закодировано перечислением значений всех ее переменных:

```
i = 1, j = 1, N = 1
```

То есть значение i равно единице, значение j равно единице, текущая (выполняемая) строка — первая. В какое состояние программа перейдет после выполнения очередной строки? Конечно же, в состояние

```
i = 5, j = 1, N = 2
```

Далее, обладая лишь информацией о текущем состоянии (неважно, каким путем программа в него попала), можно предугадать следующее состояние:

```
i = 5, j = 7, N = 3
```

[88] То есть отдельная ячейка памяти может содержать один, два или хоть десять байт информации, но не бесконечный объем.

осле выполнения конечного числа шагов анализа состояний мы либо йдем до конца программы (это означает, что программа завершает боту), либо вернемся к одному из уже ранее встретившихся состояний, о наводит на мысль о бесконечном цикле в исходной программе.

римеры обеих ситуаций показаны ниже (листинги 11.1 и 11.2, с. 11.1).

стинг 11.1. Программа, завершающая работу

```
  int i = 0;
: for(i = 0; i < 3; i++)
:     Console.WriteLine(i);
```

стинг 11.2. Программа, работающая вечно

```
  int i = 0;
1:   for(i = 0; i < 3; i++)
  {
2:    Console.WriteLine(i);
3:    if(i == 2)
4:        i = 1;
  }
```

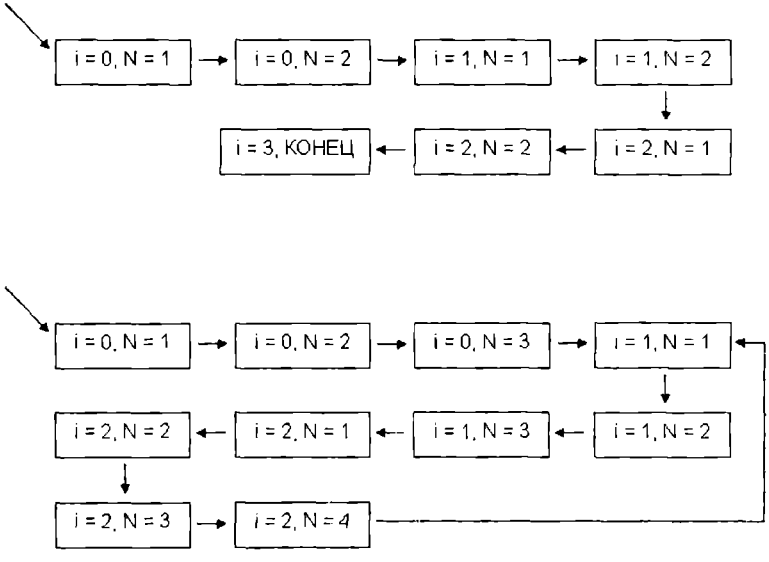


Рис. 11.1. *Диаграммы состояний обеих программ*

11.6. Проблема останова и программисты

Человек против компьютера

Я не решился обойти вниманием этот популярный вопрос: «А разве человек не может решить проблему останова? Что стоит опытному программисту, внимательно изучив текст функции, определить, завершит ли она работу или «зависнет» в бесконечном цикле?»

Ниже я еще вернусь к несколько более сильной формулировке тезиса Черча-Тьюринга, а пока отвечу кратко: по всей видимости, не может. Программист может найти ответ в простых случаях наподобие показанного на листинге 11.2. Впрочем, в такой ситуации хороший компилятор тоже сообщит о недостижимости строк, следующих за бесконечным циклом.

Примерами, повсеместно цитируемыми, являются поиск нечетных совершенных чисел и поиск уравнения, противоречащего теореме Ферма. Поскольку теорема Ферма была доказана в девяностых годах прошлого века, я остановлюсь на первом примере.

Совершенным называется число, равное сумме своих делителей: $6 = 1 + 2 + 3$; $28 = 1 + 2 + 4 + 7 + 14$. На сегодняшний день неизвестно, существует ли хотя бы одно нечетное совершенное число.

Теперь скажите, «зависает» ли следующая программа:

```
static bool IsPerfect(int k)  // определяет, является ли
                              // число k совершенным
{
    int s = 0;
    for(int i = 1; i <= k / 2; i++)
        if(k % i == 0)
            s += i;           // найти сумму делителей k

    return s == k;
}

static void Main(string[] args)
{
    int num = 1;
    for(;;)
    {
        if(IsPerfect(num))    // если текущее число совершенно
            break;            // выйти из программы
        num += 2;     // перейти к следующему нечетному числу
    }
}
```

Понятно, что результат ее работы зависит от того, существует ли хотя бы одно нечетное совершенное число, но вряд ли хоть один программист в мире знает это наверняка.

Впрочем, если приведенную программу рассматривать не как абстрактный алгоритм, а как реальную программу на C#, имеющую дело с ограниченными переменными, результат работы предсказать как раз несложно: все закончится переполнением `num`.

Сильная версия тезиса Черча-Тьюринга

Если довести сказанное о задаче останова до логического завершения, можно прийти к сильной версии тезиса Черча-Тьюринга:

Теоретически компьютер может сделать все,
что может сделать человек.

Лично мне подобная идея совсем не кажется дикой. Основные возражения так или иначе сводятся к религиозному взгляду на человека как на существо, обладающее *душой*. Мы, в свою очередь, неспособны вдохнуть душу в устройство из кремния, пластмассы и металла. Душа же (и только она) обладает многими интересными свойствами, такими как творческое начало или свобода воли; без этих качеств никакой разум, никакое самосознание не может существовать, и, следовательно, компьютер принципиально никогда не достигнет уровня человека.

Тем, кто безоговорочно верит в подобную (данную Богом и не воспроизводимую человеком) идею души, приводить аргументы бесполезно. Для всех остальных могу порекомендовать отменное, увлекательное чтение[89]. Здесь лишь скажу, что, по моему мнению, создать разумную компьютерную программу вполне возможно. Но, скорее всего, мы сами не сумеем понять, как она, собственно, работает. Иначе говоря, вложив в компьютер некие начальные правила и пустив процесс на самотек, мы потеряем не только контроль над происходящим, но и утратим собственное понимание ситуации.

О том, как можно полностью осознавать правила, лежащие в основе системы, но при этом быть не в состоянии предсказывать ее поведение, речь пойдет в конце главы.

[89] A. M. Turing. Computing Machinery and Intelligence. MIND vol. 59, 1950.
Д. Деннетт, Д. Хофштадтер. Глаз разума. Бахрах, 2003.
Д. Хофштадтер. Гедель, Эшер, Бах: эта бесконечная гирлянда. Бахрах, 2001.

11.7. Формализм Черча и функциональное программирование

11.7.1. Альтернативные модели вычисления

Думаю, вы уже заметили, что машина Тьюринга по своей идеологии очень похожа на обычный компьютер (хотя и с крайне ограниченным набором команд). Конструкции обычного императивного[90] языка программирования тоже в известном смысле напоминают выполнение операций машины Тьюринга: мы считываем и записываем переменные, находящиеся в оперативной памяти (сравните со считыванием и записью содержимого ячеек ленты), вызываем различные процедуры (это похоже на переход к состоянию, начинающему логический модуль машины Тьюринга[91]), программируем условия и циклы. Согласитесь, псевдокод, описывающий работу той или иной машины Тьюринга из предыдущей главы, всегда был похож на псевдокод обычной программы.

Ну разумеется, а разве иначе бывает? Наверно, для многих это будет неожиданностью, но бывает. Популярные сегодня языки программирования основаны на модели вычислительного процесса, предложенной Тьюрингом — записать / стереть / перейти к блоку при выполнении некоторого условия. Тем не менее, существуют языки, основанные на альтернативной модели вычисления, предложенной Алонзо Черчем — на так называемом лямбда-исчислении.

Подобные языки называются *функциональными*. Самые известные представители семейства функциональных языков — Standard ML (SML), Haskell и LISP[92]. Программируя на функциональном языке, приходится мыслить совершенно иными категориями. Не берусь судить, стоит ли нам ожидать в ближайшие годы расцвета функционального программирования (учитывая его достоинства и недостатки), но познакомиться с ним ради общего развития, безусловно, стоит[93].

---

[90] К императивным относятся все самые популярные на сегодня языки вроде Basic, Pascal, C/C++, Java, C# и все остальные языки со схожей философией кодирования алгоритмов.

[91] Трудно назвать машину Тьюринга инструментом для структурного программирования, но все же можно обвести карандашом несколько состояний и на словах описать, что конкретно эта часть машины (модуль) делает.

[92] Язык LISP также содержит некоторые возможности, выходящие за рамки функционального программирования.

[93] Прошу прощения, что экскурс в функциональное программирование оказался внутри вроде бы не относящейся к нему главы. Но выделять эту тему в отдельную главу мне совсем не хочется.

11.7.2. Совсем чуть-чуть о лямбда-исчислении

Декларативный подход к программированию

Обсуждение этой темы, конечно, выходит далеко за рамки книги. Я лишь кратко опишу идеи, положенные в основу λ-исчисления и приведу простейшие примеры функциональных программ. Все-таки модели вычислительного процесса вписываются в общее направление нашего разговора, а λ-исчисление — модель весьма известная и к тому же имеющая массу поклонников.

Основой концепции Черча является *функция* в том смысле, в каком ее понимают математики. Чем функция в математике отличается от функции в императивном языке программирования? Функция в математике по какому-то правилу сопоставляет набору значений аргументов некоторое результирующее значение. И больше ничего. Например, функция `sin` сопоставляет аргументу, равному $\pi/2$, значение 1.

На первый взгляд, функция в любом языке программирования делает абсолютно то же самое, но это не совсем так. Поскольку при выполнении программного кода могут возникать побочные эффекты (изменение глобальных переменных, считывание данных с клавиатуры или из файла), можем ли мы рассчитывать, что повторный вызов некоторой функции для одного и того же набора входных параметров приведет к получению того же самого результата? В математике запись $\sin(\pi/2)$ всегда может использоваться как синоним числа 1. Можем ли мы в программировании, однажды записав

```
v = MyFunction(5, 15);
```

везде вместо v применять конструкцию MyFunction(5, 15)? Вряд ли. Во-первых, второй вызов MyFunction() может вернуть совершенно другой результат, а во-вторых, любой программист скажет, что вызывать один и тот же алгоритм несколько раз попросту неэффективно.

В качестве примера функции, возвращающей разный результат при одних и тех же аргументах, проще всего указать метод Console.ReadLine(). Получая каждый раз пустой набор аргументов, этот метод возвращает различные значения, вводимые пользователем с клавиатуры.[94] Насчет эффективности можно заметить, что в математике запись функции не подразумевает какого-либо «вызова на выполнение». Записывая $y = k^2$,

[94] Решение вычислительной задачи подразумевает получение результатов на основе входных данных. Считывание пользовательского ввода формально не является вычислительной задачей, поэтому с помощью «чистого» функционального подхода решить ее не удастся. С другой стороны, теории теориями, а на практике, конечно, никому не нужен язык, не умеющий работать с файловой системой или консолью. Поэтому все функциональные языки содержат дополнительные средства для решения подобных рутинных задач.

я не говорю: «вычислить значение k^2 и присвоить полученный результат переменной y»; я всего лишь хочу сказать, что значение y равно k, возведенному в квадрат. При чем здесь эффективность?

По сути дела здесь сталкивается подход, более естественный для математиков, с «пошаговой инструкцией» приверженцев императивного программирования.

Программист указывает четкую инструкцию, позволяющую вычислить значения требуемых переменных. Математик же записывает с помощью формул, как интересующий нас результат может быть выведен из неизвестных. Предположим, требуется решить уравнение 10*x + 5 = 25. Программист (императивник) составляет инструкцию:

```
вычесть 5 из 25
разделить результат на 10
напечатать полученное значение
```

Математик просто утверждает, что значение x равно (25 – 5)/10. Таким образом, подход, принятый в математике, сконцентрирован прежде всего на том, чем являются результаты, а не на способе их получения. Иными словами, такие выкладки *декларативны*: мы указываем, что из чего может быть выведено, а в каком порядке все это следует вычислять — проблема компилятора (или интерпретатора, так как большинство функциональных языков реализуются именно в таком виде). Разумеется, в конечном счете любые описания будут переведены в императивные инструкции, выполняемые процессором, но выведение инструкций из математических описаний опять-таки ложится на плечи среды, а не программиста.[95]

Описания столь высокого уровня избавлены от утечек памяти, затирания глобальных переменных и ошибок, связанных с многопоточным доступом. Абстрагирование от деталей (это относится к любой парадигме программирования) позволяет создавать более элегантные и надежные программы с меньшими усилиями. Кроме того, используя функциональный подход, проще создавать новые функции (и даже целые программы) на основе уже существующих.

Ценой за избавление программиста от излишних деталей, как обычно, становится эффективность. Впрочем, нередко интерпретатор функционального языка может выполнить оптимизацию, невозможную в любом императивном языке. Например, поскольку значение функции всегда однозначно соответствует набору значений аргументов, повторное вычисление функции всегда можно заменить считыванием из кэша ранее вычисленного (и запомненного) значения.

[95] К сожалению, полностью избавиться от размышлений о том, как конкретно будет вычисляться та или иная функция, не удастся, если мы хотим создавать эффективные программы.

Основания формализма Черча

Вернемся к работам Черча. Если функциональное программирование основано на стандартном подходе математиков к решению задач, при чем здесь лямбда-исчисление?

Математические записи недостаточно формальны для построения модели вычисления. Мы всегда неявно используем много различных допущений, читая математические формулы. Например, что такое `f(x)`? Вероятно, некоторая функция. А что такое $f(x_0)$? Скорее всего, число[96]. Обычно переменная без индекса считается свободной; она лишь указывает место, предназначенное для подстановки некоторой конкретной величины. Далее, что такое $f(g(x_0 + 5))$? Композиция функций? А может, `g` — не имя функции, а обычная переменная? Тогда значение `f` определяется в точке $g^*(x_0 + 5)$.

Заслуга Черча состоит в разработке формальных правил записи функций и операций над ними. Если не углубляться в детали, суть состоит в работе с так называемыми *λ-выражениями* и *правилом редукции*.

Лямбда-выражения

Лямбда-выражение можно записать с помощью контекстно-свободной грамматики:

```
E → I | (λI . E) | (E E)
```

Здесь `E` — это λ-выражение, а `I` — некоторый идентификатор (переменная) либо константа. Константы относятся к расширениям λ-исчисления, поддерживаемым функциональными языками программирования. Также на практике допускается использование в лямбда-выражениях арифметических операций. Выражение вида (λI . E) служит для формального определения функции. Префикс λI . указывает на то, что переменная `I` — аргумент функции, задаваемой выражением `E`. Например, функция `f(x) = 5*x` в новых обозначениях будет записана так:

```
f = (λx . 5*x)
```

В лямбда-исчислении любое λ-выражение описывает функцию, имеющую один аргумент. Этот аргумент, как и возвращаемое функцией значение, сам по себе является λ-выражением.

Функции от двух и более аргументов можно описать с помощью цепочки λ-выражений. Так, функция `f(x, y) = 5*x + y` будет выглядеть как

```
f = (λx . (λy . 5*x + y))
```

[96] В математике, как и в программировании, значением функции может быть не только обычное число, но и «объект», такой как матрица или вектор. В функциональном программировании возможность указать в качестве возвращаемого значения сложный тип данных тоже предусмотрена.

Правило редукции

Правило редукции заключается в преобразовании записи вида ((λ I . E) E') в λ-выражение, получаемое заменой каждого символа `I` выражения `E` на выражение `E'`. Например:

```
((λx . 5*x) 3) = 5*3
```

Таким образом, правило редукции позволяет подставлять фактические значения аргументов, чтобы получить значение функции в интересующей точке. Для функции двух аргументов подстановка выглядит так:

```
((λx . (λy . 5*x + y)) (1 2)) = ((λy . 5*1 + y) 2) = 5*1 + 2
```

Если в результате подстановки получается арифметическое выражение, состоящее лишь из известных величин, среда программирования может его вычислить.

Обратите внимание, что функция от двух аргументов представляет собой композицию функций, то есть функцию, аргументом которой является другая функция!

Программирование на «чистом» лямбда-исчислении, вероятно, было бы запредельно сложным. К счастью, функциональные языки предоставляют программисту более удобный (но по смыслу сходный) синтаксис.

11.7.3. Примеры функциональных программ

Простейший пример: вычисление факториала

Теперь я хочу привести несколько простых примеров на настоящем функциональном языке программирования Standard ML (я выбрал его лишь потому, что знаю его лучше других языков).

Начнем с функции вычисления факториала, которая на SML выглядит почти так же, как и на императивных языках:

```
fun factorial 0 = 1
|   factorial n = n*factorial(n - 1)
```

В описании функции факториала перечислены два факта:

1. факториал нуля есть единица;
2. факториал числа `n` есть `n`, умноженное на факториал `n - 1`.

Загрузив определение функции в интерпретатор SML, можно попробовать задать несколько вопросов. Например, можно посмотреть, что представляет собой идентификатор `factorial`:

```
- factorial;
```

реда SML ответит:

```
val it = fn : int -> int
```

есь сказано, что факториал — это функция (fn), переводящая аргумент па `int` в значение, также имеющее тип `int`. Откуда взялись типы? роанализировав описание функции, среда SML определила, что мы пользуем входной аргумент в арифметическом выражении, поэтому р тип был распознан как числовой. Результат вычисления выражения, держащего числовые данные, также является числом.

ри желании типы аргументов можно указывать и вручную, но исполь-вание бестиповых переменных порою позволяет писать универсальные нкции. Допустим, требуется отсортировать массив. Какая разница, что за ъекты он содержит? Достаточно знать, что элементы массива могут быть орядочены с помощью какой-нибудь служебной функции сравнения.

Условная операция. Сокращенная схема вычислений

ссмотрим теперь определение функции, возвращающей большее из ух чисел:

```
fun max(x, y) = if x > y then x else y
```

а простая функция иллюстрирует сразу две новых конструкции. Первая них — условная операция `if`. Обратите внимание, что конструкция `if` зыке SML не имеет ничего общего с ветвлением в императивных языках. словная операция работает аналогично операции (условие ? x : y) языка : если условие выполняется, то результатом будет значение x, иначе — y. обы окончательно убедить вас, что использование условной операции противоречит принципам функционального программирования, можно реписать условную операцию в форме обычной функции:

```
fun my_if(true, x, y) = x
   my_if(false, x, y) = y
```

к, на запрос `my_if(1 < 5, 1, 5);` система SML выдаст ответ

```
val it = 1 : int
```

жным отличием стандартной операции `if` языка SML от функции _if является используемая ей сокращенная схема вычисления. Как оисходит вычисление, например, выражения `if n <> 0 then 0 div n else 0`?[97] Сначала определяется, выполняется ли условие <> 0. Если условие выполняется, вычисляется и возвращается значение 0 div n; в противном случае возвращается значение 0. Таким образом,

[97] Выражение a div b означает целочисленное деление a на b.

из трех выражений, участвующих в операции `if`, всегда вычисляется лишь два: условие и выражение, которое формирует возвращаемое значение.

При вычислении результата функции `my_if` значения всех трех аргументов будут определяться в каждом случае. Поэтому вызов, казалось бы, эквивалентной функции `my_if(n <> 0, 100 div n, 0)` для значения `n`, равного нулю, приведет к ошибке деления на нуль[98].

Вторая новая конструкция, использованная в тексте функции `max`, становится заметной, если вывести на экран описание идентификатора `max`:

```
- max;
val it = fn : int * int -> int
```

Обратите внимание, что функция принимает лишь один аргумент. Кроме того, тип аргумента на первый взгляд выглядит необычно — int * int. Тот факт, что любая функция принимает лишь один аргумент, неудивителен: именно так и должно быть, если следовать принципам λ-исчисления. Перечислив через запятую два аргумента в круглых скобках, мы, фактически, описали только один аргумент, являющийся *кортежем* из двух целых переменных (int * int).

Функции высших порядков

Если вы помните, в λ-исчислении можно записать функцию от двух аргументов в виде цепочки двух одноаргументных функций. Язык SML тоже поддерживает эту возможность, поэтому функцию `max()` допустимо также определить как

```
fun max x y = if x > y then x else y
```

На первый взгляд, почти ничего не изменилось, но посмотрите на новое описание max в среде SML:

```
val it = fn : int -> int -> int
```

Единственному целочисленному аргументу функции max теперь соответствует возвращаемое значение, само являющееся функцией. Эта функция, в свою очередь, переводит целый аргумент в целое значение. Функцию max можно использовать тем же способом, что и раньше:

```
- max 2 5;
val it = 5 : int
```

Можно также в явном виде получить функцию, которая является возвращаемым значением max после подстановки какого-нибудь значения аргумента, например, 5:

```
fun p_max x = max 5 x
```

[98] Это верно для SML, но не относится к языкам вроде Haskell, поддерживающим так называемые ленивые вычисления (lazy evaluations).

:перь можно сравнивать любое число с пятеркой, вызывая p_max:

```
- p_max 3;
val it = 5 : int
- p_max 10;
val it = 10 : int
```

Операции со списками

апоследок давайте обсудим работу со списками.

писок — это стандартное для функционального программирования едставление последовательности элементов (любого типа). Списки, заваемые как заключенные в квадратные скобки наборы перечисленных рез запятую элементов, напрямую поддерживаются языком SML. Ниже казан диалог с системой SML, сообщающий о том, что объект [1, 2, 3] ляется списком целых чисел:

```
- [1, 2, 3];
val it = [1,2,3] : int list
```

ля объявления спискового аргумента функции используется специальй синтаксис:

- запись [] обозначает пустой список;
- запись e_1 :: e_2 :: ... :: e_n :: e_{tail} обозначает список из элементов e_1, e_2, ..., e_n, за которыми следует список e_{tail} («хвост») из всех остальных элементов.

остейшим примером алгоритма, работающего со списком, может ужить функция member, которая определяет, является ли элемент e еном данного списка:

```
fun member (e, []) = false
    member (e, h::tail) = if e = h then true else member (e, tail)
```

русский язык это описание переводится так:

1. элемент e не является членом пустого списка (каким бы ни был e);
2. элемент e является членом списка, если он равен первому элементу списка; в противном случае задача сводится к определению, является ли e элементом хвоста исходного списка.

к видите, в функциональном программировании нет ни присваиваний, переменных, ни циклов[99]. Единственный способ организовать многоатное выполнение некоторой операции состоит в создании рекурсивй функции.

[99] Я говорю о «чистом» функциональном программировании. Любой развитый язык предоставляет какие-то возможности, выходящие за эти рамки.

Сортировка списка

Теперь давайте рассмотрим более интересный пример — сортировку списка чисел. Запрограммировать ее можно, конечно, по-разному. Я предлагаю следующий подход. Допустим, функция `rotate` преобразует список таким образом, что его минимальный элемент оказывается в самом начале. Далее, пусть список `(nh::ntail)` получается в результате применения `rotate` к исходному списку. Тогда можно получить отсортированный исходный список, если присоединить `nh` к отсортированному хвосту `ntail`. Вот и все.

Мне кажется, что рекурсивные конструкции на языке программирования выглядят понятнее, поэтому приступим к работе. Для начала нам потребуются две служебные функции. Первая находит минимальный элемент в заданном списке:

```
fun listmin (e::nil) = e
|   listmin (h::tail) = if h < listmin tail then h else listmin tail
```

Минимальным элементом одноэлементного списка `(e::nil)` (зарезервированное слово `nil` обозначает пустой хвост) является его единственный элемент. Если же в списке присутствует более одного элемента, следует вернуть меньшее из двух чисел: головы списка и минимального элемента хвоста.

Вторая функция добавляет новое число в конец списка:

```
fun append (e, []) = (e::nil)
|   append (e, h::tail) = (h::append(e, tail))
```

Добавить число в начало списка очень просто. Если дан элемент `e` и список `(h::tail)`, достаточно всего лишь воспользоваться записью `(e::h::tail)`. Добавить что-либо в конец уже сложнее: придется сначала дойти до конца списка, а уже потом вносить изменения.

Теперь определим функцию `rotate`:

```
fun rotate [] = []
|   rotate (h::tail) = if h = listmin (h::tail) then (h::tail)
                       else rotate (append(h, tail))
```

Если входной список пуст, преобразования не требуются. Если первый элемент списка является одновременно его минимальным элементом, ничего менять тоже не надо. В противном случае придется рассмотреть список, в котором головной элемент перенесен в самый конец. Посмотрим, к примеру, как преобразуется список [2, 3, 1, 5] в процессе вычисления функции `rotate`:

```
[2, 3, 1, 5]: условие h = listmin (h::tail) не выполняется;
                    вызываем rotate для списка [3, 1, 5, 2]
[3, 1, 5, 2]: то же самое, вызываем rotate [1, 5, 2, 3]
```

```
[1, 5, 2, 3]: голова списка равна его наименьшему элементу,
              заканчиваем работу
```

Осталось лишь написать функцию `sort`, которая выглядела бы совсем просто, если бы не пришлось использовать дополнительные служебные слова SML:

```
fun sort [] = []
|   sort (h::tail) =
    let val (nh::ntail) = rotate (h::tail) in
        (nh::sort(ntail))
    end
```

Первая часть описания, думаю, понятна: сортированная версия пустого списка представляет собой пустой список. Вторая часть (на смеси русского и ML) звучит так:

```
sort (h::tail) = (nh::sort(ntail)), где (nh::ntail) = rotate (h::tail)
```

Запись `val (nh::ntail) = rotate (h::tail)` не является присваиванием. Она лишь позволяет обращаться к результату, возвращаемому функцией rotate, как к настоящему списку, у которого есть голова и хвост.

Теперь можно увидеть результат сортировки списка:

```
- sort [5,1,2,3,4,0,7,3,4];
val it = [0,1,2,3,3,4,4,5,7] : int list
```

Как видите, только что созданная функция сортировки работает как и требовалось, но насколько она эффективна? Определение очередного элемента сортированного списка требует (возможно, многократного) «проворачивания» его хвоста. По сути дела, описанный алгоритм представляет собой разновидность сортировки обменом, когда очередной элемент списка в цикле меняется местами с наименьшим элементом хвоста. Таким образом, наша процедура не слишком-то эффективна: для сортировки списка из N элементов ей потребуется выполнить порядка N^2 операций. В то же время так называемая «быстрая сортировка» требует в среднем всего лишь порядка N * log N операций для сортировки списка того же размера.

Можно сколько угодно говорить о декларативности функционального программирования: в этом смысле любое определение функции сортировки одинаково хорошо. Однако забывать о *процедурном значении* функциональной программы (то есть о том, как она реально будет выполняться) тоже нельзя. Впрочем, как пишет о схожей проблеме декларативного языка Prolog Иван Братко, «многие программисты придерживаются взглядов, что лучше иметь в программе хоть какое-то декларативное значение, чем вообще никакого».[100]

[100] I. Bratko. Prolog Programming for Artificial Intelligence, 3rd Ed. Pearson Education Limited, 2001. Имеется русский перевод: И. Братко. Алгоритмы искусственного интеллекта на языке Prolog, третье издание. Москва, «Вильямс», 2004.

Композиции функций

Самый последний пример призван проиллюстрировать, насколько легко в SML можно комбинировать функции между собой для получения новых алгоритмов.

Функция map применяет функцию f к каждому элементу исходного списка $[e_1, e_2, ..., e_n]$, формируя результирующий список $[f(e_1), f(e_2), ..., f(e_n)]$:

```
fun map f [] = []
|   map f (h::tail) = (f h)::(map f tail)
```

Теперь можно передать на вход map любую функцию, так или иначе преобразующую отдельные элементы:

```
fun square x = x*x

- map square [1,2,3,4,5];
val it = [1,4,9,16,25] : int list
```

Можно даже отсортировать отдельные элементы списка, если они сами по себе являются списками:

```
- map sort [[1,2,0,3], [7,3,4,5], [8,7,0,0]];
val it = [[0,1,2,3],[3,4,5,7],[0,0,7,8]] : int list list
```

11.8. Сложность задач и сложность систем

После пространного отступления, посвященного формализму Черча, пора вернуться к основным темам главы.

Нет сомнений, очень важно провести этот водораздел: здесь задачи, которые мы можем решить (хотя бы в теории), а вот здесь задачи неразрешимые. Тем не менее, мир не состоит лишь из черного и белого; помимо двух крайностей — разрешимых и неразрешимых задач можно выделить задачи решаемые, но имеющие различную степень сложности.

Рассматривая различные языки, мы, конечно, уже сталкивались с различными степенями сложности задач (еще раз напомню об эквивалентности задач и языков). Так, регулярный язык в некотором смысле проще контекстно-свободного хотя бы потому, что для его распознавания требуется менее мощное устройство.

С другой стороны, в подавляющем большинстве случаев разработка реальных алгоритмов требует мощности машины Тьюринга, поэтому классификация языков, полезная при разработке компиляторов и библиотек анализа регулярных выражений, оказывается уже не столь важной. Практически все разрешимые задачи, встающие перед программистами, сразу же попадают в класс рекурсивных.

Таким образом, внутри множества разрешимых (рекурсивных) задач существует своя собственная классификация сложности, изучаемая *теорией сложности вычислений* (complexity theory).

11.8.1. Классы P и NP

Определение классов P и NP. Задачи из класса P

Значительная часть современной теории сложности так или иначе связана с изучением так называемых классов P и NP (мы другими заниматься и не будем, хотя, конечно, двумя классами представление о сложности задач не исчерпывается).

Расшифровываются эти названия довольно просто. К классу P принадлежат задачи, решаемые за полиномиальное (P) время с помощью обычной (детерминированной) машины Тьюринга. К классу NP принадлежат задачи, решаемые за полиномиальное (опять же P) время с помощью недетерминированной (N) машины Тьюринга.

Что значит «решаемые за полиномиальное время»? Пусть входные данные, записанные на ленту машины Тьюринга, умещаются в N ячеек. Тогда это условие означает, что для получения ответа машине достаточно совершить не более T(N) переходов, где T(N) — некоторый многочлен относительно N.

Возьмем, например, машину Тьюринга, распознающую язык $\{a^nb^nc^n\}$ (я знаю, что это было давно, но давайте попробуем вспомнить). Эта машина последовательно заменяет в исходной строке идущие друг за другом символы a, b и c на пробелы. Строка, в итоге оказывающаяся полностью затертой пробелами, допускается. Сколько же переходов требует подобный алгоритм? Давайте попробуем прикинуть. Худший (наиболее трудоемкий) случай возникает при анализе строки, принадлежащей языку $\{a^nb^nc^n\}$, поскольку машине при этом приходится полностью заменять ее пробелами.

Если же строка не принадлежит данному языку, этот факт будет обнаружен раньше, и работа машины завершится. Если длина входной строки (вида $\{a^nb^nc^n\}$) равна N, то длина каждой из подстрок aa...a, bb...b и cc...c равна N / 3. Отсюда следует, что для затирания всей строки пробелами машина Тьюринга должна прогуляться по ней ровно N / 3 раз[101]. Каждая такая прогулка требует выполнения N действий, поскольку исходная строка содержит N символов. Можно взглянуть на машину Тьюринга и по-

[101] Вообще-то, если вы помните, перед началом работы основного блока машины выполняется еще один рейд по строке — чтобы поставить в ее конец служебный символ ◁. Впрочем, на результаты нашего анализа это никак не влияет.

считать точно, во сколько переходов обходится любое действие, но в этом нет особой надобности, так как нас интересует лишь полиномиальность затрачиваемого количества переходов, а не его конкретное значение.

Таким образом получается, что задача решается с помощью порядка N * N / 3 переходов. Волшебная фраза «порядка стольких-то операций» означает, что те или иные константы несущественны, достаточно указать лишь общий вид получающейся функции. Поэтому выражение N * N / 3 можно еще упростить до N^2. В итоге можно сделать вывод, что задача распознавания языка $\{a^nb^nc^n\}$ принадлежит классу P, поскольку для ее решения машине Тьюринга потребуется $O(N^2)$ операций, а N^2 есть многочлен от N.

Чтобы соблюсти формальности с указанием количества переходов, напомню, что запись $O(N^2)$ можно прочесть также как «не больше $K*N^2$ переходов, где K — константа». Так что общее количество переходов действительно выражается многочленом от N.

Что применимо к машинам Тьюринга, применимо и к обычным компьютерным программам. Если некоторая задача решается за полиномиальное время машиной Тьюринга, можно написать программу, решающую задачу также за полиномиальное время. Компьютерная программа может быть намного эффективнее. Например, чтобы считать некоторый элемент с ленты, машине Тьюринга приходится двигать в цикле считывающую головку; в программе обычно можно за одно действие обратиться к любой интересующей нас ячейке памяти. Однако эти улучшения никак не влияют на принадлежность задачи к классу P.

Если машина Тьюринга, к примеру, решает задачу за время $O(N^3)$, а компьютерная программа делает это, затрачивая $O(N^2)$ или даже O(N) действий, все равно итоговое время останется полиномиальным. Даже вырожденный случай «одношаговой» задачи, решаемой за время O(1), принадлежит классу P, ибо константа тоже формально является многочленом от N (в котором степень при N равна нулю).

Таким образом, чтобы доказать принадлежность некоторой задачи классу P, достаточно изобрести любой (пусть даже и не самый эффективный) полиномиальный алгоритм ее решения.

В целом считается, что задачи из класса P требуют приемлемого количества операций для решения. Существует много популярных, с успехом применяющихся алгоритмов, имеющих сложность $O(N^2)$ или $O(N^3)$. Более того, подавляющее большинство (если не абсолютно все) процедуры, которые обычный программист использует в повседневной практике, также имеют P-сложность. Конечно, если алгоритму требуется выполнить порядка N^{50} или N^{100} операций, его трудно назвать приемлемым, но формально он все равно принадлежит классу P.

Перейдем теперь к классу NP (задачи, решаемые за полиномиальное время недетерминированной машиной Тьюринга). Для начала стоит заметить, что все задачи из класса P, очевидно, могут быть с не меньшим успехом решены и недетерминированной машиной, поэтому все они также принадлежат классу NP. Теперь поставим обратный вопрос: существуют ли задачи, решаемые недетерминированной машиной Тьюринга за полиномиальное время, но не решаемые за полиномиальное время обычной, детерминированной машиной? Иначе говоря, совпадают ли классы P и NP?

Плохая новость заключается в том, что никто в мире пока что этого не знает. Однако (и это хорошая новость) вы можете решить эту задачу и получить приз в миллион долларов, обещанный математическим институтом Клэя (Clay Mathematics Institute). Если говорить серьезно, то большинство специалистов полагают, что все-таки существуют задачи, принадлежащие классу NP и в то же время не принадлежащие классу P. Существуют довольно сильные свидетельства в пользу такого мнения (достаточно взглянуть на так называемые NP-полные задачи), однако, повторюсь, строгих доказательств на сегодняшний день еще не найдено.

Таким образом, чаще всего класс P считают подмножеством класса NP (см. рис. 11.2).

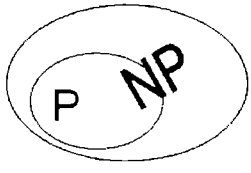


Рис. 11.2. *Классы P и NP*

Какие же задачи попадают в класс NP, но, по всей видимости, не попадают в класс P? Их очень много. Приведу лишь несколько классических примеров.

Задача выполнимости (satisfiability problem). Дается булево выражение, состоящее из переменных, знаков операций (И, ИЛИ, НЕ) и скобок. Требуется определить, выполнимо ли выражение, то есть существует ли такой набор значений переменных, при котором значение всего выражения истинно. Например, выражение (x ИЛИ y) И z выполнимо: его значение истинно, например, если x = true, y = true, z = true. Напротив, выражение x И (НЕ x) не выполнимо.

Задача коммивояжера (traveling salesman problem). Коммивояжер должен объехать N городов, не посетив ни одного из них дважды, а затем вернуться назад. Требуется определить самый дешевый маршрут, если известна стоимость проезда между любыми двумя городами.

Задача о рюкзаке (knapsack problem). В комнате лежат N предметов. Для каждого предмета известна его масса и стоимость. Требуется заполнить рюкзак предметами, максимизируя их суммарную стоимость, при условии, что общая масса взятых предметов не может превышать заданного K.

Решить любую из этих задач за полиномиальное время с помощью недетерминированной машины Тьюринга достаточно просто. К примеру, если предположить, что стоимость любого предмета является целым числом, ограниченным некоторой константой (это ограничение никак не влияет на сложность), то алгоритм решения задачи о рюкзаке на псевдокоде может выглядеть так:

```
итоговая_стоимость = сумма стоимостей всех предметов
в комнате;
ЦИКЛ
    НЕДЕТЕРМИНИРОВАННЫЙ БЛОК
        масса = 0, стоимость = 0;
        ЦИКЛ
            выйти_или_не_выйти из цикла;
            взять предмет из комнаты и поместить его в рюкзак;
            масса += масса предмета;
            стоимость += стоимость предмета;
            если комната пуста, выйти из цикла;
        КОНЕЦ ЦИКЛА

        ЕСЛИ стоимость == итоговая_стоимость и масса <= K
            напечатать множество предметов рюкзака
            в качестве ответа;

            завершить работу всей программы;
    КОНЕЦ БЛОКА
    уменьшить значение переменной итоговая_стоимость
    на единицу;

ПОКА итоговая_стоимость >= 0
сообщить, что решения не существует
```

Вероятно, недетерминированная компьютерная программа выглядит слишком своеобразно, чтобы оставлять ее без комментариев.

Общая идея решения такова. Стоимость предметов, помещенных в рюкзак, не может превышать стоимости всех предметов в комнате. Поэтому стоимость предметов, которые в итоге сформируют решение задачи, можно оценить сверху этим числом. Далее выполняется цикл:

```
ЦИКЛ
    найти решение задачи со стоимостью итоговая_стоимость;
    если решение найдено, напечатать его и выйти из программы;

    итоговая_стоимость = итоговая_стоимость  -  1;
КОНЕЦ ЦИКЛА
сообщить о том, что решения не существует;
```

Поскольку стоимость каждого предмета (а следовательно, и суммарная стоимость предметов в рюкзаке) является целым числом, можно попытаться по очереди перебрать все возможные значения стоимости итогового решения — начиная с наиболее оптимистичной оценки и заканчивая нулем.

Теперь необходимо написать блок, определяющий, можно ли найти набор предметов, суммарная стоимость которых была бы равна заданному значению итоговая_стоимость, а масса не превосходила бы K.

Здесь мы впервые встречаемся с недетерминированным поведением в строке

```
выйти_или_не_выйти из цикла;
```

В соответствии с моделью Эверетта это следует читать так: в одном мире происходит выход из цикла, в то время как в другом выполнение тела цикла продолжается.

Далее, что означает действие «взять предмет из комнаты»? Какой предмет? Любой. Эта инструкция также является недетерминированной: вместо одного мира мы получим столько миров, сколько свободных предметов осталось в комнате. В каждом из этих миров будет сделан свой собственный выбор.

Понятно, что если решение с заданной стоимостью существует, то хотя бы в одном из возможных миров будет сделан правильный выбор множества предметов, а затем произойдет выход из цикла. Затем программа распечатает ответ и немедленно завершит выполнение.

Теперь давайте оценим время работы алгоритма. На каждом шаге внутреннего цикла либо очередной предмет помещается в рюкзак, либо выполнение цикла прекращается. Таким образом, даже в худшем случае после N итераций процедура будет полностью завершена.

С внешним циклом дело обстоит несколько хитрее. Поскольку мы договорились, что стоимость каждого предмета ограничена некоторой константой (назовем ее C), суммарная стоимость всех предметов заведомо не превосходит C*N. Получается, что внешний цикл программы выполняется не более C*N раз.

Итоговая сложность алгоритма, таким образом, оказывается полиномиальной — равной $O(C*N^2)$ или просто $O(N^2)$.

Решение NP-задач на практике

Давайте теперь вернемся от недетерминированных машин к реальному миру. Здесь складывается довольно удручающая ситуация. Единственное известное решение задачи о рюкзаке[102] заключается в исследовании всех возможных наборов предметов из комнаты. Если в комнате лежит N предметов, то число таких наборов (по сути подмножеств множества {1, 2, ..., N}) равно $2^N - 1$. Таким образом, программе придется выполнить $O(2^N)$ операций, что означает экспоненциальную сложность.

В главе об автоматах я уже приводил график этой до обидного быстро растущей функции; тем не менее, напомню еще раз о том, что такое экспоненциальный рост. Предположим для простоты, что программа, решающая задачу о рюкзаке, выполняет ровно 2^N атомарных операций.

Допустим, пользователя удовлетворяет время, затрачиваемое на решение задачи при N = 18, но уже при N = 19 программу использовать не представляется возможным. Так вот, чтобы обработать за разумное время девятнадцать предметов, вам придется ускорить выполнение программы вдвое; для двадцати предметов программа должна работать уже в четыре, а для двадцати одного предмета — в восемь раз быстрее!

Справедливости ради следует отметить, что существуют приемы оптимизации, позволяющие довольно быстро отсечь солидную часть бесперспективных направлений при поиске решения (так называемый метод ветвей и границ). Эти приемы помогают несколько отодвинуть максимальное значение N, при котором попытка найти решение еще имеет смысл. Иногда этого оказывается достаточно (допустим, коммивояжер из задачи коммивояжера из принципа не объезжает больше двадцати городов за раз). Тем не менее, итоговая сложность все равно остается экспоненциальной.

Единственный известный способ решения приведенных задач для больших N состоит в использовании приближенных алгоритмов. Так, решение задачи коммивояжера с ошибкой, не превышающей 50% (то есть полученное решение будет отличаться от оптимального не более чем на 50%) можно найти с помощью полиномиального ($O(N^3)$) алгоритма Кристофидеса.

Кстати, недетерминированный алгоритм решения задачи о рюкзаке иллюстрирует один важный факт: наиболее трудоемкая его часть — внутренний цикл, собственно, требующий экспоненциального времени на детерминированной машине, в действительности решает задачу при-

[102] Аналогичными по сути (то есть переборными) алгоритмами решаются и другие описанные в разделе задачи — задача выполнимости и задача коммивояжера.

нятия решения (да/нет-задачу — помните такие?) Это в очередной раз доказывает, что задачи принятия решений могут встретиться в любом классе сложности.

11.8.2. NP-трудные и NP-полные задачи

Итак, мы уже выяснили, что все P-задачи принадлежат классу NP, но далеко не все NP-задачи могут быть решены за полиномиальное время на детерминированной машине Тьюринга. Чтобы классификация приобрела более законченный вид, придется добавить в нее еще NP-трудные и NP-полные задачи.

Прежде всего поговорим немного о *сведении задач*. Есть старый-престарый анекдот:

Задача 1. Дан пустой чайник. Требуется вскипятить воду.

Физик: Набрать в чайник воду, поставить на плиту.

Математик: Набрать в чайник воду, поставить на плиту.

Задача 2. Дан чайник с холодной водой. Требуется вскипятить воду.

Физик: Поставить чайник на плиту.

Математик: Вылить воду из чайника. Задача 2 сведена к уже решенной задаче 1.

Имея решенную задачу или запрограммированный алгоритм, нам совершенно справедливо хочется пользоваться достигнутым результатом как можно чаще. Предположим, некоторый конечный автомат задан при помощи *таблицы* переходов (строка задает исходное состояние, столбец — символ входного алфавита; на пересечении строки и столбца можно прочесть целевое состояние). Задача состоит в разработке алгоритма его минимизации. Далее, если у меня уже есть процедура минимизации конечного автомата, принимающая описание автомата в виде *списка* переходов (набора троек вида (состояние, символ, новое_состояние)), я предпочту написать подпрограмму преобразования таблицы в список и воспользоваться готовым алгоритмом минимизации, а не начинать все с нуля.

Разумеется, не всякое сведение оправдано. К примеру, поиск минимального элемента массива может быть сведен к сортировке: отсортируем массив, затем выберем его первый элемент — он, очевидно, и будет минимальным. Сортировка массива — задача более трудоемкая, и ее использование здесь похоже на стрельбу из пушки по воробьям.

Не следует забывать, что сам процесс сведения тоже требует времени. Чтобы воспользоваться готовым алгоритмом минимизации, приходится

тратить дополнительные усилия на преобразование входных параметров из табличной записи в список.

В теории сложности существует специальный термин, который нам в дальнейшем пригодится: *полиномиальное сведение*. Задача A сводится к задаче B за полиномиальное время (иначе говоря, существует полиномиальное сведение A к B), если с помощью некоторого алгоритма из класса P можно преобразовать входные параметры задачи A во входные параметры для задачи B. Затем результат, полученный при решении задачи B, опять-таки полиномиально сводится к решению задачи A.

Задача, к которой можно полиномиально свести любую NP-задачу, называется *NP-трудной* (NP-hard). Таким образом, имея алгоритм решения NP-трудной задачи, можно решить любую NP-задачу. NP-трудная задача может принадлежать классу NP, а может и не принадлежать. Например, к NP-трудным относится, как мы уже знаем, неразрешимая задача останова. К NP-трудным относятся также задача выполнимости, задача коммивояжера и задача о рюкзаке (таким образом, любую из них можно полиномиально свести к любой другой).

Задача, принадлежащая одновременно к классам NP и NP-трудных задач, называется *NP-полной* (NP-complete). Задача о рюкзаке, задача коммивояжера и задача выполнимости — NP-полные.

Не углубляясь в теоретические вопросы, я хочу лишь отметить один интересный факт, следующий из определения NP-полноты. Если хотя бы для одной NP-полной задачи удастся разработать полиномиальный алгоритм решения, это автоматически будет означать совпадение классов P и NP.

11.8.3. Феномен сложности

У истоков сложности

Такое удивительное соседство задач — простых, сложных и вообще не поддающихся решению (и это при совсем нехитрых формулировках!) — навевает мысли об истоках самого феномена сложности — сложности не только языков и задач, но и систем, явлений окружающего нас мира.

Конечно, обсуждать устройство Вселенной и, тем более, делать какие-либо выводы — задача сложная и неблагодарная. Здесь я лишь хочу озвучить некоторые показавшиеся мне интересными идеи, высказанные[103] известным физиком Стивеном Вольфрамом, автором пакета Mathematica.

[103] В книге «A New Kind of Science» (издательство Wolfram Media, 2002 г.; русского перевода, к сожалению, пока нет).

Основной постулат звучит так: даже если нечто устроено очень просто, оно может вести себя запредельно сложно. Это наблюдение можно проверить и на непрерывных системах, обычно встречающихся в природе, но, пожалуй, самым простым и наглядным примером может служить так называемый клеточный автомат.

Одномерный клеточный автомат

В простейшем, одномерном случае автомат можно описать следующим образом. Представьте себе клетчатую ленту, бесконечную в обе стороны. Каждая клетка ленты может быть либо пустой, либо закрашенной. Смысл работы автомата заключается в пошаговом изменении содержимого ленты. В свою очередь, текущее содержимое ленты однозначно определяется ее содержимым на предыдущем шаге (если вы знакомы с игрой «Жизнь» Джона Конуэя, то это как раз отличный пример двумерного клеточного автомата). Правила, по которым происходит изменение ленты, определяются разработчиком автомата. Мы ограничимся простым случаем, при котором цвет любой клетки ленты на очередном шаге определяется ее цветом и цветом обеих соседних с ней клеток (слева и справа) на предыдущем шаге.

Поскольку цвет клетки определяется цветами трех клеток предыдущего шага, общее количество правил в автомате будет равно восьми. Рассмотрим, к примеру, набор правил, показанный на рис. 11.3.

Каждый ряд из трех клеток указывает ситуацию, в которой применяется правило, а клетка снизу — результат его применения (то есть цвет центральной клетки на следующем шаге). Например, в ситуациях «три подряд идущие белые клетки» или «две белые клетки и одна черная» центральная клетка на следующем шаге окажется незакрашенной. Закрашенная же клетка, окруженная двумя незакрашенными, сохранит свой статус и после выполнения итерации.

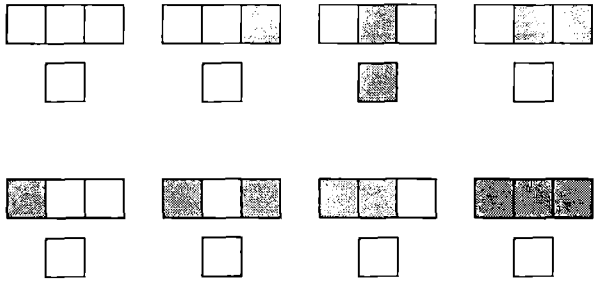


Рис. 11.3. *Правила клеточного автомата: набор №1*

Предположим теперь, что на первом шаге моделирования лента содержит лишь одну закрашенную клетку. Как будет вести себя автомат? Что ж, в случае приведенного набора правил поведение автомата предсказать нетрудно: лента вообще не будет изменяться. Если изобразить изменение ленты на графике, получится прямая линия (см. рис. 11.4).

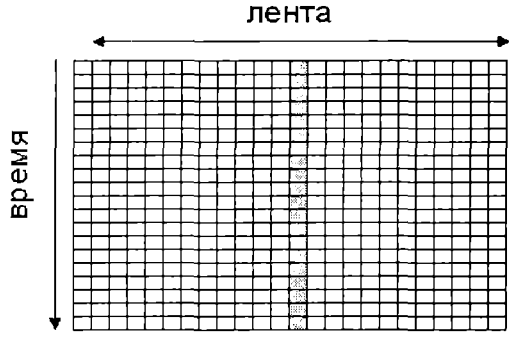


Рис. 11.4. *График снимков ленты на разных шагах моделирования (набор правил №1)*

Теперь давайте попробуем немного изменить набор применяемых правил (см. рис. 11.5).

График снимков ленты этого автомата вас приятно удивит (см. рис. 11.6; я уменьшил масштаб, чтобы захватить как можно больше шагов).

Как видите, маленькое изменение в правилах приводит к значительным изменениям. Получившийся автомат генерирует неожиданно красивый, регулярный узор, не имеющий ничего общего с простотой обычной прямой линии.

Поэкспериментировав с правилами, можно найти еще несколько автоматов, создающих красивые узоры наподобие приведенного.

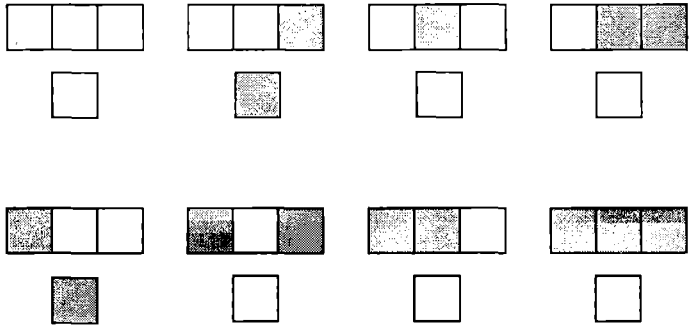


Рис. 11.5. *Правила клеточного автомата: набор №2*

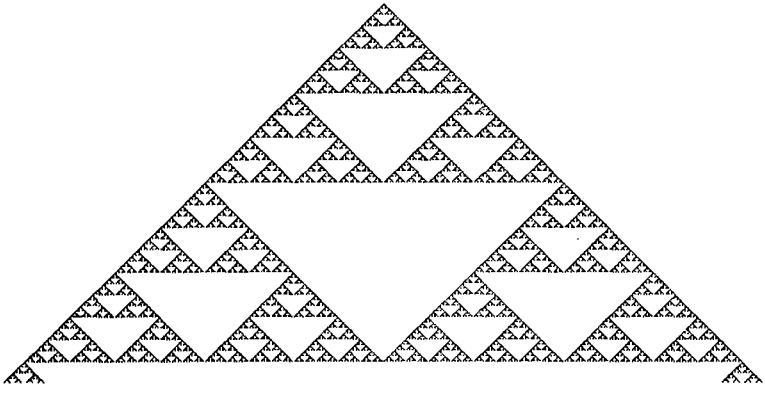


Рис. 11.6. *График снимков ленты (набор правил №2)*

Однако возможности клеточных автоматов не ограничиваются рисованием регулярных узоров. В конце концов, они красивы, но достаточно просты и предсказуемы.

Совсем иной рисунок получится в результате применения так называемого «правила №30» (см. рис. 11.7).

«Правило №30» принадлежит к числу самых интересных находок Вольфрама. Хотя на первый взгляд рис. 11.7 смотрится куда менее эстетично по сравнению с рис. 11.6, в нем есть своя изюминка.

Левая часть рисунка представляет собой довольно простой узор, чем-то напоминающий вязание крючком. В правой части наблюдается хаотическое нагромождение белых треугольников разных размеров; согласитесь,

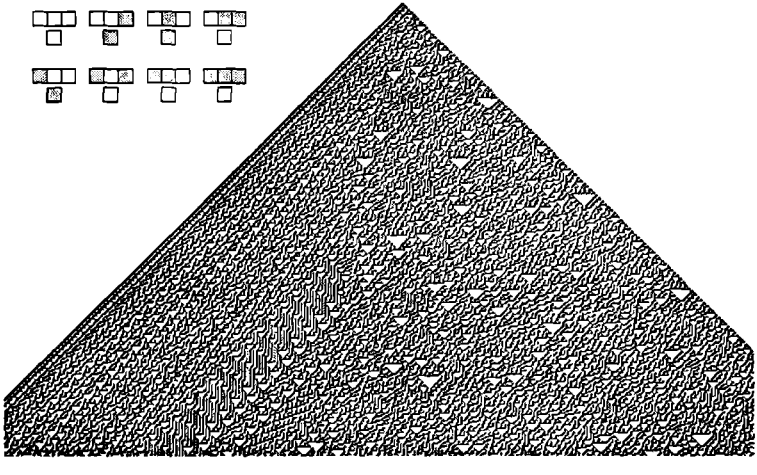


Рис. 11.7. *Результат применения правила №30*

возникновение хаоса из восьми простейших правил — уже явление интересное. Но самое интересное в этой фигуре — поведение центральной клетки ленты.

Если последовательно записывать значение цвета центральной клетки на каждом шаге, мы получим какую-то последовательность из нулей и единиц (нуль означает, что клетка не закрашена, единица — закрашена). Можно ли найти в этой последовательности хоть какую-нибудь закономерность? Так вот, удивительно, но согласно любым статистическим тестам построенная последовательность классифицируется как случайная.

Таким образом, поведение автомата оказывается столь сложным, что мы уже не в состоянии определить, по какому принципу построен приведенный рисунок. Тесты на случайность даже убедили Вольфрама воспользоваться правилом №30 в качестве основы генератора случайных чисел для системы Mathematica (вот вам и практическое применение клеточных автоматов).

Из этих наблюдений можно сделать два интересных вывода:

- Очень сложное, почти непостижимое поведение системы может быть результатом работы очень простых правил. Любуясь причудливо изогнутой раковиной моллюска или восхищаясь красотой снежинки, мы полагаем, что их созданием управляет какой-то очень сложный механизм. На самом же деле вполне может оказаться, что все гораздо проще.
- Наблюдая внешнее поведение системы, практически невозможно постичь принципы, ею управляющие. Попробуйте, например, по рис. 11.7 вывести вид правила №30. Возможно, единственный способ состоит в догадке (формулировании гипотезы) об общей идее правил, управляющих системой, с последующим перебором вариантов — то есть применением метода проб и ошибок. К сожалению, это утверждение относится не только к клеточным автоматам, но и ко многим системам реального мира. Это похоже на любимый аниматорами прием: главный герой мультфильма пугается огромной нависшей над ним тени, а потом оказывается, что тень принадлежит какой-нибудь мелкой птичке или мышке. Увы, слишком часто судить о природном явлении мы можем лишь по его «тени».

Принцип вычислительной несводимости

Еще одно замечание связано с *принципом вычислительной несводимости*. Предположим, вам нужно сложить два больших числа. Скажем, 12345 и 67890. Конечно, можно в цикле от 1 до 67890 увеличивать 12345 на единицу, но гораздо лучше воспользоваться алгоритмом сложения в столбик. Этот пример иллюстрирует, что некоторый трудоемкий процесс

(многократное прибавление единицы) порою может быть сведен к другой, гораздо более простой в смысле затрат времени процедуре.

Но всегда ли это так? По-видимому, нет. Например (хотя это не доказано), единственный способ получить состояние ленты автомата после стократного применения правила №30 состоит в моделировании ста шагов работы автомата.

Возможность использовать ту или иную математическую формулу для решения задачи в известном смысле как раз и означает сведение сложного процесса к простому. Так, чтобы узнать, когда же мы, наконец, попадем на дачу, расположенную в 30 километрах от города, двигаясь со средней скоростью 60 км/ч, нет необходимости моделировать поведение автомобиля (сложный процесс). Достаточно вспомнить, что время равно расстоянию, деленному на скорость (простой процесс). То же самое касается предсказания солнечных затмений, посадки космического аппарата на Марс или на Луну и тому подобных вещей. Получается, что рассчитать траекторию планеты или космического корабля проще, чем понять поведение простого клеточного автомата!

Вероятно, в окружающем нас мире можно найти очень много примеров несводимых вычислений. Может оказаться, что моделирование процесса «в лоб» окажется либо лучше применения любых дифференциальных уравнений (Вольфрам доказывает это на примерах), либо вообще единственным способом получения ответа.

К сожалению, математики и физики (по крайней мере, очень многие известные мне личности) склонны заниматься задачами, посильными современному математическому аппарату, не особо заботясь о практическом смысле своих результатов. Иными словами, мы просто решаем то, что мы умеем решать сегодня. Задачи же наподобие анализа поведения клеточного автомата сразу же объявляются «неинтересными». Вообще, начинающий исследователь напоминает мне муравья (простите!), стоящего у входа в изрытый тоннелями муравейник. Тоннелей очень много, а каждый из них глубок и извилист. И, вместо того, чтобы посмотреть вокруг и увидеть огромное количество интересного на поверхности, наш муравей (то есть исследователь) предпочитает выбрать один из проторенных маршрутов, затем долго и упорно двигаться по нему, пока сам процесс движения и рытья не станет целью работы.

Подавляющее большинство научных книг описывают то, что мы умеем делать. Мне кажется, что стоило бы побольше писать о явлениях, нас окружающих и каждому известных, но почти не изученных.

Итоги

- Как и любое другое устройство, описанное в книге, машина Тьюринга имеет свои ограничения. Существуют задачи, не решаемые с помощью машины Тьюринга. В частности, к ним относится проблема останова.
- В компьютерной науке обычно полагают справедливым тезис Черча-Тьюринга, утверждающий невозможность построить вычислительную машину, превосходящую по мощности машину Тьюринга. Отсюда следует определение алгоритма как машины Тьюринга: если существует алгоритм решения задачи (в неформальном смысле), то существует и машина, его воплощающая, и наоборот.
- По-видимому, человек тоже неспособен решить неразрешимую (машиной Тьюринга) задачу, в частности, проблему останова. Из этих наблюдений можно сформулировать сильную версию тезиса Черча-Тьюринга о равенстве вычислительной мощи машины Тьюринга и человеческого мозга.
- Машина Тьюринга — не единственный инструмент для описания алгоритмов. Существуют и другие императивные (машина Поста) и декларативные (лямбда-исчисление) модели.
- Даже если задача разрешима, она может иметь принадлежать к тому или иному классу сложности. Наиболее важными классами в наше время считаются классы P (задачи, решаемые детерминированной машиной Тьюринга за полиномиальное время) и NP (задачи, решаемые недетерминированной машиной Тьюринга за полиномиальное время).
- Считается, что класс P целиком лежит внутри класса NP, хотя это не доказано. По-видимому, пространство, принадлежащее классу NP, но не принадлежащее классу P, занято так называемыми NP-полными задачами. В настоящее время известны лишь экспоненциальные алгоритмы их решения на детерминированной машине Тьюринга (по сути дела выполняющие обычный перебор вариантов).
- Как показывают наблюдения, сложность вообще присуща системам нашего мира. Даже основанный на очень простых принципах механизм может вести себя запредельно сложно.

Послесловие

Любой человек, занимающийся программированием изо дня в день, примерно представляет себе, какого рода задачи ставит перед ним профессия и в общих чертах знает, в какой литературе можно найти ответы на интересующие вопросы. Но помимо жизненно важных знаний и навыков существует ещё такое понятие, как общая культура программирования.

Обычная жизнь, повседневное общение, вероятно, не требуют от человека каких-то особенных познаний (по крайней мере, субъекты, ими не отягощённые, но при этом прекрасно выживающие, встречаются с завидной регулярностью). Но общая культурная образованность способна сделать нашу жизнь если не проще, то заметно интереснее и богаче. Если во время прогулки где-нибудь в исторических кварталах города у вас перед глазами возникают живые картины из прошлого, если ежедневные ситуации ассоциируются с событиями из любимых литературных произведений, тем самым обретая новые яркие краски, если вы улыбаетесь, слушая политических деятелей, потому что твёрдо знаете, кто и когда уже говорил подобное много лет назад — уверен, такое, более глубокое понимание происходящего вокруг когда-нибудь сослужит вам хорошую службу.

Точно так же и общая культура программирования, не призванная содействовать достижению каких-либо конкретных целей, поднимает понимание программирования как вида творчества на качественно новый уровень. Взять, к примеру, такую популярную нынче идею как паттерны программирования — высокоуровневые конструкции, выражающие общепринятый подход к решению той или иной типичной задачи. Начинающий программист без труда поймёт каждую отдельную инструкцию, входящую в алгоритм-паттерн, но общая архитектура останется для него неясной. Представьте, что, глядя на загородный домик, мы бы не видели окон, стен и крыши, а понимали бы лишь, что он состоит из кирпичей, стекла и черепицы. Немыслимо в реальной жизни, но вполне обыкновенно в программировании.

Эта книга, содержащая множество доведённых до реализации алгоритмов, тем не менее посвящена «инфраструктуре» компьютерной науки, скрытой за яркими фасадами и привлекающими внимание декоративными

элементами. Хороший пример пользы общей культуры здесь уже приводился: функциональное программирование. Думаю, что подавляющее большинство читателей никогда не использовали языки наподобие Haskell или Standard ML и вряд ли когда-либо будут это делать. Тем не менее, идеологию обобщённого программирования вообще и библиотеки STL (с которой, наверняка, сталкивался каждый второй) понять гораздо труднее, не зная основ функциональной парадигмы.

Таким образом, я искренне старался не только привести несколько полезных рецептов на каждый день, но и помочь расширить понимание некоторых принципов, лежащих в основе современной компьютерной науки. Удалось ли мне сделать это хоть с какой-то степенью успеха или нет — судить вам. Буду рад откликам, предложениям и конструктивной критике. Адрес моей электронной почты — maxim_mozgovoy@hotbox.ru.

Максим Мозговой,
май 2005 г.